A Lightweight and Resource-aware Socket Emulation Layer for the Intel SCC Processor

Carsten Clauss
carsten-clauss@web.de

Abstract. This paper presents rckSock: A communication layer for the Intel SCC manycore processor that utilizes the SCC's shared on-chip memory and that does not involve the operating system for emulating a Socket-like communication programming interface. By means of this layer, common client/server-based applications can easily be ported to the SCC in a lightweight and resource-aware manner while still exploiting the SCC's distinct on-chip communication capabilities.

1 Introduction

Socket-based TCP/IP communication is obviously the most predominant and thus portable way for inter-process communication. Although the Berkeley Socket API offers a quite generic communication interface that allows to be used for quite different communication protocols, only the IP-based transport protocols UDP and TCP are commonly utilized. However, both are networking protocols and are thus intended to be used for *inter*-node communication in local and/or wide area networks. Therefore, these protocols have some drawbacks when they are used for *intra*-node communication instead: One drawback is their quite high protocol stack that has even to be run through if a message is to be passed from one process to another one on the same node. So, for example, TCP and IP headers are added on the way down the stack, and on the other side checksums, sequence numbers and other header entries are to be analyzed and removed on the corresponding way up the stack. The other drawback is that typically the operating system (e.g. in terms of a loopback network driver) gets involved into the communication progress and this often increases the communication latencies significantly.

1.1 Intra-Node and On-Chip Communication

In order to overcome this issues, two common approaches exist: The first one is to use dedicated software facilities for local communication as offered by most operating systems, such as Pipes, FIFOs, Message Queues or UNIX Domain Sockets. The other one is to use Shared Memory for the inter-process communication. While the first one still involves the invocation of the operating system for message transport, the latter one demands for a session and transport management that in this case is to be implemented by the application programmer. Therefore, in general it can be stated that operating system based intra-node communication is easy to use but not quite lightweight, especially with respect to communication latencies, whereas message-passing via shared memory is lightweight but can be more tricky to use.

However, a third approach is to introduce a small emulation layer that features a Socket-like communication interface for the application. By facilitating shared memory for communication without any involvement of the operating system, such a layer can achieve both at the same time: providing a portable Socket-based communication interface while still being slim and lightweight for featuring low-latency intra-node communication.

In the upcoming manycore era, shared memory based message-passing is expected to become even more prevalent and it is most likely that such systems will introduce further levels into the hierarchy of inter-process communication, for example, by featuring explicitly addressable on-chip shared memory regions. According to this, the different levels of inter-process communication may be labeled as follows:

1. *Inter-Core* but *On-Chip* Communication via on-chip networks and on-chip memories like caches or *scratchpads*.
2. *Inter-Chip* but *Intra-Node* Communication via memory controllers and thus shared off-chip DRAM memories.
3. *Inter-Node* Communication via network interfaces and system-, local- and/or wide-area networks, respectively.

While the third level is already covered very well by portable Socket-based communication interfaces and respective networking protocols, Socket-like but lightweight shared memory communication is yet commonly not well supported.

1.2 The Intel SCC Manycore Processor

The SCC is a 48-core concept vehicle created by Intel Labs as a platform for manycore software research [1]. In contrast to common multi-core processors, the SCC does not provide any cache-coherency between the cores and hence makes message-passing the parallelization paradigm of choice. For doing so, each core provides 8 kByte of a fast on-chip memory as a kind of *scratchpad memory* that is also accessible to all other cores. Since these on-chip memories are intended to pass messages directly between the cores, they are referred to as *Message-Passing Buffers* (MPBs) – but actually they are just a fast on-chip shared memory. These MPBs are arranged together with the cores of the SCC in a 6x4 on-chip mesh network of *tiles*, with two cores and 16 kByte MPB space per tile. By means of this network, all cores can furthermore access a global shared memory of up to 64 GByte DDR3-DRAM memory via four on-chip memory controllers. Hence, the SCC is actually a prototype for such a hierarchical manycore architecture where the communication falls into one of the three above mentioned levels.

