
SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Fisica

# Interconnection networks simulations for computing systems dedicated to scientific applications at the exascale

Dissertazione di Laurea Magistrale

*Relatore:*
Dott. Piero Vicini

*Candidato:*
Flavio Pisani
Matricola:1344386

Anno Accademico 2015/2016

# Contents

# Acronyms

**BNF** Burton Normal Form

**CNF** Chaos Normal Form

**DPSNN** Distributed Polychronous Spiking Neural Network

**FIFO** First In First Out

**FLOPS** Floating Point Operations per Second

**FPGA** Field Programmable Gate Array

**FSM** Finite State Machine

**HDL** Hardware Description Language

**HPC** High Performance Computing

**J-sim** JavaSim

**LC** Link Controller

**LIF with SFA** Leaky Integrate and Fire with Spike-Frequency Adaptation

**LQCD** Lattice Quantum Chromo-Dynamics

**MPI** Message Passing Interface

**NIC** Network Interface Card

**ns-3** network simulator 3

**NVM** Non-Volatile Memory

**OMNeT++** Objective Modular Network Testbed in C++

**QoS** Quality of Service

**RDMA** Remote Direct Memory Access

**SAF** Store and Forward

**VCT** Virtual-cut-through

# Introduction

The computational power available to scientific applications is constantly growing, giving access to unexplored regions of computational physics. The next frontier for High Performance Computing (HPC) is the exascale, i. e. machines capable of $10^{18}$ Floating Point Operations per Second (FLOPS). Taking into account the actual compute power available from a single compute unit, the number of nodes needed to reach the exascale is around $10^6$. The interconnection network is a critical part of the system and has to minimize the time required to send information over the network, while providing enough bandwidth and scalability.

The design and the optimization of an interconnection networks is a complex task because the behaviour of the full system is highly dependent on the topology, the routing algorithm and the traffic. In this perspective large scale simulations of the network architecture under design are mandatory to achieve optimal performances. The main work of this thesis is to implement a low level network simulator based on the APEnet/APElink network protocol, a proprietary interconnection system developed by the APELab of INFN and targeting HPC platforms optimized for scientific computing. An effective network simulator can be implemented using Hardware Description Language (HDL) or high level programming languages (like C++); running a simulation using an HDL is more accurate but requires more computing power and since we are interested in exascale-sized systems this approach is not feasible. Furthermore the simulator must be accurate and flexible enough to allow for fast and effective modifications of network topology, routing algorithm and injected traffic. The simulator developed in this thesis is based on the OMNeT++ C++ framework which provides enough flexibility and power to meet all the requirements. Different routing algorithms and different network typologies has been explored, evaluated and characterized using both synthetic and real application traffic; in particular a neural network simulator traffic generator has been implemented, in order to evaluate the achievable performance of the network using traffic extracted by a great challenge brain simulation application: the INFN Distributed Polychronous Spiking Neural Network (DPSNN).

# Chapter 1

# Evolution and new frontiers of computational Physics

In this chapter we will analyse the progress made by computing hardware in the last decades and its impact on computational physics. Problems that were extremely difficult to solve in the 80's now can be easily computed on consumer desktops thanks to the progress made by computer science and microelectronics. To solve computationally hard problems of the past in less time and using smaller systems is useful, but the most challenging part is to solve problems that were impossible or impractical to solve in the past using cutting-edge technology today.

## 1.1 Moore's law and the growth of computational power

In 1965 Gordon Moore published a paper on the Electronics Magazine containing a prediction:

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years". [1]

This prediction was pretty accurate as we can see from Figure 1.1. An higher transistor count means more computational power available but how much more? Physicists are not interested in counting transistors, they want to solve interesting problems using the computing platform so we need to define a more appropriate metric for computational power. Quantifying the computational power of a given system is a complex task because the performances of the platform may vary significantly from a workload to another, resulting in misleading numbers. The de-facto standard used by the supercomputing community is to measure the Floating Point Operations per Second (FLOPS) that a system can perform using a linear algebra benchmark called LINPACK [2].A list of the 500 most powerful computing systems available in the world has been published twice a year since 1993 by the top500 organization [3].

The computational power available is growing exponentially giving more power and more troubles to computational scientists. Usually the code used in scientific simulation is written using a sequential paradigm and all the instructions are exe-

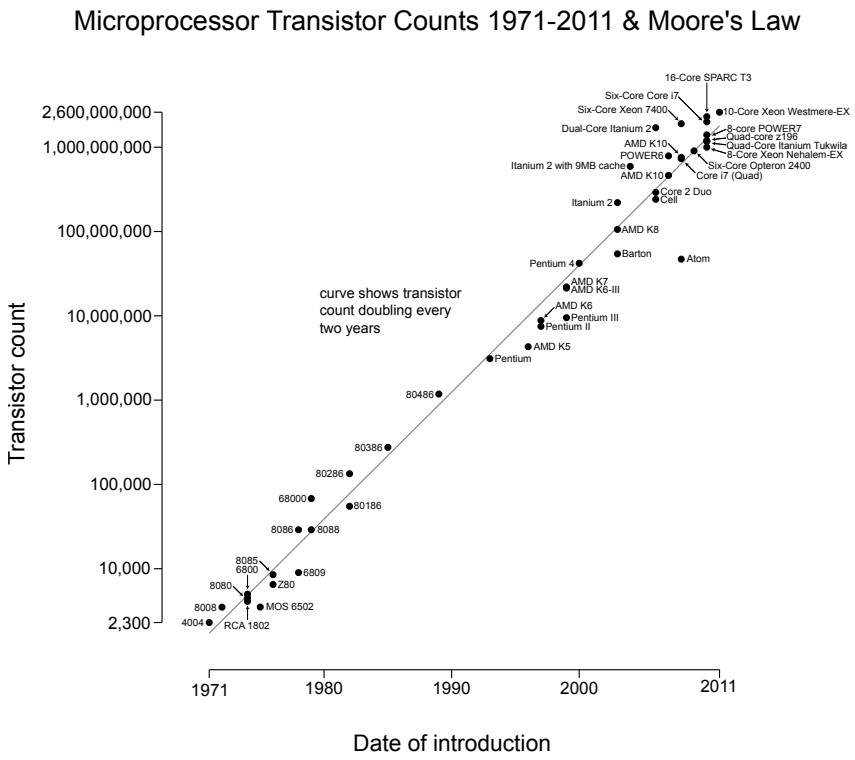



**Figure 1.1.** Transistor count of microprocessors over the years. Original image from wikipedia .

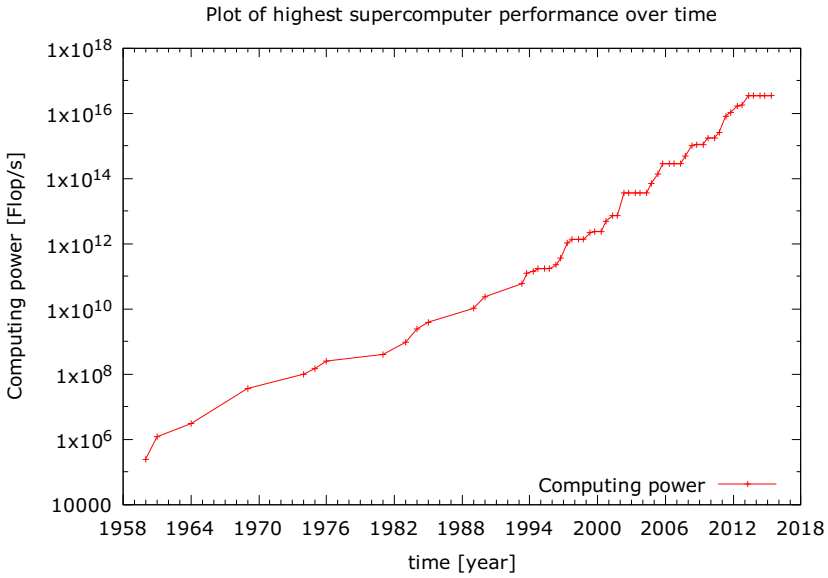



**Figure 1.2.** Supercomputing maximum performance over the years measured in FLOPS, data after 1993 are from the top500 list.

cuted by a single processing core one after the other; due to technological limitations every supercomputing system uses parallelism to grow in performances: multicore nodes are interconnected by an high-speed low-latency network infrastructure; as an example, the most powerful machine available in the world[1] (Sunway TaihuLight) has 10649600 cores and provides 93014.6 TFLOPS of computing power. To get the full computing power of the system the simulation must run in parallel onto the multiple cores available. The task of writing efficient parallel code is very challenging and it will be discussed in more detail in next section.

## 1.2 Software Complexity in parallel computing

In order to take advantage of a multi-core computing platform the programmer has to split the code into multiple program flows that should be able to run at the same time. The complexity of this task may vary from "very low" to "nearly impossible" depending on the actual algorithm. In the next sections we will discuss the main important topics in parallel computing.

### 1.2.1 Amdahl's and Gustafson's law

Before trying to parallelize an algorithm it is useful to estimate the maximum theoretical speedup[2] achievable by parallelizing a given algorithm. This maximum performance boost can be calculated using the following equation, known as the Amdahl's law proposed by Gene Amdahl [4]

$$S = \frac{1}{1 - p + \frac{p}{s}} \tag{1.1}$$

where $S$ is the theoretical speedup, $p$ is the portion of the code that can be parallelized and $s$ is the speedup achieved by the parallelization. The maximum time reduction from running a program in parallel over $n$ processes is $n$. As we can see from the (1.1) $s < \frac{1}{1-p}$ therefore the fraction of the code that is not parallel $1 - p$ puts an upper bound to the maximum performance gain achievable, disregarding the amount of processing cores available. Therefore before trying to spend computing resources increasing $s$ it is crucial to maximize the parallel portion of the code and then use an appropriate number of parallel processes. As we can see from Figure 1.3, if the portion of parallel code is not big enough the diminishing return in the speedup makes using more processes ineffective.

Amdahl's law assumes that the size of the problem is fixed while the available resources increases, a more realistic approach to the problem is given the Gustafson's law [5] which assumes that the size of the problem can change to achieve a better fit on the platform available.

$$S = 1 - p + sp \tag{1.2}$$

If we what to use more effectively the computing power available we can increase the size of the simulation. This approach is not always valuable, if the problem has

[1] According to the top500 list updated in June 2016

[2] The speedup is defined as the time reduction factor between a reference implementation and an optimized implementation of a given algorithm. For example a speedup of 2 means that the optimized version takes half of the time to perform the same task.

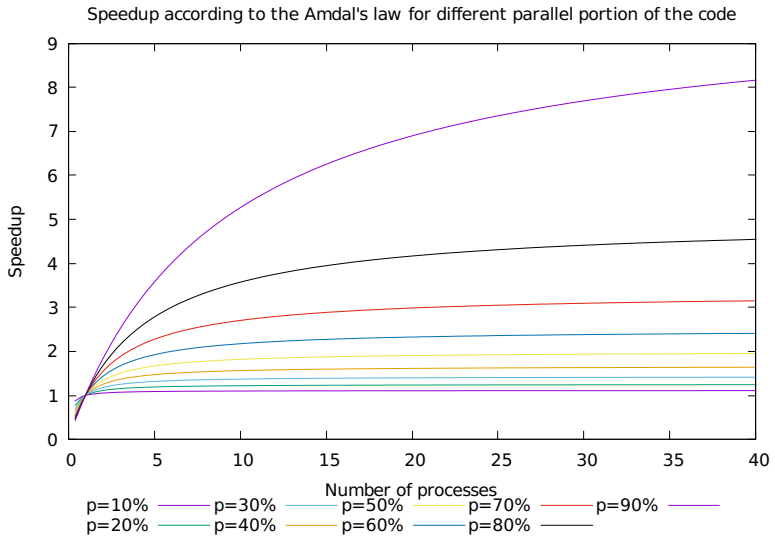



**Figure 1.3.** This figure depicts the speedup versus number of processes achievable according to the Amdal's law for different parallel portion of the code.

a fixed size or there are temporal constraints on the execution of the simulation we cannot use Gustafson's approach.

### 1.2.2 Dependencies

One of the main difficulties in parallelizing algorithms are dependencies, if there is a chain of sequential instructions depending one on the other the full program cannot execute in less time than the chain itself. Usually a program flow is composed by different chains of sequential instructions, therefore to parallelize the code we need to indentify chains that can be executed in parallel. To fully understand the problem we need to formalize the concept of dependency, to do so we can use the Bernstein's conditions [6]: we consider two separate program sections $P_i$ and $P_j$, there are no dependencies between the two if the input variables $I$ and the output ones $O$ meet the following conditions:

$$I_i \cap O_j = \oslash \tag{1.3}$$
$$I_j \cap O_i = \oslash \tag{1.4}$$
$$O_i \cap O_j = \oslash \tag{1.5}$$

Conditions (1.3) and (1.4) express a flow dependency: one program section needs the output of the other one as an input before its execution. Condition (1.5) express an output data dependency, the execution order of the two program sections affects the output value of the code. The following example shows dependant and independent portions of code:

```
1  c = a * b;
2  d = a + b;
3  e = d / c;
```

The first and the second instructions have no dependencies therefore can be executed in parallel, on the other hand the third one needs the output of the first two issuing a flow dependency, therefore it cannot be executed in parallel with the other two.

### 1.2.3 Race conditions

Multiple program flows running in parallel are usually called threads and they may share variables and data. Since the execution time of the same thread is not constant over time, and it can be very different, shared variables are not accessed in a fixed order. Every time a thread needs to read a variable containing the output of another thread we need to use a synchronization mechanism or the entire program can produce inconsistent results.

"Race" conditions are very frequent in scientific simulations and the programmer must detect and avoid them. For example if we consider the temporal evolution of a set of interacting particles we can run the simulation of every particle in parallel but we need to update the interaction potential; to update the potential we need to know the position of all the particles at a given time, which implies synchronizing the different threads. Synchronization can be achieved by using "locks" and "barriers" which halt threads to maintain the required timing. Stalling threads generates performance degradation, therefore it is extremely important to reduce race conditions to fully exploit the potential of the parallel computing platform available.

## 1.3 A case study: the DPSNN

Among the many open challenges, understanding brain behaviour through simulations is one of the most interesting ones from a scientific and technological point of view: learning how our brain works can lead to important discoveries from a medical perspective; the study of the human brain can be a source of requirements and architectural inspiration for future parallel/distributed computing systems, and a parallel/distributed coding challenge. The main focus of several neural network simulation projects is the search for:

- biological correctness;
- flexibility in biological modelling;
- scalability using commodity technology.

Among all the implementations available the DPSNN [7, 8] has been used as a case study for the network simulations of this thesis. Developed by P.S Paolucci and E. Pastorelli from the INFN APE Lab, the DPSNN is a mixed time and event-driven spiking neural network simulator implementing synaptic spike-timing dependent plasticity, designed to be natively distributed and parallel [9].

