Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1654 – 1663

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor

Benoît Dupont de Dinechin[a,*], Pierre Guironnet de Massas[a], Guillaume Lager[a], Clément Léger[a], Benjamin Orgogozo[a], Jérôme Reybert[a], Thierry Strudel[a]

[a]*Kalray S.A., 445 rue Lavoisier, Montbonnot Saint Martin F38330, France*

Abstract

The Kalray MPPA®-256 is a single-chip manycore processor that integrates 256 user cores and 32 system cores in 28nm CMOS technology. These cores are distributed across 16 compute clusters of 16+1 cores, and 4 quad-core I/O subsystems. Each compute cluster and I/O subsystem owns a private address space, while communication and synchronization between them is ensured by data and control Networks-on-Chip (NoC). This processor targets embedded applications whose programming models fall within the following classes: Kahn Process Networks (KPN), as motivated by media processing; single program multiple data (SPMD), traditionally used for numerical kernels; and time-triggered control systems.

We describe a run-time environment that supports these classes of programming models and their composition. This environment combines classic POSIX single-process multi-threaded execution inside the compute clusters and I/O subsystems, with a set of specific Inter-Process Communication (IPC) primitives that exploit the NoC architecture. We combine these primitives in order to provide the run-time support for the different target programming models. Interestingly enough, all these NoC-specific IPC primitives can be mapped to a subset of the classic synchronous and asynchronous POSIX file descriptor operations. This design thus extends the canonical 'pipe-and-filters' software component model, where POSIX processes are the atomic components, and IPC instances are the connectors.

Keywords: Distributed Memory; POSIX IPC; Network-on-Chip;

1. Introduction

Single-chip processors that integrate up to hundreds of capable cores are around the corner. Examples include the Cavium Octeon network processors, the Intel MIC family of accelerators, and the Tilera TILE-Gx series. Most of these manycore processors however, along with academic projects such as the Rigel architecture [1, 2], strive to present a single-address space accessible through load/store operations to system and application programmers, albeit with various degrees of cache coherence support. Such architectural choice implies deep memory hierarchies, which increases energy consumption per operation, and makes accurate timing prediction virtually impossible.

The Kalray MPPA®-256 manycore processor architecture follows a different approach, as it targets embedded applications where low energy per operation and time predictability are the primary requirements. Like high-end supercomputers such as members of the IBM BlueGene family, the cores of the Kalray MPPA®-256 processor

*Corresponding author. Tel.: +33-6-8834-7737.
E-mail address: benoit.dinechin@kalray.eu.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.333

are grouped into compute clusters each associated with a private address space, and the clusters are connected by specialized networks. Inside compute clusters, classic shared memory programming models such as POSIX threads and OpenMP language extensions are available. Whenever applications require the resources of several compute clusters, the key problem faced by programmers is how to distribute and coordinate work.

Two families of programming models have proved successful for the exploitation of distributed memory architectures: Single Program Multiple Data (SPMD) execution with collective operations [3], which is prevalent on all distributed memory supercomputers; and Models of Computation (MoC) such as derivatives of Kahn Process Networks (KPN) [4], which are well suited to the type of media and signal processing found in embedded applications. In the area of embedded systems, a significant trend is the move from federated architectures to integrated architectures [5], where several functions are hosted by the same computing units and some functions are distributed across computing units.

In this paper, we present the design and implementation of a distributed run-time environment for the MPPA®-256 processor toolchain that supports representatives of these three classes of programming models, and enables their composition inside applications. Specifically, The Kalray MPPA®-256 processor toolchain supports:

- a dataflow Model of Computation (MoC) that extends cyclostatic dataflow (CSDF) [6] with zero-copy features such as the Karp & Miller [7] firing thresholds;
- a distributed implementation of the Communication by Sampling (CbS) primitive, which is a cornerstone of the Loosely Time-Triggered Architecture (LTTA) for integrated embedded systems [8].
- a hybrid SPMD programming model inspired by the Bulk Synchronous Parallel library [9], where processors of the classic BSP model [10] are replaced by processes;

This distributed run-time environment is also used directly by third-party vendors as a target for their MoC-specific code generator. The run-time API is biased towards the UNIX process and Inter-Process Communication (IPC) model, where processes execute in the different address spaces of the MPPA®-256 processor, and IPC primitives are adapted to the NoC architecture. This interface is simple to explain to third parties, and appears to relieve concerns about the vendor lock-in of proprietary APIs.

The presentation is as follows. Section 2 introduces the main features of the MPPA®-256 processor relevant to this work. Section 3 motivates and details the design of the run-time environment. Section 4 discusses how this run-time environment is exploited by programming models. Section 5 presents our first implementation results.

