MPI communication on MPPA Many-core NoC: design, modeling and performance issues

Minh-Quan Ho, Bernard Tourancheau, Christian Obrecht, Benoît Dupont de Dinechin, Jérôme Reybert

MPI communication on MPPA Many-core NoC: design, modeling and performance issues

Minh Quan HO *, Bernard TOURANCHEAU *, Christian OBRECHT †,
Benoît Dupont de DINECHIN ‡, Jérôme REYBERT ‡
*Grenoble Informatics Laboratory (LIG) - University Joseph Fourier Grenoble, France
{minh-quan.ho, bernard.tourancheau}@imag.fr
†INSA-Lyon, CETHIL UMR 5008 Villeurbanne, France
christian.obrecht@insa-lyon.fr
‡Kalray Inc. S.A. - Montbonnot, France
{benoit.dinechin, jerome.reybert}@kalray.eu

***Abstract*—Power dissipation and energy consumption has become a major issue for high performance computing and embedded systems. Keeping up with the performance trend of the last decades cannot be achieved anymore by stepping up the clock speed of processors. The usual strategy is nowadays to use lower frequency and to increase the number of cores. On such recent systems, data communication and memory bandwidth can become the main barrier, since there are more and more processing units to coordinate. In this paper, we introduce an MPI design and its implementation on the MPPA-256 (Multi Purpose Processor Array) processor from Kalray Inc., one of the first worldwide actors in the many-core architecture field. A model was developed to evaluate the communication performance and bottlenecks on MPPA. Our achieved result of 1.2 GB/s, e.g. 75% of peak throughput, for on-chip communication shows that the MPPA is a promising architecture for next-generation HPC systems, with its high performance-to-power ratio and high-bandwidth network-on-chip. However, the lack of a globally addressable memory on this distributed-memory architecture still requires the developer to take care of cache coherence and to pay attention to the limited local memory space of each compute element.**

***Keywords*—Many-core, NUMA, Distributed memory, Network-on-Chip, MPI, Performance modeling, Linpack, HPL, MPPA.**

INTRODUCTION

In this paper, we propose the design of an MPI Message-Passing library [1] for the intra communication on many-core processors, using the vendor support library (MPPAIPC [2]) as the transfer-fabric to build MPI protocols from scratch, while porting any of existing MPI implementations such as MPICH or OpenMPI would not be possible due to limited on-chip memory of most recent many-core processors.

Based on studied MPPA hardware specifications presented in [3], [2], this paper does a brief hardware summary and focuses on an MPI design over (but not limited to) the MPPA architecture, with detailed implementation algorithms and formulated models following vendor-hardware characteristics (K, h) and different optimizing approaches (Lazy, Eager). These studies is generic enough to be compared/ported to other very-similar architectures, such as Tilera [4], STHORM [5] or Neo chip [6], on which doing/optimizing MPI communication over Network-on-chip is still a challenging or never-posed question.

The remainder of this paper is organized as follows: Section I briefly introduces the MPPA architecture and the MPPAIPC components used in our implementation. Section II describes our MPI architecture design. Section III resumes our MPI implementation in pseudo-codes of blocking and non-blocking communication (`MPI_Send` and `MPI_Isend`). Some optimization ideas are then proposed and developed in this section such as (1) synchronization-free "eager send" and (2) implicit local-buffered "lazy send" for short and medium sized messages respectively. A throughput estimation model based on the data transmission time is also introduced in section IV to evaluate the communication performance. Section V presents our results for the ping-pong test following two scenarios, either symmetric ranks (MPI compute node - MPI compute node) or asymmetric ranks (MPI compute node - MPI I/O), corresponding on MPPA to CC-CC and CC-I/O subsystem respectively. Different optimization approaches are also tested and compared.

Using the MPPA-MPI library, the HPL benchmark [7] [8] was ported on MPPA with the support of the standard BLAS-Netlib [9] [10] (mono-threaded) and OpenBLAS [11], an OpenMP optimized implementation. These benchmark results are summarized in section VI. Related MPI-oriented works on other many-core platforms will also be compared in section VII and conclusions are given in section VIII.

I. MPPA-256 ANDEY HARDWARE AND SOFTWARE

A. Architecture overview

The MPPA-256 Andey [3] embeds 256 VLIW compute cores grouped into 16 compute clusters (CC) and four I/O subsystems (IOS), with theoretical performance of 230 GFlops (in single-precision) and 70 GFlops (in double-precision) and an energy consumption of 10 W.

1) Compute Clusters (CC): Each compute cluster operates 2MB of banked parallel memory shared by 17 VLIW cores running at 400 MHz. These cores are divided into 16 user cores referred to as Processing Elements (PEs) and one system-reserved core known as Resource Manager (RM). Each PE and RM are fitted with private 2-way associative 8KB instruction cache and 8KB data cache. The shared memory owns one Rx and one Tx NoC interface, paired with a 8-channel DMA engine and a Debug Support Unit (DSU). An MPPA-256

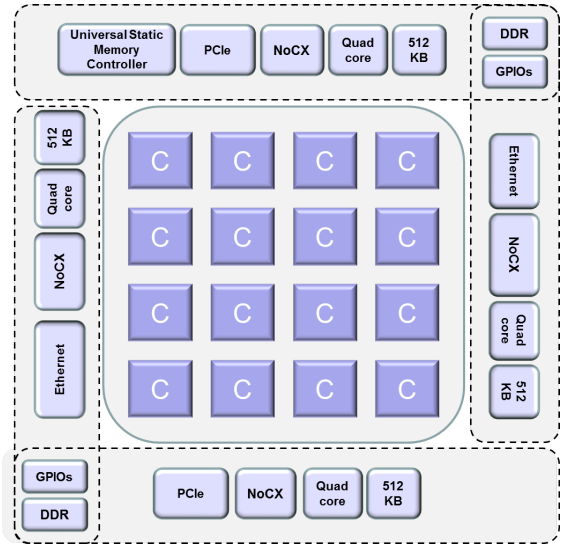


Fig. 1: MPPA-256 processor overview.

Source: Kalray Architecture documentation.

processor amounts to a total 32MB of memory distributed over the 16 CCs.