### 1.3.1 Modelling a neural network

A neural network is made of neurons connected together, the cells receives inputs from other neurons through contacts on the dendritic tree called synapses, the inputs

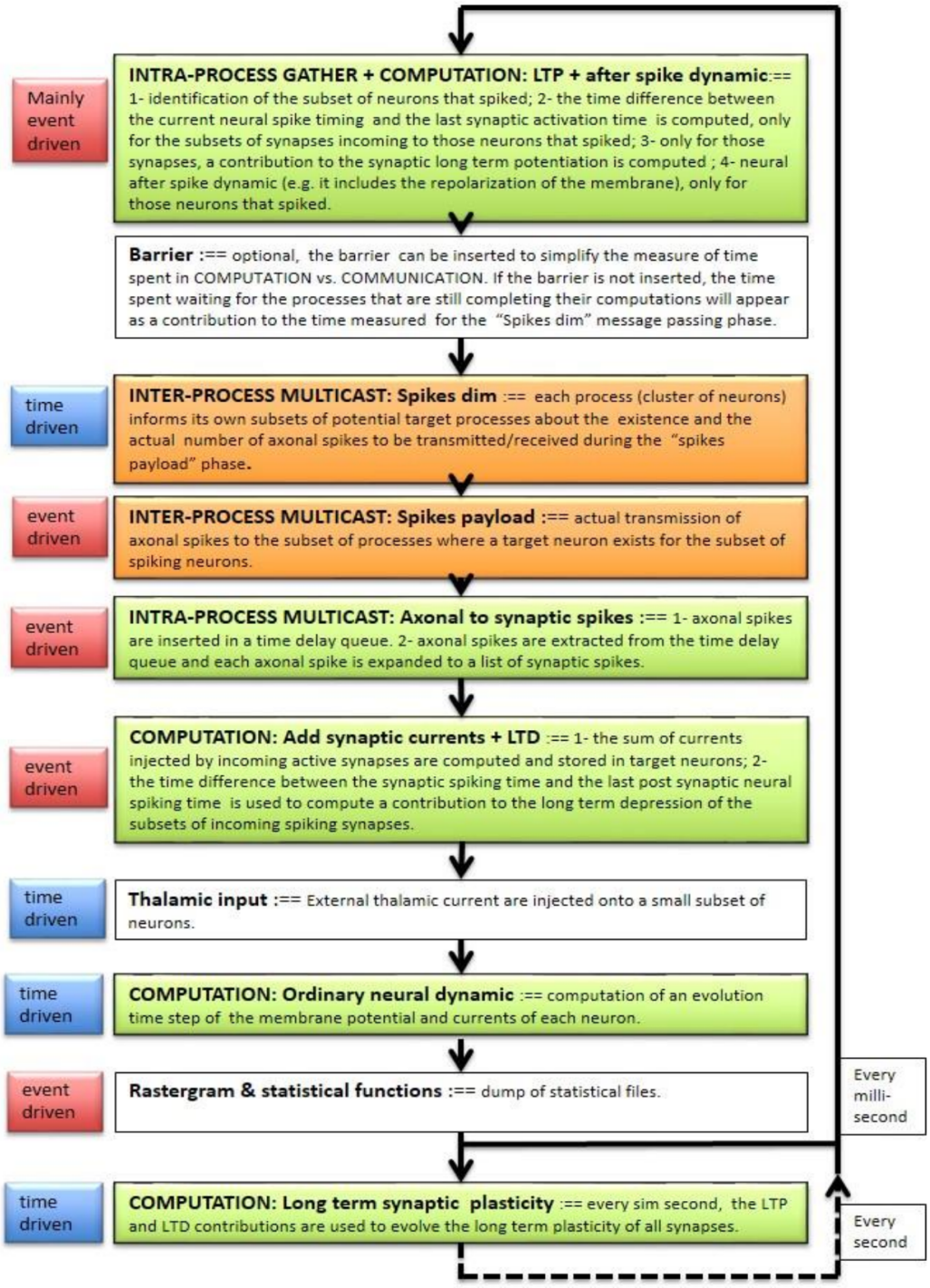



**Figure 1.4.** Flow of the DPSNN simulation. Each process iterates over the block represented in this picture that simulate: the dynamics of neurons, the spiking and plasticity of the synapses and the exchange of messages through axo-dendritic arborization.

generates electrical currents that change the membrane potential of the neuron. The modelling of the neuron defines when the electric potential changes of the neuron's membrane induce the neuron to fire a *spike*. When a neuron fires it sends an electrical signal trough the axon to the synapses of all the neurons connected to it.

From the coding point of view, this generic structure can be mapped to a network of C++ processes communicating through a message passing interface (the default communication interface is Message Passing Interface (MPI)) and the application is designed to be easily interfaced with custom communication libraries.

The full neural network is divided into clusters of neurons and their set of incoming synapses. The data structure that describes the synapse includes the information about the total transmission delay introduced by the axonal arborization that reaches it. The list of local synapses is further divided in sets according to the value of the axo-synaptic delay. Every C++ process describes and simulates a cluster of neurons and incoming synapses. The processes exchange sets of *axonal spikes* which contains the identity of the neuron that spiked and the emission time of each spike. Axonal spikes are only sent to processes having at least a target synapse for the axon.

The simulation can be divided into two phases: in the first one the neural network is created and all the synaptic connections are generated among the neurons; the second phase is the actual simulation of the dynamic of neurons and synapses. The dynamic is calculated using a combination of time-driven and event-driven approaches for synapses and neurons:

- Event-driven simulation for synaptic dynamics.
- Time-driven simulation for neural dynamics.

Figure 1.4 depicts simulation flow showing the separation between remote and local operations, and event-driven and time-driven sections.

### 1.3.2 Spiking neuron model

The neuron model used for the DPSNN simulations is the the Leaky Integrate and Fire with Spike-Frequency Adaptation (LIF with SFA) [10, 11]. The equations regulating the dynamics of the membrane potential $V_m$ and the membrane recovery variable $w$ are the following:

$$V_m < V_{th} \begin{cases} \dot{V}_m &= -\frac{V_m - E_L}{\mathrm{T}} - \frac{g_w w}{C_m} + \frac{I_e}{C_m} \\ \dot{w} &= -\frac{w}{\tau_w} \end{cases} \tag{1.6}$$

$$V_m \geq V_{th} \begin{cases} V_m &= V_{reset} \\ w &= w + A_C \end{cases} \tag{1.7}$$

where:

- $V_{th}$ is the threshold voltage, when $V_m$ reaches $V_{th}$ the neuron spikes;
- $I_e$ is the total synaptic current;

- $C_m$ is the membrane capacitance;
- $V_{reset}$ is the reset potential value, after the spiking event the membrane potential is reset to this value. To generate a non trivial dynamic $V_{reset} < V_{th}$;
- $T$ and $\tau$ are time two time constants;
- The others are constant that defines the kind of neuron and other parameters of the model.

### 1.3.3 Connectivity model

Neurons are arranged in *cortical columns*, each one is composed by $\sim 1000$ neurons; the columns are then organized into a 2D grid and the spacing between the columns is $\alpha \sim 100$ $\mu$m, the columns are then mapped onto the processes optimizing calculation and communication time. Every core on a modern CPU can easily handle more than one columns reducing the communication between processes.

The local connectivity, i.e. the synapses generated by source neurons belonging to the same column of the target neuron, has been set to 80%. The lateral (remote) intra-areal connectivity, i.e. the synapses generated by neurons belonging to different columns placed at distance $r$, is generated using the following exponential law:

$$Ae^{-\frac{r}{\lambda}} \tag{1.8}$$

with $A = 0.03$ and $\lambda = 290$ $\mu$m being respectively the peak connection probability and the exponential decay constant. A cut-off has been set to the synapses generation, limiting the projection to the subset of columns with connection probability greater than 1/1000. This generates a column connection stencil of 21x21 centred onto the source column.

In addition to the synaptic connections described above the simulation implements external synapses firing at a fixed firing rate, the number of external synapses is fixed to 540 per neuron.

# Chapter 2

# Interconnection networks

A modern HPC system is made of computing nodes interconnected via an interconnection network. The interconnection network is responsible for transferring all the data and all the synchronization information needed by the different program flows running on the system. The network must therefore reduce the communication delay in order to minimize the stall time of the different program flows and optimize efficiency.

Because of their crucial role in HPC systems, interconnection network have been extensively studied over the years producing a very complex environment full of classifications, definitions and obscure terminology. This chapter is intended to be a very brief look at interconnection networks, especially regarding the project of this thesis. A comprehensive explanation of interconnection networks can be found in [12].

## 2.1 Networks classification

A classification of interconnection networks is shown in Figure 2.1; this scheme is not fully exhaustive but it is more than adequate for our purposes. According to this schema we can categorize networks in four main categories: *shared-medium* networks, *direct* network, *indirect* network and *hybrid* networks.

Shared-medium networks use a shared communication medium to connect all the devices. Due to its shared nature the network experiences a severe performance degradation when the number of nodes increases. They usually provide good multicast/broadcast[1] performances and are used in small systems like multi-CPU nodes.

Direct networks uses a point-to-point node-to-node interconnection. Every node is a compute unit with its own processor, memory and peripherals. Each node has a router block which handles the communication with a subset of the nodes called neighbours and all the nodes are connected forming a network topology. To establish communication between non-neighbours nodes intermediates steps are used according to the routing function. Direct networks offer good scalability and are used in many HPC systems.

Indirect networks are made of nodes interconnected through switches. Every

[1] A multicast is a special send of a message from a node to many nodes, a broadcast is a particular case of multicast in which a message is sent to all the nodes

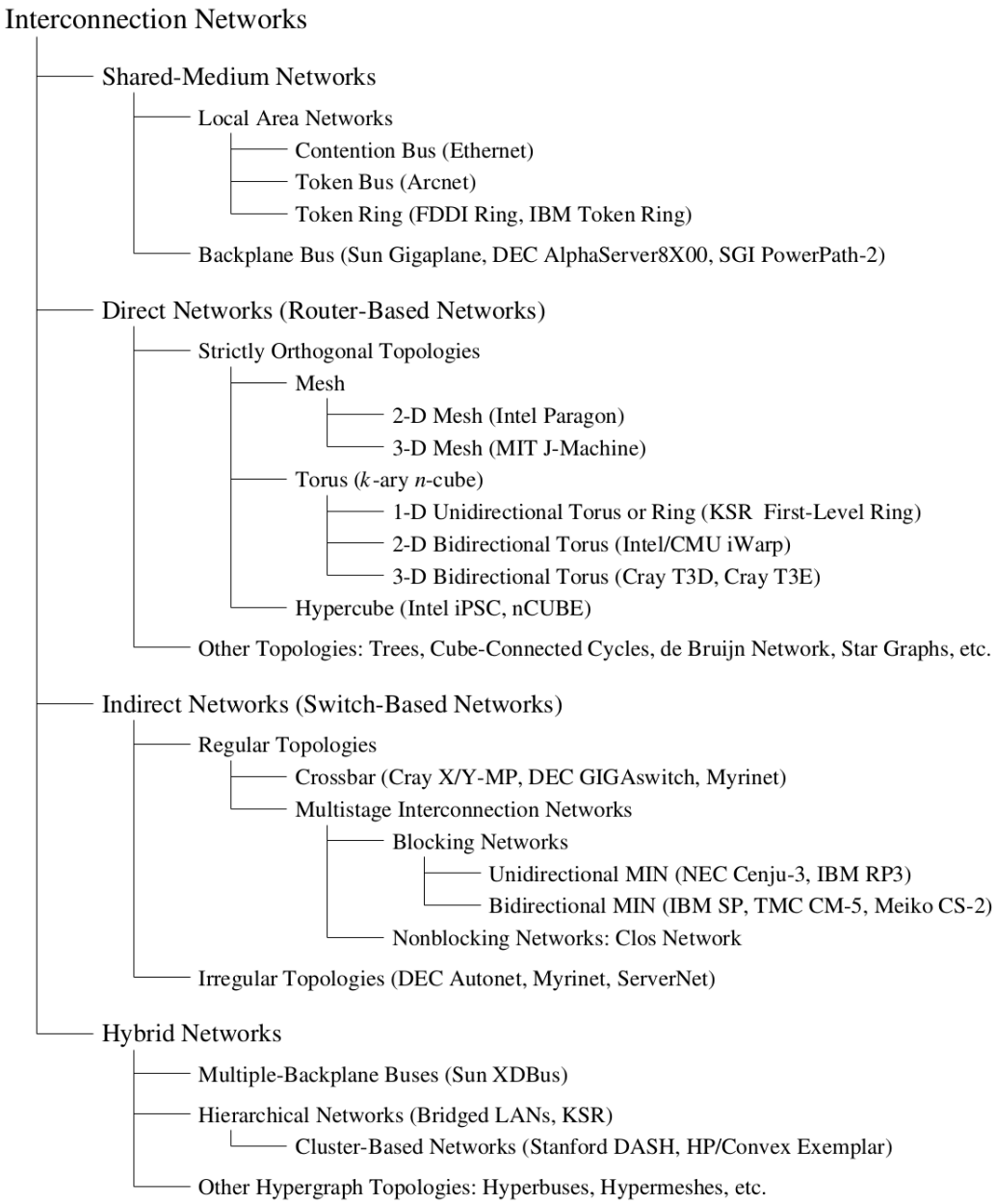



**Figure 2.1.** Classification of interconnection networks.

node is a compute unit but it has no routing capability, the only network functionality of the node is carried out by the network adapter which connects the node to a switch. Every switch has a fixed number of ports and every port can be connected to: a node, another switch or not connected. The connection of the switches generates the network topology and the routing algorithm selects the path between the nodes. Having the compute node outside of the switch increases the distance between two nodes by two producing higher network latency[2]. Indirect networks are commonly used in data centres because they are easier to maintain and do not require specialized network capabilities from the compute nodes.

Hybrid networks combine the use of shared-medium, direct and indirect network to achieve better performances. To mitigate the performance degradation of shared-medium networks a hierarchical structure can be used, generating islands of shared-medium network interconnected using direct or indirect networks. This approach is gaining acceptance into the HPC community.

## 2.2 Network topologies

In both direct and indirect networks the topology can be modelled by a directed graph $G(N, C)$, where the edges of the graph represent the $C$ unidirectional communication channels[3] and the vertices the $N$ switches or nodes for indirect and direct networks respectively. Using this simple model we can define some basic network properties from the graph representation:

- *Node degree/Switch radix:* Number of channels connecting a specific node/switch to its neighbours.
- *Distance:* The distance $d(a, b)$ between node $a$ and node $b$ of the network is defined as the minimum number of hops required for going from $a$ to $b$. Because the graph is directed the distance may not be commutative.
- *Diameter:* The maximum distance between two nodes in the network.
- *Regularity:* A network is *regular* when all the nodes have the same degree.
- *Symmetry:* A network is *symmetric* when it look alike from every node.

We will now discuss several network topologies and characterise their properties with a special focus on scalability.

### 2.2.1 N-Dimensional torus/mesh

In N-dimensional *tori* every vertex of the graph is connected to $2N$ neighbours as depicted in Figure 2.2, therefore the nodes can be imagined as distributed on an N-dimensional grid[4]. The nodes at the boundaries of the grid are connected across the border of the network providing: periodic boundary conditions, *symmetry* and

[2] Network latency is defined as the time to wait before receiving the data over the network.

[3] Bidirectional channels can be represented as a couple of unidirectional ones.

[4] The network topology specifies only the connection between the nodes and does not specify their position

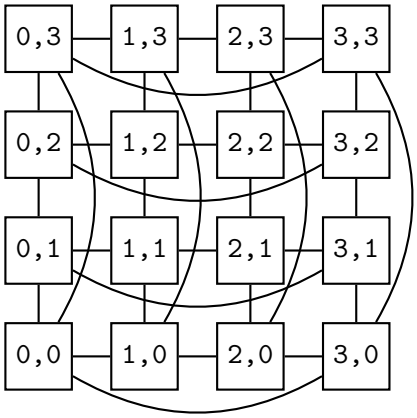



**Figure 2.2.** Example of a 4x4 2D torus.

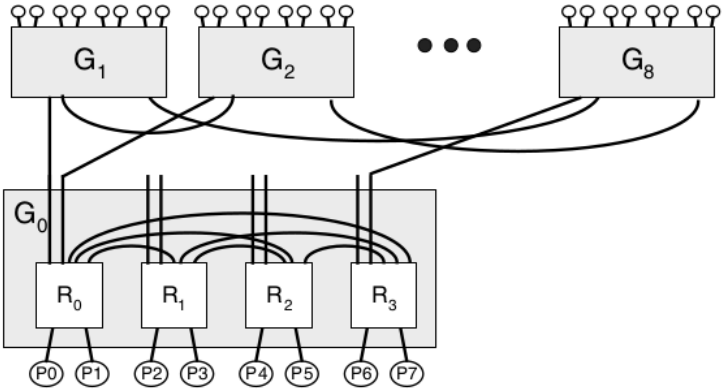



**Figure 2.3.** Fully connected topology example for a dragonfly network.

*regularity*. The diameter of the network depends on the shape of the grid and the dimension of the torus, for an N-dimensional hypercube grid of $n$ nodes the diameter is $N\frac{\sqrt[N]{n}}{2}$. From a scalability point of view this topology has a fixed node degree and a variable diameter, increasing the dimension of the network provides a better scalability at the cost of more connectivity. Thanks to its fixed degree this topology is widely used in HPC's direct networks since adding more nodes does not change the structure of the node itself.

*Meshes* are similar to tori but they have open boundary conditions, so the network is *asymmetric* and *irregular*. The degree of the nodes at the boundaries is $2N - 1$ and the radius is $N\sqrt[N]{n}$.

### 2.2.2 Dragonfly

This topology aggregates routers in an efficient way to make them behave as higher radix ones [13]. The hierarchical structure of the network is composed by three levels: router, group and system. The intra-group and inter-group interconnection network topologies can be selected to achieve the requirements of the system.

In a dragonfly network we have:

- $p$ Terminals connected to each router using *end channels*[5]
- $N$ Network terminals
- $a$ Routers in each group
- $h$ Channels within each router used to connect to other groups
- $g$ Groups in the system

Every router is connected to $p$ terminals, $a-1$ local channels and $h$ global channels, therefore the radix of every router is $k = p+a+h-1$. Because every group consists of $a$ routers connected via intra-group channels it can be considered as a virtual router with an effective radix $k' = a(p+h)$. The parameters $a$, $p$ and $h$ can have any values, however to get the optimal load balancing of the network traffic the parameters should respect the following relation:

$$a = 2p = 2h \tag{2.1}$$

This ratio is derived from the fact that a packet uses two local channels every time it uses an global channel or an end channel. The basic topology is depicted in Figure 2.3 and uses a fully connected topology for both the inter-group and intra-group networks: if we make this particular choice the network is *regular* and *symmetric*; the *radius* of the network is 3 for a direct configuration and 5 for an indirect one and the radix of the nodes is $k = \sqrt[4]{(N-1)4^3} - 1$, using the relation (2.1) for the parameters. Different topologies can be used to reduce the radix of the nodes at the cost of an higher network *radius*.

