DOI 10.12694/scpe.v18i2.1287
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 2, pp. 151–160. http://www.scpe.org © 2017 SCPE

TOWARDS HPC-EMBEDDED. CASE STUDY: KALRAY AND MESSAGE-PASSING ON NOC

PEDRO VALERO-LARA,[*] EZHILMATHI KRISHNASAMY,[†] AND JOHAN JANSSON [‡]

Abstract. Today one of the most important challenges in HPC is the development of computers with a low power consumption. In this context, recently, new embedded many-core systems have emerged. One of them is Kalray. Unlike other many-core architectures, Kalray is not a co-processor (self-hosted). One interesting feature of the Kalray architecture is the Network on Chip (NoC) connection. Habitually, the communication in many-core architectures is carried out via shared memory. However, in Kalray, the communication among processing elements can also be via Message-Passing on the NoC. One of the main motivations of this work is to present the main constraints to deal with the Kalray architecture. In particular, we focused on memory management and communication. We assess the use of NoC and shared memory on Kalray. Unlike shared memory, the implementation of Message-Passing on NoC is not transparent from programmer point of view. The synchronization among processing elements and NoC is other of the challenges to deal with in the Karlay processor. Although the synchronization using Message-Passing is more complex and consuming time than using shared memory, we obtain an overall speedup close to 6 when using Message-Passing on NoC with respect to the use of shared memory. Additionally, we have measured the power consumption of both approaches. Despite of being faster, the use of NoC presents a higher power consumption with respect to the approach that exploits shared memory. This additional consumption in Watts is about a 50%. However, the reduction in time by using NoC has an important impact on the overall power consumption as well.

Key words: Karlay, Embedded Architectures, High Performance Computing, Jacobi Method, OpenMP, Power Measurements.

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction.

Advanced strategies for the efficient implementation of computationally intensive numerical methods have a strong interest in industrial and academic community. In the last decade, we have lived a spectacular growth in the use of many-core architectures for HPC applications [8, 14, 15, 9, 10]. However, the appearance of other (low-power consumption) embedded many-core architectures such as Kalray [1] has created new challenges and opportunities for performance optimization in multiple applications. In this work, we have explored some of these new opportunities towards a supercomputing on a chip era.

Kalray integrates its own OS and is not in need of a co-processor as in the case of other many-core processors [4, 1]. In Karlay, highly expensive memory transfers from host main memory to co-processor memory are not necessary, as in other architectures, such as NVIDIA GPUs [13] or Inel MIC [12]. Besides, this architecture offers the possibility to communicate each of the processing elements via a Network on Chip (NoC) connection composed by links and routers [4, 1]. Kalray has been previously used for video encoding and Monte Carlo applications [2]. However, these works lack information of how to implement these applications and what are the most efficient programming strategies and architectonic features to deal with our embedded processor. The NoCs have been recently used as a level in-between the computing cores and shared memory [5, 17, 7]. The NoCs in these systems can be configurable depending on the particular needs of the applications. However, the NoC in Kalray is completely different. In Kalray, there are two different and independent inter-connectors, one bus which connects each of the processing elements to shared memory and one NoC which connects the different processing elements (clusters) among them.

We have chosen as a test case a widely known and extended problem, that is Jacobi method [16]. The main motivation of this work is twofold. While, on one hand, this work presents the main challenges to deal with the Kalray architecture. On the other hand, we present two different approaches to implement the communication among the different processing elements of our Kalray processor, one based on using shared memory and other based on using a Network on Chip, which works as interconnection among the set of processing cores. We detail and analyze deeply each of the approaches, presenting theirs advantages and disadvantages. Moreover, we include measurements for power consumption in both approaches.

[*] Barcelona Supercomputing Center, Spain. (pedro.valero@bsc.es).

[†] Basque Center for Applied Mathematics (BCAM), Bilbao, Spain (ekrishnasamy@bcamath.org.)

[‡] Basque Center for Applied Mathematics (BCAM), Bilbao, Spain, and KTH Royal Institute of Technology, Stockholm, Sweden (jjansson@bcamath.org).