1.3 Related Work

In the domain of high-performance computing, besides the common Socket API, another portable and much more comprehensive communication interface exists

and is quite prevalent: The Message-Passing Interface (MPI). In fact, this interface standard [2] aims at being independent from the underlying communication layers, transport protocols and networking facilities. This is achieved by an interface design that is consistently hardware agnostic and that presents (and, if necessary, pretends) a flat and homogeneous communication environment to the application level. Nevertheless, an MPI library can very well be optimized also for heterogeneous and hierarchical environments. So, for example, an MPI library may facilitate quite different transport protocols and/or communication substrates – and this even within the same MPI session – so that at any time the fastest communication channel can be exploited. That way, MPI implementations commonly make use of shared memory for intra-node communication, whereas dedicated interconnects like InifiniBand are usually used for inter-node communication – and also Socket-based TCP/IP communication may be facilitated, e.g. as a fallback channel or for wide area communication.

Moreover, for manycore-related on-chip communication, MPI libraries may take this further hierarchical step into account and may likewise provide support for on-chip communication facilities like the SCC's Message-Passing Buffers. In fact, for the SCC two of such MPI libraries exist: RCKMPI by Intel [3] and SCC-MPICH [4], which has been developed by the author of this paper. However, since the MPI standard is quite comprehensive, MPI libraries can be quite heavyweight. Therefore, Intel also provides a slimmed down and accordingly lightweight library for the SCC, which just features the essential communication functions: the RCCE communication library [5]. In turn, this library, which can moreover be extended by the so-called iRCCE library [6], builds the base for SCC-MPICH and other higher-level communication layers for the SCC.

However, in contrast to the Socket interface, the common MPI-related communication model is not client-server based but *symmetric* and follows the *single-program/multiple-data* paradigm (SPMD) where all participating cores run the *same* executable. This is due to the flat communication model of MPI that does not reflect any hierarchies or other distinct communication relations within its addressing scheme inherently.

Another communication interface for the multicore era (and thus also for the manycore era) is the MCAPI standard [7] which has recently been specified by the Multicore Association. This standard, in contrast to MPI, uses an addressing scheme that is quite Socket-like and that allows for modeling communication hierarchies in a less artificial way than MPI. A prototype implementation of the MCAPI standard for the SCC has been developed by the author of this paper, too (see [8] for implementation details).

2 Motivation

When taking a closer look at the communication libraries mentioned in the prior related work section, it becomes apparent that these libraries are not resource-aware with respect to connection establishment – and thus to the management of the on-chip message-passing buffers as the actual communication resource of

the SCC. In order to motivate the development of rckSock, this section should expose this resource-related issue. However, since SCC-MPICH and the MCAPI prototype implementation for the SCC internally make use of RCCE/iRCCE, they adapt the communication model of RCCE, too. Therefore, this section especially takes a closer look at this model and its resource management.

2.1 Resource Allocation of RCCE

In the so-called *non-gory mode*, RCCE allocates all available MPB space right at session start during its initialization procedure.[1] In doing so, it uses each local MPB region as one single but big buffer for all outgoing messages – this is the so-called *local-put/remote-get* scheme (see [9] for a detailed analysis of this pattern). That means that all outgoing messages per process have to be passed through its local MPB and that at any time only one message can be located within this local buffer. However, due to the blocking semantics of RCCE's send and receive functions, anyway only one communication request can be pending per core at the same time. Therefore, this approach has the advantage that for each message the whole available local MPB space can be used and this in turn improves the achievable data rate especially for larger messages that have to be passed piecewise through the MPB.