2) I/O Subsystems (IOS): The four I/O subsystems (North, South, East, West), each containing four cores, integrate DRAM controllers managing the off-chip DDR memory, a 8-lane Gen3 PCI-Express and Ethernet links, as well as an Interlaken link intended to extend the NoC across multiple MPPA-256 chips, and other I/O devices.

3) Network-on-Chip: CCs and IOS are connected by two network-on-chip. The both Data NoC (D-NoC) and Control NoC (C-NoC) [2] ensure reliable delivery (lossless) thanks to the credit-based flow control mechanism [12] and FIFO packet arrival using static NoC routing [13]. No acknowledgment is needed at the packet reception. As a result, there is no need to consider a TCP layer implementation when building any communication library above these NoCs (e.g MPI).

B. MPPA Inter-Process-Communication (MPPAIPC)

One of several programming models currently available on the MPPA platform that matches our working scope in this paper is MPPAIPC - an MPPA-specific POSIX-compliant library [2] following the distributed memory model, where communication and synchronization between CCs and IOS are explicitly done by developer. These actions are achieved using IPC primitives through connecting objects listed in Tab. I.

Sync object provides light-weight and low-latency synchronization barriers by exchanging 64-bits messages on CNoC. This connector can be used to implement two types of barriers:

- Master-Slaves (1 : M) between an I/O subsystem and several (or all) compute clusters in case of sequential process splitting
- All-to-all (M : M) barrier between compute clusters, similar to `MPI_Barrier()`

Portal object supports data transfer on DNoC using one-sided communication with zero-copy transfer. The sender (Tx) can write to the receiver's buffer (Rx) via a known `dnoc_tag` number, with an optional offset. This can be used to implement N-to-1-data-collection by many distinct writes from one (or several) Tx process(es) to the same Rx buffer, and at different offsets (see Fig. 2).

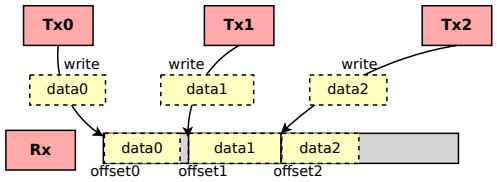


Fig. 2: Data collection on portal

RQueue object implements FIFO 120-byte-message queues with user-defined receive buffer length ($120 \times nb\_slot$). The inbound flow is controlled by a credit mechanism that ensures available buffer space to store incoming messages before being handled by the Rx rank. Callback function on message arrival can be defined to implement an Active Message Server [14].

Channel object proposes point-to-point communication link with rendez-vous behavior. The Rx rank sends its buffer size to the Tx rank in a CNoC packet. The Tx rank then reads the buffer size and sends data through DNoc on a known `dnoc_tag`.

Both *Portal* and *Channel* objects contain primitives for data transfer. Otherwise, the current implementation of *Portal* provides higher performance and is more feature-rich than the *Channel* one. *Portal* integrates one-sided communication in either unicast or multicast, supports both blocking and non-blocking data sending (Tab. II). Meanwhile, *Channel*, at the time of this writing, provides only rendez-vous blocking sends. Indeed, the *Portal* object was used to implement the transfer layer of our MPPA-MPI model. An *RQueue*-message active server is also an important module as well.

Mode	Functions
Blocking send	`mppa_pwrite()` `mppa_pwrites()`
Non-blocking send	`mppa_aio_write()` (DMA engine)
Receive	`mppa_aio_read()`
Termination	`mppa_aio_wait()`

TABLE II: Portal connector primitives associating to send/receive modes

II. MPPA-MPI DESIGN

In the MPPA context, each CC is referred to as an MPI rank. Thus, the MPPA-256 processor supports up to 16 MPI ranks. Each MPI rank owns a private memory space of 2MB. Moreover, a hybrid MPI I/O rank is introduced running on the North IOS and manages the off-chip DDR memory. This MPI I/O rank is started from the host via the `k1-mpirun` command and is responsible for spawning MPI

Type	Pathname	Tx:Rx	`aio_sigevent.sigev_notify`
Sync	`/mppa/sync/`*rx_nodes:cnoc_tag*	N : M	
Portal	`/mppa/portal/`*rx_nodes:dnoc_tag*	N : M	`SIGEV_NONE, SIGEV_CALLBACK`
RQueue	`/mppa/rqueue/`*rx_node:dnoc_tag/tx_nodes:cnoc_tag/credits.msize*	N : 1	`SIGEV_NONE, SIGEV_CALLBACK`
Channel	`/mppa/channel/`*rx_node:dnoc_tag/tx_node:cnoc_tag*	1 : 1	`SIGEV_NONE`

TABLE I: NoC connector pathnames, signature, and asynchronous I/O `sigevent` notify actions

compute ranks on CCs subsequently. To keep the portability of any MPI legacy code, this extra MPI I/O rank is not listed in the `MPI_COMM_WORLD`. Any communication with this rank can be achieved through a "local communicator" (`MPI_COMM_LOCAL`) that groups all MPI ranks within an MPPA processor (i.e 17 ranks). The MPPA-MPI architecture on each rank is then divided in two layers:

1) MPI-inter-process Control (MPIC): Each MPI transaction begins by exchanging control messages at the MPIC layer between MPI ranks. Control messages are used for:

- exchanging information about MPI transaction type (send/receive, communicator split, etc.).
- synchronization point in case of rendez-vous protocol.

We implemented an RQueue-based active message server [14] on each MPI rank (CC and IOS) to handle incoming control messages from all other ranks (including itself on loop-back). Upon control message arrival, a callback function is executed on the RM, consisting typically on saving it into an internal buffer which later will be read by MPI calls from the `main` function (PE0).

Control messages exchanged in the MPIC layer contain either one of the structures defined in Fig. 3.

```
/* Message sent by Tx to Rx (Request-To-Send) */
typedef struct send_post_s {
  mppa_pid_t    sender_id; /* ID of Tx process */
  int           mpi_tag;   /* MPI message tag */
  ...
} send_post_t;
/* Message sent by Rx to Tx (Clear-To-Send) */
typedef struct recv_post_s {
  int           dnoc_tag;  /* DNoc allocated on Rx */
  mppa_pid_t    reader_id; /* ID of Rx process */
  int           mpi_tag;   /* MPI message tag */
  ...
} recv_post_t;
```

Fig. 3: Control message structures

In an MPI send/receive, The Tx rank posts a *Request-To-Send* (`send_post_t`) to the Rx rank; idem, the Rx rank sends back a *Clear-To-Send* (`recv_post_t`) containing its allocated `dnoc_tag`, to which the Tx rank will send data. Beforehand, this `dnoc_tag` needs to be configured and linked to the receive buffer to enable remote writing.

