Task Graph Mapping of General Purpose Applications on a Neuromorphic Platform

Indar Sugiarto*, Pedro Campos†, Nizar Dahir‡, Gianluca Tempesti§ and Steve Furber¶
*¶School of Computer Science, University of Manchester, United Kingdom
†§Department of Electronics, University of York, United Kingdom
‡IT Research and Development Center, University of Kufa, Iraq
*¶{indar.sugiarto, steve.furber}@manchester.ac.uk, †§{pedro.campos, gianluca.tempesti}@york.ac.uk
‡nizar.dahir@uokufa.edu.iq

Abstract—**A task graph is an intuitive way to represent the execution of parallel processes in many modern computing platforms. It can also be used for performance modeling and simulation in a network of computers. Common implementation of task graphs usually involves a form of message passing protocol, which depends on a standard message passing library in the existing operating system. Not every emerging platform has such support from mainstream operating systems. For example the Spiking Neural Network Architecture (SpiNNaker) system, which is a neuromorphic computer originally intended as a brain-style information processing system. As a massive many-core computing system, SpiNNaker not only offers abundant processing resources, but also a low-power and flexible application-oriented platform. In this paper, we present an efficient mapping strategy for a task graph on a SpiNNaker machine. The method relies on the existing low-level SpiNNaker's kernel that provides the direct access to the SpiNNaker elements. As a result, a fault tolerant aware task graph framework suitable for high performance computing can be achieved. The experimental results show that SpiNNaker offers very low communication latency and demonstrate that the mapping strategy is suitable for large task graph networks.**

Keywords—Task graph; mapping; neuromorphic; Spiking Neural Network Architecture (SpiNNaker)

I. Introduction

In distributed and parallel computing, high performance computation can be achieved by splitting a large set of tasks into a collection of independent operations [1]. These operations work in a stream of computation, in which the output of an operation is used as inputs to others. Visualizing this flow graphically, we get the task graph that represents the dependencies between operations. In a task graph, a vertex represent one independent operation, or task, and the connection between two vertices dictate the data transfer between those tasks. In this framework operations may be performed concurrently, or serially, depending on the availability of inputs to those operations.

Task graph implementation for high performance applications is gaining popularity especially in multi-core computing platforms [2]–[4]. Such a massive distribution of processing elements is expected to provide optimal power usage and maximizes performance while maintaining high reliability by tolerating component failures. This reliability issue must be addressed when designing the mapping strategy for the implemented task graph.

The mapping strategy from a task graph representation to the targeted platform depends on several constraints, such as the number of available processing elements and the networking infrastructure in the targeted platform [4], [5]. If the number of processing elements in the targeted platform is smaller than the number of tasks, then the presence of a scheduler plays an important role in the mapping since it is supposed to manages correctly the communication between dependent tasks. In a many-core system, on the other hand, the scheduler function can be implicitly integrated into the mapping scenario since designers have more flexibility in choosing which processing element will be assigned to a certain task. However, a new challenge will arise due to the communicating nature of the interconnected elements in many-core platform. This mapping strategy should not only designed for high performance, but also to accommodate the requirement of lower energy consumption.

In this paper, we explore one possible scenario that can be used for achieving both fault tolerance features and higher performance throughput. We use SpiNNaker, a neuromorphic many-core platform, for testing the mapping strategy. The platform was originally designed as a massively parallel spiking neural network capable of modeling a part of human brain [6]. However, it is also possible to deploy it as a general purpose high performance application, as the SpiNNaker system is built on top of generic ARM microprocessors. By implementing task graphs on SpiNNaker and measuring their data communication throughput, we are able to evaluate the performance of the proposed mapping strategy. To this end, the contribution of this paper can be summarized as follows.

- We propose a fault-aware task graph framework that can be extended into a concurrent multi-user platform.
- We implement a hybrid parallelism approach that addresses the challenge of a parallel programming paradigm on a neuromorphic system.
- We present the evaluation of task graph mapping on a neuromorphic system.

The paper is structured as follows. Section II describes work related to our paper, including a previously developed task graph generator program. In Section III, we describe the mapping strategy for a task graph on a SpiNNaker system. Section IV explains how the experiments were conducted and how the results are evaluated. Section V summarizes the study and explains our envisioned extension of the mapping strategy.

II. Related Work

A. Task Graph Representation

In this paper, we focus on task graphs as directed acyclic graphs (DAGs) that represent the flow of data and computational activities. It is an intuitive way to represent the execution of parallel processes in many modern computing platforms. Recently, it has become popular also for performance modeling and simulation in a network of computers. In its most basic form, a task graph *G* contains nodes (or vertices *V*) representing computational tasks, and edges *E* representing precedence constraints between tasks.

In a processor-bound system, a task graph may also be presented as $G = (V, E, w)$, where w is the weight function $w : V \to \mathbb{N}$ that gives weight (or duration) to the task. In this paper, we use a simpler task graph representation $G = (V, E)$, since we use a many-core system with high degree of redundancy that renders it as non processor-bound.

In the task graph, a directed edge $e_{i,j} \in E$ connects a node $i \in V$ to the next node $j \in V$ such that the processing in node j starts after it receives a certain amount of data resulting from the processing in node i. This creates a distribution of processing stages: nodes in higher processing stages have their dependencies in lower stages. In our work, we created two particular nodes with special functions. We designated the SOURCE node for a node that does not have predecessor(s), and it operates as the input port for sending data into the task graph. Another node, which is labeled as the SINK node and does not have successor(s), operates as the output port for sending data out from the task graph to other systems, such as other task graphs or a host computer.

Constructing a task graph automatically from the source code of an application can be very challenging and imposes the combination of analytical and pre-simulation of individual components [7]. In this paper we focus on the mapping of a task graph on the target platform and we assume that the task graph network has been constructed beforehand either automatically or manually. For the sake of simplicity, we use a tool for graph generation from [8], which is called XL-Stage. In this paper, the XL-Stage is used to generate example task graphs that will be implemented on a SpiNNaker machine.

