POLITECNICO DI TORINO
Repository ISTITUZIONALE

Benchmarking a many-core neuromorphic platform with an MPI-based DNA sequence matching algorithm

Original
Benchmarking a many-core neuromorphic platform with an MPI-based DNA sequence matching algorithm / Urgese, G., Barchi, F., Parisi, E., Forno, E., Acquaviva, A., Macii, E.. - In: ELECTRONICS. - ISSN 2079-9292. - 8:11(2019), p. 1342. [10.3390/electronics8111342]

Availability:
This version is available at: 11583/2771341 since: 2019-12-04T14:45:56Z

Publisher:
MDPI AG

Published
DOI:10.3390/electronics8111342

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

05 September 2026

electronics

MDPI

Article

Benchmarking a Many-Core Neuromorphic Platform With an MPI-Based DNA Sequence Matching Algorithm

Gianvito Urgese [1,*], Francesco Barchi [2], Emanuele Parisi [2], Evelina Forno [2], Andrea Acquaviva [3] and Enrico Macii [1]

1 Interuniversity Department of Regional and Urban Studies and Planning, Politecnico di Torino, 10129 Torino, Italy; enrico.macii@polito.it
2 Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy; francesco.barchi@polito.it (F.B.); emanuele.parisi@polito.it (E.P.); evelina.forno@polito.it (E.F.)
3 Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi", University of Bologna, 40126 Bologna, Italy; andrea.acquaviva@unibo.it
* Correspondence: gianvito.urgese@polito.it

Received: 29 September 2019; Accepted: 12 November 2019; Published: 14 November 2019

Abstract: SpiNNaker is a neuromorphic globally asynchronous locally synchronous (GALS) multi-core architecture designed for simulating a spiking neural network (SNN) in real-time. Several studies have shown that neuromorphic platforms allow flexible and efficient simulations of SNN by exploiting the efficient communication infrastructure optimised for transmitting small packets across the many cores of the platform. However, the effectiveness of neuromorphic platforms in executing massively parallel general-purpose algorithms, while promising, is still to be explored. In this paper, we present an implementation of a parallel DNA sequence matching algorithm implemented by using the MPI programming paradigm ported to the SpiNNaker platform. In our implementation, all cores available in the board are configured for executing in parallel an optimised version of the *Boyer-Moore* (BM) algorithm. Exploiting this application, we benchmarked the SpiNNaker platform in terms of scalability and synchronisation latency. Experimental results indicate that the SpiNNaker parallel architecture allows a linear performance increase with the number of used cores and shows better scalability compared to a general-purpose multi-core computing platform.

Keywords: benchmarking neuromorphic HW; neuromorphic platform; spiNNaker; spinMPI; MPI for neuromorphic HW; Boyer-Moore; DNA matching algorithm

1. Introduction

A neuromorphic system is a massively multi-core system composed of simple processing units and memory elements communicating by message exchanging [1]. This type of approach strives to simulate the behaviour of the brain using design principles based on biological nervous systems. Neuromorphic systems differ from traditional multi-core systems in the way in which memory and processing are organised. Indeed, in this case, memory is distributed with processing units rather than centralised and physically separated from the cores. Using this strategy, it is possible to avoid the traditional bottleneck of memory access time, present in the classical Von-Neumann architectures. The main idea behind this kind of system is to process information using an event-driven protocol that lets the cores work in an asynchronous way [2]. The processing units remain in an idle state until an event is presented, triggering a reaction; after that, the units return to the idle state. Using this feature, neuromorphic systems are much more energy-efficient than traditional multi-core systems. This idea is inspired by biology; indeed, the human brain is composed of billions of neurons connected

by synapses, working asynchronously, with a power consumption lower than that of a light-bulb [3]. Another peculiarity of neuromorphic systems is the high number of interconnections between the processing units, which speeds up and simplifies communication between the cores.

Neuromorphic HW platforms are attracting the interest of many research groups, mainly for the simulation of neural network structures observed in the brain and modelled through the simulation of Spiking Neural Networks (SNN). Although initially intended for brain simulations, the adoption of emerging neuromorphic HW architectures is also appealing in fields such as high-performance computing and robotics [4]. It has been proved that neuromorphic platforms provide better scalability than traditional multi-core architectures and are well suitable for classes of problems which require massive parallelism as well as the exchange of small messages, for which neuromorphic HW has a native optimised support [5]. However, the tools currently available in this field are still weak and miss many useful features required to support the spreading of a new neuromorphic-based computational paradigm.

In this paper, we analyse and benchmark the scaling capability of the SpiNNaker neuromorphic architecture. The SpiNNaker Machine is a multi-chip, globally asynchronous locally synchronous (GALS) neuromorphic architecture that connects general purpose ARM cores in a toroidal-shaped triangular mesh. It is efficient when used to solve problems modelled as a directed graph with an important communication component.

Other works have used this platform to execute parallel general purpose computation, with positive outcomes both for scaling performances and energy efficiency. In Blin et al. [5], authors have customised the neural model of an SNN configured for reproducing the connection graph of a page rank problem, showing that the scalability rate of the neuromorphic platform outperforms the general purpose architectures; whereas Sugiarto et al. [6] have implemented on SpiNNaker an energy efficient image processing algorithm, using a task graph representation to describe the mechanism and behavior of the method. However, none of these two approaches has tested synchronous applications, since both of them used an adapted SNN simulated with the standard asynchronous framework.

In previous work [7], authors have used a minimal Message Passing Interface (MPI) framework to implement a synchronization strategy that allows configuration of the cores of the board with a distributed application implementing the N-Body problem. The authors benchmarked the performance of the board in the execution of an MPI parallel application that simulates 2 k particles on 240 processors with a speed-up of 194× and an efficiency of 80% when compared to the serial version running on a single CPU.

In this paper, we compared the scaling performance of the SpiNNaker system with that offered by a many-core general purpose architecture. We implemented a parallel processing approach for a pattern matching algorithm able to identify the similarity of DNA sequences. In our implementation, we used the Message Passing Interface (MPI), a distributed parallel programming paradigm, to synchronise the communication of the computing cores on the two architectures. By using the MPI framework, we can port on the SpiNNaker platform an algorithm normally executed on a standard architecture without any need to re-shape the algorithm in the form of a Spiking Neural Network. The focus of the research presented in this paper is threefold.