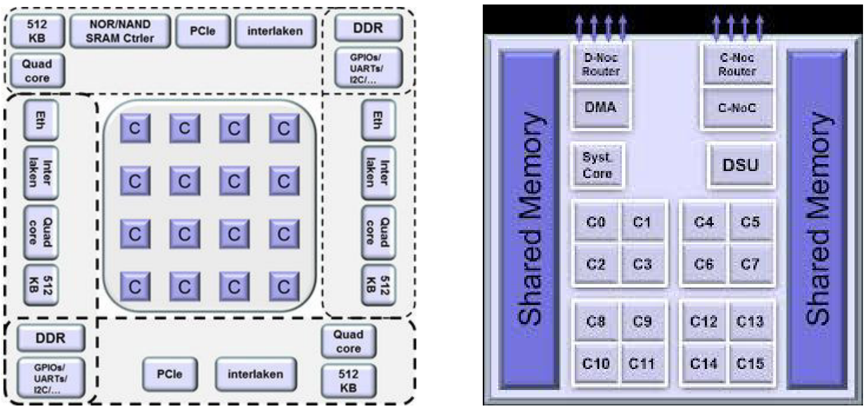

Fig. 2.1. *Kalray MPPA many-core (left) and compute cluster (righ) architecture [2].*

This paper is structured as follows. Section 2 briefly introduces the main features of the architecture at hand, Kalray. Then, we detail the techniques performed for an efficient implementation of the Jacobi method on Kalray processor in Section 3. Finally, In Section 4, it is carried out the performance analysis of the proposed techniques in terms of consuming time, speed-up and power consumption. At the end of this work, we outline some conclusions.

2. Kalray Architecture. Kalray architecture [2] is an embedded many-core processor. It integrates 288 cores on a single 28 nm CMOS chip with a low power consumption per operation. We have 256 cores divided into 16 clusters which are composed by 16+1 cores each. 4 quad-core I/O subsystems (1 core per cluster) are located at the periphery of the processing array (Figure 2.1-left). They are used as DDR controller for accessing to up to 64GB of external DDR3-1600. These subsystems control a 8-lane Gen3 PCI Express for a total peak throughput of 16GB/s full duplex. The 16 compute clusters and the 4 I/O subsystems are connected by two explicitly addressed Network on Chip (NoC) with bi-directional links, one for data and the other for control [2, 3]. NoC traffic does not interfere with the memory buses of the underlying I/O subsystem or compute cluster. The NoC is implemented following a 2-D torus topology.

The compute cluster (Figure 2.1-right) is the basic processing unit of our architecture [2]. Each cluster contains 16 processing cores (C0, C1, C2, . . . , C15 in Figure 2.1-right) and one resource management (Syst. Core in Figure 2.1-right) core, a shared memory, a direct memory access (DMA) controller, a Debug & System Unit (DSU), and two routers, one for data (D-NoC) and one for control (C-NoC). The DMA is responsible to transfer data among shared and the NoC with a total throughput of 3.2GB/s in full duplex. The shared memory compromises 2MB organized in 16 parallel banks, and with a bandwidth of 38.4 GB/s. The DSU supports the debug and diagnosis of the compute cluster.

Each processing or resource management core is a 5-way VLIW processor with two arithmetic and logic units, a multiply-accumulate & floating point unit, a load/store unit, and a branch & control unit [2]. It enables up to 800MFLOPS at 400MHz, which supposes almost 13 GFLOPS per cluster and almost 205GFLOPS in total by using the 16 clusters. These five execution units are connected to a shared register file which allows 11 reads and 4 writes per cycle. Each core is connected to two (data & instruction) separate 2-way associate caches (8KB each).

Kalray provides a software development kit, a GNU C/C++ & GDB development tool for compilation and debugging. Two programming models are currently supported. A high level programming model based on data-flow C language called $\sum$C [6], where programmers do not care about communication, only data dependencies must be expressed. The other programming model supported is a POSIX-Level programming model [4, 1]. It distributes on I/O subsystems the sub-processes to be executed on the compute clusters and pass arguments through the traditional *argc*, *argv*, and *environ* variables. Inside compute clusters, classic shared memory programming models such as POSIX threads or OpenMP pragmas are supported to exploit more than one processing core. Specific IPC takes advantage of the NoC connection. Unlike $\sum$C, the POSIX-

Algorithm 1 Jacobi OpenMP Algorithm.

```
$jacobi(A, Anew, NX, NY)$
$float\ err;$
$\#pragma\ omp\ parallel\ for$
for $int\ i = 1 \rightarrow NY - 1$ do
    for $int\ j = 1 \rightarrow NX - 1$ do
        $Anew[i * NX + j] = 0.25 * (A[i * NX + (j - 1)] + A[i * NX + (j + 1)]$
                $+ A[(i - 1) * NX + j] + A[(i + 1) * NX + j]);$
        $err = maxf(err, fabs(A_{new}[i * NX + j] - A[i * NX + j]));$
    end for
end for
$\#pragma\ omp\ parallel\ for$
for $int\ i = 1 \rightarrow NY - 1$ do
    for $int\ j = 1 \rightarrow NY - 1$ do
        $A[i * NX + j] = Anew[i * NX + j];$
    end for
end for
```

Level programming model presents more important challenges from programmer side, however it allows us to have more control on hardware and optimize both, communication and computation. In the present work, the authors have followed the programming model based on POSIX.

