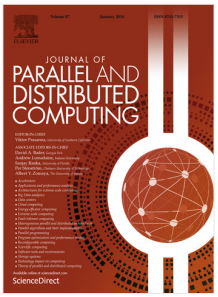
Accepted Manuscript



INRFlow: An interconnection networks research flow-level simulation framework

Javier Navaridas, Jose A. Pascual, Alejandro Erickson, Iain A. Stewart, Mikel Luján

PII: S0743-7315(19)30224-2
DOI: https://doi.org/10.1016/j.jpdc.2019.03.013
Reference: YJPDC 4030

To appear in: *J. Parallel Distrib. Comput.*

Received date : 15 June 2018
Revised date : 3 February 2019
Accepted date : 20 March 2019

Please cite this article as: J. Navaridas, J.A. Pascual, A. Erickson et al., INRFlow: An interconnection networks research flow-level simulation framework, *Journal of Parallel and Distributed Computing* (2019), https://doi.org/10.1016/j.jpdc.2019.03.013

***Highlights (f**

- We present our flow-level simulation framework INRFlow
- It is a mature, flexible and efficient tool for simulating large scale systems
- It models network, storage, scheduler and applications
- It has been used extensively for our research in the past
- INRFlow is open source and programmed in C

INRFlow: An Interconnection Networks Research Flow-level Simulation Framework

Javier Navaridas[a,*], Jose A. Pascual[b], Alejandro Erickson[c], Iain A. Stewart[d], Mikel Luján[a]

[a] *Computer Science, University of Manchester, United Kingdom*
[b] *Computer Science, The University of the Basque Country, Spain*
[c] *Computer Science, University of Victoria, Canada*
[d] *Computer Science, University of Durham, United Kingdom*

Abstract

This paper presents INRFlow, a lightweight, frugal, flow-level simulation framework for modelling large-scale networks and computing systems. INRFlow is designed to carry out performance-related studies of interconnection networks for both high performance computing systems and datacentres. It features a completely modular design in which adding new topologies, routings or traffic models requires minimum effort. Moreover, INRFlow includes two different simulation engines: a static engine that is able to scale to tens of millions of nodes and a dynamic one that captures temporal and causal relationships to provide more realistic simulations. We will describe the main aspects of the simulator, including system models, traffic models and the large variety of topologies and routings implemented so far. We conclude the paper with a case study that analyses the scalability of several typical topologies. INRFlow has been used to conduct a variety of studies including evaluation of novel topologies and routings (both in the context of graph theory and optimization), analysis of storage and bandwidth alloca-

*javier.navaridas@manchester.ac.uk

tion strategies and understanding of interferences between application and storage traffic.

Keywords: Simulation and Modelling, Interconnection networks, Large-scale systems, Supercomputers, Datacentres, Network topologies and routing

PACS: Computer modelling and simulation, 07.05.Tp, Computer science and technology, 89.20.Ff

2000 MSC: 68U20: Simulation, 68M10: Network design and communication, 68M20: Performance evaluation; queueing; scheduling

1. Introduction

For most activities our society has come to depend on information and computer systems as a way to improve productivity and so, competitiveness. This has both driven forward the development of a plethora of IT technologies and motivated the construction of increasingly larger computing facilities. For instance, in the context of business-centric computing systems, companies require increasingly higher computing power to support operations such as mining data from service records, offering on-line services and supporting increasingly large amounts of data. If we look at the world's largest companies it is speculated that they have hundreds of thousands of servers to sustain their infrastructure. As examples of well-known companies, Google may have around one million servers scattered among 13 datacentres worldwide whereas Amazon may have roughly half a million servers in 7 datacentres around the world.

An analogous trend occurs in the scientific community, where we can see systems of similar sizes and an always increasing greed for more computing

power. In this context, the typical application is computer simulation of various natures (molecular dynamics, finite elements or weather modelling, to cite a few) which are carried out using increasingly finer-grain models which are expected to be more and more accurate but also to require of higher and higher computing power. Recently, the introduction of new technologies for data analytics has opened a new form of exploitation of scientific computing sites by allowing to analyse data collected from empirical experimentation. The highest exponent of data analytics in science is the Large Hadron Collider at CERN, which generates data at a stunning rate of 50 Petabytes per year. In order to be able to analyse all the generated data a Grid-like system with over 150 computer centres all over the world is used (See Worldwide LHC Computing Grid website[1]). This data generation rate will be dwarfed by the Square Kilometer Array project which is expected to generate a mind-blowing amount of data exceeding the Exabyte per day once it is built by 2024 (See Square Kilometre Array website[2]). At any rate, the advent of data analytics within the scientific community has motivated the convergence of datacentre and HPC architectures. Consequently, simulation tools that can cope with both models are becoming increasingly important. As we will explain in this paper, INRFlow offers capabilities for both data-centric and computation-centric systems and covers the gap between simulators specially designed for only one of these models.

The interconnection network (IN, in short), a specific-purpose network that allows compute nodes to interchange messages with high throughput and low latency, is a key element of these large-scale computing platforms

[1] Available at: `http://wlcg.web.cern.ch/`

[2] Available at `https://www.skatelescope.org/`

because its performance has a definite impact on the overall execution time of parallel applications, especially for those that are fine-grained and communication intensive. Indeed, they have been widely acknowledged (e.g., [1, 2, 3, 4]) to be one of the limiting factors when it comes to scaling up computing systems, essentially because the communication and synchronisation penalties suffered by applications increase with the size of the system. Current trends show the number of nodes used in data centre networks or supercomputers can be hundreds of thousands [5, 6, 7, 8] and these numbers are expected to increase over the millions in the next decade [3]. This is the reason why we should not decide lightly about the network that interconnects compute nodes in extreme scale computing sites. The evaluation of an IN is a complex task that requires, among other concerns, deep knowledge about how parallel applications make use of the network. Our interest revolves exactly around this topic: the modelling, simulation and evaluation of large-scale computing systems with special emphasis on the IN. This paper presents INRFlow, an Interconnection Networks Research Flow-level simulation framework we have been developing since 2014 to support our experimental work.

The main features of INRFlow are its flexibility, its low resource consumption and the modularity of its design. As will be seen throughout the paper, INRFlow can be used to simulate a plethora of topologies and traffic models each with different degrees of fidelity. We believe this flexibility may tip the scale in favour of INRFlow when it comes to select a simulation tool. Furthermore, the requirements to build and use INRFlow, in terms of memory and CPU speed are frugal, and therefore it may be the environment of choice for quick deployment and fast obtaining of results. It provides the capability to simulate, on a desktop computer, systems composed by mil-

lions of nodes in reasonable time. The most extreme configurations modelled used the static engine with over 1M servers and the dynamic engine with over 64K servers, see Section 4. The main limiting factor is normally the amount of RAM, as simulations complete quite fast (typically, hours). The amount of required memory varies depending on the characteristics of the simulation (number of endpoints, switches, complexity of traffic, etc).

INRFlow is coded in C and can be built with any compliant compiler in both POSIX and Microsoft Windows environments. Most simulation parameters are given at execution-time, so that only a few decisions have to be taken at compilation time, which, in turn, greatly simplifies compilation. It currently runs single-threaded, as runtime is acceptable for our needs (a few hours for tens of thousands of nodes in dynamic mode), but extending it to perform parallel execution should be relatively simple. The source code of INRFlow (released under GPL) together with the required information for its operation (user manual) can be found at Gitlab [3].

INRFlow has been the backbone of a large part of our recent research: In [9] we developed a novel routing for recursively-defined server-centric networks DCell and FiConn. This was later extended for the HCN/BCN networks [10]. In both cases, significantly improved practical routing algorithms were obtained. In [11] we provided a minimal-path routing for DPillar. In [12] we established the *stellar* dual-port server-centric design methodology. *Any* graph can be chosen as the base and judicious choices result in networks with beneficial properties. Using generalized hypercubes as the base graphs, we constructed GQ* and compared it with the state-of-the-art FiConn and DPillar. In [13], we proposed a multi-objective optimization framework to

[3] https://gitlab.com/ExaNeSt/inrflow

automatise the selection of topologies for a large-scale, exascalable computing system, ExaNeSt [14]. In [15], we analyse the effect of data-storage policies on the interferences between storage and applications traffic and, in turn, its effect on system performance.