- To benchmark the performances of the SpiNNaker board in computing pattern matching tasks by running synchronous data-stream algorithms.
- To explore the potential of the custom shape mesh, implemented on the SpiNNaker board, in a supporting parallel application that adopts a one-to-many communication system.
- To demonstrate how it is easy to port synchronous applications, implemented for the general-purpose computer, on the SpiNNaker board by using our software component that supports MPI for SpiNNaker.

The rest of the manuscript is organized as follows: Section 2 provides background information on existing neuromorphic architectures, with a detailed focus on the SpiNNaker board and on the DNA

search algorithm. Section 3 describes the materials and methods used to carry out the study, whereas Section 4 examines experimental results. Finally, Section 5 closes with the conclusions.

2. Background

In the following, we provide a background on neuromorphic hardware in general and SpiNNaker in particular. Then we discuss the variant of the Boyer-Moore algorithm that we implemented with the MPI framework in order to benchmark the scaling capability of the SpiNNaker platform.

There are two main approaches to neuromorphic computing—VLSI architectures where neurons are modelled at transistor-level and communications are handled with connection crossbar array and custom architectures where general-purpose cores are connected to form a mesh of processors optimised for the transmission of small packets [8–10]. In the following, we report four representative architectures.

BrainScaleS is a VLSI platform developed at the University of Heidelberg [11]. The main idea behind this project is to use above-threshold analogue circuits to physically model neuronal processes, exploiting analogy between electronic circuits and the ionic circuits in biological neurons. Analogue neurons are delivered using wafer-scale integration.

Dynap-SEL is a VLSI chip called Dynamic Asynchronous Processor Scalable and Learning that is produced with four neural processing cores which implement 256 analog Adaptive Exponential Integrate and Fire neurons placed in a 16 $\times$ 16 grid with 64 programmable synapses for each neuron. In the Dynap-SEL architecture, it is available also a supplementary core 64 analog neurons and 8192 plastic synapses with on-chip learning and 4096 programmable synapses [12].

Loihi is a neuromorphic processor produced by Intel [13]. It features a many-core mesh comprising 128 neuromorphic cores, three embedded x 86 processor cores and off-chip communication interfaces that extend the mesh in 4-planar directions to other chips. All logic in the chip is digital and implemented as an asynchronous bundled-data design.

The Spiking Neural Network Architecture (*SpiNNaker*) [14] is a real-time neural network simulator following an event-driven computational approach [15]. This architecture is able to emulate neural populations and to simulate an entire Spiking Neural Network (SNN) in real-time. What sets SpiNNaker apart from all the above platforms is the fact that its architecture does not implement neurons via custom VLSI designed circuits, but it consists of a mesh of general-purpose ARM cores with a neuromorphic connectivity scheme. While the platform is designed to run SNN simulations and a software stack is provided to facilitate this purpose, in principle, the general-purpose cores can run any sort of C program compiled for ARM.

2.1. SpiNNaker Architecture

The base element of the SpiNNaker architecture is the SpiNNaker chip Figure 1, an SoC composed by 18 ARM-968 cores running at 200 MHz without a floating point unit but equipped with a custom router. Each processor has 32 kB of ITCM, 64 kB of DTCM, and shares through a system NoC 128 MB of SDRAM with the other processors in the chip. All the cores in the SoC (Application Processors) can run user applications, except one core for each chip, which is designated to be the *Monitor Processor*. This particular processor always executes the *SC&MP* program, which is a sort of operating system performing operations of memory management and acting as a packet manager, able to receive and transmit packet traffic from/to the cores. SpiNNaker chips (nodes) are connected to six neighbours and assembled on a PCB board made of 48 SpiNNaker chips (Spin5). The host computer can communicate with and configure a Spin5 via the Monitor Processor of the chip (0,0), the only one that is physically connected to an 100 Mbit Ethernet interface.

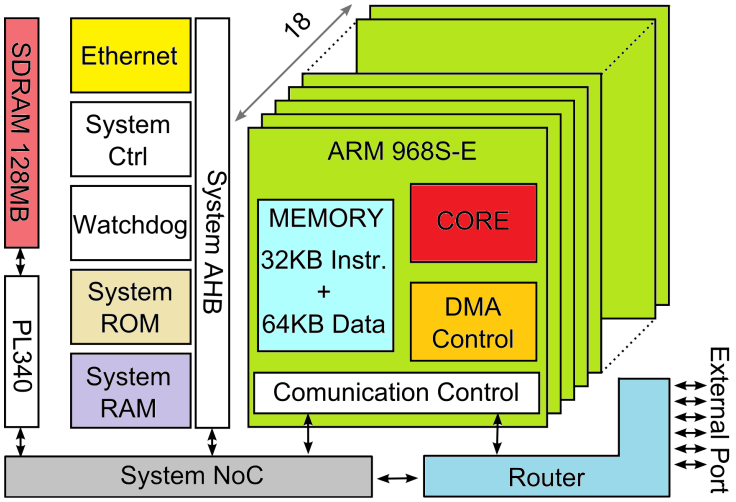

Figure 1. The SpiNNaker chip architecture.

2.2. SpiNNaker Network

The kernel of the interconnection among all cores of all chips of the simulator is the router, specifically designed to deliver packets as fast as possible (0.1 µs per hop) [16]. The particular design of the router, despite limitations on the synchronous transmission of packets [17,18], allows transmission of two operative packet types—Multicast (MC) and Point to Point (P2P). The length of these packets can be up to 72 bits and can carry a 32 bits long payload.

Multicast (MC) packets can reach many cores across the board. In neural simulations, they are widely used in order to spread neural potentials to multiple destinations. Point-to-Point (P2P) packets can be used for chip-to-chip transmissions. Each chip is uniquely identified by its coordinates (x, y), which define the chip's position in the chip mesh. P2P packets are always delivered to a chip's monitor processors.

