Optimized Task Graph Mapping on a Many-core Neuromorphic Supercomputer

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—**This paper presents an approach for improving the overall performance of a general purpose application running as a task graph on a many-core neuromorphic supercomputer. Our task graph framework is based on graceful degradation and amelioration paradigms that strive to achieve high reliability and performance by incorporating fault tolerance and task spawning features. The optimization is applied on an instance of the task graph by performing a soft load balancing on the data traffic between nodes in the graph. We implemented the framework and its optimization on SpiNNaker, a many-core neuromorphic platform containing a million ARM9 processing cores. We evaluate our method using several static mapping examples, where some of them were generated using an evolutionary algorithm. The experiment demonstrates that a performance improvement of up to 8.2% can be achieved when implementing our algorithm on a fully-utilized SpiNNaker communication infrastructure.**

1. Introduction

Research in many-core systems and high performance computing, with hundreds or thousands of cores, requires applications to be developed in an efficient way that makes use of parallelism strategies. Also, in such systems, a benchmark tool for design-space exploration is required. As opposed to real-applications, which are hard to customize (e.g. change the size, communication requirements, processing workload), these benchmarks enable rapid performance evaluation with various application characteristics such as the number of tasks, task dependencies, and input/output degrees of the tasks. For this, task graph models are developed and used for creating synthetic benchmarks that are easy to customize to meet the evaluation/exploration needs of the system designer.

In this paper we present an implementation of a cross layer task graph framework that is dedicated for modeling temporal behavior and the process model of parallel programs. The framework was implemented on SpiNNaker (Spiking Neural Network Architecture), a massive many-core system with neuromorphic-style communication infrastructure. SpiNNaker is one of the neuromorphic platforms brought into the EU Flagship Human Brain Project (HBP) with an ultimate goal to emulate up to 1% of the human brain. The SpiNNaker chip, which is the basic building block of the SpiNNaker machine, is populated with 18 low power ARM9 processors, each capable of 220 DMIPS at 200MHz. Together, those processors in the chip can be paced synchronously to achieve 3600 MOPS (million operations per second), and consumes less than 1 Watt. Hence, the SpiNNaker machine, which will be constructed from 50K of such chips, paves the way to be one of the "greenest" supercomputers in future.

Our paper tries to address the question of how this massive neuromorphic platform can be used for general purpose high performance computing. We created a task graph framework by extending the functionality of the SpiNNaker run-time kernel. The paper also describes how we optimize the mapping to achieve higher throughput for the task graph. The mapping and its optimization is a critical stage of any task graph based application running on a Networks-on-Chip (NoC) platform such as SpiNNaker. Several papers report that the mapping strategy can have a high impact on the overall system performance [1], [2], [3], [4].

Our optimization strategy runs on any existing map. In this paper we use static mappings generated by an evolutionary algorithm as proposed in [5]. The basic idea of our method is the balancing of data traffic across the SpiNNaker mesh. The load-balancing can be achieved by migrating the busiest nodes from a dense region into a lower-traffic region, creating less-stressed network traffic that leads to a higher packet transaction among nodes. To this end, the contribution of this paper can be summarized as follows.

- We evaluate task graph mappings on a many-core neuromorphic platform (SpiNNaker).
- We use heuristic-based mapping examples and improve their performance by soft load balancing.
- The proposed improvement strategy can also be applied on any other mapping scenario.

The presentation in this paper is organized as follows. Section II describes basic concepts of a task graph and its mapping. In Section III, we describe our task mapping strategy on SpiNNaker. Section IV explains how the experiments were conducted and how they were evaluated. Section V summarizes the results and concludes our work.

2. Related Work

2.1. Distributed Computation using Task Graphs

High performance computing systems are commonly used to harness the power of parallel processing for running sophisticated programs. The computation can naturally be described as a graph, in which vertices represent tasks and/or processes and edges reflect data dependencies. Such a representation, which is called a task graph, provides a convenient way to model the performance of parallel programs.