To highlight the capabilities of the simulator, we conclude the paper with an example case study where we analyse the scalability of a number of state-of-the-art topologies for HPC systems and the effects of multipath routing on some of them.

2. Related Work

The networking community has developed a large variety of network simulation tools with different approaches and objectives. Let us review a small selection, pinpointing the main differences with INRFlow. Note that, as explained before, the main characteristics of INRFlow are its flexibility and its low resources requirement, and therefore it outperforms in these two aspects to most of the tools revised here.

INSEE [16] is a cycle driven flexible, lightweight functional simulator which is also being developed and maintained by our group. INSEE models router functionality in detail and provides a more accurate alternative for simulating mid- to large-scale networks than INRFlow. INSEE is able to simulate a wide variety of router models and topologies and shares part of the code base with INRFlow.

HPC-NetSim [17] is a simulator developed to model the Tianhe-2 supercomputer, which occupies the second place in the Nov'17 Top500 list. They provide an accurate cycle-driven simulation and show the precision of their framework by comparing with the real system. However, their evaluation is

restricted to a 32-endpoint system, so its scalability is difficult to assess.

TOPAZ [18], developed at the University of Cantabria, is a cycle-accurate simulator for supercomputer INs with detailed models of the components which allows obtaining very accurate performance measurements. It has the ability to interface with GEMS5 to perform full-system simulation. TOPAZ is implemented in C++ and offers parallel execution to speed up execution.

BigNetSim [19], developed at the University of Illinois at Urbana Champaign, is a trace-driven parallel discrete event simulator. It simulates, with reasonable detail, an integrated model of computation (processors) and communication (network). The simulator allows different levels of detail to evaluate the IN: from simple latency models to detailed models of the network including k-ary n-cubes and k-ary n-trees. One of the main advantages of this system is its extreme modularity, with easy mechanisms to model new topologies and routing algorithms. BigNetSim has a parallel implementation that allows carrying out large simulations of current and future systems, and to study the behaviour of applications developed for those systems. In contrast with INRFlow, in which system configuration is given as parameters at execution-time, BigNetSim is configured at compilation time, in such a way that any change in the models require to recompile the target modules.

MARS [20] is a simulator of parallel systems developed at IBM and based on the OMNeT++ simulation framework. Its design is oriented to the evaluation of parallel systems and parallel applications, and to that purpose it includes detailed models of both the communication side and the compute nodes. MARS allows us to use several multistage topologies, and a variety of switching and routing functions. In addition, it supports multicore configurations in which each processing core has its own MPI stack.

The main strength of MARS is its conformity to MPI semantics and their ability to run in parallel. However, its scalability seems to be limited to a few thousand endpoints.

MINSimulate [21], developed at the Technical University of Berlin, is a simulator designed to evaluate multistage INs. It implements Clos and Delta networks and supports both wormhole and store and forward switching. Note that currently INRFlow does not support this kind of networks but their inclusion would require insignificant effort as it would simply require implementing connection and routing functions.

The NS-2 simulator [22], from the University of Southern California, is designed to research on wired and wireless TCP-based communication networks. Although high performance computing systems used to rely on high performance interconnects such as InfiniBand or proprietary interconnects for parallel computing workloads, Ethernet is getting a significant share of the Top500 list, as new versions of 10G Ethernet or 100G Ethernet are leveraged as low-cost INs. However, TCP-based networks are not a good alternative for HPC, as most of them are relegated to the lower positions of that list.

The COTSon Infrastructure for system-level simulation by HP Labs [23] provides a full-system simulation environment based on AMD's SimNow [4]. The tool was open sourced in January 2010 and is able to simulate clusters of many-core processing systems using a functional simulator of a network switch. However, this way of modelling the network is extremely simplistic and incapable of modelling the complexity of traffic interaction within a

[4] manual available at: `http://developer.amd.com/wordpress/media/2013/03/SimNowUsersManual.pdf`

full-fledged network.

Dimemas [24], developed and maintained at the Barcelona Supercomputing Center, was designed with the evaluation of applications behaviour in mind. It can reconstruct the execution of a parallel application in any supported architecture using a trace of that application. Dimemas models computing elements with accuracy but models the INs in a rather simplistic way: a collection of buses. The workloads used with Dimemas are modelled in detail, with lots of significant states available for each application thread. A drawback of this workload's complexity is that obtaining traces with sufficient level of detail requires an instrumented kernel. Dimemas is designed to search for bottlenecks and/or unbalances that may harm the performance of parallel applications, however the basic network model cannot identify bottlenecks occurring at the network-level.

3. INRFlow design

INRFlow is a flexible, lightweight flow-level simulator focused on modelling large interconnects. To extend scalability it models the network at a link level without a detailed model of router architecture. In addition to the interconnect, INRFlow models several subsystems of a supercomputer or datacenter such as the scheduling of applications, the allocation of resources and the mapping of tasks and data sources to computing nodes. It is also able to simulate both the novel storage network of ExaNeSt in which the storage devices are attached to computing elements and more classical approaches that rely on a Storage Area Network (SAN). Extending INRFlow is very easy because all the subsystems are independent.

3.1. Simulation Engines

INRFlow works at flow-level. This high level of abstraction was decided in order to be able to scale to the system sizes we are targeting. Note, however, that raising abstraction also reduces the accuracy of the results because the particularities of the components and fine-grain interactions between them are not covered by the model. For us, a workload is a set of pairs of source and destination nodes. INRFlow constructs the network topology and workload at runtime, routes the flows specified by the workload, using the specified routing algorithm, and, finally, reports statistics.

INRFlow implements two different simulation engines: static and dynamic. In static mode, flows are routed simultaneously and a link's capacity is assumed to be shared among all the flows routed through it. Static mode can handle very large networks and serves to report on raw performance metrics where the causal relationships between flows are not important, such as the mean hop-length of a routing algorithm or preliminary estimations of the throughput.

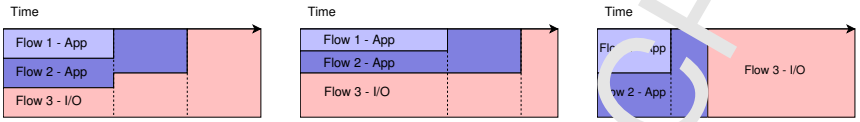
While the static engine works similarly to most simulators used by the datacentre networks community [25, 26, 27, 28], we argue that static analysis does not always accurately reflect network performance due to its lack of temporal and causal modelling. For this reason, INRFlow features a dynamic engine that is able to deal with temporal and causal aspects of the execution. In dynamic mode, the links of the network have capacities and each flow is specified with a weight reflecting the data that must be routed. In addition, the workloads prescribe computing phases and causal relationships among flows, so that some flows must finish before others begin. Dynamic mode provides a more realistic, flow-level simulation of general real-world workloads, as well as a good estimation of the completion times

of a collection of application-inspired workloads.

INRFlow has also the ability of incorporating failures into the description of the system. This is useful to ascertain the fault tolerance of a system. When failures are present, INRFlow randomly disconnects a number of network links that is given as the percentage of the total number of link or as an absolute number of failures. This can be used to measure the connectivity of the network after the failures occur counting the number of flows that the network is able to deliver to their destination, see, e.g., [12]. This is particularly useful to evaluate routing algorithms and the effectiveness of multipath policies. To employ this mode, fault-tolerant routing functions would be needed, but INRFlow will keep operating by simply dropping flows that can not be routed, if such a facility is not implemented.

INRFlow dynamic engine also supports using flow priorities at the link level. Currently, we consider two priority levels, App (high) and I/O (low) and have implemented the following policies (depicted in Figure 1) and analysed in detail in [15].

- **No priorities (NP):** This is the baseline policy in which all the flows have the same priority. As we can see in Figure 1a, link bandwidth is shared fairly among the flows.
- **Bandwidth Apportion (BA):** This policy establishes the proportion of the link bandwidth that will be used for each type of traffic. The number of flow types can vary as shown in Figure 1b where we have assigned 50% of bandwidth to inter-process traffic and 50% for storage traffic. In this case, half of the bandwidth will be shared by App flows and the rest by I/O flows. When only flows of one kind are present, they occupy the full bandwidth.



