Journal of
Low Power Electronics and Applications

Article

PageRank Implemented with the MPI Paradigm Running on a Many-Core Neuromorphic Platform

Evelina Forno [1,*], Alessandro Salvato [2], Enrico Macii [2] and Gianvito Urgese [2,*]

[1] Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy
[2] Interuniversity Department of Regional and Urban Studies and Planning, Politecnico di Torino, 10129 Torino, Italy; alessandro.salvato@polito.it (A.S.); enrico.macii@polito.it (E.M.)
[*] Correspondence: evelina.forno@polito.it (E.F.); gianvito.urgese@polito.it (G.U.)

Abstract: SpiNNaker is a neuromorphic hardware platform, especially designed for the simulation of Spiking Neural Networks (SNNs). To this end, the platform features massively parallel computation and an efficient communication infrastructure based on the transmission of small packets. The effectiveness of SpiNNaker in the parallel execution of the PageRank (PR) algorithm has been tested by the realization of a custom SNN implementation. In this work, we propose a PageRank implementation fully realized with the MPI programming paradigm ported to the SpiNNaker platform. We compare the scalability of the proposed program with the equivalent SNN implementation, and we leverage the characteristics of the PageRank algorithm to benchmark our implementation of MPI on SpiNNaker when faced with massive communication requirements. Experimental results show that the algorithm exhibits favorable scaling for a mid-sized execution context, while highlighting that the performance of MPI-PageRank on SpiNNaker is bounded by memory size and speed limitations on the current version of the hardware.

Keywords: benchmarking neuromorphic HW; neuromorphic platform; SpiNNaker; spinMPI; MPI for neuromorphic HW; PageRank

check for updates

Citation: Forno, E.; Salvato, A.; Macii, E.; Urgese, G. PageRank Implemented with the MPI Paradigm Running on a Many-Core Neuromorphic Platform. *J. Low Power Electron. Appl.* **2021**, *11*, 25. https://doi.org/10.3390/jlpea11020025

Academic Editor: Vishal Saxena

Received: 29 April 2021
Accepted: 25 May 2021
Published: 28 May 2021

1. Introduction

Presently, new techniques and technologies require a large amount of information to produce meaningful results. Computer Science theory does not grow in the same manner as hardware infrastructure, which is the main bottleneck [1,2]. In the past 10 years, new architectures have been designed, such as heterogeneous platforms integrating multicore systems with GPUs and hardware accelerators; however, the economic and environmental cost of hardware is also growing, as the increase in computational power for platforms such as GPUs comes hand in hand with exponential increases in power consumption.

To circumvent the growing limitations met by traditional architectures, a number of alternative solutions have been proposed. One such example are neuromorphic systems, which aim at overcoming current hardware constraints using a variety of design innovations inspired by biology and neuroscience [3]. Neuromorphic systems focus on the implementation of computational models based on Spiking Neural Networks (SNNs), a special type of neural network exhibiting asynchronous and sparse behavior. This event-driven approach, inspired by the human nervous system, ensures very low power consumption while allowing for efficient data exchange among several independent computational units [4].

SpiNNaker is a fully digital neuromorphic architecture integrating massive-throughput multicore and distributed-memory systems with a thick array of interconnections, building a homogeneous lattice to link all computational units. Previous work [5,6] proved that the SpiNNaker architecture can outperform classic multicore platforms when dealing with massively parallel computation, guaranteeing a better scalability for increasing problem sizes. Although many neuromorphic architectures are focused on a specific implementation of the spiking neuron model, the ARM-based SpiNNaker platform allows the simulation of a wide range of neuron models as well as fully supporting general-purpose applications [7–9]. To improve the accessibility and performance of general-purpose applications, we expanded the SpiNNaker software stack adding support for the standard MPI (Message-Passing Interface) protocol [10] with the SpinMPI library, which fully exploits the brain-inspired connectivity mesh of the platform for efficient inter-chip communication [11]. Our goal is to prove by on-board experiments that MPI allows for easy and effective implementations of general-purpose code and to compare its results to those obtained by the SNN framework.

The PageRank algorithm [12], consisting of iterative computation on a densely connected graph with heavy communication requirements, appears to be a natural fit for architecture configurations such as multicore neuromorphic hardware. The graph morphology is such that scalability issues may occur: both from the point of view of execution time and because of the number of packets exchanged through the whole network. In fact, for any iteration, each node shares its own data with all the neighboring nodes; the elaboration of a large and densely connected graph, where each node acts independently, requires an architecture that allows for a high number of parallel processes and efficient interprocess communication: SpiNNaker fully satisfies these requirements.

In their investigation, Blin et al. [5] set out to demonstrate that for massively parallel problems that also employ the exchange of many small messages within workers, neuromorphic hardware shows better scalability than traditional architectures. In this work, we propose to confirm this analysis (by introducing MPI libraries especially designed for SpiNNaker), as well as to present said MPI functionalities as a valuable tool with the potential to be used alongside Spiking Neural Networks on the SpiNNaker neuromorphic hardware. A secondary outcome of the presented analysis is the clear potential advantages that a mesh of computing elements reproducing the SpiNNaker interconnection topology can bring for running algorithms which are similar to PageRank.

2. Background

2.1. PageRank Algorithm

PageRank is a popular graph-based algorithm, used to provide a quantitative classification on a set of objects. It was originally designed by Sergey Brin and Larry Page to classify web pages based on the number of links a single page has, relative to all others [12]. When PageRank was proposed in 1998, the systems used by internet search engines to index web pages were an increasingly relevant topic, as the size of the World Wide Web grew in an exponential manner. Search engines needed to handle millions of pages, but the amount was becoming unsustainable: the PageRank model aimed at providing efficient scalability for the WWW environment, by generating high-quality classification based on hyper-textual and keyword-based search, rather than on statically indexed directories. The classification relies on assigning to each page a value, or rank, representing its "popularity" throughout the web, based on the number of the page's incoming links. A page's rank is calculated by the formula [13] in Equation (1):

$$PR(A) = \frac{1-d}{N} + d \cdot \left(\sum_{k=1}^{n} \frac{PR(P_k)}{C(P_k)} \right) \tag{1}$$

where:

- $PR(A)$ is the PageRank value of target page A
- N is the total number of pages in the domain
- n is the number of pages such that a link exists involving both P_k and A
- $C(P_k)$ is the total number of outgoing links for P_k
- d is the so-called *damping factor*, used for tuning; its typical value is 0.85.