## 2.3 Router model

Before going any further in this brief introduction to interconnection networks it is fundamental to define and describe a router model. A comprehensive and clear model is the one proposed in [12] which is a good representation of the real hardware architecture of a router. The internal architecture shown in Figure 2.4 is divided into the following major components:

- **Buffers:** *First In First Out (FIFO)* buffers are used for storing messages in transit. The model in Figure 2.4 has buffers for input and output channels but alternative designs may have input or output buffers only.
- **Switch:** This component connects input buffers to output buffers.
- **Routing and arbitration unit:** Those components implement the routing algorithm, selecting the appropriate output channel for an incoming message and setting the switch accordingly. If the same output channel is requested by multiple messages at the same time the arbitration unit must resolve the contention. The arbitration policy can be a simple round robin or a complex priority algorithm.

[5] This connection can be done internally for a direct network or externally using a port of the switch for an indirect network.

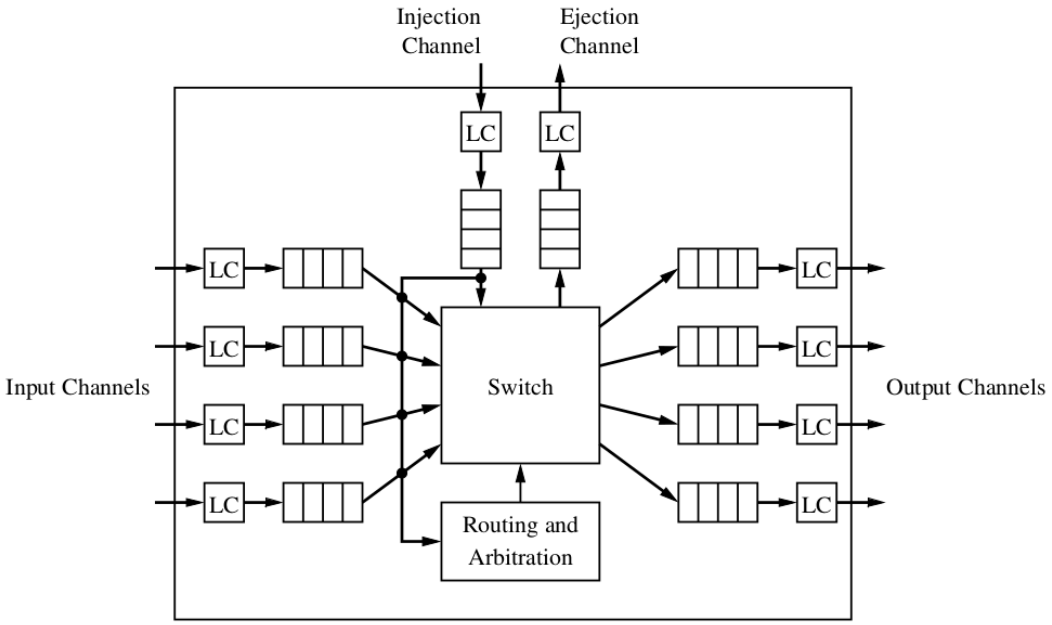



**Figure 2.4.** Internal structure of a generic router model (LC = link controller). Original image from [12]

- **Link Controller (LC):** It implements the flow of the messages across the channel between two routers.
- **Processor interface:** This component is a channel interface to the local processor instead of an adjacent router. It consists of one or more injection and ejection channels.

This router model is designed for direct networks but can be used as a switch model for indirect networks if we remove the processor interface.

## 2.4 Packet terminology

In order to transfer data over the network, they must be prepared and they may be split in chunks. This process is called packetization and it is done in the following way: the *message* to be transferred over the network is split in to chunks of a fixed size and all of the chunks are then encapsulated into *packets*. Packets are made of two parts a data one called *payload* and a protocol part called *header*.

The *packet* is the smallest unit of information that can be sent over the network, therefore the *header* must contain all the addressing data needed to deliver the payload to its destination; this information is crucial for the delivery, so the *header* is stored into the first part of the packet. Other useful protocol informations can be stored into the optional *footer* which is stored into the last part of the packet.

The *packet* is then divided into smaller sub-units called *flits*, those units are the ones whose transfer is requested by the sender and acknowledge by the receiver. *Flits* may be divided into *phits* which are the units of information that can be

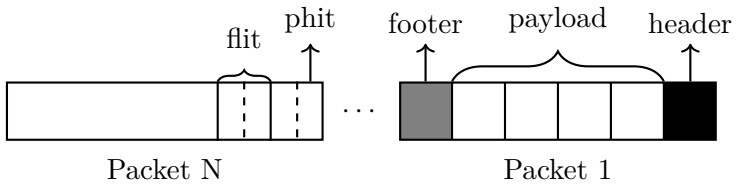



**Figure 2.5.** Fragmentation of a message into packets, flits and phits.

physically transferred in a single cycle between two nodes. All the different sub-units of information are depicted in Figure 2.5.

## 2.5 Switching techniques

A crucial aspect of a network infrastructure is how the packets are switched and saved into intermediate nodes during their path. In both direct and indirect networks some switching action is required but we have not yet defined a policy for switching the packets. In this section three of the main switching techniques are presented: *store and forward*, *wormhole* and *virtual cut through*. Whatever policy we decide to adopt, we need to manage a critical physical resource: the receiving/sending buffers. Every switching technique has to decide how to manage the free space in the buffers and when to start and stop forwarding packets. When there are packets waiting for resources, the network is in a congested state.

### 2.5.1 Store and Forward (SAF)

This switching technique is very simple and works as follows:

- The node receives al the flits of the packet and stores them into the receiving buffer of the channel.
- The header is parsed by the router and the output port is calculated.
- If the receive buffer of the next node has enough free space to store the full packet and the channel is free the packet is forwarded, otherwise it waits.

This approach let the router deal only with complete packets. The main drawback of this switching technique is latency: at every hop the switch has to wait for the full transfer of the packet before forwarding the information. In absence of congestion the latency $L$ of a packet is given by the (2.2) where: $n_{flits}$ is the number of flits of the packet, $n_{hops}$ is the number of hops between source and destination and $t_{flit}$ is the time needed to transfer a single flit[6].

$$L = t_{flit} \cdot n_{flits} \cdot n_{hops} \tag{2.2}$$

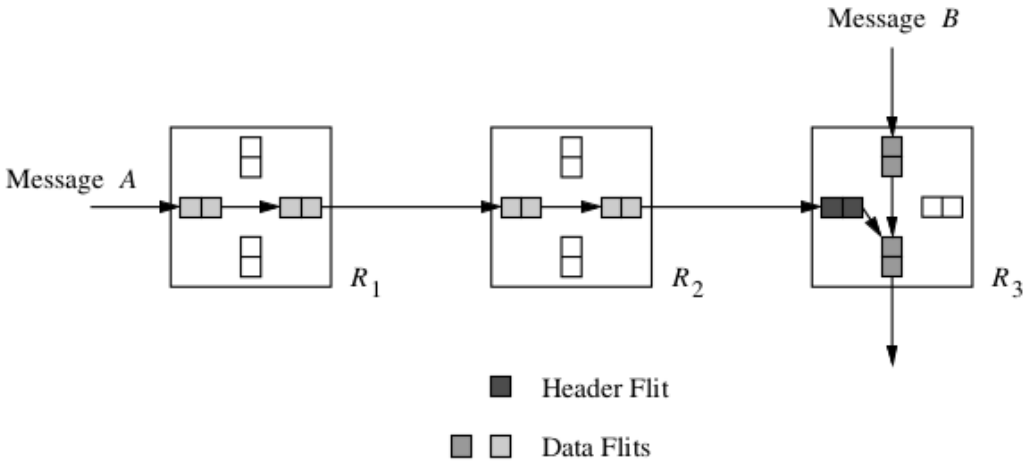



**Figure 2.6.** An example of messages travelling through a wormhole network. Message A is blocked by message B generating network congestion.

### 2.5.2 Wormhole

This technique is the opposite of the SAF one. The switch forwards the packet a flit at the time as depicted in Figure 2.6 following this schema:

- The first flits of the packet containing the header are received and stored into the receiving buffer.
- The header is parsed by the router and the output port is calculated.
- If the receive buffer of the next node has enough free space to store a single flit and the channel is free the packet is forwarded, otherwise it waits.
- Every new flit is forwarded as soon as available if the buffer and the channel are free.

In order to take the routing decision the router has to parse the entire header, this makes crucial keeping the header into the smallest possible number of flits, preferably one. The main advantage of this approach is the reduced latency by pipelining of the full packet transmission time across the hops. The latency $L$ in absence of congestion can be calculated as follows:

$$L = t_{flit} \cdot (n_{flit} + n_{hops}) \tag{2.3}$$

Wormhole routing requires a more complex flow control system which has to be able to stop and resume the transfer of the packet if congestion occurs, and it is more prone to congestion because a packet uses resource of multiple nodes at the same time.

[6] We are neglecting the time spent by the router to make the routing decision, this time is common to all the switching techniques and therefore not interesting in this comparison

### 2.5.3 Virtual-cut-through (VCT)

In this case we want to take the best from both wormhole and SAF and combine them into a single technique. A VCT network behaves like a wormhole one apart from the forwarding requirements: to forward a packet the receiving buffer must have enough free space to store the full packet. In this way we have the same flow control simplicity and congestion resources occupancy as in a SAF network but the same latency of a wormhole network in absence of congestion. The main drawback is an higher buffer capacity requirement than a wormhole network: in a wormhole network the buffers must provide at least enough free space to store a single flit; in VCT and SAF networks the buffers must provide at least enough free space for a full packet.

## 2.6 Routing algorithms

A *routing algorithm* defines which route a packet should take while travelling from its source to its destination. The selection of the algorithm can heavily affect network performances, therefore we have select it carefully. In this section we will discuss the minimum requirements for a routing function and we will discuss different approaches to the problem of routing a packet.

It is useful to give some basic definitions. A *routing function* calculates the next step that a packet has to take in its path towards its destination: in principle we could use informations gathered across the whole network to calculate the path but, if we want to achieve good scalability of the system, this approach is not practical. In this work we will consider routing functions that use only local information coming from the packet or from the node.

A routing function is *connected* if any node of the network can reach any node, and because we want all the packets to be deliverable on the network, every routing function must be *connected*. The routing function can select always the same path between two nodes or can choose between multiple ones according to the network traffic, the algorithms in the first case are called *deterministic*, the ones in the second case are called *adaptive*. An algorithm is *fully adaptive* if has the possibility to use all the possible paths. A routing function is *minimal* if it delivers a packet from $a$ to $b$ in $d(a, b)$ hops.

The minimum requirement for a routing algorithm is to deliver packets in a finite amount of time regardless of the network traffic, an in-depth study of this condition will be done especially for what concerns *deadlock* and *livelock*. Before introducing the problem we need to introduce one useful instrument to solve it: virtual channels.

### 2.6.1 Virtual channels

In section 2.5 we learned that the critical resources while sending packets over a network are the receiving buffers. A common strategy consists of optimizing the channel usage and reducing congestion using virtual channels. Virtual channels consists of a set duplicated buffers multiplexed and demultiplexed into the same physical channel by the link control logic. From the switch point of view there

is no difference between a physical channel and a virtual one because the switch sees only the buffers, all the physical layer is handled by the link controller. By adding a virtual channel we can effectively increase the number of channel in the network without adding the extra cost and complexity of the real hardware link. All the virtual channels shares the bandwidth of the physical link with a certain policy, for example round robin. If we select a non uniform policy it is possible to implement Quality of Service (QoS) into the network using several virtual channels with different priority levels. In this section virtual channels will be only used to avoid deadlock.

### 2.6.2 Deadlock

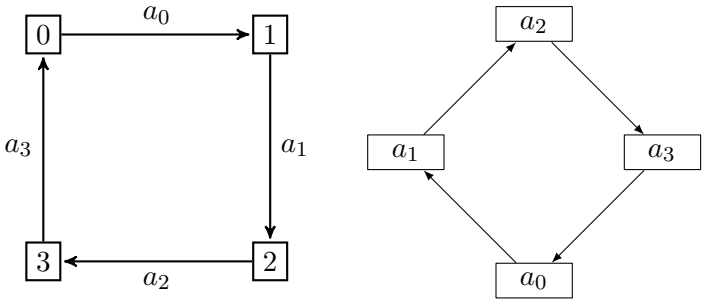
Deadlock is a *configuration* of the network in which there are packets that cannot be delivered. A *configuration* is an assignment of a set of packets or flits to the corresponding buffer. A deadlocked configuration occurs when some packets are blocked for an infinite amount of time because they are waiting for resources occupied by other congested packets. This condition must be avoided since it breaks the functionality of the network itself.

A deadlocked configuration is called *canonical* if all of the packets present into the network are blocked. We will study only canonical configurations because any other configuration as a corresponding canonical one. To obtain a canonical deadlock configuration it is sufficient to stop injecting new traffic into the network and wait for the delivery of all the non blocked packets.

The presence or the absence of potentially deadlocking configurations is complex to determine and it is a property of the routing algorithm in use. The same topology with the same number of virtual channels may be deadlock-free or not depending on the selected routing algorithm. On the other hand this property is independent of the network traffic and must be verified without making any assumption on the traffic pattern.

To avoid deadlock we will focus on two opposite approaches: the detection of and the reaction to a deadlock configuration, and the impossibility of deadlock configurations to occur. The first method requires the network to be aware of being in a deadlocked state and to react breaking the deadlock condition. If we want to certainly detect a deadlock configuration we need to use non local information which is usually difficult to collect and analyse in large networks. If we want to use only local information we can guess if a packet is deadlocked by using a *time-out* system. This approach can lead to misidentification of congested packets into deadlocked ones if the network is heavily congested, leading to performance degradation. The reaction to a detected deadlock requires the capability of retransmitting packets that need to free resources. The network must detect and react to the deadlocked configurations faster than they occur, otherwise different deadlocked configurations can pileup generating network malfunctions. The second method consists of providing a routing algorithm that cannot generate deadlocked configurations. Because this property must be traffic-independent it is not trivial to proof that a routing algorithm is deadlock-free, but this problem can be easily solved using some graph theory and an interesting theorem.

We introduce now the concept of *channel dependency* and the *channel depen-*




**(a)** Network topology **(b)** Channel dependency graph

**Figure 2.7.** A four node circular network with its channel dependency graph.

*dency graph*: there is a dependency between channels $i$ and $j$, if the routing algorithm can forward a packet holding resources on channel $i$ to channel $j$; the channel dependency graph is a directed graph which vertexes are all the unidirectional channels in the network and edges represents the dependencies between channels. The following theorem [14] gives a necessary and sufficient condition for a routing function to be deadlock-free.

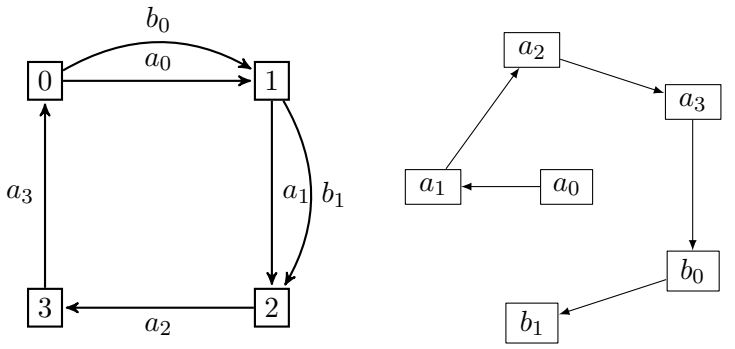
**Theorem 1.** *A connected routing function $R$ for an interconnection network $I$ is deadlock-free if and only if there exists a routing subfunction $R_1$ that is connected and has no cycles in its channel dependency graph.*

If the routing algorithm is deterministic the only connected subfuction is $R$ itself. Therefore for non adaptive routing the full channel dependency graph must be acyclic [15].

This theorem can be used to generate deadlock-free routing algorithms as in the following example: we consider a circular network of four nodes connected by unidirectional channels, as the one depicted in Figure 2.7a. Because of the simplicity of this network the only connected routing function is the one that forwards the packet through the only available channel if the destination is different from the current node. The dependency graph for this algorithm is shown in Figure 2.7b and it is not acyclic, therefore the network is not deadlock-free. In this trivial example it is very easy to find a deadlocked configuration: any configuration with all the buffers from all the nodes filled up with packets and zero packets arrived at destination is deadlocked.