2. MPPA®-256 Processor Overview

2.1. Compute Clusters

Each MPPA®-256 compute cluster (Figure 1 (a)) comprises a banked parallel memory shared by 17 VLIW cores. The first 16 cores, referred to as processing engines (PEs), are dedicated to application code. The 17th core, referred to as the resource manager (RM), is reserved for the system. It implements the same VLIW architecture as the PEs but is distinguished by its privileged connections to the NoC interfaces through event lines and interrupts. The other bus masters on the shared memory are the NoC Rx interface, the NoC Tx interface paired with a 8-channel DMA engine, and the debug support unit (DSU). Each PE and the RM are fitted with private 2-way associative instruction and data caches.

2.2. Global Architecture

The MPPA®-256 chip integrates 16 compute clusters and 4 I/O subsystems, one on each side (Figure 1 (b)). Each I/O subsystem relies on four RM cores with a shared D-cache, static memory, and external DDR access. They operate controllers for the PCIe, Ethernet, Interlaken, and other I/O devices. Each compute cluster and I/O subsystem owns a private address space, and the cores in each address space are the only ones able to operate the local peripherals, including the NoC interfaces.

The 16 compute clusters and the 4 I/O subsystems are connected by two parallel NoC with bi-directional links, one for data (D-NoC), and the other for control (C-NoC). There is one NoC node per compute cluster, and four nodes per I/O subsystem. Each NoC node is associated with a D-NoC router and a C-NoC router. The two NoC

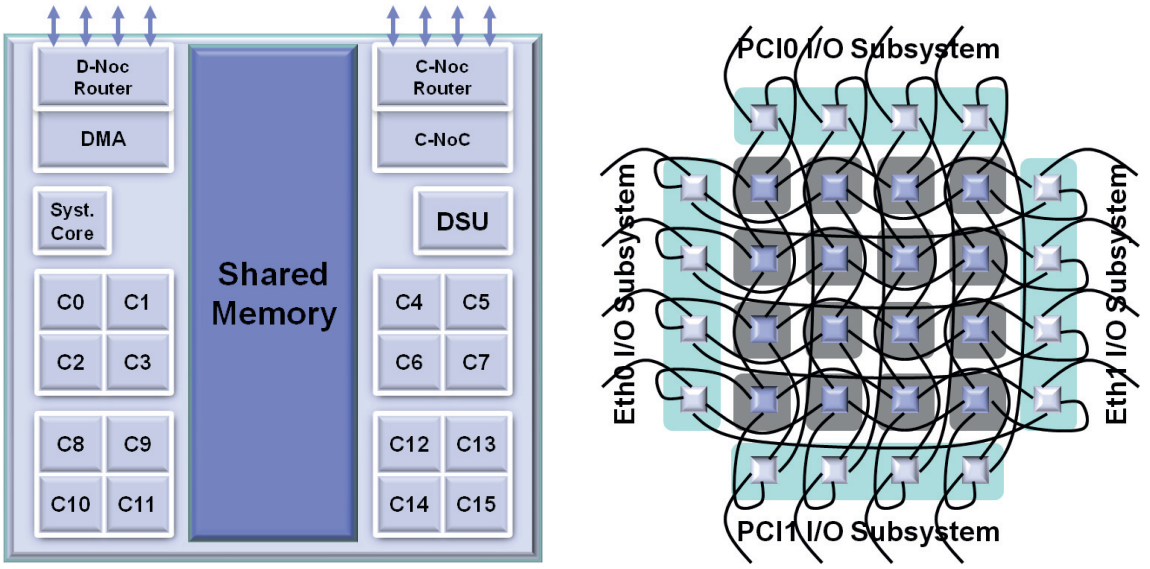

Fig. 1. MPPA®-256 compute cluster (a) and global architecture (b).

are identical with respect to the nodes, the 2D torus topology with off-chip extensions (Figure 1 (b)), and the wormhole route encoding. They differ at their device interfaces, and by the amount of packet buffering in routers. NoC traffic through a router does not interfere with the memory buses of the underlying I/O subsystem or compute cluster, unless the NoC node is a destination.

Each NoC packet is composed of a header and a payload. The header contains a bit-string that encodes the route as direction bits for each NoC node, a 'tag' that is similar to a TCP/IP port, a EOT (end-of-transfer) bit, and an optional offset. The route bit-string encodes either unicast or multicast transfer. For multicast, additional bits in the route trigger packet delivery at any NoC node on the route to the underlying compute cluster or I/O subsystem. All multicast targets receive the same payload, tag, EOT bit, and offset. The D-NoC and the C-NoC both ensure reliable delivery, and messages that follow the same route arrive in order. However, there is no acknowledgment at the source node that a message has been received at the destination node(s).

2.3. Data NoC Features

The D-NoC is dedicated to high bandwidth data transfers and supports Quality of Service (QoS), thanks to non-blocking routers and rate control at the source node [11]. On the Rx interface, the tag is used to address a Rx buffer descriptor which maintains the base address, the current write address, and the size of a memory area within the local address space. Upon packet arrival, the payload is written either at the current write address, which is incremented, or at the base address plus the offset, if the packet header specifies any. Key D-NoC features are the atomic receive of packets in the Rx buffer memory areas, and automatic wrap-around of the write pointer when not in offset mode.