3. Jacobi Method Implementation on Kalray. We have chosen as test case the Jacobi method [16]. This is a good example, which allows us to study and evaluate different strategies for communication. The parallelization is implemented following a coarse-grained distribution of (adjacent) rows across all cores. This implementation is relatively straightforward using a few OpenMP pragmas on the loops that iterate over the rows of our matrix (see Algorithm 1).

One of the most important challenges in Kalray is the communication and memory management. To address the particular features of Kalray architecture, we use the Operating System called *NodeOs* [1], provided by Kalray. *NodeOs* implements the Asymmetric Multi-Processing (AMP) model. AMP takes advantage of the asymmetry found in the clusters between the Resource Management Core (RMC) and the Processing Element Cores (PEC). RMC runs the operating system (kernel and NoC routines) on the set of RM (single-core). PEC are dedicated to run user threads, one thread per PEC. PEC can also call functions, such as *syscall* that are in need of OS support, which are received and compute by RMC. When a PEC executes a *syscall* call, it sends an event and it is locked until it receives an event from the RMC. This process is necessary to know that the *syscall* has been processed. Data and parameters are exchanged using shared memory. We have two codes, one executed by RMC (*IO* code) and other (*cluster* code) executed by PECs. The work is distributed following a master/slave model that is well suited to Kalray architecture. The *IO* code is the master. It is in charge of launching the code and sending data to be computed by slaves. Finally they wait for the final results. Otherwise, the *cluster* code are the slaves. They wait for data to be computed and send results to IO cluster.

The POSIX-Level programming model of Kalray (*NodeOs*) allows us to implement the communication among different clusters in two different ways. While shared memory (accessible by all clusters) is used for the communication in the first approach (*SM*), in the second approach (*NoC*), we use channels (links) and routers. For sake of clarify, we include several algorithms in which we detail the main steps of each of the approaches. Algorithms 2 and 3 illustrate the *IO* and *cluster* pseudocodes for the *SM* approach and Algorithms 4 and 5 for the *NoC* approach respectively.

The communication is implemented by using some specific objects and functions provided by *NodeOS*. Next, we explain each of these objects and functions. The transfers from/to global/local memory are implemented via *portals*. These *portals* must be initialized using specific paths (one path per cluster) as *A_portal* in Algorithm 2. Then, they must be opened (*mppa_open*) and synchronized (*mppa_ioctl*) before transferring (*mppa_pwrite* in Algorithm 2 and *mppa_aio_read* in Algorithm 3) data from/to global/local memory. The slaves are launched from master via *mppa_spawn* which include parameters and name of the function/s to be computed by cluster/s.

Algorithm 2 *Shared Memory* I/O pseudocode.

```
$const\ char\ *cluster\_executable = "mainCLUSTER";$
$static\ float\ A[SIZE]; static\ float\ Anew[SIZE];$
int mainIO(int argc , char *argv[] )
$long\ long\ dummy = 0; long\ long\ match = -(1 << CLUSTER\_COUNT);$
$const\ char\ *root\_sync = "/mppa/sync/128:1";$
$const\ char\ *A\_portal = "/mppa/portal/"CLUSTER\_RANGE":1";$
$const\ char\ *Anew\_portal = "/mppa/portal/128:3";$
// − −OPENING_FILES − −//
$int\ root\_sync\_fd = $ mppa_open$(root\_sync, O\_RDONLY);$
$int\ A\_portal\_fd = $ mppa_open$(A\_portal, O\_WRONLY);$
$int\ Anew\_portal\_fd = $ mppa_open$(Anew\_portal, O\_RDONLY);$
// − −PREPARE_FOR_RESULT − −//
$status| = $ mppa_ioctl$(root\_sync\_fd, MPPA\_RX\_SET\_MATCH, match);$
$mppa\_aiocb\_t\ Anew\_portal\_aiocb[1] = \{$MPPA_AIOCB_INITIALIZER
                                            $(Anew\_portal\_fd,\ Anew, sizeof(Anew[0]) * SIZE)\};$
mppa_aiocb_set_trigger$(Anew\_portal\_aiocb, CLUSTER\_COUNT);$
$status| = $ mppa_aio_read$(Anew\_portal\_aiocb);$
// − −LAUNCHING_SLAVES − −//
$char\ arg0[10], arg1[10];$
$const\ char\ *argv[] = arg0, root\_sync, A\_portal, Anew\_portal, 0;$
for $int\ rank = 1 \to CLUSTER\_COUNT$ do
    $sprintf(arg0, "\%d", rank);$
    $status| = $ mppa_spawn$(rank, NULL, cluster\_executable, argv, 0);$
end for
// Wait for the cluster portals to be initialized.
$status| = $ mppa_read$(root\_sync\_fd, \&dummy, sizeof(dummy));$
// Distribute slices of array A over the clusters.
for $int\ rank = 0 \to CLUSTER\_COUNT$ do
    $status| = $ mppa_ioctl$(A\_portal\_fd, MPPA\_TX\_SET\_RX\_RANK, rank);$
    $status| = $ mppa_pwrite$(A\_portal\_fd, (A + rank * SIZE\_LOCAL) - (NX\_LOCAL * 2),$
    $sizeof(float) * SIZE\_LOCAL, 0);$
end for
// Wait for the cluster contributions to arrive in array |Anew|.
$status| = $ mppa_aio_wait$(Anew\_portal\_aiocb);$
$return\ status < 0;$
```