If we want to make this network deadlock-free we need to add two virtual channels, as shown in Figure 2.8a and to use the following routing algorithm:

```
1  if ( destination == current )
2    packet_delivery;
3  else if ( source < current )
4    reserve(a_i+1);
5  else
6    reserve(b_i+1);
```




**(a)** Network topology **(b)** Channel dependency graph

**Figure 2.8.** Modified deadlock-free circular network and channel dependency graph of the proposed routing algorithm.

This new routing function uses dedicated virtual channel for packets with a source index smaller that the current node, this modification changes the channel dependency graph into the one depicted in Figure 2.8b which is acyclic and therefore the network is deadlock-free.

### 2.6.3 Livelock

Livelock is a misbehaviour of the routing algorithm which forward packets in a closed loop that does not contain their destination, resulting in not delivered packets and wasted network bandwidth. Any minimal routing function is automatically livelock-free, the distance $d$ between any couple of nodes into the network is a finite quantity, therefore every minimal path algorithm delivers any packets in $d$ steps. For non minimal algorithm extra attention to avoid livelock must be paid, especially if the algorithm uses only local information to calculate the path.

A simple technique to avoid livelock is to limit the number of non minimal hops that a packet can take, this limitation reduces the flexibility of the algorithm but avoids infinite looping.

## 2.7 Selected routing algorithms

In this section several routing algorithms for tori and dragonfly will be presented. For each algorithm we provide proof of the absence of livelock and deadlock.

### 2.7.1 e-cube [16]

This is a minimal non adaptive routing algorithm for N-dimensional tori and meshes. Because of the simplicity of this algorithm it is often used as a base for more complex ones.

The position of the nodes in the network is indicated by N integers giving the Cartesian coordinates of the node into the grid. The algorithm starts comparing the node coordinates against the packet destination coordinates from dimension 0 up to

dimension N. If the coordinates are different the packet is forwarded following the minimal path to the destination, if the coordinates are equal no action is required; if all the coordinates are equal the packet has arrived to its destination. The critical point of the algorithm is to manipulate the N dimensions in a fixed order[7]. Figure 2.9 depicts an example of path selected by the e-cube algorithm on 2D torus.

The algorithm uses minimal paths so it is livelock-free by definition. Before considering the deadlock freedom of the algorithm it is useful to divide the channels into classes:

- A channel belongs to the class $d^+$ if connects two node in the dimension $d$ in the increasing coordinates direction.
- A channel belongs to the class $d^-$ if connects two node in the dimension $d$ in the decreasing coordinates direction.

To proof the absence of deadlock configurations, we can use the theorem 1 and discuss the channel dependency graph. Thanks to the dimension order any channel of class $i^{\pm}$ cannot depend on any channel of the class $j^{\pm}$ where $j > i$, this removes all the possible loops between channels belonging to different dimensions. The algorithm selects the minimum path to the destination, for a given dimension $d$ the shortest path can be either in the direction $+$ or in the direction $-$ but it cannot be in both directions. Therefore it is impossible to generate dependencies between channels $d^+$ and $d^-$ in the same dimension. The only possible source of cyclic dependency left is within channel of the same class. If we consider a mesh network, packets are not allowed to cross the boundaries of the network, preventing deadlock. The torus topology requires more attention because the algorithm is not deadlock-free in the form presented before. A single channel class of a torus is exactly the same topology as the circular network depicted in 2.7a, therefore adding one extra channel dedicated to the packets that have crossed the border is enough to remove any deadlocking situation. The extra virtual channels are only required for half of the nodes in the class. In the real implementation it is often good practice to keep the nodes symmetric and add the extra virtual channel for all the nodes in the class.

### 2.7.2 star-channel [17]

This is a minimal path fully adaptive routing algorithm for N-dimensional tori and meshes based on the e-cube. This algorithm adds an extra virtual channel in addition to the ones used by the e-cube to achieve full adaptivity. The idea behind this routing function is to use the e-cube as an "exhaust valve" for any possible deadlock configuration originated by the adaptive behaviour.

The network is divided into two sub-networks: one is the e-cube subnet made of all the channels required by the dimension-order algorithm; the other network is the the "*" one which has one virtual channel on every link[8]. The routing algorithm uses the two networks according to the following rules:

[7] The order relation can be any total one and it can be selected only once. To change the order relation the network has to be completely empty.

[8] Every algorithm can have more virtual channels to implement QoS or other traffic control features, here we are discussing the minimum requirements for deadlock and livelock free routing.

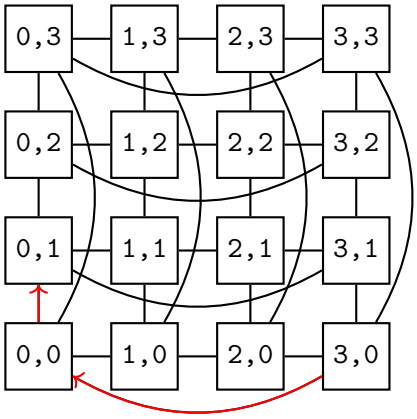



**Figure 2.9.** Path of the e-cube algorithm on a 2D torus between nodes (3,0) and (0,1).

- A packet can use a free channel of the "*" subnet to follow a minimal path to its destination in any dimension.
- A packet can use a channel of the e-cube channel only if it respects the dimension order criterion.

Livelock can be excluded because the algorithm uses minimal paths. To exclude deadlock we can use the channel dependency graph. If the e-cube subnetwork can provide a routing sub-function with an acyclic graph the full algorithm is deadlock-free according to the theorem 1. Packets can only use the e-cube network if they met all the dimensional requirements imposed by the algorithm. Because the e-cube routing function has an acyclic graph, the extra dependency added by the presence of the "*" subnet cannot generate deadlock configurations.

In a less rigorous way at every node the router can select a channel from a deadlock-free algorithm, because the algorithm is deadlock-free that channel cannot be busy for an infinite amount of time.

### 2.7.3 Smart dimension-order

This algorithm is a partly adaptive non minimal routing algorithm for N-dimensional tori and meshes based on the e-cube. While introducing the dimension order algorithm we mentioned the importance of keeping the nodes symmetric for technical reasons, therefore a real implementation of the e-cube on an N-dimensional torus leaves $2N$ unused virtual channels for every node. This algorithm takes advantage of the extra unused channel to provide adaptivity to the network.

At the first first step of the algorithm the router can misroute the packet if the selected e-cube channel is busy, the packet is forwarded in a different dimension than the one selected by the dimension order. To avoid deadlock configurations in the network the routing function uses the the unused virtual channels of the e-cube. We can partition an N-dimensional torus into $2N$ hypercubes by splitting every dimension into two halves, Figure 2.10 depicts the decomposition and shows the unused channels for a 2D torus. Every node has $N$ channel available for the misroute packets.

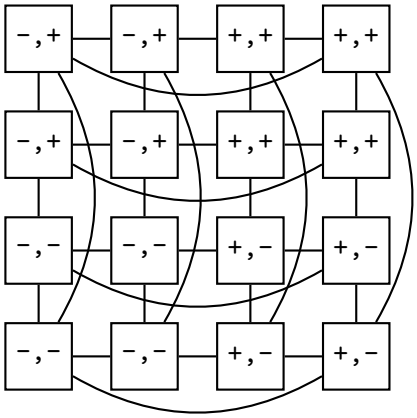



**Figure 2.10.** Decomposition of a 2D torus in 4 regions, every node has 2 free virtual channels in the direction and dimension identified by the label.

Livelock can be excluded because the algorithm uses a finite number of non-minimal steps. In this case the router can only do one non minimal hop before falling back to the minimal algorithm of the dimension order. Deadlock configurations are excluded because the algorithm performs only one adaptive step in the first node using a dedicated virtual channel. If we consider the channel dependency graph this does not changes the acyclic property of the e-cube routing sub-function.

### 2.7.4 Min-routing [13]

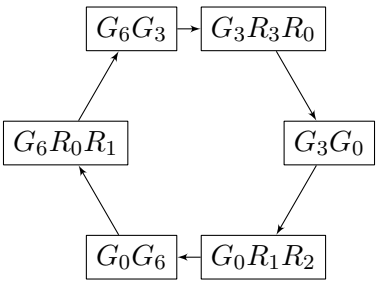
This is a minimal non adaptive routing algorithm for dragonflies and it can be considered as a base algorithm for adaptive or more complex ones.

Packets in a dragonfly network can be identified by three number:

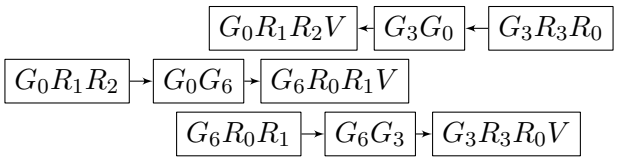
- **Group index ($G$):** it indicates the group
- **Router index ($R$):** it indicates the router within the group
- **Node index ($N$):** it indicates the node within the router

The min-routing algorithm uses a three step process to forward the packet using the corresponding local or global channel, for better clarity the algorithm can be described using this pseudo code:

```
1  if ( G_node != G_destination )
2    if ( is_connected(G_node, G_destination) )
3      forward_g(G_destination);
4    else
5      forward_l(global_to_local(G_destination));
6  else if ( R_node != R_destination )
7    forward_l(R_destination);
8  else
9    forward_i(N_destination);
```




**(a)** Standard network




**(b)** Network with extra virtual channels

**Figure 2.11.** Subsets of the total dependency graph for a fully connected dragonfly network. Graph 2.11a depicts a standard configuration with no additional virtual channels. Graph 2.11b depicts a deadlock-free configuration with extra virtual channels (identified by a V letter) for local routing of packets coming from other groups.

The function *is_connected* returns true, if the node has a direct global connection to the given group, false otherwise. The various *forward* functions forward the packet using global, local or internal port accordingly. The function *global_to_local* returns the index of the local node which as a global direct connection to the provided group. This method provides a fully connected minimal routing function.

The algorithm is minimal and therefore livelock-free. If no virtual channels are used deadlock configuration can occur because there are cyclic dependencies. We can easily create a loop between ingoing and outgoing packets between different groups, as depicted in Figure 2.11a because the same local channels are used for all the packets. In order to break those loops we need to add one extra virtual channel to the local network dedicated to packets coming from a different group. If we consider now the channel dependency all the loops are broken as shown in Figure 2.11b. Note that the dependency graphs are partial to provide a better understanding of the critical path, the graph in Figure 2.11b is not connected but this is due to the partial nature of the graph itself and it does not imply a not connected routing function.

## 2.8 Real world examples

In this section we will introduce the ExaNeSt [18] European project and the APEnet network infrastructure, developed by the INFN, as examples of a real world HPC system and of a custom direct network interconnection. Introducing the characteristics of this two projects is important because they will be used as target for the simulations implemented.

**Table 2.1.** Multi-tiered network structure of the ExaNeSt system.

| | Hierarchy | Switching |
|---|---|---|
| Tier 4 | System | Optical/Ethernet |
| Tier 3 | Rack/Cabinet | Optical/Ethernet/APEnet |
| Tier 2 | Backplane Chassis | Ethernet/APEnet |
| Tier 1 | Blade/Mezzanine | APEnet |
| Tier 0 | Node | AXI/APEnet |

### 2.8.1 The ExaNeSt project

ExaNeSt is one of three European projects that support an exascale-class computing architecture for systems built upon 64-bit ARM processors. In ExaNeSt, we will design and implement: a physical rack prototype and its liquid-cooling subsystem providing ultra-dense compute packaging; a storage architecture with distributed (in-node) Non-Volatile Memory (NVM) devices; a unified, low-latency interconnect, designed to efficiently uphold desired QoS guarantees for a mix of storage with inter-processor flows; and efficient rack-level memory sharing, where each page is cacheable at only a single node. Our target is to test alternative storage and interconnect options on actual hardware, using real world HPC applications.

#### The network hierarchy

The ExaNeSt system is composed by different hierarchical levels that generates a multi-tiered scalable interconnection system, an overview of the full structure is in Table 2.1. From a network technology point of view we have different interconnection media working together in an hybrid network: starting from the lower tier we have a shared-medium network, the AXI bus, and the APEnet direct network; in the upper tears we can find direct network technology like Ethernet. The strategy behind this non-uniform network architecture is to achieve ultra low latency from shared-medium and direct network, and scalability and cost effectiveness of indirect network for the long distance communication. The underlying network topologies are object of intensive studies to achieve the maximum available performances. Among all the available ones dragonfly and N-dimensional tori are well considered their for latency and scalability properties. One of the goals of this thesis is to produce a simulation environment that allows to explore networking solutions for the APEnet part of the full network infrastructure.

### 2.8.2 The APEnet network

The APEnet project of INFN [19] delivered a point-to-point network controller for 3D-torus topology, the key feature of this networking system are high throughput and low latency. The original target for this network card was GPU-based HPC system optimized for Lattice Quantum Chromo-Dynamics (LQCD) named QUonG. In order to get the maximum performances available the APEnet system is equipped with full Zero-copy support for data transfers involving both GPU and system memory allowing for lower latency and lower resource utilization.

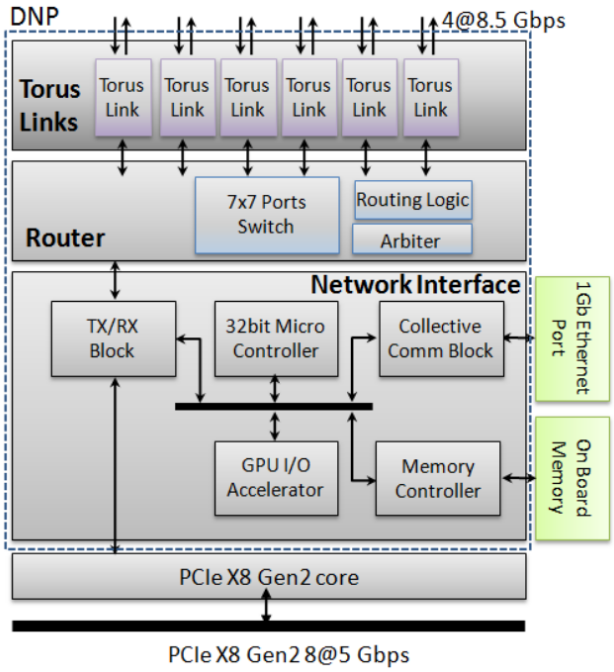



**Figure 2.12.** The APEnet DNP is the core of the architecture composed by the Torus Links, Router and Network Interface macroblocks.

The internal structure of the system is depicted in Figure 2.12 and it is composed of three parts:

- **Network Interface:** it handles all communication with the CPU and GPU through the PCIe bus and implements the Remote Direct Memory Access (RDMA) system.
- **Router:** it combines the 7x7 switch, the routing and arbiter logic. The switching technique used is VCT and the routing algorithm is the e-cube.
- **Torus Links:** this block is composed by the 6 bidirectional link required to form a 3D-torus. Every link is equipped with the required buffers and implements the proprietary protocol APElink. Any link has two RX FIFOs, one per virtual channel, and one TX FIFO.

The structure of the node is compatible with the example router model presented in § 2.3 and it adds the extra complexity needed to interface with a real system. From a networking simulation point of view the PCIe interface is not strictly relevant and therefore it will not be studied in this work, furthermore the ExaNeSt implementation of APEnet will be based onto the AXI bus instead of the PCIe one.

### The APElink: Data Link Layer

The Data Link Layer is a key part of the network implementation and its architecture has to be deeply to implement an accurate low level network simulator.

The APElink [20] protocol is based on 128bit-words[9] aggregated in packets. Every packet has an *header*, a *footer* and a *payload*, the maximum payload size is 4096 bytes.

The actual protocol itself is a word-stuffing one, meaning that the transmission of a *magic* word is required in order to stop the flow of data frames and distinguish them from control frames. To initiate a new data transmission two additional words are included into the data flow (*magic* and *start*). The efficiency of the protocol is defined as the ratio between the number of data word and the number of word sent, in this case we have at minimum 4 control words (header, footer, magic, star) for every data packet resulting in an efficiency for a payload of size $P_{max} = 256$ words of:

$$E_{max} = \frac{P_{max}}{4 + P_{max}} = \frac{256}{4 + 256} = 0.985 \tag{2.4}$$

Because of the VCT switching the protocol must include informations about the status of the receiving buffers to prevent overflow. This additional information is contained into control frames called *credits*. The sending interval of the credits is a crucial choice for network stability and performances: if the credits are sent too often the efficiency of the protocol drops because of the interleaved control frames; on the other hand if the sending interval for the credits is too large the router has to perform unneeded interruptions into the communication, if the informations about the remote buffer are not updated the router must assume that the words are still in the buffer resulting in fake congestion. In order to select the correct sending frequency we need to take into account the latency of the channel, the full latency for a credit exchange is:

$$L_T = 2(L_L + LR) \tag{2.5}$$

where $L_L$ is the latency induced by the synchronization between the receiving and transmitting side and $L_R$ is the time of flight of the packet. The actual values measured onto the real system are: $L_L = 20$ cycles, $L_R = 35$ cycles and $L_T = 110$ cycles. To avoid additional latency credits are sent every $C = 35$ cycles.