2.2 The "Late Receiver" Issue

However, when using iRCCE's non-blocking functions (see [6] for a detailed discussion about blocking vs. non-blocking RCCE/iRCCE functions), the local MPB can become a bottleneck in terms of a communication buffer that congests if a receiver is currently not ready to take the respective message.[2] This is because also in the non-blocking case, only one (and this is the first) of all locally pending send requests can be processed by passing it through the local MPB and hence all other pending requests have to wait for its completion since a message reordering is not provided. That means that one late receiver can delay the message arrival for all the other communication partners significantly – moreover, in case of a faulty receiver that never takes the respective message, the application may even become stuck.

Figure 1 shows an exemplary scenario for such a case where a single sender (Core 0) sends messages to multiple receivers (Core 1 to 3) in a non-blocking manner. As one can see, due to the dependencies caused by the order of the posted requests, the computation function is not going to be called until *all* pending requests are actually finished.

One possibility to decouple the dependencies in such a *single-sender/multiple-receivers* case is to use likewise multiple communication buffers by means of dividing the local MPBs into multiple chunks. In fact, this is what the original version of RCKMPI does: It divides the local MPB space of each core into *np*

[1] Resource allocation in *gory mode* is to be detailed in the next section.
[2] That is what is commonly called the *Late Receiver* pattern, see [10].

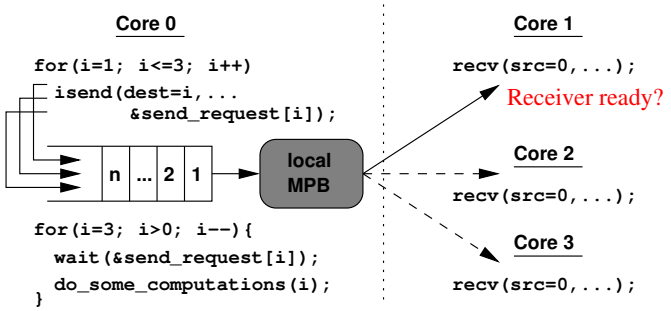

Fig. 1: *Single-Sender/Multiple-Receivers* Scenario

receive buffers for *incoming* messages from each of the remote cores (according to a *remote-put/local-get* pattern), where np is the number of processes started within the respective MPI session. This approach makes the resource allocation *dynamic* at least with respect to the fact that the number of chunks (and thus their size in a reciprocal manner) depends on the number of processes started.

2.3 Need for Application-dependent Resource Management

However, in many cases, the processes are far from communication directly with each other in a point-to-point manner. This is even true for collective communication patterns like broadcast or all-to-all because usually these patterns are performed according to in a hierarchical scheme (like a binary tree, for example), and not in a straightforward manner where each process sends a message directly to all the other processes. Hence, in many cases, not all of the $np \cdot (np-1)/2$ possible direct connections are actually needed. Moreover, allocating resources in terms of MPB space also for these unused connections right from the start decreases the achievable throughput per connection. This is because due to the limited MPB space, each further pair of possible connection partners decreases the MPB chunk size per connection and thus the gainable data rate.

Therefore, the challenge is to manage the MPB as the communication resource not only in a *dynamic* but also in an *application-dependent* manner so that the MPB allocation follows the actual communication pattern of the respective application (see Figure 2 for an exemplary scenario where the MPB and multiple sending queues are managed according to the communication pattern of the application). This is because only in this case a true decoupling of the communication pairs and their dependencies can be achieved while still retaining the maximal data rate per message and communication partner – and this is what the rckSock layer does, as it will be shown in the next section.

