An Efficient MPI Implementation for Multi-Core Neuromorphic Platforms

(Article begins on next page)

19 April 2024

An Efficient MPI Implementation for Multi-Core Neuromorphic Platforms

Francesco Barchi, Gianvito Urgese, Enrico Macii and Andrea Acquaviva

Politecnico di Torino, Torino, Italy, Email: andrea.acquaviva@polito.it

Abstract—**Multicore neuromorphic platforms come with a custom library for efficient development of neural network simulations. While these architectures are mainly focused on real-time biological network simulation using detailed neuron models, their application to a wider range of computational tasks is increasing. The reason is their effective support for parallel computation characterised by an intensive communication among processing nodes and their inherent energy efficiency. However, to unlock the full potential of these architectures for a wide range of applications, a library support for a more general computational model has to be developed. This work focuses on the implementation of a standard MPI interface for parallel programming of neuromorphic multicore architectures. The MPI library has been developed on top of the SpiNNaker multicore neuromorphic platform, featuring a toroid interconnect and packet support for multicast communication. The proposed MPI implementation has been evaluated using an N-body simulation kernel, showing very good efficiency and suggesting that the considered neuromorphic platform with our MPI library is very promising for communication-intensive applications.**

I. INTRODUCTION

In the last decade massively parallel bio-inspired systems have been designed for emulating brain activity in real time by running spiking neural networks [1, 2, 3, 4, 5]. At the same time, because of their inherent efficient support for inter-chip communication, these systems are under study for accelerating communication intensive applications involved in other computational physics and biology applications.

In this work we consider the SpiNNaker neuromorphic platform, a massively parallel architecture exploiting a toroidal inter-chip communication network. The platform considered in this work entails 48 chips each one featuring 18 ARM cores and a custom router. Since SpiNNaker has mainly been developed to run brain simulations, it does not natively support a general purpose programming model, like the Message Passing Interface (MPI). In this work, we explore the potential of this type of architectures to accelerate communication intensive applications by exploiting the MPI library that we completely developed and optimised from scratch leveraging the SpiNNaker interconnect support.

We evaluated the developed library using an N-body simulation kernel, typically used in computational biology tasks such as molecular dynamics. Results suggest that the SpiNNaker architecture with our MPI implementation is promising for tasks where communication is prevalent.

The rest of the paper is organised as follows. Section II provides an overview of the neuromorphic platform. Section III describes the design and implementation of the MPI library. Section IV explains the tests performed in order to validate the developed implementation. Section V provides the summary and the final evaluations.
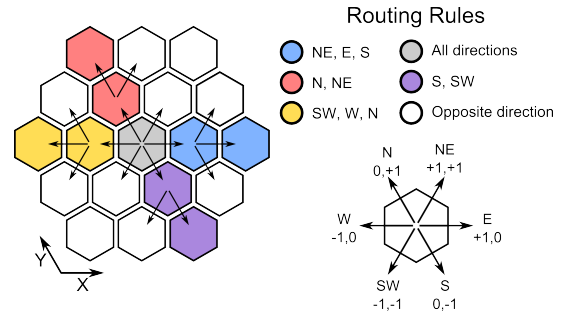

Fig. 1: The broadcast routing rule

II. BACKGROUND

SpiNNaker is a neuromorphic platform built such a hexagonal shaped triangular network of SpiNNaker chips. SpiNNaker architecture has been designed as a low-power globally asynchronous and locally synchronous system (GALS) [6]. The basic building block of the system is the SpiNN-5, a PCB with 48 SpiNNaker chips and an Ethernet interface connected to the first chip for linking the board with the external devices. Each chip embeds 18 ARM968 processors, a custom router, 128 kB of SRAM, and 128 MB of SDRAM and a NoC. The applications can be executed on 16 cores that are called Application Processors (APs), one core, called Monitor Processor (MP), is used for chip management and communication operations. The embedded router is capable of transmitting packets of 72 bit, 40 bit of header and 32 bit of optional payload, of two different types: Point to Point (P2P) and Multicast (MC). Payloads greater than 32 bit can be transmitted using the SpiNNaker Data Protocol (SDP), running on the monitor processor, that splits a message into several P2P packets.