In addition to addresses and size, the Rx buffer descriptor counts the number of bytes written to memory, and maintains a notification counter which is incremented if the header EOT bit of the received packet is set. Whenever a notification counter is non-zero, a Rx event is posted to the RM core associated with the NoC node. This core may atomically load and clear the notification counter, thus acknowledging the event. An alternate Rx buffer mode is also available where the notification counter is initialized to non-zero. Each header EOT bit set decrements the counter, and a Rx event is posted to the RM core whenever the counter reaches zero.

2.4. Control NoC Features

The C-NoC is dedicated to peripheral D-NoC flow control, to power management, and to application software messages. C-NoC traffic is not regulated and the packet payload is always 64 bits. On the C-NoC Tx interface, a

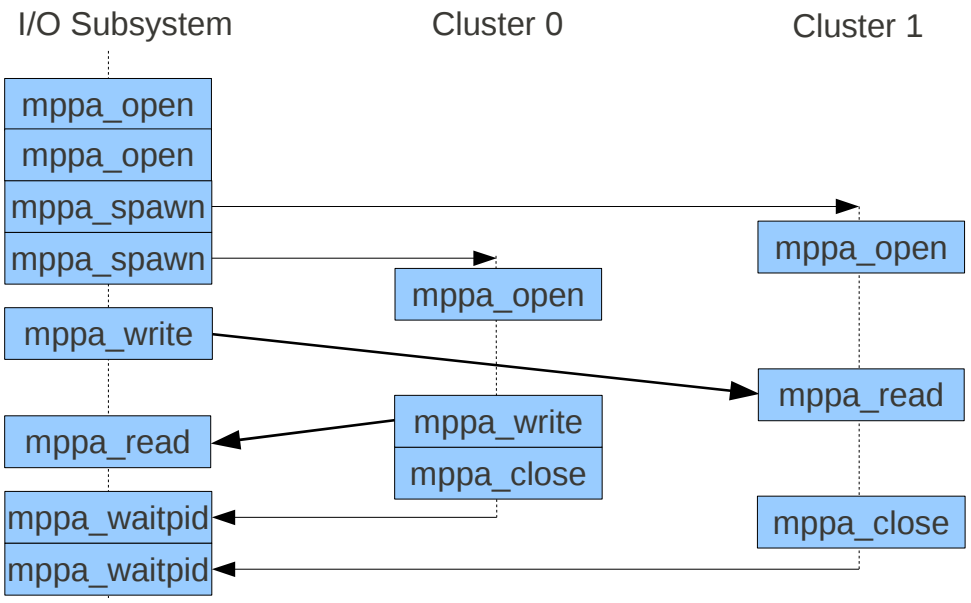

Fig. 2. Use of POSIX-like processes and IPC.

memory-mapped location records the (route, tag) header fields, while two other 64-bit memory-mapped locations receive the payload. Writing to either of these two locations sends the packet, respectively with or without the EOT bit set.

The C-NoC Rx interface comprises 64-bit mailboxes, one per tag, for the incoming payload of software messages. Each mailbox may be configured in two modes: normal, or synchronization. In normal mode, a new payload overwrites the previous mailbox content, and the RM core sees an event only if the packet carries a EOT. In synchronization mode, the new payload is OR-ed with the previous mailbox content, and the RM core sees an event only if the new content has all bits set and the packet carries a EOT.

3. The Run-Time Environment

3.1. Operating Environment

The atomic software components of MPPA®-256 applications are POSIX processes, with a main process running on an I/O subsystem, and child processes running on the compute clusters. The main process is hosted by a SMP Linux kernel, and operates the peripherals of the MPPA®-256 processor. For hard real-time use cases, the SMP Linux kernel is replaced by RTEMS the RTEMS real-time executive (`http://www.rtems.org/`) with one instance per RM core of the I/O subsystem.

The compute cluster processes are managed by a POSIX 1003.13 profile 52 operating system: single process, multiple threads, with a file system (remoted to a I/O subsystem). The RM core is dedicated to system tasks, in particular NoC Rx event management and D-NoC Tx DMA activation, while the PE cores are available under the POSIX thread interface for application code. The OpenMP support of the GNU C/C++ compilers is also available as an alternative to explicit POSIX threads.

The main process running on a I/O subsystem spawns and waits for child processes on the compute clusters by using a variant of `posix_spawn()` named `mppa_spawn()`, and the UNIX `waitpid()` renamed to `mppa_waitpid()` (Figure 2). The child processes inherit `int argc`, `char *argv[]`, as `main()` parameters. The `mppa_spawn()` arguments include the execution context as an ordered set of NoC nodes, and the rank of the child process within that set. These are prefixed to the child environment available in `extern char **environ`.