(a) No Priority (b) BW Apportion 50-50 (c) Full Priority

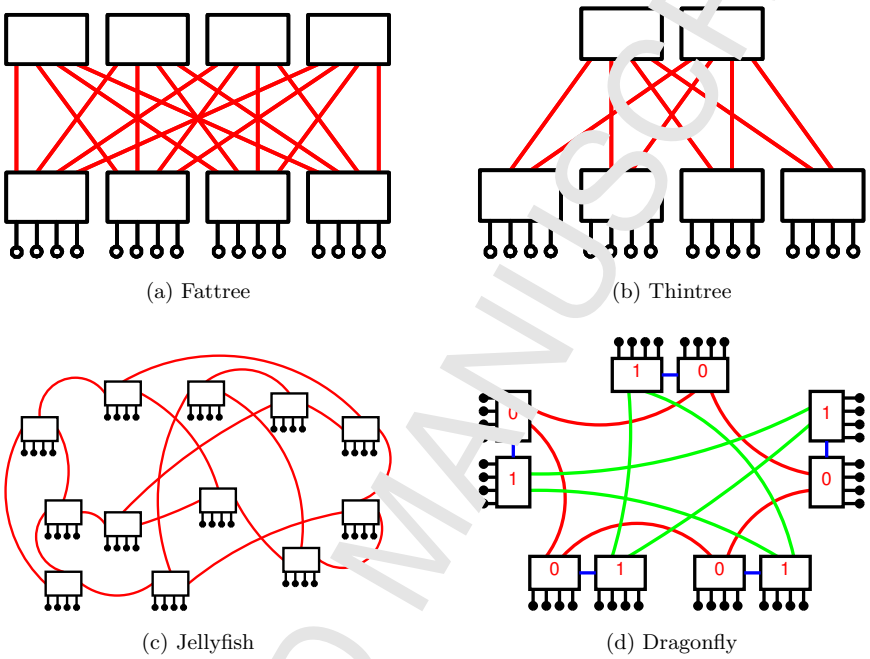
Figure 1: Examples of traffic prioritization policies. Time flows left-to-right and bandwidth is represented vertically.

- **Full Priority (FP):** App traffic has full priority over I/O traffic. Thus I/O traffic will only use the network resources that are not employed by applications. An example of this policy is depicted in Figure 1c where all the bandwidth is used to transmit application flows. When these finish, storage flows are transmitted.

3.2. Topologies

INRFlow implements a large variety of network topologies, see Fig. 2 for a few examples, both from the datacentre and the HPC communities. In particular it is able to simulate server-centric networks in which nodes and switches have the role of routing elements, and switch-centric networks in which the nodes do not perform any kind of routing. Examples of some of the datacenter topologies implemented in INRFlow are:

- **DPillar**: It is a server-centric data center network somewhat inspired by the classic butterfly topology [29]. DPillar provides several nice properties such as scalability, network performance, and cost efficiency, which make it suitable for building large scale future data centers.
- **Bcube**: Another server-centric network architecture, specifically designed for shipping-container based modular datacenters [28]. BCube



(a) Fattree (b) Thintree

(c) Jellyfish (d) Dragonfly

Figure 2: Examples of some of the topologies supported by INRFlow.

has a nice property: graceful performance degradation as the server and/or switch failure rate increases.

- **Gdcfconn**: This is the generalisation of DCell [30] and Ficonn [31], a family of recursive topologies which eliminates the requirement of any switches other than the lowest-level commodity ones. It is highly scalable to encompass hundreds of thousands of servers, while at the same time keeping low diameter.

- **HCN/BCN**: is a recursively-defined family of networks, where the BCN construct is built using (copies of) HCNs by including an additional layer of interconnecting links [32].

- **Jellyfish**: This is a random network designed to provide high connectivity in datacenters. One of its main characteristics is that it is incrementally expandable as opposed to most common topologies [33].

INRFlow also includes more HPC-focused networks such as the following:

- The **torus** is a well-known topology that has been historically used to interconnect massively parallel processors. Nodes in a torus are arranged in a d-dimensional grid with wrap-around links.

- Many **Tree-like** topologies such as the k-ary n-tree topology (fat-tree): [34], the k:k'-ary n-tree topology (thintree) or the generalized tree topology (gtree).

- **Dragonfly** is a large-radix, low-diameter, recursive family of topologies that uses a group of high-radix routers (a network group) as a virtual router to increase the effective radix of the network. Then interconnects large numbers of these network groups in an all-to-all fashion using a given connection rule [35]. Currently, we have 5 different connection rules implemented.

- General **Graph-based** topology: Sometimes we do not have the mathematical description of a topology (connection rules). For this reason we have implemented a loader that is able to load the definition of the topology from a file. This file contains the number of nodes and switches and how they are interconnected. This was developed for our topological optimization framework [13].

The definition of a topology in INRFlow normally involves implementing routing algorithms that can be used with it. There are many algorithms

already implemented and it is very easy to add new ones. INRFlow supports both single-path, where given a source and a destination node the same path is used all the time, and multi-path, where many parallel paths can be used. When INRFlow needs to perform the routing of a flow it will use all the paths provided by the routing algorithm. The design of INRFlow is very flexible allowing the implementation of different kinds of routing policies. In particular, we can implement arithmetic routings that calculate the path when they are requested or pre-computed routings in which all the path between pairs of nodes are calculated before the simulation starts. However, implementing a new routing algorithm is not required for every topology since we have implemented generic routing algorithms that can work with arbitrary topologies:

- Breadth-first search (**BFS**) routing: This is single-path routing policy that looks for one of the possible shortest paths between each pair of nodes.
- Equal Cost Multiple Paths (**ECMP**) routing: This is a multi-path routing algorithm that balances loads among all shortest paths between each pair of nodes.
- K-Shortest Path (**KSP**) routing: This is a multi-path routing algorithm that looks for the K (arbitrary) shortest paths between each pair of nodes.
- AllPath-d (**AP**) routing: This is a multi-path routing algorithm that balances load over paths which are equal of shorter than the shortest path plus an arbitrary parameter, d. It aims to increase path diversity.

These are very useful when we want to evaluate new topologies or routing

functions, since they can be used as baseline to compare with. However, they involve a relatively high cost in terms of computing time and memory consumption, so for consolidated topologies it is advisable to provide specific routing algorithms.

3.3. Workload Generator

In INRFlow, nodes are modelled in a rather simplistic way: a traffic generator/consumer. However they can use a large variety of types of traffic generators. From purely synthetic traffic patterns to traces extracted from real applications as well as reusable traffic generators developed from analysing real traces from applications.

3.3.1. Synthetic Traffic Patterns

INRFlow provides a broad range of synthetic traffic patterns that can be used to measure the performance of the communication infrastructure and only consider spatial distribution of traffic. The following are some of the traffic types that can be generated by INRFlow:

- **Random**. When a packet is generated at a node (the source), the destination is randomly selected following a given probability distribution. The built in modes are *uniform*, in which all the nodes have the same probability of being selected as destination, and the non-uniform *hot spot* and *hot region*, where a given node or group of nodes, respectively, have higher probability of being selected as destination, increasing the risk of generating congestion in some regions of the network. Finally, with *local* traffic, the probability of selecting destination nodes decreases with the distance (so that most packets are sent to nearby nodes).

- **Permutations**: Given a source node, the destination node is always the same, and is computed as a permutation of the source node identifier (generally a bit permutations). INRFlow supports classical permutations such as Perfect Shuffle, Bit Reversal, Bit Transpose and Bit Complement.

- **Bisection**: The network is split uniformly at random into two halves and every server in each half sends a flow to every server in the other half.

- **All-to-one**: A unique root server is chosen, uniformly at random, and every server sends a flow to the root.

- **All-to-all**: every server sends a flow to every other server.

- **Many-all-to-all**: For a given size s, the network is partitioned uniformly at random into $g = N/s$ groups of servers, each of size at most s. Each server sends a flow to all other servers in its group.

3.3.2. Real Application Traces

Synthetic traffic sources provide very useful insights into a network's potential. However, obtained performance metrics can be unrealistic as applications use more sophisticated communication patterns than synthetic models. For this reason INRFlow can also use traces from applications to perform trace driven simulation. To reproduce the causal relationships between events in the trace files, INRFlow requires a special data structure to store past and future events, shown in Fig. 3. Each node of the simulated applications has an event queue, which is fed from the trace file. A packet is sent through the network when an **S** (send) event is in the queue's head.