The APIs of the SpiNNaker system provide a higher level of abstraction that simplifies the usage of chip interconnection. The SpiNNaker Datagram Protocol (SDP) can be used to manage communication between processors up to 256 Bytes [14]. The Monitor Processors act as a middleware between the SDP protocol and the on-board network. A Monitor Processor that receives an SDP packet splits the whole frame into 32-bit fragments to be delivered in the internal network through the P2P packets.

2.3. SpiNNaker Software

The software used to run a simulation managing the boards involves board-side code developed in C and Assembly [19] and host-side code mostly written in Python [20].

In this work we used the software stack provided by the SpinMPI library—a partial implementation of MPI on SpiNNaker [7] able to fully exploit the communication potential provided by the architecture, using the Application Command Framework (ACF) and the Multicast Communication Middleware (MCM) to manage communications.

The ACF uses the Application Command Protocol (ACP) to implement a Remote Procedure Call (RPC) capability in SpiNNaker at the application level [21]. Moreover, this library implements the *memory entity* concept. A memory entity is a managed memory space (DTCM, SysRAM, SDRAM), identified by an integer number, on which it is possible to perform CRUD operations (Create, Read, Update, Delete) locally or remotely. A *memory entity* can be created with a size limit of 256 Byte, that is, the ACP payload limit. The MCM instead implements unicast and broadcast communications, exploiting the multicast network capabilities of SpiNNaker.

2.4. The DNA Pattern Matching Algorithm

One of the most recurrent and widely studied problems in computer science is pattern matching—this problem has several real-world applications such as fast sub-string searching for network intrusion detection, mail spam filters, protein motif search and DNA/RNA sequence alignments [22]. Given a text string T of length n and a pattern string P of length $m \leq n$, the pattern matching problem can be stated as retrieving all positions i where pattern P occurs in text T, such that $0 \leq i \leq n - m$.

A straightforward solution for the pattern matching problem consists of looking for the pattern sequence in the text position by position until every occurrence is found. Unfortunately, such an approach leads to a $O(m \cdot n)$ asymptotic complexity, which is not acceptable for large sets of data.

Given the practical relevance of this problem, many approaches were proposed in the literature for improving the naïve way. One of these is the *Boyer-Moore* algorithm [23,24], which trades space usage for time efficiency, defining rules for pruning the search space avoiding the exploration of all text positions. A C++ implementation is available in Reference [25].

Figure 2 provides an intuition for this approach; given the text in the picture, the first attempt looks for pattern "GTA" in position ⓪, which is not correct.

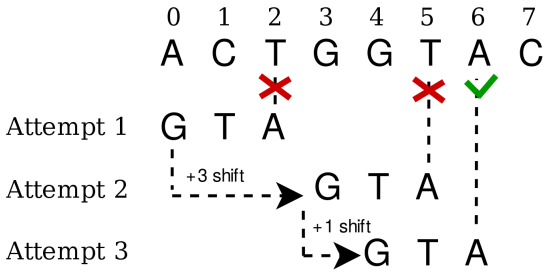

Figure 2. Intuition of the Boyer-Moore search procedure.

The naïve approach would perform the next search from position ①, but this is not ideal since the first instance of the letter "G" in the pattern occurs at position ③ in the text, meaning that searching any position in the middle is useless. Implementing this optimization requires pre-processing of the pattern to be matched; a *shift table* is computed, storing the number of text positions that can be safely skipped for each symbol in the target alphabet. Whenever a mismatch is found, given the next symbol to be searched, the *shift table* is accessed and the next position to be considered is computed.

We used a refined version of the *Boyer-Moore* algorithm, also known as *Fast string matching method for Encoded DNA sequences (FED)* [26], which takes advantage of the low-cardinality of the DNA alphabet. In the *FED* version, each of the four symbols composing the DNA alphabet is assigned a unique 2-bit code, packing four elements into a single byte, padding last bits with zeros in the case of sequences where the length is not a multiple of 4. Additionally, a bit-mask is used to distinguish valid bits from padding in the last encoded byte.

The procedure consists of two successive steps:

- *Pre-processing*, where texts and patterns are encoded and a *shift table* is computed for every pattern to be matched.
- *Matching*, where the actual search procedure is performed, is implemented as a byte-by-byte comparison between the text and pattern encoded sequences. If every byte of the pattern is sequentially found in the text, then the current position is registered as a match. Otherwise, the *shift table* is accessed to compute how many positions the pattern is allowed to skip before performing the next check.

Figure 3 summarizes the string matching procedure flow. As long as the customised *Boyer-Moore* procedure can perform a matching operation on encoded sequences, the encoding step can be

considered not part of the algorithm as it can be done offline by storing the encoded sequences in custom binary files which constitute the actual source of data for the pattern matching engine.

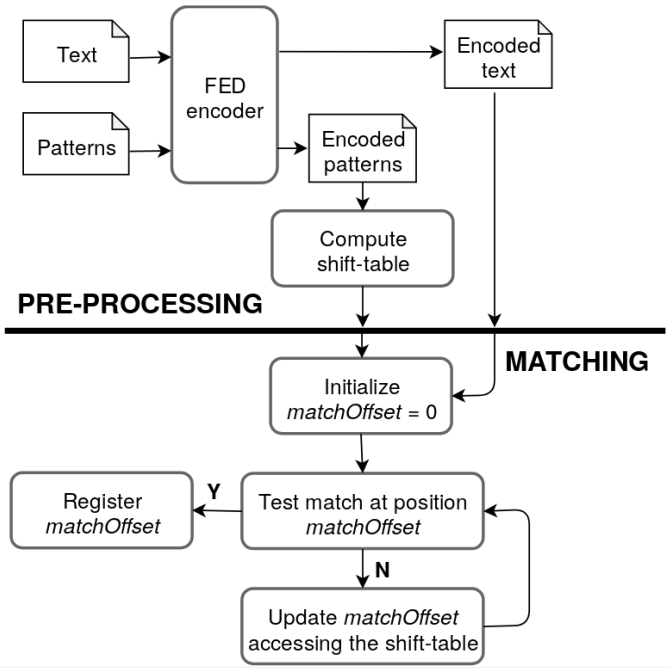

Figure 3. Flowchart of the string matching algorithm.

