iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer

Carsten Clauss, Stefan Lankes, Pablo Reble,
Jacek Galowicz, Simon Pickartz and Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
contact@lfbs.rwth-aachen.de

Version 2.0 (iRCCE FLAIR) / March 2013

1 Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [4] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. The 48 cores are arranged in a 6x4 on-die mesh of tiles with two cores per tile. The SCC chip possesses four on-die memory controllers for addressing the external main memory. Additionally, each tile possesses a small amount of fast on-die memory that is also accessible to all other cores in a shared-memory manner. These special memory regions are the so-called *Message-Passing Buffers* (MPBs) of the SCC. The SCC's architecture does not provide any cache coherency between the cores, but rather offers a low-latency infrastructure in terms of these MPBs for explicit message-passing between the cores. Thus, the processor resembles a *Cluster-on-Chip* architecture with distributed but shared memory where each core can run its own operating system instance. Communication between processes hosted by different OS instances running on different cores can then be conducted either via a dedicated region of the off-die shared memory or via the fast on-die MPBs for increased performance. The RCCE communication library [5, 6] offers a customized message-passing interface for programming these SCC features by means of a simplified application programming interface (API). This message-passing environment, that is in turn based on a simpler one-sided communication mechanism (`RCCE_put`/`RCCE_get`), offers two-sided but *blocking* (often also referred to as *synchronous*) point-to-point communication functions (`RCCE_send`/`RCCE_recv`) as well as a set of collective communication operations (Barrier, Broadcast, etc.). However, the lack of *non*-blocking point-to-point communication capabilities within the current RCCE library has driven us to extend RCCE by such asynchronous message-passing functions (`iRCCE_isend`/`iRCCE_irecv`, see Section 3). Furthermore, we have also improved the performance of some RCCE functions, as for example the blocking send and receive operations (see Section 5) and we have added *wildcard* features (`iRCCE_ANY_SOURCE`, `iRCCE_ANY_LENGTH`, see Section 6). In our latest release (V2.0, iRCCE FLAIR), we have also added support for so-called *Tagged Flags* (see Section 4.4) and for *Atomic Increment Registers* (see Section 4.5). In order not to interfere with the current RCCE library and its future updates, we have placed our extensions into an additional auxiliary library with a separated namespace called *iRCCE* [2]. In this manual, we detail the installation and the usage of these iRCCE extensions (see Section 2 and Section 9) and we present some performance comparisons between the current RCCE release and the improved iRCCE functions (see Section 8).

2 Getting Started

2.1 Installation Guide

1. Download, configure and build the common RCCE library with the *non-gory* interface.
2. Download[1] iRCCE as a TAR-File and unpack it within the desired installation folder.
   (e.g. within the RCCE root directory by calling `tar -xvzf iRCCE.tar.gz`)
3. Change into the iRCCE directory.
4. Type `./configure <path-to-rcce>` (= path to the RCCE installation).
   (e.g. `./configure ..` when you have unpacked iRCCE within the RCCE root directory)
5. Call `make [AIR=1]`[2]. All iRCCE related functions will then be added to the common RCCE library.[3]

2.2 Overview of the Basic iRCCE Functions

Library Initialization Function:

- `iRCCE_init();`

Non-Blocking Send and Receive Functions:[4]

- `int iRCCE_isend(char *, size_t, int, iRCCE_SEND_REQUEST *);`
- `int iRCCE_isend_test(iRCCE_SEND_REQUEST *, int *);`
- `int iRCCE_isend_wait(iRCCE_SEND_REQUEST *);`
- `int iRCCE_isend_push(void);`
- `int iRCCE_irecv(char *, size_t, int, iRCCE_RECV_REQUEST *);`
- `int iRCCE_irecv_test(iRCCE_RECV_REQUEST *, int *);`
- `int iRCCE_irecv_wait(iRCCE_RECV_REQUEST *);`
- `int iRCCE_irecv_push(void);`

Blocking but Improved Send and Receive Functions:

- `int iRCCE_ssend(char *, size_t, int);`
- `int iRCCE_srecv(char *, size_t, int);`

Optimized Put and Get Functions:

- `int iRCCE_put(t_vcharp, t_vcharp, int, int);`
- `int iRCCE_get(t_vcharp, t_vcharp, int, int);`

[1] The iRCCE package can be found on the website of the Intel Many-core Application Research Community (MARC) [1]
[2] With `AIR=1` the additional Atomic Increment Register functions will be built, too. (Requires sccKit 1.4 or higher!)
[3] That in turn means that you do not have to link against iRCCE but just against the extended RCCE library.
[4] For convenience, the the following function names are mirrored into the common RCCE namespace, too.

2.3 Description of the Application Example and the Benchmark Tools

In order to build the application example and the benchmark tools, just change into the `apps` folder after the installation of iRCCE and call `make all`. The following iRCCE executables will then be built:

- **pingpong** A simple Ping-Pong benchmark that utilizes the improved blocking send and receive functions of iRCCE. The benchmark reports *round-trip time / 2* as well as *bandwidth* for ascending message sizes by measuring and averaging the time for n repetitions of the Ping-Pong pattern that is shown in Figure 1. Optional arguments that can be passed to the executable are the number n of repetitions and the maximum of the messages sizes that should be tested.

- **pingping** This benchmark performs a common variation of the Ping-Pong pattern that can only be realized by means of non-blocking communication functions. In contrast to the Ping-Pong benchmark, the time is measured under the aggravating circumstance that the outgoing message is interleaved with an incoming one [3]. The benchmark reports the pure average *ping time* for n repetitions for ascending message sizes. The difference between both patterns and between their clocked benchmark times can be made clear by comparing Figure 1 with Figure 2.

- **tagged** This benchmark is a minimalistic version of the Ping-Pong benchmark that uses *tagged flags* (see Section 4.4) instead of the common send/recv functions for exchanging small messages.

- **spam** This application enqueues a line of user-chosen length of requests and processes it afterwards. Hence, this is just a stress test for simulating a scenario with a huge amount of outstanding non-blocking communication requests.

- **madmonkey** Often cited in discussions about evolution, the *infinite monkey theorem* states that a monkey typing random keys on a typewriter will certainly write any given text after a (long) period time (Shakespeare's works are often mentioned here). While parallel computer architectures are very suitable for simulating this scenario, this application demonstrates the use of non-blocking send/receive calls in situations where no one can tell when sends/receives shall occur. This application needs a word to be found as program argument. Try for example `"iRCCE"`; the search for this word should not take much longer than one minute when running the program on two cores.

- **wildcards** This is just a copy of the above described Ping-Pong and Ping-Ping benchmarks with the simple variation that it uses the new wildcards for message length and for the source rank of the sender (see Section 6). For using the `iRCCE_ANY_LENGTH` wildcard within this application, iRCCE must be built against RCCE V1.0.13 with `SINGLEBITFLAGS` disabled!

- **fortune** This application implements a *wheel of fortune* game: a master devises a number between 0 and 100. The players then may guess which number was made up and send their guess to the master. To receive the guesses from the players the master makes use of the wildcard mechanism for the source rank (see Section 6). After having received all the messages from the players the master determines the winner which is the one with the best guess.

- **multicast** This benchmark measures the throughput performance of iRCCE's multicast functions. In doing so, one process (the root) sends messages to all the other started processes by using the `msend` and `mrecv` functions (see Section 5.3). Additionally, this benchmark also demonstrates the feasibility of iRCCE's wildcard features in collective communication patterns.

Each of these executables can just be started like a *normal* RCCE application. For example like:

```
> rccerun -nue 2 -f rc.hosts pingping 1000 65536
```

...or via the much more convenient `irccerun script` (see next Section 2.4):

```
> irccerun -nue 2 -cores 00,01 pingping 1000 65536
```