2) MPI-inter-process Data-Transfer (MPIDT): is a light-weight wrapper of MPPAIPC *Portal* primitives. Once the Tx rank has got a matching control message, it configures a data transfer using received information (e.g. `dnoc_tag`). Data can then be sent in either blocking or non-blocking mode dependent on the calling MPI function, using appropriate *Portal* primitives (see Tab. III for detailed function mapping).

MPPA-MPI	MPPAIPC Portal
`MPI_Send`, `MPI_Ssend`	`mppa_pwrite`, `mppa_pwrites`
`MPI_Isend`, `MPI_Issend`	`mppa_aio_write`
`MPI_Recv`, `MPI_Irecv`	`mppa_aio_read`
`MPI_Wait`	`mppa_aio_wait`

TABLE III: MPI send/receive implementation in MPIDT level

Fig. 4 illustrates the structure of our MPPA-MPI implementation. Each rank emits control-message(s) to its involved partner(s) at each MPI call. The active server runs on the RM core and processes incoming control-messages. Furthermore, depending on MPI transactions and their status at runtime, the server can decide whether to perform a data send if this has not been or could not be done by the `main` thread, especially in case of a pending `MPI_Isend` request or a matching registered "lazy" message.

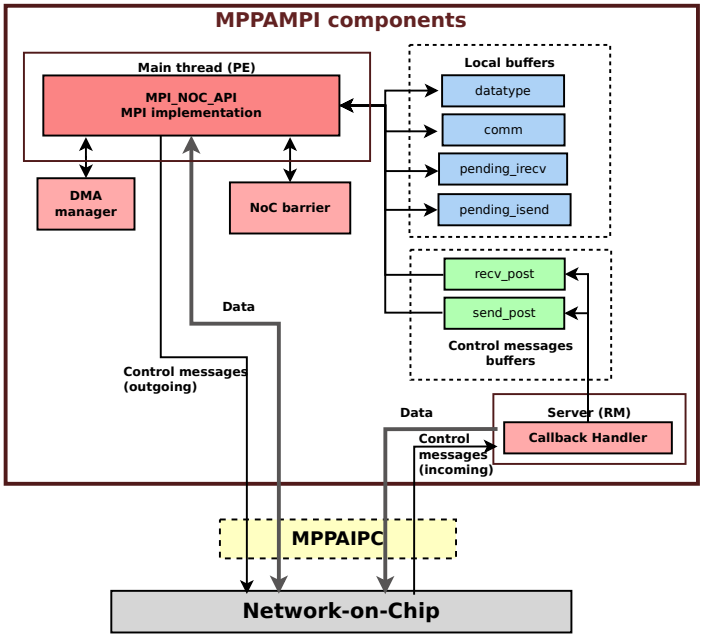


Fig. 4: MPPA-MPI components and interaction with Network-on-chip through MPPAIPC.

III. MPPA-MPI IMPLEMENTATION

As mentioned above, on-flight control messages carry essential information depending on their purpose. We present now their usage as well as algorithms of the two communication scenarios in our work:
(1) synchronous blocking send (`MPI_Send`) and
(2) asynchronous non-blocking send (`MPI_Isend`)

A. MPI_Send - MPI_Recv

Most well-known and optimized MPI libraries contain many (combined) techniques to perform the `MPI_Send` call. In the first time, we chose to implement this function with rendez-vous blocking behavior, in order to avoid extra buffer space and minimize memory usage. This choice certainly adds more synchronization cost but does not change the functionality of the send/receive transaction. Some optimization approaches will be presented in the coming sections. Algorithms 1 and 2 summarize the implementation of `MPI_Send` and `MPI_Recv`.

Algorithm 1 MPI_Recv(buf, count, datatype, source, tag, comm, status)

```
1: my_rank ← get_rank(comm);
2: dnoc_tag ← allocate_dnoc_tag();
3: /* configure to receive data on this dnoc_tag */
4: aio_request ← configure_aio_read(buf, dnoc_tag, ...);
5: if source == MPI_ANY_SOURCE then
6:     send_post ← find_send_post(count, datatype, tag, ...);
7:     real_source ← send_post.source;
8: else
9:     real_source ← source;
10: end if
11: /* send recv_post to real_source (MPIC layer) */
12: send_recv_post(my_rank, dnoc_tag, real_source, tag, comm, ...);
13: /* wait data (MPIDT layer) */
14: mppa_aio_wait(aio_request);
15: return MPI_SUCCESS;
```

Algorithm 2 MPI_Send(buf, count, datatype, dest, tag, comm)

```
1: my_rank ← get_rank(comm);
2: /* send send_post to dest */
3: send_send_post(my_rank, dest, tag, comm, ...);
4: /* wait for matching recv_post from Rx (MPIC layer) */
5: repeat
6:     recv_post ← find_recv_post(count, datatype, tag, ...);
7: until recv_post ≠ NULL
8: /* send data (MPIDT layer), using mppa_pwrite(s) */
9: send_data(buf, count, datatype, recv_post.dnoc_tag);
10: return MPI_SUCCESS;
```

The implementation of `MPI_Irecv` is the same as the one of `MPI_Recv`, except that the function returns right after having posted the receive to the sender, and the completion of reading (`mppa_aio_wait`) is done in `MPI_Wait`.

B. MPI_Isend - MPI_Recv

The implementation of `MPI_Isend` uses non-blocking *Portal* primitives on both PE and RM on the Tx side. When the Tx rank (PE0) reaches `MPI_Isend` in its execution *without* having received any matching `recv_post`, it creates a "non-started" `pending_isend` request containing related information (buffer pointer, dest, count, tag etc.) and returns. On arrival of the matching `recv_post`, the RM core (callback handler) reads the previous `pending_isend` request and triggers a non-blocking data send (to the `recv_post.dnoc_tag` of the Rx rank). The request is then set to "started" state to be distinguished from other "non-started" requests.