A low level network simulator implementing the APElink protocol has to provide VCT routing of packets composed by a one word header, a one word footer and a multiple word payload; to prevent buffer overflow a flow control based on credits messages has to be implemented.

[9] Those specifications are for the APEnet V4, the most recent version (V5) uses 256bit-words; the protocol itself does not depend on the size of the data path used and therefore we will use the V4 specification in this description

# Chapter 3

# Network simulation implementation

In this chapter we will discuss in detail the aspects of performing low level network simulations of proprietary network infrastructures, in particular we will implement a flexible and low level simulator for the APElink/APEnet protocol.

## 3.1 Prototypes and simulations

The design of an high performance interconnection network would be a demanding task on its own, but we have to factor another major problem: in the early stages of the design process the actual hardware is not available and even in the late stages it may not be available at the scale required by the problem. Prototypes are usually very expensive and they have to be used carefully in order not to waste important resources from the available budget.

The use of HDLs and programmable logic such as Field Programmable Gate Array (FPGA) reduces the complexity of designing prototypes and provides reusable components usable to test different configurations. The main drawback with approach is the time needed for the development, designing and debugging an HDL project requires a lot of resources. The use of programmable logic for prototyping does not help from the scalability point of view: it is feasible to build a small test setup to evaluate the performances on a very small scale; it is unfeasible to build a large scale test system to evaluate large scale performances.

In order to perform large scale tests the only practicable way is to *simulate the system*. If we have the HDL code available for the FPGA prototype a first idea could be using the simulation engine available for the selected hardware description language. The computing power required by HDL simulators is very high and their are not designed for large scale system testing. The use of HDL simulators is very useful but it is limited to validation of the actual hardware design on a very small scale. The next available option is to use a high level programming language, like C++, and implement a large scale simulation of the system. In order to perform a realistic evaluation of the network performances the simulation has to replicate accurately the low level behaviour of the network and performing a low level implementation of the link protocol and the switching technique. The simulator must be flexible and allow the designer to easily exchange network topology and routing algorithm to evaluate and select the best configuration for the system.

Simulations can be divided into two main classes, according to their evolution

policy, *continuous simulation* and *discrete event simulation*. In continuous simulations the state of the system is continuously tracked over time using a fixed temporal resolution. On the other hand a discrete event simulation uses *events* to keep track of the temporal evolution: the events are marked with a timestamp, which is used to update the simulation time, providing a variable temporal resolution.

A low level network simulation's timeline is usually non homogeneous and contains quickly evolving portions interleaved by static ones, thanks to its ability of skipping inactive temporal segments a discrete event approach is more effective.

## 3.2 Selection of the simulation tool

The problem of simulating a network infrastructure can be addressed in different ways according to: time and manpower available, nature of the network, data and results required, ecc; therefore it is crucial to set all the constraints and goals of the actual simulation that we are going to implement:

- Implementation of a modular and custom network architecture based on the APElink/APEnet protocol at flit level;
- Testing and comparison of different network topologies to evaluate performances and scalability;
- Testing and comparison of different routing algorithms to select the best trade off between performance and resources;
- Injection of arbitrary traffic into the network especially "traced" traffic from real scientific applications;
- Testing the "exascalability" of the network i.e. running exascale-size simulations;
- Characterize the network in terms of: latency, throughput and congestion.

With all the basic constraints fixed, we can now illustrate different possible approaches to the problem and select the one that fits better our particular case.

To implement a network simulation there are three main ways to proceed:

- **Off the shelf simulator:**
  - \+ **Pros:**
    - \+ No development is required
    - \+ Usually they are better optimized for a specific task
  - \- **Cons:**
    - \- Adding new features may be extremely difficult or impossible
    - \- Available topologies, routing algorithms, network architectures and protocols are fixed and limited
    - \- They do not provide full access to the customization of the network parameters

- **Simulation library:**
  - + **Pros**
    - + The development time can be entirely dedicated to the implementation of the network model
    - + They are designed to be expanded by the user
    - + The simulation core is usually well optimized
    - + Most of them provide out of the box parallel programming support
    - + The debugging and validation of the simulation core is done by the whole community of users resulting in a more reliable code
  - - **Cons**
    - - The learning curve of the framework may be steep, especially the library is not well documented
    - - They may provide unneeded features that may slow down the application
- **Implementation from scratch:**
  - + **Pros**
    - + Fully customizable; everything is under the complete control of the developer
    - + The code can be optimized and tailored for a specific implementation
  - - **Cons**
    - - A lot of effort goes into the development of the simulation kernel taking away resources from modelling the actual network
    - - Optimization and/or parallel programming support must be provided
    - - The code is not validated and all the bug fixing can be time consuming and lead to incorrect results

Analysing the requirements and the available solutions we can easily discard the off the shelf approach because: it does not allow the implementation of a custom network architecture, it does not allow full freedom in terms of network topology, routing algorithm and traffic injection. In principle an open source off the shelf simulator could be modified to fit the requirements of the project but this usually requires more effort than an implementation from scratch, the code may not be well documented nor designed with flexibility in mind.

On the other hand a full implementation from scratch offers full flexibility at the cost of a bigger effort. This consideration alone is not enough to discard this method but, if we take into account that the goal of this thesis is to implement an accurate model of the APEnet network and of the traffic produced by real scientific simulations, the implementation of a simulation engine is not interesting for this particular research. In addition to this the use of a mature simulation framework can lead to more trusted results due to the higher validation of the code itself. A from scratch implementation will be considered only if none of the available libraries provides enough flexibility and scalability to fulfil the requirements.

Using a simulation library/framework provide all the flexibility needed without taking away important resources from the actual topic of the project. Simulation libraries are designed with reconfigurability in mind therefore code changes can be implemented in a more efficient way. An important requirement for a good simulation framework is a good documentation; developing a simulation model using a badly documented framework can result in a bigger effort than a full custom implementation.

For this project we decided to use a simulation library for the reasons just provided. In the next section we are going to evaluate and to select a suitable framework among the available ones.

## 3.3 Selection and comparison of simulation frameworks

Over the years, many network simulation frameworks have been developed and many surveys has been conducted to help developers choosing the right framework for a specific task [21, 22]. In this thesis we will examine three of the most used, open source available ones: ns-3 [23], J-sim [24] and OMNeT++ [25].

### 3.3.1 ns-3

network simulator 3 (ns-3) is a discrete event network simulator targeted for educational and research use. The simulator is written in C++ and offers optional Python bindings. There is no dedicated scripting language used to describe the network topology. The tool provides pre-implemented standard topologies and custom ones must be implemented in C++ or Python using the provided API functions. It defines a model of working procedure of packet data networks and provides an engine for simulation. It is possible to use ns-3 to model non-internet-based models but the entire structure of the simulator is more oriented towards the internet and WiFi environment. ns-3 supports parallel execution using MPI allowing to run large scale simulation and to take advantage of parallel computing platforms.

### 3.3.2 J-sim

JavaSim (J-sim) is a component-based, compositional simulation environment. It is implemented using the autonomous component programming model which eases the task of modelling complex hierarchical hardware structures. As suggested by the name the simulator is written in Java and provides many ready to use internet-based models together with a GUI library. The latest release of the project is from 2006 so the development of the project seems to be completely blocked.

### 3.3.3 OMNeT++

Objective Modular Network Testbed in C++ (OMNeT++) [26, 27] is a C++-based discrete event simulation library developed for modelling communication networks, multiprocessors and distributed systems; thanks to an extensible and modular design it can be easily used to implement any discrete event simulation. OMNeT++ uses a hierarchical structure made by module connected through channels. Two

different kind of modules are used: *simple* modules and *compound* modules. Simple modules are the active components of the simulation, they are implemented in C++ as an object oriented specialization of a simple module base class. Compound modules are passive containers for any number of simple and compound modules. The connection between modules is defined using OMNeT++'s topology description language, NED.

This development framework provides also: support to parallel distributed simulations using MPI [28], GUI support for developing the code and for running/debugging the simulations, tools for data analysis and statistics collection and support for common network protocol like TCP and WiFi communication[1].

### 3.3.4 Framework selection

Among all the frameworks considered in the surveys done over the years and the final considerations on the three presented in this section, OMNeT++ has been selected to be the framework to use in this project for the following reasons:

- It is under constant development;
- It provides mature support for parallel distributed simulation over MPI;
- It has a comprehensive and powerful scripting language to define module and network topologies;
- It is largely used and well considered in the scientific community;
- It has a well engineered modular architecture which can be easily expanded to implement custom network architectures.

## 3.4 Implementation of the simulator

### 3.4.1 General architecture

The simulator is implemented in a modular way which allow an easy implementation of different network topologies, routing algorithms and traffic generators. The base functionality of the APEnet network is implemented in a VCT simulation library which provides the base components of a network peer described in § 2.3. All the modules of this library are completely unaware of the network topology and the routing algorithm, this is possible thanks to the power of C++ and the use of virtual functions.

The building blocks provided can be used to generate network nodes to be used in an higher level describing the network topology. For each implemented topology we need to specify, other than the structure of the node, all the topology specific parameters and variables such as network coordinates.

The last piece of this schema is to implement the specific functionality of routing algorithms and traffic generators. The behaviour of a router is strictly correlated

[1] This is not relevant for the present project because one of the goals is to implement the custom protocol of the APEnet network.

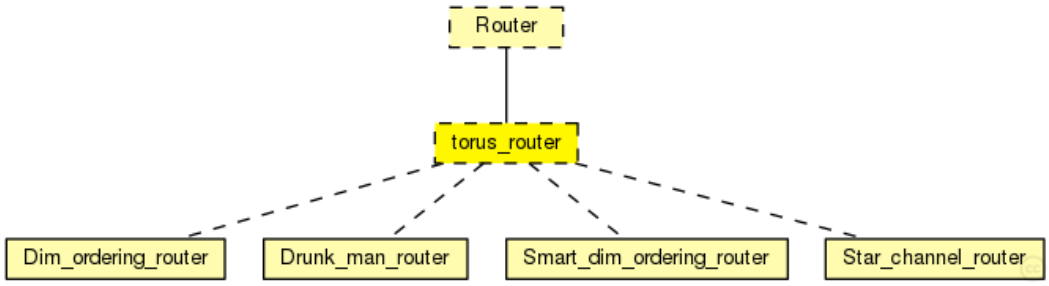



**Figure 3.1.** Inheritance diagram of the router for a torus network. The first module for above is the topology-unaware VCT simulation library component, the second is the topology specialized one and the third level contains all the different available routing algorithms.

to the underlining network topology, therefore to implement specific routers we can inherit from the generic router class for the given topology. The actual routing function is implemented as pure virtual function defined in the VCT simulation library router class. Figure 3.1 depicts the inheritance diagram with the three layers of abstractions for a torus network router.

A *traffic injector* is a component of the simulation which takes care of sending the packets into the network replacing the applications running on the real system. The emulation of the network part of an application can be represented by two entities a *consumer* and a *producer*, the first is responsible of retrieving the data from the network and the second is responsible of sending new data into the network. In order to provide a realistic reaction to the incoming traffic the consumer must interact with the producer providing informations about the inbound packets. Traffic injectors are not strictly bound to a specific network topology, therefore, to achieve a better modularity and re-usability of the code, we use a different approach to implement the consumers/producers. We create a topology unaware traffic generator which uses integer indexes to address different nodes, the same strategy used by MPI with the rank IDs, indexes are then translated into real network coordinates by the topology specific traffic handler. The use of C++'s templates makes this option viable and easy to implement.

As described in § 3.3.3 an OMNeT++ simulation model is composed by simple modules which communicate trough messages. Both messages and modules are C++ objects and can be specialized to provide the previously discussed structure. This, together with the configurable and hierarchical nature of the NED language, makes the simulation model extremely flexible and easily expandable to new topologies, routing algorithms and traffic injectors. Reusing large parts of the code helps with the consistency of the simulations between different configurations.

### 3.4.2 VCT simulation library

In this section we present all the modules necessary for building network nodes; an example of network node is depicted in Figure 3.2.

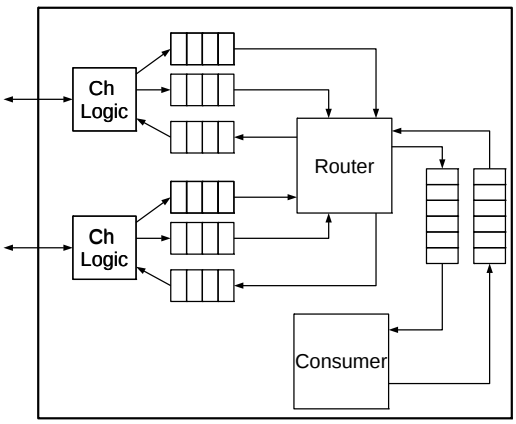



**Figure 3.2.** An exmple of a network node build using the modules provided by the VCT simulation library. This particular node has two external channels, whith two virtual channels each, and one internal channel.

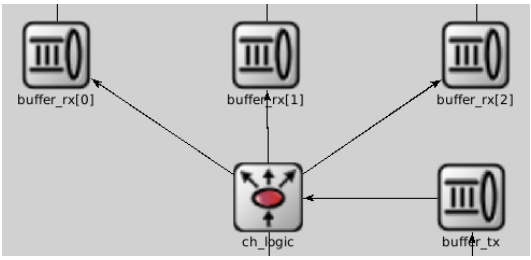



**Figure 3.3.** Internal structure of the *Channel logic* compound module for a three virtual channel configuration.

### Buffer

This module provides the FIFO buffer functionality needed to implement transmission channels in a network peer. It has: a fixed size specified by a parameter, an input port, an output port and two bidirectional control ports. The data coming from the input port are stored into the FIFO buffer, a *buffer overflow* causes a fatal simulation error. The consumer can request data sending a message to the control port of the module, buffer underflow requests are ignored. The module sends through the control ports messages containing informations about the internal buffer state to the producer and the consumer.

### Channel logic

This module implements the functionality of the APEnet link control:

- multiplexing/demultiplexing of the data into the virtual channels;
- sending flow control credits on the channel;

- forwarding received flow control credits to the router.

This module has one bidirectional port towards the physical channel, one consumer interface from the TX FIFO, one producer interface per virtual channel towards the RX FIFOs and one port to forward credits to the router. Flow control packets contains the number of consumed words in each virtual channel buffer for that specific physical channel in a differential way. Those packets are sent at fixed time intervals like in the APElink protocol and the frequency is specified as a parameter; to speed up, the simulation flow control packets are sent only if the buffer occupancy has changed since the last flow control packet. This procedure is equivalent to the one used in the APElink protocol and it is less prone to events race condition problems, nevertheless it is less reliable on a real hardware system in the event of the loss of a flow control packet.

The library provides a compound module which combines *Buffers* and *Channel logic* modules to ease the implementation of communication channels. As in the real APEnet board the buffers of the virtual channels are only replicated on the receiving side. This strategy does not change the behaviour of the virtual channels: the router takes care of forwarding only flits that have free space on the receiving RX buffer. Figure 3.3 depicts the internal structure of a three-virtual-channel module.

### Consumer

This module combines into a single entity the functionality of a consumer and a producer, so it has one consumer interfaces to the RX FIFO and one producer interface to TX FIFO. The module pops all the messages in the RX queue as soon as they are available, the incoming messages handling function provides only statistics generation and keeps popping messages from the buffer, it can be overridden to provide a more specific traffic handling.

The injection of new packets into the network can be done in two different ways:

- injecting a packet directly into the FIFO;
- injecting a packet using an internal "software" queue.

The first procedure can fail if there is not enough room available into the buffer, the use of the internal soft buffer is recommended to avoid blocking the consumer and because it gives a more realistic environment. The TX and RX FIFOs are a representation of the hardware buffers on the Network Interface Card (NIC) itself, the soft queue can be seen as a software buffer inside of the compute unit. Scheduling new packets uses an intermediate class (*Packet_info*) which contains all the information required to generate the full network packet. To generate the real packet for a specific topology the virtual function generating the header must be defined in the specialization class for that given topology.