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`
Stream	`/mppa/stream/`*rx_nodes*`:`*dnoc_tag*	$1:M$	`SIGEV_NONE`, `SIGEV_CALLBACK`
RQueue	`/mppa/rqueue/`*rx_node*`:`*dnoc_tag*`/`*tx_nodes*`:`*cnoc_tag*`/`*msize*	$N:1$	`SIGEV_NONE`, `SIGEV_CALLBACK`
Channel	`/mppa/channel/`*rx_node*`:`*dnoc_tag*`/`*tx_node*`:`*cnoc_tag*	$1:1$	`SIGEV_NONE`, `SIGEV_CALLBACK`

Table 1. NoC connector pathnames, signature, and asynchronous I/O sigevent notify actions.

3.2. Key Design Principles

The run-time environment design leverages the canonical 'pipe-and-filters' software component model [12], where POSIX processes are the atomic components, and communication objects accessible through file descriptors are the connectors [13]. Like POSIX pipes, those connectors have distinguished transmit (Tx) and receive (Rx) ports that must be opened in modes `O_WRONLY` and `O_RDONLY` respectively. Unlike pipes however, they may have multiple Tx or Rx endpoints, and support POSIX asynchronous I/O operations with call-back. Following the components and connectors design philosophy, the NoC node or node sets at the endpoints are completely specified by the connector pathnames (Table 1).

Application code remains oblivious to the node identities by using ranks within the NoC node sets in pathnames. The connector pathnames also specify the C-NoC or D-NoC tags involved at the endpoints, since the connected processes must agree on them. On the other hand, the routes between NoC nodes are not specified in the connector pathnames (Table 1). Rather, they are obtained by the Tx processes upon opening the connectors from a centralized allocator that manages the D-NoC QoS parameters. This design allows multi-path routing between pairs of nodes: whenever multiple POSIX threads from the same Tx process open a connector, each obtains a file descriptor bound to a potentially different route.

The POSIX asynchronous I/O operations of the NoC file descriptors include: `aio_read()`, `aio_write()`, `aio_error()`, `aio_return()`, and `aio_wait()` that combines `aio_suspend()` with `aio_return()`. The `aio_write()` operations are interpreted as a request to activate the Tx DMA engine. POSIX asynchronous operations rely on a `struct aiocb` control block, which itself includes a `struct sigevent` as member `aio_sigevent`. The latter records in member `aio_sigevent.sigev_notify` what action is required when asynchronous I/O completes. Besides `SIGEV_NONE`, our implementation supports the `SIGEV_CALLBACK` notify action to activate a call-back function whose pointer is provided in member `aio_sigevent.sigev_notify_function`. The call-back function executes non-deferred on the RM core and its argument is in member `aio_sigevent.sigev_value`, which typically points to the enclosing `struct aiocb`.

The Sync and the Portal connectors (Table 1), discussed below, may connect N Tx processes to M Rx processes. Likewise, the Stream connectors may connect one Tx process to M Rx processes. In all cases of multiple Rx processes, write operations are either de-multiplexed to target one among the M Rx processes, or are multicasted simultaneously to the M Rx processes by enabling the C-NoC (Sync connector) or D-NoC (Portal ans Stream connectors) multicast mode. De-multiplexing or multicasting is activated by specifying several NoC nodes in the *rx_nodes* part of the pathname, and the selection between them depends on calling `ioctl()` with the `TX_SET_RX_RANK` or `TX_SET_RX_RANKS` requests.

3.3. NoC Connector Operations

Sync connector $N:1$ operations involve a Rx process, whose NoC node is specified by *rx_nodes* in the pathname, and a number of Tx processes not specified in the pathname. On the Rx process, an initial `long long match` value is provided by calling `ioctl()` with the `RX_SET_MATCH` request. On the Tx processes, a `write()` sends a 64-bit value which is OR-ed with the current match value maintained by the Rx process. When the OR result equals `(long long)-1`, the `read()` operation on the Rx process returns.

Portal connector $N:1$ operations target a memory area on a single Rx process, whose NoC node is specified by *rx_nodes* in the pathname, and a number of Tx processes not specified in the pathname. The *dnoc_tag* parameter identifies the Rx buffer descriptor that manages the memory area in decrement mode of the notification counter. The Tx processes call `pwrite()` to send data, without the Rx process being involved except when the notification

Type	Pathname	Tx:Rx	`aio_sigevent.sigev_notify`
Buffer	`/mppa/buffer/`*rx_node*`#`*number*	1 : 1	`SIGEV_NONE`, `SIGEV_CALLBACK`
MQueue	`/mppa/mqueue/`*rx_node*`#`*number*`/`*tx_node*`#`*number*`/mcount.tsize`	1 : 1	`SIGEV_NONE`, `SIGEV_CALLBACK`

Table 2. PCIe connector pathnames, signature, and asynchronous I/O sigevent notify actions.

count reaches a trigger value that completes the Rx process `aio_read()`. The trigger value is specified in member `aio_lio_opcode` of the `struct aiocb *` argument passed to the `aio_read()` call. Whether there is a EOT bit set or not in the last packet sent by `pwrite()` is configured by calling `ioctl()` with respective requests `TX_NOTIFY_ON` and `TX_NOTIFY_OFF`.