The communication among cluster via links (*NoC*) is implemented by using of *channel*. Similar to the use of *portals*, *channels* must be initialized using one path per channel (see *C0_to_C1_channel* in Algorithm 2).

On the other hand, the synchronization is implemented by using of *sync*. They are used to guarantee that some resources are ready to be used or cluster are ready to start computing (for instance, see *mppa_ioctl* in Algorithm 2, 3, 4 and 5).

In order to minimize the number of transfers among main and local memory (*SM* approach) as well as among clusters through links (*NoC* approach), the matrix is divided into rectangular sub-blocks (Figures 3.1 and 3.2). In particular, the distribution of the workload and communication implemented in the *NoC* approach avoid multi-level routing, connecting each of the cluster with its adjacent clusters via a direct link.

Although, the ghost cells strategy is usually used for communication in distributed memory systems [11], we have used this strategy in Kalray processor to avoid race conditions among each of the sub-blocks assigned to each clusters. The use of ghost cells consists of replicating the borders of all immediate neighbors blocks. These ghost cells are not updated locally, but provide stencil values when updating the borders of local blocks. Every ghost cell is a duplicate of a piece of memory located in neighbors nodes. To clarify, Figures 3.1 and 3.2 illustrate a simple scheme for our interpretation of the ghost cell strategy applied to both approaches, *SM* and *NoC*, respectively.

Algorithm 3 *Shared Memory* CLUSTER pseudocode.

```
$\textbf{int mainCLUSTER}(\textbf{int argc}, \textbf{char} \ * \textbf{argv}[])$
$int\ i, j, status, rank = atoi(argv[0]);$
$const\ char\ \ * root\_sync = argv[1], *A\_portal = argv[2], *Anew\_portal = argv[3];$
$float\ A[SIZE\_LOCAL], Anew[SIZE\_LOCAL]; long\ long\ slice\_offset;$
$slice\_offset = sizeof(float) * (CHUNK * NX\_LOCAL +$
                                  $((rank - 1) * (CHUNK - 1) * NX\_LOCAL));$
$//$ Each cluster contributes a different bit to the root_sync mask.
$long\ long\ mask = (long\ long)1 << rank;$
$//- -$ OPENING_PORTAL $- -//$
$int\ root\_sync\_fd = \textbf{mppa\_open}(root\_sync, O\_WRONLY);$
$int\ A\_portal\_fd = \textbf{mppa\_open}(A\_portal, O\_RDONLY);$
$int\ Anew\_portal\_fd = \textbf{mppa\_open}(Anew\_portal, O\_WRONLY);$
$//- -$ PREPARE_FOR_INPUT $- -//$
$mppa\_aiocb\_tA\_portal\_aiocb[1] =$
        $\textbf{MPPA\_AIOCB\_INITIALIZER}(A\_portal\_fd, A, sizeof(A));$
$status| = \textbf{mppa\_aio\_read}(A\_portal\_aiocb);$
$//- -$ UNLOCK_MASTER $- -//$
$status| = \textbf{mppa\_write}(root\_sync\_fd, \&mask, sizeof(mask));$
$//$ Wait for notification of remote writes to local arrays $|\mathbf{A}|$.
$status| = \textbf{mppa\_aio\_wait}(A\_portal\_aiocb);$
$//- -$ JACOBIANCOMPUTE $- -//$
$jacobi(A, Anew, NX\_LOCAL, NY\_LOCAL);$
$//$ Contribute back local array Anew into the portal of master array Anew.
$status| = \textbf{mppa\_pwrite}(Anew\_portal\_fd, \&Anew[NX\_LOCAL],$
                                  $sizeof(Anew) - sizeof(float) * 2 * NX\_LOCAL, slice\_offset);$
$mppa\_exit((status < 0)); return\ 0;$
```

Figure 3.1 graphically illustrates the strategy followed by the *SM* approach. It consists of dividing the matrix into equal blocks which are sent from main memory to local memory. To avoid race condition, each of the blocks includes 2 additional rows (gray and white rows in Figure 3.1) which correspond to the upper and lower adjacent rows of the block. These additional rows work as ghost-cell, which are only used in local memory. The blocks transferred from local memory to global memory (Figure 3.1-right) do not include these additional rows (ghost rows).