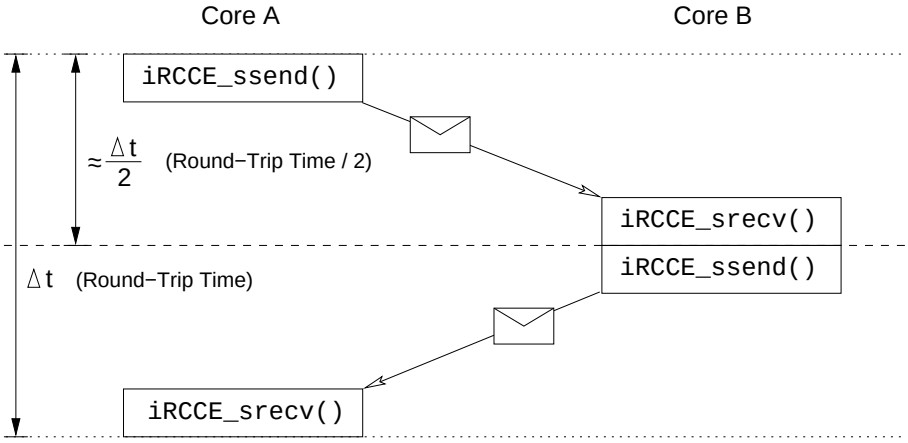

Figure 1: Communication Pattern of the iRCCE *Ping-Pong* Benchmark

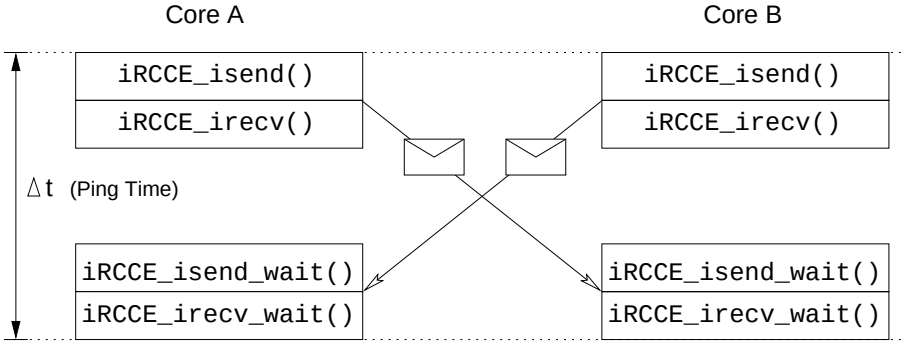

Figure 2: Communication Pattern of the iRCCE *Ping-Ping* Benchmark

2.4 The ircceruun Script

The ircceruun script is a startup tool for RCCE/iRCCE applications, quite similar to the well-known rccerun script. However, it offers some additional features that make the session handling much more convenient. So, for example, the script makes sure that the executable is copied into the shared folder before session startup. Hence, an application can directly be started from any location of the file systems mounted on the MCPC. Try `ircceruun --help` so find out some more features and option:

```
Standard Options for ircceruun:
    -nue <num cores> Number of cores to be involved
    -cores <names>   List of cores to be involved
    -corefile <file> Name of core file to be used
    -verbose|-v      Verbose startup output
    -Verbose|-V      More verbose startup output
    -tetsing|-t      Don't start but show ssh command lines

Special Options for ircceruun:
    -kill            Don't start but kill all still running executable
    -ping            Don't start but echo a 'ping' response
    -clear           Clear the MPBs in an explicit procedure (like rccerun)
    -delay           Time span to wait after clearing the MPBs
    -clock   <freq>  Set the reference clock frequency to <freq>
```

3 Non-blocking Communication

3.1 Interleaved Communication and Computation

In many situations, performance can be improved by interleaving communication and computation. This is particularly true in situations where the communication progress stalls due to a temporary lack of communication resources like intermediate buffers. In such a situation, a blocking communication function has to wait (either actively by polling or passively by sleeping) until the needed resource becomes available again. An alternative mechanism in such a situation is to use non-blocking communication functions that do not block but return back to the application immediately if the communication progress can temporarily not be fostered. Of course, it is up to the application to exploit the interim time until the communication progress can be pushed on again. This can be done either by processing computational tasks or by pushing on with other communication requests. But this in turn means that the application must check repeatedly by itself whether the communication is still stuck or not. For this purpose, non-blocking communication functions usually pass back a so-call *request handle* which can then be used by additional *push*, *test* or *wait* functions to ensure the communication's progress and eventually its completion.

3.2 An Exemplary Scenario

For example, a non-blocking *receive function* will just check if the respective incoming message is already completely available. If this is not the case (e.g. the sender has not even started the message transfer), the function just records all needed parameters (like the source and the length of the expected message as well as the address of the respective receive buffer) within the request handle and returns it immediately to the application level. The program can then perform some other application-related calculations, provided that these are independent from the data of the pending message. However, during these calculations, the program has to call a specific *push function* repeatedly in order to ensure that the communication progress is being fostered in an interleaved manner. Afterwards, the program has to call an additional *test* or *wait function* in order to check whether the message transfer has been completed. This approach should be illustrated by the following pseudo-code:

```
# Initialize the non-blocking receive request:
CALL non_blocking_receive(OUT request_handle)

# Perform the interleaved computation and communication:
# (the calculation task can be divided into n subtasks)
FOR i = 1 TO n
   CALL partial_calculation(IN i)
   CALL push_communication(VOID)
NEXT i = i + 1

# Ensure that the communication has yet been completed:
CALL wait_for_communication_completion(IN request_handle)
```

Of course, not only the receiving of a message can be conducted in such a non-blocking and interleaved manner, but also the sending of a message can be processed in a similar way. The considerations for non-blocking send requests are quite the same as stated above for the receive request and even the programming patterns are quite analogous. Therefore, we do not present an exemplary non-blocking send scenario at this point.

3.3 Interleaved vs. Overlapped Communication

On systems, where the communication can be conducted autonomously by a communication controller (e.g. a DMA engine) or by a communication thread (e.g. on a multi-core CPU), not only an interleaving but rather a true *overlapping* of communication and computation can be achieved. The main difference between interleaved and overlapped communication is that interleaving implies a concurrent but still serialized processing, whereas a true overlapping results in a parallel processing of communication and computation. Thus, the first approach helps to hide wait states and to break up message dependencies (see Section 3.5 for an example), but on the other hand it requires an explicit switch-over between computation and communication by frequently calling the push function. In contrast to this, the second approach can achieve real parallelism and thus does not need an explicit pushing for progress.[5] However, since there is no other asynchronous hardware (like a communication controller) available for the cores on the SCC, the iRCCE library (currently) just implements the first approach.

3.4 The Non-Blocking Communication Approach of iRCCE

As stated above, the iRCCE library implements an interleaving mechanism for non-blocking communication operations. This is achieved by using the standard RCCE communication functions[6] as a template for their iRCCE counterparts, but instead of waiting for a progress flag[7] to be set, the non-blocking iRCCE functions just test the respective flag[8] and return immediately if a progress cannot be made right away. In order to resume the communication late on, the current progress states are stored in request handles that are of type `iRCCE_SEND/RECV_REQUEST`[9]. These handles are set up and returned by the respective non-blocking send/receive functions (`iRCCE_isend()`/`iRCCE_irecv()`) that have to be called in order to initiate a non-blocking communication request. Subsequently, the communication progress can be pushed on by calling `iRCCE_isend/irecv_push()` repeatedly. Again, also these functions return immediately if the communication progress can temporarily not be fostered. Finally, the completion of a once pending communication request must be ensured by a call to the `iRCCE_isend/irecv_test()` or to the `iRCCE_isend/irecv_wait()` function with the respective request handle as function parameter.

Before the completion of a non-blocking operation is not ensured by a call to these functions, neither the respective receive buffer is guaranteed to be valid (it is likely that the message has yet not arrived in the receive buffer) nor the respective send buffer is allowed to be modified (it is likely that the message has yet not been copied out of the send buffer).

In order to handle multiple outstanding communication requests, the iRCCE library implements a queuing mechanism. This is necessary, because it is possible to initiate subsequent non-blocking send requests, for example, to the same destination even before the first message gets started to be transferred. Thus, without a queuing mechanism, the order of the messages could not be observed. The iRCCE library implements this queuing mechanism in terms of single-linked lists with *one*[10] send and 48 receive queues per core. If the first element in such a list is not `NULL` (that means that a previous request is still in progress) the new request is just added at the end of the list. Otherwise, the `iRCCE_isend/irecv()` function tries to complete the new request directly. If such a completion cannot be achieved, the request becomes the head of the respective list and the function returns immediately.

[5] Nevertheless, it should be mentioned that the synchronization between a communication and a computation thread, as well as the latency for setting up a communication controller like a DMA engine, can expose an enormous overhead that can even nullify the performance improvements gained by the parallel progress, especially for short messages.

[6] These are the send and receive functions (`RCCE_send()`/`RCCE_recv()`) of the *non-gory* interface of RCCE.

[7] See the `RCCE_wait_until()` calls of `RCCE_send/recv_general() in RCCE_send/recv.c`.

[8] See the `iRCCE_test_flag()` calls of `iRCCE_push_send/recv_request() in iRCCE_isend/irecv.c`.

[9] These are just C structs that are defined in `iRCCE.h`

[10] Consequently, send requests (even to different remote ranks) are always processed in the order of their respective `iRCCE_isend()` calls; whereas receive requests are handled in the order of the message arrival.

According to this, if one wants to foster the communication progress irrespectively from a specific request, just the *first* pending request in the respective queue needs to be pushed; and that is exactly what `iRCCE_isend/irecv_push()` does. In contrast to this, `iRCCE_isend/irecv_test()` only checks and pushes that request that is passed as the function argument. However, these test functions can alternatively be called with `NULL` as the argument indicating that the respective queue as a whole is meant. And likewise, when calling `iRCCE_isend/irecv_wait()` with `NULL` as the argument, the completion of the whole pending queue is waited for. Furthermore, even `iRCCE_isend/irecv()` can be called with `NULL` as the request argument. In such a case, a subsequent `wait()` call will be issued internally.