Both $PR()$ and d represent probabilities. $PR(P_k)$ is equivalent to the probability that a user might randomly query a given page P_k; d represents the chance a user may stop following links and start a totally new and uncorrelated search. Conversely, $1 - d$ is the complementary probability that the hypothetical user does proceed to a linked page. To produce the PageRank value for a generic object A belonging to a graph, each vertex in the graph needs to know the PageRank value of every other vertex.

2.2. Neuromorphic Platforms

SpiNNaker is a neuromorphic platform. Its purpose is to simulate the behavior of the animal brain by means of an event-driven mesh network of biologically plausible neuron models called Spiking Neural Network (SNN), where each computational unit is considered to be a group of neurons. Other neuromorphic platforms under active development include:

- *BrainScaleS* [14]: the project goal is to provide a hardware platform emulating biological neurons at higher-than-real-time speed. It is realized with transistors working above threshold. Synapses and neurons can be externally configured through high-end FPGAs, and analogue synapses are delivered using wafer-scale integration. The target of this architecture is the simulation of SNNs in accelerated time, so that a simulation that would normally require months or years can be executed in minutes/hours.
- *Dynap-SEL* [15]: a VLSI chip, its name stands for Dynamic Asynchronous Processor Scalable and Learning. It counts 5 neuromorphic cores and neurons are connected via a multi-router hierarchical organization following a mesh schema. Two different grids are laid out: 16x16 and 4x4. This architecture has been developed targeting edge computing applications belonging to the IoT and Industry 4.0 domains.
- *Loihi* [16]: designed by Intel in 2017. It is a self-learning neuromorphic research chip, composed of 128 cores, counting around 130.000 neurons. The whole architecture follows a digital implementation, exploiting an asynchronous design to reduce power consumption. The architecture has potential applications as a SNN-based coprocessor in heterogeneous SoCs.

Adding to the above-described platforms, the Spiking Neural Network Architecture (*SpiNNaker*) [17] is designed for real-time SNN simulations following an event-driven computational approach, approximating the one found in the human brain [18]. The main difference from other architectures consists of the fact that SpiNNaker does not rely on VLSI customization at wafer or transistor level: the platform's nodes consist of hundreds of general-purpose processors from ARM family. Therefore, SpiNNaker can natively support all sorts of C programs compiled for the ARM architecture.

2.3. SpiNNaker Architecture and SW Stack

The SpiNNaker platform has been released in two form factors. The *Spin5* board counts 48 chips, while the *Spin3* embeds 4 chips. A SpiNNaker chip is equipped with 18 ARM processors each, a custom 6-link router, a System NoC and a 128 MB SDRAM [18]. The chip is considered the basic block of the architecture; its architecture is described in Figure 1, which shows the four chips available in a *Spin3*. To guarantee low power consumption, internal clocks run at 200 MHz by default. Moreover, each microprocessor contains two tightly coupled memories for data and instructions: the DTCM and the ITCM. When using the legacy low-level software components available in SpiNNTools [19], cores are grouped in the following manner: 1 covers the role of Monitor Processor (MP), 16 are considered Application Processors (AP), while the last one is available in the case of hardware failures. It is up to the Monitor Processor to run the low-level SARK operating system; the Application Processors execute the user application.

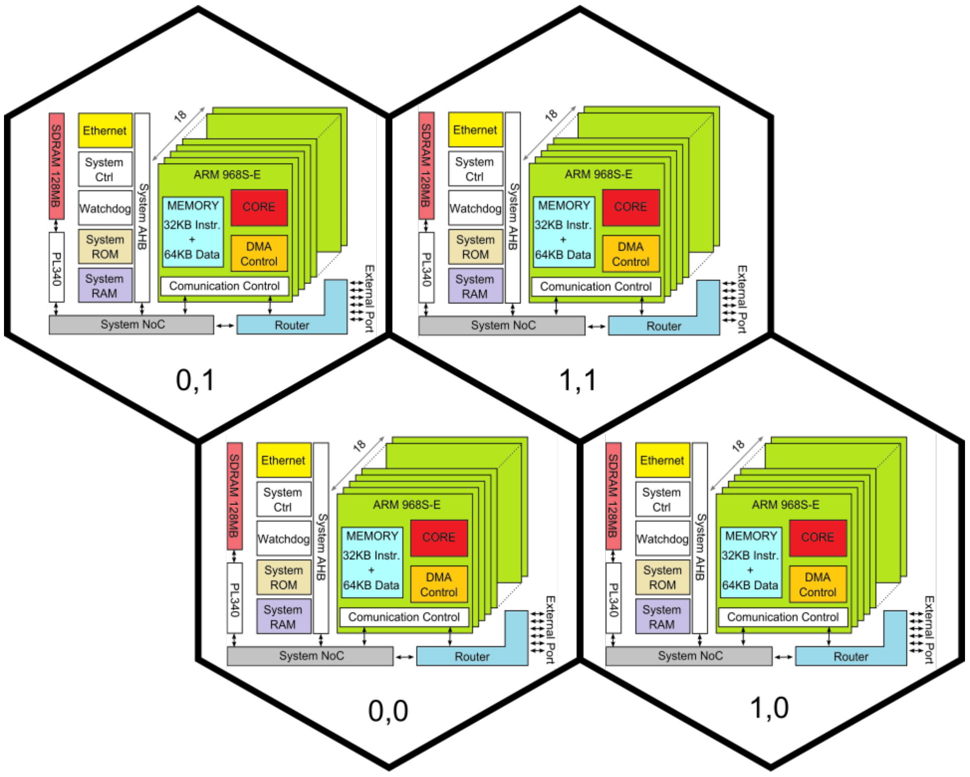

Figure 1. The Spin3 architecture.

The physical implementation of communication channels is a crucial point of the architecture. These channels may interest either two cores in the same chip, or two cores from different chips. The former is the simplest case, as "close" communication happens in a synchronous manner, exploiting a 128 MB Synchronous DRAM. In the latter case, communication is asynchronous, taking inspiration from mechanisms in the biological cortex. Packets are driven by means of a 1024-lines embedded router, one on each SpiNNaker chip. Being small CAMs, embedded routers have a short latency (∼0.1 µs per hop) [9]. The custom design of the router, despite limitations on the synchronous transmission of packets [20,21], allows transmission of two operative packet types: Multicast (MC) and Point to Point (P2P). Routers allow for easy transmission (and re-trasmission) of 72-bit packets; for longer messages, it is necessary to exploit the platform's low-level APIs and encapsulate them into datagrams. Such APIs also offer methods for packet reconstruction: the SpiNNaker Datagram Protocol (SDP) is the software stack layer managing large-packet communication up to 256 bytes [17]. The Monitor Processor is tasked with running facilities for the SDP.