On the other hand, when the `recv_post` arrives before `MPI_Isend`, the RM core saves it into the internal buffer. The PE core executing `MPI_Isend` later reads this `recv_post`, performs a non-blocking send and marks the `pending_isend` request as "started". This propriety ensures that the transfer is performed only once for each transaction, either by the PE core (in `MPI_Isend`) or by the RM core (in callback handler). At the end, "started" requests will be finished and cleaned by `MPI_Wait`. Algorithms 3 and 4 present in more details the implementation of `MPI_Isend` and of the callback handler.

Algorithm 3 MPI_Isend(buf, count, datatype, dest, tag, comm, request)

```
1: my_rank ← get_rank(comm);
2: /* send send_post to dest */
3: send_send_post(my_rank, dest, tag, comm, ...);
4: req ← new_request(buf, count, ..., PENDING_ISEND);
5: /* look for a matching recv_post (MPIC layer) */
6: recv_post ← find_recv_post(count, datatype, tag, ...);
7: if recv_post ≠ NULL then
8:     /* configure/start a non-blocking write (MPIDT layer) */
9:     aio_request ← configure_aio_write (buf, recv_post.dnoc_tag, ...);
10:    req→status := STARTED;
11:    req→aio_request := aio_request;
12: else
13:    /* Do nothing (request initialized NON_STARTED) */
14: end if
15: request ← req;
16: return MPI_SUCCESS;
```

Algorithm 4 callback_recv_post(recv_post)

```
1: /* look for a matching pending_isend */
2: req ← find_pending_isend(recv_post);
3: if req ≠ NULL then
4:     /* configure/start a non-blocking write (MPIDT layer) */
5:     aio_request ← configure_aio_write (req→buf, recv_post.dnoc_tag, ...);
6:     req→status := STARTED;
7:     req→aio_request := aio_request;
8: else
9:     save_recv_post(recv_post);
10: end if
11: return ;
```

C. Optimization

1) Eager send optimization: Our idea is to pack any MPI message which can fit into a 120-byte space, as a control-message and send it directly to the Rx's active server. In reality, the maximum data payload is about 96 bytes (24 bytes is used for control header). An `eager_buffer` needs to be allocated on each MPI rank and can be defined by the `EAGER_BUFFER_LENGTH` macro in `main.c`. This approach is synchronization-free when the `MPI_Send` call can return before a matching receive is posted (non-local). It also leads to an improvement of about 6 to 10% in performance for the HPL benchmark on MPPA (see Section VI).

For longer messages, using several "eager sends" introduces segmentation and reassembly costs. We implemented a test case where messages are splitted into "eager" pieces in order to determine the best communication trade-off in Fig. 6. Such segmentation however consumes memory for buffers and therefore puts more pressure on RM's resources and limits its usage in practice.

2) Lazy send optimization: Lazy send consists in copying medium-size message into a local buffer and returns. The RM is then responsible for sending it to the destination. Unlike `eager_buffer` on the Rx side, `lazy_buffer` is allocated on the Tx side and can be tuned via some macros (`LAZY_THRESHOLD`, `LAZY_BUFFER_LENGTH`).

This approach must be used with care because bad communication scheduling may lead to buffer wasting and lazy messages remaining for too long. Inversely, a dense communication scheme should neither be set to "lazy" mode in order to be able to send data directly rather than spending time doing `memcpy` in local memory.

3) DMA thread usage: `MPI_Isend` uses *Portal* non-blocking primitive to configure a Tx DMA thread for data sending. The DMA engine implements a `fetch` instruction that loads the next cache line while pushing the current line into the NoC. This `fetch` is nowadays not available on PE cores, meaning that outbound throughput using PE is 4 times lower than using DMA engine (1 B/cycle vs. 4 B/cycle). Thus, tuning to use non-blocking DMA on `MPI_Send` for messages of size greater or equal to `DMA_THRESHOLD` will maximize the transfer performance.

IV. MPPA-MPI Throughput modeling

The MPPA-256 Network-on-chip [15] is designed so that any path linking two CCs always contains less than eight hops (including two local hops - one at sender and one at receiver). The average switching time on a NoC router is 7 cycles, then it takes the packet at most 8 cycles to reach the next hop. In the worst case, the link distance (time a packet spends on NoC to reach its destination) is 112 cycles ($7 \times 8 + 8 \times (8-1)$). However, the necessary time to send a buffer (transmission time - t) is about $O(N)$ cycles [16] (where N the buffer size in bytes), which is much longer than the link distance [17].

As a result, we describe the transmission time t as a function of the buffer size N, a constant transfer ratio K and a default overhead h (aka. the cost of sending an empty buffer). This default overhead presents the initial cost of MPI implementation management (ID mapping, metadata setup, synchronization, error checking ...) and/or configuring the peripherals (cache, DMA) to prepare for data sending. This cost is paid on each MPI call and is independant to the subsequent data-sending process (which is presented by a data-transfer factor K). The ping-pong round-trip time (RTT) is approximately the sum of the transmission time on both sides, as the propagation time is negligible.

$$TransmissionTime: \quad t = K \times N + h \ (cycles) \tag{1}$$

$$RTT \simeq 2 \times t = 2 \times (K \times N + h) \ (cycles) \tag{2}$$

$$Throughput: T = \frac{2 \times N}{RTT} \simeq \frac{N}{K \times N + h} = \frac{1}{K + \frac{h}{N}} \ (bytes/cycle) \tag{3}$$

$$\begin{aligned} \lim_{N \to \infty} T \simeq \lim_{N \to \infty} \frac{1}{K + \frac{h}{N}} &= \frac{1}{K} \ (bytes/cycle) \\ &= 400 \times K^{-1} \ (MB/s) \\ &(at \ frequency \ 400 \ MHz) \end{aligned} \tag{4}$$

The constant K is a value specific to each send function with its own underlying transport primitive. For example, the `MPI_Isend` which uses the DMA engine with peak throughput of 4 B/cycle, would have its transfer ratio K of about 0.25. The `MPI_Send`, with default peak throughput of 1 B/cycle (no DMA engine), should obtain a transfer ratio K around 1.