The software is divided into layers: the SpiNNaker Application Runtime Kernel (SARK) that exposes several APIs for the hardware functionalities and the Spin1API for supporting the Event Driven Programming (EDP) [7]. Applications are built on top of these two libraries and loaded in the SpiNNaker cores using the Spinnaker Command Protocol (SCP) [8]. Users can also use in their applications the Application Command Protocol (ACP) [9].

Some applications, like the Spiking Neural Network (SNN) simulations, need to be configured by high-level software modules running on the host computer, like PACMAN and GHOST [10, 11]. In the same way, the MPI implementation developed during this work needs a host library.

III. METHOD

To implement a high-level message passaging interface we need some low-level functionalities: i) A synchronisation system between all computing nodes. ii) A Middle-level interface for handling the SpiNNaker native multicast communication.
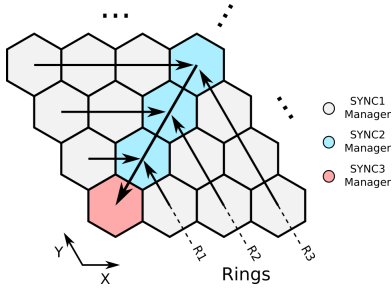
Fig. 2: The sync levels in a SpiNN-5 Board.

Fig. 3: ACP remote command flow.

iii) A high-level interface to read/write data between nodes.

Therefore, we developed two auxiliary libraries (Spin2API and SpinACP) for supporting the MPI library itself (SpinMPI). Spin2API is an extension of Spin1API, the standard application library for Spinnaker. It provides an interface to use the multicast message system in a broadcast way and implements a synchronisation system. Whereas, SpinACP implements an Application Command Protocol (ACP), exploited to send and to receive commands between SpiNNaker nodes and from/to the Host. SpinACP provides some facilities and built-in commands for sharing memory objects (referred in this paper as *Memory Entities*). The SpinMPI is built over these two layers, in particular, we implemented the receiving buffer as an *ACP Memory Entity*. Both SpinACP and SpinMPI have an on-host runtime written in Python (ACP-Runtime and MPI-Runtime).

The goal of this section is to describe the implementation of an MPI programming abstraction that exploits the SpiNNaker event-driven programming model and the on-board interconnection structure using the three designed libraries.

A. Spin2API

Spin2API provides facilities for configuring, sending and receiving in broadcast more than 32 bit of data. The payload is fragmented and carried into a flow of MC packets. Each MC packet has a header composed of four fields: i) The packet source, ii) The communication control flags, iii) The synchronisation flags, iv) The channel information field. We will refer to this packet format as SPIN2-MC.

The policies for generating routing rules for a particular source are summarised in Figure 1. We define four rules according to the relative position of the router compared to the packet source: i) The router of the source chip sends its packets to all its six neighbours. ii) The router of all chips on the x^+ axis spread packets to ¡E, NE, S¿, while the router of all chips on the x^- axis spread packets to ¡W, SW, N¿. iii) The router of all chips on the y^+ axis spread the packets to ¡N, NE¿, while the router of all chips on the y^- axis spread packets to ¡S, SW¿. iv) The router of all chips that do not belong to the categories listed above spread packets in the opposite direction from which they receive them. It is possible to cover the whole area with one routing rule for each source chip.

