Universidad del País Vasco
Euskal Herriko Unibertsitatea

Department of Computer Architecture and Technology

TOOLS FOR THE REALISTIC EVALUATION OF PARALLEL COMPUTING SYSTEMS

PhD Dissertation

Fco. Javier Ridruejo Pérez

Supervised by Prof. Jose Miguel-Alonso and
Dr. Javier Navaridas Palma

2013

Acknowledgements

I would like to thank everyone who helped me and supported my work during all these years, but I would certainly forget someone. I especially want to dedicate this thesis to those who have shared my difficulties whilst I was doing it:

- To my friends and supervisors José Miguel-Alonso and Javier Navaridas. I couldn't have removed this thorn from my flesh without their invaluable help. I will always be there for them.
- To my family, who always have shown me their unconditional support. Thanks for all the sacrifices they have made since I was a child; they taught me how to achieve my goals.
- To all the members of my research group in the University of The Basque Country, and all colleagues from other institutions who have helped in my research.
- To my partners in I2Basque for helping me and supporting my experiments.
- To my friends, for those moments we have shared, and those many more to come.
- And finally, but most importantly, Lorea, my beloved wife, for her tenacity, patience, support, and love, which made this dissertation almost as hers as mine.

Most of the research work carried out for this dissertation was supported by the Spanish Ministry of Education and Science, grants TIN2004-07440-C02-02 and TIN2007-68023-C02-02, and by the Basque Government grant IT-242-07.

Abstract

Top 500 supercomputers are very complex and expensive machines, but they are essential for scientific and technological advancement. For these reasons these distributed memory parallel computers with message passing through interconnection networks are the subject of extensive research work by groups around the world. This dissertation focuses on performance prediction and evaluation tools of those interconnection networks using simulation techniques.

All methodologies and tools we have developed have been integrated into INSEE, an Interconnection Network Simulation and Evaluation Environment. It allows evaluating different multicomputer architectures and interconnection network topologies, with different levels of detail, aiming the highest levels of accuracy while keeping low resource consumption. The input workload used in the simulation is the main factor in determining the level of simulation fidelity.

INSEE fills a gap in the simulation field as it allows the simulation of large supercomputers, and incorporates a comprehensive workload generation mechanism. For preliminary design phases INSEE provides fast statistical distributions, burst-based traffic generation and application micro-kernels. Trace-based traffic generation and full-system simulation provide the highest fidelity, necessary to fully understand all the mechanisms affecting the performance of a system. Traffic generation has been thoroughly studied and is the backbone of this dissertation, which focuses on explaining how the different traffic generation models were devised, developed, implemented and tested into INSEE. All these features have made INSEE to be used by other research groups around the world.

All traffic generation models are designed to reflect better the way scientific applications exchange messages, incorporating causality among them, a feature that classical traffic generation based on statistical distributions lack. These traffic generation models were designed and implemented as they were needed during our research.

INSEE and these traffic models have been used in the context of this dissertation to research on IN topologies and their performance, analyze network-level congestion control mechanisms and network-level policies, and to examine the implications of using realistic full-system simulation. Use cases and conclusions of this research using INSEE and its traffic generation models are provided as results of this dissertation.

Table of Contents

Part 1:

Evaluation of parallel computing systems

Chapter 1. Evaluation of parallel computers using simulation

This dissertation collects the research work performed by the author in the field of tools and methodologies for the evaluation and prediction of performance of parallel applications and computer systems. Moreover, it pulls together some research works where these tools and methodologies have been used. It must be pointed out that:

1. The work described in this thesis has been developed within the research group to which its author belonged, the Intelligent Systems Group of the University of the Basque Country UPV/EHU. Substantial parts of this work have been performed in collaboration with other group members. However, this dissertation focuses on the contributions by its author.

2. We have centered our work on specific system architecture: multicomputers[1], or distributed memory parallel computers, with message passing communication between nodes as seen in Figure 1. In particular, we have paid most of our attention to an important part of the multicomputer: the interconnection network (IN) that allows compute nodes (and, therefore, applications) to communicate and synchronize.

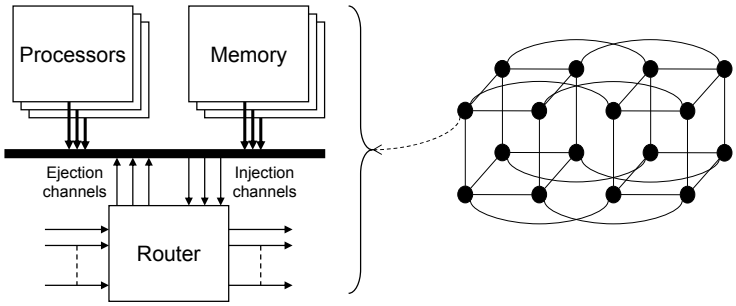


Figure 1: Communication-centric detail of a *2-ary 4-cube* multicomputer node

Multicomputers are made up of many components, and it is difficult to assess how a change in one of these components affects the overall system behavior without taking into

1 Multicomputers are a special kind of Multiple Instruction Multiple Data (MIMD) processors, according to Flynn's computing systems taxonomy

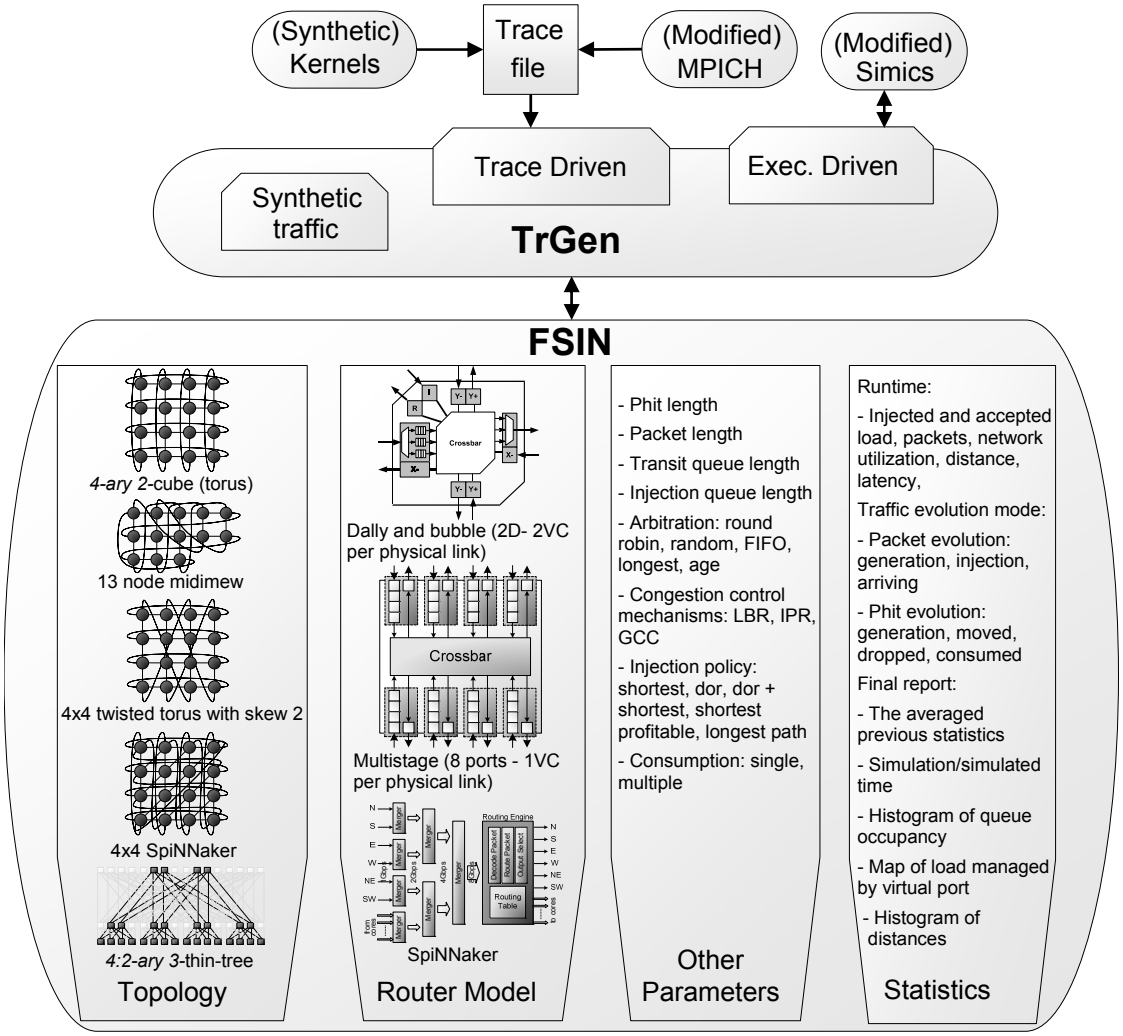
account all possible interactions. Evaluation of multicomputers can be done using different approaches, such as analytical methods (Markov chains, queuing theory or Petri nets), simulation, and empirical testing.

Allegedly, the best way of evaluating the performance of a computer would be just running on it an application of interest and measuring the execution time. However, this approach has several important limitations that we need to bear in mind, mostly related to its lack of flexibility. It does not allow us to predict performance figures on a different kind of computer system, or on a scaled version of the same system, or on the same computer system but with a different application.

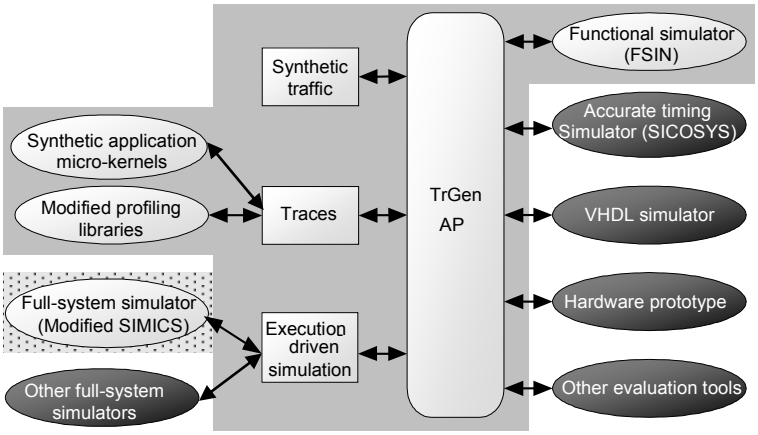
Simulation is acknowledged by the computer architecture community as a valid and flexible way to evaluate parallel computing systems. We can model, with different degrees of fidelity, the system under study, and set up different scenarios to understand its behavior under selected circumstances, such as using different components or running different workloads. A simulation environment should offer:

- Modeling flexibility: the ability of modeling a variety of systems with multiple features and sizes.
- Multiple workloads: the ability to accept many kinds of inputs or workloads to feed the simulated system.
- Multiple measurements: the ability to offer different kinds of information and metrics of the simulated computing system, from the running time of an application, to a detailed trace of all messages exchanged in the IN by all processes.

This work pays special attention to the different kind of workloads that can be generated and used for the simulation of parallel systems. Many alternatives to generate workloads have been proposed and used in this context. Some are very generic and could be used, for example, to study the behavior of the simulated system under different levels of pressure on the communication network. Others are much more specific, allowing the researcher to identify the exact behavior of a given parallel application running under very particular conditions, like a specific MPI version installed on a cluster constituted by nodes with a specific hardware.



a)



b)

Figure 2: Two views of the overall design of INSEE. (a) INSEE modules as shown in Chapter 8. (b) Interactions between INSEE modules and third-party modules as shown in Chapter 10, adding the afterwards developed application micro-kernels.

Tools and methodologies presented in this dissertation have been integrated into **INSEE**, the Interconnection Network Simulation and Evaluation Environment developed at the University of the Basque Country UPV/EHU. INSEE collects the most versatile collection of tools and techniques for the evaluation of multicomputers we are aware of. They allow the evaluation of different kinds of multicomputer architectures and IN topologies; they can do simulations with different levels of fidelity and detail, dealing with a wide range of options for workloads, from the most simplistic to the most realistic. Moreover, they provide detailed measurements of every aspect of the simulation, from simple metrics to a detailed trace of every event in the simulation. The accuracy of INSEE has been cross-validated with SICOSYS [PGB02] a much more complex simulator which achieves hardware-level precision, resulting in negligible differences. Despite this high accuracy, it has a remarkably low footprint: even an entry-level consumer-grade compute platform is enough to simulate systems of the size of those in the first positions of the TOP500 list, as shown by our research work on the SpiNNaker multicomputer that has a two-dimensional triangular torus network of 65,536 nodes [NLM09], the study of the twisted torus topologies with experiments simulating the Bluegene/L with 65,536 nodes [CMV10], the analysis of indirect topologies with up to 14,541 nodes [NMRD09] or those with indirect cubes with 7,680 nodes [NM11].

INSEE is composed of two main modules, **TrGen** and **FSIN**. FSIN is the Functional Simulator and models the network details, including the topological arrangement of the system, the router architecture and the transmission of the packets through the network. TrGen is the module that implements all traffic generation mechanisms. Figure 2a) shows the INSEE design, centered on the relation of its principal modules and components, while Figure 2b) shows the INSEE design from the point of view of TrGen and the workloads it can provide, along with all third-party modules it can connect to, in order to provide additional features. The author has driven the development of TrGen being in charge of the implementation of the interfaces which allow performing simulation with realistic traffic.

1.1. Typical analyses performed with INSEE

INSEE can be used to carry out many different researches on INs, so different measures and output data are needed as result of the simulations: injected/accepted load, injected/consumed/dropped packets, average number of packets in the system, distance average traveled by packets, average/maximum latency, histogram of the queue occupancy, map of load average of each port, histogram of distances, simulation time, actual time to perform simulation, etc. The results can be obtained for the whole simulation in a final report, as an average from different intervals, or cycle-by-cycle for debugging purposes. The user can directly process this data, or import it to other tools for further analysis. In Figure 3 we have included some example plots of INSEE-generated data, useful for analysis of different INs properties:

a) Throughput analysis.

b) Latency analysis.

c) Application performance analysis.

d) Network balance analysis.

e) Distance distribution analysis.

f) Task-to-task communication analysis.

g) Fault-tolerance analysis.

h) System evolution analysis.

i) Injected packets analysis.

The INSEE environment has been used frequently within the Intelligent Systems Group of the UPV/EHU. However, it has also been used and referenced by other research groups. A non-exhaustive list of external institutions that have contributed to INSEE, or have used it to carry out research on interconnection networks, or have used results obtained using INSEE, includes:

- The University of Adelaide [IV12]
- Universidad de Cantabria [SCVB12][CMB12][CMV10]
- Universidad de Castilla-La Mancha [EGQ11]
- Shangai Normal University [WZP12]
- The University of Melbourne [GB11][GB12]
- The University of Manchester [NLM09][RNJ11]
- The University of York [BBS12]

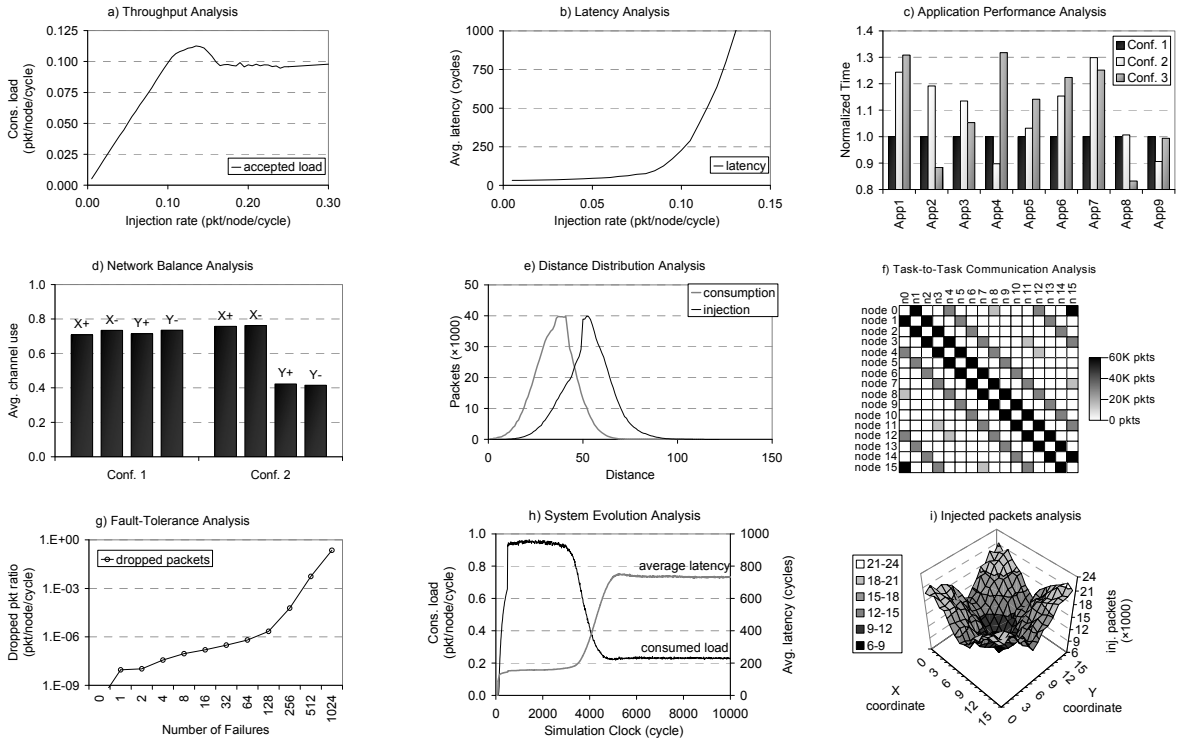


Figure 3: Examples of IN analysis supported by INSEE as shown in Chapter 8.

These are some examples of studies carried out with our toolset:

- Study of throughput fairness in INs [IV12] [IZU09]

- Causes and consequences of congestion in IN, and congestion control techniques [EGQ11] [WZP12]
- Novel topologies for interconnection networks [GB11] [GB12] [CMB12] [CMV10] [NLM09] [RNJ11] [SCVB12]
- Parallel scheduling algorithms [BBS12] [NPM09]

There are many other research groups around the world interested in simulating multicomputers, including both the nodes and the network that connects them. However, most of them are only interested in either the performance evaluation of workloads on servers, or in the assessment of a particular micro-architectural improvement. Different needs often leads to different evaluation tools. With INSEE we aim to provide a tool supporting a wide spectrum of evaluation needs. A comprehensive list of available simulators is outside of the scope of this introduction, but the interested reader can look at Chapter 8 for an extensive review of network simulators and at Chapter 10 for full system simulators.

1.2. Resource requirements of INSEE

INSEE uses a small amount of resources (memory and CPU time), allowing large-scale simulations in a single off-the-shelf computer, but only when the simulation is done with synthetic workloads, those based on statistical distributions or micro-kernel applications. Using more complex workloads (for example, application traces) requires considerably more resources (CPU time and memory), which increase linearly with the number of nodes. The most demanding set-up is full-system simulation, in which many instances of the (resource-demanding) Simics program [MCEFH02] take care of accurately simulating the nodes (including the operating systems, device drivers, and MPI libraries that run the unmodified parallel applications) while INSEE makes a (comparatively fast) simulation of the IN. This kind of simulation requires a computing platform with a size in the same order of magnitude of that of the system being modelled, and suffers from severe slowdowns when compared with the actual execution of applications on an actual system.

A comparison of the resources used by INSEE for several different simulations can be observed in Figure 4. These experiments are done using off-the-self computers based on a Pentium 4 at 3.0GHz with 1.5 GB of RAM. We can see that time and memory needed depends almost linearly on the number of routers. For example, Figure 4a) shows the simulation of several 2D-torus INs from 16 to 4096 nodes with uniform synthetic traffic; time and memory scale linearly to the number of nodes and their queues' capacity. A similar study in *2-ary n-trees*, Figure 4b), shows that resources scale along with the number of switches in the order of O(n log n) for the same number of nodes.

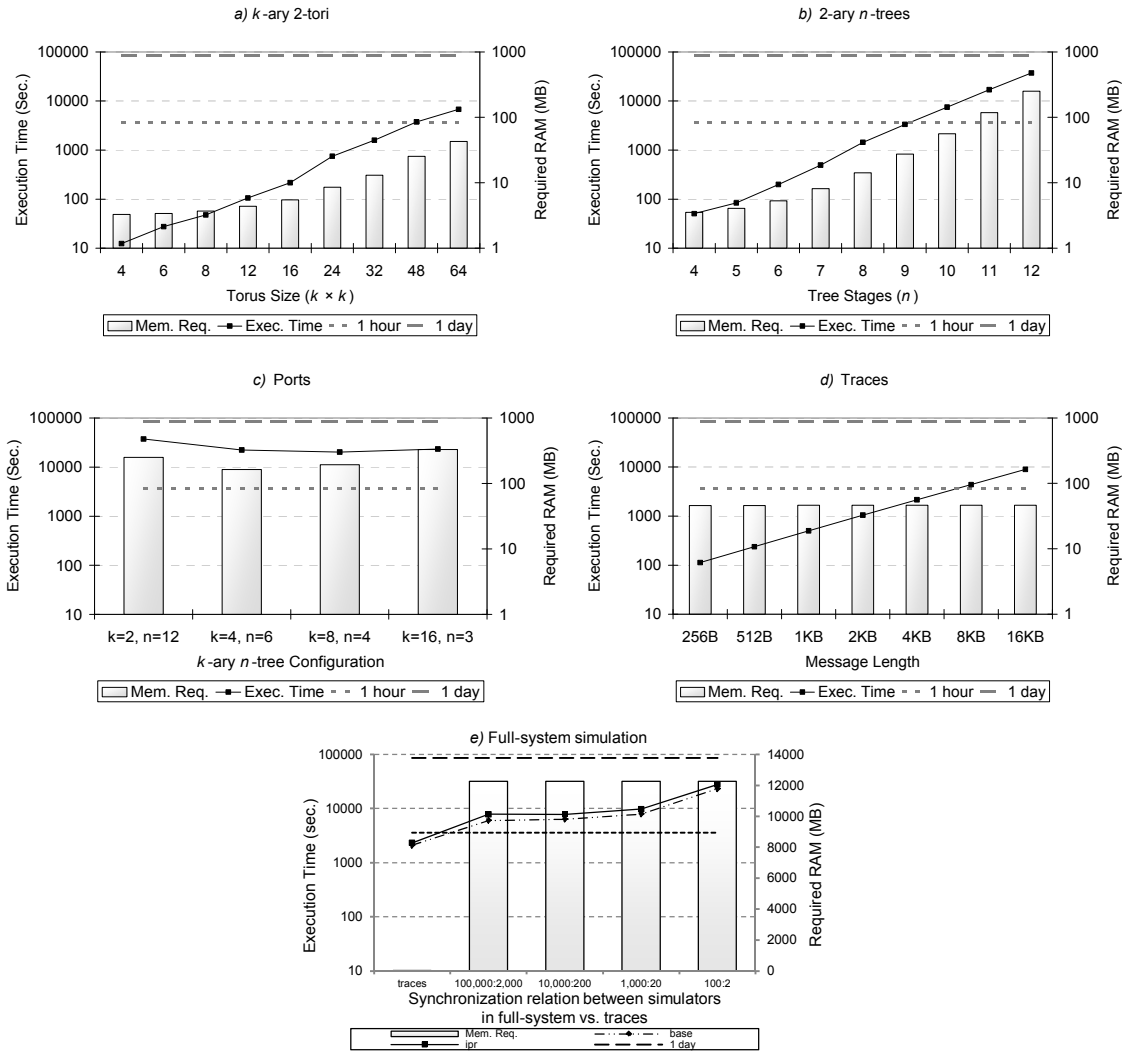


Figure 4: Resources needed to perform INSEE simulations under different configurations. Torus topologies a) vs. trees b). Different configurations for the same tree topology c). Synthetic traffic a), b) and c) vs. trace-based traffic d) vs. full-system simulation e).

The topology being modelled also has a bearing on the amount of resources needed. We can see in Figure 4c) that different fat-tree topologies, all of them for the same number of compute nodes, can differ in terms of resources usage by a 4x factor. Moreover, message length increases linearly the execution time but not the memory used, as shown in Figure 4d). When we compare resources used by full-system simulation with those using traces-based traffic, we can see in Figure 4e) how the execution of full-system simulation experiments requires above two orders of magnitude more time (and that is using 16 CPUs rather than a single one) and almost three orders of magnitude more RAM memory. The figure also shows the impact that synchronisation has on the execution time of full system simulation: the more frequently simulators synchronise the slower the simulation is. This is investigated in more detail in Section 8 of Chapter 3 and in Chapter 10.

Chapter 2. Workloads and traffic generation

Simulation can be used from the very first stages of the design of a multicomputer. In these first stages a fast simulation environment is used to explore as many alternatives as possible, even if accuracy and fidelity are not perfect. However, in their last design phases a highly accurate simulation must ensure the adequacy of design decisions. Most simulation tools center their efforts on accurately modeling the architecture of the system under evaluation, but in our opinion, the choice of the workloads that will feed the simulation is also of paramount importance.

traffic model		spatial pattern	Causality	complexity	evaluates
independent traffic sources	**random**	Random	No	very low	raw performance
	permutation	worst case	No	very low	raw performance
	estimation of distributions	application-like (origin-dependent)	No	low / medium	selected application (origin-dependent)
bursty traffic sources	**random**	Random	coarse-grained	very low	raw performance
	permutation	worst case	coarse-grained	very low	raw performance
	estimation of distributions	application-like (origin-dependent)	coarse-grained	low / medium	selected application (origin-dependent)
application-based	**application kernels**	application-like	application-like	low / medium	usual communication patterns
	trace-driven (inject-at-will)	application-like (origin-dependent)	No	medium	raw performance when congested
	trace-driven (causal)	application-like (origin-dependent)	application-like (origin-dependent)	medium / high	selected application (origin-dependent)
	execution-driven	actual application	actual application	very high	selected application

Table 1: Description of different traffic models used for simulation-based evaluation of parallel systems

In our context, workloads are the collection of messages generated by the applications on the computing elements that traverse the IN. These may be packetized before being put into the network, and reassembled at their destination nodes.

TrGen [RGM05] is the component of INSEE that encompasses all the traffic generation mechanisms. The characterization of the traffic that the IN has to deal with is important in every phase of the design. Different programs executed on top of an IN generate diverse workloads that impose a variety of stress levels into the communications fabric. A flexible

traffic generation subsystem, such as TrGen, allows the researcher to emulate what kind and volume of traffic will be managed by the IN, resulting in a rich environment to test its performance. Table 1 obtained from Chapter 8 shows a list of the different workloads INSEE can generate and several of their features.

In the following sections we describe the traffic generated models incorporated into INSEE. The order is not arbitrary: it is determined by the order in which each particular model was incorporated into the toolset.

2.1. Synthetic traffic

We started the design of TrGen using the mechanisms described in the literature to define simple workloads that are based on probability distributions and specific permutations. This kind of **synthetic traffic** patterns provide workloads that are useful to quickly evaluate the "raw" performance of a system: maximum throughput, minimum delay, or expected saturation level, among others. Another advantage is that the results obtained with this kind of traffic can be validated with mathematical models of the implemented topologies, for example, throughput analysis for uniform traffic and distance-related characteristics of the topologies. These probability distributions are usually applied to destination of packets, packet size, and inter-generation times. Parameters of the probability distributions used in the experiments can be set up manually, or can be inferred from actual traffic, analyzing logs of applications by means of a distribution fitting procedure.

2.2. Traces

Soon we found synthetic traffic very limiting and not as realistic as we intended it to be, mainly because its inability to capture the temporal relationships among messages. Application tasks do not only communicate, but also synchronize, and pass through different phases of high and low communication. Causality among messages is not addressed either. For this reason, we developed and integrated into TrGen a more realistic approach to traffic generation: feeding the simulator with **application traces** obtained from actual applications executed on existing multicomputers. However, as explained in Chapter 9, capturing and generating traces from applications is not a straightforward process and there are several alternative methodologies that could be used. We decided to enhance the logging module available in MPICH (a popular, freely available MPI implementation [GLDS96]) to generate extended[2] traces and implemented a module into TrGen to feed simulations with them, using the captured message sizes, destinations and causal relationships. In some cases, though, we disregard injection and consumption times because they are too tied to the system in which traces were captured.

[2] Extended traces are like the regular log files generated by MPICH, but making visible the point-to-point operations used to implement collective communications that, by default, are hidden.

We can see in Figure 5 an example of the extended trace for the CG.W.8 application of the NAS Parallel Benchmarks [NPB] suite (NPB) representing only a small part of the execution. In the plot, different block colors in the timelines (one per task) represent different states (e.g. yellow is a barrier, purple is a reduction, red is a wait). The remaining (black background) is computation time. Arrows between two tasks represent explicit messages between them. These arrows are the most important information for TrGen as they contain the information about the time when nodes need to inject messages or stall until reception. Further details of the information in a trace and its generation can be consulted in Chapter 9, while the different phases of the CG.A.64 application are explained in Chapter 12.

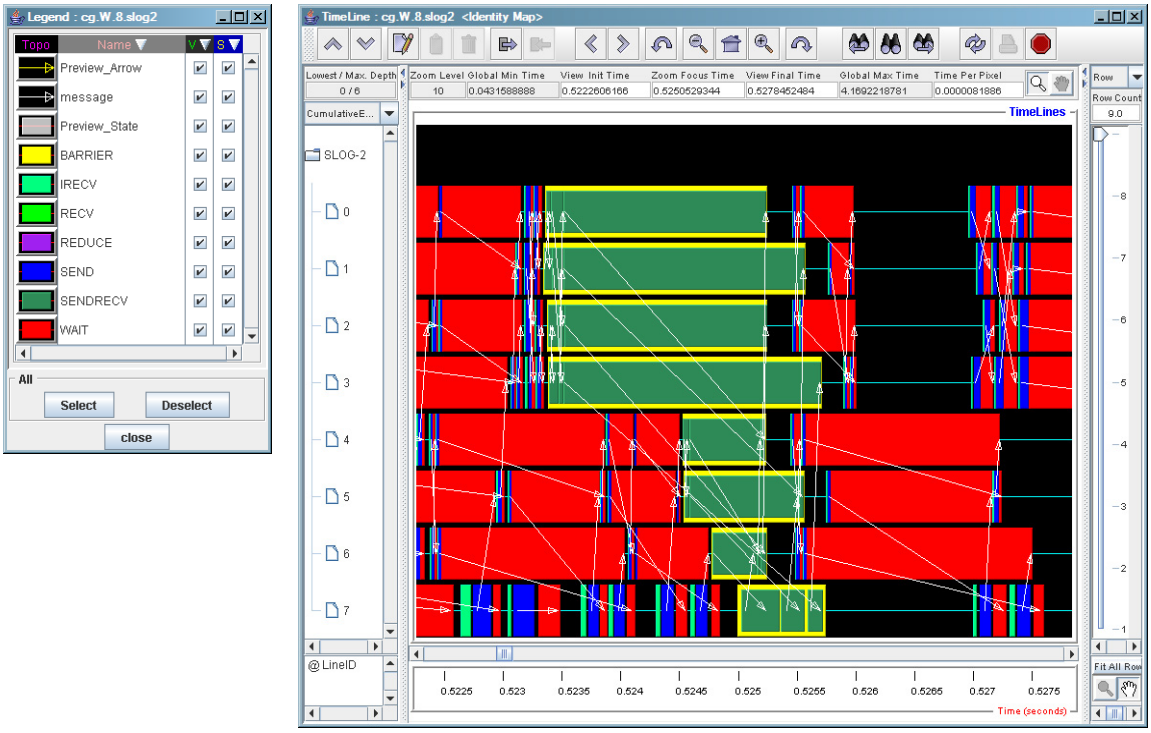


Figure 5: Screenshot of Jumpshot [ZLGS99] visualizing an extended trace file (CG.W.8).

The design of the TrGen module in charge of the trace-based simulation is shown in Figure 6 and it is explained in Chapter 9. We also define there a methodology to carry out performance prediction studies: INSEE is able to estimate the run time of an application on an architecture different from the one used to capture the traces.

Despite being a rather realistic source of traffic, traces present several limitations, mostly related to their lack of flexibility: a trace represents the run of one application in one specific system using a given number of processors and a specific input data, so traces may have embedded some of the characteristics of the system in which they have been obtained. Therefore, a single trace cannot be used directly to carry out scalability studies. Besides, traces could be inexact due to the intrusion of the logging mechanism. And finally, traces of actual applications running on large multicomputers are difficult to obtain, even if they are available, adding difficulties to our performance evaluation studies.

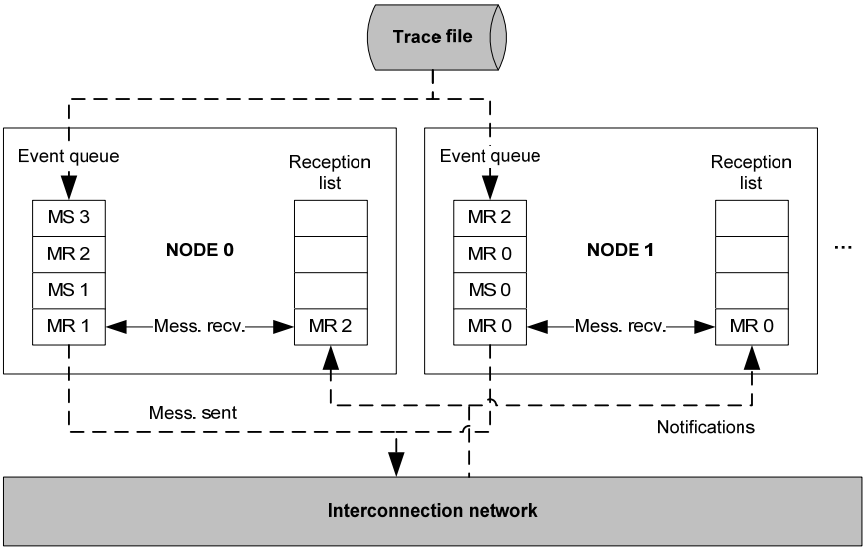


Figure 6: Graphical representation of the extended trace file processing by TrGen showing the data structure containing the information from a trace. In the boxes, “MS n” means that a message is sent to node #n, and “MR n” means that a message needs to be received from node #n in order to continue with the execution.

2.3. Full-system simulation

Bearing in mind some of the former limitations of application traces, we incorporated into INSEE yet another means of generating realistic traffic: **full-system simulation**. This method analyzes a system by simulating all its components: computing nodes, IN, operating systems, drivers, communication protocol stacks, MPI libraries and parallel applications running on top and injecting traffic into the simulated IN. Communication among nodes is real and all causal relationships, synchronization, injection pace and computing times are those of the actual application being run on the simulated system.

Our approach consisted on simulating computing nodes using Simics [MCEFH02], integrating them with INSEE via the TrGen module as seen in the design of Figure 7. Network traffic generated by Simics computing nodes is injected into INSEE, which delivers it to the destination node. All the execution nodes are kept synchronized, among them and with INSEE by means of a custom-made synchronization protocol explained in Chapter 10 and shown in Figure 8.

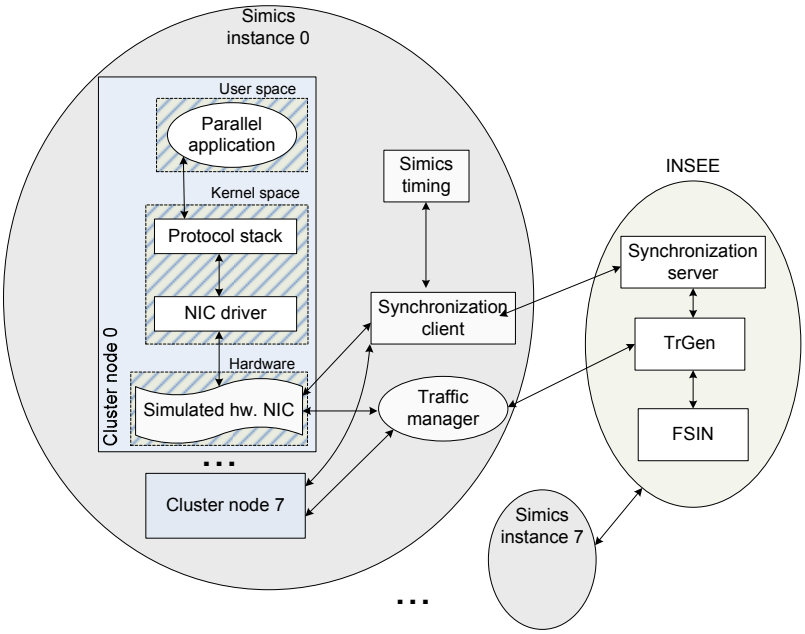


Figure 7: Elements of our full-system simulation environment that simulates an MPI application running on top of an INSEE (simulated) network.

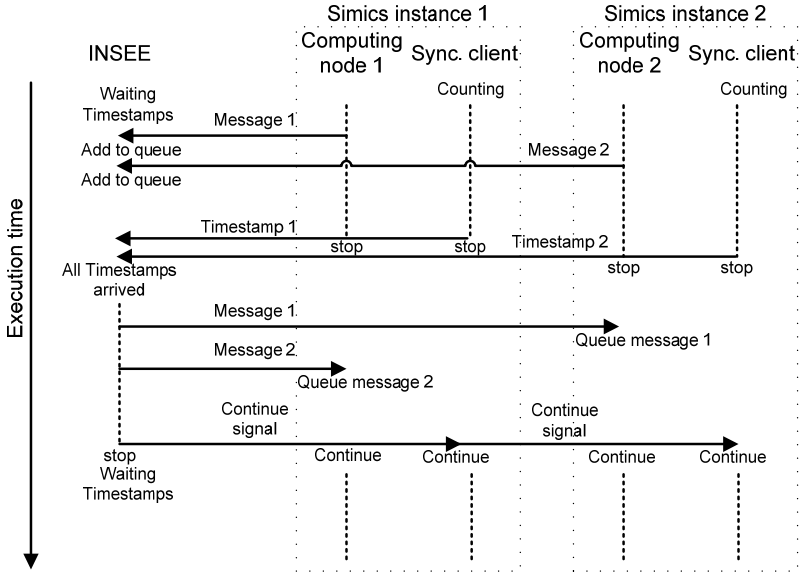


Figure 8: Synchronization mechanism between Simics instances and INSEE, and message routing. Only one computing element per Simics instance is shown

As far as we were aware of, by the time we developed this environment, there was no other system with the same purpose (validation of IN designs using simulation of the complete parallel computers). Either existing IN simulators incorporated very simplified models for the computer nodes connected to the IN, or the IN modeled was too simplistic for our needs. Our choice was to build a full-system simulation environment targeting IN studies by means of connecting two different simulators, an external, existing one for the nodes, and INSEE for the IN, ensuring the appropriate synchronization among simulation entities.

Full-system simulation is an error prone task, because there are many components that interact and that have to be carefully tuned. Otherwise, the potential profit of a design option can be completely hidden by unexpected interactions with surrounding elements – or can also be overstated.

Another important issue is the huge amount of resources needed to simulate a multicomputer; if we compare resources used by full-system simulation to those used by trace-based simulation (see again Figure 4e). Execution is 3 to 4 times slower, even when we are using 16 processors. Similarly, memory usage rockets from 35MB to 12GB of RAM. This is because RAM use increases linearly with the number of simulated Simics nodes.

Full-system simulation is thoroughly explained in Chapter 10 where congestion control is used as a case study.

We cross-validated the trace-driven simulation with our full-system simulation environment as reported in [NRM07], finding out that full-system simulation is the most realistic possible workload, with results equivalent to those of traces, but attenuated due to nodes computing time.

2.4. Bursty traffic

In view of the difficulties of finding traces of large multicomputers and the impossibility of full-system simulate them, we tried to enhance the generation of synthetic traffic models by introducing different levels of synchronization. We first added the **bursty traffic** generation, which emulates coarse-grained barrier synchronization among tasks. It works as follows: each node injects a burst of packets and then waits until all nodes' bursts have been released and consumed. A comparison among bursty traffic, application traces and full-system simulation is done in [RNMI07]. Bursty traffic is explained in detail in Chapter 12, where we use congestion control as a case study.

2.5. Application micro-kernels

The rough synchronization model offered by bursty traffic was a good starting point to emulate the behavior of applications at a lower cost than using traces or full system simulation. However, a wide range of applications use task-level synchronization models. Although there are plenty of different applications using fine-grained synchronization, a detailed investigation of common applications running on supercomputing facilities unveiled that there are some particular communication and synchronization patterns that are used frequently. We captured some of those patterns in the form of **application micro-kernels** with some examples of them represented in Figure 9. A micro-kernel provides a spatial pattern together with the causality properties of messages, and can be configured with an arbitrary number of nodes and message size. These application micro-kernels can be understood as short trace-like workloads, but with much more flexibility, because they can be adapted to different system sizes and different "communication intensities". This workload generation mechanism is described in Chapter 11 and used in Chapter 13 to evaluate a family of indirect networks.

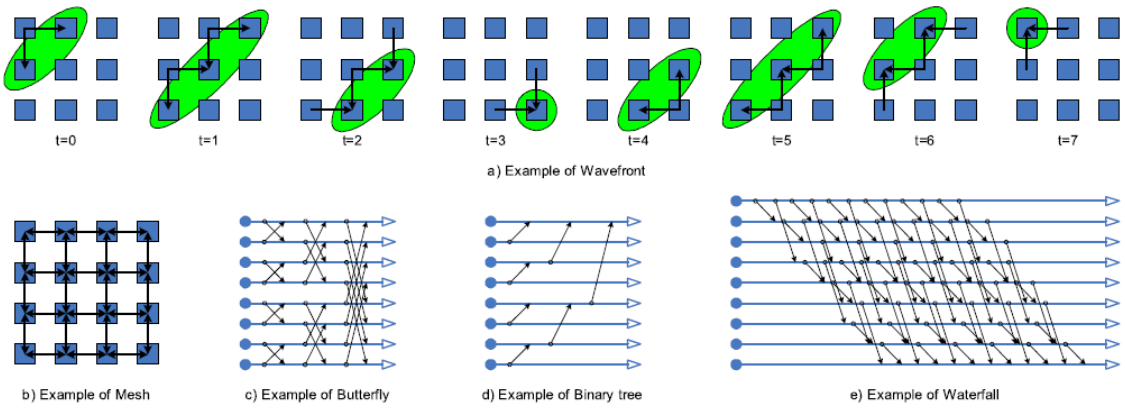


Figure 9: Graphical representation of the traffic patterns: Time flows from left to right. Grey lines and squares represent nodes. Each black arrow start means a message send. The end of the black arrows means that the node has to stop until receiving the corresponding message. a) Wave-front (Green) in a 3x3 2D-mesh. b) Neighbor interchange in a 4x4 2D-mesh. c) Butterfly that emulates N-to-N collectives (8 nodes). d) Binary tree that emulates N-to-1 collectives (8 nodes). e) Waterfall pattern (9 nodes).

Chapter 3. Case studies

INSEE has been used in multiple research works carried out by our group. In this section we have selected a collection of representative case studies, analyzing them from the point of view of the role played by the different workloads used in the analyses.

The case studies can be split into three groups based on the kind of analysis carried out. These groups are:

- Research on topologies: studies applying INSEE to the research of different IN topologies, some commonly used in supercomputing systems, and some novel.
- Network–level policies and mechanisms: study of different policies, techniques and mechanisms applied to the routers in order to improve the performance of an IN.
- Cautions using full-system simulation: study on how to do full-system simulation of IN, issues to take into account when doing it, and main trade-offs to take under consideration.

Research on topologies

3.1. Estimating execution times of applications in different topologies

We define in Section 4.3 of Chapter 9 a methodology to predict the performance of an application being run on a different IN architecture using **trace-based workloads**. We validate our methodology in Section 6.2 of that chapter.

We compared the real execution of the CG benchmark on the MareNostrum Supercomputer [BSC] (built around a fat-tree implemented with Myrinet 2000) to three 8x8 tori of different speeds, using the traces obtained from the real MareNostrum execution.

MareNostrum	8x8 torus 100 Mb/s	8x8 torus 1 Gb/s	8x8 torus 10 Gb/s
0.54s	3.49s	0.55s	0.25s

Table 2: Actual execution time of CG.W.64 in the MareNostrum, and estimated times for three different target architectures.

In that study we learnt that CG is a communication-intensive application that can take advantage from network improvements as Table 2 shows. The 100Mb/s torus shows that almost all execution time is dedicated to communication, because the speedup is of 6,35 when

compared to the 1Gb/s network, and when comparing this faster network to the even faster 10Gb/s one the speedup is only 2.2. In high speed networks, most of the running time is spent on computation.

The small difference between the MareNostrum with a Myrinet 2000 network (operating at 2Gb/s) and the 1Gb/s torus can be explained by the fact that experiments in the MareNostrum were done with the machine in production, so we do not have guarantees that our 64 nodes were on consecutive leaves of the 10K-nodes fat-tree network, or that the network was being heavily used by other, concurrently running, jobs.

3.2. Influence of the topology on application runtime

We also studied in detail in Chapter 11 how workloads behave in different network topologies. We captured the temporal evolution of the load imposed by applications into the network to test how network topology and workload execution are directly related. So we chose to compare three topologies: a 64-port crossbar as the ideal network, a *2-ary 6-tree* and an *8-ary 2-cube* (8x8 torus). The workloads were the following **application micro-kernels**: Binary Tree (BT), Wave-Front 3D (3W), Butterfly (BU) and Distribution 2D (2M). Results are shown in Figure 10; the represented values are the measure of the average consumed load in small periods of 10 cycles.

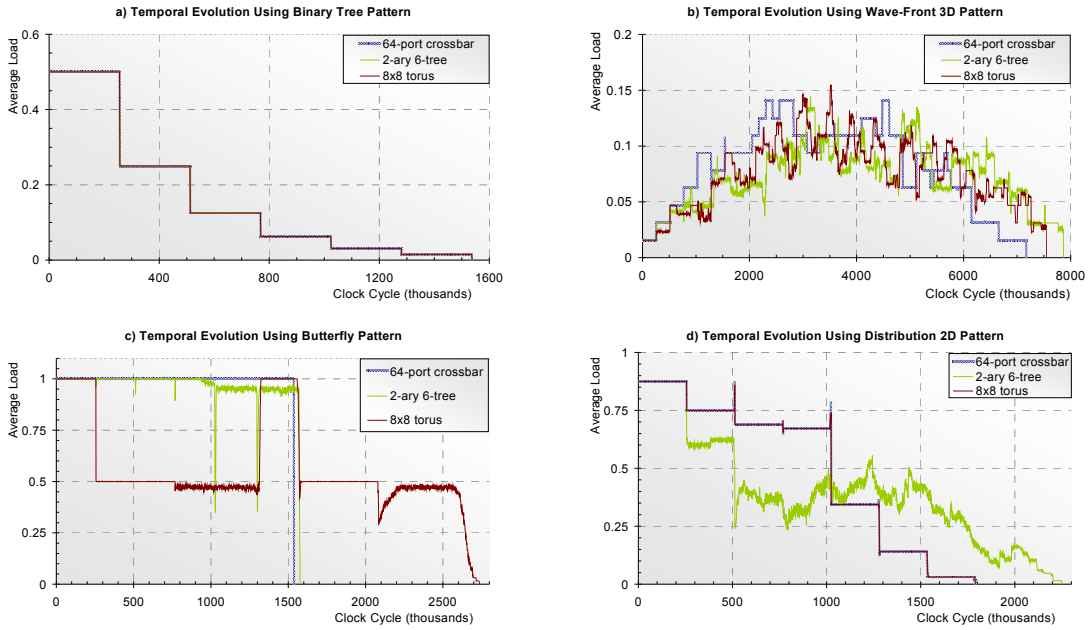


Figure 10: Temporal evolution of the average consumed load measured in phit/cycle/node for different traffic patterns: a) BT pattern. b) 3W pattern. c) BU pattern. d) 2M pattern.

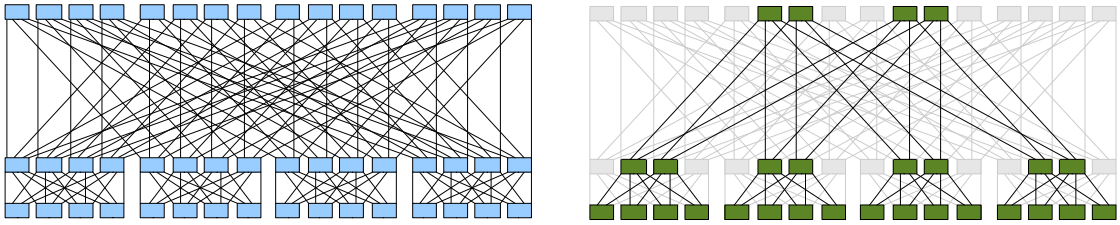
When load is light the network does not saturate, and paths of messages do not overlap. See BT in Figure 10a), where we can see how all topologies behave the same way, consuming the same load as the ideal crossbar topology, and without introducing additional delays due to (temporary) congestion. However, if network communication is intensive, as in Figures 10c) and 10d), the topology has a big influence in how the workload is processed, resulting in an additional (topology-dependent) latency. BU in Figure 10c) fits better in a tree-like topology,

while in a torus topology, messages overlap, so the torus needs twice the time of the fat-tree to process all messages. On the contrary, when the traffic pattern is 2M (data distribution on a mesh) as in Figure 10d), the mapping of the workload to a torus topology is optimal, while the fat-tree delivers the workload at a 25% smaller rate. Also, when the workload is complex but not very communication-intensive, as happens with 3W (a diagonal sweep of a mesh), topologies behave very differently over time, but end in similar times compared to the ideal crossbar, as can be observed in Figure 10b).

We can conclude that particular workloads fit better on specific topologies, and run in a non-optimal way on other topologies. This should be taken into account when designing a new proposal of IN for running a specific application, or when designing a parallel application for an specific supercomputer. Moreover, we have shown how communication requirements change over time in real applications.

3.3. Evaluation of the Thin-tree topology

We used INSEE in Chapter 13 to assess the performance of a new topology based on the well-known *k*-ary *n*-tree as seen in Figure 11. This topology, *k:k'*-ary *n*-thin tree, was studied using analytical methods, and was evaluated using simulation with different kinds of workloads, synthetic traffic and micro-kernel applications.

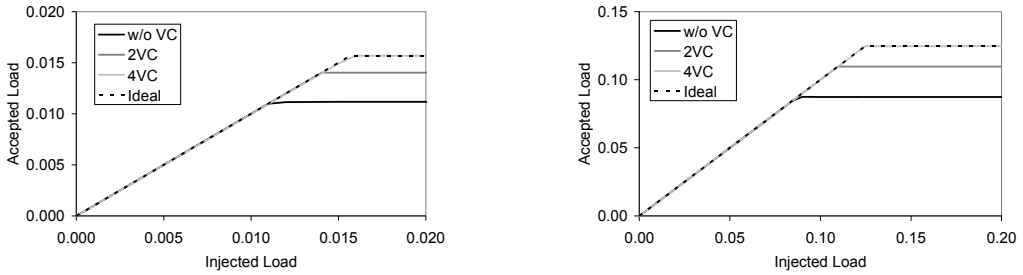


a) b)

Figure 11: Samples of the topologies under study to build 64 nodes networks. a) 4-ary 3-tree. b) 4:2-ary 3-thin-tree.

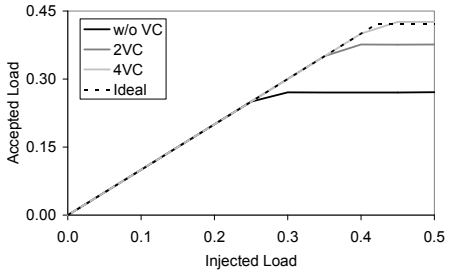
We first studied the theoretical throughput of the topology, using **synthetic uniform traffic** with independent sources, which corroborated the analytical study. These preliminary, non-realistic tests could make us think that thin-trees are not a good topological choice, as their ideal limits are far from the ideal limit of 1 phit per cycle consumption for a regular *k*–ary *n*-tree, as shown in Figure 12.

However, a study using **micro-kernel applications** showed us a different story. The load that traverses the last (upper) stages of a tree-like topology is smaller than that traversing the first (lower) stages; therefore, we can build trees that are thinner at the upper stages without adversely affecting the execution time of applications—but with a very positive impact in terms of budget because it uses less network elements. It is important to remark that an analysis using trace-based workloads was not feasible, because there were no available traces for tree-based INs large enough for our experiments.