If an **R** (receive) event is in the head, it is necessary to access the pending notifications queue to check if the expected event has happened already; otherwise, processing of events is blocked until the network notifies the awaited reception. The pending notifications queue at each node, thus, stores reception events that arrive before the application requests them, and it is a crucial element to keep event causality. The complete process of trace-driven simulation is akin to the one we presented in [26] and works as follows:

1. Enqueue in each node's event queue all the events it has to execute.
2. Initialize the pending notifications list as an empty list. Nodes sequentially execute the events in their event queue.
3. If the first event is a *send*, remove the event and inject the corresponding message into the network.
4. If it is a *reception*, check if a corresponding message (matching origin, destination, tag and size) is in the pending notification list. If it is there, remove both entries. Otherwise, keep in this state until the required message is received by the node and is accordingly found in such list.
5. If it is a *computation* event, put the node on hold for the required period of time, using a selected CPU-scale factor.
6. When the network delivers a message, put it in the pending notifications list.

An example of this procedure is depicted in Fig. 3. In the figure, nodes 0 and 15 are waiting for a message from node 1. Node 0 has received a message from node 15 whereas node 15 has received a message from node 2. These are stored in the corresponding notification lists. This mechanism reproduces the actual way messages were interleaved when running the application,

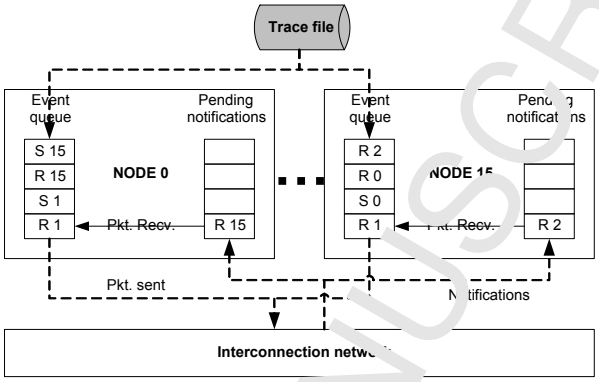


Figure 3: Diagram of the data structures needed to support trace-driven simulation.

complying with the causal order between a reception and the subsequent sends it may trigger, see [36].

3.3.3. Pseudo-Applications Traffic

INRFlow is also able to generate many application-inspired workloads. These workloads cover some representative network traffic scenarios that can be found in existing datacenters and HPC systems and use the same data structures as the trace-based simulations. A non-exhaustive list of these workloads includes:

- **Scientific applications**: Inside this group we include models that mimic scientific code traditionally used by the HPC community. In particular, 2D and 3D *stencil* and *sweep* codes, i.e., applications that communicate following 2D and 3D grid patterns (similar to what we found in many scientific codes). In addition, we also incorporated a *n-body* application, code used to solve the n-body problem that involves the prediction of individual motions of a group of objects interacting with each other.

- **Datacentre applications**: This group of applications includes models that mimic the traffic that appears in datacentres, including the popular Mapreduce in which after a phase of scatter data, the tasks of the application communicate using an alltoall traffic pattern and finishing with a gather phase. We also emulate unstructured applications such as graph analytics using causal random traffic and a model that we call dcntraffic in which all the nodes of the network communicate with each other using an 80% of short flows and a 20% of long flows as reported in [37]. This latest model emulates, not only the traffic of the applications, but also the management traffic present in datacentres.

- **Benchmarking patterns**: This group contains causality-enhanced versions of the traffic patterns traditionally used in the evaluation of network topologies (similar to the ones in Section 3.3.1). To enforce causality, flows are generated into phases. Each phase has a fixed number of flows and requires all the flows from the previous phases to be delivered before beginning. The smaller the phase size, the more tightly-coupled the application, i.e., the higher the causality.

3.3.4. Markov chain-based Application Model

Given the wide variety of applications that we need to consider (HPC from several scientific domains, big data analytics for scientific, engineering and commercial purposes, and business-intelligence applications) and their different needs in terms of communication and storage, INRFlow also supports a generic model based on Markov chains which can be fine-tuned to model different flavours of application by changing the transition probabilities between the different states. Fig. 4 shows the model we constructed based on an analysis of a number of applications we had access to within

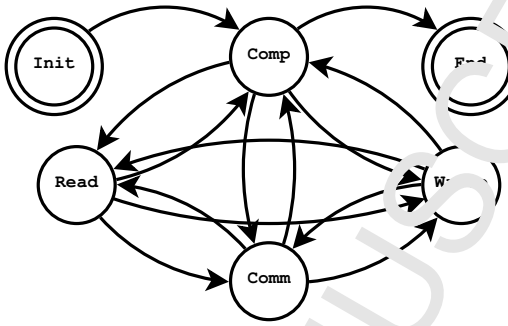


Figure 4: Representation of the Markov chain used to generate synthetic applications and parameters used to generate traffic in our experiments.

our projects. An interested reader can look at some utilization examples of how this application model in [15]. This model is composed of 6 states each of them representing the different types of operations that can go on during the execution of an application. Storage is split into two states to be able to model applications with varying I/O needs (e.g. read- or write-intensive, or more balanced access to storage).

- **Init**: This state represents the moment in which an application gets scheduled into the system and the required resources (i.e. processing nodes) are assigned to it, including all source data preparation (caching).
- **End**: When this state is reached the application will finalise. Transitioning to this state will free all the computing resources of the application and will also trigger updating the data origins with the results of the application.
- **Comp**: A computation-intensive phase without any data moving. It is the first one after Init and the last one before End to model the

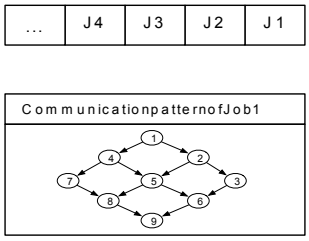
creation/destruction of application's data structure

- **Comm**: A communication-intensive phase in which computing nodes will communicate and/or synchronise with each other. This phase can model different patterns, in particular any of the ones discussed above.
- **Read**: During this phase, the processing nodes will read data from storage according to the storage policies implemented in INRFLow.
- **Write**: During this phase the application writes data to storage, covering for storage of execution results, updating of data, snapshots of the application status or check-pointing of the application.

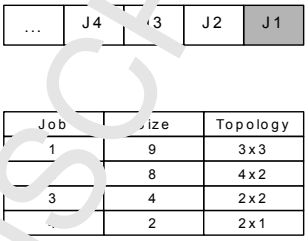
The transition between phases is performed using a probabilities transition matrix. The value of each element of the matrix, $M[a][b]$, indicates the probability of a transition from phases a to phase b. Therefore, the sum of each row and column has to be 1. These probabilities are fully configurable and allow to emulate several types of applications such as I/O-intensive, computation-intensive, communication-intensive or mixes of them. Additionally, there are many other parameters that can be configured, such as the application size, the communication pattern during the Comm phase, the lambda parameter for the duration (exponential) of the Comp phase, or the storage servers and transfer sizes for storage traffic.

3.4. Scheduling Model

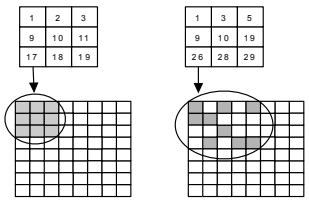
In large-scale multi-tenant systems, applications need to follow several steps upon submission before they are actually executed. The piece of software in charge of that is called the *scheduler* and performs the following stages: Job Selection, Resource Allocation and Task Mapping (see Fig. 5).



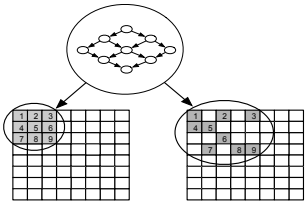
(a) List of Jobs in the queue and communication pattern of Job 1.



(b) Job selection. Job 1 is selected among all the jobs in the queue.



(c) Resource allocation. Contiguous (left) vs non-contiguous (right).



(d) Task mapping. Each task of the job is allocated to a specific server.

Figure 5: Examples of the different stages of the scheduling process.