In general, mapping a task graph into a target platform requires a scheduling strategy to obtain an efficient execution of the application graph. The efficiency can be achieved by optimizing some objective function, most usually the total execution time. The complexity of task scheduling scales up with the size of the graph and becomes an NP-complete problem for most optimization algorithms. Heuristic methods have attracted many researchers recently and provide convenient yet powerful way to solve optimization problems. One example of such a method is the evolutionary algorithm (EA), which is proposed in [9].

B. Hybrid Communication Protocol

In standard computing, among several parallel programming models, there are two that have been used widely. The first prominent model is the shared memory multiprocessing (SMP). It is widely accepted that for a multicore processor based system, the SMP approach is a convenient way to achieve high performance computation through parallelizing many tasks/processes. This multithreading mechanism implements the fork-join model that can be used to exploit task and/or data dependency of a program running on the same machine. One popular implementation of this mechanism is the OpenMP, which is available on most platforms and provides library routines directly callable by C, C++ and Fortran based programs.

With many-core system, on the other hand, the direct benefit of SMP is inhibited by the fact that the system memory is distributed across several chips and/or machines. In this infrastructure, the second model, which is based on a message passing technique, is favourable to achieve high scalability. The MPI (Message Passing Interface) provides a standardized communication protocol that is commonly used in a wide variety of parallel computing architectures including supercomputer clusters. In our work, we are inspired by the work of Nguyen et al. that introduces latency-tolerance MPI translation from C source code into a data-driven form [10].

It is natural to think that by combining both models, higher performance and scalability can be achieved [11]. Intel Cluster OpenMP is an example of a commercially available OpenMP framework that aims to increase scalability of OpenMP over a commodity cluster using a massage passing model [12] . With this spirit, we develop a task graph scheduling and mapping framework that emphasizes the hybrid utilization of complementary approaches: shared memory based parallelism and distributed computation via message passing. In the SpiNNaker infrastructure, the shared memory parallelism can be achieved by utilizing a MC (multicast) packet communication protocol; whereas message passing mechanism can be implemented using SDP (SpiNNaker Datagram Protocol). These protocols are described in Section III-B. MC communication protocol in SpiNNaker has been proved to support high performance computation (see for example [13], [14]); however, SDP is yet to be explored in any general purpose application on SpiNNaker.

III. Task Graph Mapping on a SpiNNaker Machine

A. Brief Overview of SpiNNaker System

SpiNNaker (Spiking Neural Network Architecture) is a neuromorphic system designed for emulating a massive spiking neural network in biological real time. In its top hardware structure, a SpiNNaker machine is made up of many SpiNNaker boards Going down to the chip level, each SpiNNaker chip is composed of up to 18 ARM low power processors, running at a modest frequency about 200 MHz. Currently, SpiNNaker construction is underway. When complete, it will host 64K chips; hence, it will have one million processors running in parallel.

Fig. 1 shows the currently available SpiNNaker machine. It is constructed using standard 19 inch cabinets (up to 10 cabinets in total) with 120 SpiNNaker boards hosted in each. The SpiNNaker board (also shown in Fig. 1) contains 48 SpiNNaker chips. The SpiNNaker chip itself is actually a system-on-chip (SoC) with a 128MB SDRAM mounted on top of the microprocessor die. The chip also incorporates a network-on-chip (NoC), capable of driving inter- and intra-chip communication protocols.

Although originally designed for simulating spiking neural networks, SpiNNaker is also envisioned to be a general computing machine with a neuromorphic paradigm: running at low power consumption and having high reliability by virtue of high redundancy. The multi-core fabrication of the SpiNNaker chip was also for anticipating flawed devices. The idea was: if there were one or a few malfunction cores, the entire chip should not be entirely shut down. In fact, only 65% of the batches were flawless chips and the remaining chips usually had 17 or fewer working cores [6]. This will provide a high degree of redundancy as the foundation of the SpiNNaker fault tolerance architecture. It can be used to protect against run-time faults by offloading work. Normally, a SpiNNaker program is expected to spare at least one core as a stand-by core on each node so that it can accommodate a run-time fault without too much effort since the majority of the data is held in a separate but shared SDRAM.

Regarding the software stack, SpiNNaker follows the route of a library operating system. An application program that will be implemented on SpiNNaker will be compiled against the SpiNNaker library. Thus, when the application runs on SpiNNaker, it will interact with the underlying run-time kernel already present in the machine, which is called SpiNNaker Application Run-time Kernel (SARK). In our work, we extend the functionality of SARK such that future task graph-based applications can run smoothly without too many low level SARK's routines calls.

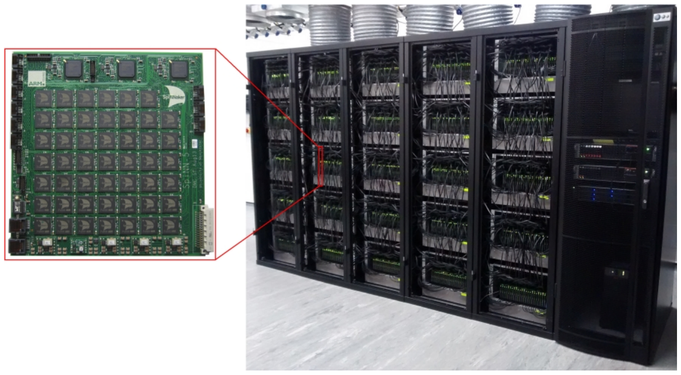

Fig. 1. Semi-completed part of the SpiNNaker machine. Each cabinet hosts 120 SpiNNaker boards, and each board (shown to the left) has 48 SpiNNaker chips.

B. SpiNNaker Networking Infrastructure

Each SpiNNaker chip has six bidirectional links to other chips, and in a fully interconnected SpiNNaker network, a doughnut-shape torus can be created. However, if only a few SpiNNaker boards are connected, the SpiNNaker networking topology will be a two-dimensional triangular torus. Fig. 2 shows the interconnection layout of three SpiNNaker boards (as used in this paper).