V. Results and Discussion

Using the MPPA Developer platform [18] with AB01 board and MPPA-256 Andey processor integrated, we set up ping-pong tests between:
(1) MPI rank 0 (CC_0) - MPI rank 15 (CC_15) and
(2) MPI I/O 128 (IOS_128) - MPI rank 15 (CC_15).
All MPI cluster ranks run at the same clock frequency of 400 MHz. The North IOS running the MPI I/O rank is configured to use the DDR controller at the default frequency of 600 MHz.

In each case, the same MPI send function is used on both sides (`MPI_Send` or `MPI_Isend`). At the first time, all tests are run without any optimization in order to calibrate the proper throughput of each context (Fig. 5). At the second time, we enable all optimization on the `MPI_Send` test and compare our optimization approaches in term of latency, throughput and messages sent per second (Fig. 6).

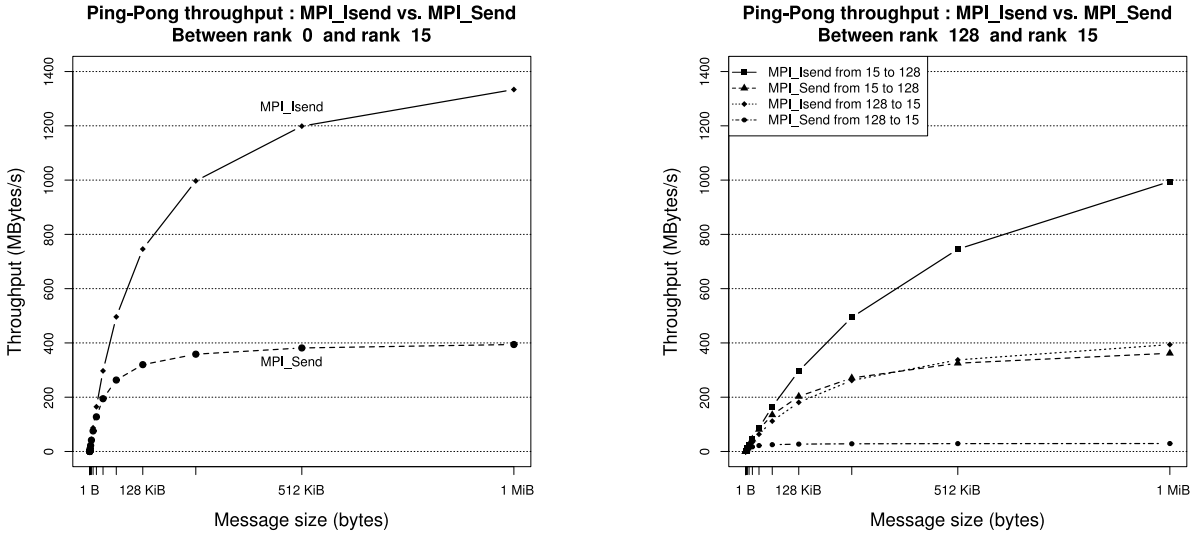
Each ping-pong is repeated 50 times. We assume that there is no waiting time inside the MPI send function, since all ranks start at the same time and run at the same clock speed. Hence, the duration of the MPI send function can be considered as the transmission time. Depending on the send context, the measured transmission time is fitted into a linear correlation $K \times N + h$ presented in Tab. IV. The standard deviation from all obtained results is always less than 0.2%.

A. Compute cluster ↔ Compute cluster

Communication links between CCs are bi-directionally symmetric. According to our model and the K values from Tab. IV, the estimated maximum throughput (given by $400 \times K^{-1} \ MB/s$) should be around 408 MB/s and 1481 MB/s for

From	To	MPI_Send	MPI_Isend
CC_0	CC_15	$t = 0.98 \times N + 31430$	$t = 0.27 \times N + 33690$
CC_15	CC_0	$t = 0.98 \times N + 30240$	$t = 0.27 \times N + 32850$
IOS_128	CC_15	$t = 13.52 \times N + 159544$	$t = 0.84 \times N + 181300$
CC_15	IOS_128	$t = 0.98 \times N + 129200$	$t = 0.26 \times N + 144500$

TABLE IV: Transmission time (cycles).



(a) Symmetric : Between ranks 0 and 15

(b) Asymmetric : Between ranks 128 and 15

Fig. 5: Ping-pong throughput `MPI_Send` (PE core) vs. `MPI_Isend` (DMA).

`MPI_Send` and `MPI_Isend` respectively. The ratio $\frac{h}{N}$ can be ignored in this case. Fig. 5a shows obtained results that match with our estimation model.

B. Compute cluster ↔ I/O subsystem

Contrary to the symmetric communication performance between CCs, the transmission rate on I/O subsystem relies on the DDR bandwidth, which is much lower than the on-chip memory on CCs. We observed higher K values and much more considerable overhead h on the IOS_128, showing that the communication link from IOS to CCs might be the bottleneck on the MPPA. It is then difficult ignoring $\frac{h}{N}$ in this case. Keeping on our throughput estimation by $400 \times (K + \frac{h}{N})^{-1}$ now matches with experiment results on Fig. 5b, where the performance gap between the CC_15 and the IOS_128 is also illustrated.

C. Optimization comparison

We focus now on finding, on a given message size, the best send method among the four (Normal, Eager, Lazy and DMA) to use on `MPI_Send`, in order to obtain lowest latency (round-trip-time) and/or highest ping-pong throughput, by enabling all optimizations and re-running our experiments between CCs. We also evaluate the number of messages sent per second in each approach by dividing the clock frequency (400 MHz) by the duration of the `MPI_Send` call (in cycles). As the message will now be "eagerly" sent or "lazily" buffered and `MPI_Send` returns right afterward, this duration on Eager(-splitting) or Lazy could no longer be evaluated as the transmission time in the Tab. IV, but respectively by :

$$E \times (floor(\frac{N}{96}) + 1)\ (cycles, E \approx 3800) \tag{5}$$

$$O_{memcpy}(N) = 1.28 \times N + 5300\ (cycles) \tag{6}$$

where E is the constant necessary cost to send 1 eager-split and O_{memcpy} is a linear function of `memcpy` cost. Note that in the Lazy approach, the message is sent in background by the RM.