Because most large-scale computing systems are reliant on this kind of tools, we have implemented a model of the scheduler in INRflow to be used in our research [15].

First, the *Job Selection* stage in which the next application to be executed is selected. At the moment the following policies are implemented:

The simplest and most common policy is the First-Come First-Serve policy (FCFS) [38], which imposes a strict order in the execution of jobs. These are arranged by their arrival time and order violations are not allowed. The main drawback of this policy is that it severely

reduces system utilization. When the job at the head of the queue cannot be put to run because the required resources are not available, all the jobs in the queue must wait due to the sequentially ordered execution of jobs. As a result, many processors could remain idle, even when other waiting jobs could be eligible to use them.

- Aggressive Backfilling (BF) [38] tries to overcome this drawback of FCFS by allowing the head job to be overtaken and allowing to schedule other application(s) which fit in the available resources. BF is a variant of FCFS, based on the idea of advancing jobs through the queue. If the job at the head cannot be launched due to resource constraints, a reservation time for it is calculated using the estimated termination time of currently running jobs. Using this policy, the system utilization is improved because more jobs can be put to run without delaying the expected starting time of other jobs. Note that there exists an alternative called conservative Backfilling in which all the jobs at the queue receive a reservation but it is too strict and is barely used in practice, so it is not yet implemented in INRFlow.

- Shortest Job First (SJF) [38] selects the jobs in order, with the shortest (in terms of estimated execution time) being executed first. The idea behind this policy is to avoid short jobs having to wait for much longer jobs in order to reduce the average waiting time of the jobs at the queue. The jobs are ordered using the expected value of the runtimes of each job.

Note that the use of both BF and SJF policies requires the expected runtime of the jobs being scheduled. As this depends on many variables

such as the problem to solve, the type of hardware assigned or the status of the network it is impossible to provide an accurate value. For this reason these times are, either predicted using simple models based on the history of the execution of similar jobs [39] or, more recently developed models based on machine learning techniques [40]. However, in real production systems these times are provided by the users when submitting the jobs [41]. This is the approach that we use in INRFlow, so the estimated runtime needs to be provided for each application or if it is not, then it will not be able to overtake previous jobs.

Once the application is selected, the *Resource allocation* stage selects the physical resources (servers) to execute it. This stage tends to be guided by applications requirements such as memory, storage capacity, processor architecture, OS, etc. However, many authors argue that exploiting application locality by placing applications in a set of nodes which maintain some form of contiguity provides a more efficient utilization by reducing network latency and interference between jobs [42, 43, 44].

Finally, the *Task mapping* stage assigns each task of the application to the allocated servers. This stage can have a high impact on the performance of the applications [45] and to be effective it should be done considering specifics of the communication patterns used, amount of data exchanged, etc. For this reason, there is a large body of research and many approaches on how to improve this stage of the scheduling process [46, 47, 48, 49, 50]. INRFlow implements two simple strategies which are valid for any network: (1) consecutive which assigns the task to the set of reserved nodes in sequential order and (2) random which assigns the tasks to the nodes randomly.

In INRFlow we have implemented each of the stages as independent modules. This way it is very easy to implement new policies for any of

the three stages. In the following pictures we have depicted a scheduling example for one application, from its arrival to the queue to their execution.

The quality of the scheduling process is typically measured using a set of specific metrics which are implemented in INRFlow:

- Waiting time: It is the time that a job spend in the queue, that is, the time since it is submitted to the system until it is selected to be executed.
- Runtime: It is the time required by a job to be executed.
- Total time: This time is the combination of the waiting time and the runtime. It represents the time since a job is submitted until it finishes.
- System Utilization: This metric represents the proportion of computing resources that is used during a time period.
- Throughput: This is the number of jobs that finish per unit time.
- Makespan: This is the total time required to process the whole sequence of jobs since the first is submitted to the queue until all of them finish.

3.5. Storage Subsystem Model

INRFlow also incorporates a full model of the storage subsystem (see Fig. 6 for a detailed diagram and [15] for a concrete use case). In the current model there is local storage attached to each computing element as well a central storage accessible via a storage network (SAN). Local storage can be cached in memory. Based on these two storage models, there are many data access mechanisms available in INRFlow:

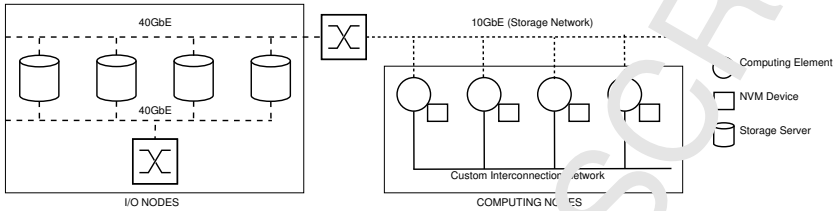


Figure 6: Visual representation of the ExaNeSt storage architecture. The local NVMs are attached to the computing nodes sharing the main IN (solid). An Ethernet network is provided for central data storage (dashed).

- Data mapped locally in memory is accessed immediately.
- Read and writes into the local storage device is limited by the PCI-e controller or the device (configurable independently).
- Access to data mapped in remote nodes is limited by the IN.
- Data in the centralized storage requires using the SAN network.

Once an application has been selected to be run, the allocator will select a set of computing nodes to place the tasks of the application. In that moment, the application will request access to the required data. We have currently three possible policies to perform storage assignment in the local-devices.

- Local: All the local storage devices are available to load the data for the application. This is the ideal scenario where all the storage traffic remains local within the computing elements. As a consequence there is no interference with other traffic.
- Internal: In this case only some of the local storage devices are available. This situation could happen if other applications have requested

some of these storage devices previously.

- External: In this case all (or part thereof) the storage devices are outside of the partition assigned to the application. Now all the accesses to the data will be remote, generating intra- and inter-applications interference.

4. Case study: Scalability Analysis of HPC Topologies

In order to showcase the capabilities of INRFlow, we have conducted a simple experiment to assess the scalability of a few different topologies, as well as their performance when dealing with realistic application models. Note that the objective of the experiment is not to perform a thorough, detailed evaluation of the networks but, simply, to give a taste of the kind of studies INRFlow can support. The experiments performed here are two-fold. First we show some results produced with INRFlow static engine using random uniform and hot-region traffic patterns as a measurement of raw performance when handling balanced and unbalanced loads. In this first analysis we look into systems with up to 1 million of endpoints and consider the following figures of merit:

- Aggregate Restricted Throughput: measures the throughput when all flows are routed at the speed of the (slowest) bottleneck flow; this simulates applications that are tightly coupled and so must wait for the completion of all flows before being able to advance.
- Aggregate Non-restricted Throughput: measures the throughput in applications that are loosely coupled, where each flow can be processed as it arrives and so no slowdown is introduced by the slowest flows.

- Overall Throughput per Port: is calculated from the two previous taking into consideration the number of switch ports (links), so to ascertain how efficiently resources are used.

Afterwards, we use INRFlow's dynamic engine to analyse instances of the topologies of around 64K-node handling some realistic workloads. In this set of experiments the figure of interest is the execution time of the applications and they focus on assessing how the above raw gains are translated into applications speed-up. We consider a broad range of application models as explained in Section 3.3.3.

4.1. Scalability Results

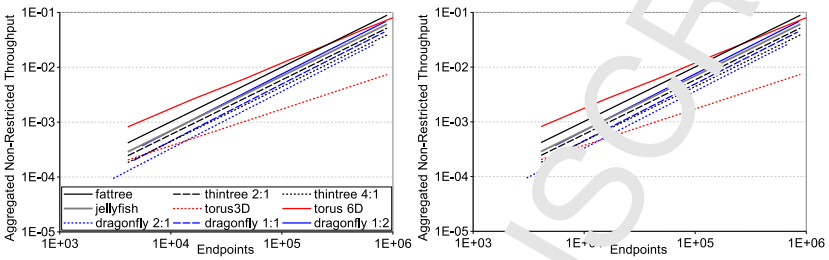
Fig. 7a and 7b show, respectively, the Non-Restricted and Restricted Throughput of the different topologies as the number of endpoints is increased. First, we can see that the 6D torus provides the highest throughput for relatively small networks. However, it does not scale as nicely as the other topologies, indeed it eventually gets outperformed by the fattree as the systems scale up and the best dragonfly (1:2) would also outperform it if we extended our experimental space a little further. It can be noted as well that all the topologies within a family of topologies have similar trends (gradient in the plots) with the tori showing the lowest slope. The trees and the dragonflies have similarly good trends, with the dragonflies having a slightly higher cost. Jellyfish requires special consideration as it behaves differently in terms of restricted and non-restricted throughputs. In the unrestricted case it follows a trend similar to the trees whereas in the restricted case, it scales poorly like the torus. This is because of the random nature of the topology generates quite a lot of bottlenecks when static routing is used. This is a known limitation of the topology, but can be alleviated by means

of multipath routing, we will explore this issue later. With regards to the Overall Throughput per Port, shown in Fig. 7c, we can see how all of the topologies, except for the tori, have a very nice, nearly flat, scalability in terms of throughput per port. While there is little difference among the topologies, it is worth highlighting that the Jellyfish is the one with the best relation between throughput and links. This motivated our study in less structured topologies, such as the optimisation framework proposed in [13].