3. Materials and Methods

3.1. Implementation of DNA Sequence Matching with MPI

Pattern matching over DNA sequences can be considered an embarrassingly parallel application, because the average use case consists in matching millions of patterns against multiple text sequences, independently [27].

The inputs for the benchmark application are two binary files, storing the encoded texts and patterns to be analyzed. From an algorithmic point of view, running *FED* on already encoded sequences is equivalent to loading plain sequences and encoding them online. For the sake of benchmarking the communication effort in the target platforms, we decided to encode sequences off-line. Moreover, we split text sequences into a set of chunks with a given fixed size. This step is required because in bioinformatics applications, generally, the text represents one or more genomes and its size is not suitable to be sent in a single shot as it is.

Our parallel implementation of the search algorithm identifies two main roles among the MPI processes—the *MPI control process*, which is the role adopted by the MPI process with rank 0, and the *MPI worker*, which is the role adopted by all remaining MPI processes.

The algorithm works in two distinct steps, outlined in Figure 4: configuration (A) and match (B). During configuration step (A), the *MPI control process* accesses the file system, loads the *FED* encoded patterns and distributes them among the *MPI workers* so that each working process handles approximately the same workload. Pattern distribution is implemented as a set of point-to-point communications, using MPI_Send/MPI_Recv primitives. Once an *MPI worker* receives its patterns it computes the *shift table* for them, completing the pre-processing phase shown in Figure 4. This strategy allows both to reduce the amount of data sent over the communication network, as the patterns are already encoded and to distribute the pre-processing efforts equally among all available working nodes, as long as any *MPI worker* finalizes the pre-processing step on its patterns only.

During the matching step (B), the *MPI control process* loads the encoded chunks of text and broadcasts them one at a time to all the *MPI workers*, which are in charge of performing the actual pattern matching procedure by calling the search primitives. As shown in Figure 4, once a match is found, it is saved into a buffer local to the MPI instance that discovered it. Once every chunk has been analysed, all the MPI instances synchronize to produce two report files containing information about the matches found and the run-time needed for accomplishing their tasks.

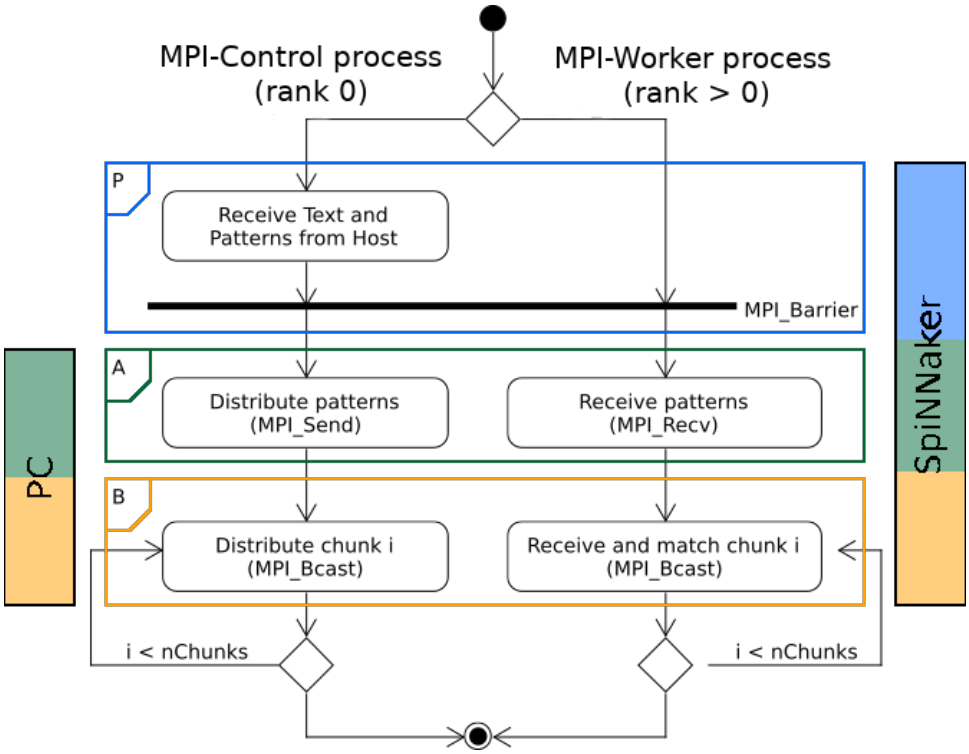

Figure 4. Flowchart of the implementation of MPI-FED on a general purpose architecture and on SpiNNaker. The step A performs the configuration, the step B execute the matching, whereas during the step P our implementation implement a preliminary phase for transferring the data to the SpiNNaker board.

3.2. Adaptation of FED with MPI for SpiNNaker

The implementation of *FED* with MPI for SpiNNaker retains the configuration (A) and match (B) phases from the previous section, as depicted in Figure 4. However, an additional preliminary phase (P) is required in order to transfer the problem data to the board. The configuration step (A) will then be performed by one of the SpiNNaker cores, taking up the role of *MPI control process*.

Using the SpinMPI Python library, the host launches the *MPI Runtime* and creates an *MPI Context* declaring the number of chips and cores that will be used by the application on the Spin5 board. The *MPI Runtime* is also in charge of loading and starting the application binary on the board.

In the preliminary phase (P), the communication between the computer host and the on-board application is performed through the use of ACP memory entities (MEs). First, the binary files containing the genome and the search patterns are read by the *MPI Runtime*. In this phase, the host will write into a ME belonging to processor (0, 0, 1) (the *MPI control process*) two integers indicating the number of chunks (`nchunks`) and patterns (`npatterns`) which will be loaded into SpiNNaker. The *MPI control process* allocates in SDRAM the memory necessary to contain all chunks and patterns. After allocation is performed, the addresses of these memory blocks are read by the *MPI Runtime*, again using ACP. The *MPI Runtime* can proceed to fill the *MPI control process* memory with the genome and the search patterns previously read.