Hence, Eager and Lazy methods provide lower latency and higher message rate on short buffers, since they were designed to get rid of two-sided synchronization and the buffer size is still small enough not to be outperformed by the DMA's high-throughput capacity.

Fig. 6a shows that the ping-pong latency for [1 .. 256 B] using eager-splitting is reduced by half compared to DMA or Normal. Otherwise, this latency increases radically as soon as its transmission time, despite being smaller at the beginning, getting repeated as many times as split segmentation ($floor(\frac{N}{96}) + 1$). On the other hand, using DMA on large

buffers optimizes bandwidth utilization compared to Normal (using PE) or Lazy (using RM) methods. (Fig. 6b).

Fig. 6c illustrates the message-rate of the four send methods. Not only this kind of measure gives user a high-level point of view about the implementation's capacity to support communication load, but it shows interesting advantages of Eager and Lazy methods in tuning MPI applications, thanks to their fast sending time for short messages and synchronization-free algorithm.

VI. High Performance Linpack (HPL) on MPPA-256

HPL benchmark was ported on MPPA-256 using our MPI implementation and cross-compiling of BLAS-Netlib and OpenBLAS. Each MPI compute rank, assigned to a compute cluster, only disposes 2 MB of memory, that make a total on-chip memory of 32 MB, enable to store up to 4 million double precision floating point numbers or a 2000×2000 matrix. Operating system space and user code (BLAS, MPI, HPL) must be taken into account as well. In practice, the HPL can run on the MPPA-256 with 1250×1250 matrix, which is a very small problem size for this kind of benchmark. As a result, communication, local indexing etc. has a significant cost with respect to the number of floating point operations ($O(N^3)$). Fig. 7b. does an estimation on further problem sizes if MPPA disposes more on-chip memory than its current capacity.

Fig. 7a shows the HPL result on MPPA-256 using BLAS and OpenBLAS. Note that 70 GFLOPS of annouced theoretical performance is for all the 256 cores, while the best benchmark score (R_{max} = 1.2 GFLOPS) was achieved using only 1 core per CC (i.e 16 cores in total) and MPI "eager" send. Also, we have seen no performance change by enabling MPI "lazy" optimization. This can be explained by well-scheduled HPL overlapping [8] in which, either MPI processes arrive to the communication step at the same time, or all heavy sends are done asynchronously by `MPI_Isend`, while lazy sending only shows its advantage in bad-scheduled `MPI_Send`. Furthermore, multi-threading on MPI compute ranks (OpenMP on CCs) did never give better HPL result, because of the small working set and the OpenMP overhead.

VII. Related works

A. MPI libraries design

Our design is similar to the *co-processor-only* MPI model on the Intel Xeon-Phi ™ platform [19], with support of OpenMP for hybrid multi-threaded programming. Besides MPI ranks running on CC, we introduce an MPI I/O rank running on an I/O subsystem of the chip as bridge to communicate with the host through the PCI-e interface, while there are no direct communication link between the host and the compute clusters on MPPA. Along the way, some collective MPI functions were also implemented (`MPI_Comm_split`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce` and `MPI_Barrier`).

Our message-trigger handling mechanism using the RM core was inspired by the similar work of Prylli and Tourancheau [20] [21] implementing the BIP protocol for an optimized MPI implementation over the Myrinet network, taking advantage of its dedicated hardware, an extra core like the MPPA RM core.

B. Similar architectures

Today, there exist other multi/many-core processors similar to the MPPA. Some of them has an MPI implementation, others do not. This section reviews these achitectures and, if existing, summarizes any MPI-oriented libraries and their performance related to our work.

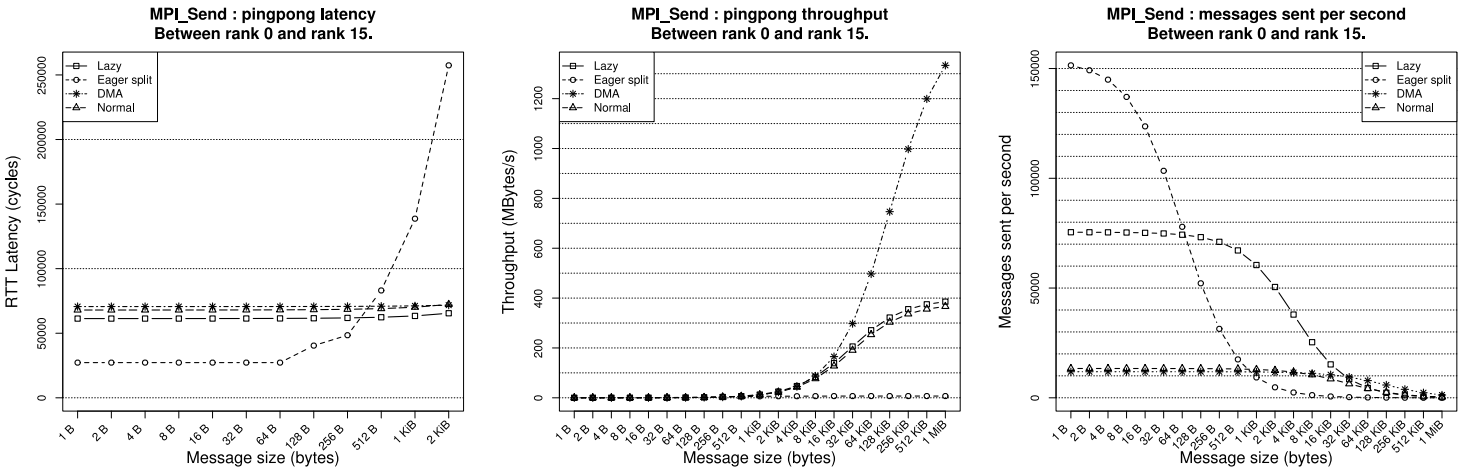
Freescale P4080 multi-core processor [22], based on the PowerPC architecture and resources virtualization technology from IBM, is mostly used in avionic industries and critical real-time systems. The P4080 processor has eights cores connected through CoreNet - the proprietary NoC of Freescale. Cache-coherence is guaranteed by a hardware mechanism. Our research for MPI implementation and topic on this platform did not encounter any relevant reference.