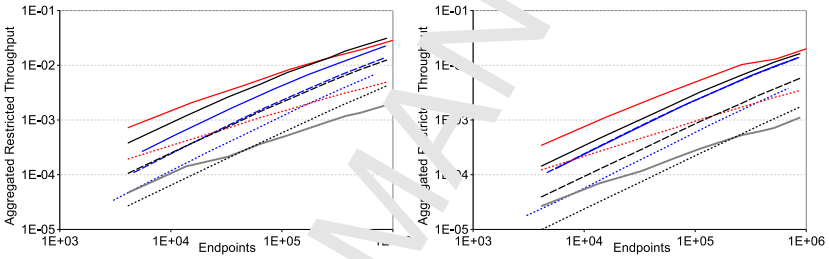
4.2. Applications Execution Time

Let us analyse now the results for the different groups of applications using the dynamic engine of INRFlow. For simplicity we assume all links run at 10Gbps, but mixed link bandwidths are supported. Fig. 8 summarizes the results with the realistic workloads. Given the wide range of execution times, the results are normalized, so to show how many times slower execution could be if an inadequate network is chosen. The first group of workloads is formed by these traditionally used in HPC environments. The Stencil workloads are executed much faster in the tori because the topology matches perfectly the communication pattern, hence there is no contention at the network level. Among the rest of the topologies the best results are achieved by the fattree and the thintree with 1:2 oversubscription ratio. As expected, the worst results were obtained using Jellyfish due to the use of an unstructured topology to execute a completely structured workload. In the case of n-body, the high causality of its communication pattern minimises the differences between topologies.

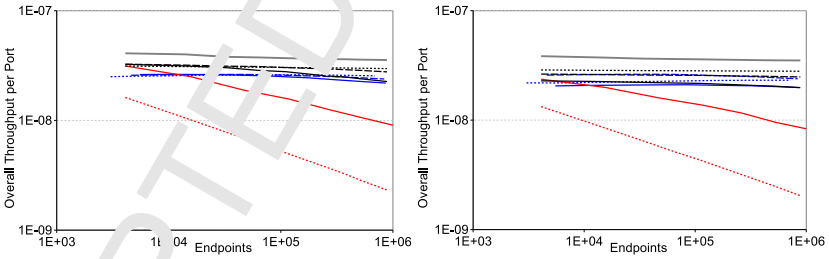
The second group of workloads mimics the behaviour of applications in datacentres. If we look at the Unstructured workload the best performance is achieved using the Torus 6D. The rest of the topologies perform similarly



(a) Aggregate Non-restricted Throughput



(b) Aggregate Restricted Throughput



(c) Overall Throughput per Port

Figure : Scalability analysis with Uniform (left) and Hotregion (right) traffic.

except the thintree with a 1:4 ratio in which the reduced bandwidth in the upper tiers severely affects the performance of the application. Special attention should be paid to the good performance achieved by the dragonflies and the jellyfish topologies due to the fact that both have a random compo-

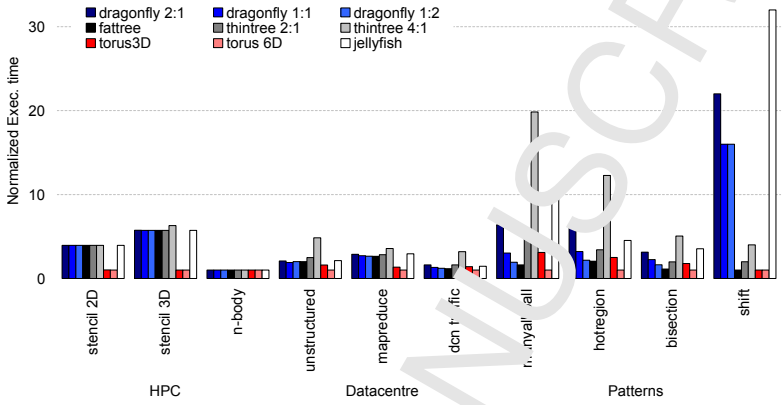
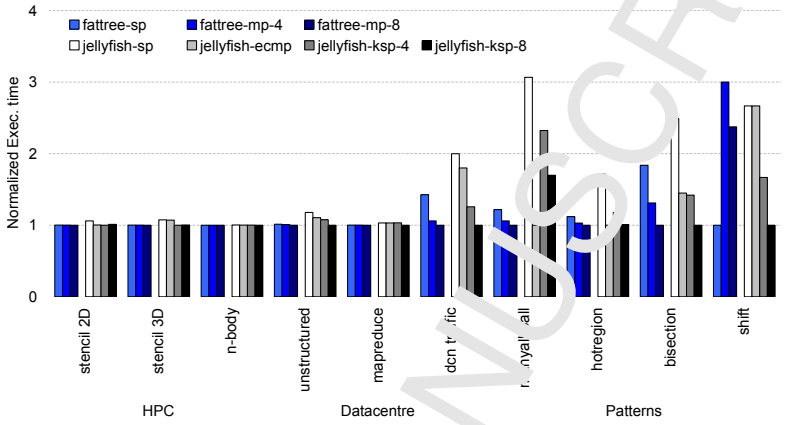


Figure 8: Execution time of the realistic workloads.

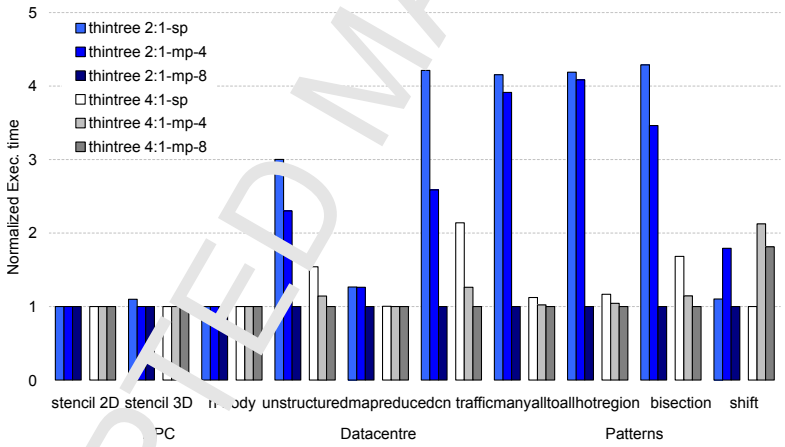
nent: valiant routing and the random construction respectively. Regarding MapReduce, the best performance is achieved using the 3D- and 6D- tori with the remaining topologies performing similarly. Finally, dcntraffic works best with the torus 6D and the fattree and worst, again, with the thintree with a 1:4 ratio.

The third group evaluates the topologies using more traditional patterns. In this case the patterns running in the torus 6D and the fattree require the lowest execution time. Let us remark that the worst topology is again, in all cases except Shift, the thintree with an oversubscription ratio of 1:4.

Overall it is worth noticing the effect that the topology may have on the execution time of applications is considerable, with up to one order of magnitude slower execution if run in an inadequate topology. In the results, we observe the great potential of torus-like and tree-like. Although high oversubscription in thintrees affects negatively the performance of the applications, these kind of networks are still strong candidates in larger networks when locality can be achieved. This will be studied thoroughly in future



(a) Fattr[illegible] Jellyfish.