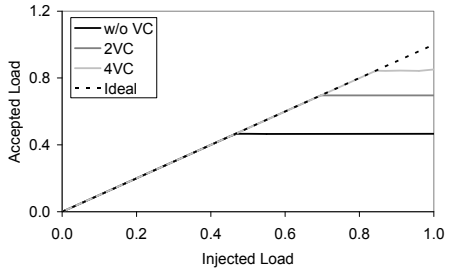


a) 8:2,4-tree

b) 8:4,4-tree



c) 8:6,4-tree

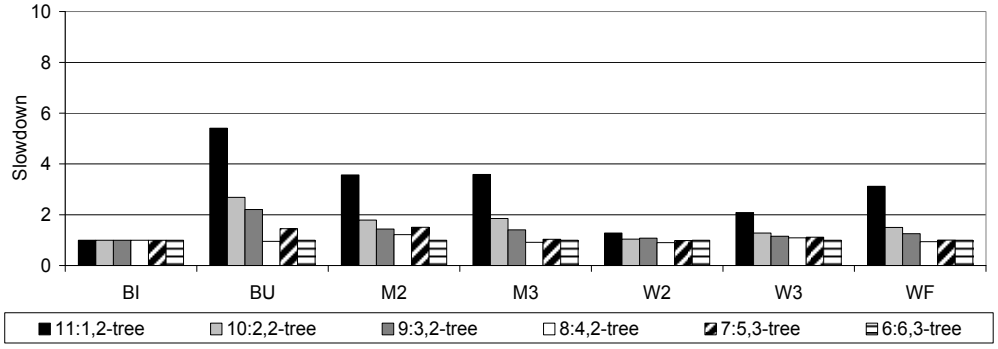


d) 8:8,4-tree

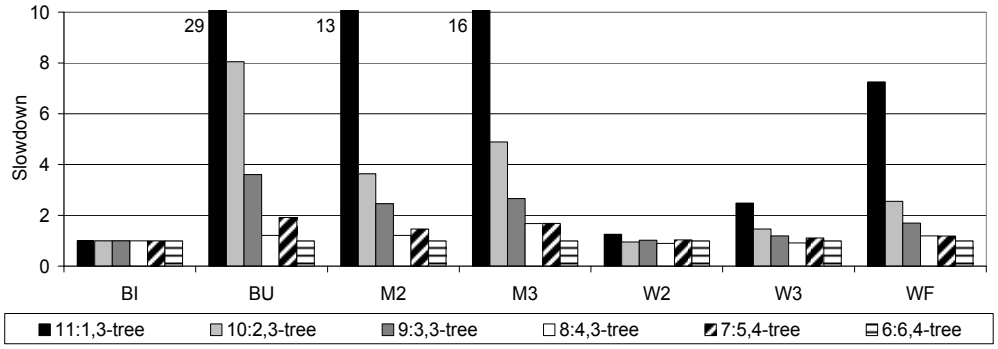
Figure 12: Relative throughput for networks under study with uniform traffic with independent traffic sources. With 1, 2 or 4 VC and the upper boundary of the throughput. Note the difference in the axis ranges of each topology.

We carried out some experiments comparing different thin-tree network topologies built with switches with the same radix (12 ports). Results are plotted in Figure 13. In general, the slimmed topologies have an additional advantage: the increased ability to exploit locality in communication. In these topologies the unused upward ports are rearranged to work as downward ports. To a certain extent, this compensates the reduction of links and switches in the upper levels. The result is that slimmed topologies may outperform the complete trees as we can see in the BU pattern, where the 8:4,2-tree is able to consume the workload in less time than the 6:6,3-tree.

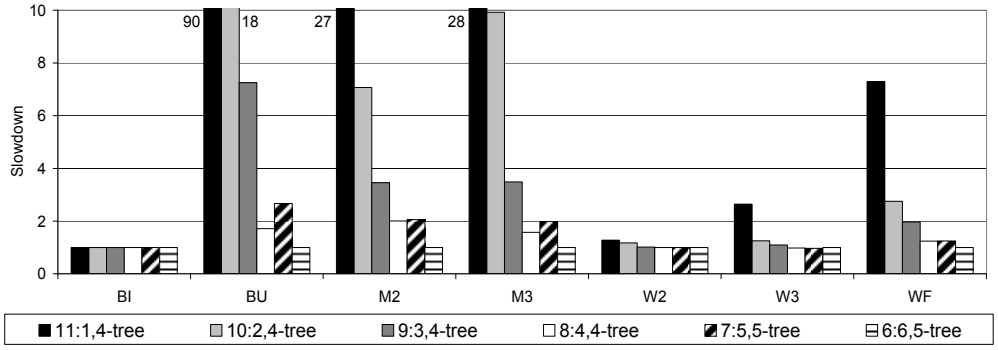
We concluded that for the lighter workloads (BI, W2, W3), there is almost no difference between regular *k*-ary *n*-tree and thin-tree topologies, whereas for heavy loads (BU, M2, M3) networks with 2:1 slimming ratio between upwards and downwards ports perform acceptably well, or even better, than those with 1:1 (the full-fledged fat-tree). These results are far different from the first estimations obtained using uniform synthetic traffic, which demonstrates the advantages of using different workloads for different purposes. This advocates the utilization of flexible tools such as INSEE.



a) 64-node workloads



b) 512-node workloads



c) 4096-node workloads

Figure 13: Normalized time to perform all communications of each traffic pattern in the networks with radix-12 switches.

Network –level policies and mechanisms

3.4. Evaluation of routing strategies and virtual channel management

In Chapter 9 section 6.1 we first compare two routing strategies, dimension-order oblivious-routing with one or three virtual channels, and adaptive routing with three virtual channels. Simulations were done using **trace-based workloads** from applications of the NPB. The results of experiments are summarized in Figure 14.

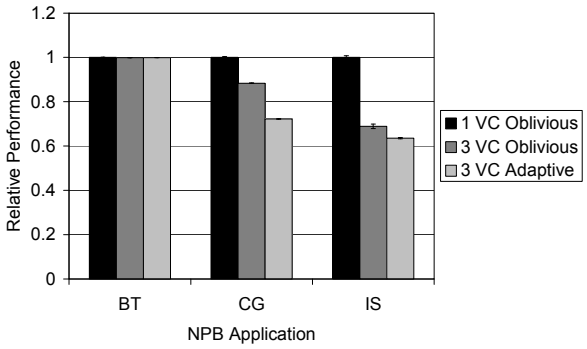


Figure 14: Effects of using 3 VC per physical link, and of adding adaptivity.

Local communications as those in the BT application (BT is mainly composed of messages to neighbor nodes) neither benefit from using multiple virtual channels, nor from adaptivity, while patterns that interchange frequent long distance messages can take advantage of using several virtual channels as in the IS application (which is implemented using non-optimized N-to-N collectives), because it reduces the head of line blocking.

Furthermore, adaptivity is useful to balance the use of network resources, explaining why it helps CG but not IS. CG is composed of long chains of messages with a high degree of causality that should not saturate the network, but in the oblivious case these message compete intensely for resources in the X axis, which becomes a bottleneck. Adaptive routing is beneficial to balance the traffic along axes. In contrast, IS makes an homogenous use of the network that cannot be further improved by adaptivity.

3.5. Evaluation of network-level congestion control

Congestion is a well-known problem in computer networks that also affects INs. It appears when injection pressure goes beyond what the network is able to cope with. Its negative effects include reduced throughput and increased network latency. It can have its origin in any part of the IN but, if not adequately addressed, it soon propagates to the whole network. For example, if a network router has all the buffers in a dimension full, but new packets still arrive (from the attached node or from neighbor routers), congestion will propagate to nearby routers as they fill up their buffers as well.

The influence of network congestion control in the IN is one of the problems we have studied more thoroughly, using different workloads, topologies and applications (see Chapters 10 and 12). We used two different network level congestion control mechanisms based on locally available information.

In-transit Priority Restriction (IPR) gives priority to in-transit traffic. For a given fraction *P* of cycles, injection of a new packet is only allowed if it does not compete with packets already in the network. *P* may vary from 0 (no restriction) to 1 (absolute priority to in-transit traffic). This is the method applied in IBM's BG/L torus network [BCC03], while the Alpha 21364 network [MBL01] incorporates the similar concept in its "rotary rule".

The Local Buffer Restriction (LBR) mechanism has been designed specifically for adaptive routers that rely on Bubble Flow Control to avoid deadlock in the escape sub-network [PIGB01]. The bubble restriction also provides congestion control for the escape sub-network [IMG06]. LBR extends this mechanism to the rest of the VCs. That is, a packet can only be injected into an adaptive VC if such action leaves room for at least *B* packets in the transit buffer associated to that VC. Parameter *B* indicates the buffer space reserved for in-transit traffic.

These two congestion control mechanisms have been evaluated using **synthetic bursty traffic** (Figure 15), **traces** (Figure 16) and **full system simulation** (Figure 17). This allows us to have a wide vision of the performance of the mechanisms and also to cross-validate the trace-driven and the full system simulation as they were using the same applications. We can see how the results with the two kinds of simulations are very similar, if those from the full system simulation show attenuated differences because computation is considered in them.

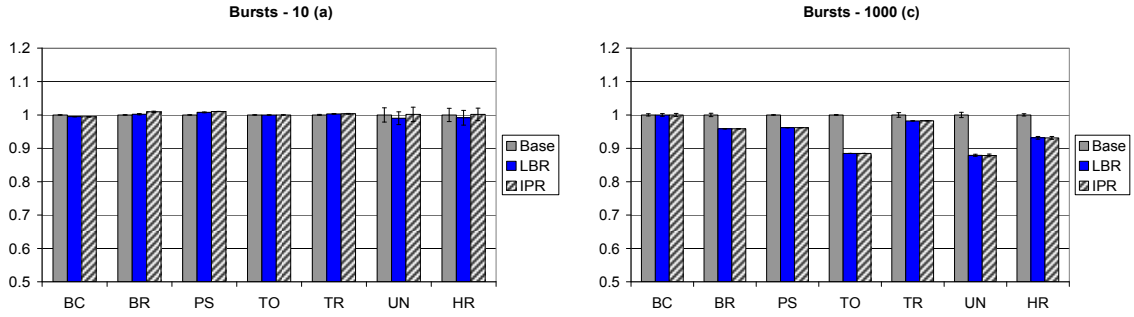


Figure 15: Performance for burst-synchronized traffic, for burst sizes 10, and 1000. Normalized times to consume a burst. Averages from 10 runs, and 99% confidence intervals.

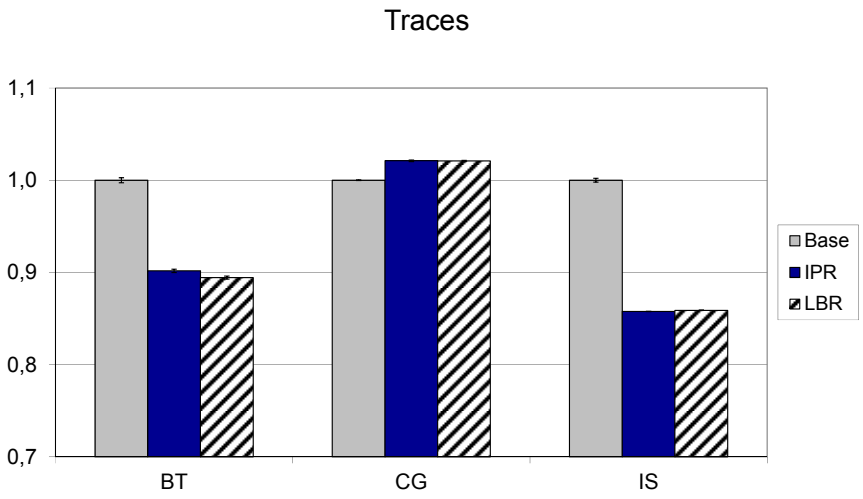


Figure 16: Trace driven simulation. Times to complete a run of BT, CG and IS, relative to Base case, without network congestion control, and 99% confidence intervals.

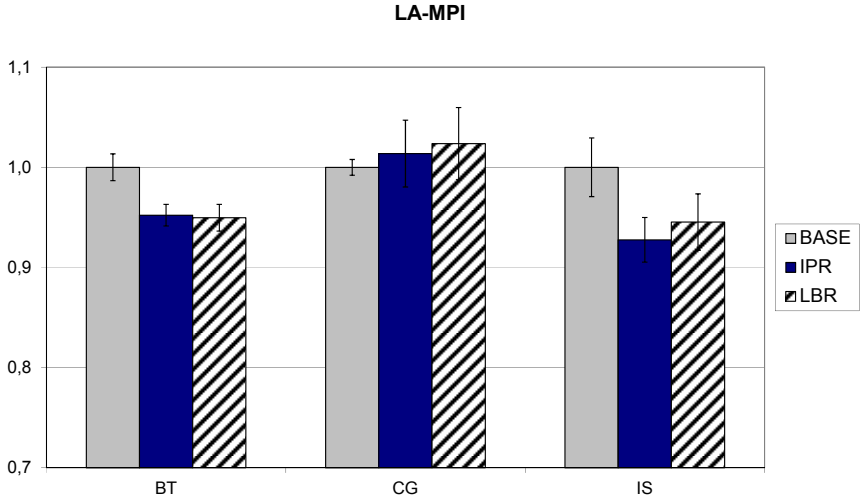


Figure 17: Results of the full system simulation using LA-MPI on a 1D ring of 64 nodes with a synchronization relation of 10000:200 Simics:FSIN cycles.

Bursty traffic simulations were done with different synthetic traffic patterns and different bursts sizes, that mimicked from very tightly-coupled applications (burst size 10) to loosely-coupled applications (burst size 1000). This was to study how the burst size influences the performance of the mechanisms.

Results show that tightly-coupled applications do not saturate the network, thus congestion control is ineffective or even harmful, because of the restrictions when injecting new packets into the network. CG, whose communication is local and formed by small messages, belongs to this group. In contrast, congestion control is beneficial for workloads with long messages (or bursts) that saturate the network as happens with BT (composed by large messages which generate some degree of saturation) and IS (composed by non-optimized all-to-all collectives that homogenously saturate the network).

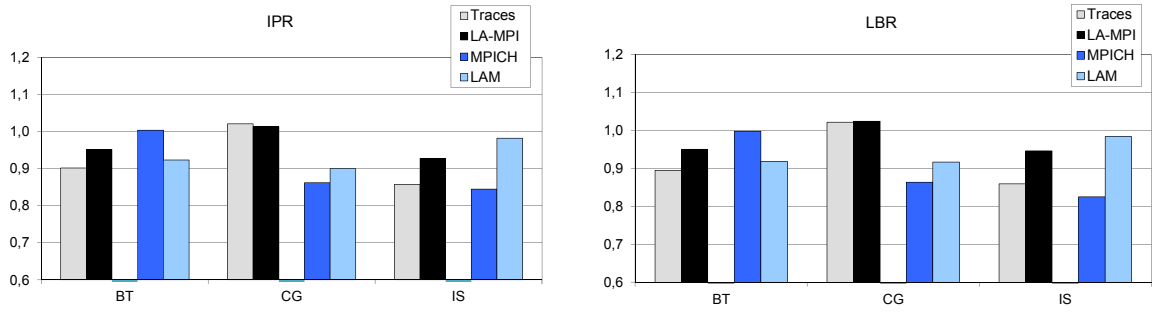
We can summarize from these experiments that congestion control is beneficial for most traffic patterns providing that the message size is large enough, and that when the network is not actually saturated the performance lost is almost negligible.

Again having a broad range of workloads to evaluate the performance of the mechanisms gave us the possibility of better understanding how these mechanisms influence the performance of the IN.

Precautions when using full-system simulation

3.6. Interaction of different levels of congestion control

When we first compared results obtained with trace based-simulation to those obtained using **full-system simulation** in [RMN07] using a protocol stack composed of MPICH/TCP/IP/Ethernet, we found a great disparity of results between the two configurations. A thorough analysis discovered unforeseen interactions between network level congestion control (as described in 3.5) and the end-to-end congestion control implemented in TCP.



a) b)

Figure 18: Measured execution times relative to the base case (IPR and LBR deactivated). a) Results obtained when IPR is activated. b) Results obtained when LBR is activated. Full-system simulation is done with a synchronization of 10000 Simics cycles every 200 INSEE cycles. Simulated network speed is 128Mb/s.

Network congestion control like IPR helped to reduce delay and jitter so TCP works better, and the flow of packets through the network is accelerated. However, without network-level congestion control like IPR, jitter is higher due to congestion produced at network so TCP retransmits packets and activates the slow start mechanism, causing severe performance drops. TCP is very sensitive to jitter, so in-transit traffic delays due to saturation of the network affect even more to TCP. We analyze these interactions in Section 7.2 of Chapter 10.

We compared trace-based workloads to full-system simulation using two different network level congestion control mechanisms, IPR and LBR. The experiments are shown in Figure 18 and confirmed our previous conclusions. We also introduced in the simulations other non-TCP protocol stacks to avoid unexpected interactions: LA-MPI over UDP and LAM over UDP.

LA-MPI performs error control at application level to avoid message losses when intermediate buffers are full. This error control barely affects the execution time. Despite this, the absence of TCP made LA-MPI a good platform for experimentation as it does not have end-to-end congestion control with slow start mechanism. However, LAM over UDP is much slower as shown in Table 3 because it routes all messages through a daemon present in every node, thus introducing two additional hops to every message. In addition LAM also includes an additional congestion control and error recovery similar to that of TCP which, again, invalidated our results.

	BT	CG	IS
MPICH	4.52s	5.89s	4.21s
LA-MPI	4.51s	5.60s	4.10s
LAM/UDP	5.09s	10.88s	10.38s

Table 3: Simulated time needed to run an iteration of each benchmark for different MPI implementations in the full-system simulation environment. Average of 10 runs.

The corollary is that although reutilization of components like protocol stacks reduces the time of setting up the evaluation environment and reduces the programming errors, it may introduce unforeseen interactions that may affect or even invalidate the results obtained.

3.7. Relation of network speed and congestion control interactions

We wondered if interactions between end-to-end congestion control and network level congestion control were affected by network speed. So we repeated simulations in Chapter 10, Section 7.3, under two different network speeds with MPICH over TCP with and without network-level congestion control.

We can summarize that an increase in network speed makes TCP perform even worse when there is no network-level congestion control mechanism. As we have seen in the previous case study 3.6, network-level congestion control reduces jitter and TCP can properly estimate its timers, delivering traffic without delays. But when network speed is increased, variance of packet arriving times is also increased, therefore the higher the network speed, the worse the performance of TCP in a network without network-level congestion control. As we can see in Figure 19a) from Chapter 10, Section 7.3, effects of network congestion control mechanisms have more incidence reducing jitter at higher network speeds. We can see in Figure 19b) the huge difference of applying congestion control on the network for CG and IS, the two most communication intensive applications, while in Figure 19a) the difference is far smaller. These results mean that TCP behaves worse in faster networks without network congestion control, that is, with large jitter.

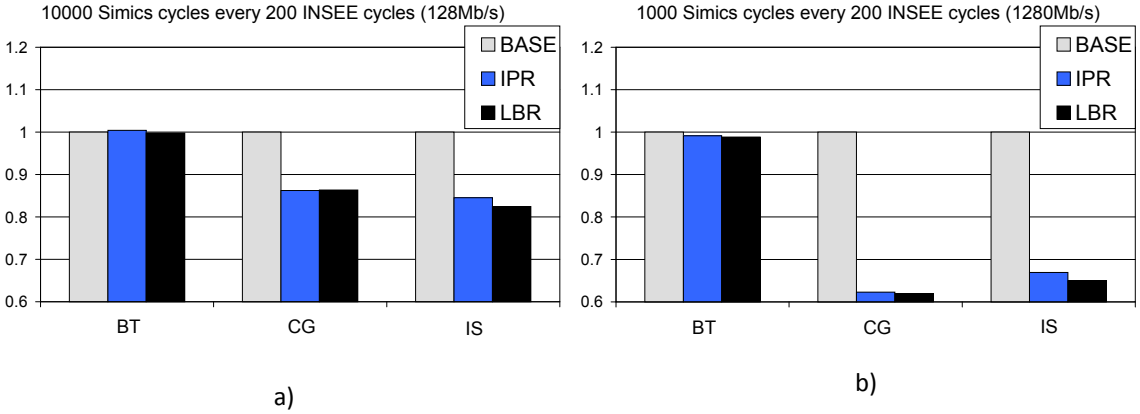


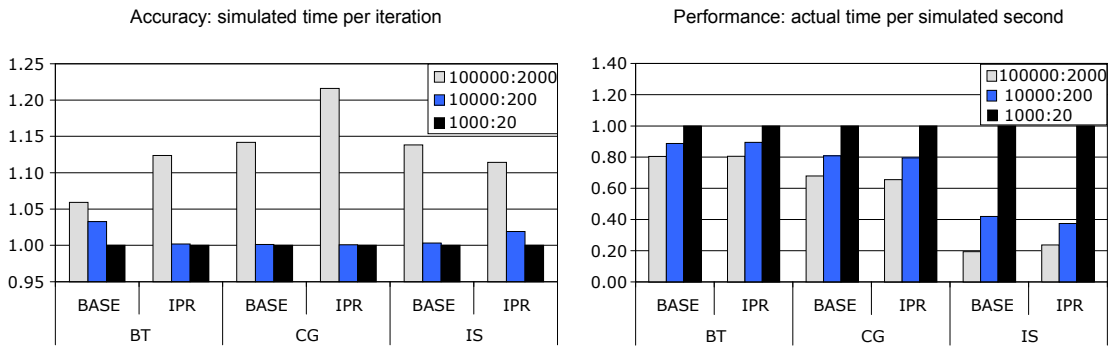
Figure 19: Measured execution times relative to base case (IPR and LBR deactivated).
a) Obtained in a 128Mb/s network. b) Obtained in a 1280Mb/s network

3.8. Effects of synchronization in full system simulation

INSEE connects to Simics when using **full-system simulation** set-ups. The use of this kind of workload needs to link two different simulators, and one of the main issues is how to synchronize them so that the synchronization does not consume excessive resources and the accuracy is kept at its maximum. We need to define the period of time a simulator runs without synchronizing with the others; we define this period of time as *slice* duration. The length of the slice affects both accuracy and simulation performance: the longer the slice, the better the performance, because fewer resources and time are used in synchronization. However, accuracy is negatively affected, because additional delays are introduced, packets have to wait to be injected in the network until the next synchronization slice. On the contrary,

when a short slice is chosen there is more synchronization overhead so performance is worse but the accuracy is better because simulation-induced delays are smaller.

Figure 20 captures the trade-off between accuracy and simulation speed. This figure is thoroughly explained in Chapter 10 Section 7.4. We fixed network speed and compared three different synchronization scenarios, from the largest slice: 100000:2000, to the shortest: 1000:20, being in the first case 100000 the number of Simics cycles run before synchronization and 2000 the number of INSEE cycles. When synchronization is infrequent, as in the 100000:2000 case of Figure 20a), huge delays are artificially introduced, as we force packets to wait up to 99999 (500µs) before being injected into the IN. With these unstable, long delays TCP cannot make good estimations of delays, and activate too often the slow-start mechanism, worsening the accuracy of the simulation. In Figure 20b) we can see the actual time to simulate a second of each benchmark. Moreover, if the synchronization slice is large, both simulators run alone long time without stopping to synchronize, thus adding very little additional execution time due to synchronization. However if the slice is very short, simulators stop many more times to synchronize, adding a lot of sync time, as we can see in Figure 20b), where the larger slice runs the IS simulation in only 20% of the time needed if we compare it with the shortest slice.



a) b)

Figure 20: Time measurements for each benchmark with different synchronization parameters. Results relative to the most synchronized case (1000:20) a) Simulated time to complete each iteration. b) Actual time to simulate a second

In our case, the 10000:200 slice appears to be a good choice, because we only loose a 3% on accuracy on the worst case, and the simulation lasts up to 60% less time than the case with the shortest synchronization slice.

We conclude this case study stating that it is essential to study the speed versus accuracy trade-off when linking different simulators into one complex system that must be synchronized.

Chapter 4. Contributions and future work

This work is centered on traffic generation for IN evaluation, and on the implementation of different workload models on the INSEE simulator. INSEE was designed to address several shortcomings detected in other tools: most previous tools were unable to model large INs, because of the huge amount of resources needed; furthermore, no simulation environment was found in the literature able to deal with a large diversity of workload alternatives.

While our research on IN progressed, INSEE backed it up, as more functionalities were added in order to support the changing requirements of the new experiments. INSEE is now a powerful simulator that provides multiple configurations allowing simulating a wide range of IN alternatives. It also provides very detailed output reports to explore events during simulation and after it.

The author has contributed to the development and design of INSEE from its very beginning, both to FSIN (the simulator itself) and especially to the TrGen module (the one managing traffic generation). He is responsible for its structured design, which makes it versatile and easy to add new functionalities, such as new topologies, routing strategies, congestion control techniques or workloads.

Our simulator incorporates a wide range of workloads to feed simulation, from the very first design phases, with fast statistical distributions, bursty traffic, parallel application micro-kernels, trace-based traffic and full-system simulation, ordered from lower to higher fidelity to real application traffic. Each of these workloads has proven useful to show different properties of the networks under simulation. Important results summarized in this dissertation would never have been obtained using classical workloads based on statistical distributions.

The author has been deeply involved in the development of traffic generation, especially on trace-based traffic and full-system, making the interfaces to integrate the external modules like Simics or the MPI log processing system.

INSEE has contributed to the advancement of the state-of-the-art in the field of IN. It has been used to research different topics, such as congestion control, throughput fairness, new IN topologies or parallel scheduling algorithms. Other research groups around the world have cited our tool, and they have used it to conduct their own investigations.

In the dissertation the author has demonstrated the usefulness and importance of choosing the right workloads in the simulation of IN by means of a large collection of case studies which include a wide range of traffic generation models. The author has designed several new methods with varying degrees of realism and resource requirement: full-system simulation, trace-driven simulation, application micro-kernels and synthetic traffic based on bursts.

These new traffic generation methods have been implemented into our IN simulator INSEE. The accuracy analysis of these methods in relation to actual traffic has been studied. The traffic generation methods implemented are more realistic, some are more affordable and resemble better the real way scientific applications behave and interchange messages, representing causality and its coupling. Moreover, workloads can be used to test topologies that don't even exist, and full-system simulation allows injecting real traffic to feed the simulation.

We can summarize the main contributions of this dissertation as follows:

Traffic generation methodologies and implementation

- Full-system, as the mechanism to achieve the ultimate fidelity in the simulation of actual compute nodes running actual applications. The author was the responsible for the design of the methodology, the full implementation into TrGen, and all experimental work done within our research group based on this kind of traffic.
- Trace-driven, to take into account the causality among messages of actual applications, and to predict performance of an application under different IN designs. Even though some methodologies to capture and process traces already existed, we highlighted common pitfalls and designed a rigorous methodology which extracts more information from the applications while complying with MPI standards. The author was in charge of the part of the INSEE injecting and processing the traces, and also of all the experimental work discussed in this dissertation.
- Micro-kernels, to overcome the lack of flexibility and problematic procurement of traces, while keeping into consideration the communication characteristics of a large set of real applications. The author collaborated in their design, and also carried out the experimental parts of this dissertation in which this kind of workload was used.
- Bursts, to add to synthetic workloads the capability of emulating barrier style synchronization, at different levels of granularity. The author implemented this traffic generation mechanism into the simulator, and performed all experiments based on this workload that are described in this thesis.

Research on topologies:

- A new methodology for predicting performance of applications on IN has been designed. This methodology can be used to predict performance on topologies that actually don't exist.
- Topologies have a great influence on performance, which depends on the application being run. Some applications that fit perfectly on a topology, running at its maximum

performance, suffer from additional delays when running on different networks, due to application-induced bottlenecks.

- Several topologies have been studied, both direct and indirect networks, developing novel ones like twisted torus or thin trees, and researching on well-known INs such as midimew, k-ary n-cube meshes, SpiNNaker, or k-ary n-trees. The author has been especially involved in the design and evaluation of thin threes.

Network-level policies and mechanisms:

- The influence of routing strategies and virtual channel management on the performance of IN has been studied.
- Large radix networks with multiple injection sources are prone to congestion. We have described several studies about congestion control mechanisms on large IN, and their mostly beneficial effect on the IN. Congestion control mechanisms reduces network bottlenecks, reduces latency and jitter, and keeps steady performance when network traffic is over the saturation level.

Precautions when using full-system simulation:

- Network-level congestion control can interfere with end-to-end congestion control (such as that implemented by TCP), hindering or exaggerating performance results, so all components in the simulation must be carefully chosen to avoid this kind of interferences.
- When network speed increases, interactions between network-level and end-to-end congestion control become larger, because jitter also increases and protocols to avoid congestion activate often and unnecessarily, reducing performance.
- The synchronization of different simulators, INSEE for the IN, and Simics for the compute nodes, makes a trade-off between accuracy and simulation speed. If the time between synchronizations is long, the accuracy is worse, as latency is added to the network, and simulation speed increases because there is less time lost due to synchronizations. And the same is true but reversed, if time between synchronizations is short.

For the future we plan to further develop INSEE to continue supporting our research on this field of knowledge, increasing its functionalities with new topologies and router designs, and enhancing our traffic generation mechanisms to improve our simulations and the simulator performance. These tasks will be simplified by the modular design of INSEE.

Chapter 5. List of publications

Throughout the development and writing of this dissertation the following list of journal papers and conference presentations has been published, being in **bold** those that have been included in Part 2 of this thesis.

5.1. International Journals

Javier Navaridas, José Miguel-Alonso, Jose Antonio Pascual, Francisco Javier Ridruejo. "Simulating and evaluating interconnection networks with INSEE". Simulation Modelling Practice and Theory 19(1): 494-515 (2011)

Javier Navaridas, Jose Miguel-Alonso, Javier Ridruejo, Wolfgang Denzel. "Reducing Complexity in Tree-like Computer Interconnection Networks". Parallel Computing 36 (2010) 71–85. doi:10.1016/j.parco.2009.12.004

F.J. Ridruejo, J. Miguel-Alonso, J. Navaridas. "Full-System Simulation of Distributed Memory Multicomputers". Cluster Computing. Volume 12, Number 3, September 2009, pages 309-322. DOI 10.1007/s10586-009-0086-y.

J. Miguel-Alonso, J. Navaridas, F.J. Ridruejo. "Interconnection network simulation using traces of MPI applications". International Journal of Parallel Programming. Volume 37, Issue 2 (2009), pages 153-174. DOI 10.1007/s10766-008-0089-y

5.2. International conferences with peer-review

Javier Navaridas, Jose Miguel-Alonso, Francisco Javier Ridruejo. "On synthesizing workloads emulating MPI applications". The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08). April 14-18, 2008, Miami, Florida, USA.

F. J. Ridruejo, J. Navaridas, J. Miguel-Alonso and Cruz Izu, "Realistic Evaluation of Interconnection Network Performance at High Loads", 8th Int. Conf. on Parallel and Distributed Computing Applications and Technologies - PDCAT 2007, Adelaide, Australia, 3-6 December 2007.

Fco. Javier Ridruejo, Jose Miguel-Alonso, Javier Navaridas. "Concepts and Components of Full-System Simulation of Distributed Memory Parallel Computers". Proc. HPDC'07, June 25–29, 2007, Monterey, California, USA.

Javier Navaridas, Fco. Javier Ridruejo, Jose Miguel-Alonso. "Evaluation of Interconnection Networks Using Full-System Simulators: Lessons Learned". Proc. 40th Annual Simulation Symposium, Norfolk, VA, March 26-28, 2007.

F.J. Ridruejo, J. Miguel-Alonso. "INSEE: an Interconnection Network Simulation and Evaluation Environment". Lecture Notes in Computer Science, Volume 3648 / 2005 (Proc. Euro-Par 2005), Pages 1014 - 1023.

F.J. Ridruejo, A. Gonzalez, J. Miguel-Alonso. "TrGen: a Traffic Generation System for Interconnection Network Simulators". International Conference on Parallel Processing, 2005. 1st. Int. Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems (PEN-PCGCS'05). ICPP 2005 Workshops. 14-17 June 2005 Page(s): 547 – 553

5.3. National conferences

F. Javier Ridruejo, José Miguel-Alonso y Javier Navaridas. "Full-system simulation of distributed memory parallel computers using Simics". Actas XVIII Jornadas de Paralelismo. Zaragoza, Septiembre 2007.

F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso, C. Izu. "Evaluation of Congestion Control Mechanisms using Realistic Loads". Actas de las XVII Jornadas de Paralelismo. Albacete, Septiembre 2006.

F.J. Ridruejo, J. Miguel. "Simulación de redes de interconexión utilizando tráfico real". Actas de las XVI Jornadas de Paralelismo. Thomson, 2005 (ISBN 84-9732-430-7). Pages: 109 - 116.

F.J. Ridruejo Pérez, A. González Castro, J. Miguel-Alonso. "TrGen: sistema de generación de tráfico para simuladores de redes de interconexión". Actas de las XV Jornadas de Paralelismo. Almería, Septiembre 2004.

J. Ridruejo, J. Agirre, J. Miguel-Alonso. "Estrategias de instalación y gestión de clusters con software libre". Actas de las XIV Jornadas de Paralelismo. Leganés (Madrid), Sept. 2003.

Chapter 6. Summary in Spanish – Resumen

Esta tesis presenta el trabajo del autor en el área de herramientas y metodologías para la evaluación y predicción del rendimiento de aplicaciones paralelas en sistemas de computación de altas prestaciones, especialmente en computadores paralelos de memoria distribuida (supercomputadores).

El trabajo descrito en esta tesis se ha desarrollado dentro del grupo de investigación al que pertenece el autor, el Grupo *Intelligent Systems Group* de la Universidad del País Vasco UPV/EHU. Partes considerables del trabajo han sido también desarrolladas en colaboración con miembros del mismo grupo, aunque en esta tesis nos centramos en las contribuciones desarrolladas por el autor.

El trabajo trata sobre un tipo particular de arquitectura de computadores: los multicomputadores, computadores paralelos de memoria distribuida con paso de mensajes entre los nodos. Ejemplos de estos multicomputadores pueden encontrarse entre los supercomputadores y los clusters con mayor potencia, normalmente dedicados al cálculo científico. En particular, esta tesis se centra en un importante componente de los multicomputadores, la Red de Interconexión (IN) que permite comunicarse y sincronizarse a las aplicaciones paralelas que se ejecutan en los nodos de cómputo.

Los supercomputadores están compuestos de multitud de componentes. Esto hace que la evaluación de la mejora en uno de sus componentes sea difícil de realizar sin tener en cuenta todas las posibles interacciones con el resto del sistema. Existen diferentes métodos para realizar la evaluación de un supercomputador, como los analíticos (cadenas de Markov, teoría de colas, redes de Petri), la simulación y la prueba empírica.

Nuestro grupo de investigación se centra en la evaluación de estos sistemas usando técnicas de simulación, ampliamente aceptadas como válidas por la comunidad científica. Se puede modelar con diferentes grados de fidelidad un sistema a estudiar, y analizar su comportamiento bajo circunstancias concretas, por ejemplo usando distintos componentes o diferentes cargas de trabajo. Un entorno de simulación debería ofrecer:

- Flexibilidad de modelado: posibilidad de modelar múltiples sistemas con diferentes características y tamaños.
- Múltiples cargas de trabajo: posibilidad de aceptar variados tipos de tráfico para alimentar las simulaciones.

- Disponibilidad de mecanismos de medida: diferentes tipos de información y métricas, desde tiempo de ejecución hasta una traza detallada de todos los mensajes intercambiados en la red de interconexión por todos los procesos.

En esta tesis se presta especial atención a los diferentes tipos de cargas de trabajo que pueden ser generadas y usadas en la simulación de los sistemas paralelos. Las herramientas y metodologías presentadas han sido integradas en la herramienta INSEE (*Interconnection Network Simulation and Evaluation Environment – entorno de simulación y evaluación de redes de interconexión*). INSEE recopila la más versátil y completa colección de herramientas y técnicas para la evaluación de multicomputadores que conocemos hasta este momento.

Usando INSEE podemos evaluar diferentes tipos de arquitecturas de multicomputadores y topologías de redes de interconexión, con diferentes tipos de detalle y niveles de fidelidad, con un amplio rango de opciones de cargas de trabajo, desde la más simple hasta la más realista. Además, INSEE proporciona medidas detalladas de cualquier aspecto de la simulación.

A pesar de su alta precisión, INSEE tiene un consumo muy reducido de recursos de computación con ciertas cargas de trabajo, como las basadas en distribuciones estadísticas o los micro-núcleos de aplicación, lo que permite simular sistemas del tamaño de las primeras posiciones de la lista del TOP500 en ordenadores personales. Sin embargo, cuando usamos otro tipo de cargas de trabajo como las trazas de aplicaciones o la simulación de sistema completo, la utilización de recursos se incrementa linealmente con el número de nodos, tanto en uso de tiempo como memoria. Por ejemplo la simulación de sistema completo, debido a su detallada simulación de los nodos de cómputo usando Simics (se simulan sistemas operativos, drivers, y librerías MPI que ejecutan las aplicaciones paralelas sin modificar), requiere del uso de unos recursos parecidos a los del sistema que se simula, sufriendo una fuerte penalización en el tiempo de ejecución del sistema simulado comparado con el real.

INSEE ha sido usado frecuentemente en las investigaciones del grupo de investigación *Intelligent Systems Group* de la Universidad del País Vasco, así como por otros grupos de investigación de todo el mundo, por ejemplo: Universidad de Cantabria, Universidad de Castilla-La Mancha, *Shangai Normal University, The University of Melbourne, The University of Manchester* o *The University of York*.

Los estudios realizados por otros grupos de investigación se pueden clasificar en: estudios del uso de los recursos en las IN, causas y consecuencias de la congestión en IN y técnicas de control de la misma, nuevas topologías para IN, y algoritmos de planificación de ejecuciones paralelas.

6.1. Cargas de trabajo y generación de tráfico

La simulación puede usarse desde las primeras etapas de diseño de un multicomputador. En estas primeras etapas, es aconsejable usar un sistema de simulación que permita explorar tantas alternativas como sea posible de una manera rápida, aunque la precisión y fidelidad de los resultados no sea perfecta. Sin embargo, en las últimas fases de diseño es necesario asegurar la exactitud de las decisiones tomadas, por lo que se hace imprescindible un sistema

de simulación muy preciso. La mayor parte de las herramientas de simulación centran sus esfuerzos en modelar con exactitud el sistema evaluado, pero nuestra opinión es que la elección de las cargas de tráfico con las que se evaluará el sistema es también de suma importancia.

En nuestro contexto, entendemos por cargas de tráfico la colección de mensajes generados por las aplicaciones que se ejecutan en los elementos de computación y que recorren la red de interconexión. Estos mensajes pueden ser troceados en paquetes antes de ser inyectados en la red, y reordenados en los nodos destino.

TrGen es el componente de INSEE en el que se implementan todos los mecanismos de generación de tráfico disponibles. La caracterización del tráfico que la red de interconexión tiene que tratar es importante en cada fase del diseño. Un subsistema flexible de generación de tráfico, como TrGen, permite al investigador emular el tipo y cantidad de tráfico que tendrá que gestionar la red de interconexión, obteniendo como resultado un entorno magnífico para probar su rendimiento. Podemos dividir los métodos de generación de tráfico en cinco tipos según su implementación cronológica en INSEE.

6.1.1. Tráfico sintético

Son los mecanismos clásicos de generación de tráfico que podemos encontrar en la literatura, y que se basan en distribuciones de probabilidad o en permutaciones. Proporcionan cargas de tráfico útiles para evaluar rápidamente el rendimiento bruto de un multicomputador: máxima productividad, mínimo retardo, o nivel de saturación esperado. Otra ventaja es que los resultados obtenidos con este tipo de tráfico pueden ser validados con modelos matemáticos de las metodologías implementadas, por ejemplo, el análisis de la productividad para el tráfico uniforme y los parámetros relacionados con la distancia que viajan los paquetes en cada topología. Estas distribuciones de probabilidad se aplican normalmente al destino de los paquetes, el tamaño de los mismos, y los tiempos entre su generación. Los parámetros de las distribuciones de probabilidad usados en los experimentos se pueden configurar o inferirse del tráfico real, mediante un proceso de ajuste de distribuciones.

6.1.2. Tráfico basado en trazas

El tráfico sintético es limitado y poco realista para ciertos estudios, principalmente por su imposibilidad de capturar las relaciones de causalidad entre los mensajes. Los procesos de una aplicación no sólo se comunican, sino que también se sincronizan, y pasan por diferentes fases con distintas relaciones de computación / comunicación. El tráfico basado en trazas obtenidas de la ejecución de aplicaciones reales es una aproximación más realista. Sin embargo, como se explica en el Capítulo 9, la captura y generación de trazas de aplicaciones no es un proceso trivial. Nuestra propuesta, integrada en TrGen, mejora el sistema estándar de captura de eventos de MPICH para generar una traza extendida usable en la simulación. También definimos una metodología para llevar a cabo estudios de predicción de rendimiento, de esta manera INSEE permite predecir el tiempo de ejecución de una aplicación en una arquitectura diferente de la que se usó para capturar las trazas.

Las trazas también presentan varios problemas relacionados con su falta de flexibilidad: una traza representa la ejecución de una aplicación en un sistema específico usando un número de procesadores y unos datos de entrada concretos, teniendo implícitas algunas de las características del sistema en el que han sido obtenidos. Así pues, no pueden ser directamente usadas para llevar a cabo estudios de escalabilidad y podrían ser inexactas debido a la interferencia de los mecanismos que las registran. Por último, las trazas de grandes multicomputadores son difíciles de obtener, lo que dificulta nuestros estudios de evaluación de rendimiento.

6.1.3. Simulación de sistema completo

Es el método de generación de tráfico más completo, y fue introducido para suplir las carencias del sistema basado en trazas. Este método analiza un sistema simulando todos sus componentes: los nodos de cómputo, la red de interconexión, sistemas operativos, drivers, protocolos de comunicaciones, librerías MPI y las aplicaciones paralelas que se ejecutan en estos sistemas que inyectan tráfico en la red de interconexión simulada. La comunicación entre los nodos es real y todas las relaciones causales, sincronización, los ritmos de inyección y los tiempos de computación son aquellos de la aplicación real siendo ejecutada en el sistema simulado.

Nuestro sistema de simulación completa se basa en la simulación de los nodos de cómputo con Simics, integrándolos con INSEE mediante el módulo TrGen. La comunicación de red producida en los nodos de cómputo Simics se inyecta en INSEE y se dirige a su nodo de destino. Todos los nodos de cómputo y sus respectivos elementos de comunicación se mantienen sincronizados.

La simulación de sistema completo es propensa a errores, debido a los múltiples componentes que interactúan entre sí y que se tienen que ajustar con mucho cuidado. De otra manera, el potencial beneficio de un diseño puede ser escondido por interacciones inesperadas con los elementos de la simulación – o, por el contrario, exagerada. Otro tema a considerar es la importante cantidad de recursos necesarios para simular un multicomputador.

6.1.4. Tráfico a ráfagas

Este tipo de tráfico se introdujo en TrGen debido a las dificultades de conseguir trazas de grandes multicomputadores y a la imposibilidad de simularlos al completo. Es una evolución de los modelos de tráfico sintéticos introduciendo diferentes niveles de sincronización. Emula la sincronización de grano grueso mediante ráfagas, introduciendo barreras después de cada inyección de ráfaga: cada nodo inyecta una ráfaga de paquetes y entonces espera hasta que todas las ráfagas hayan llegado a su destino. El tamaño de las ráfagas es configurable, pudiendo simular de esa manera diferentes tipos de comportamientos de aplicación.

6.1.5. Micro-núcleos de aplicación

El tráfico a ráfagas introduce en las cargas puntos de sincronización tipo barrera. Sin embargo, hay muchas aplicaciones que usan modelos más complejos de sincronización entre los nodos, de grano mucho más fino. Una investigación detallada sobre las aplicaciones más

usadas en supercomputación nos desveló que hay algunos patrones de comunicación y sincronización de uso frecuente. Capturamos algunos de esos patrones en forma de micro-núcleos de aplicación. Un micro-núcleo proporciona un patrón de destinos junto con las causalidades entre los mensajes, y puede ser configurado con un número arbitrario de nodos y tamaño de mensajes. Podemos entenderlos como cargas de trabajo basadas en trazas cortas, pero con una flexibilidad mucho mayor, porque se pueden adaptar a diferentes tamaños e intensidades de comunicación.

6.2. Casos de estudio

INSEE ha sido usado en muy diversos trabajos de investigación relacionados con las redes de interconexión. En el marco de esta tesis nos vamos a centrar en una colección de estudios en los que se utilizan distintos tipos de tráfico para la realización de la parte experimental. Los casos de estudio seleccionados se agrupan en: investigación sobre topologías de red; mecanismos y políticas aplicados a nivel de red para mejorar su rendimiento; y decisiones a tomar cuando se hace simulación de sistema completo, interacciones y problemas a tener en cuenta.

6.2.1. Investigación sobre las topologías

Hemos realizado estudios sobre las características topológicas de diferentes redes de interconexión, directas o indirectas, analizando sus rendimientos mediante estudios teóricos y simulación en redes de tipo *toro retorcido, midimew, King*, cubo indirecto y *árbol adelgazado*. Estudiamos principalmente el impacto de la topología en el rendimiento de la red de interconexión usando diferentes tipos de cargas de tráfico. Una mejora en el diseño de la topología debería de trasladarse en la misma medida a una mejora en el rendimiento de las aplicaciones que se ejecutan sobre ella. Sin embargo, hay casos en los que esta relación no es directa, como se muestra en el caso del análisis sobre los *árboles adelgazados*, donde se explica cómo una reducción drástica de la complejidad de una topología tipo *n*-árbol *k*-ario se traslada en una pequeña diferencia en el rendimiento de las aplicaciones, pero en diferencias substanciales en cuanto a gastos de construcción y mantenimiento. Estos resultados se pueden observar bajos determinados tipos de tráfico, pero no mediante los tradicionales estudios mediante tráfico estadístico o teóricos analizando el tráfico máximo soportado por la bisección.

También se han realizado estudios sobre cómo evolucionan las cargas de tráfico a bajo nivel, según la topología en la que se procesan, llegando a demostrar que, en algunos casos, la carga de tráfico, su planificación y posicionamiento con respecto de los nodos de la red puede ser beneficiosa para la ejecución de una aplicación, o por el contrario puede ser perjudicial, prolongando el tiempo de ejecución. Los retrasos inducidos por las comunicaciones pueden ser provocados por el mal ajuste de la carga a la topología de la red, de manera que los mensajes tengan que recorrer una mayor distancia, por ejemplo, haciendo que una gran parte del tráfico atraviese los enlaces periféricos en un toro o suba hasta el nodo raíz de un árbol, aumentando la probabilidad de saturación.

6.2.2. Políticas y mecanismos de red

Este conjunto de casos de estudio se centra en los mecanismos y políticas que se aplican a la red con intención de mejorar su rendimiento. Por ejemplo, se realiza un estudio sobre la influencia del uso de canales virtuales en las redes de interconexión, que concluye que reducen el efecto de contención que produce el bloqueo de primero de línea, y se analizan las diferencias entre estrategias de encaminamiento adaptativo y no adaptativo, siendo el adaptativo beneficioso para distribuir de forma más homogénea la carga por la red.

Por otro lado se hace un profundo estudio de la congestión, sus efectos sobre la red, y las técnicas para controlarla. La congestión es un problema conocido en las redes de computadores que afecta también a las redes de interconexión. Surge cuando la presión de la inyección en la red es mayor de la que la red puede tratar. Se traduce en un aumento de la latencia y una menor productividad. Puede surgir en cualquier punto de la red y propagarse por toda ella. Los multicomputadores en la actualidad son mucho mayores y los nodos que contienen más potentes, con múltiples núcleos, múltiples puertos de inyección, y con varios canales virtuales por puerto de inyección, aumentando la cantidad de tráfico que un nodo puede inyectar en la red; lo que hace que sus redes de interconexión sean aún más propensas a la congestión.

Hemos realizado estudios sobre el efecto de la congestión y la aplicación de dos técnicas locales de control de la misma, IPR y LBR, usando diferentes métodos de generación de tráfico desde el sintético, ráfagas, trazas, hasta la simulación de sistema completo. En el Capítulo 10 usamos INSEE para evaluar dos técnicas de control de congestión locales usando simulación de sistema completo. En el Capítulo 12 evaluamos estas dos técnicas usando diferentes modelos de generación de tráfico. La conclusión general es que el control de congestión beneficia en la mayoría de modelos de comunicación de aplicaciones, especialmente cuando éstas son capaces de saturar la red, y que también introducen un muy pequeño retraso en aquellas ejecuciones de aplicaciones que no tienen un componente de comunicación tan acusado, no llegando a saturar la red.

6.2.3. Precauciones a tener en cuenta en la simulación de sistema completo

En este conjunto de casos analizamos en profundidad todo lo relacionado con las cargas de tráfico basadas en la simulación de sistema completo. En el transcurso del diseño de nuestro modelo de generación de este tráfico, nos dimos cuenta de que el control de congestión de extremo a extremo realizado por protocolos como TCP podía producir interferencias indeseables en los experimentos, al activar su mecanismo de comienzo retardado que reduce la inyección, añadiendo retrasos innecesarios en la ejecución. El control de congestión a nivel de red ayuda a reducir la variabilidad en los retardos de los mensajes, lo que se traduce en una drástica reducción del uso de este mecanismo. Confirmamos las conclusiones obtenidas haciendo experimentos con otras pilas de protocolos MPI, con control de congestión de extremo a extremo diferente, como LAM sobre UDP, o inexistente, como en el caso de LA-MPI.

El aumento de la velocidad de las redes también afecta a la fidelidad de los resultados obtenidos en las simulaciones, ya que una red más rápida en la que existe congestión la

incidencia del control de congestión es aún mayor, por el aumento de la fluctuación en los retardos, que activa de forma no deseada retransmisiones y activación del mecanismo de comienzo retardado en TCP. Esto se evita en redes que, siendo rápidas, incorporan sus propios mecanismos de control de congestión y, por tanto, mantienen a raya la fluctuación.

De forma similar, la sincronización realizada entre los nodos simulados por Simics y la red de interconexión simulada por INSEE también influye en la fidelidad de la simulación, así como en el tiempo de ejecución de la misma, de manera que hay que establecer un compromiso entre fidelidad y rendimiento. Cuanto mayor es el período entre sincronizaciones, mayor es el tiempo que ambos simuladores pasan sin comunicarse y por ello mayor es el retardo de los paquetes que esperan a ser inyectados en la red en la próxima ejecución, perjudicando así a la fidelidad de los resultados. Por el contrario, cuanto menor es el tiempo entre sincronización de ambos tipos de simuladores, más tiempo se dedica a labores de sincronización, por lo que el tiempo de ejecución aumenta, aunque la fidelidad aumenta al no introducirse retrasos adicionales.

6.3. Estructura de esta tesis

Una vez introducidos el contexto y propósito de la tesis, pasamos a presentar su estructura, y cómo encaja cada capítulo en el contexto global de la tesis. Este documento ha sido diseñado como una compilación de publicaciones científicas relevantes del autor, pero no incluye todos los artículos publicados (estos se pueden consultar en el Capítulo 5), sino sólo una selección de los aspectos más importantes que el autor ha estudiado, tratando de evitar redundancias innecesarias. Se estructura de la siguiente manera:

6.3.1. La simulación como una herramienta para evaluar computadores paralelos. INSEE

Este artículo describe INSEE y los módulos que lo constituyen: el simulador de redes de interconexión FSIN, y el generador de tráfico TrGen. La mayor contribución de este artículo en el contexto de esta tesis es el resumen de los métodos de generación de tráfico que se incorporan a TrGen, con una explicación en profundidad de los métodos de generación de tráfico sintético.

INSEE es la herramienta clave de la investigación que se ha realizado y por ello es de crucial importancia conocer su estructura y capacidades, para entender qué tipo de estudios se han realizado con ella, incluyendo los que se incluyen en esta tesis.

En este artículo el autor ha contribuido al diseño e implementación de INSEE, incluyendo sus dos módulos, y las interfaces con terceros sistemas como Simics, o diferentes tipos de generación de tráfico, desde trazas a simulación de sistema completo.

6.3.2. Cargas de tráfico basadas en trazas

En este artículo nos centramos en el mecanismo para producir tráfico basado en trazas de aplicaciones reales y alimentar con ellas nuestras simulaciones, así como en las limitaciones de este tipo de carga. La principal ventaja del tráfico basado en trazas es que respeta la causalidad

de los mensajes intercambiados en la red, reflejando fielmente el funcionamiento de las aplicaciones paralelas.

Se muestran dos casos de estudio realizados con este tipo de tráfico: la evaluación de estrategias de encaminamiento y gestión de canales virtuales, y la estimación de rendimiento de una aplicación real en tres tipos distintos de multicomputadores. También hemos usado este mecanismo de generación de tráfico para estudiar nuevas propuestas de topologías.

La generación de tráfico basada en trazas también tiene limitaciones: sólo representa una de las ejecuciones posibles, está muy ligada al sistema en el que se han obtenido, y pueden haber interferencias con otras aplicaciones que se estén ejecutando en ese momento en la red; además, las trazas de grandes multicomputadores son complicadas de obtener.

El autor ha trabajado en la captura de los ficheros de trazas extendidos de un MPICH modificado y su procesamiento necesario para poder usarlos por el módulo TrGen. Además ha realizado los trabajos experimentales para estos casos de estudio.

6.3.3. Simulación de sistema completo

Para superar las limitaciones de las trazas decidimos añadir más realismo a la simulación incluyendo un módulo en TrGen para realizar simulación de sistemas completos. Este sistema ejecuta las aplicaciones paralelas sin modificación alguna sobre un sistema real que se comunica sobre una red de interconexión simulada. Los nodos son simulados sobre Simics, un simulador de sistema completo.

En el momento de escribir este artículo los sistemas de simulación completa tenían muy poco detalle en sus nodos o en sus redes para nuestras necesidades. Este artículo muestra las decisiones tomadas y los problemas que surgieron, así como el compromiso entre fidelidad en la simulación y tiempo de ejecución, y las interacciones no deseadas surgidas entre los controles de congestión a nivel de red y a nivel de nodo (TCP).

Con este sistema hemos comprobado con gran fidelidad varias propuestas de control de congestión a nivel de red de interconexión, para comprobar cómo afectan a protocolos como TCP, y caracterizar las interacciones entre los muchos elementos de simulación.

El autor ha realizado todos los detalles de la simulación de este tipo de generación de tráfico basada en la simulación de sistema completo, incluyendo diseño, implementación, y la integración entre TrGen y Simics. También ha sido responsable de todo el trabajo experimental realizado. Se han llevado a cabo dos casos de estudio: control de congestión a nivel de red, y su interacción con el implementado por TCP de extremo a extremo.

6.3.4. Micro-núcleos de aplicación

Posteriormente, se refinó el modelo de generación de tráfico sintético para fuese más realista, definiendo una serie de micro-núcleos usados por aplicaciones paralelas reales. Estos micro-núcleos reflejan el esquema de intercambio de mensajes (incluyendo la causalidad entre mensajes) de partes comunes a muchas de las aplicaciones paralelas más utilizadas. Son

sintéticos puesto que pueden producirse arbitrariamente en tamaño y no necesitan ejecutar ninguna aplicación real (lo que sí es necesario para obtener trazas).