Raw processor [23], designed by the Computer Science Laboratory at MIT, combines 16 identical compute units, called tile. The 16 tiles are connected by one static NoC and two dynamic NoC. The static network is used for predefined memory access pattern at compile time, the dynamic ones are used for communication scheme at runtime. Psota and James [24] propose rMPI, the first MPI library over the Raw achitecture by inheriting some design aspects from MPICH and LAM/MPI also other specific implementation belonging to the Raw hardware. The highest throughput obtained on the ping-pong test of the Raw processor is about 150 MB/s with buffer size of 3.2 MB (100K words) [25].

STHORM processor [5] is a four-cluster-based Network-on-chip many-core from ST Microelectronics. Each cluster (named ENCore) hosts 16 STxP70-V4 processors (PE). Moreover, a special cluster unit, named fabric controller (FC) is responsible for interaction with the host memory and coordination of clusters. Coupled with an ARM processor as host, STHORM architecture mostly supports offloading model using either OpenCL [26] or through OpenMP [27] with extension primitives similar to `#pragma omp target` in OpenMP 4.0. Otherwise, no MPI library supporting the STHORM architecture was found as of this writing.

Neo chip [6], announced on March 2015 by Rex Computing, is a 256 VLIW core MIMD, organized on a 2D Mesh NoC. It is designed to aim the HPC market with disruptive exascale power ratio. The cache and memory system on Neo is rethough to reduce energy consumption by cutting off the virtual memory translation and other unecessary components. This choice would further produce more complexity and security issues when implementing operating system and software stack over Neo chip.

Tilera processors [4] are mainly used in high performance embedded systems such as networking and multimedia. The TilePro64 processor defines a flat 2D-mesh with 64 identical VLIW cores connected through the Tilera iMeshTM network-on-chip. Cache coherence on TilePro64 is guaranteed by a hardware mechanism called Dynamic Distributed Cache (DDC) [4]. Kang et al. [28] propose an MPI implementation on Tile64 processor which delivers up to 250 MB/s on `MPI_Send/MPI_Recv` communication, with the largest message size of 256 KB due to the limited memory per core. At this buffer size, our MPI implementation on MPPA delivers 400 MB/s on `MPI_Send` or up to 1 GB/s using DMA.

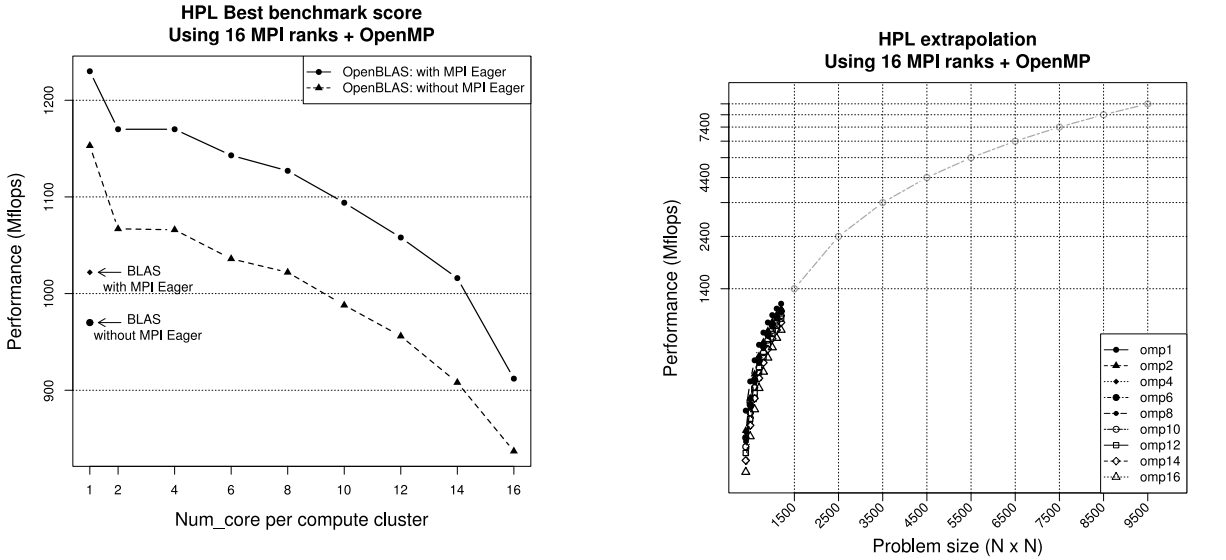


(a) Ping-pong latency (RTT) on short buffers

(b) Ping-pong throughput

(c) Number of messages sent per second

Fig. 6: Optimization approaches comparison.



(a) HPL benchmark on MPPA-256 against number of cores: BLAS vs. OpenBLAS.

(b) HPL score extrapolation with increasing problem size.

Fig. 7: HPL current performance (a) and extrapolation (b)

Intel Single-Chip Cloud (SCC) is a prototype aimed to promote many-core processor. Its 48 cores are organized in 24 dual-core clusters with access to off-chip DRAM shared/private region for all/each core through a look-up table (LUT), also a dedicated shared on-chip Message Passing Buffer (MPB). This memory architecture gives extra advantage for implementing quick message sending based on shared buffers. However, the use of the dynamic NoC routing on Intel SCC (instead of static NoC in MPPA) makes it difficult to evaluate the maximum communication latency [29] also incurs unordered packets, hence inappropriate for hard real-time applications.

The SCC-specific MPI-like native communication library (RCCE) delivers peak throughput of 55 MB/s on the ping-pong test [30]. By the same test, Clauss et al. [31] presented iRCCE (an improved RCCE version) and SCC-MPICH (an MPICH-based implementation over iRCCE) that reach respectively 150 MB/s and 120 MB/s of throughput. RCKMPI, an Intel MPI implementation for SCC is also bounded by the performance of the iRCCE layer. Our MPI library on MPPA was built from scratch over the MPPAIPC library, without any TCP/UDP layer, while an MPICH-based solution would not fit the cluster private memory space (2 MB). A such MPICH implementation on MPPA can be extrapolated to 1.0 GB/s by adding the same overhead of 20% of SCC-MPICH over to iRCCE.