3.5 An Example Code: The Ping-Ping Pattern

In this section, we want to detail the communication kernel of the Ping-Ping benchmark that is part of the iRCCE distribution (see Section 2.3). The Ping-Ping pattern (see Figure 2) is an example for common communication patterns where messages are exchanged in a *symmetric* manner. In contrast to the notorious Ping-Pong pattern (see Figure 1), the Ping-Ping pattern is symmetric in this respect that both participating processes do exactly the same: (1) initiate a send request, (2) initiate a receive request and then (3) wait for their completion:

```
iRCCE_SEND_REQUEST send_request;
iRCCE_RECV_REQUEST recv_request;

char send_buffer[length];
char recv_buffer[length];

int my_rank     = RCCE_ue();
int remote_rank = (my_rank + 1) % 2;

RCCE_barrier(&RCCE_COMM_WORLD);


timer = RCCE_wtime();

for(round=0; round < numrounds+1; round++)
{
   /* (1) send PING via non-blocking send: */
   iRCCE_isend(send_buffer, length, remote_rank, &send_request);

   /* (2) receive PING via non-blocking recv: */
   iRCCE_irecv(recv_buffer, length, remote_rank, &recv_request);

   /* (3) wait for completion: */
   iRCCE_isend_wait(&send_request);
   iRCCE_irecv_wait(&recv_request);
}

timer = RCCE_wtime() - timer;
```

As one can see, without non-blocking communication functions, this pattern could not be realized because the symmetric blocking send calls would both stuck in anticipation of the matching but subsequent receive calls.

4 Extended Range of Functions

Besides the yet presented iRCCE functions for handling non-blocking communication, we have also added some more higher-level functions for a more convenient handling of outstanding non-blocking requests. This additional part of the iRCCE API should be detailed in this section.

4.1 Overview of the Additional iRCCE Functions

Cancel and Waitlist Functions:

- `int iRCCE_isend_cancel(iRCCE_SEND_REQUEST*, int*);`
- `int iRCCE_irecv_cancel(iRCCE_RECV_REQUEST*, int*);`
- `void iRCCE_init_wait_list(iRCCE_WAIT_LIST*);`
- `void iRCCE_add_to_wait_list(iRCCE_WAIT_LIST*,iRCCE_SEND_REQUEST*,iRCCE_RECV_REQUEST*);`
- `int iRCCE_test_all(iRCCE_WAIT_LIST*, int*);`
- `int iRCCE_wait_all(iRCCE_WAIT_LIST*);`
- `int iRCCE_test_any(iRCCE_WAIT_LIST*, iRCCE_SEND_REQUEST**, iRCCE_RECV_REQUEST**);`
- `int iRCCE_wait_any(iRCCE_WAIT_LIST*, iRCCE_SEND_REQUEST**, iRCCE_RECV_REQUEST**);`

Tagged Flags Functions:

- `int iRCCE_flag_alloc_tagged(RCCE_FLAG*);`
- `int iRCCE_flag_write_tagged(RCCE_FLAG*, RCCE_FLAG_STATUS, int, void*, int);`
- `int iRCCE_flag_read_tagged(RCCE_FLAG, RCCE_FLAG_STATUS, int, void*, int);`
- `int iRCCE_wait_tagged(RCCE_FLAG, RCCE_FLAG_STATUS, void*, int);`
- `int iRCCE_test_tagged(RCCE_FLAG, RCCE_FLAG_STATUS, int*, void*, int);`

Atomic Increment Register Functions: and Improved Collectives:

- `int iRCCE_atomic_alloc(iRCCE_AIR**);`
- `int iRCCE_atomic_inc(iRCCE_AIR*, int*);`
- `int iRCCE_atomic_read(iRCCE_AIR*, int*);`
- `int iRCCE_atomic_write(iRCCE_AIR*, int);`

Improved Collectives:

- `int iRCCE_barrier(RCCE_COMM* comm);`
- `int iRCCE_bcast(RCCE_COMM comm);`
- `int iRCCE_msend(char *, size_t);`
- `int iRCCE_mrecv(char *, size_t, int);`

4.2 Functions for Canceling Requests

First of all, we have added functions for canceling already enqueued but not yet started send and receive requests (`iRCCE_isend_cancel()`/`iRCCE_irecv_cancel()`). By means of these functions, it can be *attempted* to remove such a request from the respective waiting queue until it becomes the head of it. However, if the request has already become the first and the actual communication has already been started, a subsequent canceling is no longer possible. In such a case, the cancel function returns the information that the requested withdrawal has failed. But this in turn means that the started request has to be matched by a respective send or receive call on the remote side.[11] Therefore, the application programmer should be clear in one's mind about the fact that a cancel call may fail and that the application's communication pattern must be designed in such a way that all started non-blocking requests are resolved.

4.3 Functions for Handling Multiple Outstanding Requests

In many cases one wants request some amount of non-blocking send/receive operations, then do calculations in some kind of loop and from time to time test if the send/receive operations have succeeded. For bigger amounts of messages one will certainly write a loop for testing all those requests. Those loops are an unnecessary and uncomfortable duplication of code, so we wrote functions to handle this in a more elegant manner. `iRCCE_test_all()` will traverse a linked list of send/receive requests and return success if there have been no pending requests, while `iRCCE_wait_all()` is not returning until there are no pending requests left. One only have to initialize a data structure of type `iRCCE_WAIT_LIST` and add requests to it first. Analogous to this, `iRCCE_test_any()`/`iRCCE_wait_any()` can be used for testing or waiting for completion of *any* pending operation in the list.

4.4 Functions for Handling Tagged Flags

Former versions of the original RCCE (up to RCCE release V1.0.13) used one whole cache-line of 32 Byte per `RCCE_FLAG`; at least when *not* using the "single bit flags" mode.[12] In the common use of such cache-line-sized flags, only one integer word of 4 Byte is used for the actual synchronization whereas the remainder is unused. However, when using iRCCE's *Tagged Flags*, the remaining 28 Bytes can be used for small payloads in a *piggyback* fashion. This is because on hardware level, all data exchange via the mesh is conducted in cache-line granularity. For sending such payload alongside with a flag, one can use the function `iRCCE_flag_write_tagged()` which is quite similar to the common `RCCE_flag_write()` function with the exception that it also takes a pointer and the length of a payload buffer as additional arguments.

Caution: If the payload length is bigger than the remaining bytes of the cache-line, the message gets truncated and only the first 28 Bytes will be transferred!

Please note that when using the OpenMP-emulated mode of RCCE, there is one more word needed for synchronization. Therefore, only 24 Bytes are left for payload in this case. Hence, if one wants to write portable iRCCE applications, one should check the actual available payload size[13] before using `iRCCE_flag_write_tagged()`. On receiver side, a call of `iRCCE_wait_tagged()` will not only block until the respective flag is set, but will also copy the sent piggyback payload into an additional receive buffer.

[11] This is because otherwise the pending request will wait eternally at the head of the queue and will hinder all following requests from being processed.
[12] Please notice that Tagged Flags *cannot* be used in "single bit flags" mode!
[13] This can be done via the macro `iRCCE_MAX_TAGGED_LEN` or via a call of `iRCCE_get_max_tagged_len()`.

As a simple example, just consider the following Ping-Pong scheme, which is quite similar to that which can be found in `iRCCE_tagged.c`:

```
RCCE_FLAG mailbox;
iRCCE_flag_alloc_tagged(&mailbox);
. . .
if(my_rank == 0) {

    /* Ping: */
   iRCCE_flag_write_tagged(&mailbox, RCCE_FLAG_SET, remote_rank, send_buffer, length);

   /* Pong: */
   iRCCE_wait_tagged(mailbox, RCCE_FLAG_SET, recv_buffer, length);
   iRCCE_flag_write(&mailbox, RCCE_FLAG_UNSET, my_rank);
}
else {

   /* Ping: */
   iRCCE_wait_tagged(mailbox, RCCE_FLAG_SET, recv_buffer, length);
   iRCCE_flag_write(&mailbox, RCCE_FLAG_UNSET, my_rank);

   /* Pong: */
   iRCCE_flag_write_tagged(&mailbox, RCCE_FLAG_SET, remote_rank, send_buffer, length);
}
```

A major advantage of using tagged flags for exchanging small messages instead of using the common send/recv functions is a significant latency improvement. In fact, when building iRCCE against RCCE V1.0.13, even the latencies for small messages sent via the common `iRCCE_ssend()`/`iRCCE_srecv()` functions can benefit from the internal usage of tagged flags.[14] Since up to RCCE V1.0.13 there is no difference between tagged flags and common flags, both types can be allocated and used with both APIs (RCCE in *gory* mode and iRCCE). However, when building iRCCE against a more recent RCCE version, the new function `iRCCE_flag_alloc_tagged()` must be used for allocating tagged flags in order to assure that a whole cache-line is assigned to such a flag. Please notice that iRCCE uses the non-gory mode in which all once allocated resources can no more be freed during a session. For this reason, iRCCE offers no corresponding free function.

4.5 Functions for Handling Atomic Increment Registers

Since the release of sccKit 1.4, the FPGA features two sets of 48 atomic counters. These 32 bit counters are mapped via LUT entries into the cores' address space and can be used in terms of memory mapped registers. Therefore, each atomic counter is represented by a pair of two registers: the actual *increment register* and an additional *initialization register*. While a read operation on the increment register atomically increments its value and returns its former value, the initialization register cab be used to preload the counter and to read its value without modification.

When using the iRCCE API, each of the 96 atomic increment registers (AIRs) must be allocated via a call of `iRCCE_atomic_alloc()` before it can be used. The function `iRCCE_atomic_write()` can then be used to initialize the counter read and `iRCCE_atomic_read()` returns the current value without modification. Finally, `iRCCE_atomic_inc()` is the function that atomically increments the respective counter. **For using these functions, iRCCE must be built via > `make AIR=1`!**

[14] *But also a word of caution:* On the other hand, when using one cache-line per flag for all flags, the amount of MPB space available for larger messages gets reduced, what can in turn impact the obtainable throughput!