Regardless of the type of data, SpiNNaker comes with three main communication profiles:

- P2P (*Point-to-Point*): a core transmits, through the monitor processor, to another core placed on the same chip or on a different chip.
- *Multicast*: a single core transmits simultaneously to a subset of cores, placed on the connected chips.
- *Broadcast*: a single core transmits simultaneously to every other core on the board.

Since the base network is designed as a mesh, any chip can open a communication channel (broadcast, multicast or P2P) or work as a forwarding spot to ensure connectivity between two chips not directly connected to each other.

The methods and experiments discussed in this paper are enabled by a two-fold environment: the Host PC side, running Python modules [7], and SpiNNaker, running a software stack fully written in C and Assembly languages [19]. On this core has been built the SpinMPI library [11], a porting of the MPI paradigm for SpiNNaker, which aims to provide a high-level interface to the user to easily control communication among physical cores. This high-level interface is the Application Command Framework (ACF), relying on long datagrams to pack application commands as a Remote Procedure Call (RPC) [22]. On the hardware side, the Multicast Communication Middleware defines each packet's format depending on the communication profile (unicast or broadcast).

2.4. Previous PageRank Implementation on SpiNNaker Using SpyNNaker

Blin et al. [5] have proposed a PageRank implementation exploiting the SNN framework for SpiNNaker (*SNN-PR*). The implementation is based on the *sPyNNaker* software libraries, [7], which provide high-level utilities for developers to interface with SpiNNaker. In *SNN-PR*, *sPyNNaker* is mainly used to map the "web page" objects onto SpiNNaker neurons. To that end, a page's rank is modeled as the membrane potential of a neuron. This modeling is very powerful since it is based on the actual low-level behavior of a spiking neural network (SNN), therefore allowing exploitation of the existing simulation infrastructure on the SpiNNaker board. The actual neuron implementation is a modification of the standard *Leaky Integrate-and-Fire (LIF)* model, providing both the transient function for the PageRank application and a message processing algorithm. Though *SNN-PR* endeavors to implement a synchronous algorithm, the underlying SpiNNaker libraries do not provide support for synchronization between cores, as synchronization is not normally required in SNN simulation. Locally, *SNN-PR* uses a semaphore within each core, which synchronizes computation of its vertices. Globally, cores iterate asynchronously; therefore, a buffer mechanism is introduced to ensure that incoming messages are processed at the correct timestep.

SNN-PR shows good scaling thanks to its efficient SNN-based implementation, which fully exploits the characteristics of the SpiNNaker platform. However, the complexity of such a custom implementation is quite high, since a new neuron model must be written as well as ancillary utilities to handle message passing and asynchronous buffering. Additionally, underlying limitations in the SNN libraries allow each core to manage a maximum of 255 vertices, each corresponding to a spiking neuron. Finally, due to the high volume of packets circulating in the system, *SNN-PR* also requires the machine timestep to be increased to 25 ms; this allows the routers and cores enough time to process incoming messages and avoid simulation failures due to packet loss.

2.5. SpinMPI

In this work, we employ a library providing support for MPI (*Message-Passing Interface*) communication and synchronization primitives on SpiNNaker, called SpinMPI [11]. The MPI paradigm provides a synchronization barrier function as well as message-passing primitives, which can be used to handle synchronous communication on a platform featuring computational units with distributed memories (such as SpiNNaker). Moreover, SpinMPI provides specifications for communicator implementation. A communicator is an interface collecting methods to manage synchronous communication among differently structured groups of units; the same physical network may host several MPI communicators. The current SpinMPI release implements point-to-point and broadcast communicators. Within the MPI paradigm, each process is labeled with a unique identifier, or rank (not to be confused with a vertex's rank in PageRank). The processes participating in a communicator are identified by their rank: for instance, a point-to-point communicator involves two processes, one triggering an *MPI Send*, the other enabling an *MPI Receive*; on the other hand, the broadcast communicator contains every process in the system. The *MPI Barrier* is used to synchronize all processes. At the end of execution, results and metadata are stored into dedicated reports, downloadable directly from SpiNNaker cores memory.

SpinMPI provides synchronization mechanisms to ensure correctness in multi-process computations. Although point-to-point communication does not require synchronization, both multicast and broadcast need it, in order to ensure that messages are received by every process in the communicator. Within the SpinMPI communication logic, the board's chips are divided into logic subregions: chips are assigned to concentric ring-shaped layers by a hierarchical policy, depending on the distance of a node from the axis origin, corresponding to the chip (0, 0). The chip (0, 0) manages Ethernet communications with the host or other Spin5 boards; this lone chip constitutes ring 0. Its neighboring chips—(1, 0), (1, 1) and (0, 1)—compose ring 1; the other rings are built as depicted in Figure 2.

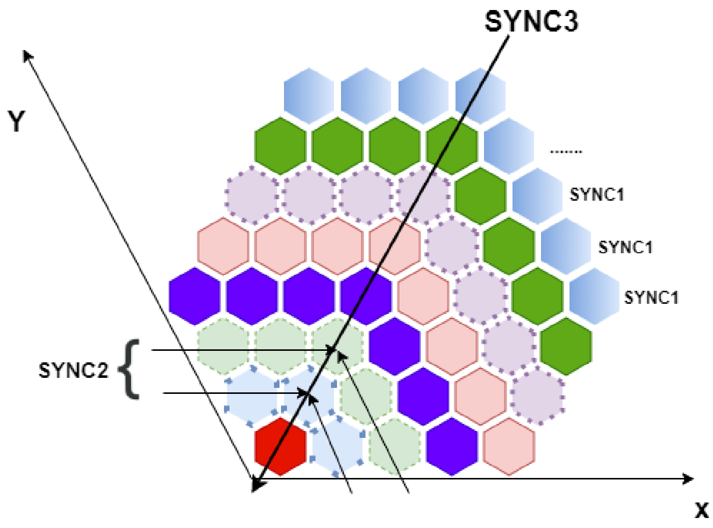

Figure 2. The synchronization rings on the Spin5 board with 48 chips.