An MPI Barrier forces all *MPI workers* to wait until the *MPI control process* has received all data from the *MPI Runtime*. Once the problem data has been transferred (phase Ⓟ), phase Ⓐ can begin. The *MPI control process* distributes the patterns among all worker cores through MPI_Send/MPI_Recv primitives and the *MPI workers* store the pattern data in their DTCM and compute the *shift tables*.

The phase Ⓑ begins after all patterns have been distributed. The *MPI control process* sends a text chunk to all *MPI workers* executing a broadcast communication. The SpiNNaker implementation of the *MPI_Bcast* function is a blocking call, as the memory limitations of the platform do not allow for large communication buffers; hence, the *MPI control process* will proceed to send the next chunk only after all workers have processed the current chunk. On the worker side, only one text buffer is allocated into DTCM, since the text chunks will be processed sequentially and a chunk can be replaced whenever a new one is obtained. When a *MPI worker* executing the *FED* algorithm finds a match position, it is stored into a linked list together with the chunk and matching pattern identifiers. Thus the position in the reference sequence can be retrieved.

After all the text chunks have been processed, the application is finalised, and the *MPI Runtime* can download the results directly from the memory of SpiNNaker cores.

4. Results and Discussion

In this section we report the results of tests designed to characterise the performance of the SpiNNaker system running a pattern matching algorithm implemented with the MPI programming paradigm.

As a preliminary evaluation, we measured the execution time and memory usage of the MPI primitives implemented on SpiNNaker that we will use for implementing the parallel FED algorithm. MPI_BARRIER, MPI_SEND/MPI_RECV, and MPI_BROADCAST primitives will be tested and the results are reported in Section 4.1. Next, we performed an evaluation of the performance of FED algorithm implemented with MPI and executed on the SpiNNaker and on the CPU-based architectures. This last analysis aims to evaluate the scalability and power efficiency of the SpiNNaker platform when compared with a standard architecture. The results are detailed in Section 4.2.

4.1. Performance of MPI on SpiNNaker

In Tables 1 and 2 and Figure 5 we report the performance of the MPI primitives on SpiNNaker. Table 1 shows the average execution time for 2000 iterations of the MPI_BARRIER synchronization primitive; the growth of the execution time is bounded with respect to the size of the context (i.e., the number of cores being used). The amount of memory necessary to store information about the context also grows slowly with respect to the number of cores.

Table 1. Table profiling the performance of 2000 iterations of MPI_BARRIER on SpiNNaker.

Cores	Time (ms)	Memory Usage
2	4.0	0.128
192	18.0	0.165
384	20.0	0.204
576	21.0	0.242
768	22.79	0.280

Table 2 shows the average execution time for the MPI_SEND/MPI_RECV unicast primitive. The average execution time grows linearly with the amount of data sent.

Finally, in Figure 5 we describe the average execution time for 2000 iterations of the MPI_BROADCAST primitive with respect to the amount of data sent and the context size. Once again the execution time grows linearly with the data sent, with overhead corresponding to the context-wide synchronization. The execution time also has a bounded growth in relation to the number of cores.

Table 2. Table profiling the performance of the MPI_SEND/MPI_RECV unicast primitive for different amounts of data sent on SpiNNaker.

Data Size	Time (ms)
1 kB	2.07
2 kB	4.12
4 kB	8.24

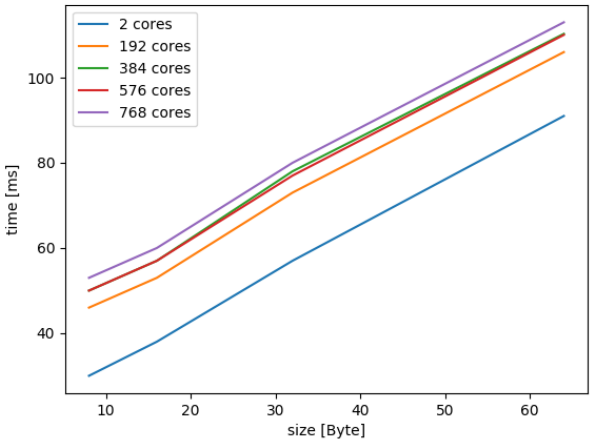

Figure 5. Graph profiling the performance of the MPI_BROADCAST primitive on SpiNNaker.

4.2. Evaluation of Boyer-Moore MPI Implementation Running on SpiNNaker

In the following, we analyse the efficiency and scalability of our optimised *Boyer-Moore* (*FED*) implementation on SpiNNaker. We compare it with the scalability on a traditional multi-core CPU using a server configuration with two Intel Silver Xeon 4114 processors, each with 10 cores and 20 threads. The *FED* algorithm is implemented in C and used to benchmark both Server and SpiNNaker architectures. The benchmark running on the general purpose Server architecture is written in C++ and compiled with g++ 7.4.0 and MPICH 3.3 parallel environment. The benchmark running on SpiNNaker architecture is written in C and compiled with gcc-arm-none-eabi 5.4.1 and SpinMPI 19w19. At this point, it is important to note that, by using the SpinMPI library, we ported the *FED* code written for a standard PC to the SpiNNaker hardware without applying any adaptation or transformation of the code.

The text used for the sake of testing is the *Escherichia coli* genome, which is about 4 million symbols long, leading to an encoded text of about 1 MB size, which is then split into a set of about 4000 chunks, each 256 Bytes long.

There exist two types of strategies to evaluate the scalability of a problem in a parallel environment:

- *Strong-scaling* [28] keeps the size of the problem fixed and evaluates the application runtime when multiple processes are used. This strategy is suitable for CPU-bounded problems.
- *Weak-scaling* [29] is used to test the scalability of memory-bounded problems, as it keeps constant the ratio between the problem size and the number of working processes used.