5 Other Improvements of iRCCE to RCCE

5.1 SCC-optimized Memory Copy Functions

On the current SCC architecture, a write access does *not* perform a cache line fill even if the cache line is not present (*cache on read not on write*). Therefore, the core writes in this case directly to the main memory and, of course, this is quite expensive in terms of time. However, if a present cache line is modified, these changes are are not directly applied to main memory but just to the cache. Thus, when touching a cache line by a read access before modifying its content, subsequent write accesses to this line do not imply a write through to the main memory. The following SCC-customized functions copy memory from the on-die buffers (MPB) to an off-die region while pre-fetching the cache lines of the destination: `iRCCE_put()` / `iRCCE_memcpy_put()`. Therefore, these functions avoid the above mentioned SCC-specific bad behavior of write misses. In turn, if the destination is located in on-die memory (MPB), classical pre-fetching techniques[15] are used by the following functions in order to increase the copy performance: `iRCCE_get()` / `iRCCE_memcpy_get()`. All these functions are internally used by the iRCCE communication functions; but they can also be utilized outside of the library. Please notice: Allocating MPB space in an explicit manner via `RCCE_malloc` is actually not supported in RCCEs non-gory mode. Therefore, one can use the corresponding iRCCE function `iRCCE_malloc`, which is (more or less) just a wrapper that is available in non-gory mode, too.

5.2 Pipelined Send and Receive Functions

The common RCCE send and receive functions[16] use the core-local MPB space for sending messages to remote cores. As one may know, these local buffers are 8KByte MPB space per core. If a message to be sent is bigger than the available local MPB space, it must be divided into chunks that are then sent piecewise. For this purpose, the common RCCE functions determine a chunk size that is equal to the amount of the available local MPB space. However, when using the whole available local MPB space for one single message chunk, the message transport becomes necessarily a serialized process: (1a) sender puts chunk into the MPB, (1b) sender signalizes the chunk's arrival, (2a) receiver gets chunk from the MPB, (2b) receiver signalizes the chunk's removal (see Figure 3). This processing scheme must then be performed iteratively until the whole message has been transferred.

Therefore, a smarter approach is not to use the whole local MPB as one big chunk but to divide it into two smaller chunks. This is because in this case sender and receiver can work on the MPB simultaneously in a pipelined and parallelized manner: (1a) sender puts message chunk A into the MPB, (1b) sender signalizes the arrival of chunk A, (2a) sender puts message chunk B into the MPB; and *meanwhile*, the receiver can remove chunk A from the MPB, (2b) sender/receiver signalize the arrival/removal of chunk B/A; and so on (see Figure 4). The differences between both approaches can be made clear by comparing Figure 3 and Figure 4.

This pipelining approach is implemented within iRCCE in terms of the blocking `iRCCE_ssend()` and `iRCCE_srecv()` functions. It is important that each call of these functions is matched with a call of its respective counterpart on the remote side. That means that a message that is sent via `iRCCE_ssend()` must necessarily be received via `iRCCE_srecv()`; and vice versa. This is because the pipelining requires that both the sender and the receiver cooperate *synchronously* according to the pipelining technique. That this technique can really help to improve the communication performance especially for larger messages can be proved by applying appropriate benchmark tool, as for example shown here in Section 8. However, one should remember that this technique (at least in its current implementation) just takes effect for messages that are bigger than the local MPB space of 8KByte.

[15] See `include/scc_memcpy.h` for more details.
[16] as well as the non-blocking and the blocking send and receive functions of iRCCE

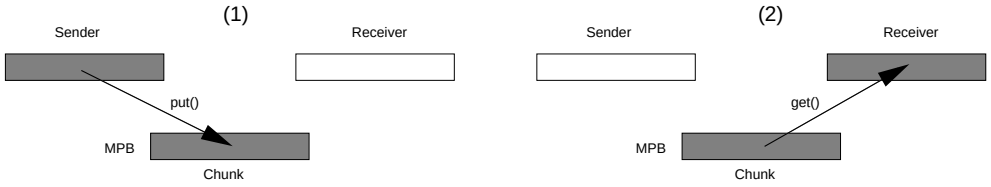

Figure 3: Serialized Transport of Message Chunks through the MPB Space

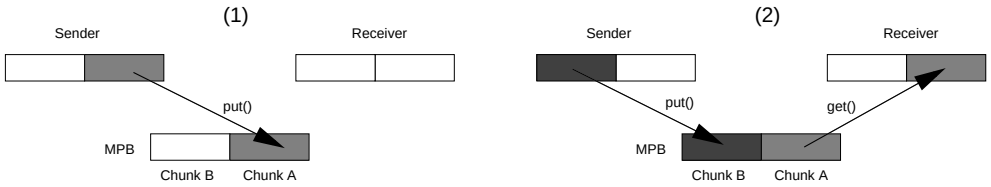

Figure 4: Pipelined Transport of Message Chunks through the MPB Space

5.3 Some Improved Collective Functions

The first improved collective function offered by iRCCE performs a *broadcast* (or multicast) pattern. On the one hand, the improvement (with respect to throughput performance) of this function is based on the pipelined communication functions of iRCCE. However, instead of just using these function e.g. in the context of a common tree-based broadcast approach, we have modified them to realize a *single-writer/multiple-reader* pattern. While the sender (the source of the broadcast) puts/writes the message (or a at least a message chunk) into its local MPB, all receivers concurrently get/read the message from this remote MPB region. This approach helps to increase the throughput performance especially in cases where a lot of receivers are involved into the broadcast pattern, as one can see in Table 2 in Section 8. For doing so, one can use the iRCCE function `iRCCE_bcast()`. However, in fact this function is just a wrapper around the so-called *multicast* functions of iRCCE: `iRCCE_msend()` and `iRCCE_mrecv()`, as it is shown in the following code fragment.[17] As one can see, these improvements just take effect when the `RCCE_COMM_WORLD` communicator is used. Otherwise, the `iRCCE_bcast` function just makes a fallback to a call of the common broadcast function of RCCE:

```
int iRCCE_mcast(char *buf, size_t size, int root)
{
  if(RCCE_IAM != root)  return iRCCE_mrecv(buf, size, root);
  else  return iRCCE_msend(buf, size);
}

int iRCCE_bcast(char *buf, size_t size, int root, RCCE_COMM comm)
{
  if(memcmp(&comm, &RCCE_COMM_WORLD, sizeof(RCCE_COMM)) == 0)
        return RCCE_bcast(buf, size, root, comm);
  else  return iRCCE_mcast(buf, size, root);
}
```

[17] This code is taken from iRCCE application *multicast* (see `iRCCE_mcast.c`).

The second improved collective is the *barrier* function. Here, the extended Atomic Increment support (for the SCC a set of 96 Atomic Increment Counters/Registers) is used for the implementation of a simple SMP-like centralized barrier algorithm. The implementation is straight forward: Within a gather phase, each incoming UE increments one of two counters. Target counter is iteratively exchanged, dependent if the epoch is even or odd. A reset of the counter value is triggered by the last UE which has entered the barrier to release the barrier. Accordingly, the release phase consists of busy waiting on the initialization register of target Atomic Increment Counter except for the last UE. As this behavior obviously creates a high contention, a certain backoff (`RC_wait(...)`) is necessary to avoid an overload situation for target off-chip located Synchronization Register [8].

For the common RCCE implementation, the communicator holds information on exactly two synchronization flags (location within a cache-line and pointer to cache-line and byte containing the flag) for the realization of a barrier support. This communicator structure corresponds to the linear barrier algorithm, where one flag is used to gather and the other flag is used to release all UE's in a master follower approach.

iRCCE allocates the first two Atomic Increment Counters (`iRCCE_atomic_barrier`) for the realization of the described central barrier algorithm as a workaround for the global communicator (`RCCE_COMM_WORLD`). For the access of atomic operations, the new interface from Section 9.10 is used. As a fallback solution for all other communicators, the common linear algorithm of RCCE is used.

```
int iRCCE_barrier(RCCE_COMM comm)
{
  int counter, epoch;

  if( comm == RCCE_COMM_WORLD )
  {
    flag_read(comm.gather,&epoch);
    iRCCE_AIR* reg = iRCCE_atomic_barrier[epoch];
    iRCCE_atomic_inc(reg, &counter);
    if(counter < comm.size)
      while(iRCCE_atomic_read(reg,&counter)) RC_wait(...);
    else
      iRCCE_atomic_write(reg,0);

    epoch = !epoch;
    flag_write(comm.gather,epoch);

      return(RCCE_SUCCESS);
  }
  else
    return RCCE_barrier(comm);
}
```

6 Wildcard Mechanisms

The recent version of iRCCE also supports wildcards for message length (`iRCCE_ANY_LENGTH`) and source parameter (`iRCCE_ANY_SOURCE`) when calling the blocking or non-blocking receive function of iRCCE (`iRCCE_srecv()`/`iRCCE_irecv()`). This chapter details their usage and shows a short example code.