### Router

This module has all the required gates to connect with an arbitrary number of external ports using *Channel logic* modules and an arbitrary number of internal ports using *Consumer* and *Buffer* modules. The *Router* takes care of forwarding

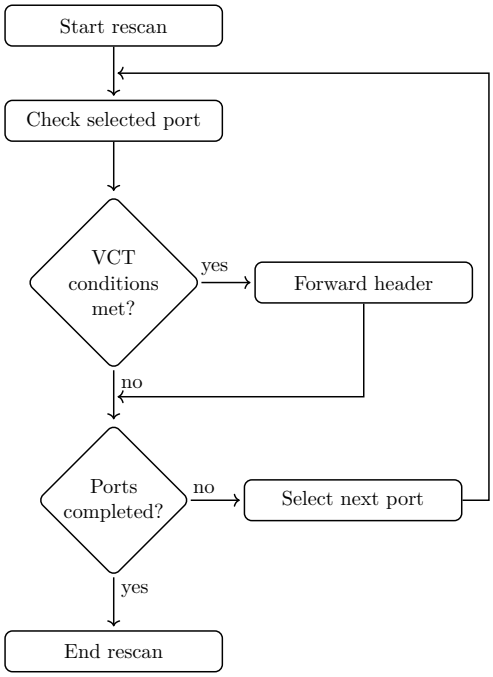



**Figure 3.4.** Flowchart of the port rescan of the router. This cycle through all the output ports for headers ready to send.

all the incoming packets to their next destination, according to the selected routing algorithm, and avoiding local and remote buffer overflow. Every time a new header is popped from one of the RX FIFOs it is stored into a dedicated slot, one for every input channel. The routing function is called to get the correct output port for the packet. The routing function is implemented as a pure virtual function in the C++ code and it must be implemented in the specialized version of the Router class. After the port selection, a request is issued to the arbitration system. Concurrent requests are handled using priority queues to offer the possibility of traffic prioritization in the future, the default priority algorithm is just a FIFO one based on the arrival time of the header, different ones can be implemented in order to experiment with QoS.

The Router checks the request queues for all the ports for packets that can be forwarded as depicted in Figure 3.4. A packet can be forwarded only if it matches all the constraints imposed by the VCT routing schema explained in § 2.5.3. If one or more of this conditions are false the packet is blocked until the congestion is over. If all these conditions are true then the port is assigned to that packet and all the subsequent flits will follow the header as soon as they are available. After forwarding the footer the channel is freed and it becomes available to the next packet. Because of the event driven nature of the simulation the scan of the requests queues is triggered every time a new event may change the status of one of the conditions. Table 3.1 indicates all the incoming events that triggers a request rescan.

**Table 3.1.** Events that triggers an output port queue rescan because the may alter the VCT conditions for a queueing packet.

| Incoming event | Possible effect on the VCT conditions |
|---|---|
| New incoming header | The requested channel may be available |
| New incoming footer | One channel is now free for other packets |
| Local TX FIFO occupancy update | The local TX FIFO may now have enough room for the next packet |
| Remote RX FIFO occupancy update | The remote RX FIFO may now have enough room for the next packet |

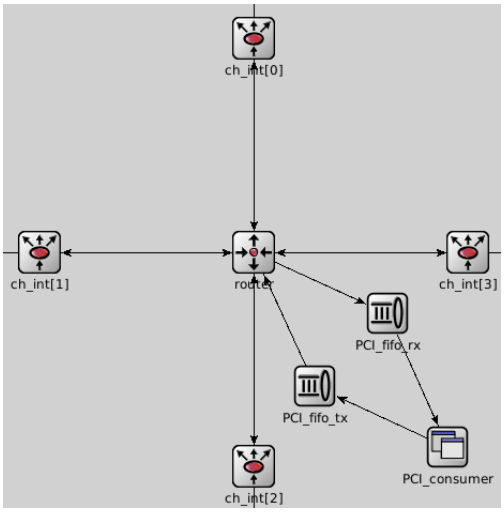



**Figure 3.5.** Internal structure of a 2D-torus node. The 4 external ports required by the topology are implemented using *Channel logic* modules, the internal port uses a *Consumer* module and two *Buffer* modules for the hardware buffering. The *Router* module in the middle provides routing functionality between the ports.

### 3.4.3 Torus topology implementation

In this section we will discuss how to implement a torus network using the VCT simulation library.

#### Building the network

First we need to build a node for the network. An N-dimensional torus node as $2N$ bidirectional ports connected to the first neighbours and an internal port connected to the local compute unit, the interconnection between those ports is provided by a router with $2N$ external ports and one internal port. Figure 3.5 shows the internal structure of a 2D torus node. To connect all the nodes and create a torus topology we need to create a network of nodes using the NED language. As an example of NED use, listings 3.1 and 3.2 show respectively the connections for a 2D and a 3D tori, the nodes are instantiated as an unidimensional array which is then treated as

**Listing 3.1.** NED connections for a 2D tourus.

```
1 connections:
2   for i=0..(rows−1), for j=0..(columns−1) {
3     node[i∗columns+j].gate[0] <−−> Channel <−−> node[i∗columns+(j+1)%
          columns].gate[1];
4     node[i∗columns+j].gate[2] <−−> Channel <−−> node[((i+1)%rows)∗
          columns+j].gate[3];
5   }
```

**Listing 3.2.** NED connections for a 3D tourus.

```
1 connections:
2   for x=0..(x_max−1), for y=0..(y_max−1), for z=0..(z_max−1){
3     node[x + y∗x_max + z∗x_max∗y_max].gate[0] <−−> Channel <−−> node
          [(x+1)%x_max + y∗x_max + z∗x_max∗y_max].gate[1];
4     node[x + y∗x_max + z∗x_max∗y_max].gate[2] <−−> Channel <−−> node
          [x + ((y+1)%y_max)∗x_max + z∗x_max∗y_max].gate[3];
5     node[x + y∗x_max + z∗x_max∗y_max].gate[4] <−−> Channel <−−> node
          [x + y∗x_max + ((z+1)%z_max)∗x_max∗y_max].gate[5];
6   }
```

a multidimensional one.

### Expanding the library for tori

In order to implement a torus topology we need to expand the VCT simulation library providing topology specialized *Router* and *Consumer* modules. Those modules must be capable of determining positions of nodes on the network therefore we need to specify an address system, for an N-dimensional torus the obvious choice is to use Cartesian coordinates to identify the position of the nodes. Because the nodes are declared in the NED network as an unidimensional array we need to provide a conversion function from array indexes to network coordinates to allow any *Router* and *Consumer* to establish its own coordinates; if we want to take advantage of the templated *Consumers* we need to provide a conversion function for network coordinates to indexes. For a three dimensional torus the conversion equations are the following:

$$x = index \% x_{max} \tag{3.1}$$

$$y = (index / x_{max}) \% y_{max} \tag{3.2}$$

$$z = index / (x_{max} y_{max}) \tag{3.3}$$

$$index = x + y x_{max} + z x_{max} y_{max} \tag{3.4}$$

All the operations are performed on integer numbers so the / and the % operators represents integer division and modulo respectively.

### Implementing routing algorithms

The algorithms implemented for N-dimensional tori are the ones presented in § 2.7, in this section only technical details related to the actual implementation will be discussed.

All the algorithms need to calculate the shortest path to the destination on a given dimension and to correctly identify packets that have already crossed the boundaries of the network. To calculate the shortest path direction we can check if the distance between the current node and the destination is smaller than the size of the torus divided by two. In the $x$ dimension if we call $x_p$ the destination coordinate of the packet and $x_n$ the coordinate of the node, the shortest path is in the + direction if this condition is true:

$$\left((x_p \leq x_n) \wedge \left(|x_p - x_n| \leq \frac{x_{max}}{2}\right)\right) \vee \left((x_p > x_n) \wedge \left(|x_p - x_n| > \frac{x_{max}}{2}\right)\right) \quad (3.5)$$

This condition can be used in all the other dimensions by changing the coordinates to the appropriate ones.

In order to correctly identify packets that have crossed the network boundaries in direction + (−) we could check if the source coordinate is smaller (greater) than the coordinate of the node. This condition can be used only if we assume that packets use only minimal paths, therefore it is not valid if we want to implement non minimal algorithms. To correctly identify the packets in both minimal and non minimal conditions we can add a special field into the packet header which contains this information.

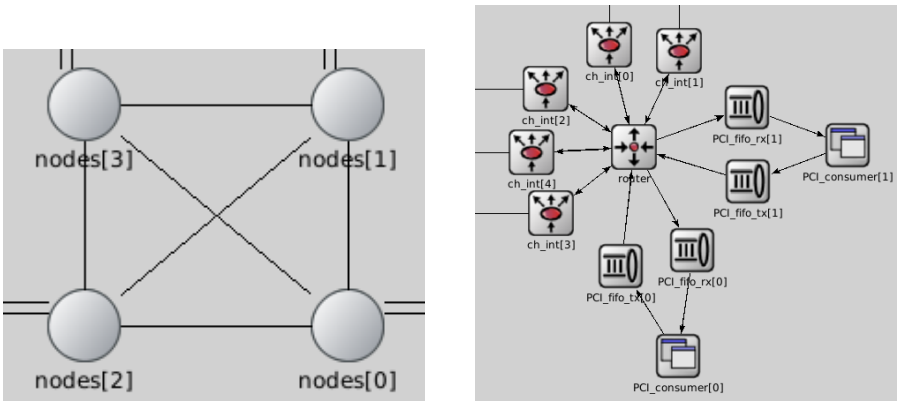
## 3.4.4 Fully connected dragonfly topology implementation

In this section we will discuss how to implement a direct fully connected dragonfly network using the VCT simulation library.

### Building the network

To implement a dragonfly topology we need groups, routers and nodes. Because we want to create a direct network version of the dragonfly we combine routers and nodes in a single entity called node, every node has a number of internally connected endpoints identified by a port. To build this network topology using OMNeT++ we need to create two compound modules one for the groups and one for the nodes and then connect them using a fully connected topology. Figure 3.6a and Figure 3.6b depicts the internal structure of the group and node compound modules respectively. The number of external and internal ports of a dragonfly node are determined by the parameters introduced in § 2.2.2. The parameters can be modified in order to generate different network configurations and to evaluate the behaviour of the topology.

The fully connected topology can be described using the NED language in a parametric way as shown in listings 3.3 and 3.4 for connections within a group and between groups respectively.




**(a)** Example of the group compound module

**(b)** Example of the node compound module

**Figure 3.6.** Internal structure of the group and node compound modules for 72 nodes fully connected dragonfly.

**Listing 3.3.** NED intra-groups connections for a fully connected dragonfly.

```
1  connections:
2      for k=0..(nodes_in_group-1), for j=0..((groups-1)/nodes_in_group)
           -1 {
3          gate++ <--> nodes[k].gate[j];
4      }
5
6      for k=0..(nodes_in_group-1), for j=((groups-1)/nodes_in_group)..((
           groups-1)/nodes_in_group)+nodes_in_group-1 {
7          nodes[k].gate[j-1] <--> Channel <--> nodes[j-((groups-1)
               /nodes_in_group)].gate[k+(groups-1)/nodes_in_group] if (
               j-((groups-1)/nodes_in_group))>k;
8      }
```

**Listing 3.4.** NED inter-groups connections for a fully connected dragonfly.

```
1      connections:
2          for k = 0..groups-1, for j = 0..groups-1{
3              groups_modules[k].gate[j-1] <--> Channel <-->
                   groups_modules[j].gate[k] if j>k;
4          }
```

### Expanding the library for dragonflies

We need to expand the topology unaware VCT library and to specialize the *Router* and *Consumer* classes for fully connected dragonfly topology. As we did for N-dimensional tori, we need to specify an addressing system to identify in a unique way all the nodes in the network. The dragonfly topology has a 3 level hierarchical structure therefore an effective and smart way of identifying the nodes is to use three integer indexes one for the group, one for node inside the group and one for the internal port in the node. The network is built using an array of groups and every group has internally an array of nodes, every node can determine its position in the network by using the indexes of the two arrays as group and node index.

In order to take advantage of the templated consumer library, we need to provide conversion functions between network coordinates and global network indexes. A possible mapping can be described by the following equations:

$$g = index/\left(n_{max}p_{max}\right) \tag{3.6}$$
$$n = \left(index\%\left(n_{max}p_{max}\right)\right)/p_{max} \tag{3.7}$$
$$p = index\%p_{max} \tag{3.8}$$
$$index = p + np_{max} + gp_{max}n_{max} \tag{3.9}$$

Where $g$, $n$ and $p$ represents group, node and port indexes respectively. All the operations are performed on integer numbers so the / and the % operators represents integer division and modulo respectively.

### Implementing routing algorithms

The algorithm implemented for fully connected dragonflies is the one presented in § 2.7, and in this section, only technical details related to the actual implementation will be discussed.

The full routing problem can be decomposed into routing packets on a fully connected topology, which can be easily achieved by using this pseudo code:

```
1  if (index > ref_index)
2      result = index − 1;
3  else
4      result = index;
```

Where index is the destination coordinate and ref_index is the current node coordinate.

For a packet with a different $g$ index the routing function must be able to determine if the packet needs to be routed in the local or directly into the global one. This decision can be made by checking if the resulting port belongs to the current node or not. Packets arriving from different groups must use a dedicated virtual channel to avoid deadlock.

## 3.4.5 Consumers implementation

In this section we will discuss the implementation of specific packet injector used to characterize the different networks configurations. The traffic generator are spe-

cialization of the *Consumer* class presented in § 3.4.2 and they use global indexes to address nodes in different network topologies.

The traffic injectors provided offer synthetic and real-world packet generation to characterize and optimize networks for different workloads. In particular we can use different uniform traffic injectors, a ping-like probe sender and DPSNN traffic handler.

### Uniform traffic injector

This traffic injector sends random packets at a fixed bandwidth. All the packets have a random size and they are sent to randomly selected nodes at an adeguate frequency to maintain the selected bandwidth. The pseudo random number generator uses the Mersenne Twister algorithm implemented inside the OMNeT++ framework, this generator has a period of $2^{19937} - 1$ and 623-dimension equidistribution property [29]. The packets are sent using the software queue therefore, if a network congestion occurs, the effective injected bandwidth may be lower that the selected one. This traffic pattern is very useful to characterize networks against a non specific application.

### Uniform constant packet number injector

This synthetic traffic generator simulates the behaviour of an application producing a random uniform traffic. In a real application a packet is usually sent after receiving some information from other nodes; to mimic this traffic pattern the injector sends a new packet every time a packet is received. The new packet is sent to a random destination and it has a random size, while random numbers are generated using the Mersenne Twister algorithm between the minimum and maximum values allowed. At the beginning of the simulation every node sends a fixed number of packets to random destinations, the number of packets injected by every node into the network can be configured using a parameter to generate variable traffic conditions. The effective behaviour of this injector is to keep constant the number of packets travelling over the network.

### Ping traffic injector

This traffic injector can be used to measure the latency between a couple of selected nodes called *source* and *sink*. The source sends packets minimum size packets to the sink at a fixed rate, the sink calculates the latency of the received packets. This generator tags the probe packets to distinguish them, this makes possible to other traffic generator below this one to study how other traffic can affect latency between two point. It must be noted that the latency between fixed points of the network is a local observable and it changes significantly over time, therefore it cannot be used to fully characterize a network configuration. Nevertheless it can be useful to measure the latency on critical paths of the system.

### DPSNN traffic injector

This traffic reproduces the traffic generated by the DPSNN application described in § 1.3. To reproduce the application behaviour the traffic injector uses a *trace file* containing the number of spikes sent by the MPI processes at every simulation iteration. The injector reproduces all the simulation iterations of the file and keeps the nodes synchronized emulating the real application.

Every simulation step of the application can be simplified as follows:

1. MPI barrier[2] to sync all the nodes
2. Every node sends and receives spikes using MPI alltoallv
3. Every node processes and calculates the next step

This simplified schema is effective for simulating the network traffic generated by the application and it requires the implementation of two communication patterns: *barrier* and *alltoallv*. In order to implement the simulation cycle in an event driven simulation like OMNeT++ we can use a Finite State Machine (FSM) which uses the three states defined previously. We can divide the states into two classes: network driven states and time states. A network driven state is a state which needs to receive a certain amount of packets in order to be completed. A time driven state is completed after a certain amount of time. The calculation state is a time one while all the others are network states. A time state can be easily implemented using a self scheduled message which triggers the update of the FSM when received. Network states are more complex and need a different implementation. When the module enters into a network state we need to track down the number of expected packets from every node, this can be easily done using an associative array, every time a packet is received we can decrease the number of expected packets from that given node. This first naive proposal could work if all the different nodes change state in a synchronous way but, because we do not have any external control over the global state of the nodes, we have to expect packets from different states at the same time, therefore we need to flag packets according to their state. Received packets belonging to a different state are saved into a buffer system on the *Consumer* itself to avoid loosing information. Packets into the buffer system are then recovered every time the FSM changes states into a new network one. We can summarize the general structure of a network state as follows:

1. All the outgoing packets are sent using the "software" buffer of the consumer
2. The number of expected packets from every node is stored into the associative array
3. The packets belonging to the current state are recovered from the buffer system
4. All the received packets for the current state are counted, the others are saved into the buffer system

[2]The barrier is not needed by the application itself but it is useful if we want to profile the different part of the application

5. When the number of expected packets is zero the machine transitions into the next state

The next step is to implement the network traffic pattern of the MPI [30] functions used by the DPSNN simulation: *barrier* and *alltoallv*. The *barrier* function synchronizes all the MPI processes of the network exchanging packets between the nodes, the OpenMPI's implementation of the barrier function uses a root node approach and can be described by the the following steps:

1. All the nodes but the root one send a packet to the root node signalling that they reached the barrier
2. The root node waits for one packet from every node
3. When it receives all the packets the root node sends an acknowledgement to the nodes and releases the barrier
4. All the nodes leave the barrier after receiving the ack from the root node

The root node must be correctly identified by all the nodes, in this implementation we use the node with index zero as root node. For large networks this communication scheme is not effective because it induces congestion in the root node, to mitigate this effect we can use a hierarchical structure of local and global root nodes: every node sends the barrier packet to the local root node, the local root node send on barrier packet to the global one after receiving all the barrier packets, the global root node sends the barrier ack to the local root nodes and the local root nodes propagates the ack to their local nodes.