The NoC inside each SpiNNaker chip is responsible for managing various event-driven communication protocols. The router module in the NOC is responsible for delivering many small SpiNNaker packets to one or more cores in the chip and/or to the external links. The SpiNNaker router has a simple architecture in which ports are hierarchically merged into a single pipelined queue so that only one packet can use the

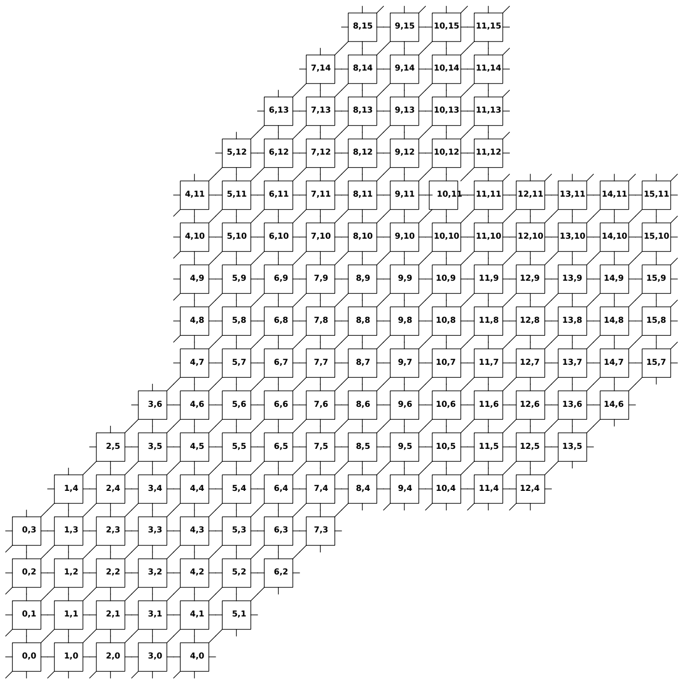

Fig. 2. An example layout of three SpiNNaker boards connected in a non-torus topology as used in this paper. Different layout may be produced for different cabling connection on those boards.

routing engine at once. It has a high speed content addressable memory module to store the routing table, allowing it to run at high bandwidth of about 8 Gbps; hence, it can avoid a bottleneck since the router load will be very low and will not burden the packet processing by the ARM cores. However, due to many factors of low power chip design and fabrication, the actual network link throughput is measured to be limited at roughly 250 Mbps.

The communication protocols in SpiNNaker system can be in a form of multicast mode or point-to-point mode. In either case, the router supports either 40 or 72 bits data packets. The multicast (MC) communication is the foundation of spiking neural network simulation on SpiNNaker. This communication protocol has a remarkable characteristic such that it reduces the pressure at the injection ports and the number of packets traversing the network [13]. In the case of point-to-point communication, a higher communication protocol can be constructed by encapsulating several packets into one stream which is called SpiNNaker Datagram Protocol (SDP). However, this higher level encapsulation has a cost in a higher processing overhead at the processing core; hence, the SDP has lower bandwidth compared to the MC communication.

In this work, both communication protocols (MC and SDP) are used for different purposes. The MC communication is mainly used inside the chip for emulating hardware multithreading useful for task duplication and parallelism, but also outside the chip for process migration purposes.The SDP communication, on the other hand, is primarily used for the message passing mechanism between nodes in a task graph.

We use the SDP as the main communication protocol in the task graph framework for the following reasons:

1) Data flow in a task graph is basically a point-to-point mechanism and SDP, which is constructed from SpiNNaker Point-to-Point (P2P) packets, is most suitable for this purpose. Furthermore, an SDP packet can carry a payload of up to 272 bytes, much more than any other communication protocol in SpiNNaker.
2) Although the MC protocol runs faster than SDP, it is very difficult to manage its routing at run time. This makes the process migration very difficult to be implemented using MC packets.

An SDP packet is constructed from several lower level P2P packets. It consists of a 10-byte header, which contains source and destination addresses of communicating cores, and a data segment, which can be used for general purpose communication. The default routing of P2P packets is determined during the boot-up process of the SpiNNaker kernel. Once the boot-up process is completed, each chip will hold a P2P routing table that contains 64K entries. Each entry determines which of the six chip's links will be used to deliver a P2P packet to all other chips in the machine. An example configuration of P2P routing table in a single SpiNNaker board is shown in Fig. 3. The figure shows which link will be used to deliver P2P/SDP packets to the chip at coordinate (0,0). As we can see from the figure that the direction of such a routing is not uniform; some chips prefer the horizontal link rather than the diagonal link as used by majority chips. This condition might have an impact on the task graph communication channels as described later in this paper.

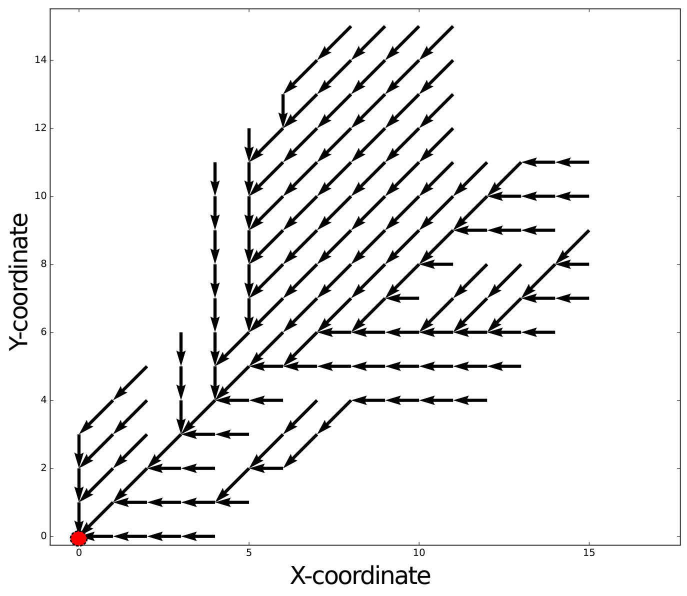

Fig. 3. Graphical representation of the default P2P routing towards chip (0,0) that was defined during the SpiNNaker boot-up process. The arrow indicates the preferred link selected by the corresponding chip when sending P2P packets to chip (0,0).