En este artículo, además del diseño e implementación de este tipo de tráfico, se ilustra su utilidad mostrando la relación directa que tiene la topología con el patrón de comunicación de la aplicación ejecutada. Puede darse el caso de que el patrón de tráfico se ajuste perfectamente a la topología de red, con lo que la aplicación se ejecuta rápidamente y sin sufrir retrasos adicionales, o en el caso opuesto, hacer que su tráfico provoque un cuello de botella en la red debido a que la topología sobre la que se ejecuta haga que se concentre éste en unos pocos enlaces congestionados. Asimismo, se muestra la evolución temporal del tráfico para estas aplicaciones, mostrando cómo los recursos de red usados varían en el tiempo.

El autor de la tesis es responsable de parte del diseño de este tipo de carga de trabajo, de los experimentos y sus conclusiones.

6.3.5. Caso de estudio 1: Control de congestión en redes directas

Nuestro principal interés investigador está en los grandes multicomputadores, de los que es difícil conseguir trazas, y no es posible su simulación de sistema completo, así que diseñamos otro sistema de generación de tráfico sintético que imitaba el comportamiento de aplicaciones paralelas que se sincronizan mediante barreras, lo llamamos generación de tráfico basada en ráfagas.

En este artículo mostramos su diseño y validamos su utilidad en un estudio sobre alternativas para el control de la congestión en la IN, siendo suficientemente fiable para evaluar decisiones de diseño de redes de interconexión comparándolo con la generación de tráfico basada en trazas.

El autor realizó la implementación del tráfico basado en ráfagas dentro del módulo TrGen, los experimentos de simulación y la extracción de las conclusiones.

6.3.6. Caso de estudio 2: Análisis de una topología de red indirecta

En este artículo testeamos los micro-núcleos de aplicación en el estudio de una nueva topología de red indirecta: el *n*-árbol *k:k'*-ario adelgazado, un tipo de árbol que reduce drásticamente los recursos necesarios para construir la red comparado con el habitual *n*-árbol *k*-ario.

Este artículo ha sido seleccionado para mostrar cómo se usa INSEE para la evaluación de nuevas topologías y realizar estudios de predicción de rendimiento. Sin embargo, la principal aportación de este artículo es la definición de esta nueva topología, que tiene similar rendimiento que un árbol completo, pero con una fracción de su coste en mantenimiento y construcción. Otra de las contribuciones del artículo es la propuesta de una metodología para medir el factor coste/rendimiento de las redes de interconexión.

El autor de esta tesis participó en el diseño e implementación de la topología en FSIN, los experimentos de simulación, su comparación con los árboles habituales y la extracción de algunas de las conclusiones del trabajo.

6.4. Conclusiones

El trabajo de investigación compilado en esta tesis se ha centrado en los métodos de generación de tráfico, diseñados e implementados en la herramienta de simulación de redes de interconexión INSEE, que fue creada para mejorar ciertas carencias detectadas en otros simuladores. La gran mayoría de los simuladores existentes no pueden simular las redes de interconexión de los grandes multicomputadores, a causa de la gran cantidad de recursos necesarios. Además de que ningún otro simulador encontrado en la literatura dispone de una diversidad tan grande de métodos de generación de tráfico.

El autor ha contribuido al diseño y desarrollo de INSEE desde sus inicios, trabajando tanto en FSIN como, especialmente, en el módulo TrGen, siendo responsable de su diseño estructurado, que lo hace versátil y sencillo a la hora de añadir nuevas funcionalidades, como nuevas topologías, estrategias de encaminamiento o técnicas de control de congestión.

La principal contribución del autor ha sido el diseño e implementación o mejora de las alternativas de generación de tráfico. Estas cargas de trabajo se pueden usar desde las primeras fases de diseño de la red de interconexión hasta la última fase, empezando por rápidas distribuciones estadísticas, el tráfico basado en ráfagas, o los micro-núcleos de aplicación, el tráfico basado en trazas, y la simulación de sistema completo, ordenadas de menos a más fidelidad al tráfico real. Algunas de ellas son novedosas en su concepto, como el tráfico basado en ráfagas o los micro-núcleos de aplicación, y las demás han sido mejoradas para aportar características que mejoran los experimentos realizados, o que permiten realizar otros nuevos.

INSEE se ha usado en diferentes temas de investigación, como la investigación sobre el rendimiento de diferentes topologías de red, o el diseño de nuevas topologías, el control de congestión, la desigualdad en el acceso a recursos de la IN, o las precauciones que hay que tener en cuenta y decisiones a tomar para realizar simulación de redes de interconexión de sistema completo. Otros grupos de investigación por todo el mundo han citado la herramienta y la han usado en sus propias investigaciones.

De estas investigaciones han surgido abundantes publicaciones científicas en las que por ejemplo se ha establecido que las redes de gran tamaño con múltiples fuentes de inyección son propensas a la congestión y por eso deben emplear mecanismos de control de congestión, como hemos publicado en varios estudios.

Con respecto a las topologías de las redes de interconexión se han estudiado tanto redes directas como indirectas, y se han creado nuevos tipos, como *toro retorcido,* o *n*-árbol *k:k'*-ario adelgazado, además de investigaciones en redes bien conocidas como *midimew, n*-cubo *k*-ario, *SpiNNaker,* o *n*-árbol *k*-ario.

También el autor ha investigado la precisión de las cargas de trabajo diseñadas con respecto del tráfico real. Se han creado nuevos tipos de tráfico sintéticos que se parecen más a la manera en que las aplicaciones científicas intercambian mensajes, y cómo se representa la causalidad de los mismos y su acoplamiento. Por otro lado, se han diseñado tipos de generación de tráfico realistas para testear redes de interconexión sin necesidad de construirlas, y mecanismos para obtener tráfico real que alimente al simulador. Además de lo anterior, se han analizado detenidamente los distintos tipos de generación de tráfico, su relación con la topología, su fidelidad en cuanto a las aplicaciones reales, o cuando nos referimos a la simulación de sistema completo, las interacciones entre el control de congestión de red y el realizado por los protocolos de extremo a extremo, la influencia de la sincronización respecto del rendimiento del sistema, o el efecto de la velocidad de la red simulada sobre el sistema de simulación y las interacciones presentes.

Seguiremos desarrollando INSEE para que siga siendo la columna vertebral de nuestra investigación en esta área de conocimiento. Esto será ayudado por su diseño modular, que permite la rápida introducción de nuevas funcionalidades.

Chapter 7. References

[BBS12] Andrew Burkimsher, Iain Bate, Leandro Soares Indrusiak, “A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times”, Future Generation Computer Systems, Available online 27 December 2012, ISSN 0167-739X

[BCC03] M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-Burrow, T. Takken, P. Vranas. “Design and Analysis of the BlueGene/L Torus Interconnection Network” IBM Research Report RC23025 (W0312-022) December 2003

[BSC] Barcelona Supercomputing Center Home Page: Available at http://www.bsc.es. Accessed 18 August 2013

[CMB12] Camarero, C.; Martinez, C.; Beivide, R. “L-Networks: A Topological Model for Regular Two-Dimensional Interconnection Networks,” Computers, IEEE Transactions on, vol.PP, no.99, pp.1, 0

[CMV10] JM Camara, M Moreto, E Vallejo, R Beivide, J Miguel, C Martinez, J Navaridas. “Twisted Torus Topologies for Enhanced Interconnection Networks”. IEEE Transactions on Parallel and Distributed Systems, Volume 21, Issue 12 (Dec. 2010). pp. 1765-1778

[EGQ11] J. Escudero-Sahuquillo, P. J. Garcia, F. J. Quiles, J. Flich, J. Duato, “Cost-effective queue schemes for reducing head-of-line blocking in fat-trees”. Concurrency Computat.: Pract. Exper., 23: 2235–2248. May 2011

[GB11] Garg, S.K.; Buyya, R.; “NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations”, Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on, vol., no., pp.105-113, 5-8 Dec. 2011

[GB12] S.K. Garg, and R. Buyya, “An Environment for Modelling and Simulation of Message-Passing Parallel Applications for Cloud Computing”, Software: Practice and Experience, ISSN: 0038-0644, Wiley Press, New York, USA, 2012

[GLDS96] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI Message-Passing Interface standard. Parallel Comput. 22(6), 789–828 (1996)

[IMG06] C. Izu, J. Miguel-Alonso, J.A. Gregorio. “Effects of Injection Pressure on Network Throughput”, in Proc. PDP 2006 14th Euromicro Conference on Parallel, Distributed and Network based Processing. Montbéliard-Sochaux – France. February 2006

[IV12] C. Izu, E. Vallejo, “Throughput Fairness in Indirect Interconnection Networks”, The Thirteenth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). Beijing, China, December 2012.

[IZU09] C. Izu. “A throughput fairness injection protocol for mesh and torus networks”, High Performance Computing (HiPC), 2009 International Conference on , vol., no., pp.294-303, 16-19 Dec. 2009

[MBL01] S. Mukherjee, P. Bannon, S. Lang, A. Spink and David Webb, “The Alpha 21364 Network Architecture”, IEEE Micro v. 21, n. 1 pp 26-35, 2001

[MCEFH02] PS Magnusson, M Christensson, J Eskilson, D Forsgren, G Hallberg, J Hogberg, F Larsson, A Moestedt, B Werner. “Simics: A full system simulation platform”. IEEE Computer, Volume 35, Issue 2, February 2002, pp.50-58. DOI: 10.1109/2.982916

[MIG08] J. Miguel-Alonso, C. Izu, J.A. Gregorio. "Improving the Performance of Large Interconnection Networks using Congestion-Control Mechanisms". Performance Evaluation 65 (2008) 203–211

[NLM09] J Navaridas, M Lujan, J Miguel-Alonso, LA Plana, SB Furber. “Understanding the Interconnection Network of SpiNNaker”. 23rd International Conference on Supercomputing (ICS'09), June 8 to 12, 2009, York Town Heights, New York, USA.

[NM11] Javier Navaridas, Jose Miguel-Alonso, "Indirect cube: A power-efficient topology for compute clusters", Optical Switching and Networking, Vol. 8 no. 3 July 2011 pages 162-170.

[NMRD09] Javier Navaridas, Jose Miguel-Alonso, Javier Ridruejo, Wolfgang Denzel. “Reducing Complexity in Tree-like Computer Interconnection Networks”. Parallel Computing 36 (2010) 71–85. doi:10.1016/j.parco.2009.12.004

[NPB] NASA Advanced Supercomputing (NAS) division. “NAS Parallel Benchmarks” Available (May 2008) at http://www.nas.nasa.gov/Resources/Software/npb.html

[NPM09] Javier Navaridas, Jose Antonio Pascual, Jose Miguel-Alonso. "Effects of Job and Task Placement on Parallel Scientific Applications Performance". Proc 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Weimar, Germany, February 18-20 2009.

[NRM07] Javier Navaridas, Fco. Javier Ridruejo, Jose Miguel-Alonso. "Evaluation of Interconnection Networks Using Full-System Simulators: Lessons Learned". Proc. 40th Annual Simulation Symposium, Norfolk, VA, March 26-28, 2007.

[PGB02] Puente, V., Gregorio, J.A., Beivide, R.: “SICOSYS: an integrated framework for studying interconnection network in multiprocessor systems”. In: Proceedings of the IEEE 10th Euromicro Workshop on Parallel and Distributed Processing, Gran Canaria, Spain (2002)

[PIGB01] V. Puente, C. Izu, J.A. Gregorio, R. Beivide, and F. Vallejo, “The Adaptive Bubble router”, Journal on Parallel and Distributed Computing, vol 61, no. 9, Sept. 2001.

[RGM05] F.J. Ridruejo, A. Gonzalez, J. Miguel-Alonso. "TrGen: a Traffic Generation System for Interconnection Network Simulators". International Conference on Parallel Processing, 2005. 1st. Int. Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems (PEN-PCGCS'05). ICPP 2005 Workshops. 14-17 June 2005 Page(s): 547 – 553

[RMN07] Fco. Javier Ridruejo, Jose Miguel-Alonso, Javier Navaridas. "Concepts and Components of Full-System Simulation of Distributed Memory Parallel Computers". Proc. HPDC’07, June 25–29, 2007, Monterey, California, USA.

[RNJ11] AD Rast, J Navaridas, X Jin, F Galluppi, LA Plana, J Miguel-Alonso, C Patterson, M Lujan, S Furber. “Managing Burstiness and Scalability in Event-Driven Models on the SpiNNaker Neuromimetic System”. International Journal of Parallel Programming, Volume 40, Issue 6 (Dec. 2012), pp. 553-582

[RNMI07] F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso, C. Izu, “Evaluation of Congestion Control Mechanisms using Realistic Loads”. Proc. of the XVII Jornadas de Paralelismo. Albacete, September 2006

[SCVB12] Stafford, E.; Castillo, E.; Vallejo, F.; Bosque, J.L.; Martinez, C.; Camarero, C.; Beivide, R.; "King Topologies for Fault Tolerance," High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on , vol., no., pp.608-616, 25-27 June 2012

[WZP12] YQ. Wang; MX. Zhang, QC. Fu, ZB. Pang, “Adaptive Bubble Scheme with Minimal Buffers in Torus networks”, International Conference On High Performance Computing and Communications (HPCC 2012), pp. 914-919, Liverpool, JUN 25-27, 2012

[ZLGS99] Zaki, O., Lusk, E., Gropp,W., Swider, D.: Toward scalable performance visualization with Jumpshot. High Perform. Comput. Appl. 13(2), 277–288 (1999)

Part 2:

Selected publications

Chapter 8. Simulation as a tool to evaluate parallel computer systems. INSEE

Full reference:
Javier Navaridas, Jose Miguel-Alonso, Jose A. Pascual, Francisco J. Ridruejo
Simulating and evaluating interconnection networks with INSEE
Simulation Modelling Practice and Theory, Volume 19, Issue 1, January 2011, 494-515

This paper describes INSEE. It includes a thorough description of its two main modules, FSIN (the simulator) and TrGen (the traffic generator) together with the different ways they can be configured and the outputs that can be generated. In the context of this dissertation, the main contribution of this paper is the overview of the different mechanisms of traffic generation incorporated into TrGen. In particular, a very detailed explanation of all methods for synthetic traffic generation is provided, as well as a brief introduction of all the rest of traffic generation mechanisms.

INSEE is the core tool in the performed research and, therefore, knowing how its modules work, and their capabilities, will help understanding what kind of studies can be carried out with this tool, including those described within this dissertation.

In this publication the author has contributed to the very first design and implementation of INSEE, including the simulator FSIN and the workload generator TrGen. In particular, he constructed all interfaces to third party systems like Simics, and contributed to the implementation of different kinds of workloads, from traces to full-system simulation.

Simulation Modelling Practice and Theory 19 (2011) 494–515

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Simulating and evaluating interconnection networks with INSEE

Javier Navaridas *, Jose Miguel-Alonso, Jose A. Pascual, Francisco J. Ridruejo

Department of Computer Architecture and Technology, The University of the Basque Country, P. Manuel de Lardizabal 1, 20018 Donostia, Spain

ARTICLE INFO

Article history:
Received 12 May 2010
Received in revised form 13 July 2010
Accepted 17 August 2010
Available online 22 August 2010

Keywords:
Interconnection networks
Modeling and simulation
Performance evaluation

ABSTRACT

This paper describes INSEE, a simulation framework developed at the University of the Basque Country. INSEE is designed to carry out performance-related studies of interconnection networks. It is composed of two main modules: a Functional Simulator of Interconnection Networks (FSIN) and a TRaffic GENeration module (TrGen), together with several other modules that extend INSEE's functionality. The router models and topologies implemented in FSIN are included in this description. Likewise, the available methods to generate traffic are described thoroughly. Finally, the resource consumption of INSEE when managing different systems and workloads is also evaluated. INSEE has been used to conduct a variety of studies, including evaluation of novel topologies, diagnosis of causes of network congestion and evaluation of parallel scheduling algorithms.

1. Introduction

Supercomputing is a very valuable resource which is continuously growing in importance in business and science. Most current scientific studies rely on modeling and analyzing different natural phenomena and/or technological processes, which often require a huge amount of computing power impossible to attain using regular *off-the-shelf* computers. For instance, physicists, chemists or pharmaceutical researchers simulate, for different purposes, interactions between huge amounts of molecules. Likewise, in the business context, corporations demand large amounts of computing power in order to use data mining software over large databases, with the objective of extracting knowledge from raw data, and to use that knowledge to their advantage. Obtaining patterns of consumer habits, boosting sales, optimizing costs and profits, estimating stocks or detecting fraudulent behavior are just a few interesting application domains. At any rate the required computing power is only limited by the available resources. In general, when resources are increased then the number of runs, the grain-size (whichever this means in the particular application context), the size of the datasets or any other parameter that affects execution time is increased accordingly to fully utilize the compute power. In other words, the magnitude of the experiments is scaled up to the available assets. Briefly, there is a permanent demand of supercomputers able to cope with these challenging workloads.

A supercomputer is not only a piece of hardware. It is actually a multipart system that integrates a large collection of hardware and software elements. Therefore, the design of a supercomputer is a complex task that comprises the selection and design of multiple components, such as compute elements, storage, interconnection network, I/O infrastructure, operating system, high performance libraries and parallel applications. The performance of all these components has to be properly evaluated in order to select the most effective (again, the exact meaning of *effective* depends on the context) taking into account the purpose of the system and the workloads that are planned to be executed on them. Furthermore, the complete

* Corresponding author.
E-mail addresses: javier.navaridas@ehu.es (J. Navaridas), j.miguel@ehu.es (J. Miguel-Alonso), joseantonio.pascual@ehu.es (J.A. Pascual), franciscojavier.ridruejo@ehu.es (F.J. Ridruejo).

1569-190X/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2010.08.008

system has to be evaluated as a whole, because unexpected interactions among components can make the system suffer severe performance losses.

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 any supercomputer 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. Any delay suffered by the messages while traveling through the network will harmfully affect the execution time of applications. This is the reason why we should not decide lightly about the network that interconnects compute nodes in a high-performance computing site. 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 evaluation of INs. In our research, simulation is the key means to evaluate them. The tool to carry out these simulations is INSEE, an Interconnection Network Simulation and Evaluation Environment. A preliminary description of the design of INSEE was performed in a previous paper [40]; however, INSEE has evolved noticeably since then. Some of the features explained in that paper were *planned*, and now they are a reality. Enhancements and modifications have improved markedly the usefulness of the tool in many aspects: execution speed, memory requirement, router models and topologies, workload generators, attainable statistics, etc. For this reason in this paper we review the updated version of INSEE, describing all its current features. We also include references to several research works in which INSEE has been used as the main evaluation tool.

The two main characteristics of INSEE when compared to other simulation tools are *flexibility* and *frugality* in the use of resources. As will be seen throughout the paper, INSEE can be used to simulate a wide variety of topologies and router models, and the networks can be fed using traffic models with different degrees of fidelity to actual application traffic. This flexibility may tip the scale in favor of INSEE when it comes to select a simulation tool. Furthermore, the requirements to build and use INSEE, in terms of memory and CPU speed are frugal, and therefore it may be the environment of choice for quick deployment and fast obtention of results. It provides the capability to simulate, on a desktop computer, systems composed by tens of thousands of nodes in reasonable time, hours or a few days at most.

INSEE basically consists of a set of ANSI-C source files that can be compiled with any compliant compiler. It can be built and executed without problems 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, simplifies compilation. These decisions affect the complexity of the data structures included in the simulation, which increase memory needs. For example, as trace-driven and execution-driven simulation require specialized data structures for their operation (as will be described later), giving support to these modes is optional and can be defined at compilation time. The source code of INSEE (released under GPL) together with the required information for its operation (user manual) can be found at SourceForge [50].

For trace-based studies, the availability of a customized MPICH implementation is needed only to obtain the traces [22], but not to use them. A Simics installation is required for full-system simulation. Special care has been taken regarding memory footprint, in order to be able to cope with large-scale networks. The amount of required memory varies depending on the characteristics of the simulation (number of routers, virtual channels, buffer sizes, complexity of injected traffic, etc.). Some extreme configurations simulated in an off-the-shelf desktop computer were: a massively parallel high-performance system with the size of the largest BlueGene/L ($64 \times 32 \times 32$ nodes) [9], a complete SpiNNaker network (256×256 nodes) [24], and tree-like topologies connecting over 16 K nodes [26]. It is to be said that the main limiting factor is the amount of RAM (2 GB in our environment), as the simulation were executed quite fast (in the order of hours). With more RAM (a resource that is increasingly cheap and available) it would be possible to deal with even larger configurations; note that, in that case, the

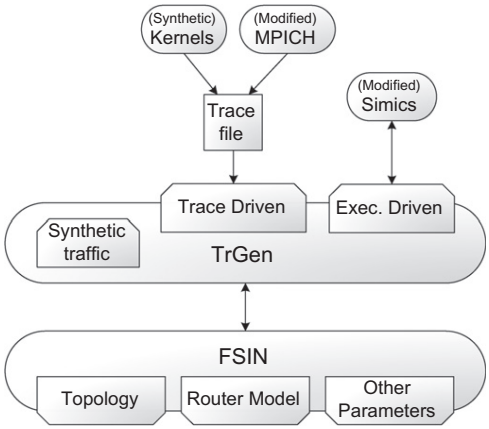


Fig. 1. Overall design of INSEE.

execution time may become the main limiting factor. For more details on resource requirements of INSEE, reader can refer to Section 5.

INSEE is composed of two main modules: FSIN and TrGen. A schematic depiction of the structure of INSEE is shown in Fig. 1. FSIN, described in Section 2, is a Functional Simulator of Interconnection Networks that allows modeling different topologies and routers. TrGen is a TRaffic GENeration module that is in charge of feeding FSIN with the desired traffic model. INSEE also includes some other additional modules that interface with TrGen to provide application-driven workloads: a modified MPICH library to obtain traces, and some modules implemented within Virtutech's Simics to carry out full-system simulation. TrGen and its companion modules are explained in Section 3. Section 4 discusses the limitations of our environment. A performance evaluation of the simulation environment is performed in Section 5, in which we show the memory requirement and execution time of several configurations of interest in our research. Section 6 performs a review of several related simulation environments, showing how they differ from INSEE. Finally, Section 7 closes the paper with some concluding remarks and an outlook of future plans to improve INSEE.

2. FSIN

The core of our environment is FSIN, a flexible, lightweight functional simulator (meaning that the router functionality is modeled in detail, but the hardware is not) that allows us to rapidly assess the performance of large-scale systems. Time is measured in terms of an abstract *cycle*, defined as the time required by a routing element to route and forward a *phit* (physical transfer unit). FSIN is able to simulate a wide variety of router models and topologies; we review those in this section. We want to remark that FSIN can be expanded in functionality by means of code additions/modifications. For example, to add a new topology, the only requirement is to formulate neighborhood and routing functions (using the C programming language). Other additions of higher complexity, such as adding a new router model, may require a deeper knowledge of the internal data structures.

2.1. Topologies

Several topologies, both direct and indirect, have been implemented in FSIN. Some of them are widely used in actual systems and thoroughly studied by the community. Some others are not as well known, and have been the subject of our own research work.

2.1.1. Direct topologies

Direct topologies are those in which every switching element or router is connected to a compute node. FSIN has topological models of standard meshes and tori for 1, 2 and 3 dimensions. Furthermore models for several alternative topologies have been added: two- and three-dimensional twisted tori, the Midimew topology and the SpiNNaker topology. All these

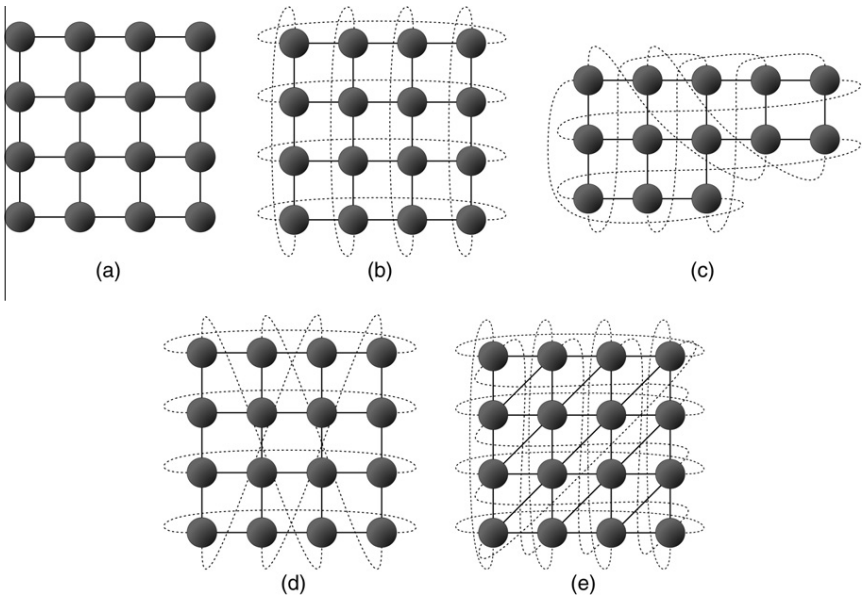


Fig. 2. Examples of the direct topologies implemented in FSIN: (a) 4×4 mesh; (b) 4×4 torus; (c) 13-node midimew; (d) 4×4 twisted torus with skew 2 and (e) 4×4 SpiNNaker topology.

topologies will be discussed in the following paragraphs, and are depicted in Fig. 2. In these graphs, nodes (and their corresponding routers) are represented by dark circles and links are represented by lines between them. Note that in FSIN all direct topologies (but meshes) can be built using unidirectional or bidirectional links.

The implemented direct topologies are the following:

- A *mesh* [11] is probably the simplest direct topology; nodes are arranged in a d-dimensional array and identified by their Cartesian coordinates. Some links at the peripheral nodes are disconnected—there are no wrap-around links. A 4×4 mesh topology is shown in Fig. 2a. This topology was historically used in high-performance computing systems because of its ease to scale up, as new nodes can be attached to the unconnected links. Currently, this topology has been abandoned in this context, in favor of networks with better topological characteristics, such as the torus. It is noticeable that the mesh topology is gaining popularity in the context of networks-on-chip because of the simplicity to deploy in silicon [23].
- The *torus* [11] is another well-known topology that has been historically used to interconnect massively parallel processors. Nodes in a torus are arranged in a d-dimensional array and identified by their Cartesian coordinates. In opposition to the mesh, the nodes in the boundaries of the topology are connected among them by means of wrap-around links. A 4×4 torus is shown in Fig. 2b.
- The *midimew* [7]—standing for MInimal DIstance MEsh with Wrap-around links—is a two-dimensional symmetric topology based on circulant graphs. It provides the best distance-related characteristics for any given number of nodes using routers of radix 4. A representation of a 13-node midimew network is shown in Fig. 2c.
- The *twisted torus* [9] is an optimization of the regular torus as it provides, for the same number of nodes, better topological characteristics: bisection bandwidth, average and maximum distance. An interesting property of this topology is that, for mixed-radix networks in which the number of nodes in one dimension doubles the number of the other dimensions ($2a \times a$ and $2a \times a \times a$) symmetry can be maintained, which help balancing the use of the channels of the different dimensions: the network does not present bottlenecks. A representation of a 4×4 twisted torus with a twist of 2 from dimension Y over dimension X is shown in Fig. 2d.
- The *SpiNNaker* topology [34] is a triangular toroidal network, in other words, a torus network with additional diagonal links to add redundancy to the design. This redundancy is deliberately devised to be exploited for fault-tolerance. A 4×4 instance of the SpiNNaker topology is represented in Fig. 2e.

2.1.2. Indirect topologies

Indirect topologies comprise all topologies in which there are switching elements that are not directly attached to computing nodes. This includes multi-stage and multi-level topologies. In multi-stage interconnection networks (MINs) all the traffic within the network flows in the same direction (usually represented from left to right) and the distance between every pair of nodes is the same: the number of stages of the network. In contrast, in multi-level topologies, the traffic must go up from the source to one minimal common ancestor of source and destination, and then down to the destination. This way, the distance between two nodes depends on the number of levels needed to reach a common ancestor.

Two indirect, tree-based multi-level topologies are implemented in FSIN, the k-ary n-tree and a reduced version of this well-known topology that we have called k:k'-ary n-thin-tree. In the graphical representations of the topologies (see Fig. 3), boxes represent switches and lines represent links between them. Note that we neither show the compute nodes connected to the first-level switches and their links, nor the last-level of upward links (which are left unplugged). These elements are hidden for the sake of clarity. We call the relation between the number of downward ports of a switch and the number of upward ports the *slimming factor*. For example, taking a look at the switches in the topology shown in Fig. 3b, four ports are downward ports, linked to switches in the next lower level. The remaining two ports of each switch are upward ports that connect to switches in the next higher level; therefore the slimming factor is the relation between 4 and 2, that is, 2:1 (or simply 4:2). Details of the indirect topologies implemented within FSIN are as follows:

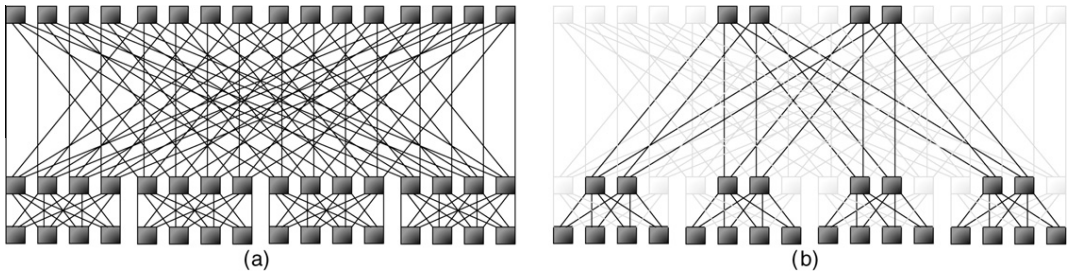


Fig. 3. 64-Node example networks of the multi-stage topologies: (a) 4-ary 3-tree and (b) 4:2-ary 3-thin-tree.

- In *k-ary n-trees* [33], k is half the radix of the switches—actually, the number of links going upward (or downward) from the switch—and n the number of levels. They can be seen as particular cases of the thin-trees (to be formally defined later), with a slimming factor 1:1. Fig. 3a shows a depiction of a 4-ary 3-tree. The main advantages of this topology are the high bisection bandwidth and the large number of routing alternatives for each pair of source and destination nodes—a path diversity that can be exploited via adaptive routing. Nevertheless, it might be expensive and complex to deploy, because of the large number of switches and links required.
- We define a *k:k'-ary n-thin-tree* [26] as a cut-down version of a k-ary n-tree in which we apply a given slimming factor. k is the number of downward ports, k' is the number of upward ports and n is the number of levels. The slimming factor is, obviously, the ratio between k and k'. Note that k does not need to be a multiple of k' so that we can produce a thin-tree with arbitrary values of k and k'. A 4:2-ary 3-thin-tree is depicted in Fig. 3b. Note the shadowed switches and links, that represent those elements that would be removed from a complete 4-ary 3-tree to form the thin-tree. In this topology the bisection bandwidth has been reduced, as well as the number of switches and links. For this reason, thin-trees are easier to deploy than regular trees and, if k and n values are kept, the radix of switches is smaller.

2.2. Modeled routers

Models of several router architectures with different purposes have been implemented in FSIN. The Dally router and the bubble router are designed to be used in direct networks, arranged in the typical 2D or 3D topologies commonly used in massively parallel processors (MPPs). The multi-stage switch allows the simulation of tree-based, indirect topologies, such as those used in many large-scale clusters. Finally the SpiNNaker router is a bespoke router architecture designed for fault-tolerance and low power consumption.

2.2.1. Dally router

A router architecture designed to be used in toroidal k-ary n-cube topologies was introduced by Dally in [10]. This architecture, which we call *Dally* router, uses a virtual channel scheme that allows deadlock-free routing using two virtual channels as Escape channels following oblivious dimension-order routing (DOR). The cycles embedded in each dimension ring are eliminated by means of restrictions in the use of the Escape channels, which cut the physical cycle into non-cyclic combinations of virtual channels. In Fig. 4, a depiction of the behavior of several modes of this scheme is shown. Adaptivity is possible increasing the number of virtual channels: two virtual channels (usually 0 and 1) are used as Escape channels (using deadlock-free DOR) while adaptive routing can be used in the remaining ones. The overall arrangement (according to Duato's theorem [13]) is an adaptive, but deadlock-free routing algorithm. Several strategies to route packets are implemented in FSIN to be used together with the Dally router:

- In *trc* all packets are injected in the same Escape channel. Packets remain in that channel, until reaching the wrap-around link of a toroidal topology. Then, it is switched to the other one. In other words, the cycle is transformed into a spiral (acyclic) as can be seen in Fig. 4b.
- In *basic* those packets that have to cross the wrap-around links of a dimension—let us call them P_W—circulate through one Escape channel in that dimension. Those that do not have to use the wrap-around links—denoted P_D—use the other Escape channel. This way, the physical ring is split into two separate virtual chains, which do not include cycles, see Fig. 4c.

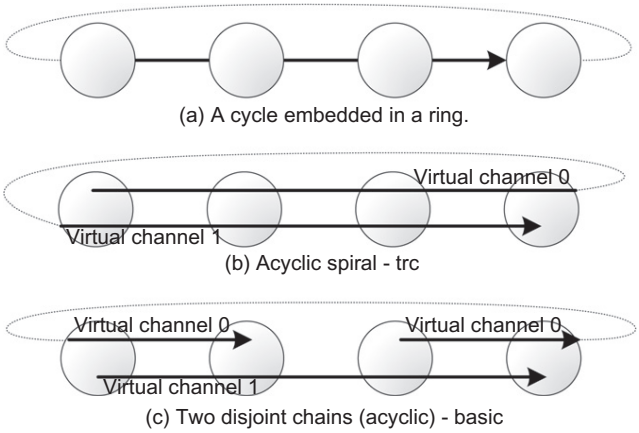


Fig. 4. Dally scheme to avoid deadlock in a unidirectional ring. (a) A cycle in a ring. (b) The cycle is cut by changing the virtual channel at crossing the wrap-around link (*trc* policy). (c) The cycle is removed by splitting the physical ring into two separate virtual chains (*basic* policy).

- *Improved* is an optimization of the basic policy to obtain better balancing in the utilization of both VCs. P_W packets are forced to transit by one of the Escape channels. P_D packets can use any of the two Escape channels. This way, P_W packets cannot block P_D packets, and thus P_D packets eventually will use the other channel and reach their destination. Note that this policy may lead to starvation of P_W packets, as P_D packets may make intensive use of the two Escape channels.
- *Adaptive* works as the improved policy, but adding routing adaptation capabilities. Two virtual channels work as Escape channels following the improved policy and the remaining virtual channels are used as adaptive channels (always using minimal paths). The packets randomly select a viable adaptive output port and, only in the case that no adaptive channel is available, the packets try to use an Escape channel as dictated by the improved strategy (P_W packets are restricted to one of the Escape channels, while P_D packets are free to use any of the two Escape channels).

The modeled router has queues in the input ports, and a crossbar that interconnects input and output ports as depicted in Fig. 5. Note that the crossbar has as many ports as virtual channels (plus additional ports for injection and consumption). Therefore, the utilization of VCs increases crossbar complexity.

2.2.2. Bubble router

The *bubble* router [39] is also designed to be used in *k*-ary *n*-cube topologies, having the same internal architecture of the Dally router (see again Fig. 5) but follows a different approach to avoid deadlocks; instead of avoiding cycles, this scheme avoids all the buffers in a ring (cycle) becoming full. To do so, packets can only enter an Escape channel from injection or from other channel when there is room to store at least two packets, one for the entering packet and another one to ensure that there is, at least, one free slot (*bubble*) to be used by the packets inside the Escape channel. This behavior, depicted in Fig. 6, requires a single Escape channel to avoid deadlock. In the depiction, only node 0 and node 3 can inject into the ring, as they have room enough in the queues of the ring to comply with the bubble restriction. Alternatively, the other two nodes (node 1 and node 2) are not allowed to inject as they do not comply with that restriction. Adaptivity can be easily incorporated by adding additional virtual channels with adaptive routing, but always checking the bubble restriction when moving packets from an adaptive channel to the Escape channel.

The bubble router can route packets following several strategies to decide, when a packet is awaiting at the head of the input queue (or at the injection queue), to which output VC a request will be made to forward the packet.

- With *oblivious* request mode all VCs follow the bubble restrictions (all of them behave as Escape channels), using DOR routing. Once a packet enters into a VC, it never abandons it—in short, there is no adaptivity.
- When using the **random** request mode, the router tries to route packets waiting at an input port through the output port of the same VC. If this is not available, any profitable adaptive VC can be requested, choosing it randomly. Finally, if no profitable adaptive VC is available then packets try the Escape channel.
- The **shortest** request mode works as the previous one but, when trying a profitable adaptive VC, selection is done considering the space available in the channel's queue, choosing the one with more room.
- If the **smart** request is selected, a packet is injected initially into a random channel and tries to continue in the same dimension and virtual channel. If this is not possible, then it is moved to a profitable adaptive virtual channel in another dimension. If no adaptive, profitable channel is found, the packet tries to move to the Escape channel. This strategy tries to avoid congested links by changing the traveling dimension every time it reaches a port that is in use.

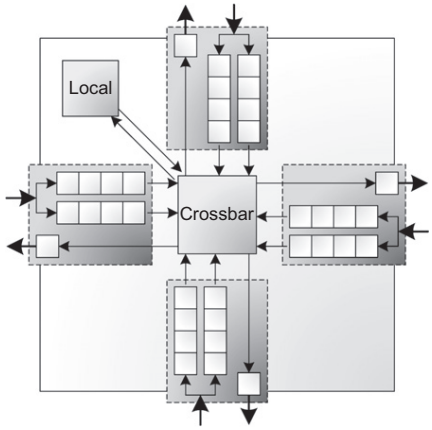


Fig. 5. Architecture of Dally and bubble routers for a 2D topology and two virtual channels per physical link.

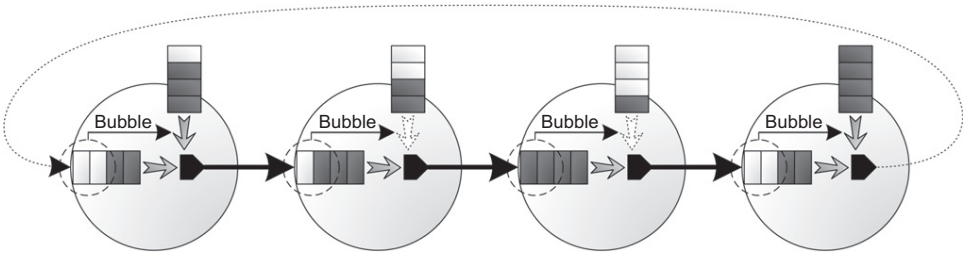


Fig. 6. Bubble scheme to avoid deadlock in a unidirectional ring. Grey arrows represent allowed data movements. Dotted arrows represent not allowed data movements.

Several other experimental strategies have been implemented for the bubble router but are not discussed here for the sake of brevity.

2.2.3. Multi-stage router

Routers for multi-stage INs are simpler than those for k-ary n-cubes, as routing in multi-stage topologies is deadlock-free provided that acyclic routing algorithms (such as up∗/down∗ [44]) are used. The router has a centralized crossbar and can have any arbitrary number of ports and virtual channels. Queues are located at the input ports. The model of the router for multi-stage topologies is depicted in Fig. 7. Routing can be static or adaptive, but in both cases shortest path routing is used to guarantee deadlock-freedom:

- In *static* routing the upward path is defined statically and depends on the source of the packet (*source_id* mod k). The downward path is also static and depends on the destination of the packet (*destination_id* mod k). Furthermore, if physical links are split into several virtual channels, one virtual channel is randomly selected at injection and the packet never leaves it.
- When using **adaptive** routing the downward path is static and depends on the destination, but the packets can adapt when traveling upwards. A credit-based adaptive routing is used, which works as follows: a packet at the head of an input queue tries to go through the profitable output channel with most available room in the queue of the neighbor input port (credit). If several output channels have the same credit, one of them is selected randomly. Note that it is assumed that credit transmission is performed out-of-band and, therefore, does not interfere with regular traffic.

2.2.4. SpiNNaker router

The SpiNNaker router [34] is also implemented within INSEE. This is a specific-purpose router designed for a large-scale machine with austere hardware constraints. The main focus of the SpiNNaker system is on low power consumption and fault-tolerance, and the design of the router also follows these concerns. As crossbar-based routing engines require excessive hardware resources, they can not be part of the SpiNNaker router; therefore, it was designed using a more frugal approach. All the ports of the router are hierarchically merged in such a way that the routing engine is accessed by a single packet at once. This way, a packet that can not be forwarded will force all the packets in the router to wait until it is forwarded or discarded, an undesired effect known as Head-of-Line blocking. Fortunately, as the routing engine is relatively faster than the transmission ports, this situation is unlikely to happen. A depiction of the architecture of the router is shown in Fig. 8.

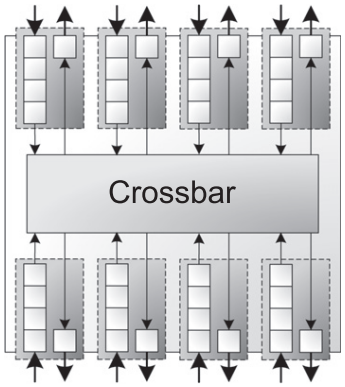


Fig. 7. Architecture of a multi-stage router with eight ports and one virtual channel per physical link.

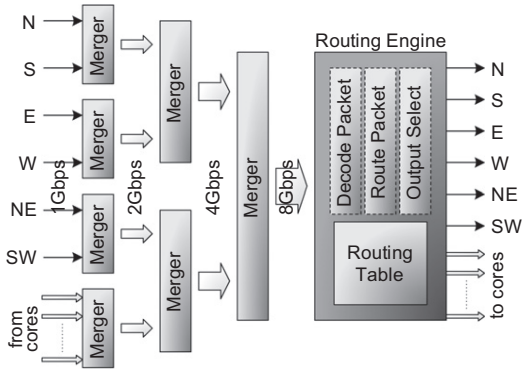


Fig. 8. Architecture of the SpiNNaker router.

Routing is performed by means of the routing tables inside each router. However instead of being destination-based, the routing tables choose the output port(s) for a given packet taking into account the source of the packet. This behavior makes the source nodes and the packets unaware of the destination node(s). The way routing tables are filled is application-dependent.

2.2.5. Other router parameters

The *phit length* (physical transfer unit, measured in Bytes) and the *packet length* (measured in phits) can be set to any arbitrary value. Note that the packets have a fixed length and when a smaller packet is required, it will carry some empty phits. The *transit queue length* and the *injection queue length* (measured in terms of room to store packets) can be set to any arbitrary value larger than one packet. Similarly, any desired number of virtual channels can be used.

The *arbitration* of output ports (selection among all the requesting input queues) can be performed in several ways; some of them are unaware of the traffic, and some others take into account some attributes of the packets to give priority to certain flows. The arbitration policy can be defined for all the router models except the SpiNNaker router, because it treats packets once at a time. The defined policies are the following:

- *Round Robin* arbitration.
- *Random* arbitration selects randomly among all the requesting input ports.
- *FIFO* arbitration selects the first input port that requested the output.
- *Longest* arbitration selects the input port having the highest queue occupation.
- *Age* arbitration selects the input port containing the oldest packet, measured since the packet was injected into the network.

Furthermore, we have implemented and evaluated several *congestion control mechanisms* around the Dally and the bubble routers. Two of them are *local* mechanisms in which routers detect congestion taking into account only the state of their own queues. The other one is a *global* mechanism that throttles the injection taking into account the state of the whole system. These mechanisms are described as follows:

- In-transit Priority Restriction (IPR). For a given fraction P of cycles, priority is given to in-transit traffic, meaning that, in those cycles, injection of a new packet is only allowed if it does not compete with packets already in the network. P may vary from 0 (no restriction) to 1 (absolute priority to in-transit traffic). This mechanism can be used along with the two router models and is the method applied in the IBM's BlueGene/L torus network [2].
- Local Buffer Restriction (LBR) mechanism was designed specifically for adaptive bubble routers. A previous study showed that the bubble restriction, in addition to guaranteeing deadlock-freedom, also provides congestion control for the Escape sub-network [17]. LBR extends this mechanism to all new packets entering into the network. That is, a packet can only be injected into an adaptive virtual channel if such action leaves room for at least B packets in the transit buffer associated to that virtual channel. The parameter B indicates the number of buffers reserved for in-transit traffic. In other words, congestion is estimated by the local buffer occupancy.
- Global Congestion Control (GCC) estimates network congestion by examining the status of the whole network. Injection is stopped when network occupancy reaches a given threshold G, which is given as a fraction of the whole network capacity. In our implementation, the network utilization is measured every T cycles (being T a simulation parameter), and then transmitted out-of-band to all nodes, in such a way it does not interfere with application packets.

2.3. Node model

In INSEE, nodes are modeled in a rather simplistic way: a traffic generator/consumer plus an arbitrary number of injection queues or *injectors*. In the case of direct topologies, nodes are attached to the routers through a specialized connection. Several *injection* policies are implemented to be used with direct topologies:

- *Shortest* is the simplest injection policy. It inserts a newly generated packet into the injection queue with more room.
- When using *dor* policy, there is a dedicated injection queue per network dimension/direction, which significantly reduces Head-of-Line blocking effects [17]. A dimension-order pre-routing phase is carried out to decide in which queue will be injected the new packet. If the selected queue is not available, the injection will be stopped.
- The *dor + shortest* policy tries to inject using the *dor* policy but, if the selected injection queue is full, then it uses the shortest policy.
- In the *shortest profitable* policy, the packet will be injected in the valid queue (once pre-routed) that has most empty space.
- *Longest path* policy injects packets in the queue associated to the direction through which the packet has to travel a larger number of hops.

Furthermore, the specialized connection of direct routers allows for two alternative *consumption* modes:

- *Single* consumption policy considers the consumption port as a regular output port following the selected port-arbitration strategy.
- *Multiple* consumption policy assumes a consumption port wide enough to accept simultaneously a phit from each input port, meaning that several packets can be consumed simultaneously.

In the case of indirect topologies, nodes are connected to the network through regular links with the same characteristics of the links within the network. For this reason, both injection and consumption are serialized and do not allow any special policy (shortest injection and single consumption).

2.4. Output data generated by FSIN

Depending on the kind of research work that is being carried out with INSEE, different types of results are needed: throughput/delay analysis, channel utilization, time to deliver a workload, dropped packets, system evolution, etc. For this reason FSIN is able to capture and report a wide variety of statistics from the performed simulations. These statistics can be captured for the whole simulation or, in contrast, can be averaged from different time intervals. For debugging purposes, cycle-by-cycle information can be generated as well. As the execution of FSIN is deterministic (given a seed used to generate random numbers, if needed), a summary report of the input simulation parameters is printed in order to allow repeating a given execution.

2.4.1. Run-time statistics

Run-time summaries of statistics can be obtained at pre-configured time intervals. This allows us to follow the evolution of the system. Note that applications usually pass through different stages of communication patterns and, therefore, of network usage. The statistics captured along the execution are the following (always related to the measurement time interval):

- *Injected and accepted load*: The average injection and consumption rates of network nodes, measured in phits/node/cycle.
- *Packets*: The number of packets that have been injected, consumed and dropped due to different reasons (blocked at the head or tail of injection queues and/or dropped in-transit).
- *Network utilization*: The average number of packets in the queues of the system.
- *Distance*: The (averaged) distance traversed by consumed packets.
- *Latency*: Several latency-related measurements are captured including: average, standard deviation and maximum latencies. Note that latencies are measured since the moment at which packets are generated, as well as since the moment they are injected into the network.

2.4.2. Traffic evolution mode

Using this mode, the wandering of packets and phits can be shown in order to have a better understanding of how the network evolves. Its main interest, though, is for debugging purposes. Selecting *packet evolution*, the simulator will show in its output the following events: packet generation, packet injection, packet arriving to a node, packet leaving a node, packet dropping and packet consumption. For *phit evolution*, the output of simulator will show the following events: phit generated, phit moved, phit dropped and phit consumed.

2.4.3. Final report

Once the simulation is concluded, FSIN shows a comprehensive report with a summary of the whole simulation. All the previously discussed figures are averaged for the whole simulation. In order to allow a better understanding of the results, the standard deviation and the maximum of all these figures are also shown, as well as the total simulation time, measured in cycles. The actual time required to perform the simulation is also provided.

If requested, FSIN can capture statistics to allow an even more detailed understanding of the simulation. A histogram of the queue occupancy along the simulation can be shown, which is useful to understand which (and how) network queues were occupied. This can be used, for example, to diagnose anomalies in network utilization.

A map of the load managed by each (virtual) port of the system can also be generated. This is useful to find network bottlenecks, and also to understand the virtual channel management strategy used by the system.

If our interest revolves around the distances traversed by packets, FSIN can show a histogram of distances, as well as a map showing pair to pair communication. Both histogram and map are generated for the distance distributions at the injection *and* at the consumption. If the captured data for injection and consumption are notably different, this can suggest that there are nodes that are suffering starvation. Note that the pair-to-pair map gives more information, but may require excessive memory if the system is large, as the memory requirements scales quadratically with the number of system nodes.

2.4.4. Using the results generated by FSIN

Performance data generated by a single run of FSIN can be useful for some studies, but often many runs with different random seeds are required in order to obtain statistically meaningful results. Additionally, for many studies a parameter sweep is required. It is up to the researcher to combine the results of the simulation runs in the form of suitable tables and graphs, although FSIN helps by means of outputting structured text files (in CSV format) that can be easily imported into spreadsheets and other data analysis tools. For illustrative purposes, Fig. 9 depicts some graphs obtained from FSIN results, which are similar to those used in our research works.

2.5. Validation of results

Given that, in most cases, we do not have access to the systems we simulate with INSEE, a direct comparison of the behavior of actual versus simulated systems can not be carried out. However, the results obtained with INSEE have been cross-validated with mathematical models of the different implemented topologies. These mathematical models include throughput

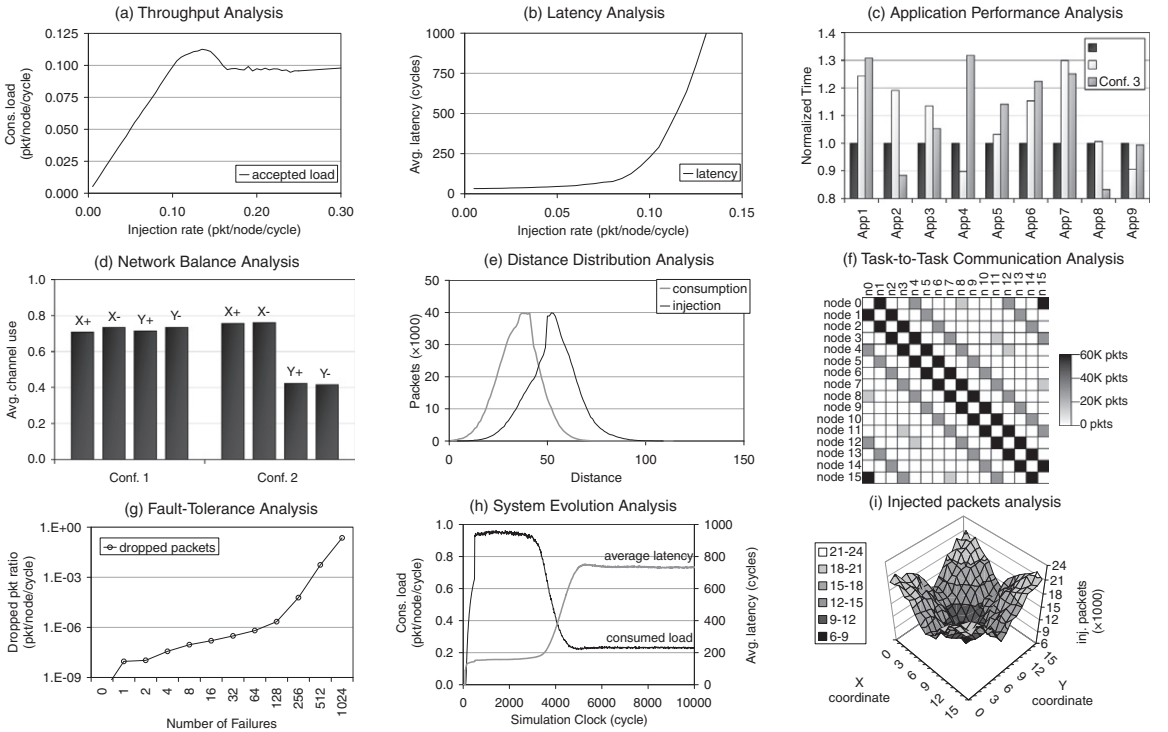


Fig. 9. Examples of IN analysis supported by INSEE: (a) throughput analysis; (b) latency analysis; (c) application performance analysis; (d) network balance analysis; (e) distance distribution analysis; (f) task-to-task communication analysis; (g) fault-tolerance analysis; (h) system evolution analysis and (i) injected packets analysis.

analysis for uniform traffic as well as distance-related characteristics of the topologies. In some of our previous works, the reader can find examples of those validations for different topologies: twisted tori [9], regular tori [21], 2D triangular torus such as the SpiNNaker topology [24] and tree-like topologies [26]. Interestingly, in [9], an erroneous mathematical model for the network throughput when managing uniform traffic was detected using INSEE.

We have also carried out cross-validation with other simulation tools, specially with SICOSYS [36,48], see details in Section 6. Results obtained with INSEE have always been consistent with those obtained with other tools, for similar configurations and parameters.

3. TrGen

TrGen—standing from TRaffic GENerator—is the module in charge of feeding the simulation with the desired kind of traffic. It interfaces with FSIN and passes the traffic in the form of network packets. TrGen is capable of generating purely synthetic traffic patterns, as well as of extracting communication patterns from traces or from the actual execution of parallel programs in a simulated computing environment (interfacing with Simics).