6.1 ANY_LENGTH: Receiving Messages with arbitrary Sizes

By using the additional `iRCCE_ANY_LENGTH` symbol[18] as a wildcard for the message length parameter, the receive functions are told to accept incoming messages of an *arbitrary* size. After receiving a message by using this wildcard, the length of the actual payload can then be determined by calling the `iRCCE_get_length()` or `iRCCE_get_size()` function (see Section 9.8). For that purpose, the current message length is passed to the receiver prior to the actual payload. This is done by "*hijacking*" the synchronization flags that are needed to get a message through the MPB. That means that the sender not only signals the receiver that a message (or the first chunk of a message) has been copied into the MPB but also informs the receiver about the total length of this message by means of these flags.[19] However, this in turn implies that such a flag must be typed as an integer and not just as a single bit. For this reason, this wildcard mechanism only works in the so-called *big flags* case of RCCE release V1.0.13 where a whole cache line is used per flag.

Therefore, the `iRCCE_ANY_LENGTH` wildcard can only be used when iRCCE is built against RCCE release V1.0.13[20] with single bit flags disabled[21]. This is because cache line-sized flags are needed to submit the actual message length to the receiver. Moreover, when using the `iRCCE_ANY_LENGTH` wildcard, the application programmer has to ensure that the provided receive buffer is large enough to take in the largest possible message that may occur for a specific receive function call![22]

6.2 ANY_SOURCE: Receiving Messages from arbitrary Source Ranks

Using the new symbol `iRCCE_ANY_SOURCE` as a wildcard for the source parameter in `iRCCE_srecv()` or `iRCCE_irecv()`, the receiver is able to receive a message from an *arbitrary* UE. In case of the blocking version this is achieved by a periodic check of the potential sources in terms of a set sent flag. When a source is found the function proceeds as if it was called with a source specified. To find a source in case of the non-blocking version additional effort is needed. A non-blocking call with `iRCCE_ANY_SOURCE` as parameter for the source first tries to find a source by checking the sent flags of the potential sources just like discussed above, however with a little difference. A potential source is not only distinguished by a set sent flag but also by an empty `iRCCE_irecv_queue` of that source. That means no other open receive request may pend on that source. If a source is found an appropriate request is initialized and the futher behaviour is equivalent to a respective `iRCCE_irecv()`-call with that source. Otherwise there is a new queue (`iRCCE_irecv_any_source_queue`) for reserved requests with no defined source. This queue contains initialized received requests with the source member set to `iRCCE_ANY_SOURCE`. When calling one of the functions for handling outstanding requests (see Section 9) an attempt is made to find a source for those requests. On success the request is taken out of the `iRCCE_irecv_any_source_queue` and treated as a normal request. The application programmer only has to be aware that `iRCCE_irecv()`-calls with specified source have *precedence*[23] over those called with `iRCCE_ANY_SOURCE`. Apart from this a call with this wildcard behaves as usual.

[18] In fact, this symbol is just a global and constant integer value. (`const int iRCCE_ANY_LENGTH = -1`)
[19] Please notice that iRCCE does *not* add any headers to the payload of the messages.
[20] This version can be found here: `http://marcbug.scc-dc.com/svn/repository/tags/RCCE_V1.0.13`
[21] Which is the default case! (`make [SINGLEBITFLAGS=0]`)
[22] That means that iRCCE does not check for buffer overflows!
[23] That is because an empty `iRCCE_irecv_queue` is one of the criteria for potential sources.

6.3 A simple Master/Worker Example

The following code shows an example for a Master/Worker pattern that is realized in a smart way by using the `iRCCE_ANY_SOURCE` and `iRCCE_ANY_LENGTH` wildcards:

```
#define max_length 10
char buffer[max_length];

int my_rank = RCCE_ue();
int num_ues = RCCE_num_ues();

if(my_rank == 0)
{
   /* I am the Master! Receive strings from workers:  */

   for(i=1; i<num_ues; i++)
   {
      iRCCE_srecv(buffer, iRCCE_ANY_LENGTH, iRCCE_ANY_SOURCE);

      printf("Message from Worker %d: %s\n", iRCCE_get_source(NULL), buffer);
   }
}
else
{
  /* I am a Worker! Send a random string to the master: */

  sprintf(buffer, "%d", rand());

  iRCCE_ssend(buffer, strlen(buffer), 0);
}
```

6.4 Using Wildcards in Multicast/Broadcast Patterns

One major advantage of having the split functions `iRCCE_msend()` and `iRCCE_mrecv()` is that on receiver side even wildcards can be used for a collective broadcast pattern. So, for example, the receiving processes need not to be informed about the actual message length before the broadcast operation starts: The receivers just call `iRCCE_mrecv()` with the `iRCCE_ANY_LENGTH` wildcard and only the sender has to state the actual message length in the `iRCCE_msend()` call. Moreover, the same applies on receiver side for the actual sender ID: If a process is sure to be receiver, it does not need to know who the actual sender is. It only needs to use the `iRCCE_ANY_SOURCE` wildcard for this purpose. Therefore, the function `iRCCE_mcast()` (see code example in Section 5.3) could also be implemented like this:

```
int iRCCE_mcast(char *buf, size_t size, int root)
{
  if(RCCE_IAM != root)

        return iRCCE_mrecv(buf, iRCCE_ANY_LENGTH, iRCCE_ANY_SOURCE);

  else
        return iRCCE_msend(buf, size);
}
```

7 Known Problems and Issues

- **Differences between MPI and iRCCE Semantics**
  Although some function names of iRCCE are quite similar to their counterparts of the Message Passing Interface Standard (MPI) [7], usage and semantics of both APIs differ in detail. So, for example, the test function of iRCCE only checks and pushes that non-blocking request that is passed via the respective function call. In contrast to this, the MPI test function also triggers the so-called progress engine that ensures that all waiting requests are checked for progress irrespectively from the passed request of the function call.

- **Matching of Non-Blocking Requests with Blocking Function Calls**
  Although not extensively tested, a call e.g. of the blocking RCCE send function (`RCCE_send()`) should match with a call of its non-blocking iRCCE counterpart (`iRCCE_irecv()`) on the remote side; and vice versa. *But beware!* This is true for matching non-blocking iRCCE calls with blocking RCCE calls, but not for mixing non-blocking calls with the blocking but *pipelined* functions of iRCCE, as for example `iRCCE_srecv()`.[24] Moreover, calling a blocking RCCE function after initiating a still pending non-blocking iRCCE function can cause deadlocks because of the missing `push()` calls![25]

- **Mixing of Non-Blocking Function Calls with Collective Operations**
  Do not use collective communication operations (like `RCCE_bcast` or `RCCE_barrier`) when there are still outstanding non-blocking communication requests! This is because since RCCE does not use message tags like MPI, iRCCE cannot distinguish internally between collective and point-to-point requests. Thus, an overlapping of non-blocking transfers with collective communication patterns can lead to message mismatches and deadlocks.

- **Allocating MPB Space or Flags during Non-Blocking Communication**
  Do not allocate new flags or new MPB regions when there are still outstanding non-blocking communication requests! This is because the amount and the position of the MPB space used for the asynchronous data transfers is recorded at creation time of the respective non-blocking communication request and internally used throughout the request's completion.

8 Performance Results and Comparisons

In this section, we want to present some performance results. In Figure 5, one can see the Ping-Pong bandwidth[26] (measure with the iRCCE Ping-Pong benchmark, see Section 2.3) for different message sizes and different optimization approaches. All measurements were done between the cores `rck00` and `rck01` with network and memory running at 800MHz and a core frequency of 533MHz (`Tile533_Mesh800_DDR800`). The used libraries were standard RCCE (V 1.0.13, `big_flags`, `nongory`) and our iRCCE. The figure shows a performance comparison between the *standard RCCE* functions (`RCCE_send/recv()`), the utilization of *improved memory copy* functions (`iRCCE_memcpy_put/get()`) and the the applying of *Pipelining* (`iRCCE_ssend/srecv()`) for long messages ($length \geq 8192$). As one can see, iRCCE outperforms RCCE with respect to the communication bandwidth especially in case of larger messages.

Regarding the latencies for short (1 Byte) messages,[27] the performance depends on the RCCE version which iRCCE is built against. When using a version up to RCCE V1.0.13 (which uses one cache line per flag), the latencies can benefit from iRCCE's *Tagged Flags* feature, as it is shown in Table 1.

[24] This is because the pipelining requires that both the sender and the receiver cooperate *synchronously* according to the pipelining technique (see Section 5.2).
[25] Better use `iRCCE_isend/irecv()` with `NULL` as the request argument in this case because the internal `wait()` call avoids such deadlocks (see Section 3.4).
[26] See [6] for a detailed analysis of the bandwidth curve progression and its correlation with cache level sizes.
[27] Please notice that iRCCE's `iRCCE_ssend/srecv()` functions are (in contrast to RCCE's `RCCE_send/recv()`) *synchronizing* also for *Zero Byte* messages!

However, when using a more recent version of RCCE, the latencies measured with the iRCCE Ping-Pong benchmark for `iRCCE_ssend/srecv()` suffer a little bit in comparison to the original RCCE functions. This is because the iRCCE functions additionally try to trigger still pending non-blocking request before starting the actual communication and this causes an additional overhead.