C. Task Mapping on a SpiNNaker Machine

As described in the previous subsection, the SpiNNaker system was originally designed for simulating massive spiking neural network; thus, the multicast communication protocol plays very important role in the SpiNNaker framework to mimic the similar mechanism of information broadcasting in the spiking neural model of the brain. However, we argue that this multicast fashion is not the best mechanism to carry information in a point-to-point communication centric such as in a task graph. Hence, we developed a mapping strategy for task graphs on SpiNNaker using the SDP mechanism, even though it runs slower due to its pre-processing overhead.

Regarding nodes mapping into SpiNNaker resources, we opt to use one-to-one mapping such that one node in a task graph will be mapped into one SpiNNaker chip. The selection of a mapping scenario is determined by the preferred task granularity. In general, a larger task granularity will reduce communication and task creation overhead. However, smaller grained tasks may result in better load balancing. In this circumstance, it is possible to create arbitrary mapping for flexible task granularity, such as one-to-many or many-to-one mapping. We are also working on this aspect as a part of an extension as described in Section III-C2. Nonetheless, for the sake of simplicity, they are not evaluated in this paper.

With one-to-one mapping, the communication overhead can be minimized. In the one-to-one mapping, MC packets are mainly utilized for intra-chip communication, such as master-worker thread coordination and fast data sharing. Thus the entire network will not be polluted by intensively traversing MC packets between chips. However, small MC packets are still circulated outside the chip to accommodate the process migration in the fault tolerance scenario. Also with this one-to-one mapping, we can use the SDP packets to implement message passing protocol for task graphs.

1) Redundancy and Task Migration for Fault Tolerance: The SpiNNaker platform inherently provides the basic infrastructure for fault tolerance. At a chip level, it has a low power multi-core processor system, equipped with fault-aware NoC and a clock-less asynchronous communication channel. On top of these, higher level software-defined fault-tolerance features can be developed, such as emergency routing, dropped packet re-insertion, watchdog mechanism, etc. [6]. In this work, we introduce an additional fault handling mechanism via task migration as an inherent part of the task graph framework.

The task graph framework is intended to be the underlying mechanism of parallel- and distributed computation to achieve high performance computing on many-core systems. In a many-core system such as SpiNNaker, fault tolerance is a compulsory aspect that needs to be handled properly to maintain service reliability of the system. Here, we take advantage of a high degree of redundancy in SpiNNaker resources and provide additional feature for fault tolerance by providing a task migration capability for a task graph based applications. The source of a fault that triggers the task migration can be from anywhere in the system, either from hardware or software. For experimental purpose, we created simulated faults by manually trigger a node to start the migration process.

Regarding the impact of migration on a running task graph, we are interested to evaluate the overall performance of the running program after migration takes place. This is important for evaluating the performance of the proposed task graph mapping scenario. Migrating a task from one node to another node in the mesh might introduces performance degradation that affects the overall performance. Thus, we evaluate this migration effect as presented in Section IV-B.

2) Task Duplication to Improve Performance: In this paper, we define a node in a task graph as a unit of processing that will be implemented on a single SpiNNaker chip. This constraint should be used for any other automatic task graph generation program that relies on our task graph framework.

The framework uses the task graph description given by the XL-Stage program (see [8]). Here, a task has an output triggering condition that depends on its input dependency. For example, the node P4 in a 9-node task graph shown in Fig. 4 has three outputs that depend on two inputs. The zoomed-in representation shows the triggering condition for each output. For example, output to node P5 and P6 depends on P3: after receiving 1 packet from P3 the node P4 will generate 7 packets to node P5, but node P4 will generate 4 packets to node P6 only after receiving 5 packets from node P3. Whereas output to node P7 depends on both P2 and P3: 3 packets will be sent to P7 after P4 receives 4 packets from P2 and 4 packets from P3.

For implementing such a traffic model on SpiNNaker, we created a task splitting and spawning mechanism. Although the concept of task splitting and spawning is rather well defined in the literature, their practical implementation on many-core systems is far from worked out or trivial. In this work, we took the advantage of the multi-core properties of a SpiNNaker chip: we spawned the task on several cores, and then modified the output link of each spawned task to a single target node. With this scenario, we expect nodes in task graphs to have the maximum output link of 16 nodes, which corresponds to the number of available cores in a SpiNNaker chip.

To improve the performance further, we applied an amelioration strategy such that a task may also be copied/multiplied on empty cores in the chip. These identical copies can then use the shared-memory mechanism to harness hardware multi-threading. A similar mechanism has been used successfully to achieve high performance image processing on SpiNNaker as described in [14]. The zoomed-in representations of node P4 and P8 of the 9-node task graph illustrate this amelioration strategy. For example in node P4, the task that targets P5 has been duplicated on a set of cores C1,C4,C5,C6,C7. This set of cores then uses MC packets inside the chip for creating shared-memory parallelism similar to the standard OpenMP protocol used in conventional parallel computing. Likewise, a set of cores C2,C8,C9,C10,C11 works in parallel targeting node P6. In this example, we use 5 cores for parallel processing mechanism in node P4, which leaves the remaining two cores of the corresponding SpiNNaker chip in an idle state. These idle cores will then serve as the backup core to support fault tolerance property of our task graph framework.

Similarly, node P8 in Fig. 4 will have higher degree of both parallelism and redundancy. As illustrated, the node has 10 cores for parallelizing task P8 and has 7 backup cores for fault tolerance support. Hence, one might think that this approach is not the optimal one, because the load is not balanced across the entire SpiNNaker system. We are aware of this condition and we are also working on an extension with different splitting strategies, by which a task is also distributed to other free chips. However, for the sake of simplicity, we do not cover this advanced and complex strategy in this paper.