There are several variant of task graphs. In the standard form, a task graph is presented as a graph $G = (V, E)$, where 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. In this representation, the impact of scheduling strategy is implicit and can be diminished by the redundancy in a many-core system.

The majority of application task-graph models express the application as a set of nodes (tasks) and arcs (dependencies) usually in the form of a DAG (directed acyclic graph) and may include the communication requirements among the tasks. However, these graphs usually lack models for the temporal behaviour of these tasks and the process model. This narrows the use of these models to high-level design-space exploration. To overcome this issue, authors in [6] proposed a cross-layer tool for DAG generation and implementation. It consists of a set of tools that generate the task-graph as well as the process models for each node. In our work, we use this tool to generate synthetic task graphs useful for evaluating the SpiNNaker in general purpose computing scenarios beyond its neuromorphic origin.

2.2. Heuristic Task Graph Mapping

The challenge of optimal task-to-node allocation in a many-core array creates a very large solution space, especially when multiple objectives are considered. A many core system presents a large number of candidate nodes for a particular task, but the complexity increases exponentially once multiple applications are mapped to the same array. This transforms the mapping problem in task graphs into an optimization problem. The efficiency of mapping can be achieved by optimizing some objective functions, mostly the total execution time.

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 presents a promising strategy for the exploration of very large solution space. In this paper, we use the EA-based mapping proposed in [5] for mapping a task graph into the SpiNNaker platform. The EA in [5] was applied on a many-core virtual platform. Hence, it did not take into account the pre-processing overhead of the communication channel of a real platform. When we implemented the algorithm on SpiNNaker, we found that some nodes become congested by the traffic. Hence, we propose an improvement by load balancing the communication overhead as described in section 3.3.

Our method, which we call scattering-and-grouping approach, performs load balancing by migrating tasks from a dense region to other more relaxing region in the mesh. A similar approach, called scatter-gather operations, is proposed by Kumar et al. in [7]. The difference between their approach to ours is that their approach works for optimizing processor's cache utilization at source-code level; whereas our method works at the node level through a task migration mechanism.

3. Task Graph Mapping on SpiNNaker

3.1. Brief Overview of SpiNNaker System

SpiNNaker is a many-core computing platform originally designed for emulating a massive spiking neural network in real time. The SpiNNaker machine is made up of many multi-core SpiNNaker chips. Currently, the full SpiNNaker machine construction is underway and when complete, it will host 50K chips; hence, it will have one million processors running in parallel.

Fig.1 shows the currently available SpiNNaker machine. The machine is constructed using standard 19" cabinets (up to 10 cabinets in total). Each cabinet hosts 5760 SpiNNaker chips, which are mounted on 120 SpiNNaker boards. The SpiNNaker board (also shown in Fig.1) contains 48 SpiNNaker chips. Each chip contains 128MB SDRAM mounted on top of the microprocessors die.

Each SpiNNaker chip has six bidirectional links to connect to other chips with conceptual bandwidth of up to 250 Mbits/s. In a fully interconnected SpiNNaker network, a doughnut-shape torus can be created. If only a few SpiNNaker boards are connected, then the SpiNNaker networking topology will be a two-dimensional triangular mesh.

The chip also incorporates a network-on-chip (NoC) capable of the driving lightweight packet-switched communications fabric. The communication protocols in SpiNNaker system can be in a multicast (MC) mode or a point-to-point (P2P) mode. The MC communication is the foundation of the spiking neural network simulation on SpiNNaker and plays important role in our task graph framework for carrying a small distributed message. In the case of the P2P communication, a higher communication protocol such as SpiNNaker Datagram Protocol (SDP) can be constructed by encapsulating several P2P packets.

In our work, the MC communication is used inside the chip for emulating hardware multithreading useful for task duplication and parallelism, and also outside the chip for process migration purposes. The SDP communication is primarily used as a message passing mechanism between nodes in a task graph, and for sending/receiving data from/to host-PC. Currently, SDP packets can only be communicated between a SpiNNaker board and the host PC using a 100 Mbit/s Ethernet connection.