Broadcast synchronization relies on a 3-level hierarchy. Each level is controlled by one or more *managers*, which are in charge of collecting all synchronization packets from its level. Then, the manager generates a synchronization message addressed to the upper level (Figure 2). The hierarchy levels are:

1. *Chip level*: a *SYNC1* packet is sent to all cores within a SpiNNaker chip. Once all synchronization packets have been collected, a *SYNC2* packet is prepared and sent to the upper level.
2. *Ring level*: chips with the same distance from chip (0, 0) constitute a ring. The chips labeled with $(x, y)|x = y$ are the *SYNC2* managers, responsible for collecting the synchronization packets of the group. Each ring master knows how many *SYNC1* packets should be produced; once they have received all the expected packets, they produce a *SYNC3*.
3. *Board level*: all level 2 managers send *SYNC3* packets to the level 3 manager, i.e., chip (0, 0). Once all level 2 managers have sent their packet, the level 3 manager sends a *SYNCunlock*, i.e., an ACK packet, over MPI Broadcast. This concludes the synchronization phase.

3. Methods

3.1. Implementation of PageRank with MPI

The computation of PageRank for a given blob of pages is not complex over a single iteration; however, a parallelization framework is often required due to the large size of the typical web graph. Since any node's rank depends on the rank of its neighbors, each worker's computation is not independent: a synchronized data exchange step is required at the end of every iteration. The input to our MPI-PageRank (*MPI-PR*) implementation is a binary file containing the list of edges in the graph. An edge is represented by a tuple of two integers *(Source, Destination)*, each representing a node's ID.

The program can be divided in two steps, outlined in Figure 3: the configuration step (A) and the PageRank loop (B). During the configuration step (A), the MPI worker with rank 0 is designed as the MPI Master. This core is tasked with retrieving the edges list from the filesystem and transmitting it to the other MPI workers; every worker core receives its own copy of the whole graph description. This decision was taken foreseeing the porting to SpiNNaker, which is a distributed-memory platform. The transmission of the problem data is expressed by a single MPI_Bcast call, and the segmentation of the graph data into packets is fully managed by the underlying SpinMPI library. At the end of this transmission, the core previously acting as the MPI Master resumes work as a regular MPI worker.

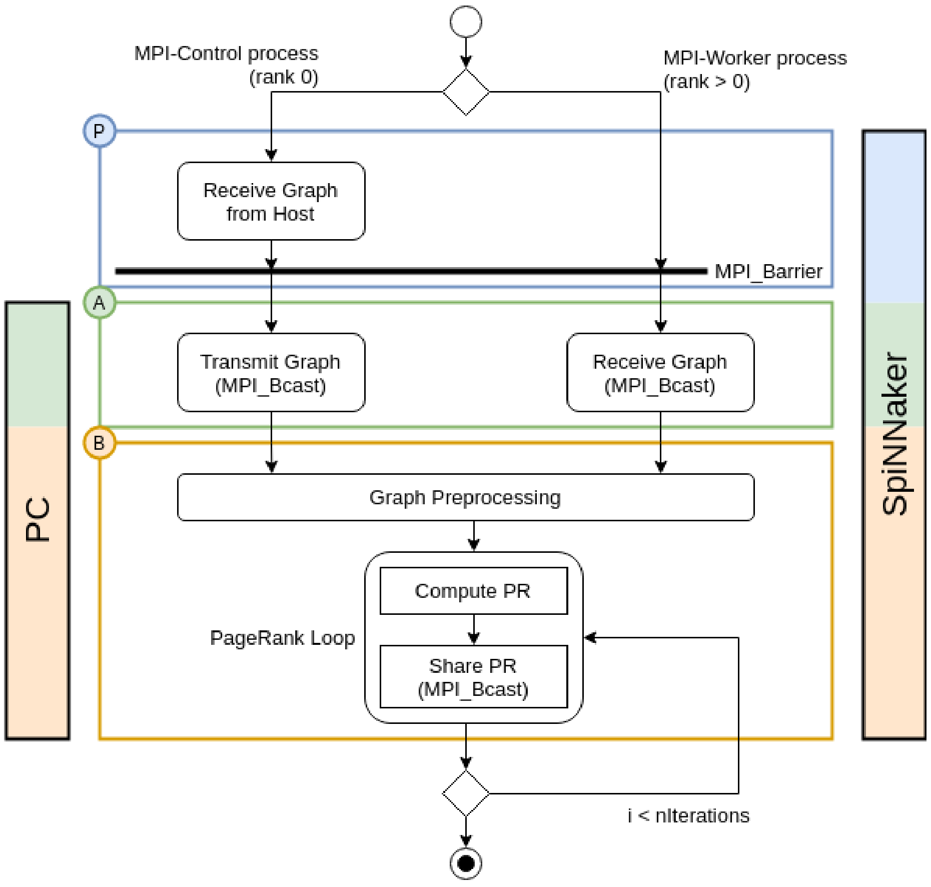

Figure 3. Flowchart of the implementation of *MPI-PR* on a general-purpose architecture and on SpiNNaker. Step A performs the configuration, step B performs the PageRank calculation; step P is a preliminary phase required to transfer the problem data to the SpiNNaker board.

Once all MPI workers have received the edges list, each performs a graph preprocessing step. Each worker is assigned a subgraph of size $k = n_{vertices}/n_{workers}$; for each node in the subgraph, the worker scans the edge list, builds the list of incoming links (represented as the ID of the source vertex), and counts the outgoing links of the node. Thus, each worker prepares the minimal data structures needed to perform PageRank calculation for each node, and the preprocessing effort is divided equally among the MPI workers.

During the PageRank loop (B), each worker updates the rank for all its assigned vertices, as in Equation (1). Then, each worker takes turns sending its new ranks to every other worker: this is also accomplished with a single MPI_Bcast instruction. In order to reduce computational costs, each worker sends its nodes's ranks already divided by the number of their own outgoing links [23]. Loop (B) is repeated until convergence or for a fixed number of steps. At the end of computation, each worker outputs the list of PageRank values for its assigned vertices.

3.2. *Adaptation of PageRank with MPI for SpiNNaker*

The MPI implementation of PageRank described in Section 3.1 is perfectly suitable to run on any MPI-enabled machine, including traditional PCs. In order to execute the same program on SpiNNaker, an additional phase (P) is added to the process: in this phase the data are transferred from a host computer to the SpiNNaker board.