The synchronisation function uses the SPIN2-MC format and implements a multilevel synchronisation infrastructure composed of signaller and synchroniser nodes, as shown in Figure 2. During a synchronisation phase inside a $\texttt{level}_x$, a signaller sends a $\texttt{SYNC}_x$ message to the synchroniser of the level. Once the synchroniser collects all the expected signals it becomes an $x + 1$ signaller and sends a $\texttt{SYNC}_{x+1}$ message to the synchroniser of the next level. At the end, the synchroniser of the last level sends in broadcast a $\texttt{SYNC}_{\text{UNLOCK}}$ message. Inside the SpiNN-5 three levels are used: $\texttt{level}_1$
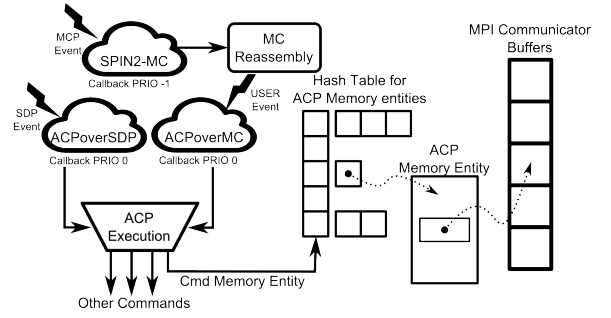
the *SpiNNaker Chip*; $\texttt{level}_2$ the *Ring*, a set of chips at the same distance from the Root Chip; $\texttt{level}_3$ the *Board* itself.

B. SpinACP

In this work we extended and simplified the ACP that was initially born as a system to read/write predefined variables through commands sent over the SpiNNaker network [9]. We removed from ACP some features like fragmentation and composition including them in supplementary high-level libraries. The ACP library can be summarized into three functional parts: i) Command management ii) Command spreading over the network, ACPoverSDP or ACPoverMC. ii) MemoryEntities management and built-in commands.

It is possible to create commands and use them as callback functions, but the most useful commands are the built-in commands for manage *Memory Entities* described as structs and collected in hash tables.

The command flow is shown in Figure 3. A single execution node is supplied from two ACP interpreter nodes: one for commands sent over SDP, and one for commands sent over SPIN2-MC. When a SPIN2-MC packet arrive a packet reaches a node it is processed and the payload extracted. According to the packet source address and communication control the payload is added to a buffer for the reconstruction. The SpinACP library handles a circular buffer that it is used to save and re-compose multiple ACPoverMC messages. Once the message is re-composed, it is consumed by a lower priority callback that interprets and executes the command.

C. SpinMPI

The MPI reference implementation, OpenMPI, provides two components: i) mpicc is a wrapper of a C compiler that provides the environment variables to include the library files and to link the object files. ii) mpirun is the OpenMPI runtime environment that launch and manages the execution of the application over multiple nodes.

Differently, a SpinMPI application is compiled with spinnaker_tools, the compiler tool-chain provided with the SpiNNaker board. The launcher is the Python package MPI-Runtime that loads the compiled application on a set of processors called MPI-Context. Moreover, it initializes the application with the context info such as MPI Rank and MPI Communicator via the ACP.

The MPI-Context is identified by the number of rings involved and by the number of processors used for each chip: Context : $(\text{RingID}_{\text{MAX}}, \text{VID}_{\text{MAX}})$. Each Ring contains a variable number of chips. For example, Ring 0 contains the chip (0,0), Ring 1 the chips (0,1), (1,1), (1,0) and so on, as

shown in Figure 2. Each processor is identified by a VID[1]. For example, for parallelizing a program on 32 processors we can choose a context of C(1,8), C(3,2) or C(5,1), each of them describes a set of 32 processors. When the context is defined, the application is loaded on SpiNNaker.