The SpiNNaker platform provides a fast, core-local data memory (DTCM) of 64 kB. This memory constraint allows to store at most 100 *FED* patterns per node, totalling 40 kB in size. Given this memory constraint, we decided to use a *weak-scaling* benchmarking strategy to scale our benchmark up to the 768 nodes available on SpiNNaker. The problem size must be calibrated in order to claim a condition of equivalence and perform a fair comparison between different architectures; in our case, a condition of equivalence is met whenever the same *FED* execution time t_{FED} is observed using a single *FED* worker. When SpinMPI is requested to match 1000 *FED* chunks against 100 *FED* patterns on a single node, a run-time of 26,970 ms is measured; the same run-time, for the MPICH implementation

with 1000 *FED* chunks, is obtained when the single *FED* worker used is in charge of 12,500 *FED* patterns. This preliminary assessment is needed to evaluate only the scalability features of the two architectures, without considering the difference in computing power of the single working node for the two architectures. The reason for this comparison is to put the performance of MPI on SpiNNaker in a familiar perspective, as the CPU-DualSocket server is a widespread general purpose machine that allows to use MPI; however, the communication on the Xeon is networkless message passing happening entirely in RAM, while the message passing on SpiNNaker makes efficient use of the board's interconnection scheme.

A general strategy for evaluating the parallel scaling of an MPI application is computing the scaling efficiency, which measures how good the application is at using every node the parallel environment has. Given an environment with N workers and a problem that requires $t_{\mathrm{FED},i}$ units of time to be solved with i workers, the *weak-scaling* efficiency E_N can be measured as in Equation (1). The speed-up S_N can be easily inferred from the efficiency and computed with Equation (2).

$$E_N = \frac{t_{\mathrm{FED},1}}{t_{\mathrm{FED},N}} \tag{1}$$

$$S_N = E_N \cdot N \tag{2}$$

Figures 6 and 7 report the speed-up and efficiency of the *FED* with MPI algorithm on the Server and SpiNNaker architectures. The horizontal axis represents the number of MPI workers used; both systems were tested until saturation, with the Server reaching 40 parallel workers through Intel hyper-threading and the Spin5 board utilizing all 768 available physical cores. Tests were performed for genomes of 500, 1000 and 2000 chunks.

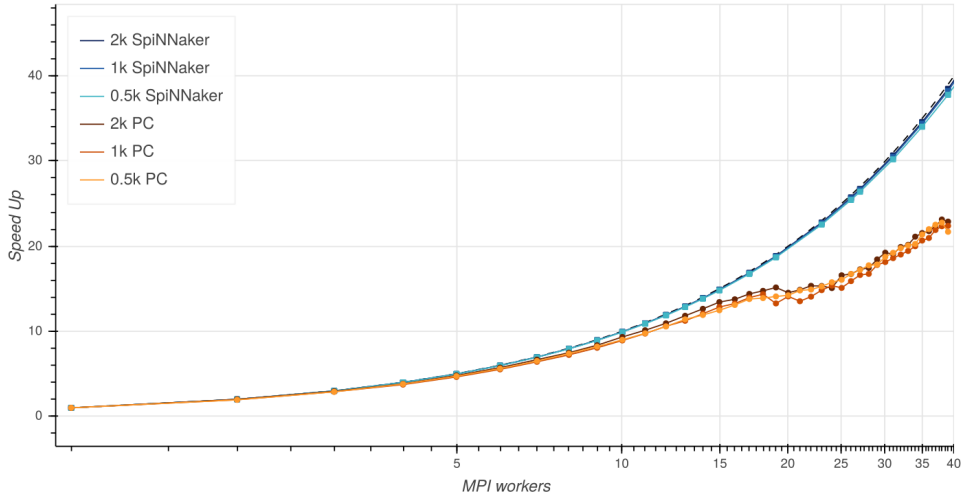

Figure 6. Comparison of Weak-scaling speed-up for MPI-FED on a general purpose architecture and on SpiNNaker.

In Figure 6 we can see how the massively parallel architecture of SpiNNaker influences the speed-up. The high number of physical cores on the machine lets the speed increase linearly, avoiding the discontinuities that a general-purpose processor has at critical points when hyperthreading is activated to provide the required number of workers (note, in the graph, the inflection point at 20 MPI workers for the PC version, i.e., the point at which the maximum number of physical threads on the Xeon is reached).

In Figure 7 SpiNNaker demonstrates excellent scalability, with efficiency values close to 95% for up to 200 workers. Additionally, we can see that the performance markedly improves for longer text sequences; the efficiency for 768 workers processing 2000 chunks is 87.83%. The reason for this happening is that as the size of the data to be processed increases, the ratio of processing time

to communication time in the overall algorithm increases, since the data are only sent once at the beginning of processing and then gathered at the end. The bottleneck due to the communication overhead thus becomes less prevalent, and the efficiency improvement due to massive parallelism is more evident.

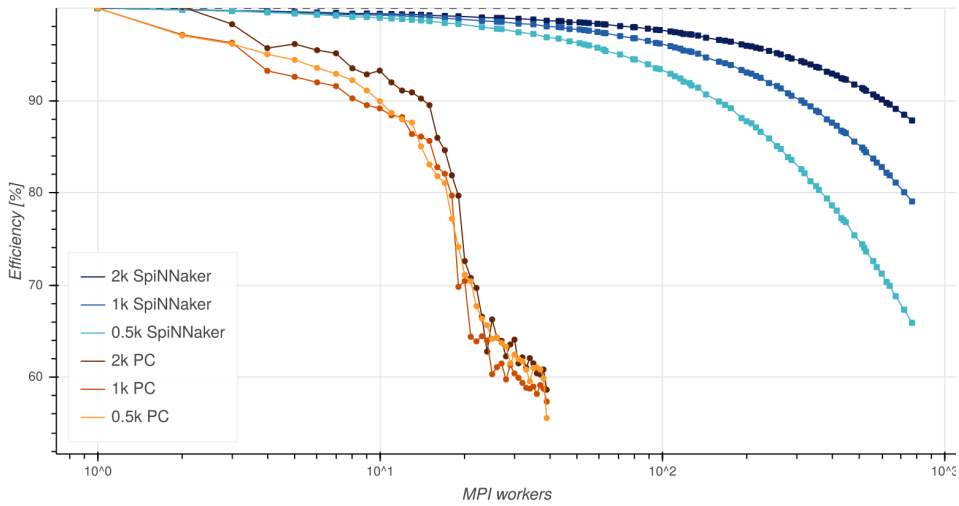

Figure 7. Comparison of Weak-scaling Efficiency for MPI-FED on a general purpose architecture and on SpiNNaker.