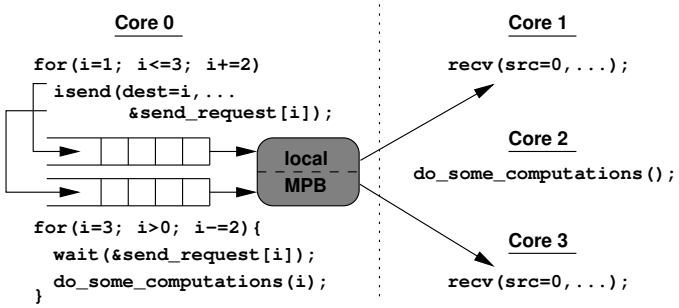

Fig. 2: Application-depended MPB Management

3 Implementation

As a Socket emulation layer, rckSock follows the client-server paradigm, where communication connections have to be established explicitly by calling respective *connect* and *accept* functions, which in turn helps to make the actual communication patters of the applications tangible. Moreover, as a resource-aware communication layer, rckSock manages each connection internally as a pair of MPB regions for passing messages in a bidirectional manner, while externally these connections are abstracted by communication handles: Namely by *Sockets*, which are actually just pointers to internal data structures that in turn contain pointers and information about the respective MPB regions.

In order to create a lightweight communication layer, rckSock is based on RCCE as likewise lightweight communication substrate. However, neither RCCE's *non-gory* nor its *gory* interface could be used directly. Instead, additional interface functions had to be implemented that accommodate more the client-server semantics than the SPMD-related RCCE functions do. This is because although the *gory* interface of RCCE allows for allocating and freeing MPB space in a dynamic manner (by calling `RCCE_malloc()` and `RCCE_free()`), RCCE uses a so-called *symmetric memory model* where these functions have *collective* semantics and hence must actually be called by *all* processes of a RCCE session.

Figure 3 shows an exemplary scenario where three processes collectively allocate a chunk of MPB space. As one can see, according to the symmetric memory model, each process allocates one local chunk respectively, resulting in three distributed ones that have to be addressed via the returned pointers plus source/destination IDs by means of RCCE's *put* and *get* functions.

3.1 Breaking with RCCE's Symmetric Memory Model

The first step towards the rckSock implementation was to add new send and receive functions, derived from those of the gory interface, that take pointers to the local MPB and to the local source or destination buffers, but that take *offsets* (instead of pointers) for dealing with remote MPB regions. The idea behind this

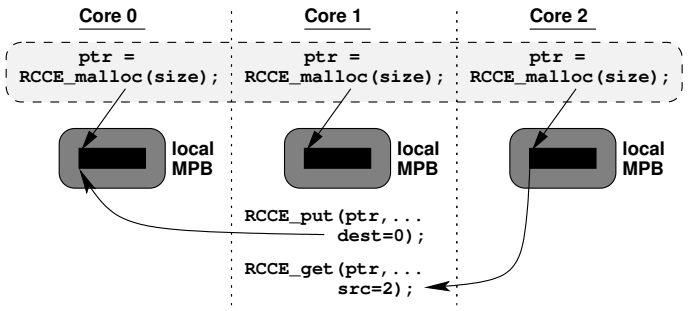

Fig. 3: Collective Memory Allocation via `RCCE_malloc()`

is that while pointers are only valid within the context of a process's own virtual address space, offsets (in relation to a virtual base address of a shared region) are valid in a system-wide manner. Because the shared but distributed MPB regions are all mapped into the virtual address spaces of all processes, pointers to sub-addresses and thus to sub-chunks of local and remote MPB regions can easily be determined and communicated by exchanging the respective offsets and just adding them to the locally valid base-pointers of the respectively addressed MPB regions. That way, the symmetric memory model can be broken and each process can manage its local MPB space (by allocating and freeing chunks of it) *independently* from one another. Moreover, by exchanging the respective chunk offsets, the processes are still able to use these chunks in a shared manner.

This should be illustrated by the following example (see Figure 4): Assume a server process that calls the `accept()` function and waits for clients to connect. When a willing client process calls the respective `connect()` function, it initially allocates a local MPB chunk X and connects to the server by telling it about the offset (`x_offset`) and the size (`x_size`) of this chunk in relation to the local base pointer. Upon this, the server allocates a likewise local MPB chunk Y and replies to the connection request by sending the information about offset (`y_offset`) and size (`y_size`) back to the client. Afterwards, both can communicate via these two MPB regions X and Y in a bidirectional manner and without the risk of interfering with other processes because these two regions are dedicated to this client-server pair and hence are to be used exclusively by these two processes. Due to the fact that each server can accept multiple connections from several clients and that each process can act as a server as well as a client, arbitrary communication relationships can be established and mapped to exclusively allocated MPB chunks – at least as there is enough MPB space left after each allocation for realizing further ones. However, at this point, two questions arise:

1. How can the offsets be communicated during the connect/accept procedure between client and server if there is yet no connection established?
2. How big should the size of the MPB chunks, that are to be allocated by client and server, be chosen?