This schema of communication can be implemented in the network state machine paradigm defined before by using two network states: *MPI_BARRIER_SEND* and *MPI_BARRIER_ACK*. In the *MPI_BARRIER_SEND* state all the nodes sends a packet to the root node and do not wait for any packet, the root node waits for one packet from every node and does not send any one. In the *MPI_BARRIER_ACK* state all the nodes waits for one packet from the root node while the root node sends one packet to every node.

In the *alltoallv* every node sends packets to the other nodes and waits for packets from other nodes, therefore this communication pattern can be mapped using one network state. The user must specify the number of packets to send and receive and size of every packet, this is done by using vectors. If the number of packet is zero the send or the receive is skipped. Every cycle of the DPSNN uses two different *alltoallv* one for sending the number of spikes and one for sending the actual spikes, therefore two network states will be used to implement the two different communications.

The time spent by the application to calculate the next state of the neurons inside the process can be considered proportional to the number of spikes received, the average calculation time per single spike can be estimated by profiling the real application. The processing time can change drastically by changing the underling computing hardware, nevertheless we can still compare different networking solutions.

# Chapter 4

# Selection and analysis of simulations and benchmarks

The benchmarking and characterization of an interconnection network is difficult due to the behaviour of the network. It is very susceptible to: the traffic pattern, the size of the buffers, the size of the network, the number of virtual channels and many other parameters. In this chapter we will define the standard metrics used for benchmarking networks, we will discuss the results for different networks configuration using synthetic benchmarks and we will evaluate the performances using the DPSNN traffic.

## 4.1 Metrics for an interconnection network

The most used metrics for an interconnection network are *latency* and *accepted traffic*.

*Latency* is defined as the time elapsed from the beginning of the packet transmission and the receive of the message at the destination node. This definition can be interpreted in different way according according to the context: if we consider a complete system with all the software stack implemented the latency will include all the time spent into the different software layers; on the other hand, if we want to characterize raw network performances, only the time spent into the network will be taken into account. Because we are benchmarking low level implementation of networking hardware we will not add the *consumer* soft queue time into the packet latency. It is important to note that the goal of the simulations performed in this thesis is not to give a good modelling of the software stack or the system bus interfacing the network hardware to the processor, so adding the soft queue delay to the latency will not provide a correct modelling of the real application latency.

Latency changes from packet to packet because it takes into account: the distance between source and destination, the size of the packet and the congestion of the network. The value for a single packet is not meaningful, especially if we consider a synthetic traffic pattern; so we used the average value to give a global picture of the network behaviour. The standard deviation of the latency is also important to get an indication of when the network is heavily congested.

*Accepted traffic* or throughput is defined as the amount of information delivered by the network per time unit. Because the amount of information delivered

**Table 4.1.** Server characteristics.

| | |
|---|---|
| CPU | 2 x Intel Xeon CPU E5620 @ 2.40 GHz |
| Memory | 48 GB |
| Network card | Mellanox MT26428 Connectx2 |
| OpenMPI version | 1.10.3 |
| Linux version | CentOS 7.2 kernel 3.10.0-327.22.2 |

depends on the number of nodes and the bandwidth per node this this value must be normalized in order to compare different network configurations. In our test configurations, since the bandwidth to and from every node does not change, we normalize the accepted traffic dividing it by the number of nodes in the network.

It is important to note the difference between applied load and accepted traffic: the first is an input parameter of the simulation and determines the amount of data injected into the network; the latter is an output of the simulation and determines the amount of data delivered by the network.

## 4.2 Performance plots

There are two different standards for the visual representation of the performance plots: Chaos Normal Form (CNF) and Burton Normal Form (BNF).

The CNF method uses two different graphs one for accepted traffic vs normalized applied load and one for latency vs normalized applied load. The use of two plots makes the results more readable and they show a clear picture of network behaviour before and after saturation.

The BNF method uses a single combined plot of latency vs accepted traffic. Because the accepted traffic is not an independent variable the resulting plot may not be a function, resulting in a more complex visualization.

The results of this thesis will be presented using the CNF method because it is more readable.

## 4.3 Application scaling

Before testing the performances of the simulated networks it is interesting to test the scaling of the simulator on a commodity cluster: the computing platform used is based on intel CPUs and mellanox network interfaces, the full hardware/software configuration is shown in Table 4.1.

The scaling test has been performed on a 2D torus 96x96 and the results are shown in Figure 4.1. The performance speedup of the application is very promising and close to the theoretical one: the simulation running on 96 processes achieves the 81% of the theoretical limit. This good performance is due to the highly optimized OMNeT++ MPI implementation and to the short range communication of the torus.

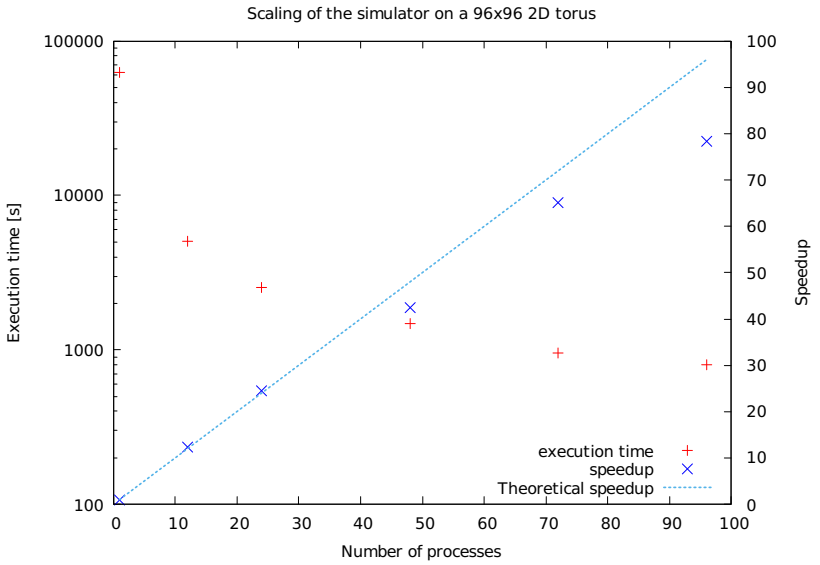



**Figure 4.1.** Scaling of the simulator for a 96x96 2D torus using MPI.

## 4.4 Synthetic tests

As a first evaluation of the network performance we use synthetic benchmarks on different network configurations. The traffic generator used for this first test is the random uniform consumer described in § 3.4.5.

The tests where performed for 2D, 3D tori and dragonfly in the following configurations:

| Network topology | routing algorithm |
|---|---|
| 2D torus 10x10 | e-cube, star-channel, smart dim-order |
| 2D torus 32x32 | e-cube, star-channel, smart dim-order |
| 3D torus 10x10x10 | e-cube, star-channel, smart dim-order |
| Fully connected dragonfly 72 nodes | min-routing |
| Fully connected dragonfly 1056 nodes | min-routing |

The results of the CNF plots for the accepted traffic and latency are shown in Figure 4.2 and 4.3 respectively.

### 4.4.1 Accepted traffic

The analysis of the accepted traffic plot shows a linear region common to all the different network configurations tested. When the network is in the linear region it is below its critical congestion point and can handle properly the incoming traffic, enhancing the applied load results in higher accepted traffic. If the applied load is above the saturation point the accepted traffic starts to exit from the linear region of the plot and becomes flat. It is important to note that the plateau value may not be the maximum value that the network can deliver, this is more visible in the 32x32 torus with a drop of almost a factor two between the maximum and

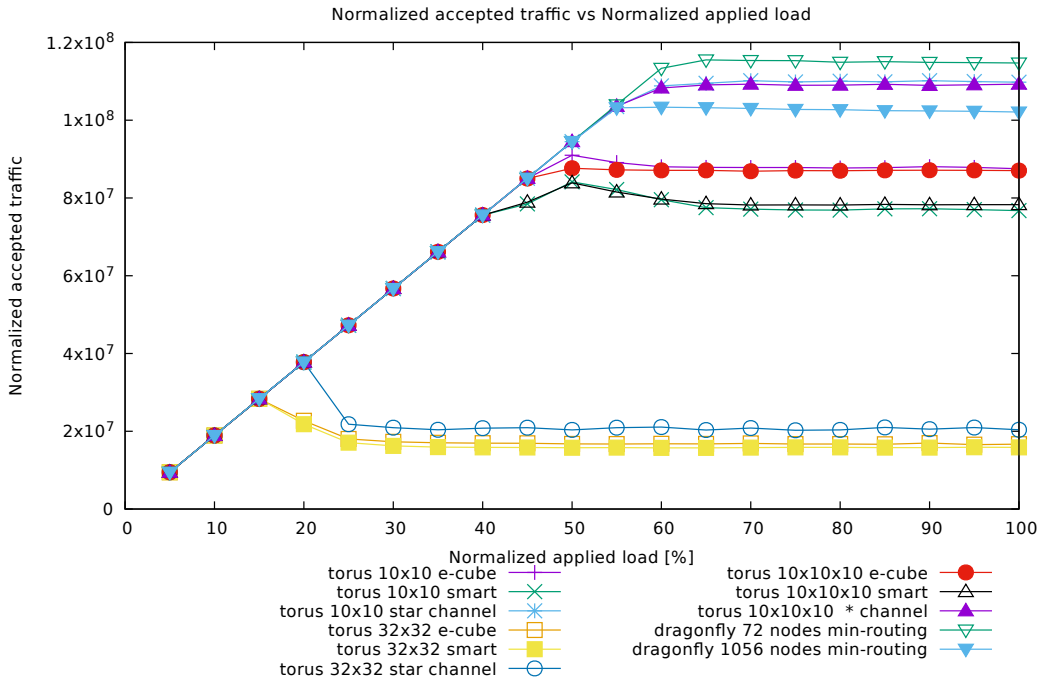



**Figure 4.2.** Normalized accepted throughput vs applied load for the different configuration tested.

the plateau. This performance drop is due to higher pressure applied to network infrastructure resulting in packets stealing resources each other. in order to reduce this back-pressure effect a possible solution is to apply throttling techniques, which limits the applied traffic, in order to prevent an oversaturation of the network.

### 4.4.2 Latency

The latency graph shows a similar picture with an increasing trend while the applied load increases and a sudden jump of about one order of magnitude after the network reaches saturation. It is important to note that even if the network is not saturated and the accepted traffic has not reach the limit yet, latency is increasing together with the applied load leading in longer delivery time for the packets. This aspect must be taken into account when designing the system in order to met the requirements.

### 4.4.3 Comparison of tori

The different network topologies tested react differently to the applied traffic reaching different levels of saturation for both throughput and latency. The two 2D tori show how this simple topology can handle 40-50% normalized applied load in a 10x10 network configuration, but only 15-20% in a 32x32 configuration. Tori and meshes are not optimized for uniform network traffic because of their small set of neighbour nodes, and when the network radius goes from 10 (for the 10x10) configuration to 32 (for the 32x32 one) the effect on performances is clearly evident.

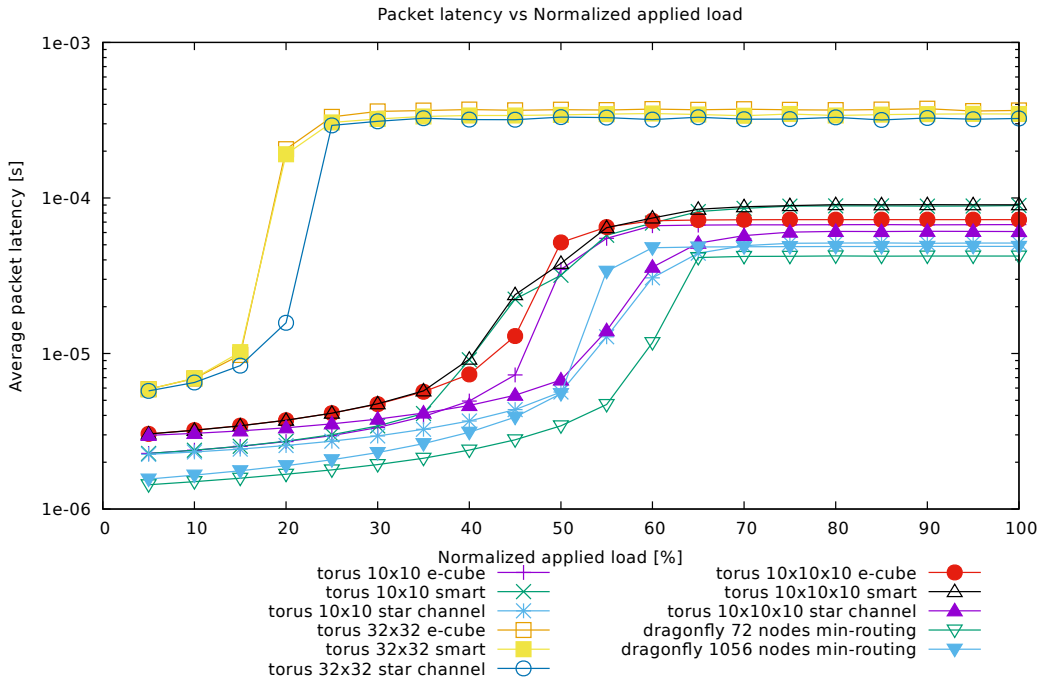



**Figure 4.3.** Latency vs applied load for the different configuration tested.

In order to reduce the network radius while keeping the same kind of topology we can move from a 2D to a 3D torus. The 10x10x10 3D torus tested reaches results similar to the 10x10 one for both accepted traffic and latency, while having almost the same number of nodes of the 32x32 network.

### 4.4.4 Comparison of routing algorithms

For the configurations tested, the use of the fully adaptive star-channel routing algorithm provides a better use of the available network resources resulting in higher sustained loads and lower latency than the ones achievable by using the simpler e-cube routing.

The smart dimension-order routing function does not provide significant improvements over the standard dimension ordering and leads to performance degradation in certain configurations. The main problem with this algorithm is the lack of global knowledge regarding the network state and the limited adaptivity provided by reusing wasted resources. The rationale behind the algorithm is to avoid hot spots of network congestion in specific dimension by misrouting the packet in a different one; in a uniformly loaded network, hot spots are less likely to occur therefore using non minimal paths to avoid them it is not a good strategy.

### 4.4.5 Evaluation of dragonfly

The fully connected dragonfly shows good performances in the two configuration tested. The configuration with 72 end nodes performs better than tori with the fully adaptive algorithm, while the 1056 configuration shows better performances

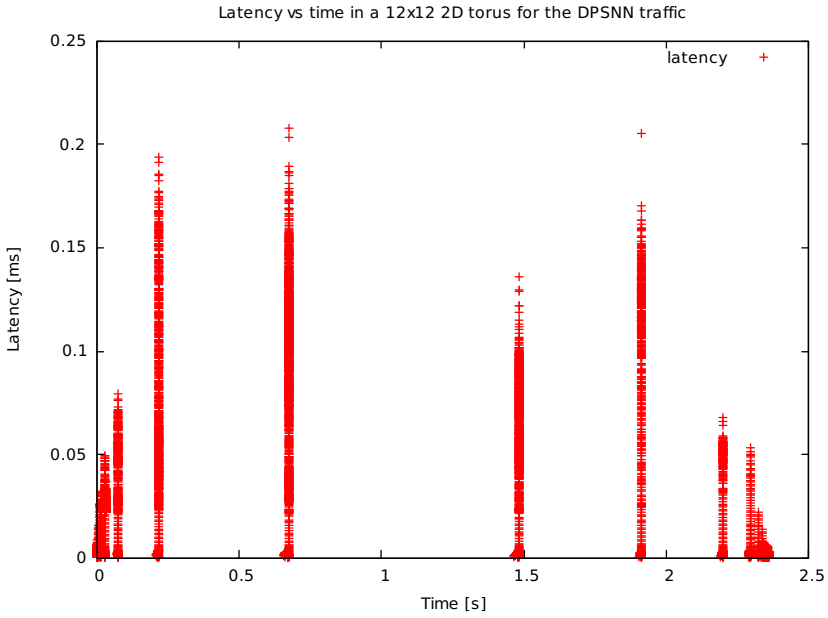



**Figure 4.4.** Latency of the received packets during a 12x12 processes DPSNN simulation mapped onto a 12x12 torus.

than the non adaptive tori but lesser accepted traffic that the fully adaptive ones. It must be noted that the fully connected dragonfly is an higher degree/radix network than the comparable size torus, so the 72 nodes dragonfly requires routers with 5 external and 2 internal ports while the 1056 nodes one requires 11 external and 4 internal ports. On the other hand the 10x10 and the 10x10x10 tori require 4 and 6 external ports respectively and only one internal port. We can say that building direct network's nodes with an high port count is a difficult task and there are strong limits imposed by the technology available.