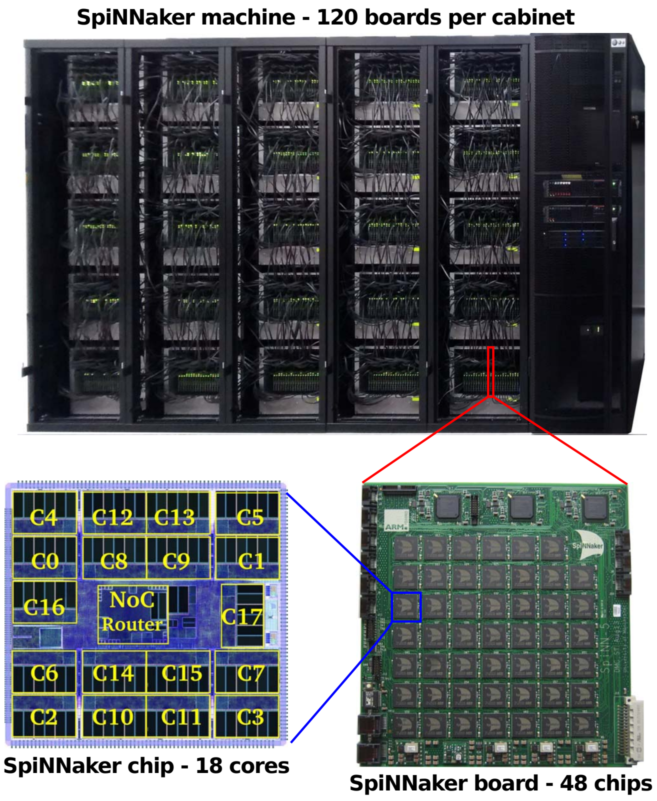
SpiNNaker machine - 120 boards per cabinet
C4
C12
C13
C5
C0
C8
C9
C1
C16
NoC Router
C17
C6
C14
C15
C7
C2
C10
C11
C3
SpiNNaker chip - 18 cores
SpiNNaker board - 48 chips

Figure 1. A partial SpiNNaker machine. The machine will consist of 10 cabinets with 120 SpiNNaker boards each. Each SpiNNaker board has 48 chips, and each chip has 18 ARM968 cores.

3.2. Task Mapping on a SpiNNaker Machine

As described in the previous subsection, the SpiNNaker system was originally designed for simulating massive spiking neural networks; thus, the multicast communication protocol plays very important role in the SpiNNaker framework to mimic the similar mechanism of information broadcasting using 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 our mapping strategy for task graphs on SpiNNaker using SDP protocol, even though it runs slower due to its pre-processing overhead.

Regarding node 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 the 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.

With one-to-one mapping, the communication overhead can be minimized. Here, 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 interchanging MC packets between chips. However, small MC packets are still circulated outside the chip to accommodate process migration in the fault tolerance scenario. In addition to the use of MC packets, SDP packets are used to the implement message passing protocol for task graphs.

Our 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 many-core systems such as SpiNNaker, fault tolerance is a compulsory aspect that needs to be handled properly to maintain service reliability of the system. In our work, 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 application.

Our framework uses the task graph description given by the XL-Stage program (see [6]). Here, a task has an output triggering condition that depends on its input dependency. For example, the node P6 in a 9-node task graph shown in Fig.2 has two outputs that depend on two inputs. The zoomed-in representation shows the triggering condition for each output. For example, output to node P7 depends on P0: after receiving 1 packet from P0 the node P6 will generate 7 packets to node P7. Whereas output to node P8 depends on both P0 and P2: 3 packets will be sent to P8 after P6 receives 4 packets from P0 and 4 packets from P2.

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 our work, we took 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.

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 [8]. The zoomed-in representation of node P6 of the 9-node task graph illustrate this amelioration strategy. For example in node P6, the task that targets P7 has been duplicated on a set of cores $\{C1,C3,C4,C5,C6,C7,C8,C9\}$. 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 $\{C2,C10,C11,C12,C13,C14,C15,C16\}$ works in parallel targeting node P8. In this example, we use 8 cores for parallel processing in node P6, which leaves the remaining core C17 in an idle state. This idle core will then serve as the backup core to support fault tolerance property of our task graph framework.