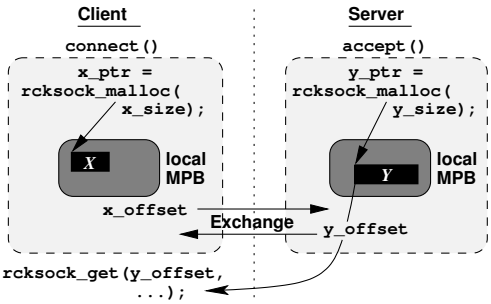

Fig. 4: rckSock's `connect()` and `accept()` functions

3.2 Communicating during Connection Establishment

The answer to the first question is that a minimalistic but already initially established communication channel between each possible pair of processes is needed. This can, for example, be conducted via the off-chip shared memory region or even via "regular" TCP/IP connections to be established by means of a kernel driver like *rckmb* (see [11] for details about the SCC's common TCP/IP drivers). However, rckSock instead uses the common communication facilities provided by RCCE as they are available when using the non-gory interface – but with the difference that not the whole available MPB space is used but only a quite small chunk of it:[3] At the very beginning of a session (that is during the `RCCE_init()` function), each process allocates such a local and usually quite small piece of MPB space dedicated for the later connection establishment. This chunk can then be used as in RCCE's common non-gory communication mode by multiplexing outgoing connection requests and replies to other processes through it. That way, the connection function actually becomes a `RCCE_send()` operation whereas the first part of the accept function is just a call of the matching `RCCE_recv()` function. The content of this connection request message can in turn be composed of the information about the offset of the later MPB communication buffer regarding the new Socket connection to be established and, for example, of additional Socket options. However, upon acceptance, the server has to answer to the request with a quite similar message containing the offset of its own freshly allocated MPB chunk and a reply to the inquired Socket options. Hence, the connection establishment becomes a two-way handshake based on the common send and receive functions of RCCE.

3.3 The Issue of an Optimal Chunk Size

The second question is not such easy to answer because the optimal sizes of the MPB chunks to be allocated for the Socket connections depend on the communication patterns of the respective applications. Therefor it is hard for the

[3] *"quite small"* means in the range of a few cache lines.

Socket emulation layer to choose an appropriate size without further information from the application layer. One approach to solution was to implement a special Socket option that enables the application programmer to specify the sizes of the MPB buffers to be used for the Socket communication explicitly. However, that in turn means that the application programmer has to modify the application according to the desired priorities concerning the handling of multiple Socket connections.

At this point it should be mentioned that porting Socket applications by means of rckSock usually requires a little code rework. This is because despite the above mentioned Socket options also the addressing scheme differs between the common TCP/IP-based usage and the ID-based addressing scheme of RCCE. For that reason, rckSock allows only for one "port" for each serve to listen on because the address to be used for connecting to other processes is just the core/process ID. However, multiple Socket connections between a pair of processes are very well possible and each process can act as server, client or both. Moreover, when using non-blocking Sockets, even loop-back connections to the same process are possible with rckSock.

3.4 Implementing a Dynamic Buddy System

However, the issue of choosing an optimal chunk size still exists and has led us to the following consideration: Why not using the whole still available MPB space for the communication unless further connections get requested? According to this new approach, the first Socket connection of a process would get the whole locally available MPB space (this is 8kByte minus the initially allocated chunks for the connection establishment[4]) assigned for communication until a further connection gets requested. In this case, the previously assigned MPB chunk of the first connection has to be divided (usually halved) so that from now on both connections can each use half of the initially available MPB space exclusively, and so on.