3.1. Some considerations about workloads

When it comes to evaluate the interconnection network of parallel or distributed systems, the way we model the workload imposed to the network is of crucial importance. In the first phases of the design of a system, it may be desirable to obtain results as fast as possible, even if they are not fully accurate. Alternatively, a complex model providing high fidelity may be required in the last phases just before deployment, to ensure that all the pieces of the system perform as expected when they are put together and also that the applications make efficient usage of the underlying hardware.

Synthetic traffic patterns from independent sources [11] provide a good first approach to evaluate a network, because they allow us to rapidly asses the raw performance of a network, and because it can be supported by any of the previously discussed analytical studies. Very often, randomly generated traffic is used to evaluate systems: uniform, hot region and hot spot traffic patterns have been used in a large collection of studies. Other commonly used patterns are those that send packets from each source node to a destination one as indicated by a certain permutation, defined as a function that takes as input the address of the source.

Nevertheless, actual applications that communicate internally using patterns like these synthetic ones, in which traffic-generating nodes produce messages without coordinating among them, are rare. For this reason, trace-driven simulation is often preferred, in order to perform a more realistic evaluation of a system. Feeding a simulator with a trace is not an easy task. To evaluate only the network of a parallel system we could implement a *dummy* model of the processing node, allowing it to inject messages into the network as fast as it can, ignoring the causality of messages and the computation intervals. This approach is a stress test of the network, because of the contention caused by all nodes injecting at the maximum pace.

Alternatively, it would be more realistic to maintain the causal relationship between the messages in the trace; in other words, if the trace states that there is a reception before a send, the node has to wait for that reception to be completed before starting with the send. This mechanism provides more fidelity than the inject-at-will model. To further improve the simulation accuracy, compute intervals (periods in which nodes do not inject load into the network) should be taken into account, maybe applying a *CPU-scaling-factor* in order to simulate a system with faster (or slower) CPUs than those used to capture the trace.

Still, there are some problems with the trace-driven approach that we should not ignore. Firstly, the information captured within the trace could be inexact due to the intrusion effects of the trace logging mechanism. Secondly, traces may reflect some of the characteristics of the system in which they have been obtained. Finally, traces from actual applications running in a large set of processors are difficult to obtain, store and manage, and these are precisely the ones of interest in our performance evaluation studies.

A hybrid between the utilization of synthetic traffic patterns and traces is the estimation of probability distributions for destinations, inter-generation times and message lengths, using data extracted from actual traces to feed some distribution-fitting program. For example, the spatial distribution of several application prototypes were shown in [5]. Once we have the distributions that model the application used to obtain a trace, we can generate random traffic resembling it. However, as stated before, in actual applications causal relationships among messages are common, and this technique does not capture them. And, again, the inexactitudes of the information within the trace (due to the characteristics of the system in which the trace was captured, and the intrusion of the logging process) may generate estimated distributions, or parameters of those, that are not valid.

In order to introduce causality in the simulation, and to fill the gap between trace-driven simulation and independent sources traffic, a bursty traffic model can be used. This model uses the previously discussed synthetic traffic patterns, but emulates application causality using a coarse-grained approach. The message generation process passes through a certain number of *bursts* or steps, during which nodes can inject at will until the burst finishes, and then stall until the starting of the next burst. Synchronization among nodes is included in this model, but in a very primitive way (roughly a barrier); fine-grained synchronization among messages/tasks are not considered.

A further refinement of the bursty model is by means of application kernels. These kernels are implemented using point-to-point synchronization and communication primitives, and can include different levels of causality such as long chains of dependencies. Application kernels emulate the behavior of small parts of actual applications, but when compared with regular traces they are more flexible because they are fully configurable in terms of number of communicating tasks, message size, task coupling, etc. This gives an advantage when compared to traces, as the latter are difficult to capture and manage for large-scale systems. Furthermore, as kernels are only small parts of applications, their execution is orders of magnitude faster than trace-based configurations, while still providing a reasonable level of accuracy. These application kernels are inspired in communication patterns observed in actual applications. In some cases they reproduce virtual topologies, or implementations of collective communication primitives, while in others they reproduce programming models such as master–slave.

Finally, the most accurate methodology to evaluate a parallel computer would be running a detailed full-system simulation that includes the interconnection network, the compute nodes, the operating system, some support libraries and the applications running on them. This is a very complex, error-prone task, as well as a high resource-consuming methodology that could need a system similar in dimension to the one we want to evaluate. These are the main reasons to justify the limited utilization of execution-driven simulation to evaluate medium-to-large size distributed memory parallel computers.

Table 1 closes this section summarizing the methodologies to generate traffic to fed simulations. Columns show the spatial pattern of the workload, the causality among messages, the complexity of generating, managing and performing simulation, and the kind of evaluation supported by the traffic model. Note that all these workload generation modes are supported by TrGen.

3.2. Synthetic workloads

Our simulation environment provides a wide variety of synthetic workloads that can be used to measure the performance of the communication infrastructure. These workloads have different levels of fidelity to actual application workloads, in terms of spatial and temporal/causal patterns.

3.2.1. Independent traffic sources

A widely accepted and used mechanism to feed simulations is the use of synthetic traffic patterns from independent sources [11]. This kind of workload allows tuning the injection rate of nodes; they try to inject at this rate, following a Bernoulli distribution. There is no causality among receptions and injections. When using this kind of traffic, the simulation run is split in three phases. The first phase simulates a given number of cycles without capturing statistics and is used as a warm-up phase. When warm-up finishes, a phase in which convergence is checked starts. During this second phase network statistics are captured every a given number of cycles (convergence intervals). If three consecutive intervals are within a given range, it is assumed that convergence has been reached. Finally a statistics-capturing phase starts. During this phase a given number of samples or batches are run, capturing the statistics during these phases, showing their average and their standard deviation.

The spatial traffic patterns supported by TrGen are the following:

Table 1
Description of different traffic models used for simulation-based evaluation of parallel systems.

Traffic model	Spatial pattern	Causality	Complexity	Evaluates
Independent traffic sources				
Random	Random	No	Very low	Raw performance
Permutation	Worst case	No	Very low	Raw performance
Estimation of distributions	Application-like (origin-dependent)	No	Low/medium	Selected application (origin-dependent)
Bursty traffic sources				
Random	Random	Coarse-grained	Very low	Raw performance
Permutation	Worst case	Coarse-grained	Very low	Raw performance
Estimation of distributions	Application-like (origin-dependent)	Coarse-grained	Low/medium	Selected application (origin-dependent)
Application-based				
Application kernels	Application-like	Application-like	Low/medium	Usual communication patterns
Trace-driven (inject-at-will)	Application-like (origin-dependent)	No	Medium	Raw performance when congested
Trace-driven (causal)	Application-like (origin-dependent)	Application-like (origin-dependent)	Medium/high	Selected application (origin-dependent)
Execution-driven	Actual application	Actual application	Very high	Selected application

- *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* (UN), in which all the nodes have the same probability of being selected as destination, and the non-uniform *hot spot* (HS) and *hot region* (HR), in which 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. In local (LO), the probability of selecting destination nodes decreases with the distance (so that most packets are sent to nearby nodes). Furthermore, TrGen can read and follow user-defined distributions (for example extracted from actual applications) which can be introduced as a *histogram* or as a *population*.
- *Distribution*: Distribution patterns send packets sequentially to each one of the remaining nodes. The initial destination node can be the next one in order of identifier (SD), or can be selected randomly (RD).
- *Permutation*: Given a source node, the destination node is always the same, and is computed as a permutation of the source node identifier (generally bit permutations) [11]. The permutations implemented in TrGen are the following: Bit Complement (BC), Bit Reversal (BR), Bit Transpose (BT), Butterfly (BU), Perfect Shuffle (PS) and Tornado (TO). Their mathematical descriptions are shown in Table 2. Identifiers of source and destination are denoted as s and d, respectively. For bit permutations, l is the number of bits, s_i and d_i are the ith bit of the source and the destination respectively. For the tornado permutation n_x is the number of nodes in the X and Y dimension of a 2D cube topology. $\langle s_x, s_y \rangle$ and $\langle d_x, d_y \rangle$ are the Cartesian coordinates in the 2D cube of the source and destination nodes, respectively.
- In some scenarios we study the performance of a parallel job composed of a collection of communicating tasks that uses only one part of the network. This job could run in solitary, while the non-used processors are empty. However, it would be more realistic if the job experienced interference from other jobs sharing the machine. To model these scenarios, is it possible to combine application-based (or application-inspired) workloads to realistically simulate the target job, while using traffic from independent sources (normally, random traffic) to emulate background network utilization. We used this combination in some studies on parallel application mapping [31].

3.2.2. Reactive traffic

As part of the evaluation of the SpiNNaker system we have implemented a traffic model that resembles biologically plausible neural traffic [29]. The definition of this traffic model is quite simple: simple independent traffic in which nodes may generate, with a given probability, a packet (or a collection of packets) after the reception of a packet. The parameters for this kind of simulation are the same that for the independent traffic (spatial pattern and injection rate) plus a probability to trigger new traffic and the number of packets triggered (given as an interval). Note that this behavior models the activation and firing of a neuron.

3.2.3. Bursts

Bursty traffic sources provide a simple model to introduce system-level synchronization when managing synthetic traffic, emulating the execution of a barrier synchronization operation every b packets. To do so, nodes are allowed to inject b packets as fast as they can, and then stall until all packets of the burst have been injected and consumed by all the nodes. Note that small values of b resemble tightly-coupled applications while large values of b resemble loosely-coupled applications. In this mode, the figure of merit to measure network performance is normally the time taken to deliver all the packets in one or several bursts, being the faster the better. All the spatial patterns discussed in Section 3.2.1 can be used with *bursty* traffic sources. This burst-synchronized behavior avoids starvation of nodes, as all of them are allowed to inject exactly the same amount of traffic into the network.

3.2.4. Application-inspired workloads

One of the contributions of our group, within the INSEE environment, is a set of synthetically generated traffic patterns that resemble the way actual scientific applications communicate. These traffic patterns are a further refinement of bursty traffic in which point-to-point synchronization is supported. They can be considered application micro-kernels as they mimic different communication patterns widely used on parallel applications. This mimicry is done both in terms of spatial patterns and causality. The utilization of this kind of workloads is based on the same infrastructure used for trace-based simulation, described later. A (standalone) INSEE module takes the appropriate parameters and generates a synthetic trace file. This file can be used in a simulation as a regular trace file.

Table 2
Mathematical description of the permutation patterns implemented in TrGen.

Pattern	Destination	Example
Bit Complement	$\forall i : 0 \leqslant i \leqslant l-1, \quad d_i = \sim s_i$	BC(11011000) = 00100111
Bit Reversal	$\forall i : 0 \leqslant i \leqslant l-1, \quad d_i = s_{l-i-1}$	BR(11011000) = 00011011
Bit Transpose	$\forall i : 0 \leqslant i \leqslant l-1, \quad d_i = s_{(i+l/2) \bmod l}$	BT(11011000) = 10001101
Butterfly	$d_{l-1} = s_0, \quad d_0 = s_{l-1}$	BU(11011000) = 01011001
Perfect Shuffle	$\forall i : 0 \leqslant i \leqslant l-1, \quad d_i = s_{(i-1) \bmod l}$	PS(11011000) = 10110001
Tornado [46]	$d_x = (\frac{n_x}{2} + s_x) \bmod n_x, \quad d_y = s_y$	$TO_{8\times8}(\langle 3, 2 \rangle) = \langle 7, 2 \rangle$

We can arrange the application-inspired workloads into three different groups. Some of them are reproductions of the way MPI collective operations are implemented relying on point-to-point communications (this is, when hardware support for collectives is not available), taking into account optimized as well as non-optimized implementations. The second group includes communication patterns that mimic those applications that rely on virtual topologies, passing messages to immediate neighbors. The third group is more generic, implementing different modes of random generation of synchronized interchanges of messages. The exact definition of all these kernels can be found in [25,27] and is not discussed here for the sake of brevity. Application micro-kernels provide a reasonable level of accuracy at a reduced cost in terms of computing power required by the simulator. Moreover, they have the capability of generate workloads with thousands of communication nodes which are difficult, if not impossible to obtain with application-guided workloads (described next).

3.3. Application-guided workloads

Synthetic 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 TrGen can also use traces from applications to perform trace-driven simulation, and even interact with Simics to perform full-system simulation. This subsection is devoted to discuss these execution modes.

3.3.1. Traces

We use a modified version of MPICH [20], one of the most popular implementations of MPI, to obtain trace files usable with TrGen. MPICH includes an easy-to-use mechanism to obtain trace files from running applications. However, these traces are not useful for our purposes because collective operations (such as barriers, broadcasts and reductions) appear as such in the trace files, that is, the actual interchange of messages necessary to implement those operations in networks without native support for collectives is not reflected. Internally, MPICH implements collective operations using point-to-point operations (when no better alternative is available). We modified MPICH in order to make those point-to-point operations visible, registering them in the trace files along with the corresponding collective operation. The intervals of time between MPI operations are considered as computation intervals. A CPU-scale factor can be applied to these intervals in order to simulate processors faster or slower than those of the system in which traces were captured.

Trace files are slightly pre-processed before using them with TrGen. Only a few, relevant fields are selected (event type, source node, destination node, message size and tag) and organized in a simplified format more suitable for TrGen. It would be possible to use this format to feed simulations with traces obtained from sources different to MPICH, a network sniffer being a good example, just building the right pre-processor.

To reproduce the causal relationships between events in the trace files, TrGen requires a special data structure to store past and future events, shown in Fig. 10. Each node of the simulated applications has an event queue, which is fed from the trace file. A packet is sent to 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 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.
(3) Nodes sequentially execute the events in their event queue.
  (a) If the first event is a send, remove the event and inject the corresponding message into the network.

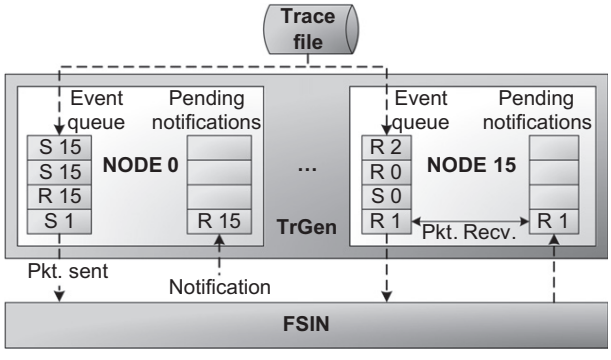


Fig. 10. Data structure used to maintain causality in the trace-driven simulation.

(b) 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.
(c) If it is a computation event, put the node on hold for the required period of time, using if selected a CPU-scale factor.

(4) When the simulator delivers a message, put it in the pending notifications list.

An example of this procedure is depicted in Fig. 10, note that **R** represents a reception and **S** represents a send; computation events (**C**) are not considered for the sake of simplicity. In the figure, node 0 cannot advance, because it is waiting for a message from node 1, even if a message from node 15 has been already received. In contrast, node 15 can advance because the required message from node 1 has been delivered. This mechanism reproduces the actual way messages were interleaved when running the application, complying with the causal order between a reception and the subsequent sends it may trigger.

Traces obtained from one system are often used to evaluate via simulation the performance potential of another, different, target system. However, this approach has some drawbacks. In the context of IN design and evaluation, traces obtained with the same collection of nodes running a parallel application with two different INs A and B may be different, because properties of A and B differ, and those properties have an influence on the way nodes interchange messages. For this reason, performance results obtained with traces may not be totally accurate [14]. A thorough discussion of the design, limitations and issues of the trace-driven simulation within the INSEE environment was carried out in [22].

3.3.2. Interaction with Simics

TrGen can interface with Virtutech's Simics [19] to perform a full-system simulation. Simics is in charge of simulating a cluster of multiprocessors, while INSEE simulates the IN. In our experimental environment, each Simics instance is executed in a different computer and can simulate up to eight compute nodes (limited by the available RAM of our machines). A single instance of INSEE simulates the network that interconnects all the simulated nodes. In order to perform a correct simulation of the parallel system, a set of interfacing modules were implemented, some for INSEE and some others for Simics. These modules implement the following functions:

- Transference of application traffic from Simics instances to FSIN, and vice versa.
- Synchronization among all the elements taking part in the simulation (FSIN and the collection of Simics instances).

Every simulated computing node has a simulated Ethernet network interface card which is instrumented to put the messages into the Traffic manager of its Simics instance. The Traffic manager sends meta-information about the messages to TrGen and is in charge of sending the complete message to the corresponding Simics instance and node, once it has reached its destination in the simulated network. In Fig. 11 we can see a depiction of all the elements involved in the execution-driven simulation.

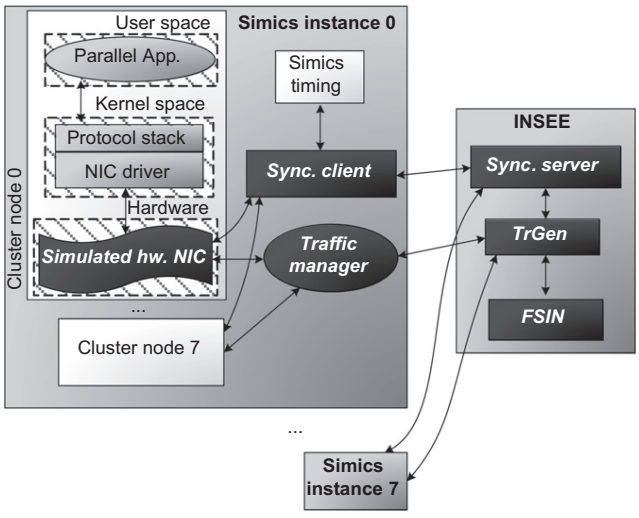


Fig. 11. Elements of our full-system simulation environment. Black elements belong to INSEE.

It is remarkable that Simics gives support to the synchronized execution of all the nodes within a given instance. Still, we need to add a synchronization client to keep synchronization among the different Simics instances, and also between them and FSIN. To this purpose INSEE is complemented with a synchronization server which keeps the whole simulation running at the same pace.

The interchange of application traffic is managed by TrGen, which stores meta-information of each message taking part in the simulation. TrGen is also in charge of injecting traffic into FSIN and of monitoring it to recognize when a message has arrived to its destination. Once a message arrives to its destination in FSIN, TrGen sends a confirmation to the source Simics instance in order to send the complete message to the destination instance and node. The traffic manager inside each Simics instance is in charge of storing the complete message and sending it to the destination node. When the Traffic manager receives a message to a simulated node it injects the message into the NIC module of the proper node.

Synchronization is fairly more complex as a two-level mechanism is implemented in order to synchronize the multiple elements taking part on the simulation. On the one hand, all the nodes simulated within a Simics instance are executed in a step-wise fashion: nodes go to execution sequentially for a given number of CPU cycles (*slice*) and, once the running one finishes its slice, the next node enters into execution for the same amount of simulated time. This mechanism is part of Simics. A synchronization client implemented inside each Simics instance is in charge of sending a message to TrGen to make it know that it has finished its slice; this is carried out when all the simulated nodes finish their corresponding slices. Once this message is sent, the synchronization client suspends the execution until a message from TrGen is received, allowing the execution of a new slice. The synchronization server, implemented within TrGen, waits until all Simics instances have finished their slices and then makes FSIN run for a period of time equivalent to the Simics slice. After FSIN finishes its slice, TrGen sends a multicast message to the Simics instances, allowing them to resume their execution. The ratio between Simics and FSIN slices (one measured in Simics cycles and the other one in FSIN cycles) determines the bandwidth of the simulated network.

Further details about performing execution-driven simulation within INSEE can be found in [41]. This work includes a thorough description of the full-system simulation in INSEE, and discusses several different approaches that may be followed in order to obtain usable results from this kind of simulation. Emphasis is put on where to capture the traffic in the simulated node, and also in synchronization, showing the need to fine tune the value of the slices in order to obtain a balance between simulation accuracy and execution time. Furthermore it discusses some issues encountered when setting up the simulation environment and carrying out performance-related experiments.

3.4. Task placement

An important consideration to take into account when launching parallel applications is the mapping of application tasks onto computing resources. Parallel applications are usually implemented following spatial distributions that can be exploited effectively by means of an adequate allocation onto the nodes. Furthermore, supercomputing sites are often used by many users that share machine resources, with several applications being executed simultaneously on different system partitions. Our previous research showed that a bad application placement may slow down the communication of applications sharing a parallel computing up to 10 times [28]. We also showed that an improvement of 10–15% in execution time of applications, obtained through a good placement policy, can compensate the high cost of a topology-aware scheduling [32].

When dealing with application (or application-inspired) traffic, INSEE has the ability of arranging the tasks of a workload onto the whole network, or onto network partitions, following different policies. It can also execute several application instances concurrently, to measure the effects of the interactions among them. Currently, INSEE supports only the simultaneous execution of several instances of the same application [28].

Some examples of the different placement strategies available for direct networks are depicted in Fig. 12. In the depictions, four applications, composed by four nodes each, share the network (these applications are represented by white, grey, black and crossed circles, correspondingly). Similarly, some example depictions of the different strategies for tree-like topologies are shown in Fig. 13. The definitions of the placement policies are as follows:

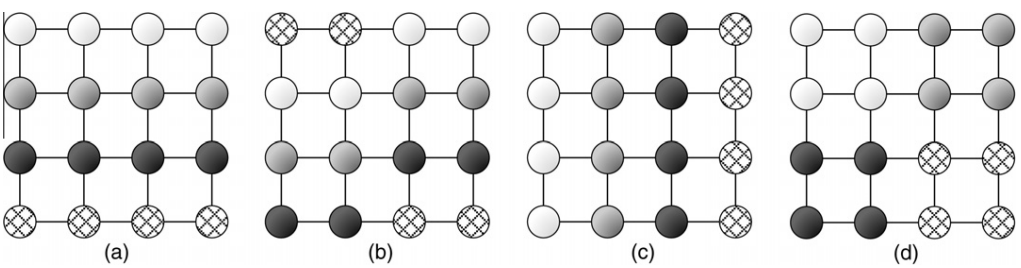


Fig. 12. Placement strategies for direct topologies: (a) row; (b) shift 2; (c) column and (d) quadrant.

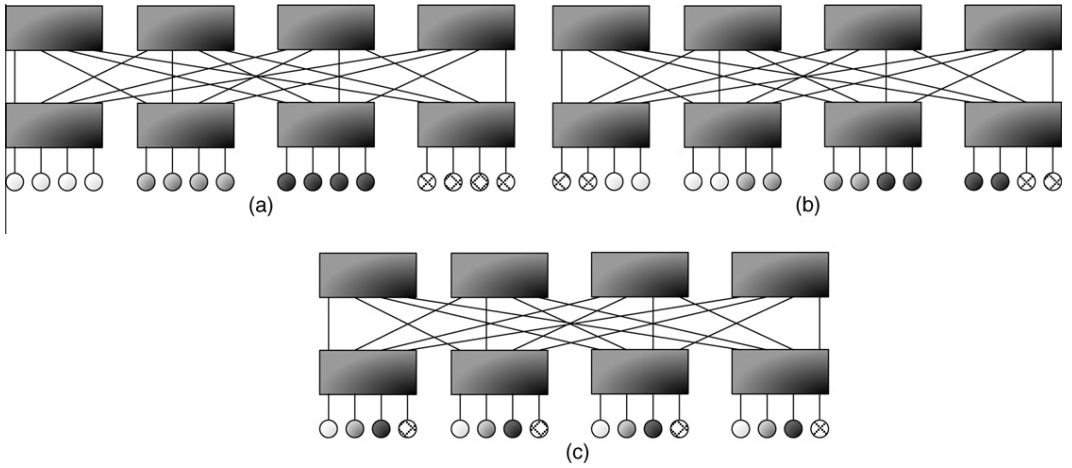


Fig. 13. Placement strategies for tree-like topologies: (a) consecutive; (b) shift 2 and (c) shuffle.

- *Consecutive/row*: Tasks and applications are placed consecutively—note that in the case of cube topologies this means filling rows in order. For example if we have an application of n nodes, it will be placed from nodes 0 to $n-1$, being its tasks arranged in order. Examples of these placements are plotted in Fig. 12a for cubes and Fig. 13a for trees.
- *Shift*: This policy is like the previous one, but adding a given shift s to the allocated node, that is, task t is located in node $t+s$. Examples of these placements are plotted in Fig. 12b for cubes and Fig. 13b for trees.
- *Shuffle*: Tasks are placed in order in the nodes attached to the first port of each switch, then to those nodes attached to the second port of each switch, and so on. If in each switch there are p ports connected to compute nodes, then the tasks will be placed in nodes: 0, p, $2p$, ..., 1, $p+1$, $2p+1$, and so on. An example of this placement is plotted in Fig. 13c. Note that this policy only makes sense in those topologies with many compute nodes attached to each network element, i.e. indirect topologies.
- *Column*: This policy only makes sense in cube-like topologies with, at least, two dimensions. Assignment is done selecting the nodes by columns, which can be seen as partitioning the network in rectangular sub-networks, taller than wide. An example of this placement is plotted in Fig. 12c.
- *Quadrant*: This policy only makes sense in cube-like topologies with, at least, two dimensions. When using several application instances, we can partition the network in perfect squares (or cubes). Within each square an application is placed following consecutive order. An example of this placement is plotted in Fig. 12d.
- *Random*: All the tasks are randomly placed along the network independently of their application. To do so, we generate a random permutation of the network nodes.
- *File*: INSEE can read the mapping information from a file. The format of this file is very simple: ⟨*node*, *task, application*⟩, meaning that the task *task* from the application *application* is mapped onto node *node*. Note that this mode allows us to test more complex mapping techniques, such as optimized task-to-node placement [31].

4. Limitations of INSEE

Although INSEE has proven to be a useful tool to evaluate interconnection systems, it has some limitations that we should not forget. Some of these limitations are easily solvable, but may require considerable effort in terms of implementation time. Additionally, some modifications may increase excessively memory requirements and/or execution times, something that would go against the philosophy of INSEE. Still, in this section we discuss some modifications that could increase the usefulness of this environment.

FSIN uses internally a time-driven engine that it is not suitable to simulate workloads with long computation times between communication events, because it wastes too much time doing nothing but making the clock advance. We are currently developing an event-driven engine for FSIN which would accelerate the simulation with low density of events.

The node model, simulated as a simple traffic generator and consumer without any internal structure, is too simplistic. We designed it this way because we are interested in the behavior of the IN, and this model is sufficient for this purpose. Another limitation closely related to this one is that INSEE only allows allocating a single task per node. This is not very realistic because, in current systems, nodes attached to the network are actually multiprocessors. The simulation of multiprocessor nodes can be implemented in TrGen just by allowing several application tasks (from Simics, traces or application kernels) to share a FSIN node. However it will require making some implementation decisions: how to arbitrate the injection infrastructure and how to perform intra-node communication. The added complexity would result in slower simulation times and larger footprints.

When performing trace-driven simulations, the causality of the messages is maintained, but the MPI semantics are not followed accurately by TrGen. For instance, all messages sends are treated equally, regardless of them being immediate, *rendevouz*, one-sided, etc. Implementing MPI semantics accurately would require increasing the complexity of the node which, as stated before, is not desirable. Furthermore, MPI has very complex (and precise) semantics, and implementing every operation supported by the standard would require huge efforts in terms of fully understanding the involved details and coding them within TrGen.

A remarkable limitation of INSEE is that FSIN does not include detailed models of some *state-of-the-art* networking technologies such as InfiniBand [45] or Myrinet [8]. The reason for this is two-folded. On the one hand, implementing these technologies would require a deep knowledge of every detail of them, as well as a non-negligible effort in terms of coding. On the other hand, given the complexity of these technologies, it would not be possible to perform large-scale simulations because of the associated requirements in terms of computing resources.

An arguably less significant limitation is that INSEE does not allow for parallel or distributed simulation. Although FSIN could be easily extended in this line—note that its time-driven engine is split into two separate loops and, in each loop, iterations are completely independent among them—this has not been done for two simple reasons. The first one is that the low footprint of INSEE allows for large-scale simulations in a single off-the-shelf computer, and therefore we did not *need* to parallelize it, even when we have parallel computers. This leads to the second reason: in our evaluations we usually require the execution of *many* experiments (hundreds, even thousands), modifying parameters such as network size, workload, placement and random seed, without any dependency between the different runs. The utilization of a cluster as a high-throughput computing resource is enough for this purpose.

5. Performance evaluation of INSEE

As stated before, INSEE is a lightweight tool that has been developed with resource scarcity in mind. In this section we show how the resource requirements scale with the size of the system to simulate, using a collection of realistic experiments in which we vary those parameters that affect memory and CPU usage. The values plotted in the figures correspond to the average value of 10 runs. Confidence intervals (99%) are three orders of magnitude below the average and therefore are not plotted.

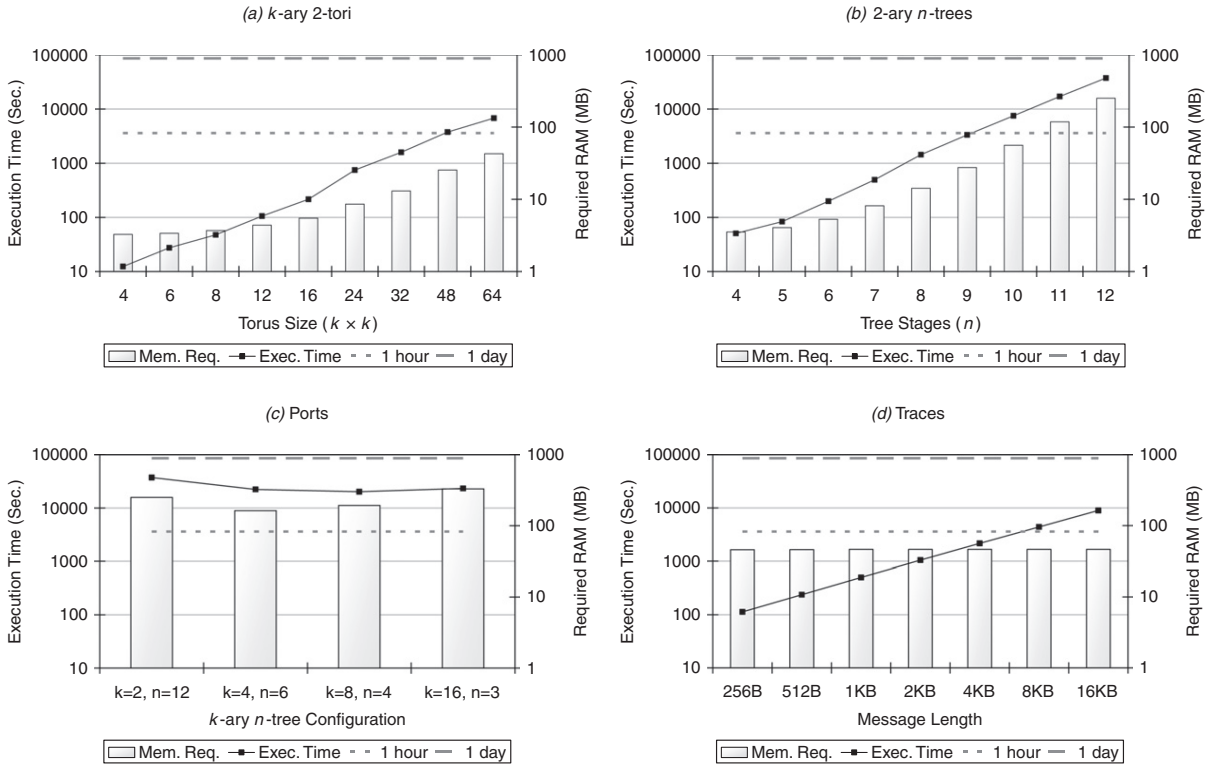


Fig. 14. Resources needed to perform simulation.

The simulation runs used in this section use parameter values that are commonly used in our day-to-day research. Tori use 2D bubble routers using the smart request strategy, while trees use the adaptive routing strategy. All physical channels are split into two virtual channels, each of them with a queue with room for storing four packets of 16 phits (physical units – 4 bytes). Fig. 14 shows plots of simulation time and memory footprint obtained from a series of simulation runs performed in an Intel(R) Pentium(R) 4 CPU working at 3.00 GHz with 1.5 GB of RAM. Note that the instrumentation introduced to obtain these measurements has a negative effect on them, and also that executions on more recent processors would be significantly faster.

In the plots both *X* and *Y* axis show logarithmic values. Two dotted lines representing respectively an hour and a day are added as hints to better understand the reported times.

A first study whose results are depicted in Fig. 14a, shows the scalability of INSEE when simulating tori. Systems arranged in a wide variety of sizes, from 4×4 (16 nodes) to 64×64 (4096 nodes), are studied,with a workload modeled as uniform traffic from independent sources at the maximum possible pace. We can see how execution time and memory consumption scale roughly linearly with the number of nodes. This is because the main memory consumption is due to the room in the queues, and the number of queues depends on the number of nodes.

In the case of the trees, depicted in Fig. 14b, we simulated 2-ary *k*-trees with the *k* parameter varying from $k = 4$ (16 nodes) to $k = 12$ (4096). The workload was also uniform traffic from independent sources at full pace. The number of switches scales in $O(n \log n)$ with the number of nodes and, therefore, so do memory consumption and execution time, as can be seen in the figure.

In Fig. 14c, four different fat-tree topologies, all able to connect 4096 nodes, are fed with uniform traffic from independent sources. We can see how the choice of topology affects noticeably the resources required to perform a simulation. In this case, both memory usage and execution time can increase up to a 400% just by using one topology or another. In the plot we can clearly seen that the sweet spot seems to be the 4-ary 6-tree.

Finally, in Fig. 14d, we generated several instances of butterfly, an application inspired workload, all of them for 4096 nodes but varying the message length from 256 bytes (1 packet) to 16 kB (64 packets). We run these workloads in a 2D torus with 64 nodes per dimension. In the plot we can see how the message size affects linearly the execution time, but the memory does not change from configuration to configuration. This is because the size of the data structures used to store messages in trace-driven simulation do not depend on the length of the (simulated) messages. However, as the simulation operates at the phit level, longer messages translate on more phits managed by the network, and this requires more time.

In general, we can conclude that INSEE's memory requirements grow roughly linearly with the number of simulated ports, which in turn depends on the number of switches and their complexity. Execution time grows with the number of ports too, but also strongly depends on the characteristics of the simulated workload. At any rate, authors want to highlight that INSEE's footprint and execution speed in current computers (more up-to-date than the one used in this evaluation) are austere: most of our experiments are completed in a few hours, using only a few hundred MB of RAM. In other words, INSEE does not have prohibitive requirements.

6. Related work

We want to remark that INSEE is only one of the network simulators available to the community. In the literature we can find references for many of them. Let us review a small selection, pinpointing the main differences with INSEE. Note that, as explained before, the main characteristics of INSEE are its flexibility and its low resources requirement, and therefore it outperforms in these two aspects to most of the tools revised here.

SICOSYS [36,48], developed at the University of Cantabria, has very detailed models of several router architectures, which allows obtaining very accurate performance measurements, close to those obtained with a hardware-level simulator, but at a fraction of the required computing resources. It is noticeably more complex and resource-demanding than INSEE and, therefore, it is restricted to simulate networks composed by a few hundred nodes at most. SICOSYS is implemented in C++, and has been used for evaluations focusing on performance and on fault-tolerance of networks-on-chip. In addition, it allows feeding the simulator with several traffic models with different levels of detail: synthetic traffic and traces of MPI or ccNUMA applications. It can also interface with RSIM [30] and SIMOS [43] to perform full-system simulation of INs.

The Chaos router simulator [51], from the University of Washington, has detailed models of *k*-ary *n*-cube networks (meshes, tori and hypercube); tree-like topologies are not implemented. Switching can be both packet-switching and wormhole. Routers can implement dimension-order oblivious routing or, alternatively, Chaotic adaptive routing [18]. It also incorporates a wide variety of synthetic traffic patterns to feed the simulation. A distinguishing characteristic of this simulator is that it may also run in animated mode, in which the temporal evolution of the simulation is graphically represented. One of the main drawbacks of this simulator is that all the changes in the design of the network to evaluate must be done at compilation time, in other words, every change in the model requires the application re-compilation. It is remarkable that, as far as we know, development of this simulator was stopped in 1996, but its source code is still available.

FlexSim [49], developed at the University of Southern California, is a C-based simulator for *k*-ary *n*-cube networks with any number of dimensions greater or equal than 2, any power of two number of nodes per dimension, and any arbitrary number of virtual channels. It has support for several router models, synthetic traffic patterns and failure models. This simulator has been used to conduct research in fault-tolerance and deadlock-free routing.

MARS [12] is a simulator of parallel systems developed at IBM and based on the OMNeT++ simulation framework [52]. 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 multi-stage topologies, and a variety of switching and routing functions. It supports multi-core configurations in which each processing core has its own MPI stack. The main difference between MARS and INSEE trace-driven simulations is the model of the node, and the conformity to MPI semantics; these differences make INSEE faster, while MARS is *a priori* more accurate.

MINSimulate [47], developed at the Technical University of Berlin, is a simulator designed to evaluate multi-stage INs. It implements Clos and Delta networks and supports both wormhole and store and forward switching. Note that currently INSEE does not support this kind of networks but their inclusion would require insignificant efforts as an implementation of a multi-stage switch is already available.

The NS-2 simulator [15], from the University of Southern California, is designed to research on wired and wireless TCP-based communication networks. Although high-performance computing systems use to rely on high performance interconnects such as InfiniBand [45] or Myrinet [8] for parallel computing, most of them support Ethernet-based networks for storage and control purposes. Furthermore, new versions of Ethernet 10 GB or the early-to-come 100 GB Ethernet can be used as low-cost INs. For these reasons we include NS-2 in this review.

BigNetSim [53], 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 for 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 behavior of applications developed for those systems. In contrast with INSEE, 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 re-compile the target modules.

Dimemas [6], developed and maintained at the Barcelona Supercomputing Center, was designed with the evaluation of applications behavior in mind. It can reconstruct the execution of a parallel application in any supported architecture using a trace of that application. Dimemas philosophy is similar to INSEE: keep it simple. 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 modeled 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. Note that while INSEE can be used for this purpose, it is not specifically designed for this function.

The COTSon Infrastructure for system-level simulation by HP Labs [4] provides a simulation environment very similar to that given by the combination FSIN+ Simics, but based on AMD's SimNow [3]. The tool has been open sourced in January 2010. It is able to simulate multi and many-core machines, and also clusters using a functional simulator of a network switch. We plan to further study the potential of its pluggable architecture, in order to check if FSIN could be integrated into this infrastructure.

We close this section with Table 3, in which the differences among the discussed simulators are summarized. Note that the NS-2 is not included because of its completely different nature.

7. Final remarks and future work

In this paper, we have thoroughly described INSEE, our Interconnection Network Simulation and Evaluation Environment [50]. This description includes all the router architectures and network organizations currently implemented in FSIN, our Functional Simulator of Interconnection Networks, as well as all the accepted router parameters and captured statistics. Furthermore, all the traffic models and procedures to generate workloads provided by TrGen, our traffic generator module, are

Table 3
Summary of the characteristics of the reviewed simulators.

	Family of topologies			Level of detail		Traffic models			Resources requirement
	Cube	Tree-like	Multi-stage	Network	Nodes	Synthetic	Traces	Full-system	
INSEE	√	√	×	High	Low	√	√	√	Low
SICOSYS	√	×	×	High	Low	√	√	√	High
Chaos	√	×	×	High	Low	√	×	×	Medium
Flexsim	√	×	×	High	Low	√	×	×	Medium
Mars	×	√	√	High	High	×	√	×	High
MINSimulate	×	×	√	High	Low	√	×	×	Medium
COTSon	√	√	√	Low	Very high	×	×	√	Very High
BigNetSim	√	√	√	Variable	Variable	×	√	×	Variable
Dimemas	√	√	√	Low	High	×	√	×	Low

explained in detail. The workloads can be synthetic, with different degrees of fidelity to actual applications, or application-based using traces and full-system simulation.

INSEE tools have been successfully used in our research group to carry out a wide variety of studies in the field of performance evaluation of INs, including: the effect of Head-of-Line blocking at injection [17], the impact in the performance of several injection interfaces as congestion control mechanisms [16], the performance of local congestion control mechanisms [21,42], the evaluation of several topological proposals [9,24,26], and the effect of task and node allocation on the performance of parallel applications [28,31], among others. A remarkable feature of INSEE is that it allows simulating large-scale networks of up to 64 K nodes in a regular off-the-shelf desktop computer with 2 GB of RAM.

We plan to add new features and operation modes to the environment. Some of them will be driven by the requirements of our research work. Some others are oriented to improve the behavior of the environment. These new additions include, but are not limited to the following.

- A new, event-driven simulation engine for FSIN is under development, in order to allow faster simulation of applications while properly modeling the network. This implementation will be further extended to be used in the field of job scheduling, allowing running together several applications as well as their management. Together with the integration of this new engine, we plan to add the capability to simulate several processing units in each compute node in order to properly simulate current clusters and supercomputers.
- We plan to model new router architectures. For example, a more detailed model of the SpiNNaker router will be added to perform a more realistic evaluation using neural-activity workloads for this target system. Other router architectures than can be added to INSEE are the HPAR router [37] or the rotary router [1]. Other subsystems to be modeled are the Immunet [38] and ImmuCube [35] fault-tolerance schemes.
- We will continue with our work on characterizing realistic workloads, in order to increase our collection of application kernels. These kernels will be used in future evaluation studies in which realistic traffic models are required to provide accurate results. One interesting starting point is the review of typical high-performance computing applications that can be found in [5], in which 13 *dwarves* are identified. Each of these dwarves represents a prototype of application because of the communication patterns, the coupling of the task or several other details.

Acknowledgements

This work has been supported by the Ministry of Education and Science (Spain), Grant TIN2007-68023-C02-02, and by Grant IT-242-07 from the Basque Government. Dr. Navaridas is supported by a post-doctoral grant of The University of the Basque Country. Mr. Pascual is supported by a doctoral grant of the Basque Government.

References

[1] P. Abad, V. Puente, P. Prieto, J.A. Gregorio, Rotary router: an efficient architecture for CMP interconnection networks, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, CA, USA, June, 2007, pp. 116–125. doi: 10.1145/1250662.1250678.
[2] N.R. Adiga, M.A. Blumrich, D. Chen, P. Coteus, A. Gara, M.E. Giampapa, P. Heidelberger, S. Singh, B.D. Steinmacher-Burow, T. Takken, M. Tsao, P. Vranas, Blue Gene/L torus interconnection network, IBM Journal of Research and Development 49 (2/3) (2005).
[3] Advanced Micro Devices Inc., AMD SimNow Simulator. <http://developer.amd.com/cpu/simnow>.
[4] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, D. Ortega, COTSon: infrastructure for full system simulation, ACM SIGOPS Operating Systems Review 43 (1) (2009) 52–61.
[5] K. Asanovic, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>.
[6] R.M. Badia, J. Labarta, J. Gimenez, F. Escale, DIMEMAS: Predicting MPI applications behavior in Grid environments, Workshop on Grid Applications and Programming Tools, June, 2003.
[7] R. Beivide, E. Herrada, J.L. Balcazar, A. Arruabarrena, Optimal distance networks of low degree for parallel computers, IEEE Transactions on Computers 40 (10) (1991) 1109–1124, doi:10.1109/12.93744.
[8] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.K. Su, Myrinet: a gigabit-per-second local area network, IEEE Micro 15 (1) (1995) 29–36, doi:10.1109/40.342015.
[9] J.M. Camara, M. Moreto, E. Vallejo, R. Beivide, J. Miguel-Alonso, C. Martinez, J. Navaridas, Twisted torus topologies for enhanced interconnection networks, Accepted for Publication in the IEEE Transactions on Parallel and Distributed Systems, in press. doi: 10.1109/TPDS.2010.30.
[10] W.J. Dally, C.L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, IEEE Transactions on Computers 36 (5) (1987) 547–553, doi:10.1109/TC.1987.1676939.
[11] W.J. Dally, B. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann Series in Computer Architecture and Design, 2004. ISBN: 0-12-200751-4.
[12] W.E. Denzel, J. Li, P. Walker, Y. Jin, A Framework for End-to-End Simulation of High Performance Computing Systems, SIMUTools'08, Marseille, France, March 3–7, 2008.
[13] J. Duato, A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks, IEEE Transactions on Parallel and Distributed Systems 6 (10) (1995) 1055–1067, doi:10.1109/71.473515.
[14] S. Goldschmidt, J. Hennessy, The accuracy of trace-driven simulation of multiprocessors, in: ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems, May, 1993, pp. 146–157. doi: 10.1145/166962.167001.
[15] Information Science Institute, Network Simulator ns-2. <http://www.isi.edu/nsnam/ns/>.
[16] C. Izu, J. Miguel-Alonso, J.A. Gregorio, Evaluation of interconnection network performance under heavy nonuniform loads, in: Lecture Notes in Computer Science, Proc. ICA3PP, vol. 3719/2005, 2005, pp. 396–405.
[17] C. Izu, J. Miguel-Alonso, J.A. Gregorio, Effects of injection pressure on network throughput, in: Proc. PDP 2006 14th Euromicro Conference on Parallel, Distributed and Network Based Processing, Montbéliard-Sochaux, France, February 15–17, 2006. doi: 10.1109/PDP.2006.32.
[18] S. Konstantinidou, L. Snyder, The Chaos router, IEEE Transactions on Computers 43 (12) (1994) 1386–1397, doi:10.1109/12.338098.

[19] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, IEEE Computer 35 (2) (2002) 50–58, doi:10.1109/2.982916.
[20] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. <http://www-unix.mcs.anl.gov/mpi/standard.html>.
[21] J. Miguel-Alonso, C. Izu, J.A. Gregorio, Improving the performance of large interconnection networks using congestion-control mechanisms, Performance Evaluation 65 (3) (2008) 203–211, doi:10.1016/j.peva.2007.05.001.
[22] J. Miguel-Alonso, J. Navaridas, F.J. Ridruejo, Interconnection network simulation using traces of MPI applications, International Journal of Parallel Programming 37 (2) (2009) 153–174, doi:10.1007/s10766-008-0089-y.
[23] M. Mirza-Aghatabar, S. Koohi, S. Hessabi, M. Pedram, An empirical investigation of mesh and torus NoC topologies under different routing algorithms and traffic models, in: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, Lübeck, Germany, August 29–31, 2007, pp. 19–26. doi: 10.1109/DSD.2007.28.
[24] J. Navaridas, M. Luján, J. Miguel-Alonso, L.A. Plana, S.B. Furber, Understanding the interconnection network of SpiNNaker, in: proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY, USA, June 8–12, 2009, pp. 286–295. doi: 10.1145/1542275.1542317.
[25] J. Navaridas, J. Miguel-Alonso, Realistic evaluation of interconnection networks using synthetic traffic, in: 8th International Symposium on Parallel and Distributed Computing, Lisbon, Portugal, June 30–July 4, 2009.
[26] J. Navaridas, J. Miguel-Alonso, F.J. Ridruejo, W. Denzel, Reducing complexity in tree-like computer interconnection networks, Accepted for Publication in the International Journal on Parallel Computing, in press. doi: 10.1016/j.parco.2009.12.004.
[27] J. Navaridas, J. Miguel-Alonso, F.J. Ridruejo, On synthesizing workloads emulating MPI applications, in: The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing, Miami, Florida, USA, April 14–18, 2008. doi: 10.1109/IPDPS.2008.4536473.
[28] J. Navaridas, J.A. Pascual, J. Miguel-Alonso, Effects of job and task placement on parallel scientific applications performance, in: Proc 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Weimar, Germany, February 18–20, 2009, pp. 55–61. doi: 10.1109/PDP.2009.53.
[29] J. Navaridas, L.A. Plana, J. Miguel-Alonso, M. Luján, S.B. Furber, SpiNNaker: effects of traffic locality and causality on the performance of the interconnection network, Submitted to the ACM International Conference on Computing Frontiers, 2010.
[30] V.S. Pai, P. Ranganathan, S.V. Adve, RSIM: an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors, in: IEEE Technical Committee on Computer Architecture Newsletter, October, 1997.
[31] Jose A. Pascual, Jose Miguel-Alonso, Jose A. Lozano, Optimization-Based Mapping Framework for Parallel Applications. Technical Report EHU-KAT-IK-02-10, University of the Basque Country, April, 2010. Submitted to Elsevier's Journal of Parallel and Distributed Computing.
[32] J.A. Pascual, J. Navaridas, J. Miguel-Alonso, Effects of topology-aware allocation policies on scheduling performance, in: Proc. 4th Workshop on Job Scheduling Strategies for Parallel Processing in Conjunction with IPDPS 2009, Rome, Italy, Lecture Notes in Computer Sciences, vol. 5798/2009, May 29, 2009, pp. 138–156. doi: 10.1007/978-3-642-04633-9_8.
[33] F. Petrini, M. Vanneschi, *k*-ary *n*-trees: high performance networks for massively parallel architectures, in: Proceedings of the 11th International Parallel Processing Symposium, Geneva, Switzerland, 1–5 April, 1997, pp. 87–93. doi: 10.1109/IPPS.1997.580853.
[34] L.A. Plana, S.B. Furber, S. Temple, M.M. Khan, Y. Shi, J. Wu, S. Yang, A GALS infrastructure for a massively parallel multiprocessor, IEEE Design and Test of Computers 24 (5) (2007) 454–463, doi:10.1109/MDT.2007.149.
[35] V. Puente, J.A. Gregorio, Immucube: scalable fault-tolerant routing for *k*-ary *n*-cube networks, IEEE Transactions on Parallel and Distributed Systems 18 (6) (2007) 776–788, doi:10.1109/TPDS.2007.1047.
[36] V. Puente, J.A. Gregório, R. Beivide, SICOSYS: an integrated framework for studying interconnection network performance in multiprocessor systems, in: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, Canary Islands, Spain, January 9–11, 2002, pp. 15–22. doi: 10.1109/EMPDP.2002.994207.
[37] V. Puente, J.A. Gregorio, R. Beivide, C. Izu, On the design of a high-performance adaptive router for CC-NUMA multiprocessors, IEEE Transactions on Parallel and Distributed Systems 14 (5) (2003), doi:10.1109/TPDS.2003.1199066.
[38] V. Puente, J.A. Gregorio, F. Vallejo, R. Beivide, Immunet: dependable routing for interconnection networks with arbitrary topology, IEEE Transactions on Computers 57 (12) (2008) 1676–1689, doi:10.1109/TC.2008.95.
[39] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo, J.M. Prellezo, The adaptive bubble router, Journal of Parallel and Distributed Computing 61 (9) (2001) 1180–1208, doi:10.1006/jpdc.2001.1746.
[40] F.J. Ridruejo, J. Miguel-Alonso, INSEE: an interconnection network simulation and evaluation environment, in: Lecture Notes in Computer Science, Proc. Euro-Par, vol. 3648/2005, 2005, pp. 1014–1023.
[41] F.J. Ridruejo, J. Miguel-Alonso, J. Navaridas, Full-System Simulation of Distributed Memory Multicomputers, Cluster Computing, Published Online, March 28, 2009. doi: 10.1007/s10586-009-0086-y.
[42] F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso, C. Izu, Realistic evaluation of interconnection network performance at high loads, in: 8th International Conference on Parallel and Distributed Computing Applications and Technologies, Adelaide, Australia, December 3–6, 2007, pp. 97–104. doi: 10.1109/PDCAT.2007.73.
[43] M. Rosenblum, S.A. Herrod, E. Witchel, A. Gupta, Complete computer system simulation: the SimOS approach, Parallel and Distributed Technology: Systems and Applications 3 (4) (1995) 34–43, doi:10.1109/88.473612.
[44] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, R.M. Needham, T.L. Rodeheffer, E.H. Satterthwaite, C.P. Thacker, Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-point Links, SRC Research Report 59, December, April 21, 1990.
[45] T. Shanley, InfiniBand Network Architecture. Addison-Wesley, 2002 (November). ISBN: 978-0-321-11765-6.
[46] B. Towles, W.J. Dally, Worst-case traffic for oblivious routing functions, IEEE Computer Architecture Letters 1 (1) (2002), doi:10.1109/L-CA.2002.12.
[47] D. Tutsch, M. Brenner, D. Luedtke, A. Walter, MINSimulate. <http://dontcry.pdv.cs.tu-berlin.de/minsimulate/index.html>.
[48] University of Cantabria, SICOSYS. <http://www.atc.unican.es/SICOSYS/>.
[49] University of Southern California, Information on FlexSim1.2. <http://ceng.usc.edu/smart/FlexSim/flexsim.html>.
[50] The University of the Basque Country, INSEE. <http://insee.sourceforge.net/>.
[51] University of Washington, The Chaos Router Simulator. <http://www.cs.washington.edu/research/projects/lis/chaos/www/simulator.html>.
[52] A. Vargas, The OMNeT++ Discrete Event Simulation System. <http://www.omnetpp.org/download/docs/papers/esm2001-meth48.pdf>.
[53] G. Zheng, T. Wilmarth, P. Jagadishprasad, L.V. Kalé, Simulation-based performance prediction for large parallel machines, International Journal of Parallel Programming 33 (2–3) (2005), doi:10.1007/s10766-005-3582-6.

Chapter 9. Trace-based workloads

Full reference:
J. Miguel-Alonso, J. Navaridas, and F. J. Ridruejo.
Interconnection network simulation using traces of MPI applications.
International Journal of Parallel Programming, Volume 37, Issue 2, April 2009, 153-174

Very early in our work we realized that synthetically generated traffic was not realistic enough when carrying out some performance evaluation studies and, thus, decided to enhance TrGen with additional mechanisms. This paper focuses on analyzing possible methodologies for generating trace-based traffic and describes the actual implementation in TrGen. We discuss different approaches to capture traces from real application runs, the way they can be used during the simulation, and the limitations of this approach. A major contribution of this paper is the methodology used to replicate causal relationships between messages, detecting them and triggering the generation of new messages when a previous is received. The result is a traffic injection mechanism that accurately replicates the interchange of messages between application tasks.