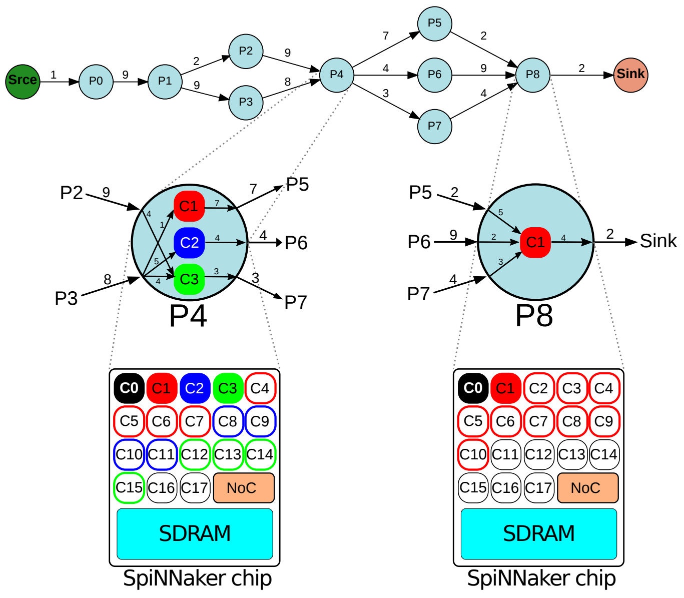

Fig. 4. An example 9-node task graph for our experiment. The number associated with an edge represent the number of packets generated by the "parent" node to its "child" node. The zoomed-in representations of node P4 and P8 illustrate task splitting and spawning mechanisms inside the SpiNNaker chips, as well as simple amelioration strategy by task duplication. Some cores remain idle and can be used for fault tolerance purpose. The black-colored core in each chip is a special core called "Monitor", which is dedicated for SpiNNaker kernel management and cannot be used by any other user/application.

IV. Evaluation and Discussion

A. Experimental Set-up

In this paper, we are interested to evaluate both the SpiNNaker networking infrastructure and the overall performance of the task graph framework. Hence, we conducted two experiments to address these two issues.

1) Intrinsic Latency: In the first experiment, we were interested to evaluate the intrinsic latency of the SpiNNaker machine. In this experiment, we mapped task graphs on a SpiNNaker platform and distributed the nodes according to the distance-adjacency (DA) parameter listed in Table I. We created two scenarios: single-app mapping and multi-app mapping. In the single-app mapping, we used only one network shown in Fig. 4 and distributed the nodes according to the DA parameter listed in Table I. The purpose of this scenario is to evaluate the intrinsic latency due to the inter-chip communication delay of the SpiNNaker platform when only one application running on it. We expect some graceful degradation on the overall performance as the nodes move away from their initial position.

In the multi-app mapping, we run three task graph networks: two instances of the network shown in Fig. 7 in addition to the evaluated network shown in Fig. 4. The purpose of this second scenario is to evaluate the future use of our task graph framework on SpiNNaker. In future, we expect many users/applications will run on a single SpiNNaker machine concurrently.

In both scenarios, the task migration for the network shown in Fig. 4 is involved: when switching from one DA experiment to another, the tasks are moved further. Fig. 5(a) shows the experiment for single-app scenario using DA-0, where each task graph node is mapped directly close to the others. Fig. 5(b) shows the experiment for single-app scenario using DA-1, where one idle node is placed between two active nodes. Fig. 5(c) shows the experiment for single-app scenario using DA-2, where two idle nodes are placed between two active nodes, signifying further separation that might introduce additional communication delays. Fig. 6 shows the experiment for multi-apps scenario using DA-3, where two instances of 25-node task graph shown in Fig. 7 (shown as green and blue nodes) are mapped and run alongside the evaluated 9-node task graph. Here, the nodes of 9-node task graph are placed regularly and spaced by three other nodes (either by idle nodes or by nodes from the 25-node network).

TABLE I. DISTANCE-ADJACENCY PARAMETER

Parameter	Description
DA-0	Direct adjacency: no spacing between active nodes.
DA-1	Active nodes are spaced using one idle node (or using a node from another running task graph).
DA-2	Active nodes are spaced using two idle nodes (or using nodes from another running task graph).
DA-3	Active nodes are spaced using three idle nodes (or combination of idle nodes and nodes from another running task graph).

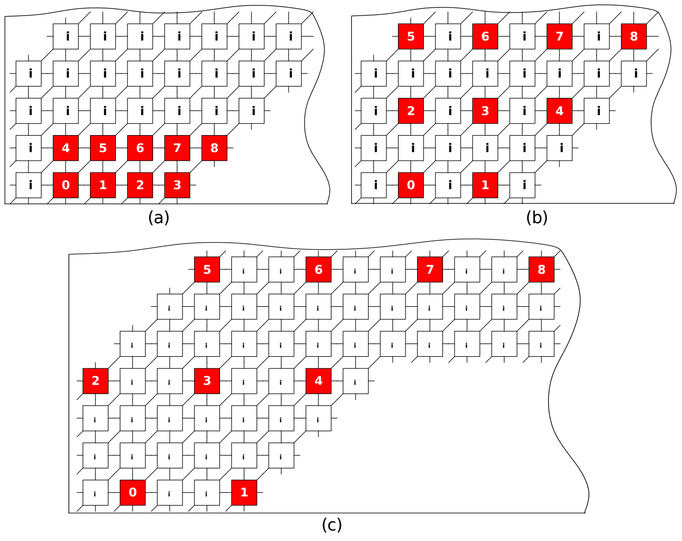

Fig. 5. The mapping of the 9-node task graph shown in Fig. 4 for different DA parameters in the single-app scenario.

2) Overall Performance: In the second experiment, we are interested to evaluate the overall performance of several possible task mappings on SpiNNaker. Here, we use 25-node task graph shown in Fig. 7. For this experiment, we used the mapping produced by the EA approach from [9]. The fault tolerance mechanism is achieved by providing idle nodes as backups for malfunctioning nodes. One important metric used as the objective function in the EA approach is the distance from a running node to a nearby idle node. In this regard, we are interested to evaluate the overall performance of a task graph running on a SpiNNaker platform when the nodes are moved around to create empty space of idle nodes.