An MPI application starts with the `MPI_Init(...)` function, whereas on SpiNNaker we need to call the `MPI_Spinn(...)` that initialize the libraries to implement the event driven programming and insert in the scheduler queue the `mpi_main` function that contain the MPI application. A code template is reported below.

```
#include "mpi.h"
...
void c_main() {
  MPI_Spinn(mpi_main);
}
void mpi_main(uint arg1, uint arg2) {
  MPI_Init(NULL, NULL);
  ...
  MPI_Finalize();
}
```

The `MPI_Spinn(...)` is performed in three steps: i) In the Callback registration step, the Spin2API and SpinACP[2] libraries register several callbacks to manage the incoming packets: SDP from the Monitor Processors and the MC from the router. In this way, only incoming SDP on port 7 and incoming SPIN2-MC packets are dispatched to the ACP. ii) The second step is to initialise the support for multicast connections. Spin2API register the routing rules for each possible source and SpinACP register an internal callback that will be used for reconstructing the `ACPoverMC` messages. iii) In the last step, the `mpi_main` handler that contains the application code is scheduled on a low priority event queue before to leave the control to the event scheduler that is then forced to execute the `mpi_main` function. We detach the MPI application from `c_main()`[3] and include it in a standalone function.

Inside the `mpi_main`, the MPI library is initialised through the `MPI_Init(...)` function. The initialization procedure is divided into three phases: i) Initialize ACP `Memory Entities` that are recorded and will be used to receive from the `MPI-Runtime` the processor Rank and the coordinates (x, y, vid) of all processors involved in the `MPI-Context`. ii) Receive the context information from the runtime and configuration variables for the SPIN2-MC synchronisation feature. iii) Finally, each processor waits for a signal from the `MPI-Runtime`. Once the `MPI-Context` has verified that all the processors have been properly configured, it sends a signal and the `MPI_Init(...)` function ends.

MPI offers two types of communications: Point to Point (1to1) and collective (1to*, *to1, *to*). The 1to1 communications have three properties: i) Blocking/Non-Blocking[4], ii)

		Data Multiplicity			
		1 to 1	1 to *	* to 1	* to *
Group Multiplicity	1 to *	MPI_Bcast	-	MPI_Scatter	-
	* to 1	-	MPI_Gather	-	-
	* to *	-	MPI_Allgather	-	MPI_Alltoall

TABLE I: MPI collective functions.

Synchronous/Asynchronous[5], iii) Buffered/Unbuffered[6]. The specification defines MPI_Send and MPI_Recv as blocking functions and MPI_Isend and MPI_Irecv as non-blocking functions. These functions can be also Synchronous or Buffered. The user can force the use of synchronous functions via MPI_Ssend and MPI_Issend. On SpiNNaker, the implementation of MPI_Send is blocking, synchronous, and non-buffered, while MPI_Recv is blocking and buffered. MPI 1to1 communication functions use ACPoverSDP to send the write command for modifying the content of the `Memory Entities` used as communication buffer in the target processor. Specifically, the MPI_Recv buffer is an ACP `Memory Entities` of 60 Bytes. For sending more than 60 Byte it is necessary to fragment the data into groups to be sent individually. Each fragment is followed by a confirmation signal from the receiving processor, thus obtaining a synchronous communication.

The collective communications differ depending on the type of operation and the multiplicity of addresser/addressee. In particular, there are three types of operations: synchronism (described in section III-A), data reduction (to be implemented), and data movement (described in the following). The data movement functions can be classified according to the number of nodes involved in the data transmission (group multiplicity), and the number of data units inside the receiving and sending buffers (data multiplicity), see Table I. The group multiplicity of 1To* identifies an operation where one sender node spreads data to all nodes involved in the communication (including the sending node). Instead, the data multiplicity of 1To* identifies an operation where the buffer size of the sender nodes (one or many) is the same size of the item to be transmitted, while the buffer size of the receiver nodes (one or many) is equal to the item size multiplied with the number of nodes involved in the communication. The SpinMPI library provide MPI_Bcast and MPI_Allgather that are functions for replicating data on the nodes using the SpiNNaker-native multicast/broadcast transmission system. This type of functions can exploit SPIN2-MC protocol and are implemented using the ACPoverMC functions.