Stream connector operations support data broadcast from one Tx process, not specified in the pathname, to several Rx processes, whose NoC nodes are specified by *rx_nodes* in the pathname. The *dnoc_tag* parameter identifies the Rx buffer descriptor that manages the memory area in wrap-around mode at each Rx process.

The Tx process calls `write()` to send data, without the Rx processes being involved, and the last packet sent by `write()` always has the EOT bit set. Each Rx process must post a receive buffer by calling `aio_read()`. The `aio_offset` member of the `aiocb *` argument passed to the `aio_read()` is automatically adjusted to the base of the latest payload received. So the Rx process may read this value to access fresh data as its own pace, without having to consume explicitly all data sent by the Tx process.

RQueue connector operations support atomic enqueue of *msize*-byte messages from several Tx processes, and dequeue from a single Rx process. The *rx_node* and *dnoc_tag* parameters in the pathname specify the NoC node for the Rx process and the Rx buffer descriptor involved. The NoC nodes associated with the Tx processes and C-NoC mailbox used for flow control are respectively specified by the *tx_nodes* and *cnoc_tag* parameters in the pathname. The Rx process extracts messages by calling `read()`. The effect is to copy the message body to the user supplied buffer, and also to compute a credit value by adding a bias[1] to the total count of messages read so far from this Tx process. This credit value is then sent back to the C-NoC mailbox of the originating Tx process, whose identity is available thanks to a 64-bit software header prefixed to the NoC packet payload by the `write()` operation. When a Tx process calls `write()`, the count of its previously sent messages is compared to a local credit value. The call blocks as long as the local credit value of the Tx process compares lower. Each C-NoC credit message sent by the Rx process has the EOT bit set. When the RM core processes the corresponding event, it copies the C-NoC mailbox content to the local credit value of the Tx process, and unblocks the pending call to `write()` if any.

Channel connector operations effectuate a rendez-vous between a Tx process and a Rx process, paired with zero-copy data transfer. The *rx_node* and *dnoc_tag* parameters in the pathname specify the NoC node for the Rx process and the Rx buffer descriptor involved. The *tx_node* and *cnoc_tag* in the pathname specify the NoC node for the Tx process and the C-NoC mailbox used to agree on the transfer size. The Rx process calls `read()` to configure the Rx buffer with the supplied memory area base address and size. This size is sent to the Tx process through the C-NoC mailbox. When the Tx process calls `write()`, it samples the C-NoC mailbox to get the read size, and computes the minimum with the size provided as `write()` argument. Data to write are pushed to the D-NoC Tx interface, with the EOT bit set on the last packet. When the RM core processes the EOT event of the Rx buffer, it accesses the hardware counter of bytes written to memory. This is used to provide the return value to the pending call to `read()`, which is unblocked and returns.

3.4. PCIe Connector Operations

The coupling between a PCIe host and the MPPA®-256 for application acceleration is achieved by providing an implementation of `mppa_spawn()` and `mppa_waitpid()` on the host, and by using connectors called Buffer and MQueue. The main differences between these PCIe connectors and the similar Portal and RQueue NoC connectors are implied by the capabilities of the architectural resources involved. In particular, no D-NoC Rx buffers or C-NoC mailboxes are used. This is translated in the PCIe connector pathnames, which are summarized in Table 2. In this table, *number* refers to user-defined integers whose only purpose is to disambiguate connectors.

[1] The bias value is the initial credit of messages available to each Tx process.

Buffer connector operations operate on a memory buffer in the Rx process address space where the Tx processes can write at arbitrary offset. The Rx process is not aware of the communication except for a notification count that unlocks the Rx process when the `trigger` supplied to `aio_read()` is reached. The *rx_node* must be named `host` or belong to the NoC nodes of the PCIe I/O subsystem. The Buffer connector operations are translated to commands to the MPPA®-256 PCIe interface DMA engine. As this DMA engine is able to perform remote reads in addition to remote writes between the host system memory and the MPPA®-256 DDR memory, the process on the host may call `pread()` to read data at arbitrary offset within an area of the MPPA®-256 DDR memory that has been previously identified by calling `aio_read()` with a `trigger` value of zero.

MQueue connector operations manage a message queue between the Rx process and the Tx process that can queue up to *mcount* messages with a maximum of *tsize* total size in bytes. The *rx_node* and *tx_node* must be different, and must be named `host` or belong to the corresponding PCIe I/O subsystem. The MQueue connector data structures are hosted in a area of the I/O subsystem static memory which is directly accessed by the host CPU memory instructions through the PCIe interface. Unlike the RQueue messages, the MQueue messages of a connector instance are not constrained to have the same size.

4. Support of Programming Models

4.1. Distributed Computing Primitives

Split Phase Barrier The arrival phase of a master-slave barrier [14] is directly supported by the Sync connector, by mapping each process to a bit position. The departure phase of a master-slave barrier [14] is realized by another Sync connector in $1 : M$ multicasting mode.