For this experiment, we used two scenarios. The first scenario uses a quad-link mesh topology, in which the number

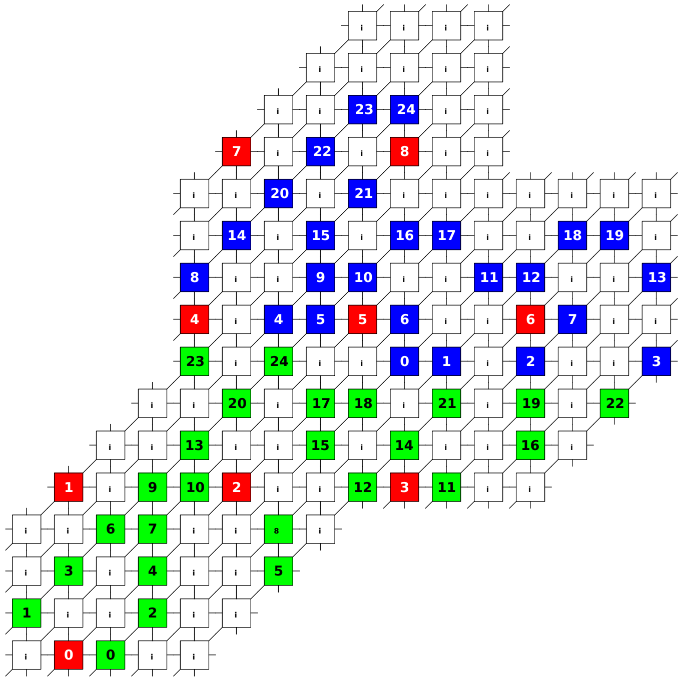

Fig. 6. The mapping of three task graphs in the multi-app scenario. Red nodes belong to the 9-node task graph shown in Fig. 4, whereas green and blue nodes belong to two 25-node task graph shown in Fig. 7. The nodes of two larger graphs are placed among the nodes of 9-node graph in order to evaluate the effect of overlapping maps.

of links of each SpiNNaker chip is reduced to four; the diagonal links (labeled as NorthEast-link and SouthWest-link) were disabled. This is to closely mimic the condition assumed in [9]. The second scenario uses the hexa-link topology, in which all links in each SpiNNaker chip were activated. This is to evaluate the impact of additional links provided by the SpiNNaker system compared to the conventional rectangular 2D-mesh network. For both scenarios, three maps generated by the EA as shown in Fig. 8 were used. Table II describes the characteristic of those three maps.

TABLE II. THREE MAPPING SCENARIOS FOR THE 25-NODE TASK GRAPH SHOWN IN FIG. 7

Map	Description
MAP-1	Good fault tolerance with poor performance: at least one idle node is preserved for each active node.
MAP-2	Good performance and poor fault tolerance: the distance between coupled nodes are kept close while idle nodes are not guaranteed directly adjacent to those nodes.
MAP-3	Balance between performance and fault tolerance: the availability of backup/idle nodes are guaranteed while preserving the close distance between coupled active nodes.

B. Performance Evaluation

1) Intrinsic Latency: In the first experiment, we were interested to investigate the effect of SpiNNaker intrinsic latency to the overall performance of a task graph running on it. Conceptually, by moving further nodes the task graph might experience some additional delays. For this, we run two scenarios: single-app mapping and multi-app mapping. The result is presented in Fig. 9 that shows the number of received packets at the SINK-node of the task graph shown in Fig. 4 as the function of DA parameters (see Table I).

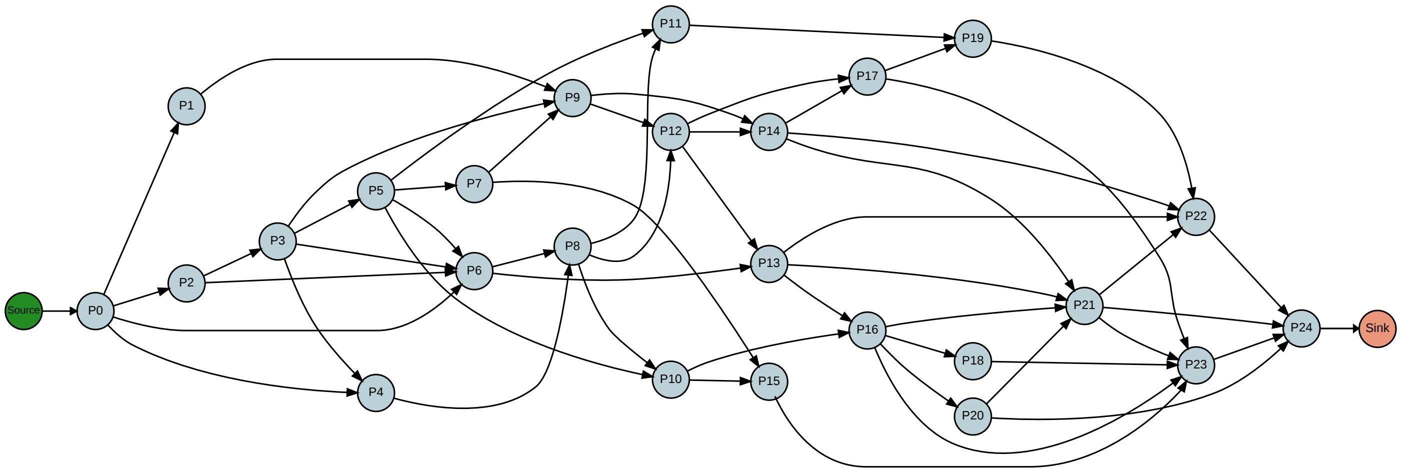
Fig. 7. An example task graph with 25 nodes.