For example, the MPI_Allgather is implemented as a simple linear cycle where each processor, in turns, sends an ACP to write command to all others processors. This is possible only if all involved processors are synchronised. To synchronise all MPI contexts, it is possible to use the synchronisation feature provided in the Spin2API.

IV. BENCHMARK

The NBody simulation consists of N particles each with a position in a D-dimensional space $\vec{x}^p \in \mathbb{R}^D$ and with a mass m_p. Integration of motion equations with Velocity Verlet is discretized by step equal to $\tau = \Delta t$ and consists of three

---

[1]Each processor has a physical ID (location on the die) and a virtual ID assigned when the machine is powered up.

[2]Spin2API callbacks are registered directly on the Spin1API events, while the SpinACP callbacks are registered on Spin2API.

[3]The standard entry point for SpiNNaker application

[4]A blocking function is released only when the data buffer to be sent can be modified, otherwise non-blocking functions are released immediately and the submission is postponed or delegated to a competing thread.

[5]Synchronous functions (send only) require a receiving acknowledgement from the receiver before considering the communication successful

[6]The message before being sent and/or received is copied into a system buffer

equations to be solved for each particle p: i) Equation 1: position update, ii) Equation 2: calculation of forces due to gravitational interaction iii) Equation 3: velocity update.

$$\vec{x}_{t+1}^{p} = \vec{x}_t^{p} + \vec{v}_t^{p}\tau + \frac{1}{2}\vec{a}_t^{p}\tau^2 \tag{1}$$

$$\vec{a}_{t+1}^{p} = \frac{1}{m_p}\vec{F}_{t+1}^{p} = \frac{1}{m_p}\sum_i \vec{F}_{t+1}^{i,p}$$
$$= \frac{1}{m_p}\sum_i -G\frac{m_i m_p}{|\vec{x}_{t+1}^{p} - \vec{x}_{t+1}^{i}|^3}(\vec{x}_{t+1}^{p} - \vec{x}_{t+1}^{i}) \tag{2}$$

$$\vec{v}_{t+1}^{p} = \vec{v}_t^{p} + \frac{1}{2}(\vec{a}_t^{p} + \vec{a}_{t+1}^{p})\tau \tag{3}$$

The NBody simulation was parallelized with MPI by distributing the particles, equally, on each computation node. Each node will have to update only the positions and speed of its particles, bringing complexity from $O(N^2)$ to $O(\frac{N}{P}N)$ where P is the number of processors.

To calculate the force exerted on a particle p each node must know the position of each particle of the system. At each iteration a particle position update step is then performed by the function `MPI_Allgather(···)`. All calculations were executed in fixed points. As seen in Section III-B, we implemented `MPI_Allgather(···)` with a broadcast transfer mediated by MC packets, the complexity of communication passes from $O(P^2)$ to $O(P)$ as data replication is done in parallel by architecture routers. We evaluate the performances of the implementation in terms of speed-up χ_n and efficiency η_n as the number of used computational nodes increases. $\chi_n = \frac{\Delta T_1}{\Delta T_n}$ $\eta_n = \frac{\chi_n}{n}$
We performed two series of simulations, one with 1 k particles and another with 2 k particles, for evaluating the scalability of the MPI implementation from 1 to 240 processors and for analysing the impact on the efficiency when the number of particles to be calculated for each node is increasing.

As shown in Figure 4 the results show good scalability performances, comparable with state-of-the-art implementations on parallel computing platforms [12]. The speed-up is directly proportional to the number of cores until 100 nodes, and reach 194 x when 240 nodes are used to simulate 2k particles (156 x for 1 k particles).

The efficiency stays above 90% with 64 processors for the 1 k simulation and up to 128 processors for the 2 k simulation. With 240 processors, we obtained an efficiency of 65% for the simulation with 1 k particles and more than 80% for the simulation with 2 k particles. With this results we can speculate and hypothesize that is convenient to distribute the problem on additional processors.