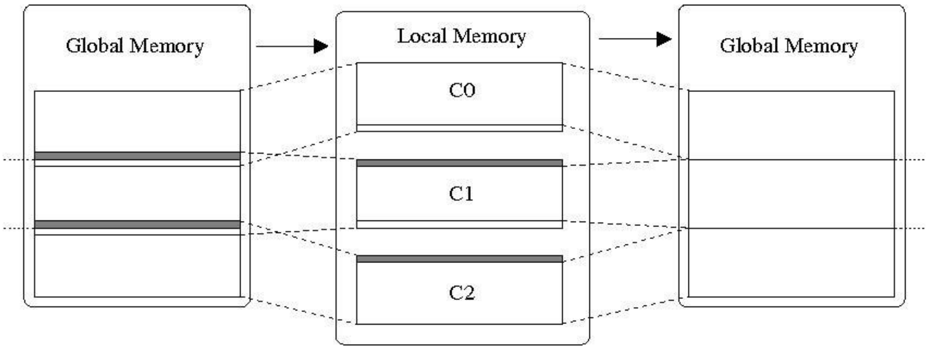

FIG. 3.1. *Master (Global Memory) ↔ Slave (Local Memory) Communication.*

Otherwise the communication among global and local memory is not necessary in the *NoC* approach. The master (IO code) is only used for synchronizing. The synchronization is necessary at the beginning and at the end of each *Master* code. I/O core and the rest of cores in each of the clusters must be also synchronized. In particular the synchronization between IO core and computing cores ($I/O- > C1$ *sync* section in Algorithm 5)

Algorithm 4 *NoC* I/O pseudocode.

```
$const\ char\ *global\_sync = "/mppa/sync/128:1";$
$const\ char\ *IO\_to\_C0\_sync = "/mppa/sync/0:2"; \dots$
$const\ char\ *C0\_to\_C1\_channel = "/mppa/channel/1:1/0:1"; \dots$
$static\ const\ char\ *exe[CLUSTER\_COUNT] = \{"mainCLUSTER0",$
                $"mainCLUSTER1", \dots\};$
int mainIO$(int\ argc, const\ char\ *argv[])$
// Global sync.
$int\ ret, global\_sync\_fd =$ mppa_open$(global\_sync, O\_RDONLY);$
$long\ long\ match = -(1 << CLUSTER\_COUNT);$
mppa_ioctl$(global\_sync\_fd, MPPA\_RX\_SET\_MATCH, match));$
// − −IO_TO_C#_SYNC − −//
$int\ IO\_to\_C0\_sync\_fd =$ mppa_open$(IO\_to\_C0\_sync, O\_WRONLY);$
$int\ IO\_to\_C1\_sync\_fd =$ mppa_open$(IO\_to\_C1\_sync, O\_WRONLY); \dots$
// − −LAUNCHING_SLAVES − −//
for $int\ i = 0 \rightarrow CLUSTER\_COUNT$ do
    mppa_spawn$(i, NULL, exe[i], argv, 0);$
end for
// Wait for other clusters to be ready.
mppa_read$(global\_sync\_fd, NULL, 8);$
// Write into I/O− > C# sync to unlock C# cluster.
$mask = 1;$ mppa_write$(IO\_to\_C0\_sync\_fd, \&mask, sizeof(mask));$
mppa_write$(IO\_to\_C1\_sync\_fd, \&mask, sizeof(mask)); \dots$
// − −WAITING TO THE END OF CLUTERS EXECUTION − −//
for $int\ i = 0 \rightarrow CLUSTER\_COUNT$ do
    ret = mppa_waitpid$(i, \&status, 0);$ mppa_exit$(ret);$
end for
```

is necessary to guarantee that there are no cluster reading into channels before the corresponding cluster has opened the channel. After computing the Jacobi method in each of the clusters, some rows of the local blocks must be transferred to/from adjacent clusters. The first row computed (white upper row C1 in Figure 3.2) must be transferred to the upper adjacent cluster (C0) to be stored in the last row. Also, the last row computed (gray lower row C1 in Figure 3.2) must be transfered to the lower adjacent cluster (C2) to be stored in the first row. This pattern must be carried out in all clusters except the first and last clusters where a lower number of data-transfers is necessary.