Intel Many Integrated Core (MIC), known as Intel Xeon

Phi co-processor family, is a many-core computer architecture with autonomous on-chip Linux operating system and x86 compatibility. Intel MIC proposes three MPI programming models [19] which are (1) offload (host-only), (2) co-processor-only and (3) symmetric (both host and co-processor). The MPI communication in the intra-MPPA context corresponds to the co-processor-only intra-MIC case. Potluri et al. [32] studied the communication throughput of the MVAPICH2 library on Xeon Phi and their results show that a MIC-optimized MVAPICH2 library can delivers more than 9 GB/s of uni-direction throughput for messages up to 1 MB.

VIII. Conclusions

In this paper, we have introduced the design and performance issues of an MPI implementation on the Kalray MPPA-256. The MPPA-MPI library provides 1.2 GB/s of throughput for any inter compute-cluster point-to-point communication and this performance depends on the underlaying MPPAIPC library. Optimization ideas such as eager send and lazy message are proposed, implemented and compared to determine the best approach based on threshold. A synthetic model is also presented for each approach to evaluate their communication latency and throughput. We also learn that supporting MPI programming model is not an easy task on recent many-core processors, including MPPA, since MPI has become a large API with high-level abstractions and many-core hardware is taking more diversity and complexity. Thus, optimizing an MPI implementation on each of these platforms is even more not trivial.

On the other hand, the HPL benchmark is also successfully ported on MPPA as a validation test of our MPI library. The best performance of 1.2 GFLOPS is achieved by using only 16 cores instead of 256 available core due to limited on-chip memory makes us believe that traditional benchmark methods for supercomputers could hardly run effectively on many-core systems nowadays, given limited on-chip memory and the off-chip bandwidth, as well as the cache coherence challenge. Some dedicated benchmark suites for embedded systems such as Core Mark [33] and software APIs (like MCAPI [34]) may be a good match.

The next-generation MPPA processor aims at supporting global addressable DDR off-chip memory (Distributed Shared Memory) on clusters and will be more energy efficient. In a future work, we will study the performance gain and detailed power consumption of this new MPPA processor.

References

[1] William D Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT Press, 1999.

[2] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[3] Benoıt Dupont de Dinechin, Renaud Ayrignac, P-E Beaucamps, Patrice Couvert, Benoıt Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[4] Tilera Corporation. Tile processor architecture overview for the Tilepro series, February 2013.

[5] Julien Mottin, Mickael Cartron, and Giulio Urlini. The STHORM Platform. In *Smart Multicore Embedded Systems*, pages 35–43. Springer, 2014.

[6] Nicole Hemsoth. The Tiny Chip That Could Disrupt Exascale Computing, March 2015. http://www.theplatform.net/2015/03/12/the-little-chip-that-could-disrupt-exascale-computing.

[7] Antoine Petitet, Jack Dongarra, et al. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, September 2008.

[8] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[9] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[10] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[11] Z Xianyi, W Qian, and Z Chothia. OpenBLAS, version 0.2. 8. *URL http://www. openblas. net/. Fe tched*, pages 09–13, 2013.

[12] Siavash Khorsandi and Alberto Leon-Garcia. Robust non-probabilistic bounds for delay and throughput in credit-based flow control. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 2, pages 577–584. IEEE, 1996.

[13] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Rutuparna Tamhankar, Stergios Stergiou, Luca Benini, and Giovanni De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *Parallel and Distributed Systems, IEEE Transactions on*, 16(2):113–129, 2005.

[14] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. *Active messages: a mechanism for integrated communication and computation*, volume 20. ACM, 1992.

[15] Kalray Inc. MPPA-256 Cluster and I/O Subsystem Architecture, 2015. Specification documentation.

[16] Kalray Inc. MPPAIPC Performance, 2013. Benchmark report.

[17] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Öberg, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112. IEEE, 2002.

[18] Kalray Inc. Kalray platforms and boards. Accessed March 30, 2015.

[19] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[20] Loic Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on myrinet. In *Parallel and Distributed Processing*, pages 472–485. Springer, 1998.

[21] Loïc Prylli, Bernard Tourancheau, and Roland Westrelin. Modeling of a high speed network to maximize throughput performance: the experience of BIP over Myrinet. *Parallel and Distributed Processing Techniques and Applications-PDPTA*, 2:341–349, 1998.

[22] Freescale Semiconductor. QorIQ P4080 Communcations Processor Product Brief; Sep., 2008; Freescale Semiconductor.

[23] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[24] James Ryan Psota. *rMPI: An MPI-compliant message passing library for tiled architectures*. PhD thesis, Massachusetts Institute of Technology, 2005.

[25] James Psota and Anant Agarwal. rMPI: message passing on multicore processors with on-chip interconnect. In *High Performance Embedded Architectures and Compilers*, pages 22–37. Springer, 2008.

[26] Pierre G Paulin. OpenCL Programming Tools for the STHORM Multi-Processor Platform: Application to Computer Vision, 2013. 13th

International Forum on Embedded MPSoC and Multicore, July 15-19, 2013. *Otsu, Japan*, 24.

[27] Spiros N Agathos, Vassilios V Dimakopoulos, Aggelos Mourelis, and Antonis Papadogiannakis. Deploying OpenMP on an embedded multicore accelerator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 180–187. IEEE, 2013.

[28] Mikyung Kang, Eunhui Park, Minkyoung Cho, Jinwoo Suh, D Kang, and Stephen P Crago. MPI performance analysis and optimization on tile64/maestro. In *Proceedings of Workshop on Multi-core Processors for SpaceOpportunities and Challenges Held in conjunction with SMC-IT*, pages 19–23, 2009.

[29] Bruno dAusbourg, Marc Boyer, Eric Noulard, and Claire Pagetti. Deterministic Execution on Many-Core Platforms: application to the SCC. In *4th Many-core Applications Research Community (MARC) Symposium*, page 43, 2012.

[30] Timothy G Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, et al. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[31] Carsten Clauss, Stefan Lankes, Pablo Reble, and Thomas Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525–532. IEEE, 2011.

[32] Sreeram Potluri, Khaled Hamidouche, Devendar Bureddy, and Dhabaleswar K DK Panda. MVAPICH2-MIC: A High Performance MPI Library for Xeon Phi Clusters with InfiniBand.

[33] The Embedded Microprocessor Benchmark Consortium (EEMBC). The Coremark benchmark suite, 2014.

[34] The Multicore Association (MCA), 2014.