Results show that the considered neuromorphic architecture with the proposed MPI library is a promising solution for accelerating communication intensive applications.

V. CONCLUSION

In this work, we presented an implementation of the MPI paradigm on the SpiNNaker neuromorphic platform. The MPI-SpiNNaker implementation required the development of multiple abstraction-level software grouped in three new libraries. i) Spin2API implements broadcast connection/synchronisation methods and a hash table ADT. ii) SpinACP implements memory entities and network command functionalities. iii) SpinMPI implements the MPI on SpiNNaker. We implemented an N-Body simulation to benchmark and evaluate the performance of the board in the execution of an MPI parallel application. In this simulation, 2 k particles were simulated on 240 processors with a speed-up of 194 x and an efficiency of 80% when compared to the serial version running on a single CPU.

REFERENCES

[1] Michael Beyeler et al. "Efficient spiking neural network model of pattern motion selectivity in visual cortex". In: *Neuroinformatics* 12.3 (2014), pp. 435–454.

[2] Liam P Maguire et al. "Challenges for large-scale implementations of spiking neural networks on FPGAs". In: *Neurocomputing* 71.1 (2007), pp. 13–29.

[3] Andreas Grübl. "VLSI implementation of a spiking neural network". In: (2007).

[4] Paul A Merolla et al. "A million spiking-neuron integrated circuit with a scalable communication network and interface". In: *Science* 345.6197 (2014), pp. 668–673.

[5] Steve B Furber et al. "Overview of the spinnaker system architecture". In: *Computers, IEEE Transactions on* 62.12 (2013), pp. 2454–2467.

[6] Evangelos Stromatias et al. "Power analysis of large-scale, real-time neural networks on SpiNNaker". In: *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE. 2013, pp. 1–8.

[7] Andrew D. Brown et al. "SpiNNaker Programming Model". In: *IEEE Transactions on Computers* 64.6 (June 2015), pp. 1769–1782.

[8] Steve Temple. *AppNote 5 - Spinnaker Command Protocol (SCP) Specification*. 2011. URL: https://spinnaker.cs.manchester.ac.uk/.

[9] Alessandro Siino et al. "Data and commands communication protocol for neuromorphic platform configuration". In: *MCSoC, 2016 IEEE 10th International Symposium on*. IEEE. 2016, pp. 23–30.

[10] Steve B Furber et al. "The spinnaker project". In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665.

[11] Gianvito Urgese et al. "Optimizing Network Traffic for Spiking Neural Network Simulations on Densely Interconnected Many-Core Neuromorphic Platforms". In: *IEEE Transactions on Emerging Topics in Computing* (2016).

[12] Roberto Capuzzo-Dolcetta, Mario Spera, and D Punzo. "A fully parallel, high precision, N-body code running on hybrid computing platforms". In: *J. of Computational Physics* 236 (2013), pp. 580–593.
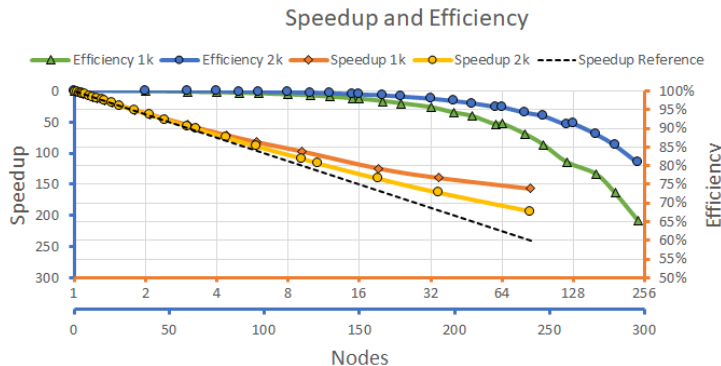
Fig. 4: Speedup on blue axes and efficiency on orange axes measured for two simulation sizes, 1 k and 2 k particles.