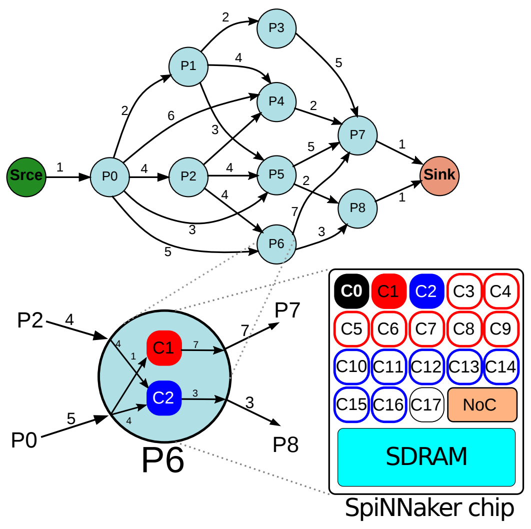

Figure 2. An example 9-node task graph for our experiment. The zoomed-in representation of node P6 illustrates task splitting and spawning mechanisms inside a SpiNNaker, as well as simple amelioration strategy by task duplication.

3.3. Proposed Optimization Strategy

As described in Section 3.2, the node-to-chip mapping is used to achieve the highest efficiency in the one-to-one mapping scenario. For mapping the entire task graph on SpiNNaker we opt to use a heuristic method. In this paper, we use the results of the EA proposed in [5], and propose an algorithm to further improve the overall performance of the mapped task graph. Our algorithm, which is called scattering-and-grouping, is basically a load-balancing strategy that can be applied on any existing map.

The load-balancing can be achieved by migrating the busiest nodes from a dense region into a lower-traffic region. This technique is applied as follows: we re-arrange the node list such that a small number of "heavy" nodes are selected and redistributed accross the network, and the remaining nodes are sorted and grouped according to their adjacency to those heavy nodes. This scattering-and-grouping method is inspired by the CP/MISF (critical path/most immediate successor first) algorithm, which is an improvement of the classical static scheduling algorithm called Highest Level First with Estimated Times (HLFET) [9], [10]. Such an algorithm utilizes the concept of a bounded number of processors (BNP) for an homogeneous environment (such as the SpiNNaker system).

The scattering-and-grouping method takes into account the sum of computation cost of nodes in the list labeled as TG-Node-List (produced previously by the evolutionary algorithm), and then assigns priorities to the nodes with the highest volume first. Afterwards, the immediate successors of those nodes are selected to surround the parent node. Our method is presented in algorithm(1).

Algorithm 1 Scattering and grouping algorithm.

```
Require: TG-Node-List
1: Build Node's Traffic Level (NTL) table from TG-Node-List
2: Sort NTL descendingly
3: Create $n$-groups with one highest NTL node for each group. This highest NTL level is referred to as group's top level node (TLN)
4: Assign the heaviest node in each group as the top level node (TLN). Remove TLN from the NTL table
5: Send to and distribute image in SpiNNaker
6: for $i = 0$ to $N - 1$ do
7:     select up to 6 another nodes with closes adjacency to TLN(i)
8:     Remove the node from the NTL table
9: end for
```

4. Evaluation and Discussion

4.1. Experimental Set-up

In this paper, we evaluate the performance of our proposed improvement described in Section 3.3 using a 25-node graph shown in Fig.3, the same network used in [5].

Three example maps have been generated by the EA (shown in Fig.3 in [5]), and we added two additional mappings for the evaluation. Table 1 describes five mappings that are evaluated in this paper. For the experiment, we used two scenarios. The first scenario used a quad-link mesh topology, in which the number of links of each SpiNNaker chip was reduced to four; the diagonal links (labelled as NorthEast-link and SouthWest-link) were disabled. This is to closely mimic the condition assumed in [5]. The second scenario used a 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 compared to the conventional rectangular 2D-mesh network.