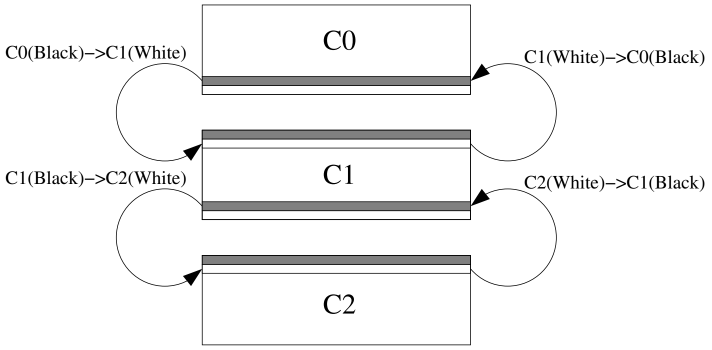

FIG. 3.2. *Pipeline (Bus) Communication.*

4. Performance Study. In this section, we analyze deeply both approaches, *SM* and *NoC*, focusing on communication, synchronization and computing separately. In order to find/focus on the performance of both approaches, we have used a relatively small problem which can be fully stored in local memory.

Algorithm 5 *NoC* CLUSTER pseudocode

```
int mainCLUSTER1(int argc, char *argv[])
float A[SIZE_LOCAL], Anew[SIZE_LOCAL];
// Open all the resources needed for transfers.
 //Global sync.
int global_sync_fd = mppa_open(global_sync, O_WRONLY);
 // C1− > C2 channel.
int channel0_fd = mppa_open(C1_to_C2_channel, O_WRONLY);
 // C2− > C1 channel.
int channel1_fd = mppa_open(C2_to_C1_channel, O_RDONLY);
 // C1− > C0 channel.
int channel2_fd = mppa_open(C1_to_C0_channel, O_WRONLY);
 // C0− > C1 channel.
int channel3_fd = mppa_open(C0_to_C1_channel, O_RDONLY);
// I/O− > C1 sync.
int IO_to_C1_sync_fd = mppa_open(IO_to_C1_sync, O_RDONLY);
long long match = −(1 << 1/* We sync only with I/O cluster */);
mppa_ioctl(IO_to_C1_sync_fd, MPPA_RX_SET_MATCH, match)
 // Write into global sync to unlock I/O cluster.
long long mask = 1 << mppa_getpid();
mppa_write(global_sync_fd, &mask, sizeof(mask))
 // − −WAIT_FOR_IO_TO_C1_SYNC − −//
mppa_read(IO_to_C1_sync_fd, NULL, 8);
// − −CLUSTERS COMMUNICATION − −//
// Write data for cluster 0.
mppa_write(channel0_fd, &A[NX_LOCAL * (NY_LOCAL − 2)], sizeof(float) * NX_LOCAL);
// Read data from C0.
mppa_read(channel1_fd, A, sizeof(float) * NX_LOCAL);
// Read data from C2.
mppa_write(channel2_fd, &A[NX_LOCAL], sizeof(float) * NX_LOCAL);
// Write data for cluster 2.
mppa_read(channel3_fd, &A[NX_LOCAL * (NY_LOCAL − 1)], sizeof(float) * NX_LOCAL);
mppa_exit(0);
```

Next we present the commands used to compile and launch both approaches: Compiling lines:
k1 − gcc −O3 −std = c99 −mos = rtems io.c −o io_app −lmppaipc
k1 − gcc −O3 −std = c99 −fopenmp −mos = nodeos cluster.c −o cluster −lmppaipc
k1 − create − multibinary − − cluster cluster − − boot = io_app −T multibin
Launching line:
k1 − jtag − runner − − multibinary multibin − − exec − multibin = IODDR0 : io_app

The communication among I/O and computing cores in the *NoC* approach is more complex and it is in need of a higher number of synchronizations. This causes a higher execution time with respect to the *SM* approach, being almost 2.5× bigger (Figures 4.1 and 4.2). Note that we use a different vertical scaling in each of the graphics illustrated in Figures 4.1 and 4.2. Despite of the overhead caused by a higher number of synchronizations, the use of the NoC interconnection makes the *NoC* approach (Figure 4.2) about 55× faster than the *SM* approach.

As expected the time consumed for computing the Jacobi method is equivalent in both approaches. The time consumed by synchronization, communication and computing in the *NoC* approach is more balanced than in the *SM* approach. This can be beneficial for future improvements, such as asynchronous communication.

Finally, we analyse the performance in terms of GFLOPS. First, we compute the theoretical FLOPS for the Jacobi computation. The variant used in this study performs six flops per update (Algorithm 1). Therefore, the theoretical FLOPS is equal to the elements of our matrix multiplied by six.