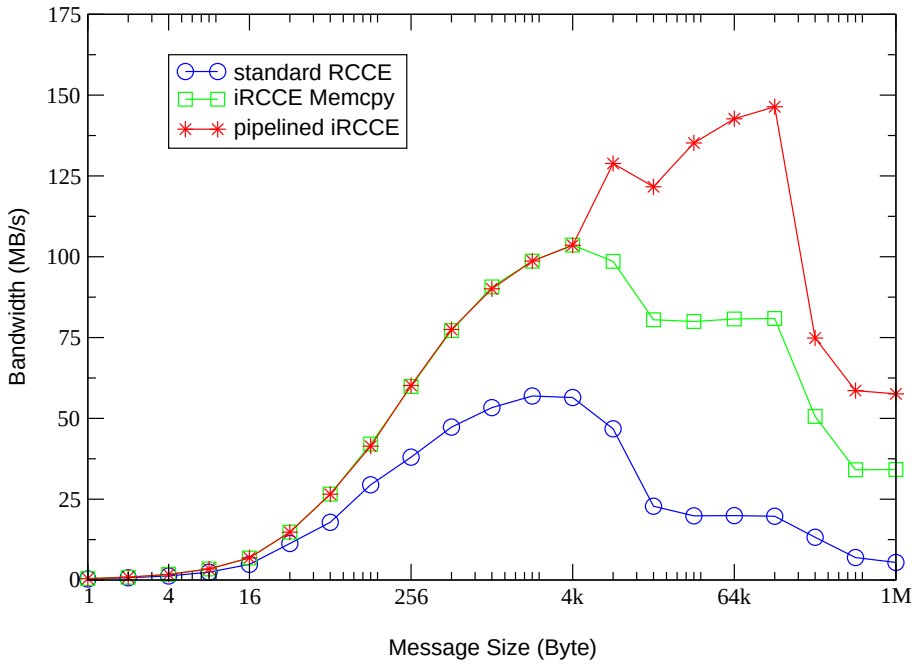

Figure 5: Some Results of the *iRCCE Ping-Pong* Benchmark: Performance comparison in terms of communication bandwidth between the *standard RCCE* functions (`RCCE_send/recv()`), the utilization of *improved Memory-Copy* functions (`iRCCE_memcpy_put/get()`) and the the applying of *Pipelining* (`iRCCE_ssend/srecv()`) for long messages ($length \geq 8192$)

Latencies in μs	RCCE	iRCCE
RCCE V1.0.13	2.84	1.65
RCCE V1.1.0	1.78	1.83
RCCE V2.0	1.78	1.83

Table 1: Latencies measured by using the iRCCE Ping-Pong benchmark for 1 Byte messages between `rck00` and `rck01`. All compilation was done by using the `icc` (not the `icpc`) compiler and the SCCs configuration was set to `Tile533_Mesh800_DDR800`.

RCCE	RCCE_comm	iRCCE	iRCCE AIR
3.3 MB/s	26.6 MB/s	71.3 MB/s	77,7 MB/s

Table 2: Broadcast/Multicast Performance: Bandwidth/Throughput for a 64kB message sent from one sender (the root) to 47 receivers. The SCCs configuration was set to `Tile533_Mesh800_DDR800`.

9 The iRCCE Application Programming Interface

9.1 Library Initialization Function

int iRCCE_init(void)

This function must be invoked before any other iRCCE function is called. The function initializes crucial data structures of the iRCCE library as for example the queues for pending communication requests. The return value should always be `iRCCE_SUCCESS`.

9.2 Functions for Non-Blocking Sending

int iRCCE_isend(char *buffer, size_t length, int dest, iRCCE_SEND_REQUEST *request)

This function initiates a request for a non-blocking send operation. The function initializes a request handle of type `iRCCE_SEND_REQUEST` that must be passed as a pointer and that is used later on for checking for the operation's completion. The function returns `iRCCE_SUCCESS` in that case that the request could be finished already within this function call. However, usually the function returns `iRCCE_PENDING` or `iRCCE_RESERVED` indicating that the communication has been started but not yet finished or that there are prior requests pending in the send queue and that the request is being reserved. For more details about this function see Section 3.

`buffer`	starting address of the message to be sent
`length`	length of the outgoing message in bytes
`dest`	rank (ID) of the target process (UE)
`request`	request handle that is used later on for checking for completion (if set to `NULL`, a subsequent `iRCCE_isend_wait()` call will be issued internally)

int iRCCE_isend_test(iRCCE_SEND_REQUEST *request, int *flag)

This function checks whether an initiated request for a non-blocking send operation is already completed. The function checks and pushes only that request that is passed as the request handle argument. However, if `NULL` is passed as the argument, the function checks for the completion of the whole pending send queue. The function is non-blocking and the return values and their meanings are the same as for `iRCCE_isend()`. Therefore, the argument `flag` can also be omitted by passing `NULL` instead.

`request`	request handle returned by `iRCCE_isend()`
`flag`	flag that indicates whether the message has been sent (`1`) or not (`0`)

int iRCCE_isend_wait(iRCCE_SEND_REQUEST *request)

This function can be used for waiting for the completion of a pending send request. Since this function blocks until the respective request is finished, the function also pushes the pending receive queue as well as the pending send requests that are enqueued prior to that request in order to avoid deadlocks. However, if `NULL` is passed as the request handle argument, the completion of the whole pending send queue is waited for. The return value should always be `iRCCE_SUCCESS`.

`request`	request handle returned by `iRCCE_isend()`

int iRCCE_isend_push(void)

This function pushes the progress of non-blocking communication requests that are enqueued in the pending send queue. The function is non-blocking and the return value is `iRCCE_PENDING` in case of still pending send requests or `iRCCE_SUCCESS` if the send queue is empty.

9.3 Functions for Non-Blocking Receiving

int iRCCE_irecv(char *buffer, size_t length, int source, iRCCE_RECV_REQUEST *request)

This function initiates a request for a non-blocking receive operation. The function initializes a request handle of type `iRCCE_RECV_REQUEST` that must be passed as a pointer and that is used later on for checking for the operation's completion. The receiver may specify `iRCCE_ANY_SOURCE` as wildcard for the source parameter, indicating that any source is acceptable (see also Section 6). The function returns `iRCCE_SUCCESS` in that case that the request could be finished already within this function call. However, usually the function returns `iRCCE_PENDING` or `iRCCE_RESERVED` indicating that the communication has been started but not yet finished or that there are prior requests pending in the receive queue and that the request is being reserved. For more details about this function see Section 3.

`buffer`	starting address of the receive buffer
`length`	length of the expected message in bytes
`source`	rank (ID) of the source process (UE)
`request`	request handle (if set to `NULL`, `iRCCE_irecv_wait()` will be called internally)

int iRCCE_irecv_test(iRCCE_RECV_REQUEST *request, int *flag)

This function checks whether an initiated request for a non-blocking receive operation is already completed. The function checks and pushes only that request that is passed as the request handle argument. However, if `NULL` is passed as the argument, the function checks for the completion of the whole pending receive queue. The function is non-blocking and the return values and their meanings are the same as for `iRCCE_irecv()`. Therefore, the argument `flag` can also be omitted by passing `NULL` instead.

`request`	request handle returned by `iRCCE_irecv()`
`flag`	flag that indicates whether the message has been received (`1`) or not (`0`)

int iRCCE_irecv_wait(iRCCE_RECV_REQUEST *request)

This function can be used for waiting for the completion of a pending receive request. Since this function blocks until the respective request is finished, the function also pushes the pending send queue as well as the pending receive requests that are enqueued prior to that request in order to avoid deadlocks. However, if `NULL` is passed as the request handle argument, the completion of the whole pending receive queue is waited for. The return value should always be `iRCCE_SUCCESS`.

`request`	request handle returned by `iRCCE_irecv()`

int iRCCE_irecv_push(void)

This function pushes the progress of non-blocking communication requests that are enqueued in the pending receive queues. The function is non-blocking and the return value is `iRCCE_PENDING` in case of still pending receive requests or `iRCCE_SUCCESS` if the receive queue is empty.

int iRCCE_iprobe(int source, int* test_rank, int* test_flag)

This function just probes if a message *could* be currently received (from a sender with ID `source`) by a call of `iRCCE_irecv()`. This function is non-blocking and returns the current status within the `test_flag` parameter. Instead of a specific `source` ID, the `iRCCE_ANY_SOURCE` wildcard can here be used, too. In such a case, the `test_rank` parameter returns the ID of the actual sender.

`source`	in: rank (ID) of the source process (UE) or `iRCCE_ANY_SOURCE` wildcard
`test_rank`	out: rank (ID) of the actual source process (UE) (can be set to `NULL`)
`test_flag`	flag that indicates whether a message could currently be received or not

9.4 Blocking but Pipelined Communication Functions

int iRCCE_ssend(char *buffer, size_t length, int dest)

This function is quite similar to the blocking `RCCE_send()` function of RCCE. The main difference is that this function is *synchronizing* and that a pipeline technique for larger messages is used (see also Section 5 and Section 8). The threshold value for the message size, when pipelining should take effect, is 8KByte per default and the function call must be matched by a remote call of `iRCCE_srecv()`.