TABLE 1. MAPPING SCENARIOS FOR THE GRAPH SHOWN IN FIG.3.

Map	Description
MAP-1	Naïve implementation: task graph nodes are mapped to chips in sequential order without any spacing in between.
MAP-2	Correspond to Fig.3(a) in [5]: good fault tolerance with poor performance.
MAP-3	Correspond to Fig.3(b) in [5]: good performance and poor fault tolerance.
MAP-4	Correspond to Fig.3(c) in [5]: balance between performance and fault tolerance.
MAP-5	Proposed improvement based on scattering-and-grouping algorithm presented in Section 3.3.

4.2. Performance Evaluation

As described in Section 4.1, we implemented several mapping strategies for the 25-node graph example. Three of the mappings were produced previously (see [5]), and two additional mappings are presented for comparison: one

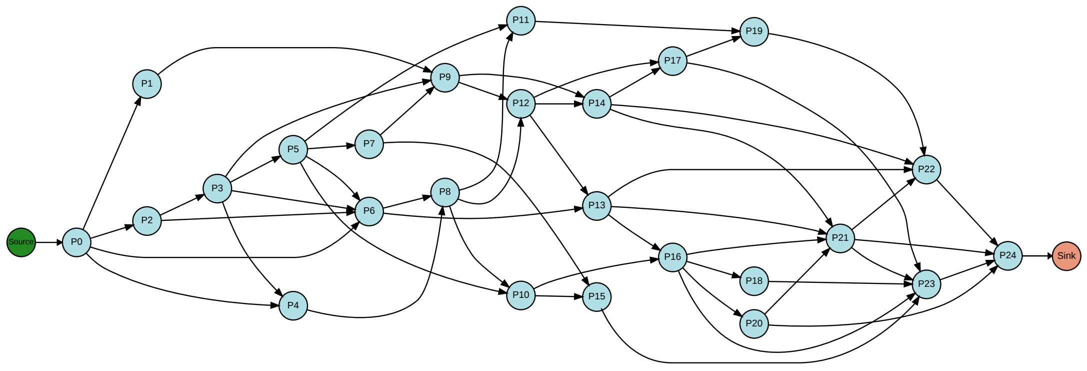

Figure 3. An example task graph with 25 nodes.

representing the worst-case mapping, and the other representing our proposed improvement using the scattering-and-grouping algorithm.

For the evaluation, we inspected each node in the graph and observed the packet processing level on that node during the execution of the task graph. Fig.4 shows the number of processed packets in each chip for the 25-node network shown in Fig.3. From the figure, we can observe the following behavior. The worst-case mapping (MAP-1) has a very high number of processing packets in its early-stage nodes, indicating that it is very active in the beginning, but becoming less active towards the output/final node. Map-4, which represents the "balance" mapping, shows many similar activities with Map-2 in the early-stage nodes, and gradually shift the similarity towards Map-3 in the later-stage nodes. Map-5, which represents our proposed improvement, shows a similar trend to Map-4. However, Map-5 has a flatter distribution compared to Map-4 as indicated by its variation level shown in Table 2. Map-4, which was notified as the balance between Map-2 and Map-3, has the variation level lower than Map-2 but higher than Map-3.

The overall throughput of those five mappings is presented in Fig.6. As we can see from the figure, our proposed algorithm improves the overall performance while preserving its fault tolerance coverage. We argue that this improvement is related to the load balancing action of the program. We observed that those mappings have a different packet processing level per node. Such a discrepancy (shown in Table 2) indicates that load balancing has a significant impact on the overall performance: lower variation (i.e., better load balancing) will produce higher overall throughput.

TABLE 2. THE AMOUNT OF VARIATION IN PACKET PROCESSING FOR EACH NODE SHOWN IN FIG.4.