When taking a deeper look at this approach, this allocation scheme appears to be quite similar to that known from so-called *Buddy Systems*. According to this approach, the local MPB space is managed by each process in terms of chunks, the so-called *Buddies*, which in turn get sub-divided into two smaller chunks each time a new connection is requested – and looking at it the other way round, closing a connection accordingly leads to a merger of two Buddies reforming a larger one. This scheme is applied in a hierarchical manner so that the Buddies to be split upon a memory request and the Buddies to be merged upon a memory release are predefined. As the smallest reasonable MPB unit is a cache line, the smallest Buddy chunk possible with rckSock is 32 Byte, which in turn results in an upper bound of $log_2(8\text{ kByte} / 32\text{ Byte}) = log_2(256) = 8$ hierarchy levels. However, this parameter is configurable and hence bigger values for the smallest Buddy size (and thus fewer hierarchy levels) are implementable.

[4] minus some cache lines for MPB-based synchronization flags

As one can see, such a Buddy System makes the resource allocation quite flexible and allows for a dynamic mapping of the connection patterns onto the whole available MPB space. However, as creating new connections also impacts the sizes of already assigned MPB chunks, additional information exchange with respect to changes made to the allocation pattern becomes inevitable. In other words: If the size of a once allocated local MPB chunk gets split due to the establishment of a new Socket connection, the new chunk size needs to be communicated to the respective counterpart on the other end of the corresponding Socket. This could be done, for example, by signaling this event via messages to be sent via that MPB chunk that has already been used for handling the connect/accept procedure.

3.5 Exploiting the "Any Length" Wildcard

However, such an explicit signaling would cause an additional overhead that is to be avoided. Therefore, rckSock chooses another way by just passing this information (this is actually the current chunk size) alongside with the payload in a *piggy-back* fashion. For this purpose, the so-called `ANY-LENGTH` wildcard mechanism [12] of the iRCCE library gets exploited: This mechanism allows for receiving messages of *any length* (that means without stating the message length explicitly during an `iRCCE_recv()`-call) by passing the information about the payload size in terms of the word-sized flags that are needed anyway for synchronizing the send and receive progress. According to this approach, the rckSock's `send()` function splits each message to be sent into sub-chunks of sizes each not to exceed the local chunk size – and the `recv()` function of rckSock in turn posts as many `iRCCE_recv()`-requests with `ANY-LENGTH` as a wildcard for the actual message length until the complete message has been recomposed. In doing so, the bookkeeping about the MPB allocation scheme and about the chunk sizes is kept local – and since sending buffers are always the local ones, no further action needs to be taken concerning changes made to the allocation scheme.

4 Evaluation

In this brief section, some performance results should substantiate the motivations for the development of rckSock. For this purpose, a synthetic benchmark application is used that features one server process and multiple client processes (each, if needed, with multiple connections to the single server process). This benchmark runs with an increasing number of clients connecting to the server so that the available MPB space per connection at the server side gets increasingly reduced. The benchmark then measures the achievable throughput performance (maximal data rate) by means of a simple Ping-Pong benchmark between a single client-server pair while the other clients are just waiting.

In the scenario applied, rckSock was configured to use 8 cache lines (= 256 Byte) of MPB space per core for the initial connection establishment and additional 8 cache lines were reserved per core for the needed synchronization flags.