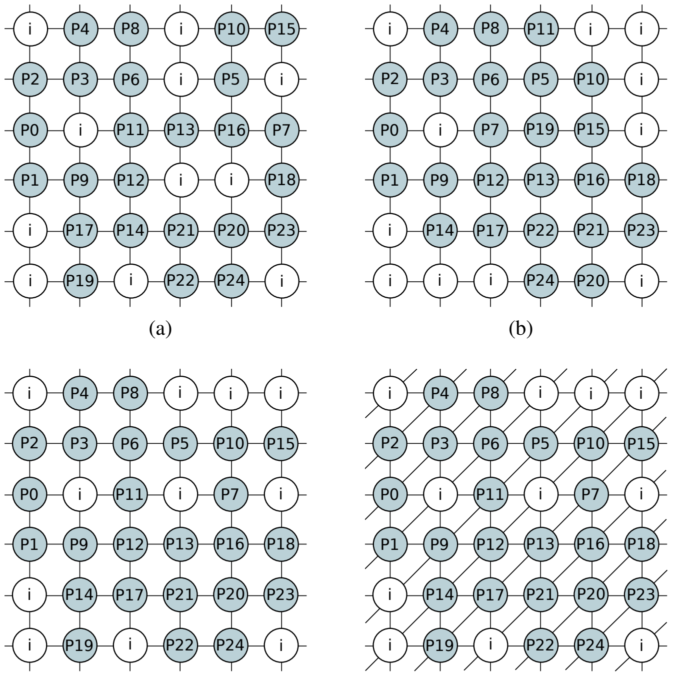

Fig. 8. Mapping the task graph shown in Fig. 7, exhibiting (a) good fault tolerance but poor performance, (b) poor fault tolerance but good performance, (c) balance between fault tolerance and performance, (d) similar to (c) but in original SpiNNaker triangular mesh.

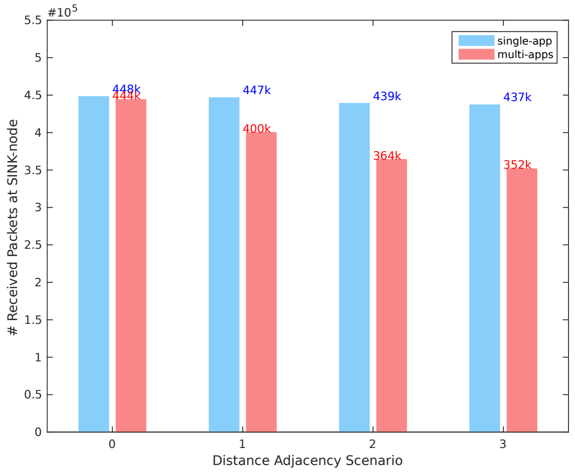

Fig. 9. The distance adjacency plot presenting the number of received packets at the SINK-node of the 9-node task graph as a function of DA parameters described in Table I. The multi-app plot shows a noticeable impact of the overlapping maps.

From the plot, we can make two observations. First, the distance between nodes indeed gives some delay. However, for a single application graph the delay is minuscule, which shows the performance drop only 2.45% when nodes were moved far away (using DA-3 scenario). In normal fault aware situation as suggested in [9], the performance drop is much reduced to only 0.22% (obtained from DA-1 scenario).

Second, running multiple task graph applications on SpiNNaker reduces the overall performance, especially when the nodes are scattered across the mesh and the network overlaps with other networks. The worst case scenario shows the performance drop up to 20.72% (using DA-3 scenario), but the normal fault-aware scenario (using DA-1) shows the drop only 9.9%. However, when we isolated the three task graphs so that they did not overlap, the performance drop returned to the normal rate of about 0.2% in the DA-1 scenario.

By examining the traffic report produced by the SpiNNaker kernel, we observed there were several P2P packet drops indicating a congestion on some SpiNNaker links. Unfortunately, the kernel does not have a native mechanism to re-inject the packet; the affected nodes simply wait for another packets, thus reducing its final throughput. One possible option for reducing the number of dropped P2P packets is by re-injecting the dropped packet back to the network. This re-injection mechanism is currently being developed for the main SpiNNaker software stack, which provides a programming environment for developing spiking neural network applications. In the future, this re-injection program might be integrated into the SpiNNaker kernel so that any program beyond spiking neural network, such as our task graph framework, can take its advantage to improve the performance.

From this experiment, we can see that multiple task graphs can run simultaneously on one machine instance as long as their mappings are not overlap. Conceptually we can run a very large task graph as long as the number of nodes is smaller than the number of available chips in the SpiNNaker machine.

2) Overall Performance: In the second experiment, we implemented three mapping examples for the 25-node graph example case. These three mappings were produced by the EA program representing possible combination of two metrices: fault tolerance coverage and communication performance. We measured the number of SDP packets arriving at the SINK node of the graph in two scenarios: using a modified rectangular mesh and the original SpiNNaker triangular mesh. Table III shows the result of both scenarios.

TABLE III. The Overall Throughput Measured as the Number of SDP Packets Arrive at the SINK Node

	Map-1	**Map-2**	**map-3**
□ **mesh**	14700	15267	15033
△ **mesh**	14650	15050	14800

From Table III, we can see that Map-2, which was indicated as the best performance map by the original author in [9], shows the highest throughput for both scenarios. Concerning the impact of the two different mesh topologies, we can see that the standard 2D rectangular mesh produces a throughput slightly higher than the triangular mesh. On average, the task graph in the triangular mesh topology produces 1.1% less throughput than that in the rectangular mesh topology. This was a surprising phenomenon that we observed in the experiment. But when we looked again into the traffic report produced by the SpiNNaker kernel, we found that the decreasing throughput of the task graph running on a triangular mesh topology was due to higher P2P packet drops. This is reported in Table IV.

TABLE IV. Number of Dropped P2P Packets

	Map-1	**Map-2**	**map-3**
□ **mesh**	700	790	596
△ **mesh**	760	812	658

As we can see in Table IV, the dropped P2P packets in the triangular mesh experiment is slightly higher than that in the rectangular mesh experiment. This fact can be understood by looking into the P2P routing table, the nodes' dependency values, and the mapped task graph shown for example in Fig. 8d. From the reported P2P traffic, we found that node P14 and node P17 are two of busiest nodes in the network. These two nodes contribute much traffic to the late-stage nodes P21 to P23, which happen to be mapped very close to each other. The burstiness of traffic from node P14 and node P17 creates severe traffic locality on those late-stage nodes. This was confirmed by looking at the P2P routing table that showed possible congestion on the link between nodes P14/P17 to those late-stage nodes P21-P23.