## 4.5 DPSNN testing

The traffic generated by the current release of the DPSNN is not particularly demanding in terms of throughput: the data structure that represents an axonal spike is only 12 bytes and evolving the neuronal dynamic produces temporal intervals between the sends. The real challenges with this application are the bursty nature of the traffic and the low latency requirements.

To analyse the performance of different network configurations with the DPSNN traffic we cannot use CNF plots because we cannot adjust the applied load. To characterize the behaviour of the network we can do a time-based application profiling based on the time spent by the consumer state machine in each state of the simulation.

Before profiling the application it is useful to look at shape of the network traffic over time. Figure 4.4 depicts the latency of the received packets vs time during the DPSNN simulation of a 12x12 cortical column matrix divided into 144 MPI processes mapped onto a 12x12 torus. As we can clearly see from the plot, the

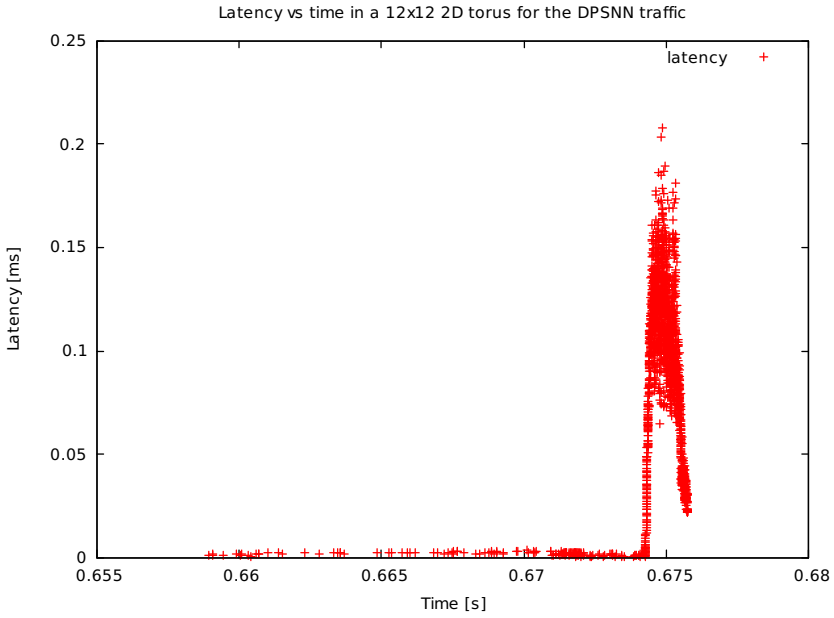



**Figure 4.5.** Latency of the received packets during a 12x12 processes DPSNN simulation mapped onto a 12x12 torus around a single spike.

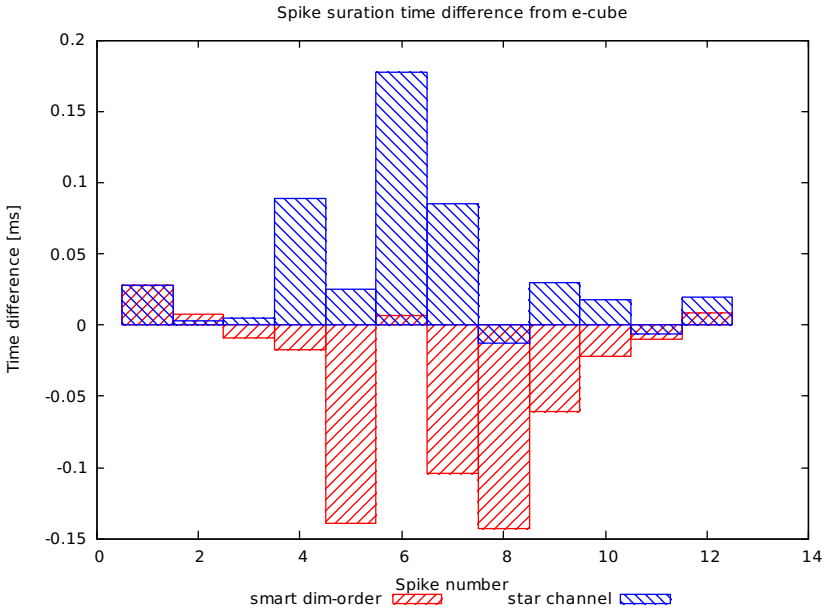



**Figure 4.6.** Time difference between the dimension order and other routing algorithms.

**Table 4.2.** Different DPSNN configuration tested.

| # of columns | Column X | Column Y | Columns/Process | Neurons |
|---|---|---|---|---|
| 288 | 16 | 18 | 4 | 357120 |
| 576 | 32 | 18 | 8 | 714240 |
| 576 | 16 | 36 | 8 | 714240 |

**Table 4.3.** Network performance difference relative to the 2D torus with the e-cube routing algorithm.

| Configuration | 2D torus star-channel | 2D torus smart dimension-order | dragonfly min-routing |
|---|---|---|---|
| 16x18 | 12.4% | -4.6% | -30.3% |
| 32x18 | 19.7% | 0.66% | -10.0% |
| 16x36 | 24.1% | 3.6% | -67.1% |

bursty nature of the traffic is confirmed by the simulation, and this traffic shape is generating congestion resulting in high latency value measured during the traffic spikes. Figure 4.5 shows a more detailed view of one of the spikes revealing the real shape of the traffic: in the first part we have low latency and therefore no congestion; in the second part we have a sudden increase of the latency which drops rapidly creating a spike. This behaviour can be explained by analysing the flow of the application: in the first part the nodes are transitioning from the *CALC* state to the *MPI_BARRIER_SEND* one sending the MPI_BARRIER packets to the root node; when the root node receives all the packets it sends the *MPI_BARRIER_ACK* packets triggering the transition to the send/receive of the synaptic spikes generating high traffic and congestion into the network.

As a preliminary study, we measured the amount of time needed to process each of the spikes shows in Figure 4.4 for different routing algorithms. Figure 4.6 shows the difference in time between the reference e-cube algorithm and the other two available for tori. The star-channel algorithm shows a reduction in the time needed by the network to deliver the packets leading to a total improvement of 458 $\mu$s. The smart dimension order offers marginal benefits under certain circumstances but the total time difference is $-452\,\mu$s. From this results we should expect an improvement of $\sim$ 8% for the star-channel algorithm and a performance degradation of $\sim$ 5% using the smart dimension ordering.

The full profiling of 10 seconds of neural simulation, i. e. 10000 cycles of the DPSNN algorithm, shows a reduction in the total time spent in the send/receive spike of 9.5% by using the star-channel algorithm instead of the e-cube. If we use the smart dimension order the time spent is increased by 6.0%.

The next step is to test different DPSNN configurations with a different number of column per process on all the available configurations in order to evaluate the difference in performances with the scaling of the simulation size. The results presented are obtained from the global profiling of 5 seconds of neural simulation mapped onto 72 processes. The details of the configurations are presented in Table 4.2 and the results in Table 4.3.

The fully adaptive routing algorithm provides solid improvements in all the con-

figuration tested, showing a superior capability to handle the bursty traffic produced by the DPSNN. The smart dimension-order has an oscillating behaviour providing small fluctuations around the e-cube performances. The dragonfly topology is not handling properly the traffic produced by the neural simulation despite its lower radius and higher connectivity, this problem is probably originated by congestion of the global channels. The communication generated by the DPSNN is based on the distance between neuronal columns, as discussed in § 1.3.3, due to the high regularity of a torus network the distance between columns is highly correlated with the distance on the network; on the other hand the dragonfly topology has a hierarchical structure resulting in discontinuous mapping between column and network distance. This intrinsic characteristic of the dragonfly topology produces unbalanced local and global network traffic resulting in extra congestion.

# Chapter 5

# Conclusion and future work

In this thesis we have discussed the progress made by HPC in the last decades and we have introduced an interesting scientific application challenge: the DPSNN. We reached the conclusion that the design of a large scale interconnection system requires software simulations, in order to optimize the infrastructure and meet the design requirements.

Since there are no exascale-size systems available now, software simulations became more and more critical. Nobody can predict the behaviour of system as complex as next generation HPC infrastructure, especially because there are no real world data available for making any useful prediction. The investment in terms of economical and human resources required by a project like ExaNeSt[1] makes uncertain performances unacceptable, therefore using a flexible scalable and accurate simulator is mandatory. The one developed in this thesis fits perfectly with the requirements and it will be used to explore network solution within the ExaNeSt European project, providing a useful instrument in the early design stages. Thanks to the flexibility offered by the implemented simulator the space of all the possible network configurations can be deeply explored selecting: optimal partitioning of the system, good balance between network connectivity and performances, efficient routing algorithms and optimal overall use of the available resources.

From the implementation point of view, we have developed a low level scalable simulator of the APEnet/APElink network protocol using the OMNeT++ framework. This simulator is modular and flexible and allows benchmarking of different network configurations using both synthetic and real application traffic. Different network configuration has been tested using the simulation software obtaining interesting results summarized in the list below:

- N-dimensional tori can successfully sustain uniform traffic, so in large scale systems the use of higher dimension tori helps with the increasing radius;
- Fully adaptive algorithms provide a substantial increase in performances and should be used to achieve optimal performances;
- Non minimal routing algorithms with limited adaptivity do not provide significant advantages;

[1] The ExaNeSt project has a budget of 8.44 M€

- The Dragonfly topology is capable of good performances under uniform load providing good throughput and latency;
- The DPSNN traffic can be efficiently distributed on tori, especially if equipped with fully adaptive routing algorithms, and it is inefficient on dragonflies.

Different network topologies and routing functions will be implemented and tested in the future. An interesting test would be implementing adaptive routing for the dragonfly topology and experimenting with non uniform queuing policies and QoS. For the ExaNeSt project it is useful to test larger scale systems and multi-tier topologies, for example a dragonfly connecting different tori.

From the simulator point of view implementing network acceleration for multicast/broadcast communications could be interesting for a future version of the DPSNN. The actual implementation of the neural simulation uses the *alltoallv* to send the neuronal spikes, a further optimization of the DPSNN may use multicast sends. In order to get optimal performances the network should provide hardware acceleration of the collectives, therefore reducing the number of packets sent at every iteration generating less congestion into the network.

# Bibliography


[1] G. E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965).

[2] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. “The LINPACK Benchmark: past, present and future”. In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

[3] *Top 500 project*. URL: https://www.top500.org/.

[4] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[5] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Communications of the ACM* 31 (1988), pp. 532–533.

[6] A. J. Bernstein. “Analysis of Programs for Parallel Processing”. In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565.

[7] Pier Stanislao Paolucci et al. “Distributed simulation of polychronous and plastic spiking neural networks: strong and weak scaling of a representative mini-application benchmark executed on a small-scale commodity cluster”. In: *CoRR* abs/1310.8478 (2013). URL: http://arxiv.org/abs/1310.8478.

[8] Elena Pastorelli et al. “Impact of exponential long range and Gaussian short range lateral connectivity on the distributed simulation of neural networks including up to 30 billion synapses”. In: *CoRR* abs/1512.05264 (2015). URL: http://arxiv.org/abs/1512.05264.

[9] E. Pastorelli et al. “Scaling to 1024 software processes and hardware cores of the distributed simulation of a spiking neural network including up to 20G synapses”. In: *ArXiv e-prints* (Nov. 2015). arXiv: 1511.09325 [cs.DC].

[10] L. Lapicque. “Recherches quantitatives sur l’excitation électrique des nerfs traitée comme une polarisation”. In: *Journal de Physiologie et de Pathologie Générale* 9 (1907), 620–635.

[11] Guido Gigante, Maurizio Mattia, and Paolo Del Giudice. “Diverse Population-Bursting Modes of Adapting Spiking Neurons”. In: *Phys. Rev. Lett.* 98 (14 2007), p. 148101. DOI: 10.1103/PhysRevLett.98.148101. URL: http://link.aps.org/doi/10.1103/PhysRevLett.98.148101.

[12] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1558608524.

[13] John Kim et al. "Technology-Driven, Highly-Scalable Dragonfly Topology". In: *SIGARCH Comput. Archit. News* 36.3 (June 2008), pp. 77–88. ISSN: 0163-5964. DOI: 10.1145/1394608.1382129. URL: http://doi.acm.org/10.1145/1394608.1382129.

[14] José Duato. "A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks". In: *IEEE Trans. Parallel Distrib. Syst.* 6.10 (Oct. 1995), pp. 1055–1067. ISSN: 1045-9219. DOI: 10.1109/71.473515. URL: http://dx.doi.org/10.1109/71.473515.

[15] C.L Seitz and Wiliam J. Dally. "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks". In: *IEEE Transactions on Computers* 36.undefined (1987), pp. 547–553. ISSN: 0018-9340. DOI: doi.ieeecomputersociety.org/10.1109/TC.1987.1676939.

[16] W. J. Dally and C. L. Seitz. "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks". In: *IEEE Transactions on Computers* C-36.5 (1987), pp. 547–553. ISSN: 0018-9340. DOI: 10.1109/TC.1987.1676939.

[17] Luis Gravano et al. "Adaptive Deadlock- and Livelock-Free Routing with All Minimal Paths in Torus Networks". In: *IEEE Trans. Parallel Distrib. Syst.* 5.12 (Dec. 1994), pp. 1233–1251. ISSN: 1045-9219. DOI: 10.1109/71.334898. URL: http://dx.doi.org/10.1109/71.334898.

[18] M Katevenis et al. "The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems". In: *Digital System Design (DSD), 2016 Euromicro Conference on*. 2016.

[19] R Ammendola et al. "APEnet+: a 3D Torus network optimized for GPU-based HPC Systems". In: *Journal of Physics: Conference Series* 396.4 (2012), p. 042059. URL: http://stacks.iop.org/1742-6596/396/i=4/a=042059.

[20] R Ammendola et al. "APEnet+ 34 Gbps data transmission system and custom transmission logic". In: *Journal of Instrumentation* 8.12 (2013), p. C12022. URL: http://stacks.iop.org/1748-0221/8/i=12/a=C12022.

[21] Elias Weingärtner, Hendrik Vom Lehn, and Klaus Wehrle. "A Performance Comparison of Recent Network Simulators". In: *Proceedings of the 2009 IEEE International Conference on Communications*. ICC'09. Dresden, Germany: IEEE Press, 2009, pp. 1287–1291. ISBN: 978-1-4244-3434-3. URL: http://dl.acm.org/citation.cfm?id=1817271.1817510.

[22] Atta ur Rehman Khan, Sardar Muhammad Bilal, and Mazliza Othman. "A Performance Comparison of Network Simulators for Wireless Networks". In: *CoRR* abs/1307.4129 (2013). URL: http://arxiv.org/abs/1307.4129.

[23] *ns-3 consortium*. URL: https://www.nsnam.org/.

[24] *J-sim project*. URL: https://sites.google.com/site/jsimofficial/.

[25] *OMNeT++ project*. URL: https://omnetpp.org/.

[26] András Varga et al. "The OMNeT++ discrete event simulation system". In: *Proceedings of the European simulation multiconference (ESM'2001)*. Vol. 9. S 185. sn. 2001, p. 65.

[27] András Varga and Rudolf Hornig. "An overview of the OMNeT++ simulation environment". In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2008, p. 60.

[28] Y Sekercioglu, András Varga, and Gregory Egan. "Parallel simulation made easy with OMNeT++". In: *The European Simulation Symposium (ESS 2003)(Alexander Verbraeck 26 October 2003 to 29 October 2003)*. SCS European Publishing House. 2003, pp. 493–499.

[29] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: http://doi.acm.org/10.1145/272991.272995.

[30] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. 2012. URL: http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.