The SpinMPI Python library allows the host to initialize the MPI Runtime for SpiNNaker and configure the size of the MPI Context, defined as the number of chips and cores to be used during computation: the size of the MPI Context corresponds to the number of available MPI workers. Details about the host-board communication and configuration process are reported in [24]. Finally, the MPI Runtime loads the application binaries to the board and triggers execution of the code.

Once the application is started, the host proceeds to write the problem data directly to the memory of the MPI worker of rank 0, corresponding to processor (0, 0, 1). In order to exploit as much as possible the memory available on SpiNNaker, we chose to represent each node ID as a 16-bit unsigned integer, allowing for a maximum of 2^{16} = 65,536 nodes in a graph.

All MPI workers are forced to wait on an MPI Barrier until this operation is concluded. After MPI worker 0 has received the problem data, phase (A) can begin; computation continues as described in Section 3.1.

4. Results and Discussion

In this section, we report the results of tests aimed at measuring the performance of the SpiNNaker system running our MPI implementation of the PageRank algorithm.

4.1. Comparison with Previous Versions of PageRank on SpiNNaker

In this section, we compare the performance of our MPI-based implementation of PageRank (*MPI-PR*) on SpiNNaker with the implementation presented in [5] (*SNN-PR*). We set up our experiments in the same way as described in the cited paper: a runtime parameter specifies how many vertices should be mapped to each worker, and each test executes 25 iterations of the PageRank loop.

In Figure 4, we compare the execution time of *MPI-PR* with that of *SNN-PR* on a fixed-size graph ($|V| = 255, |E| = 2550$) with a varying number of cores. It is worth noting that in the original work, the size of 255 was chosen because it represented the maximum number of vertices per core allowed by the PyNN framework; however, since the MPI framework imposes no such limitations, the number of vertices that can be processed is bounded only by the available memory. For a graph this size, we obtain the best performance with a single worker, where the time cost of MPI communication is never incurred. However, even as MPI is activated with the involvement of multiple workers, we obtain faster computation times than *SNN-PR*, up to 12 cores. We can also observe that the computation time with MPI improves only up to 8 workers: after this mark, the cost of MPI communication becomes more significant than the time saved by multithreading the computation, and using multiple cores becomes less advantageous than in the *SNN-PR* version. The dashed line in the graph represent the time spent by workers in the broadcast step: we can see that indeed the cost of communication grows with the number of cores at a higher rate than the cost of the PR computation step shrinks. Furthermore, in this experiment, mapping the MPI workers to the same chip or spreading them over 4 chips makes little difference in the execution time.

There is a sudden dip in computational time when going from 15 cores to 16; we can observe from the graph that this speedup is due to a reduction in the Broadcast time. This is due to the memory allocation for the PR rank array, which is read and written to multiple times during step (B). When more workers are added, the problem is split into smaller subgraphs, reducing the space required for each worker to represent the problem; at 16 workers, this leaves enough room to allocate the whole PR rank array to DTCM, making the Broadcast communications in step (B) faster.

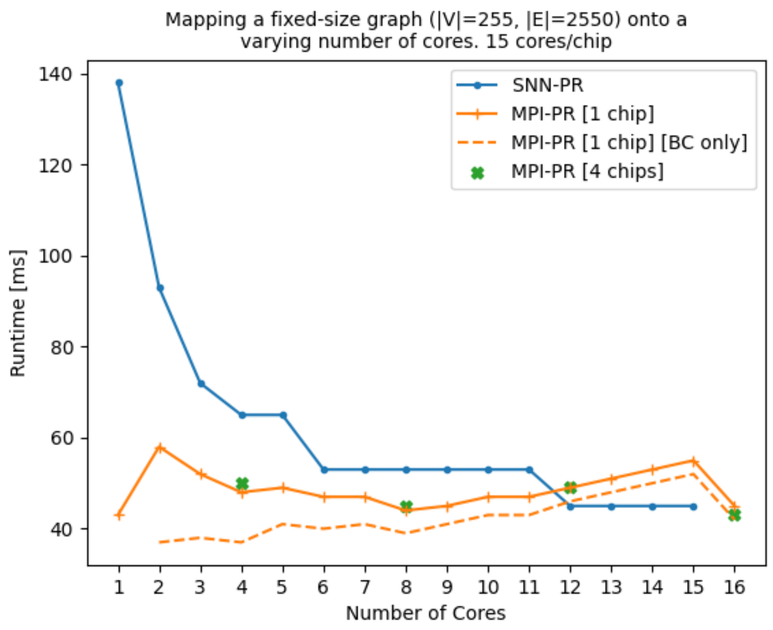

Figure 4. Execution time for *SNN-PR* and *MPI-PR* on a graph of fixed size, using only one SpiNNaker chip.

Figure 5 shows the behavior of *SNN-PR* and *MPI-PR* on a larger graph, distributed among cores using up to 4 SpiNNaker chips. We observe again a similar behavior to the one observed above; MPI reaches its best result at 10 cores, where the subgraphs are small enough that all the problem data fit in the cores' DTCM. At the same time, this is the point where the calculation/communication cost has the best tradeoff. However, due to the increased communication costs of MPI, overall *MPI-PR* does not scale as well as *SNN-PR* on this particular graph.

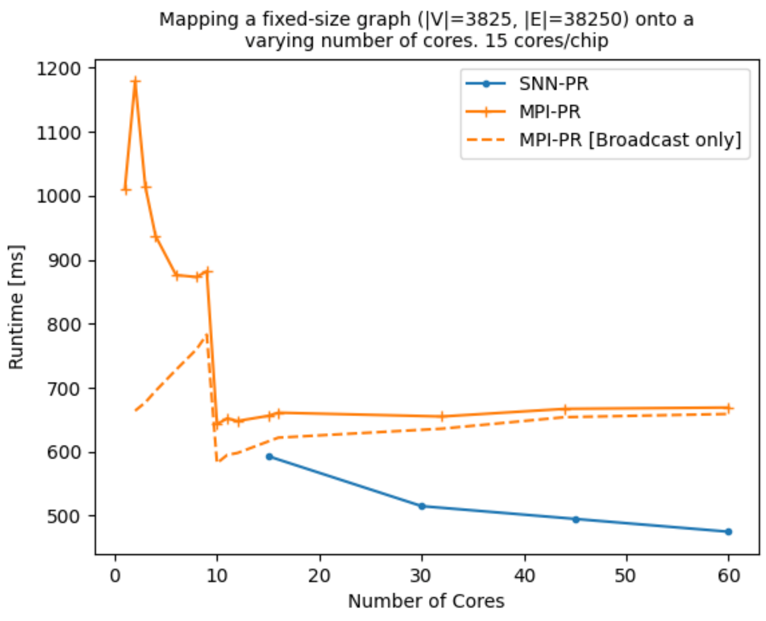

Figure 5. Execution time for *SNN-PR* and *MPI-PR* on a graph of fixed size, using up to 4 SpiNNaker chips.