Two case studies carried out using this trace-based traffic generation are provided: the evaluation of routing strategies and virtual channel management, and the performance estimation of a real application on three different multicomputers. In other publications, this mechanism has been used to study novel topologies.

An additional feature of trace-based simulation is that it opens within INSEE the possibility of carrying out performance predictions of applications running on different IN configurations: topologies, routing techniques, congestion control mechanisms, among others.

In this publication the author has worked on the capture of (extended) trace files from MPICH and on the design of the TrGen interface with INSEE. Also, the author has carried out the experimental work for the case studies.

Int J Parallel Prog (2009) 37:153–174
DOI 10.1007/s10766-008-0089-y

Interconnection Network Simulation Using Traces of MPI Applications

J. Miguel-Alonso · J. Navaridas · F. J. Ridruejo

Received: 25 October 2007 / Accepted: 7 November 2008 / Published online: 6 December 2008

Abstract This paper addresses the utilization of traces taken from MPI applications to do simulation-based performance studies of parallel computing systems. Different mechanisms to capture traces are discussed, pointing out important limitations of some of them. One of these limitations is the invisibility of message interchanges in collective operations, which is circumvented modifying a trace-capturing library. During a simulation, trace records must be simulated in causal order, to fully comply with application semantics. Alternatives to follow this order, and the risks of not following it, are presented and discussed. The techniques introduced in this paper have been implemented in an in-house developed simulation environment, which is used in two example studies to show its usefulness: an evaluation of alternatives for interconnection network design, and a performance prediction study in which traces from one machine are used to estimate the execution times of applications running in a different machine.

Keywords Interconnection network simulation · Traces of parallel applications · Message passing interface

J. Miguel-Alonso (✉) · J. Navaridas · F. J. Ridruejo
Department of Computer Architecture and Technology, The University of the Basque Country,
P.O. Box 649, 20080 San Sebastian, Spain
e-mail: j.miguel@ehu.es

J. Navaridas
e-mail: javier.navaridas@ehu.es

F. J. Ridruejo
e-mail: franciscojavier.ridruejo@ehu.es

1 Introduction

Simulation is one of the most widely used tools for performance evaluation of computing systems, including parallel computers. A simulation-based study requires a model of the system being evaluated (a model that may have very different levels of accuracy) and also a mechanism to supply a *representative* workload.

In the field of interconnection networks, in which we place this paper, many simulation-based studies use synthetic traffic patterns, such as random uniform traffic or permutations of interest, to feed the simulator. This kind of synthetic workload is of great interest because it is easy to implement, sometimes it may support analytical studies, and may be representative of the ways applications use the network. However, a comprehensive evaluation requires actual workloads, otherwise important aspects of parallel applications cannot be understood in detail. For example, many applications pass through different phases, in which the ways of using the network differ widely; pressure on the network may be very intense in some phases, but the inter-dependencies amongst processes may lower the utilization of the interconnection infrastructure in some others.

Actual traffic may be generated using an execution-driven environment, in which applications run on real (or simulated) processors and are connected to a simulated interconnection network. This set-up provides very high levels of evaluation accuracy, but cannot be easily scaled to thousands of processors. It may also fail, victim of unexpected interactions between components, as shown in [16,24]. For this reason, a frequently used alternative is the utilization of traces of parallel applications.

We can obtain traces of large systems, even using small ones. For example, given an application that uses a static logical topology, a cluster of 10 PCs can be used to generate a trace running on 200 nodes—we only need to run 20 processes per available computer. Timing information would not be representative of a real, 200-CPU computer; however, the (spatial) patterns defined by the sequence of interchanged messages are valid, and the distribution of message sizes is valid too.

The main contribution of this research work is the trace-based simulation toolset included in INSEE [23], an in-house developed simulation environment for the evaluation of interconnection networks. INSEE is able to accept many kinds of workloads for the simulation: synthetically generated messages, full system simulation and traces. The focus of this paper is, precisely, on simulation with traces. We explain the mechanisms required to capture traces from MPI-based parallel applications, point out some important limitations in the way events are captured and stored in the trace files, and explain the way we circumvent some of those limitations. After that, we describe in detail the way traces must be processed in order to fully comply with application semantics, with particular attention to the order in which events are processed to avoid violations of message causality. A simple extension of the trace processing mechanism allows us to carry out performance prediction studies. Finally, we put INSEE to work and include two simple example studies carried out with traces from actual applications. Note that the problems and solutions discussed in this paper, although presented from the point of view of the INSEE developers and users, are not specific to this tool.

The rest of this paper is organized as follows. Section 2 introduces INSEE, with focus on its utilization with traces. Section 3 describes how traces are obtained. Sections 4 and 5 discuss the mechanisms used by our simulator to process traces, preserving application semantics. In Sect. 6 we use INSEE for two performance studies. Section 7 reviews some related work. Conclusions of the paper are summarized in Sect. 8.

2 Evaluation of Networks Using Simulation: INSEE

In this section we briefly introduce INSEE [23], the Interconnection Network Simulation and Evaluation Environment developed at the University of the Basque Country. The two main elements of INSEE are FSIN, a Functional Simulator of Interconnection Networks, and TrGen, a Traffic Generator [22].

FSIN has been designed to provide a fast simulation engine for interconnection networks, both direct (meshes and tori) and multistage (trees, including fat-trees), with different architectural characteristics. Its small footprint allows us to simulate, on an off-the-shelf desktop computer, large size networks: we have carried out experiments with 64K-node networks using less than 2 GB of RAM. In the case of direct networks, each node represents a router attached to a computing element which, in fact, is the source (and sink) of the traffic managed by the network. In the case of trees, computing elements are attached to switches at the lowest level, using interface cards.

The management of workloads is carried out by TrGen. A workload is a collection of messages that are generated by computing elements (actually, by message sources), then packetized and passed to FSIN (which simulates the way they traverse the interconnection network), then reassembled and, finally, delivered to the computing elements (actually, to the message sinks). When generating a workload, we have to define

- The *spatial* distribution of messages: source nodes and destination nodes.
- The *size* distribution of the messages. These come in many sizes, depending on the application.
- The *temporal* distribution of the generation of messages. In some cases, workload generators simply generate random numbers (following a certain distribution) that determine the inter-generation intervals. In some others traffic is *reactive*, meaning that there are *causal relationships* between them: the arrival of a message to a certain destination node triggers the generation of a new message from that node.

Very simple *synthetic workloads* use statistical distributions to generate destinations, sizes and inter-generation times. A special class of this traffic is what we call application-inspired traffic [17], for which we emulate the behavior of some kernels of scientific applications, including causality. It is important to point out that simulations using traces, and full system simulation, use spatial, size and temporal distributions exactly as defined by the application that was instrumented, or that is being executed. When performing *full system simulations*, an external toolset based on Simics [11] fully simulates a collection of computing nodes (including hardware, drivers, operating system, message passing library, and running application) attached to an interconnection network, which is simulated by FSIN [16]. To perform *trace based*

simulation, we use traces obtained using the mechanisms provided by MPI implementations, although some modifications to these are required in order to get *extended* trace files, because FSIN only deals with point-to-point operations. An extended trace file includes the detailed message passing involved in collective operations, something that is not visible in *regular* trace files.

We stated before that FSIN allows the simulation of very large networks. Unfortunately, this can be done *only* with synthetic traffic. The trace capturing environment, or the ability to fully simulate collections of computers, limits the node count for the other traffic generation arrangements. At any rate, the focus of this work is on trace based simulation.

In the following sections we will explain the way INSEE deals with traces, from the mechanism used to capture them, to the way they are consumed by the simulator. We will explain how traces can be used not only for performance evaluation but also for performance prediction.

3 Generation of Traces from MPI Applications

MPICH [9] is one of the most widely used implementations of the Message Passing Interface (MPI) [12], a standard programming interface for parallel applications based on processes that communicate and synchronize explicitly, interchanging messages. The MPI standard defines a profiling mechanism called PMPI ("P" from Profiling) that allows programmers to intercept all calls to MPI functions. This mechanism is often used to implement libraries to generate traces of applications, or to obtain profiling information.

3.1 Generating Trace Files with MPE

The MPICH distribution includes MPE (Multi-Processing Environment) [10], a set of libraries and tools to generate and analyze traces of parallel applications. The tracing ability is based on PMPI, so MPE can be used with any MPI implementation, not only with MPICH. However, in our discussion we will only consider the MPICH/MPE combination. To trace-enable an application, we just need to compile it using the compiler wrappers offered by MPICH (mpicc, mpicxx, mpif77, mpif90) with the "-mpilog" flag activated—no change in the source code is required.

Trace-enabled applications run as normal but, when finished, write a trace file that consists of a set of time-stamped records (events) that describe the dynamic behavior of the application during its execution. Records in a trace file include:

– Information about MPI functions invoked by the application processes. Each function invocation generates two records: one when a process calls the function and another one when the function returns. A pair of these records represents a "state": the first one indicates when the process enters a given state, the second when the process exits from it. States are defined on a process-by-process basis. There are no "global" states. The most relevant fields of the state records are:

- Process identifier
- Timestamp
- Record type (state start/state end, MPI function)

– Information about message interchanges, only for point-to-point operations. Two records are generated per message, one when generated and another one when received. The basic information of the message records includes:
  - Process identifier
  - Timestamp
  - Record type, or operation (send/receive)
  - Identifier of the "other" party (destination for a send, source for a receive)
  - A message tag
  - Message size

From now on, we will use these abbreviations: SS means State-Start, SE means State-End, MS means Message-Send, and MR means Message-Receive.

A trace file can be analyzed using tools such as Jumpshot [29], distributed with MPE. Figure 1 shows a screenshot of this tool analyzing a CG.W.8 benchmark (Conjugate Gradient with 8 tasks, class W, included in the well-known NAS Parallel Benchmarks, NPB [2]). The legend (left) indicates the color codes used in the bars that represent states. Messages are represented by arrows.

This way of generating trace files has some limitations. For our purposes, two are the most relevant:

1. *Collective operations*. This class of operations, that involve synchronization and communication among multiple processes, are only represented as states. The trace file does not include any record that reflects the way messages are interchanged to

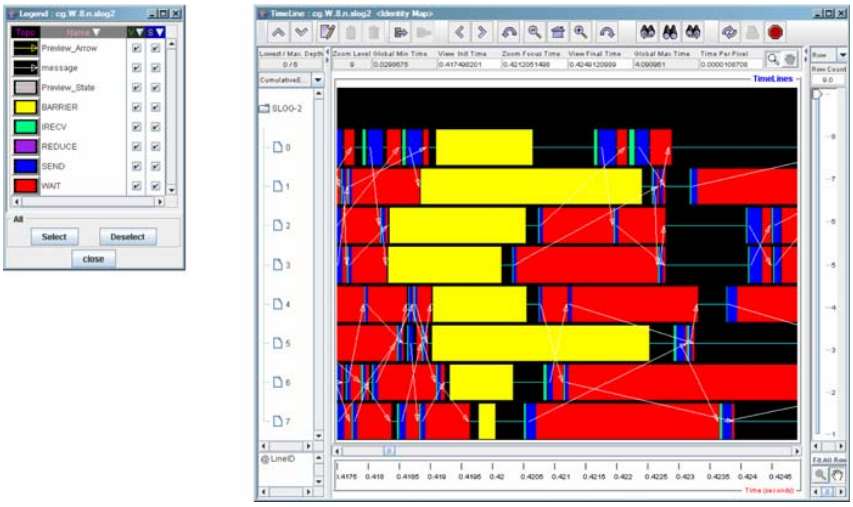


Fig. 1 Screenshot of Jumpshot visualizing a trace file generated by CG.W.8

implement collective operations. For example, in Fig. 1 it is possible to see that an MPI_Send state at a given node is related to a message that departs from that node. In contrast, the MPI_Barrier states are not related to any message.

2. *States vs. messages*. In order to fully understand a point-to-point state, you need to consider the information provided in the corresponding state records as well as information contained in separate message records. For example, an *<n ts1* SS MPI_Send> state start record indicates that the node identified as *#n* tries, at time *ts1*, to send a message, but details about the message are found in a separate *<n ts2* MS> message send record.

Collective operations can be implemented in many different ways. Often, the implementation is done at the MPI library level, using point-to-point operations to perform broadcast, reduce, gather/scatter, etc. This approach is very flexible, because collectives will work on top of any network, and is the one of choice in the case of popular MPI implementations, including MPICH. However, some networks provide support for collectives, or have topological properties that make some implementations more efficient than others, and the generic libraries cannot take advantage of these characteristics. A tailor-made implementation of collectives would be much more efficient. For example, [1] discusses the implementation of collective operations in an MPI library specifically designed for the IBM BlueGene/L system.

In the following subsection we will discuss how to overcome the first of these limitations and how, for simulation purposes, the state records (those that register when processes enter in and exit from MPI operations) can be safely ignored.

3.2 Generating Extended Trace Files

We have explained how trace files do not include detailed information about collective operations, because the details of how they are implemented are invisible to the application. A study of the internals of the MPICH implementation of MPI showed that collectives are, by default, carried out using point-to-point messages. The MPICH designers could have chosen to use some internal message-passing functions; fortunately for us, they decided instead to use the standard MPI point-to-point passing functions. For example, MPI_Broadcast is implemented using MPI_Send and MPI_Recv (the most basic message interchange functions), and MPI_Barrier is implemented using MPI_Sendrecv (a combination of MPI_Send and MPI_Recv in a single operation). The details of the default implementation of collectives are not accessible via the PMPI profiling interface, but this limitation is intentional—and makes sense, because other implementations are possible. We have modified the sources of MPICH to change this behavior, making the hidden operations visible through the profiling interface. This has no consequence in terms of communication semantics/timings.

With the modified MPICH, the generated, extended trace files follow the scheme described above, but they are longer because they include more detail. When visualized, using Jumpshot, a regular trace and an extended trace present different pictures. In Fig. 2 we can see a screenshot of a visualization of the extended version of a trace file, which corresponds to the same CG.W.8 benchmark used in Fig. 1. Note how the boxes inside the MPI_Barrier state represent the way this collective is implemented

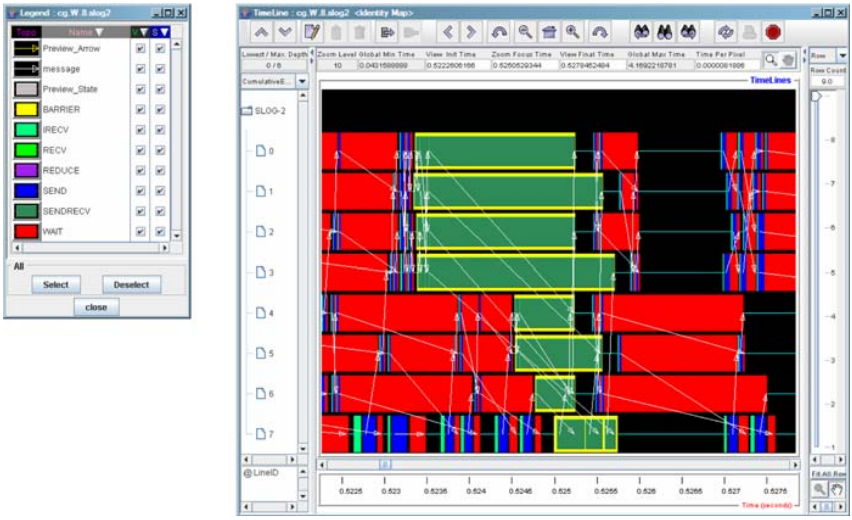


Fig. 2 Screenshot of Jumpshot visualizing an extended trace file generated by CG.W.8

using MPI_Sendrecv, and how messages interchanged in those operations are clearly visible.

From this point onwards, when discussing trace files we actually mean extended trace files.

4 Using Traces to Feed Simulations

We now describe how (extended) trace files can be used to provide realistic communication workloads to simulators of interconnection networks. We make the following assumptions:

- If we simulate a network with N nodes, the trace includes information about exactly N communicating processes; that is, there is a one-to-one relationship between application processes and network-attached nodes. For simplicity, we place processes into the nodes in consecutive order, that is, process n goes to node n.
- Simulators deal with the exchange of packets. Applications generate/consume messages of variable sizes, which need to be split into fixed or variable-sized packets (depending on the network technology). We discuss message interchanges as if the network delivered them directly, although implicitly we are considering message segmentation into fixed-size packets at origin, as well as message reassembly at destination.

If the first assumption is not met, simulation would be possible too, but the injection mechanisms and the placement policies would be more complex. There could be nodes (CPUs) without assigned processes (that would not participate in the simulation), or

nodes housing several concurrent processes. In this case a table mapping processes to nodes would be enough to deliver messages to the appropriate destination.

4.1 First Approach: Inject as Fast as You Can

An initial, and rather unrefined, approach to feeding a simulator with events taken from a trace file is as follows:

1. Ignore all the state (SS, SE) and MR records—in other words, use only the MS records with information about messages sent.
2. Split the trace file in one list per simulated node, and arrange the lists in timestamp order.
3. Make each node inject messages from its list into the network as fast as the network can accept them. Network backpressure is used to modulate the injection of load into the network.

As we can see, timestamps are used only to impose an order. The main justification for this decision is that we focus on network performance: we want to measure how fast a network can deal with a given workload, so we want to stress it, making it our bottleneck. The timing information included in a trace file is affected by issues that fall outside our control: the actual network used in the instrumented experiment, the processors and their speeds, the MPI implementation, the overhead of the instrumentation system, the number of processes that share a CPU, etc. We want to isolate the simulation from these facts. If we want to carry out performance predictions at a system level, we should take into considerations all these issues—which would make an already complex problem close to unworkable.

This approach to simulation accurately reproduces, using the information captured in the trace, the *spatial* communication pattern of the application (sources and destinations), and also the *message sizes*. However, it fails to reproduce the *temporal* pattern, which should respect message causality and reflect the actual way message exchanges are interleaved, as required by the application.

4.2 Second Approach: Follow Causal Order

The previous approach does not take into account the causal relationships between messages. In an actual execution of a parallel application, it often happens that a process stalls while waiting to receive a new message. Process execution is only resumed when the expected message arrives. We may emulate this behavior as follows:

1. Ignore all state records—in other words, use only MS and MR records.
2. Split the trace file into timestamp-ordered lists, one list per simulated node. Note that each node #*n* has an ordered record list (an "event queue") of <*n timestamp* MS *destination size*> and <*n timestamp* MR *origin size*> records. In the following steps timestamps are used only to order records.
3. Create, at each node, a receipt list, initially empty, that will store messages delivered by the network.

4. At each node, do the following:
   a. If the first record in the event queue is an MS, remove it and inject the corresponding message into the network.
   b. If it is an MR record, check if a corresponding message (matching origin, destination, tag, and size) is in the receipt list. If it is there, remove both entries. Otherwise, do nothing.
   c. When the simulator delivers a message, put it in the receipt list.

This procedure is depicted in Fig. 3. Its main implication is that an MR record puts the injection process on hold until the corresponding message is actually received from the network. In the figure, node #0 cannot advance, because it is waiting for a message from node #1, even if a message from node #2 has been received already. In contrast, node #1 can advance because the required message from node #0 has been delivered. This mechanism reproduces the actual way messages were interleaved when running the application, complying with the causal order between a receipt and the subsequent sends it may trigger.

The main drawback of this approach is that it *may* be excessively conservative. If we look again at Fig. 3, we see that the event queue of node #0 says that, after receiving a message from node #2, it is possible to send a message to node #3, and it happens that the message has been received already. However, node #0 is stalled (waiting for a message from node #1). We may wonder if application semantics is adequately emulated. Is it *really* necessary to receive the message from node #1 before advancing?

There is not a single answer to this question: it has to be discussed on a case-by-case (application-by-application) basis. For example, in [21] the causal ordering enforced by our simulator is considered valid in the context of cc-NUMA machines, because

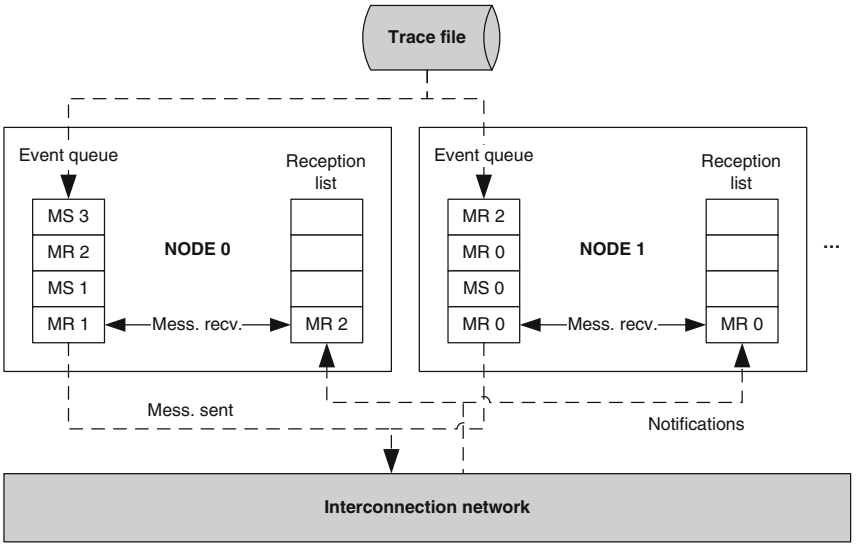


Fig. 3 Interface between the trace file and the network simulator. In the boxes, "MS *n*" means that a message is sent to node #*n*, and "MR *n*" means that a message is expected from node #*n*

message interchange is reactive: a message sent requires a response before allowing the process to advance. However, in general terms "*The parallel execution semantics, as reflected in the message communication operations and how the message data is used, determines process dependencies and message event ordering relationships, but only partially. Non-deterministic execution allows for alternative message event orderings*." [28]. We further discuss this issue in Sect. 5.

4.3 Traces for Performance Prediction

In the previous subsections we have stated that timestamp information is used to arrange events in temporal (or causal) order, but is otherwise ignored. This decision makes sense if we want to study the performance of the interconnection network. However, the network is only part of a system. Applications run on a collection of compute nodes, whose behavior is also visible through traces.

How can we use trace-driven simulation to estimate the time needed to run an application? The starting point is a *real* system, in which the application is run and traces are obtained. Those traces contain communication events, as can be seen in Figs. 1 and 2. In the figures we can also see "empty" spaces between MPI states. These spaces represent the time spent by processes outside MPI calls. If, during that time, processes are doing useful CPU work (of course, this is not always the case), then a simulator can be fed with the traces and a set of parameters that define the CPU characteristics, as well as the network characteristics, and used to predict overall application performance.

The simulator runs as described in Sect. 4.2 (following causal relationships). Times between communication states (between the SE record that corresponds to the end point of an MPI operation and the SS record that corresponds to the starting point of the following one) are converted into CPU states, with their SS and SE events. In this procedure we could scale the duration of these states, to simulate faster or slower CPUs. During the simulation run, when an "SS CPU" record for a given node is processed, injections from that node are stopped, and will be resumed only after processing the corresponding "SE CPU" event. In other words, the node is kept "busy" for the time required by the CPU state.

With this set-up, the simulation takes into consideration the characteristics of the CPUs, as well as the characteristics of the interconnection network, to estimate the time required to execute a message-passing application. Any change in the CPUs, or in the network, will be reflected as a change in the time required to consume the trace file.

5 Reproducing Application Semantics Accurately

5.1 Record Order in MPE Traces

A naïve user that analyzes an MPE trace file may think that events are recorded exactly when they happen during the application run, and that timestamps are accurate. This is not true, for several reasons. An obvious one is that the program file has been

instrumented in order to generate the trace, so the program is not running as fast as it would when not instrumented. The second reason is more subtle, and to understand it we need to explain how MPI traces are generated in the MPE environment. Our discussion is also valid for other tracing tools based on PMPI. Note that we assume that we deal only with point-to-point operations, because collective operations, if present, are also included in terms of the underlying point-to-point primitives that implement them. We focus on a subset of the MPI point-to-point operations, in order to discuss the relevant characteristics of the way they are logged without introducing unnecessary details. We consider that this subset is still sufficient, because it includes most (if not all) operations necessary to run the applications included in the NPB.

As we explained before, MPE logs are generated using instrumented versions of all the MPI functions. An <*n ts* SS MPI_*X*> record is generated when process #*n* invokes the MPI_*X* function; an <*n ts* SE MPI_*X*> record is generated when this function returns. MS and MR records (messages sent/received) are also generated by these instrumented routines, and only inside them. Note the implication of this way of working: *message send and receipt are not logged when they happen*.

– An MS record is logged *after* the process has entered into a state in which it requests sending a message (MPI_Send, MPI_Isend, MPI_Sendrecv), and *before* the process exits from that state. The message may be injected into the network much later due to a variety of reasons: semantics of MPI operations (immediate operations), previous messages queued, network congestion, decision of the kernel's scheduler, etc.
– An MR record is logged *after* the process has entered into a state in which it is actively waiting for a message or collection of messages (MPI_Recv, MPI_Wait, MPI_Sendrecv, MPI_Waitall), and *just before* the process exits from that state. It may happen that a message has been received from the network interface long before, but this receipt is not logged until the receiving process has entered a waiting state.

This behavior is clearly visible in Figs. 1 and 2. Note that MPI_Irecv states can be safely ignored, because the actual receipt of a message is recorded in a subsequent MPI_Wait or MPI_Waitall; there are no arrows arriving to the MPI_Irecv states. At any rate, when the trace file includes an MR record, it is there because the process really needs it to advance. So, in the simulation, it is necessary to receive that message before allowing the process to proceed—we will further discuss this issue in the following subsection.

The main conclusion here is that application-generated logs are not accurate because they do not reflect the exact moments in which messages are actually sent or received. The actual injection of a message may have happened later than indicated in the trace file, and the actual receipt of a message may have taken place before the time indicated by the record timestamp—sometimes, long before.

5.2 Receipts from MPI_ANY_SOURCE

A skilled MPI programmer knows that it is possible to indicate a wildcard, instead of a source process, in point-to-point receive operations: MPI_Recv(…, 3, …) executed

at process #0 forces this process to pause until a message from process #3 is received. In contrast, MPI_Recv(…, MPI_ANY_SOURCE, …) pauses the process until a message *from any source* is received. In terms of records in a trace file, the first call and the second one are indistinguishable. Both generate three records for process #0:

```
<0 t0 SS MPI_Recv>
<0 t1 MR …>
<0 t2 SE MPI_Recv>
```

The only difference *could* be in the second record: the first call guarantees that a message was received from process #3, while the other one may contain any (valid) process identifier.

The programmer may use MPI_ANY_SOURCE just for convenience: it may happen that the sender is known beforehand, so that it is not necessary to make it explicit. However, its main purpose is to allow processes to wait for messages that could arrive from any source, when the origin of the next useful message cannot be known a-priori. Let us explore this issue by means of a simplistic scenario of a master-slave application implemented using three application processes. Process #0 is the master, and processes #1 and #2 act as slaves. The protocol is as follows. A slave, when free to perform some work, sends a job request to the master. Then, the master replies with a task to perform.

A beginner in MPI programming could code the application as shown in Fig. 4, Version A. However, a more experienced programmer would use Version B of the code (Fig. 4, right).

Code Version A forces an unnecessary receipt order, which may delay program progress. For example it may happen that a message from slave #2 is already buffered, but one from slave #1 has not been received yet. The process is stalled in the first sentence of the loop, even when the third (receipt of a job request from slave #2) and fourth (sending a task to slave #2) could be executed without risk. This would not happen with Version B of the program, where the utilization of MPI_ANY_SOURCE at the receipt side would allow the process to make progress as soon as possible.

A trace file generated by Code Version A would always have message records in the same order: the one shown in the **Trace a** column of Fig. 5. It is easy to understand that a record reflecting that slave #1 sends a message to master #0 (<1*ts1* MS 0>) must be recorded in the trace file somewhere before record timestamped *ta2*; this record would match the one timestamped *ta1*. In the same way, a record <2 *ts2* MS 0> must be somewhere before record *ta4*, to match with record *ta3*. Note, again, that the actual

CODE VERSION A:

```
…
do {
  MPI_Recv(…, 1, …);
  MPI_Send(…, 1, …);
  MPI_Recv(…, 2, …);
  MPI_Send(…, 2, …);
} while pending_tasks;
…
```

CODE VERSION B:

```
…
do {
  MPI_Recv(…, MPI_ANY_SOURCE, …, &sender);
  MPI_Send(…, sender, …);
} while pending_tasks;
…
```

Fig. 4 Excerpts of sample codes for a master-slave application

Trace a:	Trace b:	Trace c:	Trace d:	Trace a′:
…	…	…	…	…
<0 ta1 MR 1>	<0 tb1 MR 2>	<0 tc1 MR 1>	<0 td1 MR 2>	<0 ta1 MR ANY>
…	…	…	…	…
<0 ta2 MS 1>	<0 tb2 MS 2>	<0 tc1 MS 1>	<0 td2 MS 2>	<0 ta2 MS 1>
…	…	…	…	…
<0 ta3 MR 2>	<0 tb3 MR 1>	<0 tc2 MR 1>	<0 td3 MR 2>	<0 ta3 MR ANY>
…	…	…	…	…
<0 ta4 MS 2>	<0 tb4 MS 1>	<0 tc3 MS 1>	<0 td4 MS 2>	<0 ta4 MS 2>
…	…	…	…	…

Fig. 5 Excerpts of trace files for Version A (Trace a) and Version B (Traces a, b, c and d) of the master-slave application. Trace a′ is a modification of Trace a using wildcard receives

receipt from node #2 could have happened before the receipt from node #1, but the trace file would not reflect this circumstance.

Now, let us suppose we used Code Version B. A sequence of events equal to that generated by Version A (probably with different timestamps) would be valid, but **Trace b**, **Trace c** and **Trace d**, also shown in Fig. 5, are equally valid. Only one of them would be actually recorded, depending on aspects such as workload assigned to those processors, relative CPU speeds, the characteristics and status of the network, etc. Let us further suppose that the trace actually recorded looks like **Trace a**.

When doing a simulation we do not want to force that particular order, because it may introduce unnecessary delays. We could use wildcard receives, because this information is in the trace file (not in the MR records, but in the corresponding state records). After a small manipulation of the MR records, **Trace a** can be modified to look like **Trace a**′—with which we feed the simulator. Immediately, we must suspend the master process (#0) at event timestamped *ta1* while waiting for a matching receipt. These are two possible scenarios:

1. The simulator delivers a message (job request) from slave #1 to the master. This unblocks the master, and the simulation continues. The master sends a message (containing a task to perform) to slave #1 and blocks again, waiting for a message from slave #2 that, eventually, will be delivered.
2. The simulator delivers a message from slave #2 to the master. This unblocks the master and the simulation continues. Then, as directed by the trace, master sends a message to #1—something that is not consistent with the event order in the trace file. We interpret this as a violation of application's semantics.

In order to obey causality relationships among messages we should not use wildcards in the trace files. The sequence of events actually stored in the trace may not be the only valid one, but at least we know that it is semantically valid.

6 Experimental Work Using Traces

First of all, we want to lay emphasis on this point: the purpose of this section is *only* to illustrate the kind of research work that can be done with a trace-driven simulation toolset, such as INSEE. We have used this environment to perform many performance studies, published elsewhere. To cite some examples, in [5] INSEE was used to

evaluate the performance impact of using twisted wrap-around links in mixed-radix twisted tori; in [16] we cross-validated the trace-driven simulation with an execution-driven environment based on Simics; and in [25] we studied the performance of a congestion control mechanism, comparing results obtained with synthetic workloads with those obtained with traces.

We will use the trace-processing abilities of INSEE to carry out two, very different, example performance studies. First we evaluate the impact on performance of different strategies of routing and virtual channel management. Then we estimate the time to execute an application (Conjugate Gradient) on three different target multicomputers. In the experiments we use a network with an 8-ary 2-cube topology—in other words, a 2D torus with 64 nodes. We have several trace files that can be used on networks of this size, which were obtained running the applications included in the NPB suite [2], class W, on 64 nodes.

6.1 Experimenting with Virtual Channel Management and Routing

Our first example study consists of an evaluation of the effects of using several virtual channels per physical link, and the impact of using adaptive routing. Figure 6 represents the network we model, and the details of each router. Each router has 4 bidirectional links (X+, X−, Y+ and Y−), each one connecting it to a different, neighboring router. In simple routers each link has associated input and output ports, with some buffer space for in-transit traffic. However, it is a common practice to associate several virtual channels (VC) to each link; each VC manages its own buffers. In the figure, there are 3 VC per link—we can see that in detail for link X+, shared by virtual channels X0+, X1+, X2+.

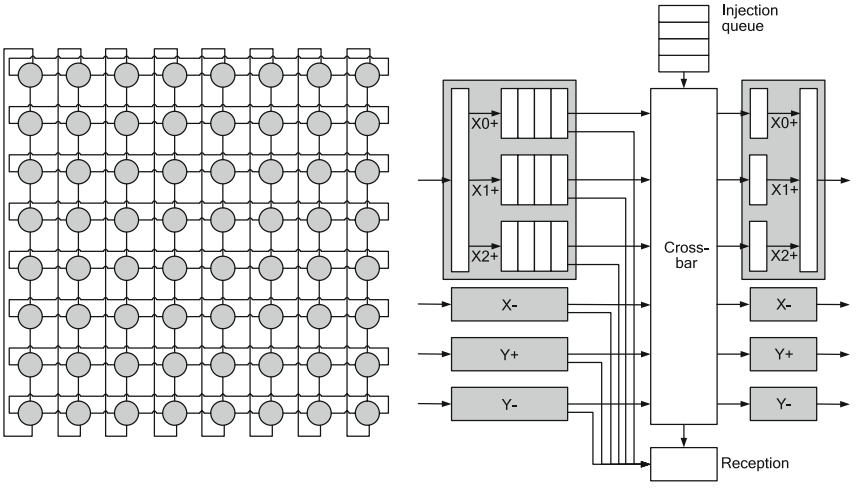


Fig. 6 Left: an 8-ary 2-cube (2D torus with 64 nodes). Right: model of the router simulated by FSIN

Routers, perform routing decisions in order to make packets advance from source to destination. There are multiple variants of routing algorithms, but we will only consider these:

1. Dimension-order, oblivious routing (DOR). A packet must traverse first as many hops as necessary in the X axis (in the row where it was injected) to reach its destination column. Then, it has to move in the Y axis (up or down) until reaching the destination.
2. Adaptive routing, using minimal paths. A packet can jump freely from a given VC to any other VC, continuing in the same axis (row or column) or switching. However, the jump must move the packet closer to the destination.

These two algorithms can lead to undesirable deadlock situations. To avoid these, we use the bubble routing mechanism described in [20] and used in the IBM BlueGene/L, so that we can state that the network is deadlock-free.

Regarding the utilization of several VC, and combining that with the routing algorithms, we compare routers built with the following designs:

1. *Oblivious 1 VC*. A single VC per physical channel. To ensure deadlock-freedom, bubble-restricted DOR is used.
2. *Oblivious 3 VC*. Three parallel VCs per physical link. Routing is bubble-restricted DOR. This arrangement reduces the effects of head-of-line blocking in the transit queues, so that when several packets are competing to use the same links they can advance faster.
3. *Adaptive 3 VC*. Three VCs per physical link. One of them, the Escape VC, uses bubble-restricted DOR, and the other two are adaptive. Packets can switch VCs, but access to the Escape VC has to follow the bubble restrictions. This arrangement provides the same advantage of the previous one. Furthermore, adaptive (but deadlock-free) routing allows a more efficient utilization of links, especially when packets have to travel long distances.

We configure FSIN to simulate networks built with these three routers. TrGen generates the workload, using the traces from class W of the NPB applications. The most interesting results are those obtained with traces from benchmarks Block-Tridiagonal (BT), Conjugate Gradient (CG) and Integer Sort (IS). Traces contain records of message interchanges, which need to be "packetized" in small blocks (packets) of 64 bytes. The queues in the routers are configured to hold up to 4 of these packets. Bandwidth of the links is 32 bits per cycle.

The simulator reports (among many other things) the number of cycles that the network needs to deliver all the applied workload. As each application is different, the numbers differ widely from one to another. For this reason we present relative values in Fig. 7; the base case (value 1) corresponds to the simpler router architecture (Oblivious 1 VC). The figure represents the average values of 10 simulation runs, as well as the 99% confidence intervals.

From the obtained results, we see that, for the traffic pattern used by BT, the utilization of several VCs per link does not offer any advantage in terms of performance. However, improvements for IS and CG are quite good. For IS, most of the improvement comes from the use of several VCs; adaptivity provides minor additional gains.

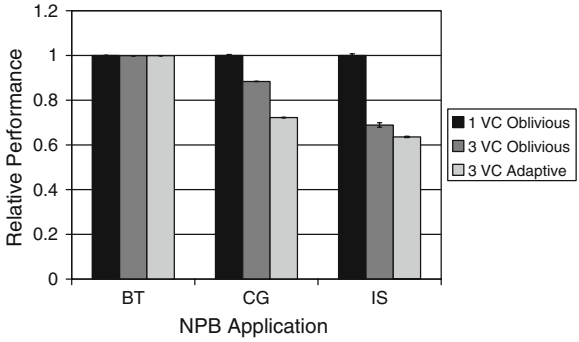


Fig. 7 Effects of using 3 VC per physical link, and of adding adaptivity. Results relative to the base case (Oblivious 1 VC). Average of 10 simulation runs, and 99% confidence intervals

CG benefits less than IS from using several VCs, but is capable of taking advantage of adaptivity. The reasons for these results have to be found in the different characteristics of the traffic patterns generated by the applications. The detailed explanation goes beyond the scope of this paper; however, we can give some clues. Local communications neither benefit from using many VCs, nor from adaptivity; this explains the behavior of BT. Patterns with intense, non-local interchanges can take advantage of many VCs, because its use reduces head of line blocking; this explains the behavior of IS. Adaptivity is useful when the utilization of network resources is not homogeneous, because it helps balancing the workload; this explains why CG improves significantly with adaptivity, and why IS does not. Remember that the improvements reported are for communication clearing time; they do not reflect actual whole-application level improvement.

6.2 Estimating Execution Times

In Sect. 4.3 we described a methodology to estimate the time an application would spend when executed in a "target" architecture different from the one used to capture the traces. As an example, we will estimate the time to execute the CG benchmark (class W.64) in three different scenarios. The original trace file was generated in the MareNostrum Supercomputer, [3] whose interconnection network is a fat-tree implemented with Myrinet-2000 adapters and switches. This network operates at link speed of 2 Gb/s. The compute nodes are PowerPC 970 at 2.3 GHz. Each MPI task runs in a different processor.

The target architectures are three 2D, 8×8 tori, whose network links work, respectively, at 100 Mb/s, 1 Gb/s and 10 Gb/s. We use the "Adaptive 3VC" configuration of virtual channels, as described in the previous experiment. We have not applied any scale to the CPU times, so these target architectures are supposed to use 2.3 GHz PowerPC CPUs. Results of the simulations, reporting estimated execution times for the three link speeds, are summarized in Table 1, along with the actual execution time in the MareNostrum.

Table 1 Actual execution time of CG.W.64 in the MareNostrum, and estimated times for three different target architectures. Times in seconds

MareNostrum	8 × 8 torus 100 Mb/s	8 × 8 torus 1 Gb/s	8 × 8 torus 10 Gb/s
0.54	3.49	0.55	0.25

We can see how CG, a very communication-intensive application, can take advantage of network improvements. The speedup when changing from a 100 Mb/s network to one running at 1 Gb/s is 6.35; this is because at low speeds most of the execution time is due to communication. The improvement when using the 10 Gb/s network, instead of the 1 Gb/s one, is not that great (2.20), but still notable; this is because at high speeds the execution time is more computation-bound.

Note that results obtained in the MareNostrum are just *indicators* of the peak performance reachable with this machine. In this computer we are not using a 64-node network, but a full Myrinet-2000 fat-tree capable of linking the more than 10 K nodes of this computer. The 64 CPUs used in the experiment were not necessarily consecutive; in fact, we don't know the exact placement used by the scheduler, which means that the CPUs could be located in different leaves of the fat-tree. Measurements were taken when the machine was in production, and other applications were running (and using the network) at the same time, so some degree of interference was present—at least in the network, because the compute nodes were used exclusively by our applications. Therefore, it should not look strange that the predictions for our 1 Gb/s network are so close to the measurements with the 2 Gb/s Myrinet-2000 network.

7 Related Work

In the literature we can find many papers discussing different aspects of trace capturing mechanisms and utilization of traces for performance evaluation—see for example [10]. For the specific topic of interconnection networks, Chapters 23 to 25 of [7] are of particular relevance. In addition to performance evaluation and prediction, other common use of traces of parallel applications is visualization, often as a help for debugging and detection of bottlenecks. The list of references would be very long, because this is a well-established area of work, so we focus on the main topics addressed in this paper.

7.1 Alternative Ways of Obtaining Trace Files

MPE is not the only way of acquiring traces of MPI applications. There are many other options available (see [13] for a review), but most of them are based on the instrumentation of application source code, or on the substitution of standard MPI functions by instrumented versions at compile time using the PMPI interface. As they work at the application level, they cannot be totally accurate regarding message send/receive times. An alternative way of getting traces would be to capture information at a lower level. The operating system, or a set of middleware daemons providing services to

running MPI applications, should be capable of recording the actual timestamps of communication events thus providing better timing information. An example of this approach is Sun's Dtrace [18].

Yet another way of generating a trace file in environments such as networks of workstations could be using a network *sniffer* that captures packet interchanges between computers. A programmable sniffer such as Wireshark [27] could be used to this purpose. A trace file generated this way would contain records with this information: <*timestamp*, *origin*, *destination*, *data*>, with accurate timestamps. However, these tools capture network-level frames, so we do not know whether a burst of packets belongs to the same long message, or are a sequence of smaller ones. Also, we know when a packet has been delivered by the network to the receiving node, but not when it is actually available to the corresponding application process. We have a temporal order of records, but this is not a causal order, because inter-dependencies are not captured. An additional shortcoming of this approach appears when several processes share a single computer (a common scenario when we need traces for large systems): since message interchange between the processes that share a machine is done internally, the network is not used, so we do not have the associated trace records.

7.2 Performance Prediction Using Trace Files

The Dimemas tool, developed at the Technical University of Catalonia [8], can be used to carry out performance prediction studies using trace files (as well as machine descriptions) as its input. Note that the way we model the CPUs within INSEE is utterly simplistic, but the network is simulated with great detail. The Dimemas approach is the opposite: it accepts detailed descriptions of the compute nodes, so that a change in the system architecture is not simulated by simply scaling CPU states; however, the network model is very simple: a collection of parallel buses. Dimemas uses its own trace format and trace-capturing tools, that gather more information than that included in MPE's CLOG traces (for example, thread-level information, and hardware counters)—but that require kernel-level support, not always available. Trace records include not only MPI operations and communications, but also the states of each task. This means that Dimemas traces log when tasks are using the CPU, when they are blocked by other tasks (when sharing CPU) and when they are stalled for I/O operations.

Note that Dimemas was conceived as an application analysis tool that allows to inspect an application in order to locate undesirable behaviors (as bottlenecks or unbalancing between threads), so the high level of detail in CPU and application/task/thread modeling. In contrast, INSEE is used to evaluate interconnection networks, and obviously, the IN is modeled with high level of detail.

7.3 Extended Traces and Collective Support

In Sect. 3, we discussed the generation of extended trace files, as required by our FSIN simulator. It is important to remark that *we do not modify the default implementation of collectives included in MPICH*. These primitives are good for general use, but not optimized for any particular underlying communication fabric. Therefore, when we use

the extended trace files for evaluation purposes, we are testing a target machine with this particular implementation of collectives. The availability of this implementation is of great interest for us, because with it we can focus on the design and evaluation of the point-to-point abilities of the network. However, we know that a good portion of the design effort for a parallel computer should go to the supporting library, including an MPI library with customized collectives [1,4]. A fair assessment of a computer with support for collectives should be done using regular trace files. A well-known machine *with* collective support is the BlueGene [1]. In contrast, clusters built around Myrinet [14] networks *do not* include collective support—BSC's MareNostrum is a remarkable example [3]. In Myricom's implementations of MPICH (MPICH-GM on top of the older GM library, and MPICH-MX on top of the MX library [15]) implementation of collectives is not changed, using the default one provided in the original MPICH. They plan to include support for collectives in future releases of MX.

The Dimemas simulator is able to work with regular trace files, and deals with collectives using different models (simplifications). It assumes that the time to complete a collective is

$$time = (latency + (size/bandwidth)) * model\_factor$$

where the *model_factor* can be 0 (null time), 1 (constant), N (linear) or an expression to model logarithmic times. These simplifications do not take into consideration the possibility of network contention, and are not universally valid for all hardware and/or software implementation of collectives. This is another proof of the differences in objectives between Dimemas and INSEE.

7.4 Other Simulation Tools

We end this section with a review of simulation tools for interconnection networks. SICOSYS [19], developed at the University of Cantabria, performs simulations of switching components with a high level of detail, providing timing information similar to that achieved using hardware simulators. Its large footprint does not allow it to simulate very large networks, but it is extremely useful for on-chip and on-board networks. SICOSYS can be fed with synthetic workloads and application traces, and can also be integrated with other tools to perform full system simulation.

The Flexim 1.2 simulator [26], developed at the University of Southern California, shares many design principles with FSIN. A main difference is that Flexim is designed for routers using wormhole switching, while FSIN uses virtual cut-through switching. Flexim supports synthesized traffic patterns or trace-driven traffic, although no details of the mechanism involved can be found in the documentation.

The Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign maintains BigNetSim [6]. Its design is very different from INSEE, SICOSYS, or Flexim. It works alongside BigSim, a system emulator able to run applications specifically compiled for it. The emulator captures a collection of tasks (blocks of computation and communication) on a number of processors along with their dependencies and writes these tasks to trace files. BigNetSim reads the traces and

simulates the execution of the original tasks by elapsing time, satisfying dependencies, and spawning additional tasks by passing messages through a detailed network contention model. This generates corrected times for each event (advancing or delaying the simulation clock) which can be used to analyze its performance on the target machine. As happens with Dimemas, focus is more on application performance than on network analysis.

8 Conclusions

In this paper we have introduced a collection of tools and techniques to carry out evaluations, via simulation, of interconnection networks, using realistic workloads, provided by trace files obtained from the execution of actual applications. These techniques have been incorporated in the INSEE toolset.

Firstly, we needed to deal with the information contained in the trace files. As regular traces do not include the details of collective operations, we modified the trace capturing mechanism. Extended trace files allow us to deal only with simple, point-to-point message interchange records.

Then, we have pointed out the two main limitations of traces, namely the lack of accuracy in timing information, and the fact that a trace file includes only a possible valid outcome (record order) of a parallel program execution, but not the only valid one. We have explained that, for the evaluation of interconnection networks, we may ignore timing information, but not the causal relationships implicit in event order. We know that different orderings in event processing *may* be valid (respecting application semantics), but deciding about whether or not altering the order recorded in the trace file requires application-dependent knowledge, and our choice may lead to non-valid sequences of events. So the safest approach, for simulation purposes, is to follow, without exception, the exact event ordering of the trace file.

The trace processing ability integrated into INSEE offers a very flexible tool to evaluate different aspects of interconnection networks for parallel systems. As a way of showing the kind of work that can be carried out with this tool, we provide a simple but illustrative performance study: for some applications, the utilization of multiple virtual circuits per physical channel produces important performance gains, which can be even larger if using adaptive routing; the study shows the extent of the achievable gains. Additionally, we have shown how to estimate the execution time of an application running on a target architecture at different link speeds using traces captured in a real machine.

For future work, we plan to investigate how to integrate application semantics into simulation to allow different (but valid) event orderings.

Acknowledgements This work has been supported by the Spanish Ministry of Education and Science (TIN2007-68023-C02-02) and by the Basque Government (IT-242-07). Mr. Javier Navaridas is supported by a doctoral grant of the UPV/EHU. We gratefully acknowledge the utilization of resources and tools of the Barcelona Supercomputing Center/Centro Nacional de Supercomputación, Spain. We also want to thank the anonymous reviewers for their help improving this paper.

References

1. Almási, G., et al.: Optimization of MPI collective communication on BlueGene/L systems. In: Proceedings of the 19th Annual International Conference on Supercomputing, Cambridge, Massachusetts, June 20–22, ICS '05, pp. 253–262. ACM Press, New York, NY (2005)
2. Bailey, D.H., Harris, T., Van der Wigngaart, R., Saphir, W., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-010, NASA Ames Research Center (1995)
3. Barcelona Supercomputing Center Home Page: Available at http://www.bsc.es. Accessed 1 Dec 2008
4. Brightwell, R.: A comparison of three MPI implementations for red storm. In: 12th European PVM/MPI Conference, September 2005. Lecture Notes in Computer Science, vol. 3666, pp. 425–432 (2005). doi:10.1007/11557265_54
5. Cámara, J., Moretó, M., Vallejo, E., Beivide, R., Martinez, C., Miguel, J., Navaridas, J.: Mixed-radix twisted torus interconnection networks. In: Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium—IPDPS '07, Long Beach, CA, March 26–30, 2007
6. Choudhury, N., Mehta, Y., Wilmarth, T.L., Bohm, E.J., Kalé, L.V.: Scaling an optimistic parallel simulation of large-scale interconnection networks. In: Proceedings of the 37th Conference on Winter Simulation, Orlando, Florida, December 04–07, 2005
7. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan-Kaufmann, Los Altos (2004)
8. Barcelona Supercomputing Center. DIMEMAS. http://www.bsc.es/plantillaA.php?cat_id=475. Accessed 1 Dec 2008
9. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI Message-Passing Interface standard. Parallel Comput. **22**(6), 789–828 (1996). doi:10.1016/0167-8191(96)00024-5
10. Karrels, E., Lusk, E.: Performance analysis of MPI programs. In: Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing (1994)
11. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hillberg, G., Hgberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: a full system simulation platform. Computer **35**(2), 50–58 (2002). doi:10.1109/2.982916
12. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. University of Tennessee. Available at http://www.mpi-forum.org/. Accessed 1 Dec 2008
13. Moore, S., Cronk, D., London, K., Dongarra, J.: Review of performance analysis tools for MPI parallel programs. In: Cotronis, Y., Dongarra, J. (eds.) 8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science, vol. 2131, pp. 241–248. Springer Verlag, Berlin (2001)
14. Myricom Home Page: Available at http://www.myri.com. Accessed 1 Dec 2008
15. Myricom Inc.: Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, Version 1.1 January 01, 2006. Available at http://www.myri.com/scs/MX/doc/mx.pdf
16. Navaridas, J., Ridruejo, F.J., Miguel-Alonso, J.: Evaluation of interconnection networks using full-system simulators: lessons learned. In: Proceedings of the 40th Annual Simulation Symposium, Norfolk, VA, March 26–28, 2007
17. Navaridas, J., Ridruejo, F.J., Miguel-Alonso, J.: On synthesizing workloads emulating MPI applications. In: The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08), April 14–18, 2008, Miami, Florida, USA
18. OpenSolaris Community: Dtrace. Available at http://opensolaris.org/os/community/dtrace/. Accessed 1 Dec 2008
19. Puente, V., Gregorio, J.A., Beivide, R.: SICOSYS: an integrated framework for studying interconnection network performance in multiprocessor systems. In: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP 2002) (2002)
20. Puente, V., Izu, C., Beivide, R., Gregorio, J.A., Vallejo, F., Prellezo, J.M.: The adaptative bubble router. J. Parallel Distr. Comput. **61**(9), 1180–1208 (2001)
21. Puente, V., Prellezo, J.M., Izu, C., Gregorio, J.A., Beivide, R.: A case study of trace-driven simulation for analyzing interconnection networks: cc-NUMAs with ILP processors. In: Proceedings of the IEEE 8th Euromicro Workshop on Parallel and Distributed Processing, Rhodes, Greece, January 2000
22. Ridruejo, F.J., Gonzalez, A., Miguel-Alonso, J.: TrGen: a traffic generation system for interconnection network simulators. In: 1st International Workshop on Performance Evaluation of Networks for

Parallel, Cluster and Grid Computing Systems (PEN-PCGCS'05). ICPP 2005 Workshops, 14–17 June 2005
23. Ridruejo, F.J., Miguel-Alonso, J.: INSEE: an interconnection network simulation and evaluation environment. In: Lecture Notes in Computer Science, vol. 3648 (Proceedings of the Euro-Par 2005), pp. 1014–1023 (2005). doi:10.1007/11549468_111
24. Ridruejo, F.J., Miguel-Alonso, J., Navaridas, J.: Concepts and components of full-system simulation of distributed memory parallel computers. In: Proceedings of the HPDC '07, June 25–29, 2007, Monterey, California, USA
25. Ridruejo, F.J., Navaridas, J., Miguel-Alonso, J., Izu, C.: Realistic evaluation of interconnection network performance at high loads. In: Proceedings of the 8th International Conference on Parallel and Distributed Computing Applications and Technologies – PDCAT 2007, Adelaide, Australia, 3–6 December 2007
26. SMART group at the U. of Southern California: Information on FlexSim1.2. Available at http://ceng.usc.edu/smart/FlexSim/flexsim.html. Accessed 1 Dec 2008
27. Wireshark Home Page: Available at http://www.wireshark.org/. Accessed 1 Dec 2008
28. Wolf, F., Malony, A., Shende, S., Morris, A.: Trace-based parallel performance overhead compensation. In: Proceedings of the International Conference on High Performance Computing and Communications (HPCC), Sorrento, Italy, Sept. 2005
29. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with Jumpshot. High Perform. Comput. Appl. **13**(2), 277–288 (1999). doi:10.1177/109434209901300310

Chapter 10. Full-system simulation

Full reference:
Fco. Javier Ridruejo, Jose Miguel-Alonso, Javier Navaridas.
Full-system simulation of distributed memory multicomputers.
Cluster Computing, Volume 12, Issue 3, September 2009, 309-322

Trace-based simulation is an improvement over synthetically generated traffic, but it is not the silver bullet: a trace file represents only one of the valid interchanges of messages that the application performs while running, and it is tied to the system where the parallel application was run, which has a specific topology, a number of nodes and, maybe, other running applications that may interfere with the one being logged. Also, traces of large multicomputers are difficult to obtain and process, or simply unavailable.