Remote Message Queue The RQueue connector implements the remote queue [15], with the addition of flow control. This is an effective $N : 1$ synchronization primitive [16], by the atomicity of the enqueue operation [17].

Active Message Server Active messages integrate communication and computation by executing user-level handlers which consume the message as arguments [18]. Active message servers are efficiently built on top of remote queues [15]. In case of the RQueue connector, the registration of an asynchronous read user call-back enables to operate it as an active message server. Precisely, application code starts the server with a call to `aio_read()`, and the service is kept alive by the call-back function which calls `aio_read()` on its `struct aiocb *` argument after its main processing.

Remote Memory Put One-sided remote memory access operations (RMA) are traditionally named PUT and GET [18], where the former writes to remote memory, and the latter reads from remote memory. The Portal connector directly supports the PUT operation on a Tx process by writing to a remote D-NoC Rx buffer in offset mode. As the Rx buffer maintains a notification counter, it is possible to combine data transfer with oblivious synchronization [19], that is, unlocking the Rx process once a specified number of notifications is reached.

Remote Memory Get The GET operation is implemented by active messages that write to a Portal whose Rx process is the sender of the active message. As the call-back function executes on the RM core, this implementation realizes the benefits of the 'communication helper thread' or 'data server' approach [20] of supercomputer communication libraries such as ARMCI [21], where active message processing does not interfere with the user cores that execute application threads.

Rendez-Vous The classic rendez-vous that combines $1 : 1$ synchronization between two processes with data transfers is implemented with zero intra-process copy by using the Channel connector with synchronous read and write operations.

4.2. Extended Cyclostatic Dataflow

The Kalray MPPA®-256 processor toolchain supports a dataflow MoC that extends CSDF in a way comparable to the 'special channels' of Denolf et al. [22]. In this work, four types of special channels are discussed:

- Non-destructive read, which are equivalent to the firing thresholds of Karp & Miller [7].
- Partial update, where produced tokens are sparsely stored.
- Multiple consumers, with non-sequential token release.
- Multiple producers, which generalizes the partial update.

Among these, only the multiple producers channel cannot be reformulated with an equivalent CSDF pattern.

The Kalray MPPA®-256 dataflow compiler optimizes communication through local channels (those with producers and consumers inside the same compute cluster) as pointer manipulations. For the global channels, the dataflow compiler run-time uses the Channel connector to implement the regular channels, and the channels with the firing thresholds. The three other types of global special channels are implemented by PUT and GET operations over a Portal connector and a RQueue connector configured as an active message server.

4.3. Communication by Sampling

The loosely time-triggered architecture (LTTA) [8] is a software architecture for integrated embedded systems (as opposed to federated embedded systems) [5] characterized by the communication by sampling (CbS) mechanism. This mechanism assumes that [23]:

- read and write operations are performed independently by the compute units, using different logical clocks;
- the communication medium behaves like a shared memory: values are sustained and are periodically refreshed.

The CbS mechanism is implemented by using the Stream connector. Each Rx process i considers the Rx buffer as logically decomposed into k_i message slots. The Stream connector ensures that the `aio_offset` member of the `aiocb *` argument of `aio_read()` is atomically updated with the offset in the buffer of the last written slot. This value may be used by the Rx process to copy the last message received. The value of k_i is computed as $\lceil \frac{f_{Tx}}{f_{Rx_i}} \rceil + 1$ where f_{Tx} and f_{Rx_i} are the respective sampling frequencies of the Tx process and Rx process i.

4.4. Hybrid Bulk Synchronous Parallel

For application components that are better suited to SPMD execution, the Kalray MPPA®-256 toolchain supports a variant of the Bulk Synchronous Parallel (BSP) model. A BSP machine [10] is a set of n processing units with local memory, with each unit connected to the router of a network. SPMD computations are structured as sequences of super-steps. In a super-step, processors compute using their local memory, and exchange messages with each other. Messages sent during super-step s are received only at the beginning of super-step $s + 1$.

Initially, the type of communication considered for BSP execution was bulk message passing between pairs of processors. By the time the BSPLib standard [9] was consolidated, one-sided communication with PUT and GET had become the method of choice for BSP algorithms, thanks to the success of the Cray SHMEM library [24]. One-sided operations need to address remote memory locations, and this motivated the local object registration primitives of BSPLib. Super-steps are delimited by bulk synchronization, which is implemented by barrier synchronization, or by oblivious synchronization [19].

The Kalray MPPA®-256 BSP model is hybrid because the BSP machine processors are mapped to processes spawned to the compute clusters, while parallel execution between the cluster PE cores is managed by OpenMP or through the POSIX threads API. The implementation involves on each process:

- A pair of Portal connectors for each registered object;
- A RQueue connector with call-back used as a data server;
- A pair of Sync connectors for barrier synchronization;
- A pair of Portal connectors for remote memory fences.