As a final comparison in Figure 6 we compare the scalability of *MPI-PR* versus the *SNN-PR* and traditional multicore implementations analyzed in [5]. Results are presented as the normalized execution time relative to the single-core execution time, corresponding to the smallest graph. Both SpiNNaker implementations present a smoother and more favorable scaling, reflecting the efficiency of the custom toroidal-shaped, triangular-mesh communication network featured in the SpiNNaker many-core architecture. Overall, however, *MPI-PR* presents better scaling even than the PyNN implementation; when using 15 cores, *MPI-PR* scales about 1.75× with respect to *SNN-PR*.

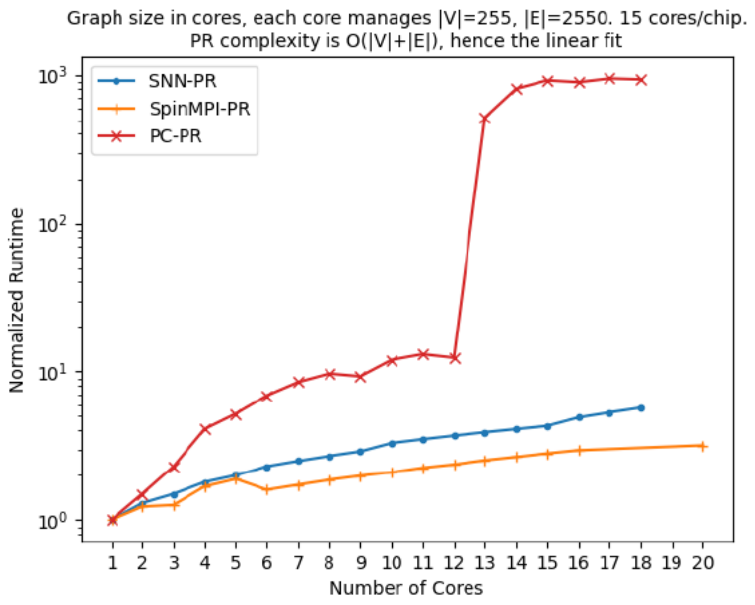

Figure 6. PageRank scalability of 3 different implementations: *SNN-PR* and *MPI-PR* on SpiNNaker, PC-PR on a traditional multicore architecture.

4.2. SpinMPI Performance Analysis on PageRank

Let us now test the performance of *MPI-PR* on larger graphs. Figure 7 portrays the PR computation time for a fixed-size graph of $|V| = 768, |E| = 7680$ as a function of the number of workers (i.e., cores) involved in the computation. For certain numbers of workers, several configurations are possible depending on the number of cores and rings involved in the context; for example, a 48-worker configuration can be realized by selecting (7 rings, 1 core per chip), (4 rings, 2 cores per chip), or (3 rings, 3 cores per chip). The black line in the graph shows the average runtime for the various equivalent configurations.

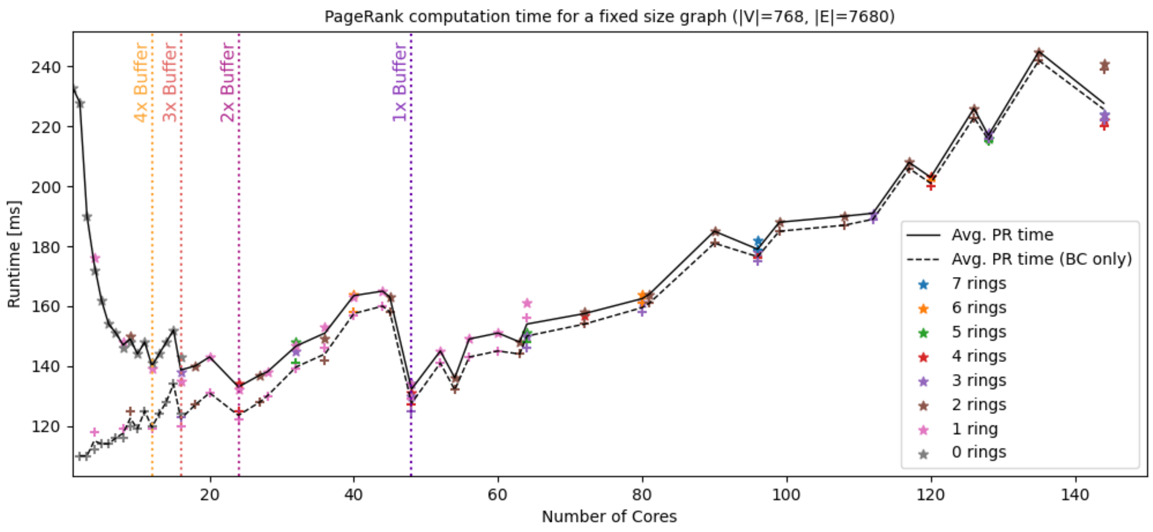

Figure 7. Computation and communication times for *MPI-PR* on SpiNNaker with a mid-sized graph. The figure highlights how the use rate of the communication buffer impacts the broadcast time.

For the PageRank task, the SpinMPI framework does not scale indefinitely; in fact, the trend of the computation time is quite irregular and tends to increase as the number of cores surpasses a certain threshold. There are a few observations to be made: first, we can observe that the trend of the PR execution time is primarily driven by the time spent in the broadcast communication step. It is natural that the communication times increase as the MPI context grows, especially when using broadcast communications which require synchronization of the entire board; as such, the optimal PR execution time is at a relatively low-sized context of 48 cores.

The irregular trend of the broadcast time can depend on several factors. As we saw in Figures 4 and 5, the memory location of the data to be sent and received is one such factor. For this experiment, we modified the program so that the PR array is always saved to DTCM, so this particular factor does not come into play. However, a factor that does come to play is the size of the MPI communication buffer.