	Map-1	**Map-2**	**map-3**	**map-4**	**map-5**
Std	90082	76909	69477	70469	69449

We inspected further the action of load balancing of our proposed mapping by converting the chip processing levels into chip activity levels. The result is shown in Fig.5. Particularly in Fig.5e, we can see that our scattering-and-grouping algorithm separates busy nodes and then distributes

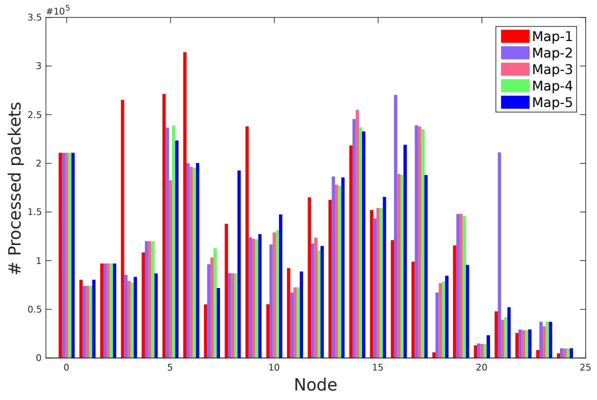

Figure 4. Measured number of processed packets in each node.

them more evenly across the mesh.

By re-distributing those busy nodes, higher reliability was achieved. This can also be inferred by observing the number of dropped packets reported by the SpiNNaker kernel. Table 3 and Table 4 present the reported dropped packets in the modified SpiNNaker mesh topology (quad-links topology) and in the original SpiNNaker mesh topology (hexa-links topology). As we can see from those tables, our proposed improvement for the given maps can reduce the dropped packets between 2.7% to 6.1%. We also observed that the number of dropped packets in the hexa-links topology tends to be higher than that in the quad-links topology. We argue that this is a normal consequence of having a higher number of communication channels. However, the higher number of communication channels also increases the chips throughput, which, in turn, will also increase the overall throughput of the application graph. This is presented in Fig.6, which shows an 8.2% improvement over the previously generated mapping using EA.

We propose to reduce the number of dropped P2P packets even further by re-injecting dropped packets back into 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 networks, such as our factor graph framework, can take advantage of this to improve the performance.

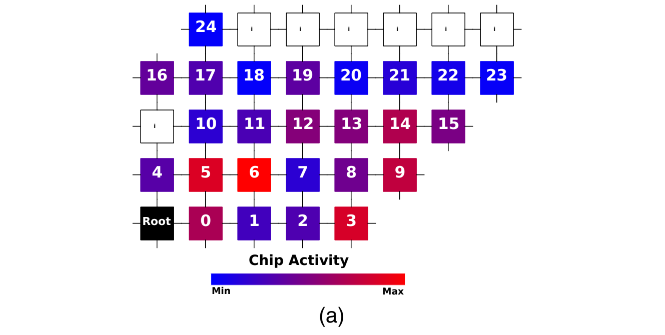
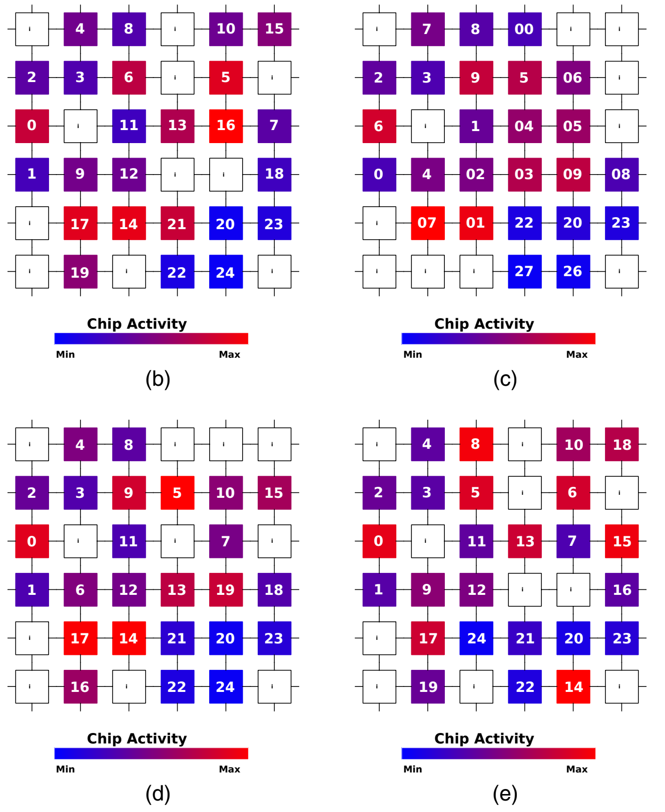