5. First Implementation Results

To evaluate the efficiency of our distributed run-time environment, we adapted the distributed memory matrix multiplication algorithm MatMulG of Gerbessiotis [24]. This algorithm operates by decomposing the input and result matrices into $\sqrt{p} \times \sqrt{p}$ tiles, which are distributed across the memory spaces of p processes. After initializing its owned tile of the result matrix, each process iterates $\sqrt{p}$ rounds, where each round contributes the multiplication of two respective tiles from the input matrices. The MatMulG algorithm relies on GET operations to fetch the input matrix tiles it needs. A super-step synchronization waits until completion of the GET operations (Figure 3). Our MatMulG variant assumes array storage in row major order instead of column major order, but

1: Let q = pid
2: Let $p_i = q \mod \sqrt{p}$
3: Let $p_j = q/\sqrt{p}$
4: Let $C_q = 0$
5: **for** $0 \le l < \sqrt{p}$ **do**
6: $\quad a \leftarrow A_{((p_i+p_j+l) \mod \sqrt{p})+\sqrt{p}*p_j}$
7: $\quad b \leftarrow B_{((p_i+p_j+l) \mod \sqrt{p})*\sqrt{p}+p_i}$
8: $\quad$ sync
9: $\quad$ Let b^t = transpose(b)
10: $\quad C_q += a \times^t b^t$
11: **end for**

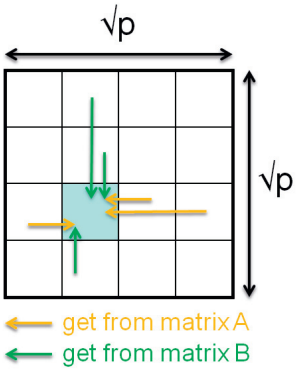

Fig. 3. The MatMulG algorithm of Gerbessiotis [24] adapted for row-major order.

this does not change the operating principles of the algorithm. More importantly, the input and result matrices are hosted in the DDR memory of one of the I/O subsystems, in order to spare memory in the compute clusters.

With our distributed run-time environment, we implement the GET operations as discussed in Section 4.1. The MatMulG algorithm requires one floating-point addition and one floating-point multiplication for each element of the $N \times N$ result matrix. On the cores of the MPPA®-256 processor, one double precision addition may issue every cycle, and one double precision multiplication may issue every other cycle. So the minimum time for multiplying two $N \times N$ double precision matrices is $3N^3$ cycles. Inside the compute clusters, we rely on the standard GCC OpenMP support based on POSIX threads to distribute parallel work across the 16 cores.

We execute the MatMulG algorithm on $p = 16$ clusters of the MPPA®-256 processor, that is, $\sqrt{p} = 4$. The A, B, C are double precision matrices dimensioned 576×576, and the compute clusters operate on tiles a, b dimensioned 144×144. The theoretical minimum time for double precision tile multiplication on 16 PEs is $3 \times 144^3/16$=0.56MCycles, but we observe 1.10MCycles that include cache miss effects, OpenMP run-time overheads, and the tile transposition cost of 0.09MCycles. The MatMulG algorithm on $p = 16$ iterates four times, and we measure 7.41MCycles for the total running time. This means that $7.41 - 4 \times 1.10 = 3.01$MCycles are not overlapped with computation, so the current MatMulG algorithm efficiency is 59%.

The main contributor to this overhead is the DDR bandwidth exploitation on the I/O subsystem, which could and will be significantly enhanced by ensuring a better spatial locality of the memory accesses in order to benefit from the DDR controller page management. The other significant contributor is related to waiting on the locks that prevent several threads from concurrently accessing the NoC Tx interface resources. Even though this thread safety is implied by the specification of the classic POSIX file operations, we plan to provide a faster implementation where only the thread that opened a NoC connector is allowed to operate the resulting file descriptor.

6. Conclusions

We describe a distributed run-time environment for Kalray MPPA®-256, a commercial single-chip 288-core processor with 20 memory spaces distributed across 16 compute clusters and 4 I/O subsystems. This environment supports a dataflow MoC based on cyclostatic dataflow, the Communication by Sampling (CbS) primitive of the Loosely Time-Triggered Architecture, and a SPMD programming model inspired by the Bulk Synchronous Parallel (BSP) model. This distributed run-time environment is also used directly by third-party vendors as a target for their code generation flow.

The run-time API is biased towards the UNIX process and IPC model, where processes execute on the different address spaces of the MPPA®-256 processor, and IPC primitives are adapted to the NoC architecture. Communication and synchronization between processes is achieved by file descriptor operations on special files locally opened with flags `O_WRONLY` for Tx or `O_RDONLY` for Rx. This design leverages the canonical 'pipe-and-filters' software component model, where POSIX processes are the atomic components, and IPC primitives are the connectors. Components authored in different programming models can thus be assembled and run as a single application on the MPPA®-256 processor.