(b) Fattree vs Thintree.

Figure 9: Effects of multipath routing.

wo[illegible]cks. As occurred with the static experiments above, the low performance achie[illegible]ed by Jellyfish requires extra consideration. Again, the culprit is that the si[illegible]gle path routing cannot take advantage of the high connectivity of the network. In order to assess how much of the low performance can be

attributed to that reason, we next evaluate both type of topologies, tree-like and jellyfish, using multi path routing.

Fig. 9 shows how the availability of multipath routing affects the performance of the fattree, the Jellyfish and the thintree. We can see that in the case of the fattree, the multipath schemes not only do not help greatly to improve the performance but, in fact they can actually harm it significantly for adversarial traffic patterns. This is because the large interconnection resource provided by the topology can be shared evenly with a static, single-path routing, whereas the use of multipath can actually generate some areas of contention that would not appear otherwise. With Jellyfish, on the other hand, we can see that applying multipath routing algorithms can be considerably beneficial (up to $2-3\times$ faster) with KSP being generally better than ECMP.

Note that, although the fattree cannot really benefit from multipath routing in many cases, oversubscribed trees are able to benefit from it up to a certain level. There we can see that the slightly oversubscribed thintree 2:1, can achieve speedups in the range of $2-4\times$ for many of the workloads considered here. This is because with relatively small oversubscription ratios, it is more likely to generate contention in the topology, but there is still a large variety of paths that can be exploited by the multipath algorithm to distribute the traffic more evenly across the higher levels of the topology. On the other hand, the more aggressively oversubscribed topology, thintree 4:1, extracts little benefit from the multipath scheme because the low availability of paths means that there are not many occasions in which the traffic can be spread more evenly across the great bottleneck that is the last level of the interconnect.

5. Conclusions

This paper has described exhaustively the design of the INRFlow simulation framework for large-scale networks and computing systems. INRFlow is a mature, flexible and frugal tool that has shown its capabilities in a wide range of previous research work within the area of interconnection networks for datacentres and HPC computing systems. It models many aspects of such systems, including the scheduling process, the storage subsystem, the interconnection network and the application traffic. Our description includes the large number of topologies and routings implemented already, the wide variety of traffic generators and the different subsystems included in our models.

In top of that description we complete the paper with a case study in which we investigate the scalability of typical interconnection networks with up to one million nodes. There we see that high-dimensionality torus can offer the best raw performance as well as exploit it appropriately to obtain the fastest execution times of applications. We also show some examples of topologies where multipath routing can be necessary in order to speed up the execution of applications.

Finally, we want to remark that INRFlow is an open source platform and we would like to invite all researchers in the area of interconnection networks and related ones, to try it and use it for their own purposes as well as to contribute to its design and development.

Acknowledgements

The development of INRFlow started as part of the INPUT project which was funded by the Engineering and Physical Sciences Research Council (EP-

SRC) through grants EP/K015680/1 and EP/K015699/1. Its development has continued during ExaNeSt and EuroEXA which are supported by the European Union's Horizon 2020 programme under Grant Agreements No. 671553 and 754337.

Bibliography

[1] J. Escudero-Sahuquillo, P. J. Garcia, High-performance interconnection networks in the exascale and big-data era, The Journal of Supercomputing 72 (12) (2016) 4415–4417. doi:10.1007/s11227-016-1893-6.

[2] D. Abts, B. Felderman, A guided tour through data-center networking, Queue 10 (5) (2012) 10:10–10:23. doi:10.1145/2208917.2208919.

[3] P. M. Kogge, P. L. Fratta, J. Vance, Facing the exascale energy wall, in: 2010 International Workshop on Innovative Architecture for Future Generation High Performance, 2010, pp. 51–58. doi:10.1109/IWIA.2010.9.

[4] O. Lysne, S. A. Reinemo, T. Skeie, A. G. Solheim, T. Sodring, L. P. Huse, B. D. Johnsen, Interconnection networks: Architectural challenges for utility computing data centers, Computer 41 (9) (2008) 62–69. doi:10.1109/MC.2008.391.

[5] Y. Ajima, T. Inoue, S. Hiramoto, T. Shimizu, Y. Takagi, The Tofu interconnect, Micro, IEEE 32 (1) (2012) 21 –31.

[6] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta, Vl2: A scalable and flexible data center network, SIGCOMM Comput. Commun. Rev. 39 (4) (2009) 51–62. doi:10.1145/1594977.1592576.

[7] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, S. Lu, Dcell: A scalable and fault-tolerant network structure for data centers, in: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08, ACM, New York, NY, USA, 2008, pp. 75–86. doi:10.1145/1402958.1402968.

[8] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, J. J. Parker, The IBM Blue Gene/Q interconnection network and message unit, in: Procs. of 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis, ACM, New York, NY, USA, 2011, pp. 1–10.

[9] A. Erickson, A. Kiasari, J. Navaridas, I. A. Stewart, Routing algorithms for recursively defined data centre networks, in: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 3, 2015, pp. 84–91. doi:10.1109/Trustcom.2015.616.

[10] A. Erickson, I. A. Stewart, J. A. Pascual, J. Navaridas, Improved routing algorithms in the dual-port datacenter networks HCN and BCN, Future Generation Comp. Syst. 75 (2017) 58–71. doi:10.1016/j.future.2017.05.004.

[11] A. Erickson, A. E. Kiasari, J. Navaridas, I. A. Stewart, An optimal single-path routing algorithm in the datacenter network dpillar, IEEE Trans. Parallel Distrib. Syst. 28 (3) (2017) 689–703. doi:10.1109/TPDS.2016.2591011.

[12] A. Erickson, I. A. Stewart, J. Navaridas, A. E. Kiasari, The stellar transformation: From interconnection networks to datacenter networks,

Computer Networks 113 (2017) 29–45. doi:10.1016/j.comnet.2016.12.001.

[13] J. A. Pascual, J. Lant, A. Attwood, C. Concatto, J. Navaridas, M. Luján, J. Goodacre, Designing an exascale interconnect using multi-objective optimization, in: 2017 IEEE Congress on Evolutionary Computation (CEC), 2017, pp. 2209–2216. doi:10.1109/CEC.2017.7969572.

[14] R. Ammendola, A. Biagioni, P. Cretaro, O. Frezza, F. L. Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, F. Simula, P. Vicini, G. Taffoni, J. A. Pascual, J. Navaridas, M. Lujan, J. Goodacre, N. Chrysos, M. Katevenis, The Next Generation of Exascale-Class Systems: The ExaNeSt Project, Vol. 00, IEEE Computer Society, Los Alamitos, CA, USA, 2017, pp. 510–515.

[15] J. Pascual Saiz, J. Lant, C. Concatto, A. Attwood, J. Navaridas, M. Luján, A. Goodacre, On the effects of allocation strategies for exascale computing systems with distributed storage and unified interconnects, Concurrency and Computation: Practice and Experiencedoi:10.1002/cpe.4784.

[16] J. Navaridas, J. Miguel-Alonso, J. A. Pascual, F. J. Ridruejo, Simulating and evaluating interconnection networks with insee, Simulation Modelling Practice and Theory 19 (1) (2011) 494 – 515.

[17] W. Zhou, J. Chen, C. Cui, Q. Wang, D. Dong, Y. Tang, Detailed and clock-driven simulation for hpc interconnection network, Frontiers of Computer Science 10 (5) (2016) 797–811. doi:10.1007/s11704-016-5035-3.

[18] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente, J. A. Gregorio, Topaz: An open-source interconnection network simulator for chip multiprocessors and supercomputers, in: 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip, 2012, pp. 99–106. doi:10.1109/NOCS.2012.19.

[19] G. Zheng, T. Wilmarth, P. Jagadishprasad, L. V. Kalé, Simulation-based performance prediction for large parallel machines, International Journal of Parallel Programming 33 (2) (2005) 183–207. doi:10.1007/s10766-005-3582-6.

[20] W. E. Denzel, J. Li, P. Walker, Y. Jin, A framework for end-to-end simulation of high-performance computing systems, SIMULATION 86 (5-6) (2010) 331–350. doi:10.1177/0037549709340840.

[21] D. Tutsch, Performance analysis of network architectures, Springer Science & Business Media, 2007.