Figure 5. Chip activity level useful for observing the load balancing. (a) to (e) correspond to the activities on Map-1 to Map-5 respectively. The chips activity level were produced by normalizing the number of processed packets in each node (as shown in Fig.4) to the minimum and maximum values in the corresponding map.

TABLE 3. Dropped P2P packets in the rectangular mesh.

	Map-1	Map-2	map-3	map-4	map-5
#Packet	1976	700	790	596	580

5. Conclusion and Future Work

This paper presents the evaluation of task graph mapping on a many-core system. We also propose a scattering-and-grouping algorithm for improving its overall performance. The algorithm tries to reduce traffic congestion by relocating high-traffic nodes out of a dense region. Our experiment with the scattering-and-grouping algorithm shows that the previously generated mapping can be improved further by utilizing soft load balancing on the network. The algorithm simply re-distributes busy nodes whilst maintaining direct connectivity of those nodes with their children/parent nodes. In the experiment, we first modified the existing routing topology of the SpiNNaker machine to mimic the conventional 2D rectangular mesh, and then applied some mapping scenarios. We then reverted to the default SpiNNaker original 2D triangular mesh topology, and applied the same mapping again. From both scenarios, we evaluated the behavior of the maps and observed that our proposed algorithm can improve the performance at about 2.8% to 8.2% from its original version. This demonstrates that our proposed algorithm works well with the platform and can improve the efficiency of a task graph mapping. This result can lead into a larger investigation which deals with power and traffic management system for online optimization. We regard this as our future work.

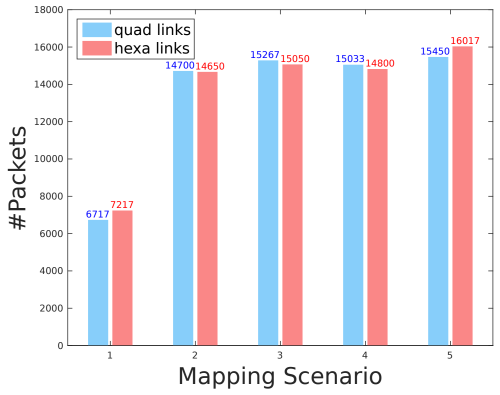

Figure 6. Overall throughput on two different mesh topologies. By redistributing busy nodes to create less-stressed network traffic, our proposed algorithm produced higher packet transaction among nodes as shown on the rightmost bars.

TABLE 4. Dropped P2P packets in the triangular mesh.

	Map-1	Map-2	map-3	map-4	map-5
#Packet	2622	760	812	658	618

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] T. A. Bartic, D. Desmet, J. Y. Mignolet, J. Miller, and F. Robert, "Mapping concurrent applications on network-on-chip platforms," in *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.*, Nov 2005, pp. 154–159.

[2] S. G. Pestana, E. Rijpkema, A. Radulescu, K. Goossens, and O. P. Gangwal, "Cost-performance trade-offs in networks on chip: a simulation-based approach," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, Feb 2004, pp. 764–769.

[3] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, pp. 60–76, 2013.

[4] N. Chatterjee, S. Paul, P. Mukherjee, and S. Chattopadhyay, "Deadline and energy aware dynamic task mapping and scheduling for network-on-chip based multi-core platform," *Journal of Systems Architecture*, vol. 74, pp. 61–77, 2017.

[5] 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.

[6] 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.

[7] M. Kumar, M. Serrano, J. Moreira, P. Pattnaik, W. P. Horn, J. Jann, and G. Tanase, "Efficient implementation of scatter-gather operations for large scale graph analytics," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7.

[8] 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.

[9] Y.-K. Kwok and I. A, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, May 1996.

[10] 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.