In the SpinMPI framework, the data to be sent is copied into a fixed-size communication buffer. The size of this buffer is defined at compile time and its default value is 64 Bytes. The more cores take part in the context, the smaller the subgraphs assigned to each core: at 48 cores, each core handles 16 vertices; as the PR ranks are represented as 4-Byte fixed-point numbers, the size of the send/receive buffer for this configuration is exactly 64 Byte. Therefore, 48 is the minimum number of workers required to fill the communication buffer only once; by reducing the write/read access to this buffer, the broadcast time is reduced as well. The vertical lines in the graph highlight some of the points where the broadcast time drops, corresponding to changes in the number of required buffer accesses.

Finally, in Figure 8 we inspect the behavior of *MPI-PR* when dealing with a very large graph. The 65536-vertex graph is the largest graph the algorithm can evaluate, due to the 16-bit integer representation of the vertex ID. In this figure we show the trend in execution time for different ring configurations as the number of cores per chip increases. Again, we can observe how every ring configuration has an optimal number of cores per chip, after which the communication costs become too high with respect to the time saved in PageRank computation. Most interestingly, we can observe how the execution times are overall higher when the same number of workers is concentrated in a small number of chips (less rings), while they become faster when the workers spread out over several chips (more rings). This happens because for a large graph such as this, none of the problem data can be saved to DTCM; indeed, both the vertex information and PR arrays are stored into RAM. Since RAM is chip-local, all cores on a single chip compete for access to the same RAM bank. We see then how the RAM access time is another important factor for the performance of SpinMPI, as with less workers competing for the same chip's RAM yields better results.

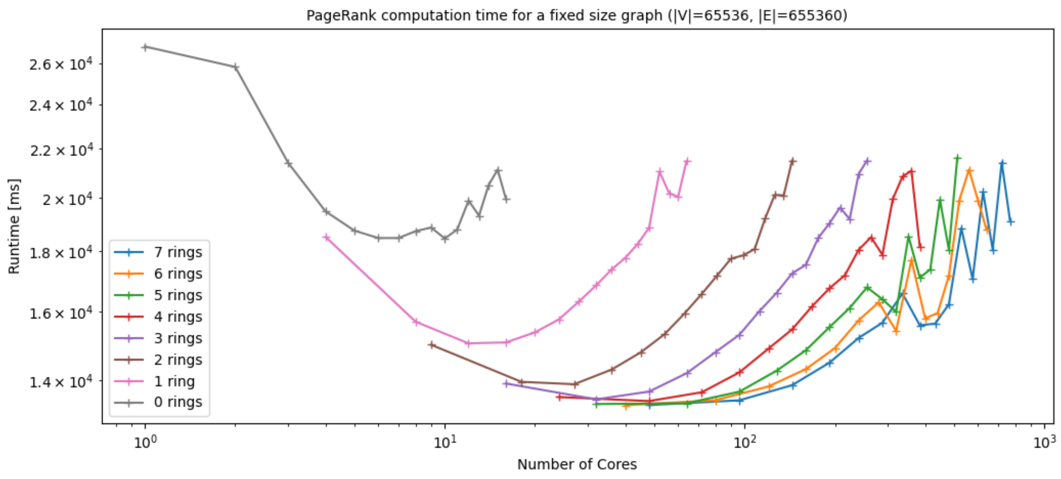

Figure 8. Computation and communication times for *MPI-PR* on SpiNNaker with a large-sized graph. The figure highlights how different placements of the same number of cores impact the execution time.

All in all, the experiments here described demonstrate that for communication-heavy applications such as PageRank, the factors influencing the MPI execution time on SpiNNaker are many and complex, as well as not easy to identify. To summarize our analysis, such factors include (but are likely not limited to):

1. The tradeoff between the computation time saved by parallelization vs. the cost of MPI Broadcast communication, which grows with the number of workers (Figures 4 and 5).
2. The memory location of the data involved in the MPI communication (DTCM vs. SDRAM) (Figures 4 and 5).
3. The mismatch between the size of the data to be sent vs. the size of the MPI communication buffer (Figure 7).
4. The density of workers on each chip, which influences the SDRAM access time (Figure 8).

5. Conclusions

In this work, we presented an MPI-based implementation of PageRank for evaluating the scalability of the SpiNNaker multicore architecture when running a parallel algorithm with huge communications needs and a relatively small computational effort per node. We compared the easy-to-use MPI paradigm with the custom solution (*SNN-PR*) proposed by Blin, which relies on a modified spiking neuron model for performing the rank update, exploiting the standard SNN communication infrastructure. Compared to *SNN-PR*, our approach has the advantage of supporting larger graphs as well as worker synchronization and a lower computational cost per core. As a last point, we confirmed that the SpinMPI library—which provides MPI support for SpiNNaker—allows users to easily port any MPI algorithm implemented for standard computers to the SpiNNaker neuromorphic platform, effectively acting as an interface between any C-language, MPI-compliant program and the native SpiNNaker communication framework, without the need to modify said original program.

The highly efficient interconnection architecture in the SpiNNaker platform, besides being well-suited for SNN applications, shows promise for the low-power parallel execution of tasks in the edge computing domain. On the other hand, being a collection of massively parallel computation elements immersed in a distributed-memory environment with linearly scaling inter-core communication, the SpiNNaker architecture may in fact be the ideal silicon implementation for the MPI paradigm, to the extent that it might be worth consideration even for the realization of systems on a higher scale.

In future works, we plan to continue development on the SpinMPI library (including the implementation of multicast communication, which has the potential to greatly benefit locally connected graph problems such as PageRank), and to employ the library to further explore the potential of MPI for solving general-purpose problems on a neuromorphic platform.

Author Contributions: Conceptualization, G.U. and E.F.; data curation, E.F.; formal analysis, E.F. and A.S.; funding acquisition, G.U. and E.M.; investigation, G.U., E.F. and A.S.; methodology, G.U. and E.F.; project administration, G.U. and E.M.; resources, G.U. and E.M.; software, E.F. and A.S.; supervision, G.U.; validation, E.F. and A.S.; visualization, G.U., E.F. and A.S.; writing–original draft preparation, G.U., E.F. and A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by European Union Horizon 2020 Programme [H2020/2014-20] grant number 785907 [HBP-SGA2].