By contrast, the efficiency of the Server dips much faster, dropping below 90% as soon as the requested MPI workers outnumber the physical cores. It also remains fairly constant when changing the number of chunks. This appears reasonable as, for the high-speed CPU used in the test, the computation time is very small, but it suggests that other phases of the computation such as inter-process communication and thread management have a significant impact on the efficiency of the algorithm.

As a side-experiment, we evaluated the impact of the size of the *FED* buffer distributing data among the *MPI workers* on the measured scaling efficiency. Figure 8 shows the scaling efficiency of two experiments—the former distributes the *FED* chunks to be analyzed as 1000 256-Byte packets. The latter broadcasts the same amount of data, formatted as 125 2-kB packets. Figure 8 highlights that the two scaling efficiency tracks are comparable, meaning that the size of packets used to distribute *FED* chunks among the *MPI workers* does not impact the benchmark results for the general purpose architecture.

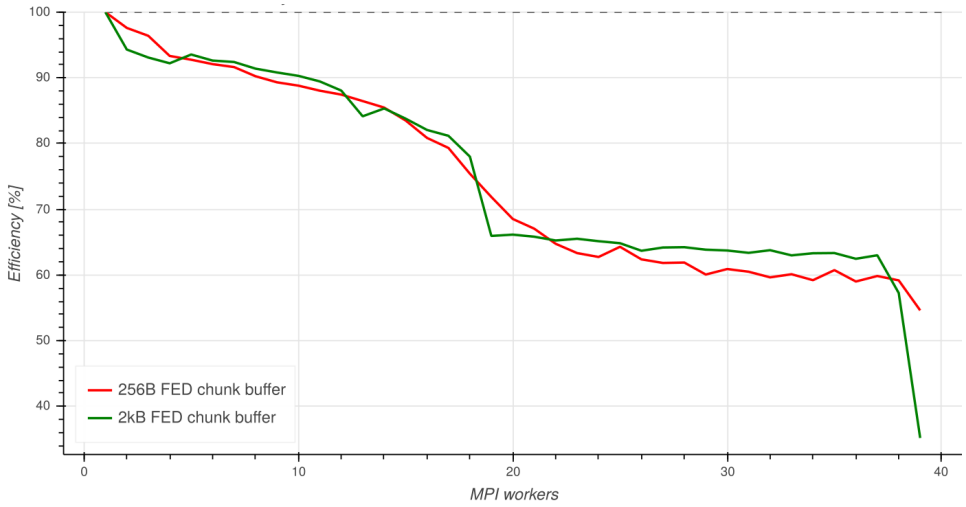

Figure 8. Efficiency of the general purpose architecture for different *FED* buffer sizes.

Finally, we can make a comparison of the power efficiency on the two architectures by using estimated consumption based on the nominal values from the CPU [30] and SpiNNaker [31] data-sheets. For the Intel Xeon, we consider the peak and idle powers at the values of $P_{peak} = 11,030$ mW and $P_{idle} = 6320$ mW, and we hypothesize that the number of active physical cores (out of the available 20), $f(x)$, can be expressed as a function of the active MPI workers x as $f(x) = ceil(\frac{x+1}{2})$. The appearance of the term $x + 1$ rather than x is because there is one Controller process that has the task of distributing the data and patterns to the MPI workers. Based on this assumption, we assign a power consumption of P_{peak} to the active cores and of P_{idle} to every other core; thus the estimated power consumption with respect to the number of MPI workers x is $P(x) = P_{peak} \cdot f(x) + P_{idle} \cdot (20 - f(x))$.

On the other hand, for SpiNNaker we consider the values of Idle Power per Chip $C_{idle} = 360$ mW, Idle Power per Core $P_{idle} = 20$ mW, Peak Power per Core $P_{peak} = 55.56$ mW, and the Off-Chip-Link power, $P_{link} = 6.3$ mW. The power estimation for SpiNNaker depends on the MPI execution context, which can be described by a pair of values (p, k) where $p \in [1, 16]$ is the number of active processors per chip and $k \in [1, 48]$ is the number of active chips. The power estimation formula can be expressed as a function of the number of active processors and chips as $P(p, k) = k \cdot (C_{idle} + (P_{peak} - P_{idle}) \cdot (p + 1) + P_{link}) + (48 - k) \cdot C_{idle}$. Counting $p + 1$ processors to include the Monitor Processor on each core. Then, the estimated power given the number of MPI workers x is $P(x) = P(p, k)|min_k[p \cdot k = x + 1]$ As in the CPU case, we count $x + 1$ processes to include the Controller process.

Given the architectural difference between the SpiNNaker and CPU machines, it is necessary to outline a fair method to evaluate the efficiency of the algorithm's implementation. We define power efficiency as the energy consumed to align a single pattern to the reference, measured in units of mJ/pattern, as a function of the parallelisation effort of the given system, expressed as a percentage of the total resources. The maximum energy efficiency is obtained when all resources are in use, corresponding to a parallelisation effort of 100%. For SpiNNaker it is easy to assume that 100% utilisation occurs when all 768 cores are busy (i.e., at 767 MPI workers), corresponding to an average energy consumption of 37.3 mJ/pattern. For the CPU utilisation, we can either consider 100% utilisation to be the situation where all physical cores are active, or the one where all the virtual cores are active (20 physical + 20 virtual, providing 39 MPI workers). In the first case, the estimated average energy consumption is of 51 mJ/pattern, with an estimated power saving of 27% in favour of SpiNNaker. In the second case, the energy is 43 mJ/pattern, with SpiNNaker consuming 13% less.