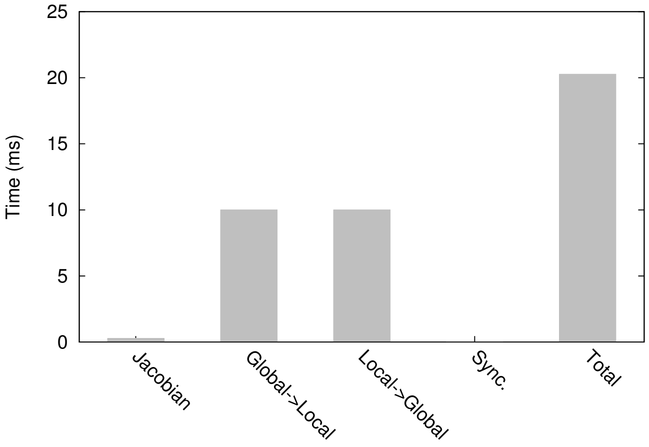

Fig. 4.1. *Time consumption for the* SM *approach.*

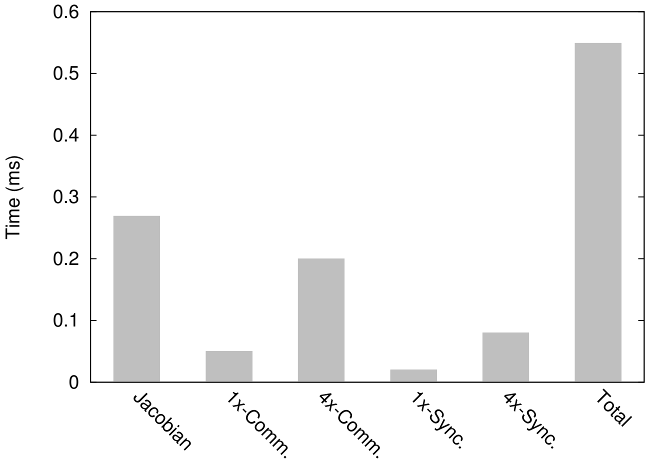

Fig. 4.2. *Time consumption for the* NoC *approach.*

In order to evaluate the overhead of each of the strategies, first, we show the GFLOPS achieved by the Jacobi computation without the influence of the synchronization and communication (see *Jacobian* in Figure 4.3). It achieves almost the peak of performance of our platform (*GFLOPS-Peak* in Figure 4.3). The computation of the Jacobian method is exactly the same in both approaches (*SM* and *NoC*). Next, we include the overhead of the communication. Although both approaches present a fall in performance when taking into account the time consumed by the communication, the fall shown by the *NoC* approach is not so dramatic as the overhead suffered by the *SM* approach (Figure 4.3).

The software development kit provided by Kalray allow us to measure the power consumption of our applications. This is done via this command:

$$k1-power\ --k1-jtag-runner\ --multibinary\ multibin\ --exec-multibin = IODDR0 : io\_app$$

Executing our binary using *k1-power* we obtain the power achieved in terms of Watts. The average power achieved by the *NoC* approach is about $8.508W$, while the *SM* approach achieves an average of $5.778W$ in every execution. This is almost a 50% more power when executing the *NoC* approach. However, the reduction in

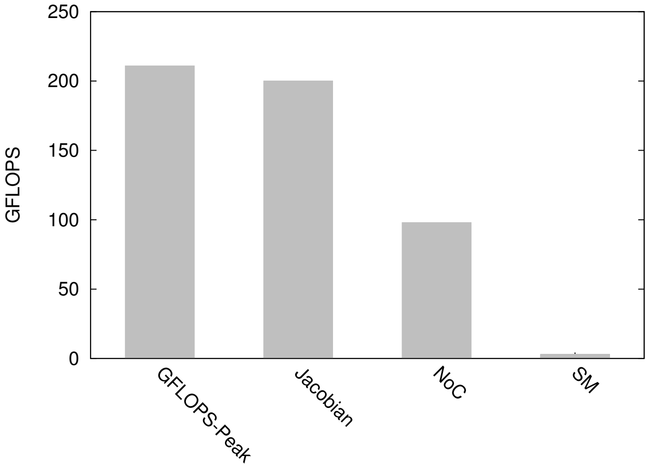

Fig. 4.3. *GFLOPS achieved by both approaches.*

execution time obtained by the NoC approach has an important impact on the overall power consumed. Joules are computed by following the next expression:

$$Joules = Watts \times Time$$

obtaining an overall consumption about $0.0047J$ and $0.16J$ for the NoC and the SM approaches respectively. This is a 96% less of power consumed by the NoC approach.

5. Conclusions and Future Work. Embedded many-core architectures such as Kalray have emerged as a new HPC platform to deal with the problem of the excessive power consumption.

In this work, we have presented two different approaches to implement the communication among the processing elements of the Kalray architecture. Both approaches implement a ghost-cell strategy to avoid race conditions among the different blocks assigned to each of the processing elements (clusters). This strategy has been adapted to the particular features of our embedded processor and approaches, *SM* and *NoC*, to minimize the number of transfers.