We decided to overcome some limitations of trace-based traffic generation by adding more realism to simulation, so we included a module into TrGen to run full-system simulation. This is the most accurate kind of traffic, it allows executing unmodified parallel applications on real operating systems on top of a simulated IN. To be able to fully simulate the computer nodes of a multicomputer, INSEE is connected with Simics, a full-system simulator.

At the moment this paper was written, full-system simulators had very little detail in either network or nodes for our needs, so we decided to implement our own simulation platform, assembling the well-reputed Simics software with INSEE. In this paper we explain the design decisions we took and the problems that arose, such as finding a trade-off between simulation fidelity and execution time, and dealing with unexpected interactions between IN-level and end-to-end (TCP) congestion control.

Full-system simulation has allowed us to test with high fidelity several proposals for congestion control at the IN level, to study how those proposals affect higher level protocols (such as TCP), and also to characterize the interactions between simulation components, contributing to improve the state of the art of IN simulation and congestion control.

In this publication the author has carried out all aspects of the full-system simulation, its design, implementation and integration between our TrGen modules and the third-party simulator Simics. The author is also responsible of the experimental work required in the two case studies (congestion control at the IN, and it interactions with end-to-end congestion control).

Cluster Comput (2009) 12: 309–322
DOI 10.1007/s10586-009-0086-y

Full-system simulation of distributed memory multicomputers

Fco. Javier Ridruejo · Jose Miguel-Alonso · Javier Navaridas

Received: 8 October 2008 / Accepted: 8 March 2009 / Published online: 28 March 2009

Abstract In this paper we discuss environments for the full-system simulation of multicomputers. These environments are composed of a large collection of modules that simulate the compute nodes and the network, plus additional linking elements that perform communication and synchronization. We present our own environment, in which we integrate Simics with INSEE. We reuse as many Simics modules as possible to reduce the effort of hardware modeling, and also to simulate standard machines running unmodified operating systems. This way we avoid the error-prone effort of developing drivers and libraries. The environment we propose in this paper enables us to show some of the difficulties we found when integrating diverse tools, and how we were able to overcome them. Furthermore we show some important details to have into account in order to do a valid full-system simulation of multicomputers, mostly related with synchronization and timing. Thus, a trade-off has to be found between simulation speed and accuracy of results.

Keywords Full-system simulation · Interconnection networks · Multicomputers · Clusters

F.J. Ridruejo · J. Miguel-Alonso (✉) · J. Navaridas
Dep. of Computer Architecture and Technology, The University of the Basque Country, P. Manuel de Lardizabal, 1, 20018 Donostia-San Sebastian, Spain
e-mail: j.miguel@ehu.es

F.J. Ridruejo
e-mail: franciscojavier.ridruejo@ehu.es

J. Navaridas
e-mail: javier.navaridas@ehu.es

1 Introduction

Supercomputing is a very valuable resource which is continuously growing in importance in business and science. Current scientific studies rely on analysis and modeling of different natural phenomena that require a huge amount of computing power, which is not attainable using regular off-the-shelf computers. For example, physicists, chemists or pharmaceutical researchers simulate, for different purposes, interactions between large numbers of molecules. Likewise, in the business context, corporations demand large amounts of computing power in order to use data mining software over their huge data bases, with the objective of extracting knowledge from raw data, and to use that knowledge to their advantage. Obtaining patterns of consumer habits, boosting sales, optimizing costs and profits, estimating stocks or detecting fraudulent behavior are just a few interesting application domains. At any rate, in both contexts, the required computing power is only limited by the available resources. In general, if available computing resources are doubled the number of runs, the grain-size (whichever this means in the particular application context), the size of the datasets or whatever other parameter that affects execution time will be increased in order to fully use the new resources. In other words, the magnitude of the experiments is scaled up to the available resources. This means that there is a permanent demand of high-performance computers able to cope with these challenging workloads.

A supercomputer is not only a piece of hardware. It is actually a multipart system that integrates a large collection of hardware and software elements. Therefore, the design of a supercomputer is a complex task that comprises the selection and design of multiple components, such as compute elements, storage, interconnection network, access elements,

operating system, high performance libraries, parallel applications, etc. Depending on budget and availability, elements can be designed from scratch; often, however, off-the-shelf components are reused, either directly or modified to fulfill tasks different to those they were designed for.

During the preliminary phases of the design of a supercomputer, elements are tested and evaluated separately, in order to assess (and, if possible, improve) their performance. These evaluations are carried out using synthetic loads, based on statistical distributions, which allow for fast simulations, but may not be truly representative of actual workloads. Simulation speed is important in this phase, in order to be able to explore a wide range of options to help in the decision-making process, choosing the more promising alternatives.

In subsequent design phases the simulated model grows in complexity, and more realistic evaluations are performed, mixing a complex model of the component under evaluation with simpler models of the rest of the system, usually working with traces obtained in actual machines. Traces are more realistic than synthetic workloads, but may comprise some characteristics of the system in which they were taken that are not valid in the system under evaluation [17].

Once system components have been chosen, a validation of the whole design is required to confirm that the behavior is the expected one, and to check that there are no undesirable interactions between components that may have passed unnoticed before, due to the simplification of models and simulations. This validation is usually done with full-system simulators made from scratch or, in most cases, using different simulators for each component of the system, and doing some linking work to put them to operate together.

Our interest is mainly focused on interconnection networks for distributed memory parallel systems, a kind of specific-purpose network that allows compute nodes to interchange messages with high throughput and low latency—which is required to run efficiently parallel applications. In the rest of this paper we will use "IN" as the acronym for interconnection network, and multicomputer as a shorter way of naming distributed memory parallel computers.

We will describe the components that take part in a full-system simulation of a multicomputer, making a proposal that mixes two very different tools: Interconnection Network Simulation and Evaluation Environment [29] (INSEE for short) in charge of the IN, and Simics [13], used to simulate the compute elements. We will also discuss several approaches to interface these two classes of simulators, and problems that may arise when doing full-system simulation, some due to reutilization of components of the simulated hosts, and some due to unexpected interactions between components. Moreover, we will explain the complexity of fine-tuning all components and their interfaces, in order to find a trade-off between simulation accuracy and resource usage (simulation time).

The rest of the paper is organized as follows. In Section 2 we review related work. Section 3 explains the elements that can take part in a full-system simulation of a multicomputer, as well as some approaches to glue together an IN simulator with the simulators of the compute nodes. Section 4 discusses how to synchronize these simulators. Section 5 introduces our proposal for full-system simulation of multicomputers, combining INSEE with Simics. In order to show the capabilities of this tool, a set of experiments related with congestion control are defined in Section 6. Results are shown and analyzed in Section 7. Finally, in Section 8 we enumerate the conclusions of this work.

2 Related work

There are other research groups around the world interested in full-system simulation. However most of them are only interested in either the performance evaluation of workloads on servers, or in the assessment of a particular microarchitectural improvement. We can enumerate several full-system simulators specifically designed to perform these kinds of studies: RSIM [24], ML-RSIM [32], SimOS [31], Simics [13], M5 [3], SIMFLEX [37], GEMS [14], and Mambo [6]. A longer list is available in [15].

Models used for the IN are often too simplistic. These tools implement networking systems based on Ethernet, which is valid for most of the usual performance evaluations of server systems that run OLTP (On-Line Transaction Processing) workloads. As far as we know, none of them implement a sophisticated IN such as those used in high-performance clusters or massively parallel processors.

It would be possible to integrate more complex IN models into available full-system simulation tools; however, IN simulators already exist (SICOSYS [27], the Chaos Router Simulator [34], FlexSim [33], OPNET [23], NS [35] and many others), and it would be easier for designers to integrate available tools, instead of starting from scratch with, for example, a collection of Simics modules for an IN. For instance, SICOSYS can interface with RSIM to do full-system simulation of shared-memory parallel computers, providing an accurate timing model, as discussed in [28]. However, this setup consumes a huge amount of resources (memory, CPU time) and only allows for the simulation of a few tens of interconnected compute nodes.

Simics memory timing mechanism is too simplistic: all instructions and memory accesses take the same amount of time. Consequently several tools have been designed that extend Simics functionality to perform detailed memory simulation. Two of these are GEMS [14] and SIMFLEX [37]. GEMS was designed to study a wide variety of memory hierarchies and systems ranging from broadcast-based SMPs to hierarchical directory-based Multiple-CMP systems. It

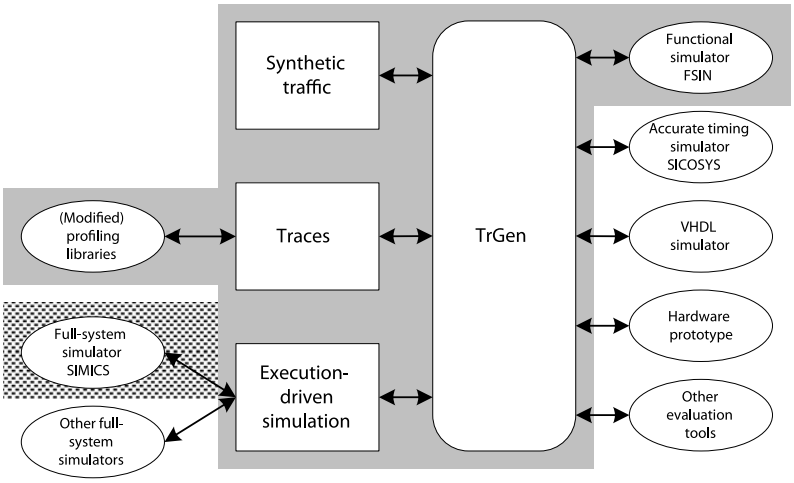


Fig. 1 Overall design of INSEE (from [29]). *Elements with grey background* form part of INSEE, while the remaining parts are external modules. In this paper we discuss the integration with Simics

can model current high-end, non-uniform memory access (NUMA) multiprocessor systems which have their memory controllers connected via an interconnection network, including directory-based and snooping-based systems. It has not been designed to provide support to large simulation of multicomputers with their compute nodes connected through another kind of IN.

SIMFLEX uses statistical sampling to accelerate simulations. It estimates the performance of an application by measuring several samples of it. Those samples are carefully chosen using established statistical sampling methods from a library of application checkpoints that represent the behavior of the whole application with a high confidence level, depending on the number of samples being executed. It also requires finding out the warming period previous to the measurement of each sample. SIMFLEX can model chip multiprocessor and distributed-shared-memory multiprocessor systems. However, it does not save the state or keep account of the queues in the interconnection network, it runs the example more times to warm up the system to allow the queues to fill into a steady state. That approach could alleviate a congestion state in the interconnection network, hiding or attenuating effects of congestion. Moreover, there may be different execution paths when trying with different topologies and other parameters. A big effort has to be done to integrate an interconnection network simulator and SIMFLEX, and to prepare every library of samples for each benchmark.

The M5 simulator is specifically designed to conduct research in TCP/IP networking and provides features necessary for simulating networked hosts, including full-system capability, a detailed I/O subsystem, bus timing, coherence effects of network DMA transfers [2] and the ability to simulate multiple networked systems deterministically in a single instance of the M5 simulator. M5 connects network interface cards of individual systems using an Ethernet link object, modeled as a lossless, full-duplex channel of configurable bandwidth and delay [3], a very simple approach to do IN simulation. New components can be developed and incorporated into the M5 structure.

As we can see, most of these tools are heavyweight, and focus either on LAN technologies, or on shared-memory machines. Our interest is on distributed-memory parallel machines, and we already have a simulation environment to evaluate proposals for the INs used in this kind of architectures: INSEE [29]. This environment, depicted in Fig. 1, includes FSIN, a time-driven, lightweight, flexible, and functional IN simulator that allows us to simulate distributed memory parallel computers with thousands of nodes arranged in different topologies (k-ary n-cubes, multistage networks). The TrGen module allows synthetic traffic generation, trace based generation and, as we will describe later, full-system simulation. The integration of INSEE with Simics, the tool of choice to provide full-system simulation of compute elements, provides a great environment to experiment with cluster and MPP technologies.

3 Interfacing IN and compute element simulators

There are many possible approaches to perform full-system simulation of a multicomputer. In Fig. 2 we can observe a collection of components that take part in the simulation. All of them are software-based components, but some simulate pieces of hardware.

A single instance of an IN simulator simulates the flow of packets through a network. Several instances of full-system simulators mimic in detail the operation of the compute elements. The level of detail includes the simulation of the hardware devices, as well as the ability to run unmodified software, including operating system, support libraries (for example, an MPI library) and parallel applications.

We will pay special attention to these elements:

- The NIC (Network Interface Card) that interfaces with the IN; this is a software module that simulates the NIC.

- The driver, in kernel space, for that NIC.
- The protocol stack, in kernel space, providing higher-level access to the IN.
- The support library, on top of the protocol stack, that provides the MPI API and the necessary run-time support for parallel computing.
- A process of a parallel application.

Note that there are many instances of these elements, at least one per simulated compute node. Additionally, the simulation environment includes a synchronization module that make all simulators advance in synchrony, and a traffic manager module that allows the interchange of information (frames, packets) between node and IN simulators, making format translations if required.

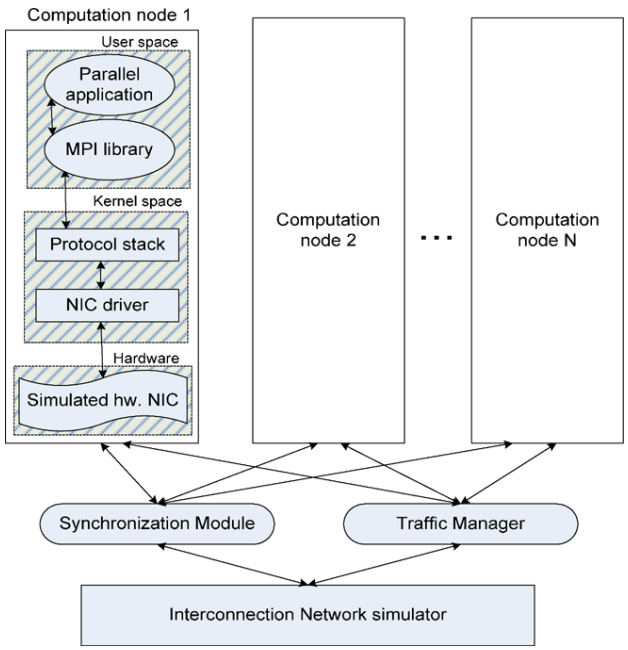


Fig. 2 Elements taking part in a full-system simulation of a multicomputer

Now we focus on the different mechanisms available to implement the traffic manager. Issues about synchronization will be discussed later.

In order to better understand the following sub-sections, we show in Fig. 3 several protocol stacks commonly used in cluster environments. The first three are for Ethernet hardware, commonly used in low-cost clusters: MPICH [18] over TCP, LA-MPI [10] over UDP and LAM [11] over UDP using *lamd* daemons. More expensive clusters use dedicated system area networks such as Myrinet and Infiniband. For Myrinet, Myricom provides two low-level interfaces (GM and MX), and a corresponding modification of MPICH to run over them [19]. In the case of Infiniband, the figure corresponds to MVAPICH, a modification of MPICH that runs on top of either VAPI or OpenIB low-level interfaces [12]. Note that there are many other possible stacks, for different combinations of MPI implementations (both commercial and free) and drivers for interconnection networks.

3.1 Substitution of the NIC driver

A possible approach to interface compute nodes with IN (obviously, in terms of simulation) would be via a substitution of the NIC driver, using another one that, in addition to doing its regular work, intercepts network traffic and passes it directly to the Traffic Manager. The main advantage of this option is that we can reuse the NIC model that comes with the full-system simulator, and also the Linux protocol stack. However, this approach has the disadvantage of requiring us to program a network driver for Linux that must register itself in the kernel as a general network driver. Furthermore, this driver must interact properly with the protocol stack because otherwise it would be impossible to reuse this stack. Note that the trickiest part here would be writing a driver that, while running in a simulated machine, interacts with an external module running in the simulation environment.

3.2 Substitution of the NIC simulated module

Another option would be to implement our own NIC module, which would mimic the operations of the original one

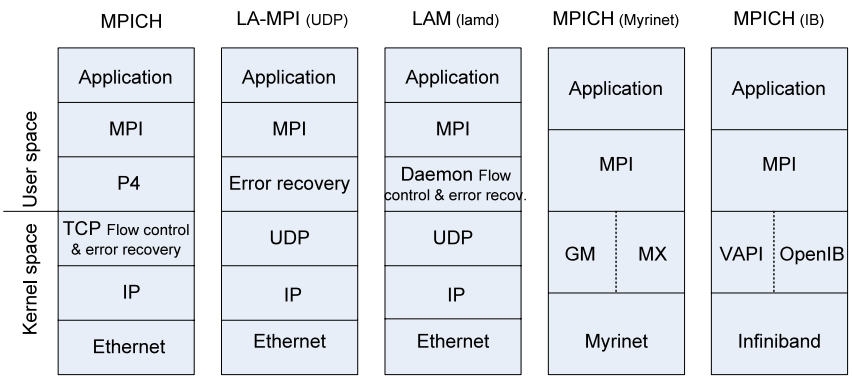


Fig. 3 Examples of protocol stacks involved in the communication at every compute element for different MPI implementations and IN

while adding capabilities to interact with the Traffic Manager (sending and receiving frames). This way we can reuse protocol stack and NIC driver. However, we need to implement all the details of the simulated hardware, with all its control registers and low level accesses (writes in memory mapped registers, interruption handling, DMA accesses, etc.) Neither this option nor the previous one requires us to manipulate or re-implement the protocol stack.

Usually full-system simulation environments come with default hardware and drivers for that hardware, like Ethernet NICs. Support for other INs such as Myrinet [4], Infiniband [25] or the torus network of the Bluegene/L [1] is not readily available. These environments provide mechanisms to add new, user-designed hardware modules that can be integrated into the simulators. If we have an accurate description of a certain NIC, and we program a module that simulates this NIC, we could reuse existing software (drivers and protocol stacks) designed to run on actual hardware. For example, if we implement a very realistic Myrinet card module, we could re-use the GM drivers and the MPICH-GM MPI implementation. However, due to the difficulty of doing this accurate hardware modeling, this approach is often rejected, and multicomputer experimentation is done using default hardware (Ethernet) and protocol stacks (TCP/IP-over-Ethernet), drastically simplifying setting up the experiments.

3.3 Substitution of the full protocol stack

The third and most complex option is to program the NIC module for the simulator, the driver to run in Kernel space, and a full protocol stack—including a customized MPI implementation—on top of it. The NIC module would interface with the Traffic Manager, and the driver would take advantage of the (simulated) high-speed IN. The obvious advantage of this option is that we would have full control of the IN; experiments could be done evaluating the hardware, the software, the MPI implementation, or a combination of them. Results would be very realistic, but only if we are able to provide good-quality, bug-free software. This is, in fact, the main drawback of this option: the implementation effort is huge, and difficult to reuse. Any improvement in the simulated hardware propagates upwards: it may require driver changes, and probably MPI changes in order to take advantage of it. We need, thus, to find a trade-off between programming effort and flexibility. Reutilization of components allows us to use in our experiments good quality, well-proven software, but at the cost of using off-the-shelf components. The accurate simulation of a completely new proposal for an IN would require implementing the components that would be required if the network hardware were real, plus a detailed model of that hardware.

4 Synchronization mechanisms

In the previous sections we explained how full-system simulation of multicomputers is carried out via the combination of a collection of different simulators. These simulators are separate software entities that have different views of the passing of (simulated) time. This means that they have different simulation clocks, with different time units, and may even have different mechanisms to make those clocks advance. For example, Simics is event-driven and time is measured in CPU cycles, whose translation to actual time depend on the CPU speed, while INSEE is cycle-driven and its unit of time is a more abstract cycle: the time needed to route and move a phit (*physical* transmission *unit*) from an input port to an output port. Obviously, mechanisms are required to coordinate and synchronize those clocks in such a way that simulators for compute elements and IN advance at the same pace, as if a global clock was in use. The synchronization module takes care of this task.

The synchronization model can be strict or relaxed. Strict models are unapproachable, in terms of execution time, when performing a full-system simulation of a multicomputer, because they make exploitation of available parallelism (in the simulation platform) almost impossible. Thus, we will only consider relaxed models. In order to keep discussion simple only two simulators are considered: one that takes care of compute elements, and another one for the IN. However, discussed mechanisms can (and will) be extended to consider several, concurrent simulators for the compute elements.

One synchronization alternative is to allow the simulators to advance in lock-step mode. The compute elements simulator advances for a given amount of time (let us call it slice) and then stops. The IN simulator starts its execution and simulates the same amount of time (an equivalent one, if a translation of time units is required). It then stops and the compute elements simulator resumes its operation. Note that both simulators never run in parallel.

When a message is generated at a compute element, it is stored (with a timestamp) at an interfacing queue. Later, the IN simulator will simulate the same time slice. It will process the queue, taking care of this injection, at the right time. When the IN signals that a message has to be delivered to a compute element, again this event is stored at an interfacing queue. However, we have a problem here: that queue will not be processed until the next slice. The compute elements simulator cannot process a message from the past; therefore all messages received during a given slice will be processed at the beginning of the next slice. In other words, messages will suffer, due to this relaxed synchronization approach, a false delay, ranging from 0 to the duration of the slice.

The other alternative allows the exploitation of parallelism in the simulation environment. We can let both simulators advance in parallel, without interchanging information. After consuming a slice, both simulators exchange lists of events. The compute elements simulator passes the list of messages generated at the slice just consumed, to be processed by the network, and the IN simulator passes the list of messages that have arrived to the destination compute elements. Note how this approach introduces two artificial delays: injection is delayed until the beginning of the next slice. Delivery, as in the previous option, is also delayed. Again, the importance of these delays depends on the slice length. A second effect is that message injections into the IN are done in bursts, at the beginning of each slice, which may impose unnecessary contention.

In both models, a very short slice length would provide maximum fidelity, but at the cost of stopping simulators very often. A long slice substantially accelerates experimentation, but introduces artificial delays that can have important, negative effects on our measurements

5 A proposal for full-system simulation of multicomputers

As we have already stated, our tool of choice to simulate the compute nodes that interchange packets through a network is Simics. From the options described in Sect. 3 to interchange information between simulators, we have chosen the second one: we have substituted the module that models an Ethernet NIC (a DEC21143), using another one almost identical, but capable of communicating with an external Traffic Manager module. Regarding synchronization, we use the second of the options presented in Sect. 4: the network simulator and Simics run in lock-step mode. A certain degree of parallel simulation is performed, but only when running Simics—further details will be given later in this section.

INSEE provides a flexible environment to perform simulations of INs. Its two main modules (see again Fig. 1) are FSIN (a cycle-driven, functional simulator of interconnection networks) and TrGen (a traffic-generation module). The latter allows us to feed simulations with three different kinds of workloads: synthetic traffic patterns defined by statistical distributions, traces obtained from actual parallel application executions, and full-system simulations as described in this paper.

In the experiments detailed in the next sections, we will present results for INs built using ring topologies. The reader should note that we have chosen rings *only* because they are more prone to congestion than other kind of networks like torus of higher dimensions or fat-trees, not because we think this is a good choice of topology; in fact, INSEE can deal with k-ary n-cube networks of higher degree and radix and

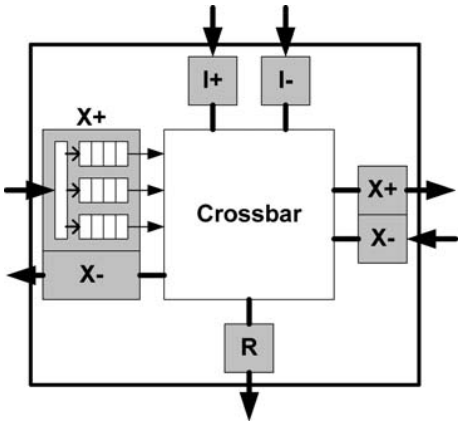


Fig. 4 Model of router simulated by FSIN for 1D networks, with a detailed view of the $X+$ input port showing the 3 virtual channels that share this link

even with multistage networks as fat-trees. At any rate, the network is composed of a collection of routers, each of them connected to several (for this work, just two) neighboring routers and to a compute node. Figure 4 represents a model of these routers. Three virtual channels (VCs) share each physical channel of the router: an Escape channel (governed by the bubble routing rules [26]), and two adaptive channels.

Note that a ring has just one minimal path from source to destination, *i.e.* packets cannot adapt. Thus, the only difference between the Escape VC and the other two is that accesses to the adaptive VCs are not restricted by the bubble rules. Each node is able to simultaneously consume several packets arriving to the reception port. There are two injection ports, and the interface should perform a pre-routing decision: packets moving towards the $X+$ axis are stored in the $I+$ injection port, and those towards $X-$ go to the $I-$ injection port. Transit and injection queues are able to store 4 packets of 16 phits (unit of transit through the wires) each. Phit length is 4 bytes and therefore the link bandwidth is 32 bits per cycle.

Regarding the simulation of the compute nodes, we use 8 instances of Simics, each one simulating 8 nodes, for a total of 64 simulated nodes. Each node runs a full Red Hat 7.3 operating system, and can be configured to use a variety of MPI implementations. As we use an Ethernet-like simulated NIC, we can use either TCP/IP/Ethernet or UDP/IP/Ethernet as low-level protocol stacks.

The process of sending a message to another different process in a multicomputer comprises several steps. In Fig. 5 we expose the different modules performing these actions in our environment. First, the message is segmented into network (Ethernet) frames, which are encapsulated with different headers depending on the protocol stack used on the multicomputer. Then, these frames are sent to the NIC driver that injects them into the IN. Instead of injecting them on a real IN, the Traffic Manager module extracts them from

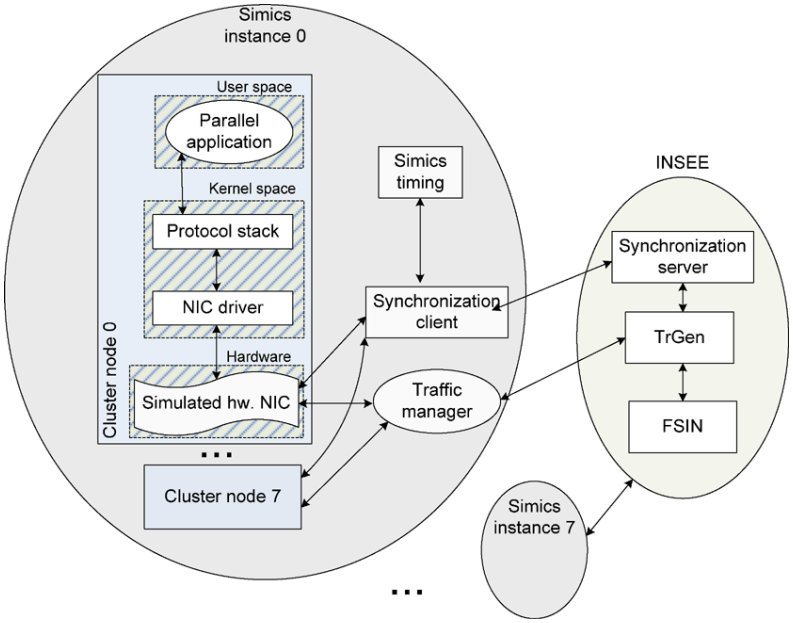


Fig. 5 Elements of our full-system simulation environment that simulates an MPI application running on top of an INSEE (simulated) network

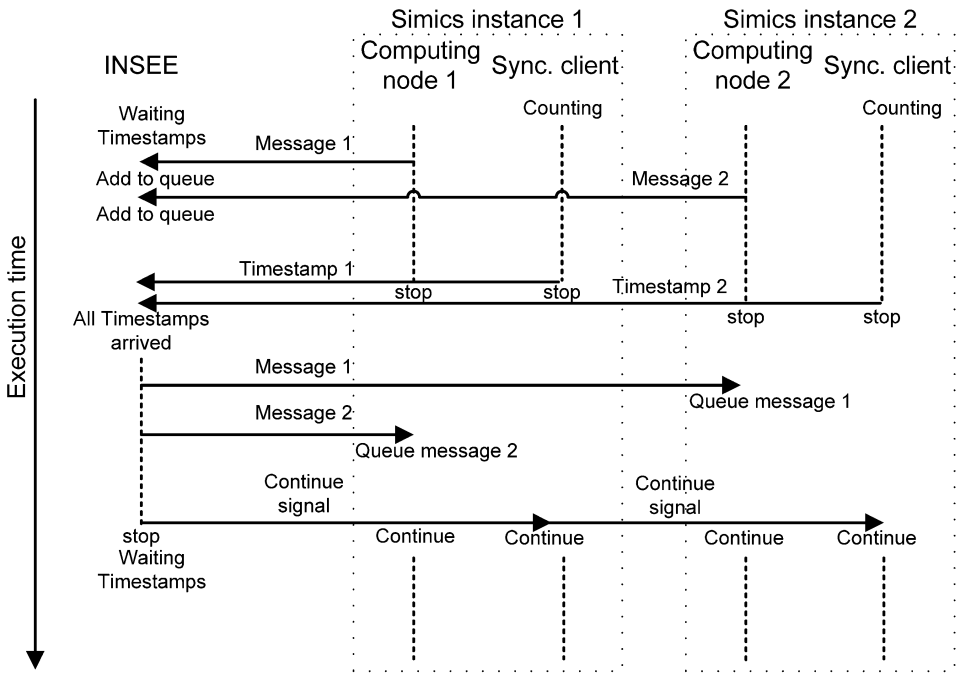


Fig. 6 Synchronization mechanism between Simics instances and INSEE, and message routing. Only one computing element per Simics instance is shown

the simulated hardware NIC and sends them to INSEE. This Traffic Manager module is responsible for sending and receiving frames from the TrGen module of INSEE. As we use a cluster of commodity computers to simulate a multicomputer, the traffic Manager sends the frames to the computer that runs INSEE.

TrGen is in charge of providing the workload for FSIN, the IN simulator core of INSEE. Frames received in TrGen are further divided into packets that are put in the right injection queue of the corresponding FSIN router. When the synchronization server signals FSIN to run, it simulates how packets travel through the network, and delivers them to their destination routers. Once all packets of a frame have arrived at the destination, TrGen will regenerate the frame putting packets together and will send the frame to the corresponding Traffic Manager on the destination node.

The Traffic Manager injects the received frames into the simulated hardware NIC. When the synchronization client resumes the run of Simics, the arrival of a frame causes an interruption that will be attended by the NIC driver. The

frame is then processed, *i.e.* the headers of the protocol stack are removed and the message is rebuilt and delivered to the application process.

The ability of running several compute nodes in each Simics instance requires the utilization of *two* different synchronization mechanisms. The first one is used to coordinate all compute nodes inside a Simics instance: every node runs a specific number of cycles (slice) and then the next compute node and so on, in a round-robin fashion. This mechanism is integrated into the own Simics working. The second synchronization mechanism is used to coordinate the compute nodes and INSEE. We use a client-server model in a lock-step mode to do it. There is a synchronization server in INSEE and a synchronization client on each Simics instance, as shown in Fig. 5. When all nodes in a Simics instance have completed a slice, the synchronization client stops that instance, and then sends a *timestamp signal* to the synchronization server asking for permission to run another slice. During a running slice, all generated network traffic is kept in the injection queues.

When the synchronization server has received timestamp signals from all Simics instances, INSEE allows FSIN to run a slice, routing the received packets up to their FSIN router destination. When FSIN finishes its slice, it sends a multicast timestamp signal to all synchronization clients, allowing them to resume the execution of the compute nodes. This is shown in Fig. 6.

Recall that this synchronization mechanism allows network injections to be simulated precisely, but deliveries are artificially delayed until the start of the next slice. The main difficulty here is to find a trade-off between the high execution speed provided by long slices and the accuracy obtained from short ones.

6 Experimental environment

As an example of the capability of our environment, our proposal has been used to study the effects of network-based congestion control in the execution speed of MPI parallel applications. In order to better understand the experimental work, we start this section defining the congestion control mechanisms under study.

6.1 Network congestion control

Congestion may appear when the utilization of resources inside the IN is close to its limits; its negative effects include throughput and delay degradation. If no action is taken when congestion appears, it soon spreads through the whole network. Congestion control techniques usually limit packet injection as soon as the network presents signs of congestion. There are different ways of diagnosing these signs and techniques to avoid congestion [8], based on global knowledge like in [36], distributed like RECN [5] or based on information of the local router buffers like LBR [16]. The torus network of the IBM BlueGene/L includes a mechanism that works prioritizing in-transit traffic—we call this IPR (In-transit Priority Restriction).

In an initial set of experiments, we evaluated two of these congestion control mechanisms for INs, IPR and LBR, both based on locally available information, with good performance and minimal implementation costs [16]. When using IPR, priority is given to in-transit traffic for a given number P of cycles. In those cycles, the injection of a new packet is only allowed if it does not compete with packets already in the network. P may vary from 0 (no restriction) to 1 (absolute priority to in-transit traffic). The Local Buffer Restriction (LBR) mechanism has been designed specifically for adaptive routers that rely on Bubble Flow Control [26] to avoid deadlock in the escape sub-network. LBR extends the bubble restriction to all new packets that enter the network. Subsequently, a packet can only be injected into an adaptive virtual channel if such action leaves room for at least B packets in the transit buffer associated to that virtual channel. The parameter B indicates the number of buffers reserved for in-transit traffic. In other words, congestion is estimated by the level of buffer utilization.

6.2 The experimental set-up

The effects of congestion were studied on the previously described environment. The testbed system is composed by 64 compute nodes connected through a ring network. The number of nodes (64) is a consequence of the availability of resources to obtain actual traces and run the full-system simulation environment. As we stated before, the choice of a ring (instead of a more reasonable 8×8 torus) is because the ring is, for a given number of nodes, more prone to congestion than a 2D torus. The system was composed by 64 Intel Pentium-4 processors, running Red Hat 7.3 with kernel 2.4 at 200 MHz (1 Simics cycle = 5 ns), with 64 MB of RAM. Unless otherwise stated, the synchronization slice was 10000:200, meaning that 200 INSEE cycles are made equivalent to 10000 Simics cycles; this results in a link bandwidth of 128 Mb/s, approximately the speed of a Fast Ethernet.

We used the IPR and LBR network congestion control mechanisms. For the following experiments, we fixed the values: $P = 1$ (the maximum priority) and $B = 3$ (inject in an adaptive channel only when its queue is empty or almost empty). Note that the "Base" case corresponds to both mechanisms being deactivated, *i.e.* $B = 0$ and $P = 0$.

The applications used to perform the experiments were a subset of the A class of the NAS Parallel Benchmarks [20] (NPB), a well-known, allegedly representative set of parallel application workloads often used to assess the performance of multicomputers. To simplify descriptions and

discussions, we have focused on three of these benchmarks: BT, CG and IS. BT is computation-biased (although not embarrassingly parallel) and should show how gains at the network level do not have a great impact on execution time. In contrast, CG and IS are communication-intensive, so the effects of network changes should be more clearly visible.

We started our study with an initial evaluation using traces captured in a small-size cluster with 8 dual-core Intel Xeon nodes, connected via Gigabit Ethernet. Several tasks time-shared each processor, so that the timing information about CPU intervals stored in the trace records is not valid. Nevertheless, in the trace-based simulations we do not use CPU times, just keep causal relationship between messages, as if we were using infinite-speed processors. The mechanisms used to capture the traces, as well as the methodology to perform the simulation, are described in detail in [17, 30]. With this trace-based simulation we obtained some *predictions* of the maximum performance improvement achievable with the use of congestion control mechanisms. Then, we used our full-system simulation environment to verify these predictions. Obviously, results obtained with this environment should be consistent with those predicted by the trace-based study, but reduced in magnitude because of the inclusion of the computation part.

As our environment allows us to use different protocol stacks, we have tested a few, different ones: from the collection shown in Fig. 3 we have evaluated those with Ethernet at the hardware level. When TCP is used at the transport level, we use the usual version of TCP Reno [7].

7 Experiments and discussion of results

In this section we describe in detail the experiments carried out, and analyze the results. The first set of experiments consists of several full-system simulations designed to evaluate the effects of network-level congestion control on the execution speed of NPB applications. The results of the trace-based study are also plotted in the figures, to compare the predictions obtained this way with the results from the full-system study. From this set we learn about the interactions between end-to-end (TCP) and network-level congestion control mechanism, an issue that is further studied in the second set of experiments. The last part of this Section is devoted to explore the effects of synchronization frequency in simulation accuracy and execution times.

Note that all the reported results normalize execution times to the Base case, which is, as described before, the one without IN-based congestion control. Experiments are repeated 10 times, and in the figures we plot the average values of results. Variances, summarized in tables, are included too.

7.1 Trace-based simulation

The most relevant results of these experiments are plotted in Fig. 7 (light-grey bars). They predict that the reduction of execution time achievable from applying congestion control mechanisms should be, in the best case, a 15% (IS benchmark). However, we should expect small performance drops in applications such as CG: around a 3%. Both network-level congestion control mechanisms are beneficial for BT and IS, because their traffic patterns consist of large messages that are able to saturate the IN. In contrast, CG is harmed by these mechanisms, due to its traffic pattern, which consists of sequences of small messages, arranged by dependency chains. Thus, congestion does not appear, and any mechanism that restricts packet injection is harmful, because it imposes unnecessary delays that accumulate and increase the overall execution time.

The reader should note that the method we use to carry out trace-based simulation exaggerates the potential advantages/disadvantages of a given congestion control mechanism, because we do not take into account the CPU time (we only simulate the interchange of messages). Real applications only spend a portion of its time performing communications; hence our prediction applies only to this portion. This limitation does not apply to full-system simulations.

7.2 Assessing the effects of network-level congestion control

The full-system simulation did not, to our surprise, confirm the predictions of the trace-based simulation: results (summarized in Fig. 7 and Table 1, along with those obtained with traces) were in some cases much more favorable than expected, and in some others poorer than expected. The causes of these mismatches were some unaccounted-for interactions between end-to-end congestion control (TCP incorporates its own mechanisms) and the network-level congestion control (those under test, IPR and LBR).

In the case of experiments made with MPICH and without any network congestion control mechanism, application execution times were negatively affected by TCP's wrong estimation of timeouts, triggering retransmissions and continuously activating the slow start protocol [7]. Those made the whole execution very slow in saturated networks. In contrast, when IPR or LBR were applied, jitter was reduced, which helped TCP to determine its timeout values which reduced the retransmissions and, consequently, the occurrences of the slow start mechanism. Therefore, both IPR and LBR caused two overlapping effects:

- For most applications, the flow of packets through the network was accelerated.
- In all cases, they helped TCP, allowing the applications to reach higher throughput.

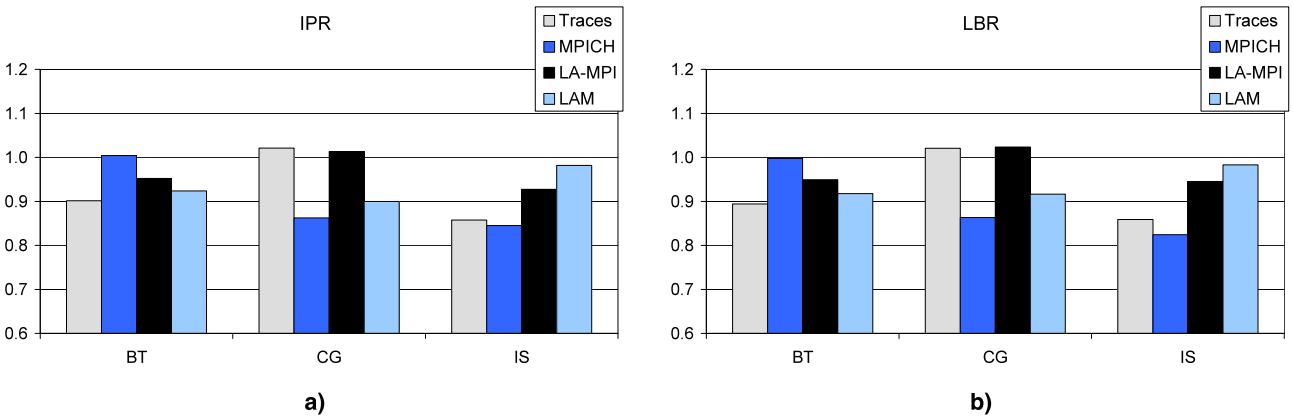


Fig. 7 Measured execution times relative to the base case (IPR and LBR deactivated). (**a**) Results obtained when IPR is activated. (**b**) Results obtained when LBR is activated

Table 1 Variances of the experiments plotted in Fig. 7

	Base			IPR			LBR		
	BT	CG	IS	BT	CG	IS	BT	CG	IS
MPICH	2.76E-04	8.12E-04	5.23E-04	2.70E-04	1.57E-05	6.47E-04	9.36E-04	5.85E-06	1.86E-03
LA-MPI	2.68E-04	9.31E-05	1.55E-03	2.80E-04	1.54E-03	1.41E-03	4.04E-04	1.95E-03	2.16E-03
LAM	7.96E-04	4.13E-04	4.98E-03	2.33E-04	1.03E-03	4.11E-03	4.29E-04	1.19E-03	4.40E-03

The second effect was unexpected, and was more significant than the first one. This explains the mismatch in our predictions.

In order to avoid these interactions, we tested other MPI implementations with protocol stacks that do not include TCP. One of those is LA-MPI over UDP. This implementation performs error control at application level to avoid message losses (messages are dropped when intermediate buffers are full). This error control slightly affects the execution time. Despite this, the absence of TCP made LA-MPI a good platform for experimentation. However, as this project is no longer supported, we decided to search for another non-TCP based MPI implementation.

LAM over UDP routes all messages through a daemon present in every compute element. This adds two hops to every message. And these daemons also do the flow control and error recovery not available while using UDP. For this reasons this configuration is very slow and also affects the performance in a similar way as TCP does. This can be seen in Table 2.

We can conclude that the reutilization of components, in this case protocol stacks, may look as a good idea, because it reduces the time of setting-up an evaluation environment and reduces programming errors, but the price to pay may be too high: it may introduce unforeseen interactions that can magnify, hide or even invalidate the results obtained. A more detailed explanation of this issue is provided in [21].

Table 2 Simulated time needed to run an iteration of each benchmark for different MPI implementations in the full-system simulation environment. Average of 10 runs

	BT	CG	IS
MPICH	4.52 s	5.89 s	4.21 s
LA-MPI	4.51 s	5.60 s	4.10 s
LAM/UDP	5.09 s	10.88 s	10.38 s

7.3 Congestion control and network speed

Once the interaction between end-to-end and network congestion control mechanisms was detected, we carried out additional experiments in order to assess the influence of these mechanisms for different network speeds. In addition to the experiments described in the previous section, for a network of 128 Mb/s, we repeated the experiments on a faster network of 1280 Mb/s and only with MPICH, one of the most widely-used noncommercial MPI implementations. Results are summarized in Fig. 8 (averages) and Table 3 (variances).

An increase in network speed makes TCP perform even worse when there is no network-based congestion control mechanism. When the traffic is not enough to fill up the network and the network speed is high, TCP estimates correctly its timers, and is able to deliver traffic with high throughput levels. However, in those phases where contention for network resources appears, because the traffic pattern requires

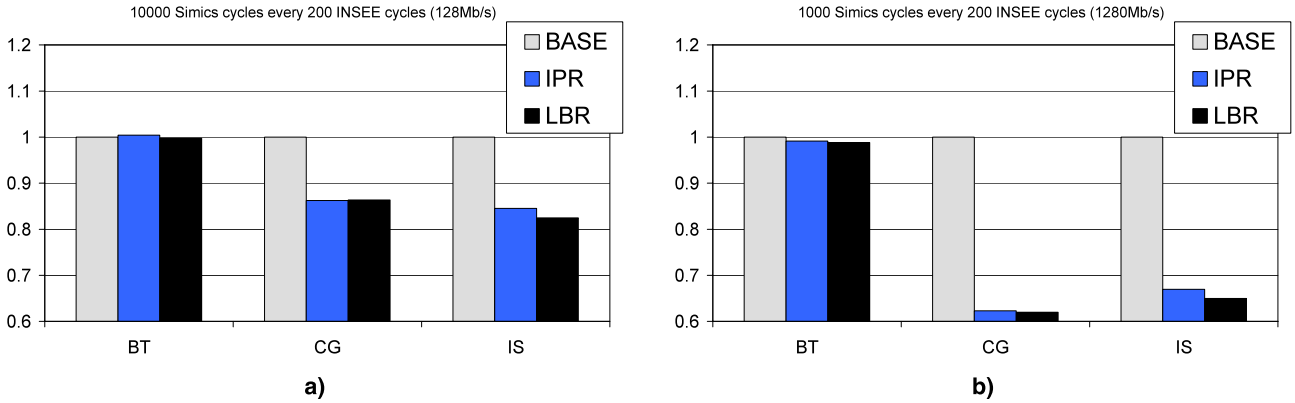


Fig. 8 Measured execution times relative to base case (IPR and LBR deactivated). (**a**) Obtained in a 128 Mb/s network. (**b**) Obtained in a 1280 Mb/s network

Table 3 Variances of the experiments plotted in Fig. 8

	128 Mb/s			1280 Mb/s		
	BT	CG	IS	BT	CG	IS
Base	2.76E-04	8.12E-04	5.23E-04	4.90E-05	2.94E-03	1.38E-02
IPR	2.70E-04	1.57E-05	6.47E-04	8.32E-05	2.24E-04	3.90E-04
LBR	9.36E-04	5.85E-06	1.86E-03	1.03E-04	3.84E-04	8.17E-05

it, packets slow down. Some of them even "get lost" from the point of view of TCP, because they arrive too late. This jitter activates TCP's flow control and error recovery mechanism. This activation slows down the performance in the same way as a slower network because the slow-start algorithm is network independent. Again, the utilization of network-level congestion control reduces jitter, and improves performance. As we can see, the effect of IPR/LBR is more beneficial at higher network speeds, not only in terms of average values but also in reduction of variance; this means that the adverse behavior of TCP gets worse in faster networks. Note the great differences in performance for the CG and IS benchmarks, those that are more communication-intensive.

7.4 The simulation speed/accuracy trade-off

We explained in previous sections that an issue when linking two different simulators is to fine-tune the synchronization among them. In particular, we need to define the slice duration: the period of time in which a given simulator advances without synchronizing with the rest. The duration of the slice affects both the accuracy and the simulation performance. The longer the slice, the larger the delay and jitter, hence the accuracy decreases. In addition, every time the simulators synchronize, a penalty has to be paid; therefore simulation performance is better for longer slices because the actual time to run the simulation is shorter. The effect of a short slice is just the opposite: more synchronization overhead, but better accuracy because delay and jitter are reduced.

We ran another set of experiments to measure the accuracy versus performance relation due to the synchronization mechanism. We used an MPICH over TCP protocol stack, fixed the network speed at 128 Mb/s, and used three different synchronization values: 100000:2000, 10000:200 and 1000:20. We took the time reported by the simulator to execute an iteration of the benchmarks under study (BT, CG and IS) with and without IPR, and also the wall-clock time that the simulator required to perform the experiment. Results are plotted in Fig. 9.

Figure 9a shows that, in terms of simulator-reported time, there is little difference in terms of simulated time per iteration between the highly synchronized case (1000:20) and the one in which synchronization frequency is reduced to one tenth of that (10000:200), in which benchmark execution is at worst only 3.28% slower (BT Base). This small difference can be explained because of the additional delays introduced by the lock-step synchronization. In the 10000:200 case, a packet generated at a node may need to wait up to 9999 Simics cycles (50 µs) before being injected into the network. This resulted in large delay variations. In the 1000:20 experiments, the worst-case additional delay was reduced to 999 Simics cycles (5 µs)—reducing jitter and allowing TCP to perform better.

When synchronization is infrequent, as in the 100000:2000 case, results change in a significant way. Huge delays and

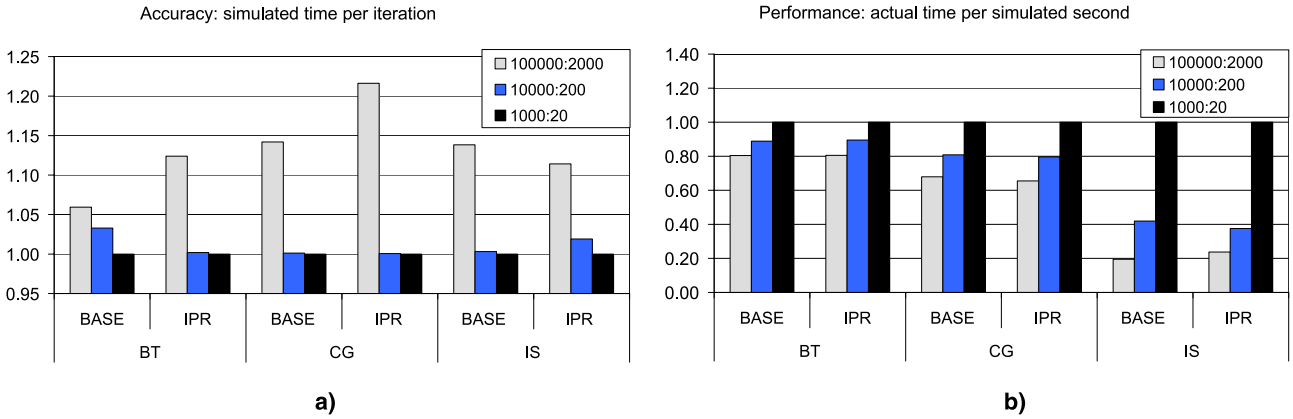


Fig. 9 Time measurements for each benchmark with different synchronization parameters. Results relative to the most synchronized case (1000:20) (**a**) Simulated time to complete each iteration. (**b**) Actual time to simulate a second

jitter are artificially introduced, as we force packets to wait up to 99999 Simics cycles (500 µs). With these unstable, long delays TCP cannot make good estimations of delays, and activate too often the slow-start mechanism.

Besides, we could unnecessarily saturate the IN. The maximum injection rate of a Fast Ethernet is of 148.800 [9] frames of minimum size (72 Bytes, that will require 2 FSIN packets) per second without collisions, which gives us an injection rate of a frame every 1344 Simics cycles, using the parameters of our simulation environment. This means that, at every synchronization step, every node could have up to 70 frames pending for injection, which could heavily saturate the network. It must be taken into account that our environment uses a regular device (Fast Ethernet) on top of regular protocols (MPICH/TCP/IP/Ethernet) over a custom IN, which is not designed for them. Therefore, simulation artifacts are triggering mechanisms of TCP that should not operate, generating inaccurate results.

Figure 9b represents the wall-clock time to simulate a second of each benchmark. This time is inversely proportional to the slice length. The longer the slice, the shorter the time needed to simulate a second, up to only 20% of the time needed if we compare the longer slice (100000:2000) with the shorter slice (1000:20) for the IS base case. The savings of time can be between 10% up to 60% if we increase a magnitude the slice duration. Therefore, it is clear that a carefully study is needed to choose the synchronization slice duration. We must choose a slice short enough to get reliable and correct results, and long enough to complete simulations in a reasonable time. In our case, the 10000:200 slice appears to be a good choice, because we only loose a 3% on accuracy on the worst case, and the simulation lasts up to 60% less time than the case with the shortest synchronization slice.

In summary, as a performance study requires many simulation runs, researchers are encouraged to spend some time exploring this speed/accuracy trade-off, searching for those slice durations that provide faster simulation runs but without sacrificing truthfulness of results. Actual values to explore depend on factors such as (relative) speeds of processors and network, so we cannot provide a rule of thumb to get the right durations. Also, some precision can be sacrificed in initial simulation runs, switching to slower set-ups when accuracy is essential.

8 Conclusions

Full-system simulation is a very complex matter, complexity that is greatly increased when trying to simulate not just a computer, but a collection of networked machines—especially if the network and the interfaces differ from the traditional LAN devices available in simulation environments. It is also an intensive resource-consuming task. The simulation of a cluster of computers may require an actual machine with similar characteristics to the one under study, but the speed at which we obtain performance results would be several orders of magnitude slower.

In this paper, we show that full-system simulation of INs requires a large collection of interrelated (software) components, which, in many cases, have to be done from scratch, or re-used from those provided by the simulation environment being used. The reutilization allows for important reductions of implementations effort and errors, but implies some risks of using, for a given purpose, components designed for different (although related) purposes, and may lead to inaccurate or even invalid simulation results. The synchronization between parts of the simulation environment needs also to be carefully designed, in order to find a good trade-off between simulation fidelity and execution time of the experiments. We have presented our proposal of combining INSEE with Simics, which has allowed us to learn, by experience, about

the plethora of factors that have an effect on the quality of results: protocol stacks, MPI implementations, drivers, synchronization modules, etc. Still, this environment allows us to evaluate, on realistic scenarios, the effects of network-level congestion control, and the interactions between end-to-end and network level mechanisms. As far as we know, no other tool (or combination of tools) is available providing the same set of features.

As future lines of work, we plan to improve our toolset including models of the hardware NICs used in current super-clusters; in particular, our target is to have a model of an Infiniband HCA, and to add models of Infiniband networks into FSIN. Also, as full-system simulation is very slow, we are working in improving the other, significantly faster, traffic-generation modules of INSEE (synthetic patterns and traces), with special focus on synthetic traffic that, while algorithmically generated, is inspired in actual applications [22].

Acknowledgements This work has been supported by the Ministry of Education and Science (Spain), grant TIN2007-68023-C02-02, and by grant IT-242-07 from the Basque Government. Mr. Javier Navaridas is supported by a doctoral grant of the UPV/EHU.