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: We thank the Advanced Processor Technologies Group led by Steve Furber at University of Manchester for the technical support.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

SpiNNaker	Spiking Neural Network Architecture
Dynap-SEL	Dynamic Asynchronous Processor Scalable and Learning
BrainScaleS	Brain-inspired multiscale computation in neuromorphic hybrid systems
DTCM	Tightly Coupled Data Memory
SNN-PR	SNN-based implementation of PageRank for SpiNNaker
MPI-PR	MPI-based implementation of PageRank for SpiNNaker

References

1. Kasabov, N.K. From von Neumann Machines to Neuromorphic Platforms. In *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 661–677.
2. Christensen, D.V.; Dittmann, R.; Linares-Barranco, B.; Sebastian, A.; Gallo, M.L.; Redaelli, A.; Slesazeck, S.; Mikolajick, T.; Spiga, S.; Menzel, S.; et al. 2021 Roadmap on Neuromorphic Computing and Engineering. *arXiv* **2021**, arXiv:2105.05956.
3. Schuman, C.D.; Potok, T.E.; Patton, R.M.; Birdwell, J.D.; Dean, M.E.; Rose, G.S.; Plank, J.S. A survey of neuromorphic computing and neural networks in hardware. *arXiv* **2017**, arXiv:1705.06963.
4. Young, A.R.; Dean, M.E.; Plank, J.S.; Rose, G.S. A Review of Spiking Neuromorphic Hardware Communication Systems. *IEEE Access* **2019**, *7*, 135606–135620. [CrossRef]
5. Blin, L.; Awan, A.J.; Heinis, T. Using Neuromorphic Hardware for the Scalable Execution of Massively Parallel, Communication-Intensive Algorithms. In Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, Switzerland, 17–20 December 2018; pp. 89–94.
6. Sugiarto, I.; Liu, G.; Davidson, S.; Plana, L.A.; Furber, S.B. High performance computing on spinnaker neuromorphic platform: A case study for energy efficient image processing. In Proceedings of the 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC), Las Vegas, NV, USA, 9–11 December 2016; pp. 1–8.
7. Rhodes, O.; Bogdan, P.A.; Brenninkmeijer, C.; Davidson, S.; Fellows, D.; Gait, A.; Lester, D.R.; Mikaitis, M.; Plana, L.A.; Rowley, A.G.; et al. sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker. *Front. Neurosci.* **2018**, *12*, 816. [CrossRef]
8. Jin, X.; Furber, S.; Woods, J. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In Proceedings of the 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), Hong Kong, China, 1–8 June 2008; pp. 2812–2819.
9. Brown, A.D.; Furber, S.B.; Reeve, J.S.; Garside, J.D.; Dugan, K.J.; Plana, L.A.; Temple, S. SpiNNaker—Programming model. *IEEE Trans. Comput.* **2015**, *64*, 1769–1782. [CrossRef]
10. Gropp, W.D.; Gropp, W.; Lusk, E.; Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*; MIT Press: Cambridge, MA, USA, 1999; Volume 1.
11. Barchi, F.; Urgese, G.; Macii, E.; Acquaviva, A. An Efficient MPI Implementation for Multi-Core Neuromorphic Platforms. In Proceedings of the 2017 New Generation of CAS (NGCAS), Genova, Italy, 6–9 September 2017; pp. 273–276.
12. Page, L.; Brin, S.; Motwani, R.; Winograd, T. *The PageRank Citation Ranking: Bringing Order to the Web*; Technical Report; Stanford InfoLab: Austin, TX, USA, 1999. Available online: http://ilpubs.stanford.edu:8090/422 (accessed on 29 April 2020).
13. Brin, S.; Page, L. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* **1998**, *30*, 107–117. [CrossRef]
14. Schemmel, J.; Grübl, A.; Hartmann, S.; Kononov, A.; Mayr, C.; Meier, K.; Millner, S.; Partzsch, J.; Schiefer, S.; Scholze, S.; et al. Live demonstration: A scaled-down version of the brainscales wafer-scale neuromorphic system. In Proceedings of the 2012 IEEE International Symposium on Circuits and Systems (ISCAS), Seoul, Korea, 20–23 May 2012; p. 702.
15. Moradi, S.; Qiao, N.; Stefanini, F.; Indiveri, G. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE Trans. Biomed. Circuits Syst.* **2017**, *12*, 106–122. [CrossRef] [PubMed]
16. Davies, M.; Srinivasa, N.; Lin, T.H.; Chinya, G.; Cao, Y.; Choday, S.H.; Dimou, G.; Joshi, P.; Imam, N.; Jain, S.; et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* **2018**, *38*, 82–99. [CrossRef]
17. Furber, S.B.; Galluppi, F.; Temple, S.; Plana, L. The spinnaker project. *Proc. IEEE* **2014**, *102*, 652–665. [CrossRef]
18. Furber, S.; Lester, D.; Plana, L.; Garside, J.; Painkras, E.; Temple, S.; Brown, A. Overview of the SpiNNaker System Architecture. *Comput. IEEE Trans.* **2013**, *62*, 2454–2467. [CrossRef]
19. Rowley, A.G.D.; Brenninkmeijer, C.; Davidson, S.; Fellows, D.; Gait, A.; Lester, D.; Plana, L.A.; Rhodes, O.; Stokes, A.; Furber, S.B. SpiNNTools: The execution engine for the SpiNNaker platform. *Front. Neurosci.* **2019**, *13*, 231. [CrossRef] [PubMed]
20. Urgese, G.; Barchi, F.; Macii, E. Top-down profiling of application specific many-core neuromorphic platforms. In Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Turin, Italy, 23–25 September 2015; pp. 127–134.
21. Urgese, G.; Barchi, F.; Macii, E.; Acquaviva, A. Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms. *IEEE Trans. Emerg. Top. Comput.* **2018**, *6*, 317–329. [CrossRef]

22. Barchi, F.; Urgese, G.; Siino, A.; Di Cataldo, S.; Macii, E.; Acquaviva, A. Flexible on-line reconfiguration of multi-core neuromorphic platforms. *IEEE Trans. Emerg. Top. Comput.* **2019**. [CrossRef]
23. Malewicz, G.; Austern, M.H.; Bik, A.J.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*; Association for Computing Machinery: New York, NY, USA, 2010; pp. 135–146.
24. Urgese, G.; Barchi, F.; Parisi, E.; Forno, E.; Acquaviva, A.; Macii, E. Benchmarking a Many-Core Neuromorphic Platform with an MPI-Based DNA Sequence Matching Algorithm. *Electronics* **2019**, *8*, 1342. [CrossRef]