Initiatives like the Multicore Association MCAPI or the Intel SCC computer RCCE library position message passing as the low-level primitive for communication and synchronization. Our approach is more related to the supercomputer communication subsystems for SPMD programming models such as Cray SHMEM, Berkeley GASNet, Argonne ARMCI, or IBM BlueGene/Q PAMI. Even though we support other programming models, we also found that one-sided communications, bulk synchronization, and active messages provide effective foundations for distributed run-time environments.

References

[1] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, S. J. Patel, Rigel: an architecture and scalable programming interface for a 1000-core accelerator, in: Proceedings of the 36th annual International Symposium on Computer architecture, ISCA '09, 2009, pp. 140–151.

[2] D. Johnson, M. Johnson, J. Kelm, W. Tuohy, S. Lumetta, S. Patel, Rigel: A 1,024-core single-chip accelerator architecture, IEEE Micro 31 (4) (2011) 30–41.

[3] S. Gorlatch, Send-Receive Considered Harmful: Myths and Realities of Message Passing, ACM Trans. Program. Lang. Syst. 26 (2004) 47–56.

[4] G. Kahn, The semantics of a simple language for parallel programming, in: Proceedings of the IFIP Congress 74, 1974, pp. 471–475.

[5] M. di Natale, A. Sangiovanni-Vincentelli, Moving from Federated to Integrated Architectures: the role of standards, methods, and tools, Proc. of the IEEE 98 (4) (2010) 603–620.

[6] G. Bilsen, M. Engels, R. Lauwereins, J. A. Peperstraete, Cycle-static dataflow, IEEE Transactions on Signal Processing 44 (2) (1996) 397–408.

[7] R. Karp, R. Miller, Properties of a model for parallel computations: Determinacy, termination, queueing, SIAM J. 14 (1966) 1390–1411.

[8] S. Tripakis, C. Pinello, A. Benveniste, A. L. Sangiovanni-Vincentelli, P. Caspi, M. D. Natale, Implementing Synchronous Models on Loosely Time Triggered Architectures, IEEE Trans. Computers 57 (10) (2008) 1300–1314.

[9] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, R. H. Bisseling, Bsplib: The bsp programming library, Parallel Computing 24 (14) (1998) 1947–1980.

[10] L. G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.

[11] Z. Lu, M. Millberg, A. Jantsch, A. Bruce, P. van der Wolf, T. Henriksson, Flow regulation for on-chip communication, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, 2009, pp. 578–581.

[12] K.-K. Lau, Z. Wang, Software component models, IEEE Trans. Softw. Eng. 33 (10) (2007) 709–724.

[13] S. Kell, Rethinking software connectors, in: International workshop on Synthesis and analysis of component connectors: in conjunction with the 6th ESEC/FSE joint meeting, SYANCO '07, 2007, pp. 1–12.

[14] O. Villa, G. Palermo, C. Silvano, Efficiency and scalability of barrier synchronization on noc based many-core architectures, in: Proceedings of the 2008 international conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '08, 2008, pp. 81–90.

[15] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, J. D. Kubiatowicz, Remote queues: exposing message queues for optimization and atomicity, in: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, SPAA '95, 1995, pp. 42–53.

[16] V. Papaefstathiou, D. N. Pnevmatikatos, M. Marazakis, G. Kalokairinos, A. Ioannou, M. Papamichael, S. G. Kavadias, G. Mihelogiannakis, M. Katevenis, Prototyping efficient interprocessor communication mechanisms, in: Proceedings of the 2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2007), 2007, pp. 26–33.

[17] M. G. Katevenis, E. P. Markatos, P. Vatsolaki, C. Xanthaki, The remote enqueue operation on networks of workstations, International Journal of Computing and Informatics 23 (1) (1999) 29–39.

[18] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, Active Messages: a Mechanism for Integrated Communication and Computation, in: Proceedings of the 19th annual International Symposium on Computer architecture, ISCA '92, 1992, pp. 256–266.

[19] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping, The paderborn university bsp (pub) library, Parallel Computing 29 (2) (2003) 187–207.

[20] V. Tipparaju, E. Aprá, W. Yu, J. S. Vetter, Enabling a highly-scalable global address space model for petascale computing, in: Proceedings of the 7th ACM international conference on Computing Frontiers, CF '10, 2010, pp. 207–216.

[21] J. Nieplocha, V. Tipparaju, M. Krishnan, D. K. Panda, High performance remote memory access communication: The armci approach, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 233–253.

[22] K. Denolf, M. J. G. Bekooij, J. Cockx, D. Verkest, H. Corporaal, Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations, EURASIP J. Adv. Sig. Proc. 2007.

[23] A. Benveniste, A. Bouillard, P. Caspi, A unifying view of loosely time-triggered architectures, in: Proceedings of the tenth ACM international conference on Embedded Software, EMSOFT '10, 2010, pp. 189–198.

[24] A. V. Gerbessiotis, S.-Y. Lee, Remote memory access: A case for portable, efficient and library independent parallel programming, Sci. Program. 12 (3) (2004) 169–183.