References

1. Adiga, N.R., et al.: Blue Gene/L torus interconnection network. IBM J. Res. Dev. **49**(2/3) (2005)
2. Binkert, N.L., Hallnor, E.G., Reinhardt, S.K.: Network-oriented full-system simulation using M5. In: Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), Feb. 2003
3. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The M5 simulator: modeling networked systems. IEEE Micro **26**(4), 52–60 (2006)
4. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., et al.: Myrinet. A gigabit per second local area network. IEEE Micro **15**(1), 29–36 (1995)
5. García, P.J., Quiles, F.J., Flich, J., Duato, J., Jhonson, I., Naven, F.: Efficient, scalable congestion management for interconnection networks. IEEE Micro **26**(5), 52–66 (2006)
6. IBM: IBM full-system simulator for the cell broadband engine processor. Available at http://alphaworks.ibm.com/tech/cellsystemsim (May 2008)
7. Jacobson, V.: Congestion avoidance and control. Comput. Commun. Rev. **18**(4), 314–329 (1988)
8. Jain, R.: Congestion control in computer networks: issues and trends. IEEE Netw. **4**(3), 24–30 (1990)
9. Karlin, S., Peterson, L.: Maximum packet rates for full-duplex ethernet. Technical Report TR–645–02, Princeton University (February 2002)
10. LA-MPI Home Page: The Los Alamos message passing interface. Available at http://public.lanl.gov/lampi/ (May 2008)
11. LAM/MPI Home Page: LAM/MPI parallel computing. Available at http://www.lam-mpi.org/ (Apr. 2008)
12. Liu, J., Wu, J., Panda, D.K.: High performance RDMA-based MPI implementation over infiniBand, Int. J. Parallel Program. (2004)
13. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: a full system simulation platform. IEEE Comput. **35**(2), 50–58 (2002)
14. Martin, M.M.K., et al.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Sigarch Comput. Archit. News **33**(4), 92–99 (2005)
15. Mauer, C.J., Hill, M.D., Wood, D.A.: Full-system timing-first simulation. In: ACM SIGMETRICS, June 2002
16. Miguel-Alonso, J., Izu, C., Gregorio, J.A.: Improving the performance of large interconnection networks using congestion-control mechanisms. Perform. Eval. **65**, 203–211 (2008)
17. Miguel-Alonso, J., Navaridas, J., Ridruejo, F.J.: Interconnection network simulation using traces of MPI applications. Int. J. Parallel. Program. (to appear). DOI 10.1007/s10766-008-0089-y
18. MPI Forum: MPICH home page. Available at http://www-unix.mcs.anl.gov/mpi/mpich/ (May 2008)
19. Myricom Documentation and Software Downloads. Available at http://www.myri.com/scs/ (May 2008)
20. NASA Advanced Supercomputing (NAS) division: NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Resources/Software/npb.html (May 2008)
21. Navaridas, J., Ridruejo, F.J., Miguel-Alonso, J.: Evaluation of interconnection networks using full-system simulators: lessons learned. In: Proc. 40th Annual Simulation Symposium, Norfolk, VA, March 26–28, 2007
22. Navaridas, J., Miguel-Alonso, J., Ridruejo, F.J.: On synthesizing workloads emulating MPI applications. In: The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08). April 14–18, 2008, Miami, Florida, USA
23. OPNET Technologies, Inc. corporate web page, available at http://www.opnet.com (May 2008)
24. Pai, V.S., Ranganathan, P., Adve, S.V.: RSIM: an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In: IEEE TCCA New., Oct. 1997
25. Pfister, G.F.: Aspects of the InfiniBand(tm) architecture. In: Third IEEE International Conference on Cluster Computing (CLUSTER'01), October 2001, pp. 369
26. Puente, V., Izu, C., Gregorio, J.A., Beivide, R., Vallejo, F.: The adaptive bubble router. J. Parallel Distrib. Comput. **61**(9), 1180–1208 (2001)
27. Puente, V., Gregorio, J.A., Beivide, R.: SICOSYS: an integrated framework for studying interconnection network in multiprocessor systems. In: Proceedings of the IEEE 10th Euromicro Workshop on Parallel and Distributed Processing, Gran Canaria, Spain (2002)
28. Puente, V., Gregorio, J.A., Vallejo, F., Beivide, R.: Immunet: a cheap and robust fault-tolerant packet routing mechanism. In: International Symposium on Computer Architecture (ISCA), June 2004, pp. 198–211
29. Ridruejo, F.J., Miguel-Alonso, J.: INSEE: an interconnection network simulation and evaluation environment. In: Proc. Euro-Par 2005. Lecture Notes in Computer Science, vol. 3648, pp. 1014–1023
30. Ridruejo, F.J., Gonzalez, A., Miguel-Alonso, J.: TrGen: a traffic generation system for interconnection network simulators. In: International Conference on Parallel Processing, 2005. 1st. Int. Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems (PEN-PCGCS'05). ICPP 2005 Workshops, 14–17 June 2005, pp. 547–553
31. Rosenblum, M., et al.: Complete computer system simulation: the SimOS approach. IEEE Parallel Distrib. Tech. **3**(4), 34–43 (1995)
32. Schaelicke, L., Parker, M.: ML-RSIM reference manual. Tech. Report 02-10, Department of Computer Science and Engineering, Univ. of Notre Dame, Notre Dame, ID (2002)

33. SMART group at the U. of Southern California. FlexSim 1.2. Available at http://ceng.usc.edu/smart/FlexSim/flexsim.html (May 2008)
34. The Chaotic Routing Project at the U. of Washington. Chaos Router Simulator. Available at http://www.cs.washington.edu/research/projects/lis/chaos/www/chaos.html (May 2008)
35. The Network Simulator ns-2. Available at http://www.isi.edu/nsnam/ns/ (May 2008)
36. Thottethodi, M., Lebeck, A.R., Mukherjee, S.S.: Exploiting global knowledge to achieve self-tuned congestion control for k-ary n-cube networks. IEEE Trans. Parallel Distrib. Syst. **15**(3), 257–272 (2004)
37. Wenisch, T.F., Wunderlich, R.E., Ferdman, M., Ailamaki, A., Falsafi, B., Hoe, J.C.: SimFlex: statistical sampling of computer system simulation. IEEE Micro **26**(4), 18–31 (2006)

Fco. Javier Ridruejo obtained his MEng in Computer Engineering from the University of the Basque Country, Gipuzkoa, Spain, in 2001. He is currently pursuing his PhD at the Department of Computer Architecture and Technology of the same university. His research interests include high-performance computing infrastructures, interconnection networks for parallel and distributed systems, and performance evaluation of parallel systems.

Jose Miguel-Alonso received the PhD in computer science from the University of the Basque Country, Gipuzkoa, Spain, in 1996. He is a Full Professor at the Department of Computer Architecture and Technology of the University of the Basque Country. His research interest include interconnection networks for parallel systems, network (cluster, grid) computing, performance evaluation of parallel and distributed systems, and scheduling for parallel processing.

Javier Navaridas obtained his MEng in Computer Engineering from the University of the Basque Country, Gipuzkoa, Spain, in 2005. Since then he is pursuing his PhD at the Department of Computer Architecture and Technology of the same university. His research interest include interconnection networks for parallel and distributed systems, and performance evaluation of parallel architectures, with emphasis on simulation and characterization of application's behaviour.

Chapter 11. Application micro-kernels

Full reference:
Javier Navaridas, Jose Miguel-Alonso, Francisco Javier Ridruejo.
On synthesizing workloads emulating MPI applications.
The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08). April 14-18, 2008, Miami, Florida, USA.

We deepened our research in the synthetic traffic generation model to better resemble actual applications by defining micro-kernel patterns that most applications use. This workload fills the gap between classical synthetic traffic based on statistical distributions, and trace-based traffic. It mimics applications behavior, implementing some of the most common communication patterns used in massively parallel applications. This workload keeps causality among messages and, as the micro-kernel applications are synthetically generated, they can be generated for every network size and topology, solving the main problems of the trace-based traffic generation.

We compared several topologies in the case study of this paper, as explained in Chapter 3.2. We tested a crossbar, a torus and a fat-tree, using micro-kernel applications to show how the network topology influence the execution of an application, because of the scheduling of its communication pattern onto network resources. We also measured the temporal evolution of the micro-kernel applications under these topologies, showing the temporal evolution of the communication resources needed by applications.

The author of this dissertation collaborated in the design of the micro-kernels, and in the experimental work.

On synthesizing workloads emulating MPI applications

Javier Navaridas, Jose Miguel-Alonso, Francisco Javier Ridruejo.
Department of Computer Architecture and Technology,
The University of the Basque Country P.O. Box 649, 20080 San Sebastian, SPAIN
Tel. +34 943018019 Fax +34 943015590
{javier.navaridas, j.miguel, franciscojavier.ridruejo}@ehu.es

Abstract

Evaluation of high performance parallel systems is a delicate issue, due to the difficulty of generating workloads that represent, with fidelity, those that will run on actual systems. In this paper we make an overview of the most usual methodologies used to generate workloads for performance evaluation purposes, focusing on the network: random traffic, patterns based on permutations, traces, execution-driven, etc. In order to fill the gap between purely synthetic and application-driven workloads, we present a set of pseudo-synthetic workloads that mimic applications behavior, emulating some widely-used implementations of MPI collectives and some communication patterns commonly used in scientific applications. This mimicry is done in terms of spatial distribution of messages as well as in the causal relationship among them. As an example of the proposed methodology, we use a subset of these workloads to confront tori and fat-trees

1. Introduction

The interconnection network is the most characteristic element of any parallel computer. Its performance has a definite impact on the overall execution time of applications, especially for those that are fine-grained and communication intensive. Thus, we should not decide lightly the network that interconnects computing elements in a high performance computing site.

The evaluation of an interconnection network is a complex task that requires a complete model of the network technology we want to assess. Once we have the model of the system, we ought to measure its performance, and some important questions arise: How should we evaluate the network? Should we measure only raw performance? Is it a better idea to fine-tune the system for the set of applications that will run on it?

There is not just a valid answer to these questions. Often, the most important measurement of a system's performance consists simply in running Linpack, whose measured performance is the sorting-key for the top500 list [7]. Supercomputing sites can climb some positions in the list, improving their prominence, by boosting this single figure. In other places the evaluation is centered on the execution speed achievable by those applications that are currently used, or those that are expected to be used in the future. Alternatively, for the networking technology engineer, the most important evaluation concern should be the raw performance of the product, which means that its design will perform acceptably well in almost all scenarios.

In this paper we discuss some methodologies used to evaluate interconnection networks, and propose a set of synthetic traffic patterns that emulate pieces of scientific applications, both in terms of the spatial distribution of messages and causal relationship between them; the evolution of message lengths is discussed too. These workloads can be considered as performance measurement micro-kernels, because they evaluate the behavior of the system under different kinds of traffic that often appear within parallel applications. However, the effort of evaluating with these micro-kernels will be orders of magnitude smaller than that required to run a large set of complete applications.

The rest of this paper is arranged as follows. In Section 2 an overview of the most common methodologies used to evaluate parallel computing systems are discussed, showing the motivations to introduce application-inspired workloads. Section 3 introduces and justifies some workloads of this class, and the way they resemble common patterns used within applications. In Section 4 we show how these workloads can be used in an actual performance measurement study, using as an example the comparison of torus and fat-tree networks. Finally we close this paper with some conclusions and future lines of work in Section 5.

This work has been supported by the Ministry of Education and Science (Spain), grant TIN2007-68023-C02-02, and by grant IT-242-07 from the Basque Government. Mr. Javier Navaridas is supported by a doctoral grant of the UPV/EHU.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

2. Related work

Synthetic traffic patterns from independent sources [6] provide a good first approach to evaluate a network because they allow us to asses rapidly what the raw performance of a network is. Often, randomly generated traffic is used to evaluate systems: uniform, hot region and hot spot traffic patterns have been used in many studies [4][12]. Other commonly used patterns are those that send packets from each source node to a destination one as indicated by a certain permutation, usually defined as a function that takes as input the address of the source. Some examples of these permutations are: matrix transpose, bit complement, butterfly, perfect shuffle, bit reversal, etc. [11].

It is not common to find actual applications that are internally implemented using patterns like these, synthetic ones, in which traffic-generating nodes produce messages without coordinating among them. We can state that synthetic traffic patterns do not accurately mimic the behavior of any actual application [15]. For this reason, trace-driven simulation is often used to perform a more realistic evaluation of a system [6]. Feeding a simulator with a trace is not an easy task. To evaluate only the network of a parallel system we could implement a dummy model of the processing node, allowing it to inject messages into the network as fast as it can, ignoring the causality of messages and the computation intervals. This approach is a stress test of the network, because of the contention generated by all nodes injecting at the maximum pace.

It would be more realistic to maintain the causal relationship between all the messages in the trace [13]; in other words, if the trace states that there is a reception before a send, the node has to wait for the reception to be completed before starting with the send. This mechanism provides more fidelity than the inject-at-will model. To further improve the accuracy of the simulation, compute intervals (in which nodes do not inject load into the network) should be taken into account, maybe applying a CPU-scaling-factor in order to simulate a system with faster (or slower) CPUs than those used to capture the trace.

There are still some problems with the trace-driven approach that we should not ignore. Firstly, the information captured within the trace could be inexact due to the trace logging mechanism. Secondly, traces may reflect some of the characteristics of the system in which they have been obtained. Finally, traces from actual applications running in a large set of processors are difficult to obtain, store and manage traces, and these are precisely the ones of interest in performance evaluation studies.

A hybrid between the utilization of synthetic traffic patterns and traces is the estimation of probability distributions for destinations, inter-generation times and message lengths, using data extracted from actual traces to feed some distribution-fitting program. With the aid of these tools we can generate random traffic whose distribution resembles that of the traced application (when running in a particular system). However, as stated before, in actual applications causal relationships among messages are common, and this technique does not capture them. And, again, the inexactitude of the information within the trace (due to the characteristics of the system in which the trace was captured, and the intrusion of the logging process) may generate estimated distributions, or parameters of those, that are not valid.

In order to introduce causality in the simulation and fill the gap between trace-driven simulation and synthetic traffic patterns, a bursty traffic model was evaluated in [15]. This model uses synthetic traffic patterns and emulates application causality using a coarse-grained approach. The message generation process passes through a certain number of "*bursts*" or steps. During a burst each node is able to inject into the network only a given number of packets (b); after doing so, it must stall until the burst finishes, that is, until all the packets of the burst (generated at all the nodes) have been injected and received, being the network completely empty. Simulations with short bursts emulate tightly-coupled applications and, therefore, long bursts emulate loosely-coupled applications. Synchronization among application tasks is included in this model, but in a very primitive way (roughly a barrier every b packets); fine-grained dependencies among messages/tasks are not considered.

The most accurate methodology to evaluate a parallel computer would be running a detailed full-system simulation that includes interconnection network, compute nodes, their operative system, and the applications running on them. This is a very complex and error-prone task, as discussed in [9]. It is also a high resource-consuming methodology that could need a system similar in dimension to the one we want to evaluate. These are the main reasons to justify the limited utilization of execution-driven simulation to evaluate medium-to-large size distributed memory parallel computers.

Within this paper we will introduce a set of pseudo-synthetic workloads inspired in application-kernels aimed to fulfill the gap between purely-synthetic and application-driven workloads. These novel workloads emulate the spatial and temporal patterns of parallel applications. We arrange them into two sets: one that mimics the implementation of collective operations in MPI libraries, and another one that mimics the behavior of applications that rely in the use of virtual topologies, mostly virtual meshes and tori.

3. Proposed workloads

In this section we describe in detail and discuss our proposal to model application-like workloads. They will be described graphically as well as algorithmically. All the proposed workloads require the specification of a few parameters: the number of communication tasks (*N*), the length of the basic message (*S*) and, in the case of the workloads that rely on virtual topologies, the number of dimensions (*D*). We assume that tasks are identified from 0 to *N*-1. We define a *basic* message as the length of the data blocks generated at the nodes; the length of the messages that actually traverse the network may vary, if the workload requires the aggregation of messages; this issue will be discussed later. When the pattern is arranged as a sequence of stages (meaning that all nodes have to wait for messages before starting with new sends), we will denote the stage number as *t*.

The reader should note that, when we talk about distances in this section, we refer to the difference between source and destination node identifiers. The actual, physical distance of those nodes will depend on the routing and topological characteristics of the underlying network.

Send_to(node, length) and *Wait_from(node, length)* functions, in the algorithmic definitions of the patterns, do what their names suggest: send a message to a destination or wait until a message from the desired node arrives. For both operations, if the other communicating part (the receiver or the sender) is not defined, the call will do nothing, just return. In the virtual topologies subsection (3.2), the *Neighbor(node, dimension, direction)* function used in the algorithmic definition of those patterns will return the neighbor of a given node for the given dimension and direction; in the case of a (*virtual*) mesh topology, the returned value could be undefined for the nodes located at the borders of the network. *Length(stage, length)* returns the message length taking into account the length evolutions we will discuss latter.

Regarding graphical description of the patterns, small black arrows represent messages: the rounded end represents a message sending, and the arrowhead represents the waiting for the message at the destination task, so if there is an arrowhead before a round, the node has to wait until the first message arrives; only then the second message can be injected. Green arrows in section 3.1 represent MPI tasks and are arranged top-down, *i.e.* the green arrow at the top of the figure represents task 0, the one at the bottom represents task 7 (*N*-1). The green squares in section 3.2 represents MPI tasks and are arranged from left to right and then top-down, so the tasks in the topmost row are task 0, task 1, …

3.1. MPI collective operations

Most of the scientific parallel applications we have studied use MPI collectives to implement parts of their functionality: from scattering information to collecting results, or just to synchronize a group of processes. For example, the Fourier Transform, used in many scientific applications, could be implemented by means of MPI_Alltoall() and MPI_Reduce() collectives: the FT program included in the NAS Parallel Benchmarks [10] is implemented this way. Thus, we found of interest to assess the performance of the network when realizing collective operations, because of the bearing it will have in the overall performance of complete applications. In this subsection we propose traffic patterns that are the foundation used to implement some of the most common collectives.

3.1.1. Binary tree

The Binary Tree pattern (**BT**) provides an efficient implementation of some N-to-1 (*all-to-one*) collective operations, used by some MPI implementations that make no assumption about the underlying network or the node placement strategy [8]. In this pattern message interchanges are performed in *O(N)* in number of messages and *O(log N)* in time. It starts with a message at odd-numbered nodes, and ends when a "root" node (in our model, node 0) has received the last message from all nodes whose identifier is a power of two (included 2^0=1). The spatial and causal pattern is defined algorithmically and graphically in Fig. 1.

The MPI_Reduce() collective is implemented using this pattern, with a constant message length (that of the type of the reduced variable, commonly a float or an integer). Note that a computation phase is introduced before each send, in order to perform the reduction operation; however, as the CPU time used to perform this operation is usually very small (the time to perform a simple operation such as an addition or a comparison) we could ignore it. MPI_Gather() also makes use of this pattern. The length of the message duplicates at each stage (increases exponentially). In contrast, in MPI_Gatherv() the message length increases each stage in an application-defined way.

3.1.2. Inverse binary tree

A variant of the previous pattern, the Inverse Binary Tree (**IBT**) is a common implementation of 1-to-N (*one-to-all*) collectives [8], with the same complexity of the **BT** pattern. **IBT** starts with a single message in the "root" node and finishes when all the odd nodes receive a message. Readers may note that spatial and causal patterns are just the opposite of those of **BT**. **IBT** is defined algorithmically and graphically in Fig. 2.

$\forall node\ in\ [0, N)$:
BinaryTree $(node, S)$:
  for t in $[\ 0, \log_2 N\ )$
    if $(node \bmod 2^{t+1}) == 0$ *then*
      Wait_from$(node + 2^t, length(t, S))$
    elsif $(node \bmod 2^t) == 0$
      Computation()
      Send_to$(node - 2^t, length(t, S))$
    endif
  endfor

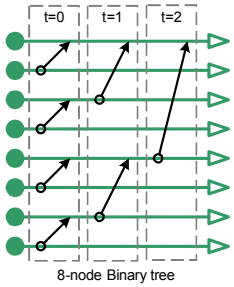


Fig. 1. Algorithmic and graphical definitions of Binary Tree pattern.

$\forall node\ in\ [0, N)$:
InverseBinaryTree $(node, S)$:
  for t in $[\ 0, \log_2 N\ )$
    if $(node \bmod 2^{\log_2 N - t}) == 0$ *then*
      Computation()
      Send_to$(node - 2^{\log_2 N-(t+1)}, length(t, S))$
    elsif $(node \bmod 2^{\log_2 N-(t+1)}) == 0$
      Wait_from$(node + 2^{\log_2 N-(t+1)}, length(t, S))$
    endif
  endfor

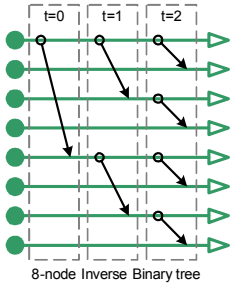


Fig. 2. Algorithmic and graphical definitions of Inverse Binary Tree pattern.

$\forall node\ in\ [0, N)$:
Butterfly $(node, S)$:
  for t in $[\ 0, \log_2 N\ )$
    Computation()
    if $(\lfloor node / 2^t \rfloor \bmod 2) == 0$ *then*
      Send_to$(node + 2^t, length(t, S))$
      Wait_from$(node + 2^t, length(t, S))$
    else
      Send_to$(node - 2^t, length(t, S))$
      Wait_from$(node - 2^t, length(t, S))$
    endif
  endfor

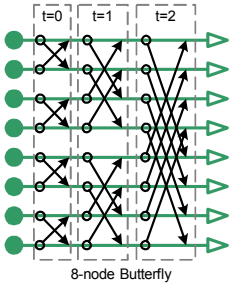


Fig. 3. Algorithmic and graphical definitions of Butterfly pattern.

MPI_Bcast() uses this pattern in some MPI implementations that rely on point to point operations to offer this functionality (often because there is not network-supported broadcast). The message length is fixed during the whole pattern, and the computation time is zero because there is no operation to perform with the received message; the node only has to send it to the next task, if necessary. MPI_Scatter() operation also uses **IBT** and the message length halves at each stage. In the case of MPI_Scatterv() the message length in an application-dependent way. Regarding the computation time, it depends on the architecture and the possibility to use DMA directly from the MPI library.

The reader should note that a usual mechanism to implement MPI_Barrier() is by concatenating a Binary Tree and an Inverse Binary Tree with message length 0. This way, when the **BT** finishes, the "root" knows that all tasks are synchronized, and starts an **IBT** to let the other tasks know that all of them have reached the barrier, *i.e.* they are synchronized.

3.1.3. Butterfly

The Butterfly pattern (**BU**) provides an efficient implementation of MPI N-to-N (*all-to-all*) collectives (MPI_Alltoall(), MPI_Allreduce(), etc.) [16]. It is also known as "recursive doubling", or "recursive halving" in the inverse butterfly case. **BU** is *O(N log N)* in number of messages and *O(log N)* in time. **BU** pattern starts with a message at each node, and ends when all the messages are received. Algorithmic and graphical definitions of this pattern are shown in Fig. 3.

Usual implementations of MPI_Alltoall() perform a Butterfly with constant-length messages. Time between stages is just the time to go through node's protocol stack (two times: up and down). MPI_Allreduce() is commonly performed using this pattern, with fixed-length messages, using some CPU time between pattern stages to perform the operation associated to the reduction. In MPI_Allgather() the message length doubles at each stage of the Butterfly. It happens similarly with MPI_Allgatherv(), in which the message length increases at each stage but in an *ad-hoc* fashion. Finally MPI_Allscatter() performs a Butterfly in which message length halves at each stage. Regarding MPI_Allscatterv(), the message length decreases each stage in an application-defined way. Again, the CPU interval depends on the system and the MPI library.

3.2. Virtual topologies

This branch of communication patterns reproduce the data interchanges performed in applications that rely on virtual topologies, such as the 2D meshes commonly used in matrix calculus. We have modeled these patterns in such a way that they are available for several dimensions (*D*). We do not discuss about the message length in these patterns, because it is constant (a chunk of the problem matrix). Also, the CPU intervals among stages within these patterns depend on the application and their matrix size. Note that a virtual topology is independent of the actual network topology, because it is just a way to arrange MPI processes.

3.2.1. Wave-front

The 2D and 3D Wave-front patterns (**2W** and **3W**) perform a diagonal sweep from the first node to the last one in MPI virtual square (or cubic) meshes. The simulation of these patterns starts with two (three for **3W**) messages in node 0, and ends with the finalization of the sweep at the last node of the network. These patterns do not impose a very heavy load on the network – note that there are only a few nodes injecting at once – but create some contention near the destination nodes, because they have to receive data from several neighbors. Regarding message distance, in **2W** it can take just two values: 1 and $\sqrt{N}$. In the case of **3W**, it can take three values: 1, $\sqrt[3]{N}$ and $\sqrt[3]{N^2}$. Spatial and causal patterns of Wave-front are defined algorithmically and graphically in Fig. 4.

We can observe this pattern in applications implementing the Symmetric Successive Over-Relaxation (SSOR) [3] algorithm, used to solve sparse, triangular linear systems. For example, application LU from the NPB suite [10] performs several consecutive bi-dimensional sweeps composed by short messages. We denominate the concatenation of **2W** Waterfall (**WF**).

3.2.2. Distribution

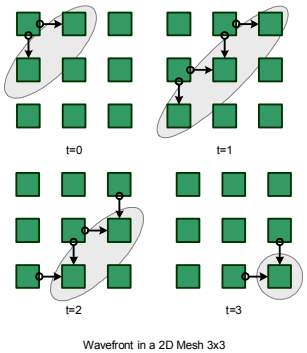
The 2D and 3D mesh patterns (**2M**, **3M**) perform data distributions in MPI virtual square (or cubic) meshes from every node to all its neighbors; after that, each node waits for the reception of all messages from its neighbors. Simulation starts with all nodes injecting one message per direction (2-4 for **2M**, 3-6 for **3M**), and ends when all the messages have arrived to their destinations. These patterns impose a very heavy load on the network, because all nodes inject simultaneously several messages at once before stopping to wait for the receptions. Regarding message distance, in **2M** it can take just two values: 1 and $\sqrt{N}$. In the case of **3M**, it can take values: 1, $\sqrt[3]{N}$ and $\sqrt[3]{N^2}$. Algorithmic and graphical definitions are shown in Fig. 5.

These patterns can be observed in scientific applications using finite difference methods [1]. In some of these applications, the spatial pattern also includes communication in the positive diagonal: each node communicates with the nodes located at ±1 in all dimensions. Furthermore, there are some patterns similar to these, but using virtual tori instead of virtual meshes, thus the nodes located in the boundaries of the virtual topology communicate between them.

3.2.3. Direction Distribution

The 2D and 3D distribution patterns (**2D**, **3D**) perform the same data distributions of **2M** and **3M** in virtual square (or cubic) meshes. However in these patterns data distributions are arranged in several steps, one for each direction. Simulation starts with all nodes injecting one message to their neighbors in the positive first dimension (X+). After that, each node wait for the message from its neighbor, and then sends it to the neighbor in the negative first dimension (X-), and so on for all the remaining directions. Simulation will end when all messages in the last direction (and obviously in all the other directions) have been delivered.

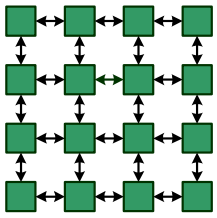
```
∀node in [0, N):
Wavefront (node, S, D):
  Computation()
  for d in [0, D)
    Wait_from(Neighbor(node, d, '−'), S)
  end for
  for d in [0, D)
    Send_to(Neighbor(node, d, '+'), S)
  end for
```



Wavefront in a 2D Mesh 3x3

Fig. 4. Algorithmic and graphical definitions of 2D Wave-Front pattern.

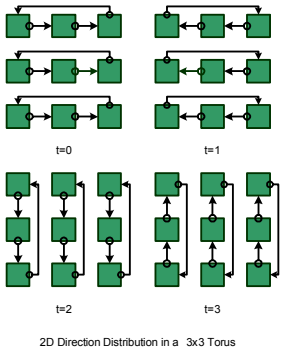
```
∀node in [0, N):
Distribution (node, S, D):
  Computation()
  for d in [0, D)
    for w in {'+', '−'}
      Send_to(Neighbor(node, d, w), S)
    end for
  end for
  for d in [0, D)
    for w in {'+', '−'}
      Wait_from(Neighbor(node, d, w), S)
    end for
  end for
```



Distribution in a 2D Mesh 4x4

Fig. 5. Algorithmic and graphical definitions of 2D Distribution pattern.

```
∀node in [0, N):
Direction_Distribution (node, S, D):
  Computation()
  for d in [0, D)
    for w in {'+', '−'}
      Send_to(Neighbor(node, d, w), S)
      Wait_from(Neighbor(node, d, w), S)
    end for
  end for
```



2D Direction Distribution in a 3x3 Torus

Fig. 6. Algorithmic and graphical definitions of 2D Dir. Dist. pattern.

These patterns impose a *not-so-heavy* load on the network, because all nodes inject simultaneously one message at once before stopping to wait for the receptions. Message distance distributions are the same of **2M** and **3M** patterns. The spatial and causal pattern is defined algorithmically and graphically in Fig. 6.

These patterns are also common in finite difference methods [1]. Some applications use some traffic patterns in which the distribution is arranged by dimensions, this is, first messages are interchanged in the first dimension (X+ and X-) and, when those interchanges have finished, the interchanges in the next dimension (Y+ and Y-) can start, and so on. Note that the same possible variants of the distribution pattern (diagonal and tori) could be applied to these patterns.

Just as a curiosity, the reader should note that formerly explained butterfly pattern could be seen as a dimension distribution in a (virtual) hypercube.

4. Evaluating network topologies using application-like workloads

As an example of the proposed methodology and also to show the temporal evolution of the load imposed in the networks by these workloads, we will perform a comparison of cube-like and tree-like topologies. In order to have a comparison yardstick, we will also study a perfect crossbar that would represent a best-case in network communications. Experiments will be carried out using simulation [14], measuring time in terms of (simulated) cycles: a *cycle* is the time required by a *phit* (a physical transfer unit, typically a few bytes) to traverse one switch.

4.1. Networks to compare

We will evaluate three small networks, all of them with a theoretical maximum throughput of 1 phit/cycle/node (limited by the bisection bandwidth, for random, uniform traffic), and built with the same networking technology. We will measure the time the networks need to deliver all the traffic generated by the workloads, and also the temporal evolution of the network throughput.

The first network under study is a 64-port crossbar. This is a particular network that is able to interconnect nodes in an unblocking, any-to-any fashion, that is, each node can send a message to any other node with just two hops, one from the source NIC (network interface card) to the crossbar and another one from the crossbar to the destination NIC. We assume a perfect crossbar, able to manage up to the number of ports (in this case 64) messages at once, given that all those messages come from different sources and do not compete for the destination ports. In other words, when bottlenecks appear, they are caused by contention at injection or consumption ports, so this network will show us the ideal execution time for the proposed traffic patterns. This is the reason we use its performance as the yardstick to compare against the other networks.

The second network is a 2-ary 6-tree built with 4-port routers, two of the ports are upwards and another two downwards. Note that this is not currently a common network topology, because today's routers have a noticeably higher radix. However, it is valid to show how some of the proposed workloads are "fat-tree-friendly", that is, execution behavior and temporal evolution of the network under these are close to that observed with the 64-port crossbar.

The last network is an 8-ary 2-cube (8x8 torus) built using 5-port routers, 1 port to communicate with the local node, and the other four connected to neighbors at directions X+, X-, Y+ and Y-. This topology is a reduced version of those used in current MPPs such as BlueGene [4] and RedStorm [5].

4.2. Model of the components

We model the node as a traffic generation source with one injection queue, which is able to store up to 8 packets. It is also the sink to the arriving messages. When generating traffic, we consider reactive sources, meaning that the reception of a message may trigger the release of a new one. This way we can model the causality inherent to actual applications traffic.

We have chosen simple input-buffered switches whose radixes are 4, 5 and 64, depending on the topology of choice. Four virtual channels share each physical channel. The arbitration of each output port is performed in a random way, that is, when several input ports request the same output port, one of them is chosen at random. Transit queues are located at the input ports, and are able to store 4 packets each. There is a schematic model of the switch in Fig. 7.

Messages are split into packets of a fixed size of 16 phits. One phit is the smallest transmission unit, fixed to 32 bits. If a message does not fit exactly in an inte-

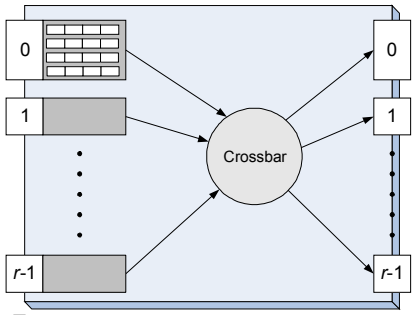


Fig. 7. Model of the switch (radix *r)*. Four virtual channels sharing port 0 are shown.

gral number of packets, the last packet contains unused phits. The switching strategy is virtual cut-through.

Routing in fat-trees is, when possible, adaptive using minimal paths. A credit-based flow-control mechanism is used, so that when several output ports are valid to reach the destination, the port with most available credits is requested. Credits are communicated *out-of-band*, so they do not interfere with regular traffic. The torus network uses one *escape* channel with DOR static routing and bubble flow control [12] to avoid deadlock situations. The other three VCs are fully adaptive using minimal paths. Obviously the crossbar routing algorithm is static because there is only a way to go from a source to a destination.

4.3. Workloads

To simplify figures and discussion, we will use in the experiments a selection of the workloads described before. Those will be **BT**, **BU**, **2M**, **3M**, **2W** and **3W**. These include a mixture of *heavy* (**BU**, **2M**, **3M**) and *light* (**BT**, **2W**, **3W**) traffic patterns, and also patterns with different spatial characteristics: *binary* (**BT**, **BU**), 2D *square*-like (**2M**, **2W**) and 3D *cube*-like (**3M**, **3W**).

In order to keep things simple, inter-stage computation times will be ignored and message length will be constant along each execution. Experiments will be repeated with three different message lengths (640, 3200 and 64000 bytes) to simulate different problem sizes. Note that in this evaluation there is an identity bijection between pattern tasks and system nodes, in other words, tasks are placed consecutively (task *i* runs at node *i*) and there is only one task into each node.

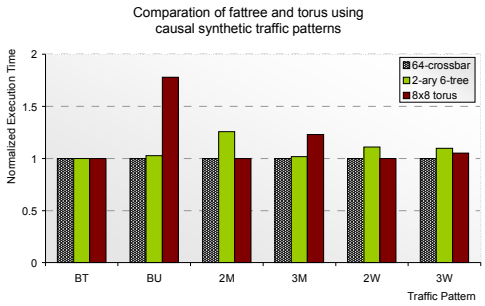


Fig. 8 Times to deliver all the messages for each topology and traffic pattern (*normalized* to crossbar)

4.4. Experiments and analysis of results

Results of the experiments for the longest messages are presented in Fig. 8 and 9. Results for the other message lengths are similar, and will not be shown for the sake of simplicity. As each workload requires a different running time, plotted times are normalized to the best case (crossbar labeled), so that plots are easier to understand. In Fig. 8, the fat-tree topology exhibits a performance close to the optimal obtained by the crossbar in all cases, but **2M**. In contrast, the torus topology runs into problems when managing heavy traffic patterns that do not match the network topology (**BU** and **3M**).

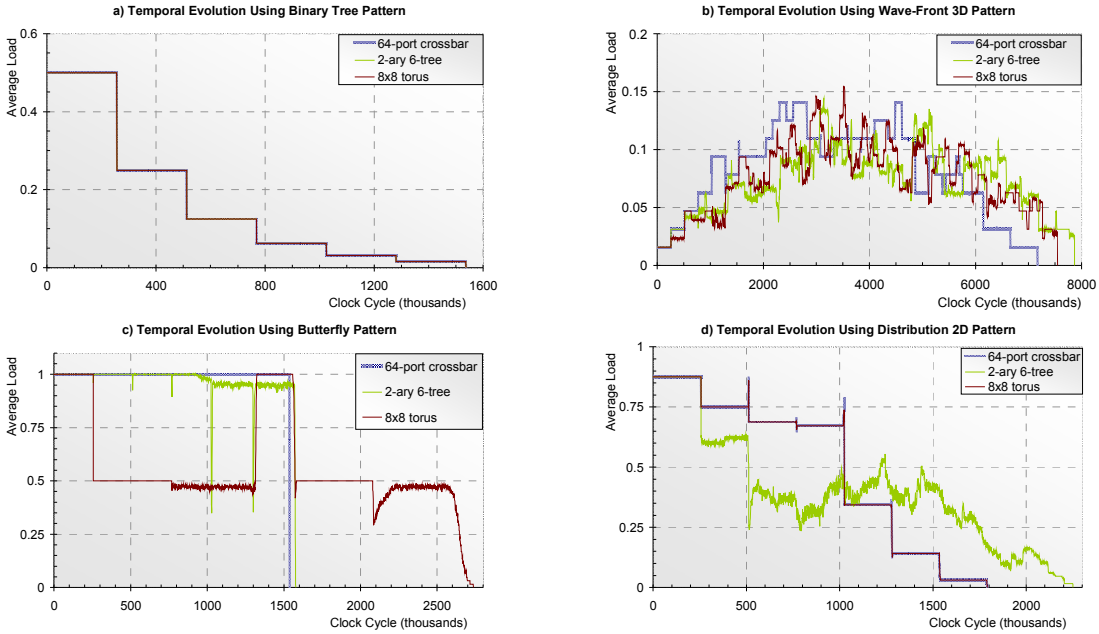


Fig. 9. Temporal evolution of the average consumed load measured in phit/cycle/node for different traffic patterns: a) **BT** pattern. b) **3W** pattern. c) **BU** pattern. d) **2M** pattern.

The temporal evolution of the consumed load (*phit/cycle/node*) for some of the workloads is plotted in Fig. 9. The broad blue line (crossbar) shows the communication needs of the different workloads, and the way they evolve with time. When those needs are light and the paths of messages do not overlap (this depends on the underlying topology) the networks are capable to deliver the workload without significant latency; Fig. 9a and Fig. 9b show two of these cases. In contrast, if communication needs are more intense and paths overlap, networks have some trouble arbitrating resources, with the resulting increase in latency, as can be seen in Fig. 9c and Fig. 9d. In the former, the spatial pattern of **BU** adapts better to the characteristics of the fat-tree topology; however, when the network is a torus, most of the messages overlap, reaching only one half of the peak performance. The latter (**2D**) shows how the mapping of a 2D mesh on a 2D torus is optimal, but when allocating the mesh over a fat-tree the network is not able to deliver the workload at maximum speed, thus communication time increases in a 25%.

Reader should note that Fig. 9b – corresponding to **3W**, one of the most complex patterns – reveals how the imposed load varies with time, and how the evaluated networks deal with it. The three networks deliver the workload in similar time, but their temporal evolutions are completely different.

5. Conclusions and future work

In this paper we have discussed methodologies to evaluate high-performance parallel systems, focusing on the workloads used in these evaluations. Workloads can be purely synthetic or based on actual applications. Also, they can use causal or independent traffic sources. We have described the pros and cons of generating and using these workloads. Furthermore, we propose new workloads that, although synthetically generated, emulate pieces of actual applications.

We have characterized and justified several application-like pseudo-synthetic workloads that mimic applications behavior with high levels of fidelity. They are organized in two main groups. The first group includes emulations of message interchanges aimed to implement collective operations in an efficient way. The second group includes emulations of the way scientific applications that make use of huge matrices communicate, taking advantage of virtual topologies.

As an example of how these synthetic workloads can be used in performance-related studies, we have done a comparison of three different network topologies: a crossbar, a fat-tree and a torus. This way we have shown that there are some pieces of the applications that are topology-friendly. Furthermore we have shown the temporal evolution of the networks under these workloads in order to show how the communication requirements of the applications change with time.

As future work we intend to increase the library of communication micro-kernels, in order to be able to emulate more applications. We plan to focus our attention on the "*13 dwarves*" [2], a collection of classes of applications representative of those actually running on high-performance computing sites. This enhanced library will be used for different performance-related studies carried out in our research groups, including comparisons of topologies, fault-tolerance strategies, routing mechanisms, etc.

References

[1] Y. Aoyama, J. Nakano. "RS/6000 SP: Practical MPI Programming". IBM Red Books SG24-5380-00, 1999.

[2] K. Asanovic et al. "The Landscape of Parallel Computing Research: A View from Berkeley". EECS Department. University of California, Berkeley. TR UCB/EECS-2006-183.

[3] E. Barszcz et al., "Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors", NAS RNR-93-007, NASA Ames Research Center, April 1993.

[5] M. Blumrich, et al. "Design and Analysis of the BlueGene/L Torus Interconnection Network" IBM Research Report RC23025 Dec. 2003.

[6] W.J. Camp, J.L. Tomkins: "Thor's hammer: The first version of the Red Storm MPP architecture. " In Proc. of the SC 2002 Conference, Baltimore, MD (2002)

[7] WJ Dally, B Towles. "Principles and Practices of Interconnection Networks". Chapter 24, Morgan-Kaufmann,2004.

[8] J.J. Dongarra, H.W. Meuer, E. Strohmaier. "Top500 Supercomputer sites". Available at: http://www.top500.org/

[9] S. Labour, "MPICH-G2 Collective Operations, Performance Evaluation, optimizations", available at http://www-unix.mcs.anl.gov/~lacour/argonne2001/report.ps

[10] J Navaridas, FJ Ridruejo, J Miguel-Alonso. "Evaluation of Interconnection Networks Using Full-System Simulators: Lessons Learned". Proc. 40th Annual Simulation Symposium, Norfolk, VA, March 26-28, 2007.

[11] NASA Advanced Supercomputing (NAS) division. "NAS Parallel Benchmarks".

[13] F Petrini et al. "Performance Evaluation of the Quadrics Interconnection Network". In the Journal of Cluster Computing, 6(2):125-142, April 2003.

[14] V Puente, et al, "The Adaptive Bubble router", Journal on Parallel and Distributed Computing, vol 61, Sept. 2001.

[15] F.J. Ridruejo et al. "TrGen: a Traffic Generation System for Interconnection Network Simulators" (PEN-PCGCS'05). ICPP 2005 Workshops. 14-17 June

[16] F.J. Ridruejo, J. Miguel-Alonso. "INSEE: an Interconnection Network Simulation and Evaluation Environment". Lecture Notes in Computer Science, Volume 3648 / 2005.

[17] F. J. Ridruejo, et al, "Realistic Evaluation of Interconnection Network Performance", PDCAT 2007, December 3-6

[18] R. Thakur and W. Gropp, "Improving the Performance of Collective Operations in MPICH", available at http://www-unix.mcs.anl.gov/~thakur/papers/mpi-coll.pd

Chapter 12. Case study 1: Congestion control in direct networks

Full reference:
F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso, C. Izu.
Realistic Evaluation of Interconnection Network Performance at High Loads
8th Int. Conf. on Parallel and Distributed Computing Applications and Technologies - PDCAT 2007, Adelaide, Australia, 3-6 December 2007

As our research interest was focused on large multicomputers, we noticed difficulties to obtain large multicomputer traces, and the impossibility of full-system simulating them, so we considered other synthetic traffic models that mimicked the behavior of barrier-based applications; i.e. bursty traffic generation. We show in this paper that using carefully chosen synthetic traffic (as bursty traffic) can be accurate enough to validate some design decisions.

We validate bursty traffic comparing its results to trace-based traffic generation on a real study on local congestion control mechanisms on large radix direct networks with multiple injectors. These networks are prone to congestion but congestion control mechanisms limit throughput degradation when traffic is beyond the saturation point. This phenomenon was unknown and, added to other research works such as [MIG08], it proves the necessity of congestion control on large radix IN.

The author of this thesis was responsible for the implementation of the bursty traffic model within TrGen, all simulation experiments and their comparison, and extracting conclusions for this work.

Realistic Evaluation of Interconnection Network Performance at High Loads

F.J. Ridruejo[1], J. Navaridas[1], J. Miguel-Alonso[1], Cruz Izu[2]
[1] *Dep. of Computer Architecture and Technology, UPV/EHU, Spain*
[2] *School of Computer Science, The University of Adelaide, Australia*
{franciscojavier.ridruejo, javier-navaridas, j.miguel}@ehu.es, cruz@cs.adelaide.edu.au

Abstract

Any simulation-based evaluation of an interconnection network proposal requires a good characterization of the workload. Synthetic traffic patterns based on independent traffic sources are commonly used to measure performance in terms of average latency and peak throughput. As they do not capture the level of self-throttling that occurs in most parallel applications, they can produce inaccurate throughput estimates at high loads. Thus, workloads that resemble the varying levels of synchronization of actual applications are needed to study the performance of interconnection networks. One approach is to use simple, burst-synchronized synthetic workloads that emulate the self-throttling of many parallel applications. To validate this approach, we compare the gains achieved by a restrictive injection mechanism under this workload with those obtained using traces from the NAS Parallel Benchmarks. This study confirms that the burst-synchronized traffic model provides reasonable performance estimates, which could be improved by taking into account dependency chains between messages.

1. Introduction

The interconnection network (IN) is a key element of any parallel computer, more so when executing communication-intensive applications. Network performance has been evaluated using many techniques, including: network simulation with synthetic loads, analytical models, trace-driven simulation and full-system simulation. Traditionally, network simulators measure performance under a well-known range of synthetic traffic patterns such as uniform, hot-spot and permutations. These patterns model worst-case scenarios of little locality and unbalanced usage of network resources [4]. Analytical models have been proposed for simple networks but they rely on unrealistic assumptions, such as uniform traffic or infinite injection and delivery queues, so they have limited use. A full system simulation, in which traffic is provided by an execution-driven simulator such as Simics [16], provides more meaningful results but limits the evaluation to small networks.

Congestion is a well-known problem in standard computer networks [8], but most INs in the literature (most with a standard size of 256 nodes) did not exhibit throughput degradation at loads beyond saturation. Such network proposals had single injection queues and, when the network is small, the head-of-line blocking at injection is enough to throttle the network and control congestion [7]. This is not the case for large networks with multiple injection sources such as IBM's BlueGene/L [2], which are prone to suffer from congestion at heavy loads. Consequently, new congestion control techniques have been proposed and evaluated for wormhole [1,15] and virtual cut-through INs [9,11,14]. Most of these works carry out evaluations using synthetic traffic patterns that assume that nodes generate traffic independently of each other. Although they ignore the different levels of coupling and synchronization that exist in parallel applications, synthetic loads seemed to provide reasonable indicators of network performance for a range of parallel benchmarks [12]. Note that most parallel applications will apply some level of self-throttling as nodes synchronize and may stop sending new messages as they wait for messages delayed by congestion. Thus, any evaluation of an IN at heavy loads, and in particular any evaluation of a congestion control technique, should be done under loads that reflect the synchronization and coupling among application processes.

In [6] burst-synchronized synthetic traffic was proposed to make a fair evaluation of the congestion control technique IPR (In-transit Priority Restriction), the one used in the torus network of the IBM BlueGene/L [2]. In this paper we will consider another congestion control technique called Local Buffer

0-7695-3049-4/07 $25.00 © 2007 IEEE
DOI 10.1109/.73

Restriction (LBR), which uses local information to regulate the admission of new traffic. We will compare results obtained using burst-synchronized loads with those obtained by applying actual loads taken from traces of the NAS Parallel Benchmarks (NPB). If the synthetic loads reflect the synchronized nature of parallel applications, both results should lead to the same conclusions regarding LBR, and we would have a means to evaluate very large systems. Note the main goal of the work is not to evaluate LBR but to find out if burst-synchronized loads are a good approximation of real loads.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the experimental, simulation-based workbench. Section 4 compares and contrasts the results of the different experiments. Finally, Section 5 summarizes the findings of this work.

2. Motivation and Related Work

Most IN studies rely solely on synthetic traffic patterns, which include uniform traffic, hot spot traffic and permutations [4]. The figures of merit are latency at low loads and peak network throughput. The need to characterize network workload and produce better synthetic models was identified long time ago [3], although there has been little progress since. Instead of developing new synthetic loads, some IN studies combine the standard evaluation with real workload evaluation [12]. Other studies use synthetic loads that mimic the bursty nature of network traffic [14] extending the standard packet generation, which followed a Poisson or Bernoulli distribution, with a sequence of ON/OFF states, so that packets are generated only during the ON state. Besides, most studies normalized the applied load to the network bisection limit, so that the networks were not tested for loads beyond their theoretical capacity. As most network proposals sustained their maximum throughput after saturation, the evaluation of the network at heavy loads was not considered of interest until recently. However, congestion is a problem for large INs with multiple injection sources, and congestion control mechanisms are tested at loads beyond saturation [1,9,14] in order to see if they prevent traffic bursts from degrading network performance.

Synthetic loads used to evaluate congestion control techniques fail into two categories: static loads, which use the same pattern and injection rate over the time each experiment runs, and dynamic loads which alternate between phases of high and low injection rates. For static loads, only the steady-state performance is studied, and the figure of merit is average sustained throughput. For dynamic loads the figure of merit is total execution time, which is a better indication of the network ability to cope with communication intensive phases.

The evaluation of congestion control techniques using static loads has shown that unbalanced traffic loads may result in network unfairness at saturation [6]: some nodes have less chance to inject traffic than others. The source of this unfairness is the unbalanced utilization of resources, derived from the traffic pattern. The more in-transit packets a router has to manage, the less opportunity it has to inject its own traffic. Besides, an unbalanced load distribution results in the formation of persistent zones of congestion at high loads. In this context, average throughput as reported in [1,8,15] is not representative of application loads, because “fast” injecting nodes will eventually wait for the slow ones to catch up. Therefore, in order to obtain useful performance figures at heavy loads, a static synthetic traffic pattern should reproduce the level of coupling that exists amongst traffic sources in actual parallel applications.

Non-static loads reflect the fact that communication in many parallel applications is not constant over time, so that an intensive communication phase (with high traffic volume) will be followed by a computation intensive phase (with low traffic volume), Recent studies using non-static traffic patterns include [1,14] in which traffic is uniform but load alternates between low and high phases, and [15] in which each high phase uses a different pattern. As mentioned before, the figure of merit is total execution time. As the high phases will exhibit network unfairness, a computing node located in a less clogged area will be able to advance to the next phase ahead of the rest, an unlikely scenario in a parallel application. In other words, although non-static synthetic loads reflect the temporal variations that occur in application loads, they still fail to model the synchronization and coupling amongst application nodes.

Burst-synchronized traffic deals with this issue by modeling the barrier synchronization primitives used in many parallel applications, either explicitly (in the form of collective operations) or implicitly. This synchronized traffic has been sparingly used in studies focused on injection issues [3,6], claiming that it represents realistic workloads, but there is no formal study confirming or denying this fact. Our work tries to fill this gap by comparing the insights obtained from synthetic loads with those from actual workloads taken from application traces.

3. Experimental Setup

3.1 The Simulation Environment

Experiments have been performed using the evaluation environment described in [13]. It consists of an IN simulator and a traffic-generation module, which provides traffic from one of these sources: synthetic generation, traffic as recorded in traces or interfacing with Simics [16] to perform a full system simulation [11].

We perform most of our experiments using a small network of only 64 nodes. This is because the trace-capture setup imposes us limits that are close to this value (due to availability of resources in a production machine, and to the sizes of the trace files). As bisection bandwidth does not increase linearly with the radix, networks with large radix reach saturation at lower injection rates, and they are more likely to suffer overloads during intensive communication phases. Thus we focus our study on 64-node rings, instead of using 8x8 tori. The ring is adequate to experiment with congestion, and results with this topology can be extended to multidimensional Ins, because congestion causes messages to rely on the escape sub-network, in which they must traverse the x-ring first.

The models of routers used in the experiments are depicted in Fig. 1. Each physical channel in the router is split into three virtual channels (VCs): an Escape channel (governed by the bubble routing rules [12]), and two adaptive channels. Note that a ring network has just one minimal path from source to destination, so packets cannot adapt. Thus, the only difference between the Escape VC and the other two is that access to the "adaptive" VCs is not restricted by the bubble rules. In the case of 2D tori, packets in adaptive VCs can use any minimal path to reach their destinations.

Each node is able to simultaneously consume several packets arriving to the reception port. There are two injection ports, and the interface should perform a pre-routing decision: packets moving towards the X+ axis are stored in the I+ injection port, and those towards X– go to the I– injection port. Transit and injection queues are able to store 4 packets of 16 phits each. Phit length is 4 bytes (32 bits).

Each experiment has been repeated 10 times, using different random seeds. We measure execution times in terms of simulation "cycles". As these times vary between patterns or benchmarks we have represented the average value (obtained from 10 simulation runs) normalized to the Base case.

Regarding synthetic traffic we use UN (uniform) and HR (hot-region). In both cases destinations are chosen randomly; in the case of HR [2], 1/4 of the traffic goes to the first 1/8 nodes, and the remaining traffic is uniform.

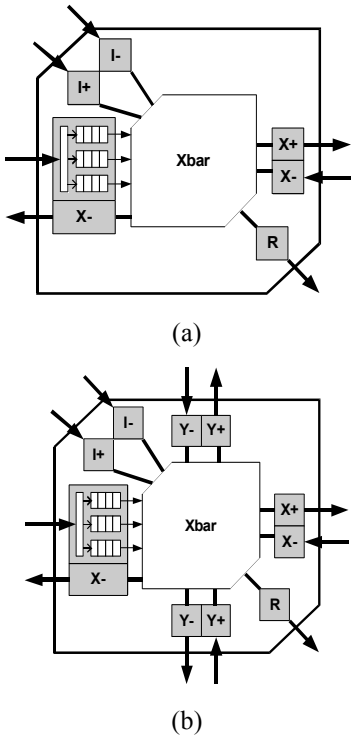


Fig. 1. Models of the simulated routers, (a) router for rings, and (b) router for 2D tori, with a detailed view of the X+ input port showing the 3 virtual channels that share its link

Traces used in this work have been obtained from clusters of commodity PCs and from the Mare Nostrum, running some of the NAS Parallel Benchmarks (NPB) [10]. To obtain these traces we modified the trace capture tool included in MPICH, in order to register all the point-to-point operations—including those that implement the collective operations [13]. As we want the network to become the bottleneck, we provided workload to the simulator as fast as we can, regardless of the timestamps found in the traces. However, in order to preserve the causal relationship among messages, we maintained the order shown between arrivals and new injections: when the trace file indicated that a message was received at a given node, injection of packets from that node sent later in the trace is delayed until that message arrives. This way, we measure the number of network cycles that, in the simulated network, are necessary to complete all the interchanges of messages stored in the trace files.