Since there is no traffic management algorithm implemented on the SpiNNaker kernel, traffic from P14 and P17 was not re-routed to different path. It seems that even though the triangular mesh topology offers additional channels, without proper traffic management algorithm some links becomes congested preventing higher throughput to be produced in this scheme. This observation confirms the fact that traffic locality and burstiness have a noticeable impact on the performance of the interconnection network of SpiNNaker as reported in [15].

V. Conclusion

This work evaluates fault-aware mapping strategies to achieve high reliability for general purpose applications running as task graphs. The resulting map is targeted on a SpiNNaker many-core neuromorphic platform, which can be considered as a non-standard high performance computing platform. The SpiNNaker machine, which is designed for hosting a million ARM processors, was used as the platform, and the task graph mappings resulting from the evolutionary algorithm (EA) used in [9] were used for test cases.

In this paper, we utilize the low-level SpiNNaker kernel that provides direct access to the SpiNNaker hardware resources, especially its network-on-chip (NoC) component. We developed a framework with a hybrid parallelism scenario that fully utilized the multi-mode parallelism protocols of SpiNNaker: from shared-memory based to message passing mechanism.

Our experiment with the intrinsic latency (see Section IV-B1) demonstrates that SpiNNaker machine reliably supports the implementation of a parallelized application running as a task graph with slight caution. As in many conventional high performance computing platforms, the communication overhead creates delays that get longer by the increasing distance between corresponding processing units. In the experiment, running a single task graph produces a negligible latency. However, running several overlapping networks on the same SpiNNaker machine produces a noticeable impact on the performance. In our current work, however, this problem can be easily solved by running several task graph networks on several SpiNNaker boards. Those boards can be easily isolated in the SpiNNaker machine, thus rendering the task graph networks independent of each other. From this experiment, we are confident that SpiNNaker is a reliable platform to implement our fault tolerance task graph framework.

We have also conducted an experiment that evaluates the impact of bidirectional hexagonal links in SpiNNaker chips on previously generated and optimized maps. In the experiment, first we modified the existing routing topology of the SpiNNaker machine to mimic a conventional 2D rectangular mesh, and then applied some mapping scenarios on it. Secondly, we reverted to the default original SpiNNaker 2D triangular mesh topology, and applied the same mapping again. From this experiment, we observe that as the number of communication links increases, the number of dropped packets also increases. This happens because the SpiNNaker's NoC router does not manage traffic pattern of the running application. This result can lead to larger investigations that deal with an adaptive power and traffic management system for online optimization in the SpiNNaker system. We regard this as our future work.

Acknowledgments

The authors would like to thank Engineering and Physical Sciences Research Council (EPSRC) for supporting our work through the Graceful project grant (EP/L000563/1). The design and construction of the SpiNNaker machine was also supported by EPSRC under grants EP/D07908X/1 and EP/G015740/1. Ongoing development of the software is supported by the EU ICT Flagship Human Brain Project (FP7-604102).

References

[1] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966. [Online]. Available: http://dx.doi.org/10.1137/0114108

[2] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini, "A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness," *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 3–36, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10766-007-0032-7

[3] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[4] J. Kurzak and J. Dongarra, "Fully dynamic scheduler for numerical computing on multicore processors – lapack working note 220," 2009. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn220.pdf

[5] Y. Robert, *Task Graph Scheduling*. Boston, MA: Springer US, 2011, pp. 2013–2025. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_42

[6] J. Navaridas, S. Furber, J. Garside, X. Jin, M. Khan, D. Lester, M. Lujn, J. Miguel-Alonso, E. Painkras, C. Patterson, L. A. Plana, A. Rast, D. Richards, Y. Shi, S. Temple, J. Wu, and S. Yang, "Spinnaker: Fault tolerance in a power- and area-constrained large-scale neuromimetic architecture," *Parallel Computing*, vol. 39, no. 11, pp. 693 – 708, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113001051

[7] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon, "Poems: end-to-end performance design of large parallel adaptive computational systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 11, pp. 1027–1048, Nov 2000.

[8] P. Burmester Campos, N. Dahir, C. Bonney, M. Trefzer, A. Tyrrell, and G. Tempesti, "XL-STaGe: A cross-layer scalable tool for graph generation, evaluation and implementation," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XVI)*, Samos, Greece, May 2016.

[9] C. Bonney, P. Campos, N. Dahir, and G. Tempesti, "Fault tolerant task mapping on many-core arrays," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Athens, Greece, Dec 2016, pp. 1–8.

[10] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, "Bamboo – translating MPI applications to a latency-tolerant, data-driven form," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[11] J. Hoeflinger, B. Kuhn, W. Nagel, P. Petersen, H. Rajic1, S. Shah, J. Vetter, M. Voss, and R. Woo, *An Integrated Performance Visualizer for MPI/OpenMP Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 40–52. [Online]. Available: http://dx.doi.org/10.1007/3-540-44587-0_5

[12] C. Terboven, D. a. Mey, D. Schmidl, and M. Wagner, *First Experiences with Intel Cluster OpenMP*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 48–59. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79561-2_5

[13] J. Navaridas, M. Luján, L. Plana, J. Miguel-Alonso, and S. Furber, "Analytical assessment of the suitability of multicast communications for the spinnaker neuromimetic system," in *IEEE International Conference on High Performance Computing and Communications (HPCC 2012)*, Liverpool, United Kingdom, Jun 2012, pp. 1–8.

[14] I. Sugiarto, G. Liu, S. Davidson, L. A. Plana, and S. B. Furber, "High performance computing on spinnaker neuromorphic platform: A case study for energy efficient image processing," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, Dec 2016, pp. 1–8.

[15] J. Navaridas, L. A. Plana, J. Miguel-Alonso, M. Luján, and S. B. Furber, "Spinnaker: Impact of traffic locality, causality and burstiness on the performance of the interconnection network," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787278