`buffer`	starting address of the message to be sent
`length`	length of the outgoing message in bytes
`dest`	rank (ID) of the target process (UE)

int iRCCE_srecv(char *buffer, size_t length, int source)

This function is quite similar to the blocking `RCCE_recv()` function of RCCE. The main difference is that this function is *synchronizing* and that a pipeline technique for larger messages is used (see also Section 5 and Section 8). The receiver may specify `iRCCE_ANY_SOURCE` as wildcard for the source parameter, indicating that any source is acceptable (see also Section 6). The threshold value for the message size, when pipelining should take effect, is 8KByte per default and the function call must be matched by a remote call of `iRCCE_ssend()`.

`buffer`	starting address of the receive buffer
`length`	length of the expected message in bytes
`source`	rank (ID) of the source process (UE)

int iRCCE_probe(int source, int* test_rank)

This function just probes if a message *could* be received (from a sender with ID `source`) by a call of `iRCCE_srecv()`. Otherwise it blocks until a matching message arrives. Instead of a specific `source` ID, the `iRCCE_ANY_SOURCE` wildcard can here be used, too. In such a case, the `test_rank` parameter returns the ID of the actual sender.

`source`	in: rank (ID) of the source process (UE) or `iRCCE_ANY_SOURCE` wildcard
`test_rank`	out: rank (ID) of the actual source process (UE) (can be set to `NULL`)

9.5 SCC-customized Put/Get and Mem-Copy Functions

int iRCCE_put(t_vcharp target, t_vcharp source, int size, int rank)

This is the SCC-optimized version of the `RCCE_put()` function (see Section 5.1 for more details). The function copies the contents of the buffer pointed to by `source` into the MPB location pointed to by `target`. The data type `t_vcharp` is similar to `volatile char*` and defined in `RCCE.h`.

`target`	an MPB address that will be converted to the appropriate address on the UE
`source`	start address of the data in private memory of the calling UE
`size`	size of the data to be put into the MPB in bytes
`rank`	rank (ID) of the target process (UE)

int iRCCE_get(t_vcharp target, t_vcharp source, int size, int rank)

This is the SCC-optimized version of the `RCCE_get()` function (see Section 5.1 for more details). The function copies the contents of the MPB location pointed to by `source` into the buffer pointed to by `target`. The data type `t_vcharp` is similar to `volatile char*` and defined in `RCCE.h`.

`target`	address of the destination buffer in private memory of the calling UE
`source`	an offset that, when combined with the remote rank, points to the source MPB
`size`	size of the data to be gotten from the MPB in bytes
`rank`	rank (ID) of the source process (UE)

void* iRCCE_memcpy_put(void* dest, const void* src, size_t num)

This function copies `num` bytes from memory area `src` to memory area `dest` in an SCC-optimized manner (see also Section 5.1). It can be used instead of the common `memcpy()` routine of `<string.h>`.

`dest`	start address of the destination memory area
`src`	start address of the source memory area
`num`	number of bytes to be copied from source to destination

void* iRCCE_memcpy_get(void* dest, const void* src, size_t num)

This function copies `num` bytes from memory area `src` to memory area `dest` in an SCC-optimized manner (see also Section 5.1). It can be used instead of the common `memcpy()` routine of `<string.h>`.

`dest`	start address of the destination memory area
`src`	start address of the source memory area
`num`	number of bytes to be copied from source to destination

t_vcharp iRCCE_malloc(size_t size)

Allocating MPB space in an explicit manner via `RCCE_malloc` is actually just supported in RCCE's *gory* mode. Therefore, this corresponding iRCCE function is (more or less) just a wrapper that is available in non-gory mode, too. However, in contrast to the behavior of the common `RCCE_malloc` function in gory mode, it is not possible to free a once allocated MPB region afterwards. For this reason, there

does *not* exist a corresponding iRCCE free function. The `iRCCE_malloc` function returns `NULL` if `size` is not a multiple of 32 or if the whole MPB space is already allocated.

`size`	number of bytes to allocate in MPB; must be a multiple of 32

9.6 Cancel Functions for Non-blocking Requests

int iRCCE_isend_cancel(iRCCE_SEND_REQUEST *request, int *flag)

This function tries to cancel a yet not finished non-blocking send request. If the request has not yet been started of being processed, the canceling should be successful, otherwise this request can no longer be canceled. The returned flag value indicates whether the canceling was successful or not. For more details about this function see Section 4.2.

`request`	request that should be removed from the waiting send request queue
`flag`	flag that indicates whether the canceling was successful (`1`) or not (`0`)

int iRCCE_irecv_cancel(iRCCE_RECV_REQUEST *request, int *flag)

This function tries to cancel a yet not finished non-blocking receive request. If the request has not yet been started of being processed, the canceling should be successful, otherwise this request can no longer be canceled. The returned flag value indicates whether the canceling was successful or not. For more details about this function see Section 4.2.

`request`	request that should be removed from the waiting receive request queue
`flag`	flag that indicates whether the canceling was successful (`1`) or not (`0`)

9.7 Functions for Handling Multiple Outstanding Requests

void iRCCE_init_wait_list(iRCCE_WAIT_LIST* wait_list)

This function initializes a wait-list and must be called before adding pending communication requests to the respective wait-list. A wait-list of type `iRCCE_WAIT_LIST` can handle both send and receive requests.

`wait_list`	a pointer to an object of the opaque iRCCE data type `iRCCE_WAIT_LIST`

iRCCE_add_to_wait_list(iRCCE_WAIT_LIST* wait_list, iRCCE_SEND_REQUEST *send_request, iRCCE_RECV_REQUEST *recv_request)

This function adds a pending send and/or receive request to a wait-list that must be initialized before by `iRCCE_init_wait_list()`. Either the send request argument or the receive request argument can be set to `NULL` if there is not request of the respective type to be added.

`wait_list`	a pointer to the respective wait-list object
`send_request`	a pointer to a pending send request to be added to the wait-list (can be `NULL`)
`recv_request`	a pointer to a pending receive request to be added to the wait-list (can be `NULL`)

iRCCE_test_all(iRCCE_WAIT_LIST* wait_list, int *flag)

This function tests for completion of *all* requests in the passed wait-list. The flag is set to `1`, if all respective requests are finished, or to `0` otherwise. See Section 4.3 for more details about this function.

`wait_list`	a pointer to the respective wait-list object
`flag`	flag that indicates whether the wait-list is processed (`1`) or not (`0`)

iRCCE_wait_all(iRCCE_WAIT_LIST* wait_list)

This function just waits for completion of *all* requests in the passed wait-list. See Section 4.3 for more details about this function.

`wait_list`	a pointer to the respective wait-list object

iRCCE_test_any(iRCCE_WAIT_LIST* wait_list, iRCCE_SEND_REQUEST **send_request, iRCCE_RECV_REQUEST **recv_request)

This function tests for completion of *any* request in the passed wait-list. It returns a pointer to that finished request or `NULL` otherwise. In case of a finished send request, `send_request` points to this request and `recv_request` is set to `NULL`; and vice versa in case of a finished receive request.

`wait_list`	a pointer to the respective wait-list object
`send_request`	returned pointer to a finished send request (or `NULL`)
`recv_request`	returned pointer to a finished receive request (or `NULL`)

iRCCE_wait_any(iRCCE_WAIT_LIST* wait_list, iRCCE_SEND_REQUEST **send_request, iRCCE_RECV_REQUEST **recv_request)

This function just waits for completion of *any* request in the passed wait-list. In case of a finished send request, `send_request` points to this request and `recv_request` is set to `NULL`; and vice versa in case of a finished receive request. The rank (ID) of the respective sender/receiver can afterwards be determined via `iRCCE_get_dest()`/`iRCCE_get_source()` (see Section 9.8).

`wait_list`	a pointer to the respective wait-list object
`send_request`	returned pointer to a finished send request (or `NULL`)
`recv_request`	returned pointer to a finished receive request (or `NULL`)

9.8 Functions for Querying Request Parameters

iRCCE_get_source(iRCCE_RECV_REQUEST* request)

This function returns the source rank (ID) that is associated with a certain receive request. When using the `iRCCE_ANY_SOURCE` wildcard within a receive call (see Section 6), this function can then be used to determine the actual sender after receiving a message. If `NULL` is stated instead of a pointer to a request, the source rank of the *last* incoming message is returned. However, a return value of `-1` indicates a faulty usage of this function.

`request`	request to be inquired

iRCCE_get_dest(iRCCE_SEND_REQUEST* request)

This function returns the rank (ID) of the receiver that is associated with a certain send request.

`request`	request to be inquired

iRCCE_get_status(iRCCE_SEND_REQUEST* send_request, iRCCE_RECV_REQUEST* recv_request)

This function returns the current status of a message that is associated with a certain send or receive request. When inquiring a send request, the pointer to `receive_request` must be set to `NULL`. And vice versa, when inquiring a receive request, the pointer to `send_request` must be set to `NULL`. In contrast to call of a test function, this function does *not* push any pending requests, but just returns the current status.

