Recent Advances and Future Prospects in iRCCE and SCC-MPICH
— Poster Abstract —

Carsten Clauss, Stefan Lankes, Pablo Reble, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{*clauss,lankes,reble,bemmerl*}*@lfbs.rwth-aachen.de*

***Abstract*—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. Intel provides a customized programming library for the SCC, called RCCE [5], that allows for fast message-passing between the cores. For that purpose, RCCE offers an application programming interface (API) with a semantics that is derived from the well-established MPI standard [7]. However, while the MPI standard offers a very broad range of functions, the RCCE API is consciously kept small [6] and far from implementing all the features of the MPI standard. So, for example, RCCE only provides *blocking* (often also referred to as *synchronous*) send and receive functions, whereas the MPI standard also defines the semantics of *non-blocking* communication functions. For this reason, we have started to extend RCCE by new communication capabilities, as for example by the ability to pass messages *asynchronously*. In doing so, we aim to avoid interfering with the original RCCE library and therefore we have placed our extensions and improvements into an additional library called iRCCE [2]. Moreover, this additional library in turn serves us as low-level communication layer for SCC-MPICH, that is an SCC-customized and full MPI-1 compliant MPI library. In this contribution, we present the recent advances and future prospects for both these SCC-related communication libraries: iRCCE and SCC-MPICH.**

***Keywords*—Many-core, Message-Passing, SCC, RCCE, MPI**

I. iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library

Due to the lack of non-blocking communication functions within the current RCCE library, we have started to extend RCCE by such asynchronous communication capabilities (`iRCCE_isend`/`iRCCE_irecv`). In doing so, we aim to avoid interfering with the original RCCE functions and therefore we have placed our extensions into an additional library with a separated namespace called iRCCE. An obvious way to realize non-blocking communication functions would be to use an additional thread that processes the communication in background. Although this approach seems to be quite convenient, it is not applicable in *bare-metal* environments where a program runs without any operating system and thread support. And since RCCE has been designed to support also such bare-metal environments, we had to waive this thread-based approach for realizing non-blocking functions. Therefore, we have followed another approach where the application must drive on the communication progress by itself. For this purpose, the non-blocking communication functions return *request handles* which can then be used by the application to trigger the progress by means of additional *push*, *test* or *wait* functions (`iRCCE_push`, `iRCCE_test`, `iRCCE_wait`). [2]

A recent improvement of iRCCE is the feature that one can use a wildcard (`iRCCE_ANY_SOURCE`) instead of a definite source rank when calling the receive function. That means that this wildcard can be used to receive *any* incoming message regardless from its actual sender. However, the application programmer still has to ensure that at least the stated message length matches between receiver and sender.

Currently, we are developing a mailbox system on top of iRCCE that can be used to exchange small (cache-line-sized) datagrams between the cores. Since this mailbox system works without interference with the common send and receive functions, it can be used to pass additional signaling information alongside with normal RCCE/iRCCE messages. Therefore, such a mailbox datagram is well structured in terms of data items that are quite similar to that of message headers: `source`, `size`, `tag` and embedded `payload`.

Our aim is to use this mailbox system to extend the current semantics of the send and receive functions. So, for example, we plan to introduce a further wildcard mechanism also for the message length (`iRCCE_ANY_LENGTH`). That means that the information about the actual message size has then just to be provided by the sender, while the receiver merely has to ensure that the receive buffer is large enough to store the message. Moreover, by introducing additional message *tags*, as known from the MPI standard, even a message prioritization and reordering by means of these tags would become possible.

For this purpose, a sender would initially post a mailbox datagram to the respective receiver, indicating that a payload message of a certain size and with a certain tag will follow. Therefore, the local mailbox on the receiver side needs to be checked frequently in order to detect such incoming messages. However, it is entirely possible that the receiver detects a message that is yet still unexpected. This is for example the case when the message tags on sender and receiver side do not yet match and thus a message reordering becomes necessary. In such a case, the receiver can either copy the incoming message into a temporary buffer or the receiving of the actual payload must be delayed by sending a corresponding response datagram. The choice for one of these two approaches depends on the message size: for short and midsize messages, a temporary buffering seems to be acceptable, whereas long messages should be delayed because the additional copy procedure would impact the communication performance. Besides this, very small messages could be embedded *into* a datagram, so that there is no need for an additional payload message in such a case.

II. SCC-MPICH: Yet Another MPI-compliant Message-Passing Library for the Intel SCC

Although the semantics of RCCE's communication functions are obviously derived from the MPI standard, the RCCE API is far from implementing all MPI-related features. And even though iRCCE extends the range of supported functions (and thus the provided communication semantics), a lot of users are familiar with MPI and hence want to use its well-known functions also on the SCC. A very simple way to use MPI functions on the SCC is just to port an existing TCP/IP-capable MPI library to this new target platform. However, since the TCP/IP driver of the Linux operating system image for the SCC does not utilize the fast on-die message-passing buffers (MPBs), the achievable communication performance of such a ported TCP/IP-based MPI library lags far behind the MPB-based communication performance of RCCE and iRCCE.

For this reason, we have started in the last year to implement an SCC-optimized MPI library, called SCC-MPICH, which in turn is based upon our iRCCE extensions of the original RCCE communication library. At about the same time, Intel also started to implement an SCC-customized MPI library, called RCKMPI [8]. While RCKMPI has already been released by Intel, we have not yet published SCC-MPICH despite the fact that it is already fully operational, too. The reason for this is that we think that our human resources are too limited to provide sufficient user support for this project in case of an official software release. However, we use SCC-MPICH as basis for our future message-passing related research on the SCC and we have launched several student projects that in turn are also based on SCC-MPICH.

A major advantage of SCC-MPICH compared to RCKMPI is that it can be installed and used as easy as RCCE. That means that one can use the `mpirun` script just instead of the known `rccerun` directly from the Management Console PC (MCPC) without installing any additional libraries or startup environments on the SCC cores. Moreover, even the cores of the MCPC can easily be involved into an SCC-MPICH session. That means that one can start x MPI processes on the SCC cores and additionally y MPI processes on the cores of the MCPC.

In doing so, SCC-MPICH does not use just TCP/IP (as the lowest common dominator) for all the communication, but rather offers *hierarchy-awareness* in such a way that always the fastest communication mode is being used. That means that MPI processes running on the SCC cores use the message-passing buffers (MPBs) to communicate among each other, while processes running on the MCPC communicate via shared memory. The communication between the MCPC processes and the SCC cores is then eventually conducted via TCP/IP. In order to start such a mixed MPI session, one just needs to issue `mpirun -nue x -mcpc y` in a console on the MCPC.

A further advantage of SCC-MPICH is that it offers SCC-optimized collective communication routines. This is because SCC-MPICH is directly based upon RCCE (with iRCCE extensions) and due to the fact that RCCE can in turn be extended by the customized collective functions of the so-called *RCCE_comm* library [1]. That way, an easy mapping of MPI collective function calls onto the optimized RCCE_comm functions becomes possible

A more detailed description of SCC-MPICH together with some performance results can be found in [3].

References

[1] Ernie Chan. *RCCE_comm: a Collective Library for the Intel Single-chip Cloud Computer*. Intel Corporation, September 2010.

[2] C. Clauss, S. Lankes, J. Galowicz, and T. Bemmerl. *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*. Chair for Operating Systems, RWTH Aachen University, December 2010. Users' Guide and API Manual.

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

[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] Isaias A. Compres Urena. *RCKMPI User Manual*. Intel Braunschweig, January 2011.