Hence, the initial available MPB space at the server side was $8192 - 512 = 7680$ Byte (= 240 cache lines). After each further incoming connection request, the available MPB gets divided and managed according to the Buddy System as described in Section 3.4.

buddy system hierarchy level	number of connections	*smallest* MPB chunk per connection	*maximal* throughput on this connection
1	1	7680 Byte / 240 CL	143.45 MB/s
2	2	3840 Byte / 120 CL	143.43 MB/s
3	3-4	1920 Byte / 60 CL	137.15 MB/s
4	7-8	960 Byte / 30 CL	119.42 MB/s
5	9-16	480 Byte / 15 CL	87.24 MB/s
6	17-32	224 Byte / 7 CL	53.84 MB/s
7	33-64	96 Byte / 3 CL	27.14 MB/s
8	65-240	32 Byte / 1 CL	9.64 MB/s

Table 1: Throughput Results of the Ping-Pong Benchmark (CL = Cache Lines)

As one can see, for small chunks of MPB space available per connection, the achievable throughput performance decreases dramatically. However, usually (and as assumed in the motivation section) one server and 47 clients do not need up to 240 connections at the same time. Therefore, managing the MPB space in a dynamic manner (as provided by the Buddy System presented in Section 3.4) helps to exploit the available MPB space according to the actual communication pattern in the most optimal way.

5 Conclusion and Outlook

Managing memory as a scarce resource according to Buddy Systems or similar approaches is actually long-know. However, as managing on-chip memories for message-passing will become more and more important in the upcoming many-core era, it is quite apparent to use these techniques also for managing such communication resources. In this paper, the rckSock library has been presented, which constitutes a lightweight and resource-aware communication layer for the Intel SCC as a prototype for future manycore processors.

In doing so, rckSock is lightweight with respect to a very thin protocol stack that avoids the involvement of the operating system. Furthermore, rckSock is resource-aware by means of a dynamic MPB management that helps to eliminate dependencies between multiple connections while still providing optimal data rates according to the communication patterns as given by the applications.

Although the rckSock layer is currently based on RCCE as the lower communication substrate, adopting this approach to other manycore-related communication interfaces seems to be quite obvious for future work. Although Intel will end support for the SCC in December 2013, other quite similar manycore architectures, that also may benefit from rckSock, are already at the ready.

References

1. 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.
2. *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, September 2012, Version 3.0. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
3. I. A. C. Ureña, M. Riepen, and M. Konow, "RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC)," in *Recent Advances in the Message Passing Interface – 18th European MPI Users' Group Meeting, EuroMPI*, Santorini, Greece, September 2011.
4. C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011), Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey, July 2011.
5. T. Mattson and R. van der Wijngaart, *RCCE: a Small Library for Many-Core Communication*, Intel Corporation, January 2011, Software 2.0-release. [Online]. Available: http://communities.intel.com/docs/DOC-5628
6. C. Clauss, S. Lankes, P. Reble, J. Galowicz, S. Pickartz, and T. Bemmerl, *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – Users' Guide and API Manual*, March 2013, Version 2.0 iRCCE FLAIR. [Online]. Available: http://www.lfbs.rwth-aachen.de/publications/files/iRCCE_FLAIR.pdf
7. *Multicore Communications API (MCAPI) Specification*, Multicore Association, March 2011, Version 2.015. [Online]. Available: http://www.multicore-association.org/workgroup/mcapi.php
8. C. Clauss, S. Pickartz, S. Lankes, and T. Bemmerl, "Towards a Multicore Communications API Implementation (MCAPI) for the Intel Single-Chip Cloud Computer (SCC)," in *Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*, Munich, Germany, June 2012.
9. P. Reble, C. Clauss, M. Riepen, S. Lankes, and T. Bemmerl, "Connecting the Cloud: Transparent and Flexible Communication for a Cluster of Intel SCCs," in *Proceedings of the Many-core Applications Research Community (MARC) Symposium*, Aachen, Germany, November 2012. [Online]. Available: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2012/4383/pdf/4383.pdf
10. F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi, "Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications," in *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, Stuttgart, Germany, July 2008.
11. R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on Intel's Single-chip Cloud Computer processor," *SIGOPS Operating Systems Review*, vol. 45, pp. 73–83, 2011.
12. C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Recent Advances and Future Prospects in iRCCE and SCC-MPICH," in *Proceedings of the 3rd Symposium of the Many-core Applications Research Community (MARC)*, Ettlingen, Germany, July 2011.