`send_request`	send request to be inquired (or `NULL`)
`recv_request`	recv request to be inquired (or `NULL`)

iRCCE_get_size(iRCCE_SEND_REQUEST* send_request, iRCCE_RECV_REQUEST* recv_request)

This function returns the size of a message that is associated with a certain send or receive request. When inquiring a send request, the pointer to `receive_request` must be set to `NULL`. And vice versa, when inquiring a receive request, the pointer to `send_request` must be set to `NULL`. If both request pointers are set to `NULL`, the length of the *last incoming* message is returned.

`send_request`	send request to be inquired (or `NULL`)
`recv_request`	recv request to be inquired (or `NULL`)

iRCCE_get_length(void)

This function is similar with a call of `iRCCE_get_length(NULL, NULL)`. It can be used to determine the actual length after receiving a message while using the `iRCCE_ANY_SOURCE` wildcard within the receive call (see Section 6).

9.9 Functions for Handling Tagged Flags

int iRCCE_flag_alloc_tagged(RCCE_FLAG *flag)

Since up to RCCE V1.0.13 there is no difference between tagged flags and common flags, both types can be allocated and used with both APIs (RCCE in *gory* mode and iRCCE). However, when building iRCCE against a more recent RCCE version, this function must be used for allocating tagged flags in order to assure that a whole cache-line is assigned to such a flag.

`flag`	pointer to an existing variable of type `RCCE_FLAG`

int iRCCE_flag_write_tagged(RCCE_FLAG *flag, RCCE_FLAG_STATUS val, int ID, void *tag, int len)

This function can be used to change (= write) the status (= value) of an (i)RCCE flag to `RCCE_FLAG_SET` or `RCCE_FLAG_UNSET`. In contrast to the common `RCCE_flag_write()` function, this functions also transfers additional payload (up to 28 Byte) as a *tag* alongside with the flag value.

`flag`	pointer to the flag to be written
`val`	flag value to be written (`RCCE_FLAG_SET` or `RCCE_FLAG_UNSET`)
`ID`	rank of target UE
`tag`	pointer to payload to be sent alongside with the flag to target UE
`len`	length of the piggyback payload to be sent

int iRCCE_get_max_tagged_len()

This function returns the upper bound of the payload length that can be transferred via one call of `iRCCE_flag_write_tagged()`. Usually, the returned value is 28 Byte. However, if one wants to write portable iRCCE programs, it is a good idea to inquire for this value via this function before using tagged flags (see also Section 4.4).

int iRCCE_flag_read_tagged(RCCE_FLAG flag, RCCE_FLAG_STATUS *val, int ID, void *tag, int len)

This function reads and returns the current status of a certain flag and additionally copies potentially tagged payload (up to size `len`) into a receive buffer pointed to by `tag` (at least if this buffer pointer is not set to `NULL`).

`flag`	flag to be read
`val`	pointer to a status variable of type `RCCE_FLAG_STATUS` for storing the flag value
`ID`	rank of target UE
`tag`	buffer pointer for the payload to be received alongside with the flag from target UE
`len`	length of the piggyback payload to be received

int iRCCE_wait_tagged(RCCE_FLAG flag, RCCE_FLAG_STATUS val, void *tag, int len)

This function polls on a flag (by using the `iRCCE_flag_read_tagged()` function internally) until the flags value is equal to the passed parameter `val`. Thereupon is copies potentially tagged payload (up to size `len`) into a receive buffer pointed to by `tag` (at least if this buffer pointer is not set to `NULL`).

`flag`	flag on which should be polled
`val`	flag value to be waited for (`RCCE_FLAG_SET` or `RCCE_FLAG_UNSET`)
`tag`	pointer to payload to be received alongside with the flag
`len`	length of the piggyback payload to be received

int iRCCE_test_tagged(RCCE_FLAG flag, RCCE_FLAG_STATUS val, int *result, void *tag, int len)

This function is quite similar to `iRCCE_wait_tagged()` with the difference that it does not block. The returned result parameter indicates whether an alternate call of `iRCCE_wait_tagged()` would block (`*result=0`) or not (`*result=1`).

`flag`	flag which should be checked
`val`	flag value to be waited for (`RCCE_FLAG_SET` or `RCCE_FLAG_UNSET`)
`result`	returned integer flags that indicates whether a respective wait() call would block or not
`tag`	pointer to payload to be received alongside with the flag
`len`	length of the piggyback payload to be received

9.10 Functions for Handling Atomic Increment Registers

int iRCCE_atomic_alloc(iRCCE_AIR** reg)

The SCC's FPGA provides 96 *atomic increment registers* (AIRs) and before one of them can be used, it must be allocated by a call of this function. Please note that there is no corresponding free function and, therefore, a once allocated register cannot be freed during a still running iRCCE session.

`reg`	returned pointer to an atomic increment register (represented by a pointer of type `iRCCE_AIR`)

int iRCCE_atomic_inc(iRCCE_AIR* reg, int* value)

This function increments the value of an AIR register in an atomic manner and returns the value of the register *before* the respective incrementation took place.

`reg`	pointer to an atomic increment register (represented by a type `iRCCE_AIR`)
`value`	returned value of the AIR register *before* the incrementation took place

int iRCCE_atomic_read(iRCCE_AIR* reg, int* value)

This function reads and returns the current value of the respective AIR register without changing it.

`reg`	pointer to an atomic increment register (represented by a type of `iRCCE_AIR`)
`value`	returned value of the AIR register

int iRCCE_atomic_write(iRCCE_AIR* reg, int value)

This function can be used to initialize the counter reading of an AIR register to a certain value.

`reg`	pointer to an atomic increment register (represented by a type of `iRCCE_AIR`)
`value`	value of the AIR register to be written (e.g. for initializing the register)

9.11 Improved Collective Communication Functions

int iRCCE_bcast(char *buffer, size_t length, int root, RCCE_COMM comm)

This is an improved version (with respect to throughput performance) of the common broadcast function that makes internally use of iRCCE's *multicast* functions (see next Section 9.12). However, these improvements just take effect when the `RCCE_COMM_WORLD` communicator is used. Otherwise, this function just makes a fallback to a call of the common `RCCE_bcast()` function.

`buffer`	starting address of the message to be sent
`length`	length of the outgoing message in bytes
`root`	rank (ID) of the process (UE) that is the source of the message
`comm`	communicator whose UEs participate in the broadcast pattern

int iRCCE_barrier(RCCE_COMM *comm)

This is an improved barrier function (with respect to latencies) that internally makes use of the *atomic increment registers* (AIRs, see Section 4.5 and 9.10) if the `RCCE_COMM_WORLD` communicator is addressed. Otherwise, this function makes a fallback to the common `RCCE_barrier()` function.

`comm`	pointer to a communicator whose UEs participate in the barrier

9.12 Send and Receive Functions for Multicast

int iRCCE_msend(char *buffer, size_t length)

This function is to be called by the sender (the *root*) of a multicast pattern where one sender sends a message to all the other processes (UEs), which in turn have to call the respective `iRCCE_mrecv()` function. The main difference between these functions and a common broadcast function like `RCCE_bcast` is that wildcards like `iRCCE_ANY_SOURCE` or `iRCCE_ANY_LENGTH` are allowed on the receiver side (see Section 6.3).

`buffer`	starting address of the message to be sent
`length`	length of the outgoing message in bytes

int iRCCE_mrecv(char *buffer, size_t length, int source)

This function is to be called by all receivers of a multicast pattern where one sender, which has to call the respective `iRCCE_msend()` function, sends a message to all the other processes (UEs). he main difference between these functions and a common broadcast function like `RCCE_bcast` is that wildcards like `iRCCE_ANY_SOURCE` or `iRCCE_ANY_LENGTH` are allowed on the receiver side (see Section 6.3).

`buffer`	starting address of the receive buffer
`length`	length of the expected message in bytes
`source`	rank (ID) of the source/root process (UE)

Version History

- Version 1.0 / February 2011: Initial Release of iRCCE
- Version 1.1 / April 2011: Bug-Fix Release
- Version 1.2 / June 2011: Added *Wildcard* Support (`ANY_SOURCE`, `ANY_LENGTH`)
- Version 1.3 / November 2011: Non-official Beta-Release (`iRCCE_ssend()`, `iRCCE_srecv()`)
- Version 2.0 / March 2013: Added Support for *Tagged Flags* and *Atomic Increment Registers*

Acknowledgment

The research and development of the iRCCE library was supported by Intel Corporation. We would like to thank especially Ulrich Hoffmann, Michael Konow and Michael Riepen of Intel Braunschweig for their help and guidance as well as Carsten Scholtes of the University of Bayreuth for his useful feedback.

References

[1] Intel Many-core Applications Research Community. `http://communities.intel.com/community/marc`.

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

[3] Intel Corporation. *Intel MPI Benchmarks – Users Guide and Methodology Description*, 2006. Version 3.0.

[4] Intel Corporation. *SCC External Architecture Specification (EAS)*, July 2010. Revision 0.98.

[5] T. Mattson and R. van der Wijngaart. *RCCE: a Small Library for Many-Core Communication.* Intel Corporation, May 2010. Software 1.0-release.

[6] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC10)*, New Orleans, LA, USA, November 2010.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* High-Performance Computing Center Stuttgart (HLRS), September 2009. Version 2.2.

[8] P. Reble, S. Lankes, F. Zeitz, and T. Bemmerl. Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*, Berkeley, CA, USA, June 2012. Poster Paper.