[22] N. Kubinidze, I. Gunchev, M. O'Droma, Network simulator ns2: Shortcomings, potential development and enhancement strategies, in: A. Nejat Ince, E. Topuz (Eds.), Modeling and Simulation Tools for Emerging Telecommunication Networks, Springer US, Boston, MA, 2006, pp. 263–277.

[23] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, D. Ortega, Cotson: Infrastructure for full system simulation, SIGOPS Oper. Syst. Rev. 43 (1) (2009) 52–61. doi:10.1145/1496909.1496921.

[24] F. M. Badia, J. Labarta, J. Gimenez, F. Escale, Dimemas: Predicting

mpi applications behavior in grid environments, in: 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip, 2012.

[25] Y. Liu, D. Lin, J. Muppala, M. Hamdi, A study of fault-tolerance characteristics of data center networks, in: IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012), 2012, pp. 1–6. doi:10.1109/DSNW.2012.6264676.

[26] Y. Liu, J. K. Muppala, M. Veeraraghavan, D. Lin, M. Hamdi, Data Center Networks - Topologies, Architectures and Fault-Tolerance Characteristics., Springer Briefs in Computer Science, Springer, 2013.
URL http://dx.doi.org/10.1007/978-3-319-01949-9

[27] D. Guo, C. Li, J. Wu, Y. Zhou, Dcube: A family of network structures for containerized data centers using dual-port servers, Computer Communications 53 (2014) 13 – 25. doi:https://doi.org/10.1016/j.comcom.2014.07.003.
URL http://www.sciencedirect.com/science/article/pii/S0140366414002461

[28] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, S. Lu, Bcube: A high performance, server-centric network architecture for modular data centers, in: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication, SIGCOMM '09, ACM, New York, NY, USA, 2009, pp. 63–74. doi:10.1145/1592568.1592577.

[29] Y. Liao, J. Yin, D. Yin, L. Gao, DPillar: Dual-port server interconnection network for large scale data centers, Comp. Networks 56 (8) (2012) 2132–2147.

[30] C. Guo, W. Haitao, K. Tan, L. Shi, G. Lu, Y. Zhang, S. Lu, DCell: A scalable and fault-tolerant network structure for data centers, SIGCOMM Computer Communication Review 38 (4) (2008) 75–86.

[31] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, S. Lu, J. Wu, Scalable and cost-effective interconnection of data-center servers using dual server ports, IEEE/ACM Transactions on Networking 19 (1) (2011) 102–114.

[32] D. Guo, T. Chen, D. Li, M. Li, Y. Liu, G. Chen, Expandable and cost-effective network structures for data centers using dual-port servers, IEEE Transactions on Computers 62 (7) (2013) 1303–1317. doi:10.1109/TC.2012.90.

[33] A. Singla, C.-Y. Hong, L. Popa, P. B. Godfrey, Jellyfish: Networking data centers randomly, in: Procs. of the 9th USENIX Conf. on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 17–17.

[34] F. Petrini, M. Vanneschi, K -ary n -trees: High performance networks for massively parallel architectures, in: Proceedings of the 11th International Symposium on Parallel Processing, IPPS '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 87–.

[35] J. Kim, W. J. Dally, S. Scott, D. Abts, Technology-driven, highly-scalable dragonfly topology, in: Procs. of the 35th Annual Intl. Symposium on Computer Architecture, ISCA '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 77–88.

[36] J. Miguel-Alonso, J. Navaridas, F. J. Ridruejo, Interconnection network simulation using traces of mpi applications, International Jour-

nal of Parallel Programming 37 (2) (2009) 153–17[illegible] doi:10.1007/s10766-008-0089-y.

[37] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken, The nature of data center traffic: Measurements & analysis, in: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, IMC '09, ACM, New York, NY, USA, 2009, pp. 202–208. doi:10.1145/1644893.1644918.

[38] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, Parallel job scheduling — a status report, in: D. G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 1–16.

[39] D. Tsafrir, Y. Etsion, D. G. Feitelson, Modeling user runtime estimates, in: D. Feitelson, L. Frachtenberg, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 1–35.

[40] É. Gaussier, D. Glesser, V. Reis, D. Trystram, Improving backfilling by using machine learning to predict running times, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015, pp. 64:1–64:10.

[41] SLURM, https://slurm.schedmd.com/sched_config.html.

[42] J. A. Pascual, J. Navaridas, J. Miguel-Alonso, Effects of topology-aware allocation policies on scheduling performance, Job Scheduling Strate-

gies for Parallel Processing,, JSSPP 2009, Rome, Italy, May 29, 2009., 2009, pp. 138–156.

[43] C. R. Johnson, D. P. Bunde, V. J. Leung, A tie-breaking strategy for processor allocation in meshes, 39th Intl. Conf. on Parallel Processing, ICPP Workshops, San Diego, California, USA, 2010, pp. 331–338.

[44] M. Modarressi, M. Asadinia, H. Sarbazi-Azad, Using task migration to improve non-contiguous processor allocation in noc-based cmps, Journal of Systems Architecture 59 (7) (2013) 468 – 481. doi:https://doi.org/10.1016/j.sysarc.2013.03.011.
URL http://www.sciencedirect.com/science/article/pii/S1383762113000350

[45] J. A. Pascual, J. Miguel-Alonso, J. Navaridas, Effects of job and task placement on parallel scientific applications performance, in: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008) (PDP), Vol. 00, 2009, pp. 55–61. doi:10.1109/PDP.2009.53.
URL doi.ieeecomputersociety.org/10.1109/PDP.2009.53

[46] A. Bhatele, N. Jain, K. E. Isaacs, R. Buch, T. Gamblin, S. H. Langer, L. V. Kale, Optimizing the performance of parallel applications on a 5d torus via task mapping, 2014 21st International Conference on High Performance Computing (HiPC), 2014, pp. 1–10.

[47] M. Deveci, K. Kaya, B. Ucar, U. V. Catalyurek, Fast and high quality topology-aware task mapping, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 197–206. doi:10.1109/IPDPS.2015.93.

[48] E. Balzuweit, et al, Local search to improve coordinate-based task mapping, Parallel Computing 51 (2016) 67–78.

[49] J. A. Pascual, J. Miguel-Alonso, J. A. Lozano, Optimization-based mapping framework for parallel applications, J. Parallel Distrib. Comput. 71 (10) (2011) 1377–1387.

[50] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, C. H. Still, Mapping applications with collectives over sub-communicators on torus networks, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 1–11.

***Author Biog**

Dr. Javier Navaridas is a Senior Research Fellow at the School of Computer science and has a long publication record with more than 50 papers on many aspects of architectures of networks and computer systems. Javier is currently leading the workpackage on interconnects of the ExaNeSt H2020 FETHPC project and also contributes to the development of the system-level interconnect of EuroEXA H2020 FETHPC project.

Jose A. Pascual obtained his M.Eng. and PhD in computer science from the University of the Basque Country, Gipuzkoa, Spain, in 2005. He is currently working at the School of Computer Science of the University of Manchester. His research interests include interconnection networks, high-performance computing, scheduling for parallel processing and performance evaluation of parallel systems.

Alejandro Erickson completed a 3-year postdoctoral research position at Durham University, United Kingdom in 2016, where he did research on various topological aspects of interconnection networks, with an emphasis on applications in datacenter networks. He received his Ph.D. in Computer Science from the University of Victoria, Canada in 2013 and his M.Math in Combinatorics and Optimization from the University of Waterloo, Canada in 2008. Dr. Erickson has published in a broad range of topics, including datacenter networks, computational geometry, graph and matroid theory, enumerative combinatorics, education, and mathematical art.

Iain A. Stewart received the M.A. and Ph.D. degrees in mathematics from the University of Oxford, United Kingdom in 1983 and the University of London, United Kingdom in 1986. He is a professor in the School of Engineering and Computing Sciences, Durham University, United Kingdom. His research interests include interconnection networks for parallel and distributed computing, computational complexity and finite model theory, algorithmic and structural graph theory, theoretical aspects of artificial intelligence, GPGPU computing, and computational aspects of group theory.

Mikel Luján received the PhD degree in computer science from the University of Manchester. He is a Professor in the School of Computer Science, University of Manchester. His research interests include managed runtime environments and virtualization, manycore architectures, and application-specific systems and optimizations