Although the communication via shared memory is more habitual and easier to implement on many-core architectures, the particular features of the Kalray architecture, in particular the communication via Message-Passing on NoC connection, offers a much faster alternative. Although, the use of NoC consumes more power, the reduction in time makes this approach more efficient in terms of power consumption.

We plan to investigate other problems and more efficient strategies for memory management and data distribution, such as the overlapping of communication and computing via asynchronous transfers. In particular, the NoC approach could take advantage of the asynchronous communication as the time consumed by its major steps is balanced.

Acknowledgments. This research has been supported by EU-FET grant EUNISON 308874, the Basque Excellence Research Center (BERC 2014-2017) program by the Basque Government, the projects founded by the Spanish Ministry of Economy and Competitiveness (MINECO): BCAM Severo Ochoa accreditation SEV-2013-0323, MTM2013-40824 and TIN2015-65316-P. We acknowledge Research Center in Real-TIme & Embedded Computing Systes - CISTER for the provided resources.

REFERENCES

[1] S. A. Kalray. *Mppa accesscore posix progamming reference manual.* 2013.

[2] B. Dupont de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. Guironnet de Massas, F. Jacquet, S. Jones, N. Morey Chaisemartin, F. Riss, and T. Strudel. *A clustered manycore processor architecture for embedded*

and accelerated applications. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, pages 1–6, 2013.

[3] B. Dupont de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. *Guaranteed services of the noc of a manycore processor.* In Proceedings of the 2014 International Workshop on Network on Chip Architectures, NoCArc '14, Cambridge, United Kingdom, December 13-14, 2014, pages 11–16, 2014.

[4] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. *Time-critical computing on a single-chip massively parallel processor.* In Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014, pages 1–6, 2014.

[5] M. Dev Gomony, B. Akesson, and K. Goossens. *Coupling tdm noc and dram controller for cost and performance optimization of real-time systems.* In 2014 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1–6, March 2014.

[6] T. Goubier, R. Sirdey, S. Louise, and V. David. Σc*: A programming model and language for embedded manycores.* In Algorithms and Architectures for Parallel Processing - 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I, pages 385–394, 2011.

[7] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. *Exploration of distributed shared memory architectures for noc-based multiprocessors.* Journal of Systems Architecture, 53(10):719 – 732, 2007. Embedded Computer Systems: Architectures, Modeling, and Simulation.

[8] P. Valero, J.L. Sánchez, D. Cazorla, and E. Arias. *A gpu-based implementation of the mrf algorithm in itk package.* The Journal of Supercomputing, 58(3):403–410, 2011.

[9] P. Valero-Lara. *Accelerating solid–fluid interaction based on the immersed boundary method on multicore and gpu architectures.* The Journal of Supercomputing, 70(2):799–815, 2014.

[10] P. Valero-Lara, F.D. Igual, M. Prieto-Matías, A. Pinelli, and J. Favier. *Accelerating fluid–solid simulations (lattice-boltzmann & immersed-boundary) on heterogeneous architectures.* Journal of Computational Science, 10:249–261, 2015.

[11] P. Valero-Lara and J. Jansson. *LBM-HPC an open-source tool for fluid simulations. case study: Unified parallel C (UPC-PGAS).* In Cluster Computing (CLUSTER), 2015 IEEE International Conference on, pages 318–321. IEEE, 2015.

[12] P. Valero-Lara, P. Nookala, F.L. Pelayo, J. Jansson, S. Dimitropoulos, and I. Raicu. *Many-task computing on many-core architectures.* Scalable Computing: Practice and Experience, 17(1):32–46, 2016.

[13] P. Valero-Lara and F.L. Pelayo. *Full-overlapped concurrent kernels.* In Architecture of Computing Systems. Proceedings, ARCS 2015-The 28th International Conference on, pages 1–8. VDE, 2015.

[14] P. Valero-Lara, A. Pinelli, J. Favier, and M. Prieto-Matias. *Block tridiagonal solvers on heterogeneous architectures.* In Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12, pages 609–616, Washington, DC, USA, 2012. IEEE Computer Society.

[15] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias. *Fast finite difference poisson solvers on heterogeneous architectures.* Computer Physics Communications, 185(4):1265 – 1272, 2014.

[16] D. M. Young. *Iterative solution of large linear systems.* 2003. Unabridged republication of the 1971 edition [Academic Press, New York-London, MR 305568].

[17] H. Zhao, O. Jang, W. Ding, Y. Zhang, M. Kandemir, and M.J. Irwin. *A hybrid noc design for cache coherence optimization for chip multiprocessors.* In Proceedings of the 49th Annual Design Automation Conference, DAC '12, pages 834–842, New York, NY, USA, 2012. ACM.

Edited by: Dana Petcu
Received: Dec 16, 2016
Accepted: Mar 26, 2017