5. Conclusions

In this work, we presented an implementation of an MPI-based DNA sequence matching algorithm for evaluating two critical aspects of using one of the more promising neuromorphic emerging technology. As the first point, we benchmarking the SpiNNaker many-core neuromorphic platform and its MPI support, showing that the scaling performances are kept linear when an increasing number of cores is used during the computation. As the second point, we demonstrated that by using the spinMPI library, which provides MPI support for SpiNNaker, we could easily port algorithm implemented for standard computers on the many-core neuromorphic platform.

The MPI standard exposes a programming model for the development of parallel applications in a distributed memory environment without knowledge of the interconnections between the computing units of the underlying architecture. The implementation of MPI for a specific architecture is therefore expected to implement the most suitable features in order to exploit the available resources and to synchronise the computing flow.

In the case of SpiNNaker, the implementation of MPI must deal with a resource limit both in terms of memory and computing power. However, it can take advantage of the technology offered by on-chip routers, obtaining efficient communication. SpinMPI is also in charge of managing communication between the *MPI Runtime* running on the host computer and the SpiNNaker cores; this is done by using the ACP protocol and memory entities. This software stack creates a simple working framework

offering a universally known programming model capable of making the SpiNNaker architecture available for a wide range of applications.

We have succeeded in performing a benchmark of the SpiNNaker board by using a highly-parallel implementation of a DNA matching algorithm. Results show that the scalability of the SpiNNaker board reaches an ideal profile (98% of efficiency) when using more than 100 processors, a 90% efficiency using 600 processors, reaching 88% efficiency when all 767 application processors are used.

Author Contributions: Conceptualization, G.U. and F.B.; methodology, G.U., F.B. and A.A.; software, E.F., E.P. and F.B.; validation, E.F., E.P. and F.B.; formal analysis, E.F., E.P. and F.B.; investigation, G.U., E.F., E.P.; resources, G.U., A.A., and E.M.; data curation, E.P.; writing–original draft preparation, G.U., E.F., E.P. and F.B.; visualization, G.U., E.F. and E.P.; supervision, G.U.; project administration, G.U., A.A., and E.M.; funding acquisition, G.U. and E.M.

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

Acknowledgments: The research leading to these results has received funding from European Union Horizon 2020 Programme [H2020/2014-20] under grant agreement no. 785907 [HBP-SGA2].

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
FED	Fast string matching method for Encoded DNA sequences

References

1. Mead, C. Neuromorphic electronic systems. *Proc. IEEE* **1990**, *78*, 1629–1636. [CrossRef]
2. Boahen, K.A. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Trans. Circuits Syst. II Analog Digit. Signal Process.* **2000**, *47*, 416–434. [CrossRef]
3. Furber, S. To build a brain. *IEEE Spectr.* **2012**, *49*, 44–49. [CrossRef]
4. Liu, C.; Bellec, G.; Vogginger, B.; Kappel, D.; Partzsch, J.; Neumärker, F.; Höppner, S.; Maass, W.; Furber, S.B.; Legenstein, R.; et al. Memory-efficient deep learning on a SpiNNaker 2 prototype. *Front. Neurosci.* **2018**, *12*, 840. [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. [CrossRef]
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. [CrossRef]
7. 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, Genoa, 6–9 September 2017; pp. 273–276. [CrossRef]
8. Furber, S. Large-scale neuromorphic computing systems. *J. Neural Eng.* **2016**, *13*, 051001. [CrossRef]
9. 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.
10. Young, A.R.; Dean, M.E.; Plank, J.S.; Rose, G.S. A Review of Spiking Neuromorphic Hardware Communication Systems. *IEEE Access* **2019**. [CrossRef]

11. 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, Seoul, Korea, 20–23 May 2012; p. 702. [CrossRef]
12. 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]
13. 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]
14. Furber, S.B.; Galluppi, F.; Temple, S.; Plana, L. The spinnaker project. *Proc. IEEE* **2014**, *102*, 652–665. [CrossRef]
15. 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]
16. 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]
17. 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. [CrossRef]
18. 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]
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. 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*. [CrossRef] [PubMed]
21. 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]
22. Soni, K.K.; Vyas, R.; Sinhal, A. Importance of String Matching in Real World Problems. *Int. J. Eng. Comput. Sci.* **2014**, *3*, 6371–6375.
23. Boyer, R.S.; Moore, J.S. A fast string searching algorithm. *Commun. ACM* **1977**, *20*, 762–772. [CrossRef]
24. Horspool, R.N. Practical fast searching in strings. *Softw. Pract. Exp.* **1980**, *10*, 501–506. [CrossRef]
25. Reinert, K.; Dadi, T.H.; Ehrhardt, M.; Hauswedell, H.; Mehringer, S.; Rahn, R.; Kim, J.; Pockrandt, C.; Winkler, J.; Siragusa, E.; et al. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *J. Biotechnol.* **2017**, *261*, 157–168. [CrossRef]
26. Kim, J.W.; Kim, E.; Park, K. Fast Matching Method for DNA Sequences. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*; Chen, B., Paterson, M., Zhang, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 271–281.
27. Xue, Q.; Xie, J.; Shu, J.; Zhang, H.; Dai, D.; Wu, X.; Zhang, W. A parallel algorithm for DNA sequences alignment based on MPI. In Proceedings of the 2014 International Conference on Information Science, Electronics and Electrical Engineering, Sapporo, Japan, 26–28 April 2014; Volume 2, pp. 786–789.
28. Amdahl, G.M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In Proceedings of the AFIPS '67 Spring Joint Computer Conference, Atlantic, NJ, USA, 18–20 April 1967; pp. 483–485.
29. Gustafson, J.L. Reevaluating Amdahl's Law. *Commun. ACM* **1988**, *31*, 532–533. [CrossRef]

30. Intel Xeon Processor Scalable Family, Datasheet, Volume One: Electrical. 2018. Available online: https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-datasheet-vol-1.html (accessed on 1 November 2019).
31. Painkras, E.; Plana, L.A.; Garside, J.; Temple, S.; Galluppi, F.; Patterson, C.; Lester, D.R.; Brown, A.D.; Furber, S.B. SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation. *IEEE J. Solid-State Circuits* **2013**, *48*, 1943–1953. doi:10.1109/JSSC.2013.2259038. [CrossRef]

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).