3.2 Congestion Control Techniques

As congestion is caused by overloading the network with too many packets, congestion control techniques deal with it by limiting packet injection as soon as the network exhibits signs of being congested. They differ in the way congestion is diagnosed.

Global methods estimate congestion by examining the status of the whole network (for example, the number of packets held in the routers, as in [15]); thus a mechanism is needed to gather and distribute that information. Local methods are simpler because each node restricts its own injection based on its own congestion level. Multiple congestion control methods are evaluated in [8]. As this is not a work on congestion control, we consider only one simple local method that has shown good performance, LBR.

Most routers split each physical link into several VCs in such a way that the combination of an escape sub-network with one or more adaptive sub-networks provides deadlock-free adaptive routing [5]. The LBR mechanism has been designed specifically for adaptive routers that rely on Bubble Flow Control to avoid deadlock in the escape sub-network [12]. A previous study showed how the bubble restriction also provides congestion control for the escape sub-network [7]. LBR extends this mechanism to the rest of the VCs. That is, a packet can only be injected into an adaptive VC if such action leaves room for at least B packets in the transit buffer associated to that VC. Parameter B indicates the buffer space reserved for in-transit traffic. In other words, congestion is estimated by the current buffer occupancy. We can vary the degree of restriction in the injection by modifying this parameter. In this work we use B=3 (out of 4-packet buffer), so packets are injected in an adaptive channel only when its queue is empty or almost empty.

4 Analysis of the Experiments

4.1 Experiments using Burst-Synchronized Traffic

The utilization of burst-synchronized traffic to model application loads was proposed in [6]. With this traffic, each node tries to inject a burst of b packets as fast as the network is able to accept them; then the node stops. Nodes will start injecting another burst only when all the packets of the previous one have been consumed. The figure of merit in these experiments is the time to consume a burst. We have considered a collection of values for b, trying to emulate different degrees of coupling among application processes, from 10 (the most tightly-coupled) to 10.000 (the most loosely-coupled). For applications that synchronize using barriers, we can interpret b to be the number of packets sent between two barriers. Note that, in a VCT network, long messages are packetized, so a long message generates a burst of packets to be injected in the network. Therefore, message size and b are directly related.

Execution times have been normalized to the time of the Base case (LBR deactivated). The time relation between base case and LBR is plotted in Fig. 2 for both random traffic and hot-region traffic. We can see that LBR is effective in reducing the time the network requires to deliver one burst. As expected, the gains increase with the number of packets sent per burst. LBR is more effective in the 2D network, as adaptive routing increases pressure on network resources and causes higher contention than in the 64-ring counterpart. Thus, the evaluation of LBR using a 64-ring give us a conservative estimate of the gains achieved by LBR on multidimensional network with similar radix.

4.2 Experiments Using Application Traces

We now explore the relationship between b and the problem size of the NPB. This suite can be compiled and run for a variety of problem sizes, denoted S, W, A, B, C and D—where S is the smallest and D is the largest. Larger problems use larger data structures, and this entails interchanges of larger messages, although the pattern of interchanges remains the same. There is, though, an exception to this rule: for LU, as the problem size increases, the number of messages interchanged also increases, in addition to its size.

For a second set of experiments, we have generated traces for a variety of problem sizes for all the NPB, and processed them through the network simulator. We have used the official sizes defined for the applications, and added some intermediate cases when necessary (for example, the difference in problem size from S to W for IS is too wide, so we have added those cases denoted as “T”). Figures 3(a) and 3(b) show the results obtained for IS and CG respectively. Compare with the curves in Fig. 2 for the synthetic patterns UN and HR. Again, as the problem grows, the benefits of LBR are more visible. Although congestion control can be counterproductive in some cases (CG), its penalty is small.

4.3 Analysis of Application Traces

Results from previous sections predict that LBR is clearly beneficial in some applications, but may have a negative impact in others.

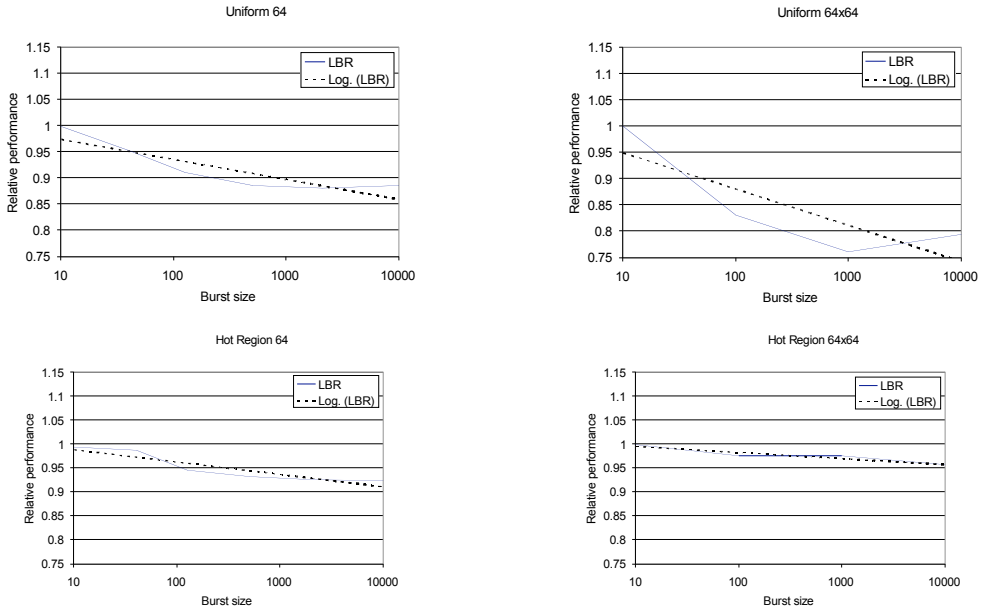
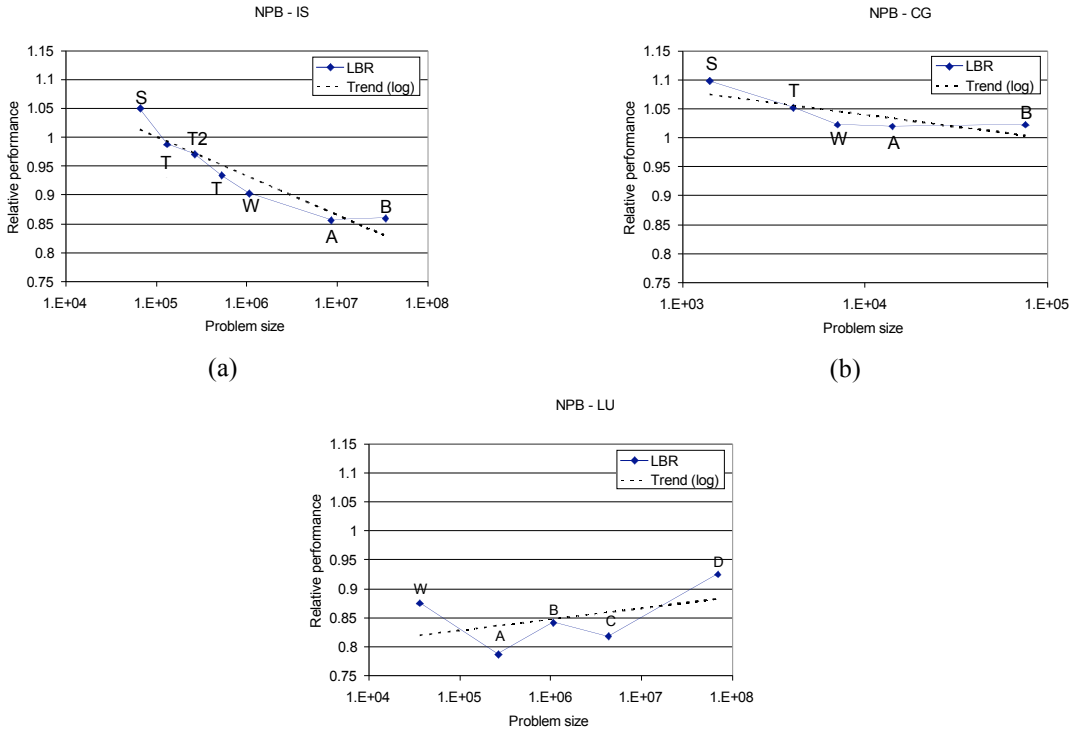


Fig. 2. Performance of 1D and 2D tori dealing with bursts of uniform and hot-region traffic. Normalized times to consume a burst, and trend lines (X axis is logarithmic).



(a) (b)

(c)

Fig. 3. Relationship between problem size and efficiency of LBR for benchmarks IS (a), CG (b) and LU (c). Measured values and trend lines. Scale in the X axis is logarithmic. Y axis represents the time to consume the applied workload, relative to the base case without LBR.

We have related this to the problem size and to the burst size b in our synthetic model, but we have also observed that this approach is not valid for all cases. In this section we analyze the traces of CG and LU (size A, 64 processors), obtained in the Mare Nostrum, to better understand the relationship between the traffic patterns and the effect of congestion control.

CG is the only application of the NPB suite that does not benefit from congestion control—just the opposite. A visualization of the trace of CG.A.64 (see Fig. 4) shows that this application consists of a series of iterations, each of them with these phases:

(1) An interchange of messages: each node sends and receives 7 messages. The first 4 are long (~14 KB) and the remaining 3 are very short (8 B). The first two long messages go to nodes nearby, while the remaining two must traverse longer distances. The first short message goes to a distant node, while the remaining two go to nearby nodes.
(2) A short computation phase.
(3) A second interchange of three very short messages, between nearby nodes.
(4) A longer computation phase.

In short, this benchmark exhibits long communication-synchronization chains: a message is sent only when triggered by the reception of another one. These chains are of length 7 in phase 1, and of length 3 in phase 3. The number of messages traversing the network simultaneously is never very large, and in most cases they go to close destinations. Any delay injecting a given message (for example, to prioritize in-transit traffic, as LBR does) results in additional delays injecting messages than depend on it. We can expect to obtain maximum benefit from congestion control in communication-intensive phases with a mixture of short-distance and long-distance packets, without interdependencies. CG does not fulfill these requirements, so the observed performance drop.

LU is the application that benefits most from congestion control. However, we expected more improvements for larger problem sizes, and this does not happen. The traces help us understanding the reasons. A study of the trace of LU.A.64 shows that this application also consists of a series of iterations, each of which has 7 phases:

(1) A cascade of short, chained messages (600 B) initiated at node #0, flowing downward to the remaining nodes; messages go to destinations at distance 1 or 8.
(2) A similar, upward cascade, started at node #63.
(3) A computation phase.
(4) An interchange of a long message (~40 KB) with a neighbor.
(5) A computation phase.
(6) Another interchange of a long message, with a node at distance 8.
(7) A computation phase.

Fig. 5 illustrates the 7 phases for LU.A.16. The size was reduced to 16 for the sake of clarity, but the patterns are similar for LU.A.64. Phases 1 and 2 are very demanding in terms of network utilization, and they take a sizeable portion of the total running time. In those phases there are dependency chains, of length 7, among messages. All messages in each chain go to distance 1 or 8. As LBR prioritizes the in-transit traffic, messages to distance 1 are injected after those going to distance 8 have passed trough, thus the total time for these phases is increased greatly. It is important to remark that when the problem size increases, both the message size and the number of chains in the cascades increase.

In phase 4 only neighbor-to-neighbor links are used, so network routers do not observe passing-by traffic. In this phase, dependency chains are of length 2. Phase 6 is similar to 4, having chains of length 2, but message destinations are at distance 8. This results in conflicts between in-transit traffic and injected traffic.

Congestion control techniques are very effective in accelerating phase 6; in fact, a micro-benchmark that reproduces only this phase reports gains using LBR over a 40%. However, they are harmful in phases 1 and 2—a micro-benchmark for this phase reports drops of a 20%. When the problem size of LU is increased, phase 6 does not experiment further acceleration, but phases 1 and 2 are longer in number of messages and, thus, the negative impact of congestion control is more noticeable. This explains the results of Fig. 3c, showing performance decreases for larger problem sizes.

The conclusion of this section is that a simple, burst-synchronized traffic model may not adequately describe the characteristics of any possible application, but can do so for some phases of the application. We need to improve the model to include the dependency chains that are present in actual applications and that interact negatively with congestion control mechanisms.

5 Conclusions

Synthetic workloads are useful during the initial design stages of an IN, as they allow exploring the network design space and providing initial performance estimates. For large networks, the synthetic model should take into account, at least, the self- throttling that real applications have implicit.

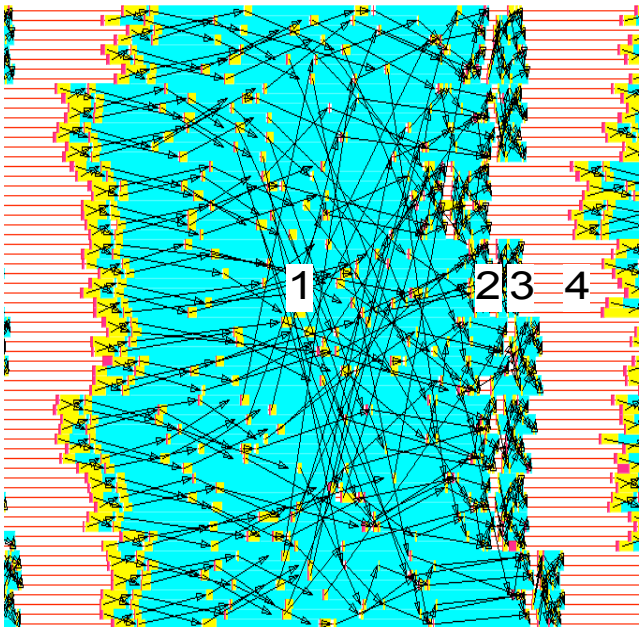


Fig. 4. Visualization of a portion of the traces for CG.A.64 (a): a yellow block (state) in the timeline represents a "send", pink means "receive", and cyan means "wait". The remaining is computation time. Arrows represent messages.

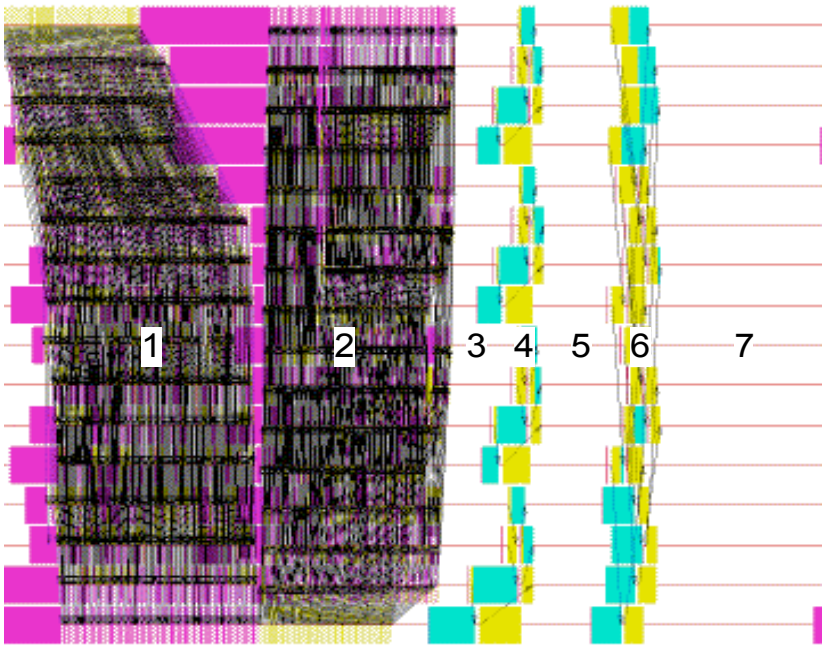


Fig. 5. Visualization of a portion of the LU.A.16 trace: in the last phase of data interchanges messages are sent to distance 4, whilst in the LU.A.64 this distance is 8. (yellow block (state) represents a "send", pink means "receive", and cyan means "wait". The remaining is computation time.

We propose the utilization of burst-synchronized traffic that, although still simple, models application behavior better than the traffic generated by independent sources. Scientific applications advance in alternating phases of computation and communication-synchronization. In our model, a burst represents a phase of intensive packet interchanges followed by synchronization primitives such as a barrier. The burst size b indicates the number of packets sent by each node in each phase.

We have used this traffic to evaluate a local congestion control mechanism (Local Buffer Restriction), after showing that the utilization of independent sources may provide misleading results. Using burst-synchronized traffic, LRB shows its potential to accelerate message interchange in a VCT network. This benefit is larger for larger values of b. In order to validate this result, we have performed additional experiments using real traffic, taken from traces of the NPB (class A, 64 nodes). For most of the experiments, congestion control shows its good performance, confirming our findings. Furthermore, the class (problem size) of the NPB is directly related to b and, again, larger problems benefit more from congestion control.

There are some exceptions, though, to this rule. For some applications, congestion control is counterproductive and, for some others, obtained results are not as good as we could expect, especially for large problems. This is because the applications, or some phases within them, have long chains of dependencies between messages. This behavior is not adequately characterized by burst-synchronized traffic, which models all traffic interchanges inside a phase as independent communication events.

In summary, this study has proven that the utilization of burst-synchronized traffic is a reasonable, although not perfect, alternative to the utilization of actual traffic, and can help in the evaluation of large networks for which the use of real traffic loads is not viable. Future lines of work include the introduction in this model of some sort of "reactiveness" to describe message chains, and also a method to find the best value of b that characterizes a given application.

References

[1] E. Baydal, P. Lopez and J. Duato, "A Family of Mechanisms for Congestion Control in Wormhole Networks" IEEE Trans. on Parallel and Distributed Systems, V. 16, N. 9, Sept. 2005, pp 772-784.

[2] M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-Burrow, T. Takken, P. Vranas. "Design and Analysis of the BlueGene/L Torus Interconnection Network" IBM Research Report RC23025 Dec. 2003.

[3] T. Callahan and S.C. Goldstein, "NIFDY: A Low Overhead, High Throughput Network Interface", in Proc. 22nd Annual Int. Symp. on Computer Architecture (ISCA), Italy, June 1995.

[4] W.J. Dally, B. Towles. "Principles and Practices of Interconnection Networks". Morgan-Kaufmann, 2004.

[5] J. Duato. "A Necessary and Sufficient Condition for Deadlock-Free Routing in Cut-Through and Store-and-Forward Networks". IEEE Trans. on Parallel and Distributed Systems, v. 7, n. 8, 1996.

[6] C. Izu, J. Miguel-Alonso, J.A. Gregorio. "Evaluation of Interconnection Network Performance under Heavy Non-uniform Loads". Lecture Notes in Computer Science, Volume 3719 / 2005 (Proc. ICA3PP 2005), pp. 396 - 405.

[7] C. Izu, J. Miguel-Alonso, J.A. Gregorio. "Effects of Injection Pressure on Network Throughput", in Proc. PDP 2006 14th Euromicro Conference on Parallel, Distributed and Network based Processing. Montbéliard-Sochaux - France-February 15-17 2006.

[8] R. Jain. "Congestion control in computer networks: issues and trends". IEEE Network, v.4 n.3, May 1990.

[9] J. Miguel-Alonso, C. Izu, J.A. Gregorio. "Improving the Performance of Large Interconnection Networks using Congestion-Control Mechanisms". EHU-KAT-IK-06-05. Dep. of Computer Architecture and Technology, UPV/EHU. Submitted.

[10] NASA Advanced Supercomputing (NAS) division. "NAS Parallel Benchmarks" Available (May 2006) at http://www.nas.nasa.gov/Resources/Software/npb.html

[11] J. Navaridas, F.J. Ridruejo, J. Miguel-Alonso. "Evaluation of Interconnection Networks Using Full-System Simulators: Lessons Learned". Proc. 40th Annual Simulation Symposium, Norfolk, VA, 2007.

[12] V. Puente, C. Izu, J.A. Gregorio, R. Beivide, and F. Vallejo, "The Adaptive Bubble router", Journal on Parallel and Distributed Computing, vol 61, no. 9, Sept. 2001.

[13] F.J. Ridruejo, J. Miguel-Alonso. "INSEE: an Interconnection Network Simulation and Evaluation Environment". Lecture Notes in Computer Science, Volume 3648 / 2005 (Proc. Euro-Par 2005).

[14] Y.H. Song, T.M. Pinkston. "Distributed Resolution of Network Congestion and Potential Deadlock Using Reservation-Based Scheduling". IEEE Trans. Parallel and Distributed Systems, v.16, N.8, 2005.

[15] M. Thottethodi, A.R. Lebeck, S.S. Mukherjee. "Exploiting Global Knowledge to Achieve Self-Tuned Congestion Control for K-Ary N-Cube Networks". IEEE Trans. on Parallel and Distributed Systems, Vol. 15, No. 3, March 2004, pp 257-272.

[16] Virtutech Inc. "Simics page". Available (June 2006) at http://www.virtutech.se/products/

Acknowledgements

This research has been supported by the Spanish Ministerio de Educación y Ciencia, under grant TIN2004-07440-C02-01. Mr. Navaridas is supported by a pre-doctoral grant from the University of the Basque Country. We also acknowledge the Barcelona Supercomputing Center (BSC) for supplying computing resources for our research.

Chapter 13. Case study 2: Topological analysis of an indirect network

Full reference:
Javier Navaridas, Jose Miguel-Alonso, Javier Ridruejo, Wolfgang Denzel.
Reducing Complexity in Tree-like Computer Interconnection Networks.
Parallel Computing
Volume 36, Issue 2-3, February 2010, Pages 71-85.

We used INSEE to test a novel network topology in this paper. The workload used for the simulation were the micro-kernel applications, which were applied into the study of a new kind of indirect network topology: the *k:k'*-ary *n*-thin-tree. It has a lower cost than *k*-ary *n*-tree (for the same number of connected nodes), because it uses less upward links, reusing the freed ports as downward ports.

This article has been chosen as case study to show the use of INSEE as tool to evaluate novel topologies and to carry out performance prediction studies. Synthetic traffic in form of application kernels was used as workloads. However, the main contribution of the paper is the introduction, discussion and evaluation of a novel indirect topology for INs. Carefully selected thin trees have almost the same performance than regular (fat) trees, but at a fraction of the cost, provided that the system scheduler makes a good work exploiting communications locality. Another contribution of this paper is the proposal of a methodology to measure the cost/performance ratio of networks, which is tested on thin-trees with different slimming ratios.

The author of this thesis was involved in the design and modeling of thin-trees in FSIN, the simulation experiments and the performance and cost comparison with regular fat trees.

Parallel Computing 36 (2010) 71–85

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Reducing complexity in tree-like computer interconnection networks

Javier Navaridas [a,*], Jose Miguel-Alonso [a], Francisco Javier Ridruejo [a], Wolfgang Denzel [b]

[a] Department of Computer Architecture and Technology, The University of the Basque Country, San Sebastian, Spain
[b] IBM Research GmbH, Rüschlikon, Switzerland

ARTICLE INFO

Article history:
Received 10 March 2008
Received in revised form 13 August 2009
Accepted 21 December 2009
Available online 28 December 2009

Keywords:
k-ary *n*-tree topology
Traffic characterization
Simulation
Performance evaluation

ABSTRACT

Interconnection networks based on the *k*-ary *n*-tree topology are widely used in high-performance parallel computers. However, this topology is expensive and complex to build. In this paper we evaluate an alternative tree-like topology that is cheaper in terms of cost and complexity because it uses fewer switches and links. This alternative topology leaves unused upward ports on switches, which can be rearranged to be used as downward ports. The increase of locality might be efficiently exploited by applications. We test the performance of these thin-trees, and compare it with that of regular trees. Evaluation is carried out using a collection of synthetic traffic patterns that emulate the behavior of scientific applications and functions within message passing libraries, not only in terms of sources and destinations of messages, but also considering the causal relationships among them. We also propose a methodology to perform cost and performance analysis of different networks. Our main conclusion is that, for the set of studied workloads, the performance drop in thin-trees is less noticeable than the cost savings.

1. Introduction

The *k*-ary *n*-tree topology [22], based on the classic fat-tree topology introduced by Leiserson [17], is often the topology of choice to build low latency, high bandwidth and high connectivity interconnection networks (hereafter IN) for parallel computers. Its main characteristics are the low mean path length and the multitude of paths from a source to a destination node, which increases exponentially with the distance between nodes (in number of hops). This high path diversity provides a good performance rate for almost all kind of workloads, independently of their spatial, temporal and length distributions.

However, its design does not take into account that parallel applications usually arrange their processes in such a way that communicating processes are as close as possible (in terms of process identifier) to each other, trying to obtain advantages from locality in communication. A network design that ignores locality could be a good option because, in some of the largest parallel systems currently operating, schedulers see processors as an unstructured pool of resources, and assigns them to parallel jobs without guaranteeing that neighbor processes (*i.e.*, consecutive identifiers) run in neighboring compute nodes (attached to the same or adjacent switches). The result is a random mapping of processes to nodes that may require high bandwidth at all network levels, because many nodes will generate messages addressed to distant pairs. Not all the schedulers function this way: there are a few that are *topology-aware* and schedule applications in consecutive partitions of the network, thus allowing for an effective exploitation of locality. In these cases, for most applications, the bisection bandwidth would no longer be the main performance limiting factor, and the upper levels of network would be under-utilized.

* Corresponding author.
E-mail addresses: javier.navaridas@ehu.es (J. Navaridas), j.miguel@ehu.es (J. Miguel-Alonso), franciscojavier.ridruejo@ehu.es (F.J. Ridruejo), wde@zurich.ibm.com (W. Denzel).

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2009.12.004

We can reduce cost and complexity of the IN by reducing the ratio between the number of links connected to upper levels and those connected to lower ones. This can be done reducing the radix of the switches or, alternatively, increasing the locality by rearranging the upward ports and making them downward. In both cases the total cost of the system is reduced: fewer switches, fewer links and, in the former case, switches of lower complexity. If parallel applications are correctly placed, performance should not suffer. They could even experience an improvement due to the increased locality of the latter case. In this paper we propose to use thin-trees to, this way, reduce cost and complexity of interconnection networks by doing what we have just described. Thin-trees are directly derived from the *k*-ary *n*-tree topology, reducing the number of upward ports of all switches.

In order to test the different networks, we have performed a throughput study for uniform traffic, both analytically and via simulation. Ideally we would evaluate performance using real traces taken from actual scientific applications running on very large systems but, as large traces are difficult to obtain and not very manageable, we have used a collection of synthetic workloads that emulate their behavior. This mimicry is done not only in terms of spatial patterns, but also in terms of the causality of the injected messages. Some of the communication patterns replicate the way collectives are implemented in common MPI libraries. Others reproduce data interchanges performed in applications that rely on virtual topologies – usually, meshes – commonly used in matrix calculus. The length of the messages and the number of nodes can be specified as parameters.

We have selected some instances of the topologies under study, fed the simulator with the proposed workloads for a variety of message lengths, and measured their performance. A comparison of alternatives is done using raw performance or a performance/cost ratio. As performance is application-dependent, we define a model to compute a performance indicator that can be tailored to fit the characteristics of a given supercomputing center. We will see that, in terms of this indicator, the *k*-ary *n*-tree shows its superiority as a general-purpose topology, although slimmed topologies perform equally well for some relevant application mixes. If cost is considered too, the complexity of the *k*-ary *n*-trees plays against them and the thin-tree is the clear winner: cost is lower and performance is good – in some cases, even better than that of the regular tree, due to a better exploitation of locality.

The rest of this paper is organized as follows: In Section 2 we discuss some topologies in use in former and current high performance computers and also some schedulers and their job placement policies. In Section 3 we present the topologies we will evaluate. The experimental environment – model of the elements, selected topologies and proposed workloads – is explained in Section 4. In Section 5 we show the experimental work and analyze results taking into account only the raw performance. To obtain a fairer comparison of the different topologies, we make a proposal of cost and performance functions, and carry out a performance/cost study in Section 6. We close this work with some conclusions and a future work outlook in Section 7.

2. Related work

Indirect interconnection networks have evolved noticeably from the first multi-stage networks as those proposed by Clos [7]. Those networks were built with low-radix switches (typically 4 or 8) and aimed to interconnect at most a few hundred nodes. Current spines, as that on the MareNostrum supercomputer [5], have switches with hundreds of ports and are able to interconnect thousands of nodes. Former trees were low-radix: the CM-5 [18] had a radix-8 data network. Current ones use switches with higher radices, as those radix-24 of the Cray XD1 [9]. There are also recent tree-like proposals as the Black Widow Clos network [11] that takes advantage of the high availability of ports (radix-64 switches) to add side-links to the common tree-like arrangement. However, the most noticeable change in these networks is that former indirect networks were built *ad hoc* for the target systems, whereas current high-performance networking technologies as QsNet [21], Myrinet [19] or InfiniBand [15] have favoured building super-clusters with *off-the-shelf* components.

Network bandwidth and latency have experienced notable improvements during the last 10 years, from the 800Mbps of the ASCI Red (1997) [26] to the 20Gbps currently available in InfiniBand [15] when using 4X, dual-data-rate connections, or the 10Gbps by Myri-10G and 10Gb Ethernet, both offered by Myricom [19]. Soon we will see offers of 100–120 Gbps (100G Ethernet, InfiniBand 12X-QDR). This takes us to a network bandwidth improvement over 100 times in 10 years. The latency of the full protocol and the network in the ASCI Red (taking into account message passing library) is 12 μs. Both Myri-10G and InfiniBand latencies are around 2 μs. Thus, latency has been improved (around 6 times), but not as noticeably as bandwidth has.

Taking a look at the most current Top500 list [12], we can see two clear trends. On the one hand, the choice of topology for *custom-made*, massively parallel computers is the 3D cube. On the other, *commodity-based* systems (super-clusters) are built around the class of trees discussed in this paper. Most of the machines in the middle positions are arranged this way, which justifies our interest in tree-like topologies.

We stated in Section 1 that common schedulers do not take into account the underlying network topology. The only supercomputer we have found that tries to maintain locality is the BlueGene family (3D tori), whose scheduler [3] puts tasks from the same application in one or more mid-planes ($8 \times 4 \times 4$). On the contrary, the scheduling strategy [1] of Cray XT3/XT4 family (also 3D tori) gets the first available compute processors. Some job queuing and scheduling managers like Sun Grid Engine [27], LoadLeveler [14], or MOAB [8] do not offer locality-aware politics, but provide mechanisms to implement them.

3. Topologies under study

In this section we will describe two different multi-stage, tree-based topologies. In these descriptions we assume that all switches used to build a given network have the same radix. For the purpose of this paper we leave unplugged the upward ports of the topmost level of switches. This assumption has advantages in terms of simplicity in the descriptions, and also provides scalability. The disadvantage is in terms of cost, because some resources are unused; this is particularly relevant for those topologies with more switches in the top level. In practical implementations, all ports of the highest switch level may be used as downward ports, eventually resulting in a larger network size. Alternatively, we may consider a single switch as an aggregation of lower radix *virtual* switches, which results in a smaller number of switches in the topmost stage of the system.

3.1. Definitions

In the graphical representations of the topologies (see Fig. 1), boxes represent switches and lines represent links between them. Note that we show neither the compute nodes connected to the first level switches and their links, nor the last level of upward links (which, as we stated before, are unplugged). These elements are hidden for the sake of clarity.

Throughout this paper we will use n to denote the number of levels in a network, and N to denote the number of compute nodes (leaves) attached to it. We will denote the total number of switches in a topology as S, and the number of switches at level i as S_i. The total number of links will be denoted as L. The switch radix will be denoted as R. L_B will denote the link bandwidth, B_B the bisection bandwidth, and B_C the number of channels in the bisection. We will denote the theoretical, ideal throughput for uniform traffic as Θ. We call the relation between the number of downward ports of a switch and the number of upward ports the *slimming factor*. For example, taking a look at the switches in the topology shown in Fig. 1b, four ports are downward ports, linked to switches in the next lower level. The remaining two ports of each switch are upward ports that connect to switches in the next higher level; therefore the slimming factor is 2:1, or simply 4:2.

In the topological descriptions that follow, we denote each switch port within the system as the level where the switch is, the position of the switch in that level, and the number of the port in that particular switch. We call the lower level of switches (those attached to compute nodes) level *0*; obviously, level $n-1$ is the one on the top of the tree. We number the switches in each level from left to right, starting from *0*. Ports in a switch are denoted as upward ($\uparrow$) or downward ($\downarrow$), and numbered from left (*0*) to right. Thus, a port can be addressed as a 4-tuple $\langle level, switch, port, direction\rangle$.

Given two ports P and P', they are linked ($P \leftrightarrow P'$) when there is a connection (link) between them. As links are full-duplex, in the expressions concerning linkage we avoid the redundancy of showing downward connections. We will call *level*, *switch* and *port* the address components of a given port, and *nlevel*, *nswitch* and *nport* the address components of the port to which it is connected (its upper neighbor). Therefore,

$$\langle level, switch, port, \uparrow\rangle \leftrightarrow \langle nlevel, nswitch, nport, \downarrow\rangle$$

Along this paper, we will refer to *heavy* and *light* workloads. Light workloads are those in which the number of messages circulating simultaneously through the network is low, and the length of the messages is short. In contrast, heavy workloads are those in which most of the nodes are injecting messages at once so that the network will experience peaks of congestion; this situation would be even worse if messages are addressed to distant destinations.

3.2. k-ary n-tree

This is the best-known of the topologies considered in this study. It will be the yardstick to compare the thin-tree against. k-ary n-trees [22], where k is half the radix of the switches – actually, the number of links going upward (or downward) from the switch – and n the number of levels, will be denoted through this paper as k:k, n-tree. Note that in this case the slimming factor is 1:1, or simply k:k.

A k-ary n-tree is typically built in a butterfly fashion between each two contiguous levels. Fig. 1a shows a depiction of a 4-ary 3-tree. The topological neighborhood description is as follows:

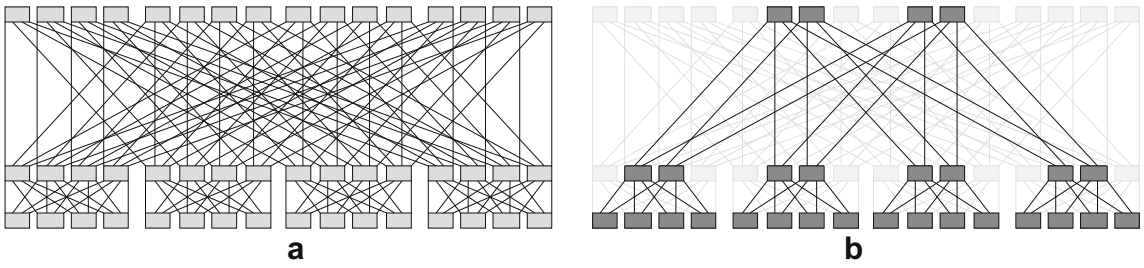


Fig. 1. Samples of the topologies under study to build 64 nodes networks. (a) 4-ary 3-tree or 4:4, 3-tree (b) 4:2-ary 3-thin-tree or 4:2,3-tree.

$$\forall\ level \in [0, n-1),\ \forall\ switch\ \in [0, k^{n-1}),\ \forall\ port\ \in [0, k):$$
$$nlevel = level + 1$$
$$nswitch = \left((port \cdot k^{level}) + \left(k^{level-1} \cdot \left\lfloor \frac{switch}{k^{level}+1} \right\rfloor\right) + (switch \bmod k^{level})\right) \bmod k^{n-1}$$
$$nport = \left(\left\lfloor \frac{switch}{k^{level}} \right\rfloor - \left(k^{level+1} \cdot \left\lfloor \frac{switch}{k^{level}} \right\rfloor\right)\right) \bmod k$$

The main advantages of this topology are the high bisection bandwidth and the large number of routing alternatives for each pair of source and destination – a path diversity that can be exploited via adaptive routing. Nevertheless, they might be expensive and complex to deploy, because of the large number of switches and links.

Note that bandwidth remains constant in all levels. Most parallel application exhibit some level of locality in communication. This means that, in actual scenarios, the higher the level, the lower is the utilization of resources in that level. This is what supports the utilization of slimming strategies: an attempt to reduce complexity in the upper levels without sacrificing application performance.

3.3. k:k′-ary n-thin-tree

We define a thin-tree as a *cut-down* version of a k-ary n-tree in which we apply a given slimming factor. We will denote them as k:k', n-tree, being k the number of downward ports, k' the number of upward ports and n the number of levels. The slimming factor is, obviously, the ratio between k and k'. k does not need to be a multiple of k' so that we can produce a thin-tree with arbitrary values of k and k'. It is remarkable that a k-ary n-tree is actually a k:k-ary n-thin-tree.

A 4:2-ary 3-thin-tree is depicted in Fig. 1b. Removed switches and links from a full-fledged k-ary n-tree are shaded. The topological neighborhood relationship between ports in a thin-tree is described as follows:

$$\forall\ level\ \in [0, n-1),\ \forall\ switch\ \in \left[0, k \cdot \left\lfloor \frac{k}{k'} \right\rfloor^{n-level}\right),\ \forall\ port \in [0, k'):$$
$$nlevel = level + 1$$
$$nswitch = \left((port \cdot k'^{level}) + (switch \bmod k'^{level}) + \left\lfloor \frac{switch}{k \cdot \left\lfloor \frac{k}{k'} \right\rfloor^{level}} \right\rfloor \cdot k' \cdot \left\lfloor \frac{k}{k'} \right\rfloor^{level}\right)$$
$$nport = \left\lfloor \frac{switch}{\left\lfloor \frac{k}{k'} \right\rfloor^{level}} \right\rfloor \bmod k$$

In this topology the bisection bandwidth has been reduced, as well as the number of switches and links (*i.e.*, cost and complexity). We want to investigate how applications suffer this reduction. Thin-trees are easier to deploy than regular trees and, if k and n values are kept, the radix of switches is smaller.

3.4. Theoretical throughput

We open this sub-section with Table 1, which summarizes the relations between network parameters (n, k and k'), number of elements (N, S, S_i, L, R) and topological properties (B_B, B_C, Θ) for the topologies under study. The computation of Θ for both topologies – a common way to evaluate interconnection networks performance – is done in the following paragraphs, via an analytical study.

Table 1
Topological characteristics of the topologies.

	k-ary n-tree	k:k'-ary n-thin-tree
Nodes	$N = k^n$	$N = k^n$
Switches	$S = n \cdot k^{n-1}$	$S = \sum_{i=0}^{n-1} k'^{(n-i)-1} \cdot k^i$
Switches per level $\forall i \in [0, n-1]$	$S_i = k^{n-1}$	$S_i = k^{(n-i)-1} \cdot k^i$
Links	$L = S \cdot k$	$L = S \cdot k$
Switch radix	$R = 2 \cdot k$	$R = k + k'$
Channels in bisection	$B_C = \frac{k^n}{2}$	$B_C = \frac{k'^{(n-1)} \cdot k}{2}$
Bisection bandwidth	$B_B = \frac{k^n}{2} \cdot L_B$	$B_B = \frac{k'^{(n-1)} \cdot k}{2} \cdot L_B$
Theoretical throughput	$\Theta = 1$	$\Theta = \left(\frac{k'}{k}\right)^{n-1}$

As the real throughput of a system depends on the link bandwidth, we will focus the study on the accepted load relative to the maximum load a node is able to inject. To do this, we assume that the link bandwidth is constant trough the whole network. This way, as every node is connected to the network via a single link, the throughput should be in (0,1]. Note that zero-throughput means the absence of connectivity.

As stated in chapter 3 of [10] the upper boundary of the throughput of a given topology could be expressed as the ratio between the bisection bandwidth and the number of nodes. Therefore, the theoretical ideal relative throughput (hereafter ideal throughput) is:

$$\Theta = \frac{2 \cdot B_B}{N \cdot L_B} \tag{1}$$

As the link bandwidth is constant, the number of channels in the bisection is calculated as:

$$B_C = \frac{B_B}{L_B} \tag{2}$$

From (1) and (2) we obtain:

$$\Theta = \frac{2 \cdot B_c}{N} \tag{3}$$

In both cases, the number of channels in the bisection is half the number of links at the last stage, *i.e.*:

$$B_C = \frac{S_{n-1} \cdot k}{2} \tag{4}$$

This way, we can compute the ideal throughput for uniform traffic as:

$$\Theta = \frac{2 \cdot S_{n-1} \cdot k}{2 \cdot N} \tag{5}$$

Thus, from (5) and looking at the values of S_i in Table 1, the ideal throughput of a k-ary n-tree is:

$$\Theta = \frac{k^{n-1} \cdot k}{k^n} = 1 \tag{6}$$

And the ideal throughput of a k:k'-ary n-thin-tree is:

$$\Theta = \frac{k'^{(n-1)} \cdot k}{k^n} = \left(\frac{k'}{k}\right)^{n-1} \tag{7}$$

4. Experimental set-up for simulation-based evaluation

We use INSEE [23] to evaluate some different tree-like networks, feeding them with a collection of application-inspired synthetic workloads. The simulator measures time in terms of cycles, the time required by a phit to traverse one switch – switching plus transmission.

4.1. Switches

For this work, we have used simple input-buffered switches whose radices range from 9 to 16, depending on the topology. In order to keep things simple, we do not use virtual channels, except if explicitly indicated. The arbitration of each output port is performed in a random way, that is, every time an output port is free it randomly chooses among all the input ports that have requested this resource. Transit queues are located in the input ports and are able to store 4 packets. A schematic model of the switch is depicted in Fig. 2.

In this work we model the node as a traffic generation source with one injection queue, which is able to store 8 packets. It is also the sink of the arrived messages. When generating traffic, we consider reactive sources, meaning that the reception of a message may *trigger* the release of a new one. This way we can model the causality inherent to actual application traffic. Messages are split into packets of a fixed size of 16 phits. One phit is the smallest transmission unit, fixed to 128 bits. If a message does not fit exactly in an integral number of packets, the last packet contains unused phits.

The switching strategy is virtual cut-through. Routing is adaptive – but restricted to shortest paths in order to avoid deadlock – using a credit-based mechanism, being the credit the space in the queue of the neighbor's input port. This mechanism works as follows: when several output ports are feasible, the one with more available credits – more room in the neighboring queue – is selected; if several ports have the same amount of credit, one of them is selected at random. Credits are communicated out-of-band, so they do not interfere with regular traffic. The use of adaptive routing allows taking the best of each of the topologies, which in turn allows for a fairer comparison among them. Reader should note that optimal static routing functions are application-dependent and have established a line of research by themselves. An interesting contribution that

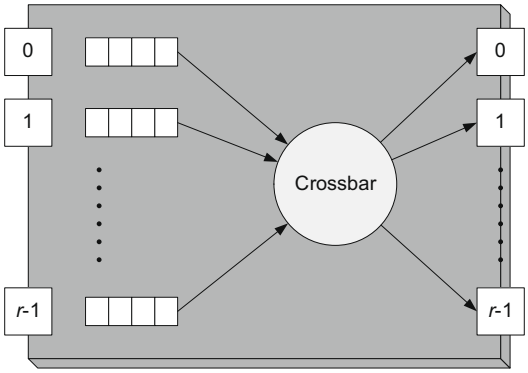


Fig. 2. Model of the switch used in the simulations given a radix *r*. Ports at the left are input ports and those at the right are output ports.

is in line with the contents of this paper is the static routing algorithm presented in [25] which was used with the kind of trees studied in this paper.

4.2. Networks under study

In this work we have performed three sets of evaluations for the topologies under study. In the first evaluation, which revolves around the theoretical throughput, we have fixed the number of downward ports per switch (set to 8), the slimming factors (set to 8:6, 8:4 and 8:2) and the target number of connectable compute nodes (set to 4096). With these restrictions, we have worked with the following topologies: 8:8,4-tree, 8:6,4-tree, 8:4,4-tree and 8:2,4-tree.

The second set of experiments used traffic from some kernels of applications to feed a wide variety of networks, all of them using switches with 8 downward ports. All the possible slimming factors have been used, from 8:8 (the complete tree) to 8:1. Furthermore we have tested three different scales of the system, able to connect 64 nodes (2 levels), 512 (3 levels) and 4096 nodes (4 levels). All the evaluated topologies and some of their characteristics are summarized in Table 2.

As we have just stated, all the switches have 8 downward ports. However, the actual radix of the switches is not always the same, being smaller in the more slimmed topologies. Thus, in these two evaluations the complete tree has advantage compared with the thinner alternatives: it uses more links, and more switches that also are larger. Thus performance measurements are biased towards the 8:8,4-tree.

In the last evaluation set we have fixed the radix of the switches (set to 12), and used all the feasible slimming factors (from 11:1 to 6:6). Under these restrictions we have created the smallest topologies capable to connect at least 64, 512 and 4096 nodes. The result of the evaluations would be fairer than in the previous set, because all the switches have the same radix. Note how thinner topologies have lower bandwidth and path diversity than regular trees, but in return locality is increased. Unfortunately, the proposed networks have different sizes, in terms of the number of compute nodes they can connect. As we are using workloads that emulate a fixed number of tasks, we are not capable of using all the trees' leaves. Table 3 summarizes some characteristics of the networks in this third evaluation set. For example note how it is possible to build a

Table 2
Characteristics of the topologies in the second set of experiments.

(a)	8:1,2-tree	8:2,2-tree	8:3,2-tree	8:4,2-tree	8:5,2-tree	8:6,2-tree	8:7,2-tree	8:8,2-tree
Nodes	64	64	64	64	64	64	64	64
Switches	9	10	11	12	13	14	15	16
Radix	9	10	11	12	13	14	15	16
Links	72	80	88	96	104	112	120	128
(b)	8:1,3-tree	8:2,3-tree	8:3,3-tree	8:4,3-tree	8:5,3-tree	8:6,3-tree	8:7,3-tree	8:8,3-tree
Nodes	512	512	512	512	512	512	512	512
Switches	73	84	97	112	129	148	169	192
Radix	9	10	11	12	13	14	15	16
Links	584	672	776	896	1032	1184	1352	1536
(c)	8:1,4-tree	8:2,4-tree	8:3,4-tree	8:4,4-tree	8:5,4-tree	8:6,4-tree	8:7,4-tree	8:8,4-tree
Nodes	4096	4096	4096	4096	4096	4096	4096	4096
Switches	585	680	803	960	1157	1400	1695	2048
Radix	9	10	11	12	13	14	15	16
Links	4680	5440	6424	7680	9256	11200	13560	16384

(a) 64-node systems. (b) 512-node systems. (c) 4096-node systems.

Table 3
Characteristics of the topologies in the third set of experiments.

(a)	11:1,2-tree	10:2,2-tree	9:3,2-tree	8:4,2-tree	7:5,3-tree	6:6,3-tree
Nodes	121	100	81	64	343	216
Switches	12	12	12	12	109	108
Radix	12	12	12	12	12	12
Links	132	120	108	96	763	648
.						
(b)	11:1,3-tree	10:2,3-tree	9:3,3-tree	8:4,3-tree	7:5,4-tree	6:6,4-tree
Nodes	1331	1000	729	512	2401	1296
Switches	133	124	117	112	888	864
Radix	12	12	12	12	12	12
Links	1463	1240	1053	896	6216	5184
(c)	11:1,4-tree	10:2,4-tree	9:3,4-tree	8:4,4-tree	7:5,5-tree	6:6,5-tree
Nodes	14641	10000	6561	4096	16807	7776
Switches	1464	1248	1080	960	4440	5184
Radix	12	12	12	12	12	12
Links	16104	12480	9720	7680	31080	31104

(a) Small-scale systems. (b) Medium-scale systems. (c) Large-scale systems.

8:4,4-thin-tree with exactly 4096 nodes, but the complete 6:6,5-tree built with 12-port switches has 7776 leaf-nodes. In terms of cost this plays against the topologies that provide the worst fit to the target number of nodes.

4.3. Workloads

First of all we fed the networks under study with synthetic uniform traffic from independent sources to perform a *classic* throughput evaluation. This set of experiments will show us a first estimation of the networks' potential, and their capacity to handle heavy workloads. It also will validate the analytical study performed in Section 3.4.

As we stated in the introduction, we would also like to test the selected networks with *realistic* traffic, ideally taken from traces obtained from applications running in actual supercomputers. Since it is difficult to obtain and handle traces of applications running on thousands of nodes, we decided to create instead some synthetic traffic generators which emulate data interchanges typically used in scientific parallel applications.

The patterns of choice will be described in the following paragraphs. In the descriptions that follow, N is the number of processes in the parallel application, identified from 0 to $N-1$. Note that we assume a mapping of one process to one compute node attached to one leave of the network, which results in the consecutive allocation of the tasks into the nodes – *i.e.*, task n goes to node n. The graphical representation of these patterns is depicted in Fig. 3.

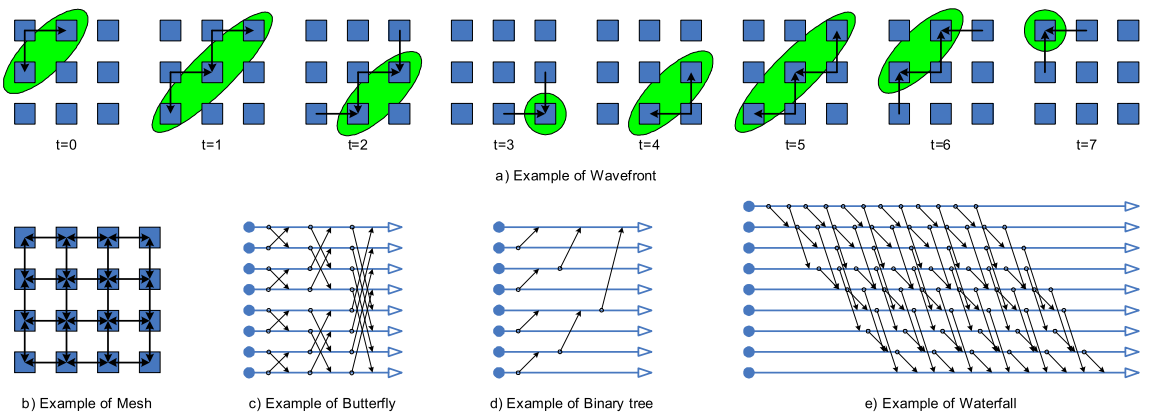


Fig. 3. Graphical representation of the traffic patterns: time flows from left to right. Grey lines and squares represent nodes. Each black arrow start means a message send. The end of the black arrows means that the node has to stop until receiving the corresponding message. (a) Wave-front (Green) in a 3×3 2D-mesh. (b) Neighbor interchange in a 4×4 2D-mesh. (c) Butterfly that emulates N-to-N collectives (8 nodes). (d) Binary tree that emulates N-to-1 collectives (8 nodes) and (e) Waterfall pattern (9 nodes).

For all patterns each node starts with an initial set of messages that have to be injected into the network. The length of these messages is configurable. Messages must be packetized before injection, thus long messages generate a burst of packets. The reception of a message may *trigger* the release of some additional ones; this means that patterns include *causal relationships*. In the simulator, we start with an empty network and measure the time used to consume the initial collection of messages, plus all additional messages generated by causal relationships, until the network is empty again.

We have generated these patterns for fixed network sizes of $N = 64$, $N = 512$ and $N = 4096$ compute nodes and message lengths of 40KB – as will be explained later, in waterfall pattern the size of the actual messages is much smaller. Most of the patterns in use in this work were further defined and justified in [20].

The 2D and 3D wave-front patterns (**W2** and **W3**) perform a diagonal *sweep* from the first node to the last one in MPI virtual square (or cubic) meshes, and then returns to the first node. The simulation of this patterns starts with two (three for **W3**) messages in node 0, and ends with the finalization of the return sweep. These patterns are considered light – note that there are only a few nodes injecting at once – but create some contention in the destination nodes because they have to receive data from several neighbors. We can observe this pattern in applications implementing the Symmetric Successive Over-Relaxation (SSOR) [6] algorithm – used to solve sparse, triangular linear systems.

The 2D and 3D mesh patterns (**M2**, **M3**) perform data movements in MPI virtual square (or cubic) meshes from every node to all its neighbors; after that, each node waits for the reception of all messages from its neighbors. Simulation starts with all nodes injecting one message per direction (2–4 for **M2**, 3–6 for **M3**), and ends once all messages arrive. These patterns impose a very heavy load on the network, because all nodes inject simultaneously several messages at once before stopping to wait for the receptions. These patterns can be observed in applications using finite difference methods [2].

The butterfly pattern (**BU**) provides an efficient implementation of MPI N-to-N collectives (MPI_Alltoall, MPI_Allreduce, etc.) [28]. It is also known as "recursive doubling". Simulation of **BU** starts with a message at each node, and ends when defined by the pattern. This is a heavy pattern because all the nodes inject at once, and also in the last stages of the butterfly the messages have to traverse the whole network, which may exhacerbate congestion.

The binary tree pattern (**BT**) provides an efficient implementation of some N-to-1 MPI collective operations, such as MPI_Reduce and MPI_Gather [16]. Simulation of this pattern starts with a message at odd-numbered nodes, and ends when node 0 receives the messages from all power of two nodes (included $2^0 = 1$). This is the lightest of the patterns because there is almost no contention in the delivery of the messages.

The waterfall traffic pattern (**WF**) is inspired in a pattern we have observed in the NAS Parallel Benchmark LU [24]. It consists of a large collection of small messages, with causal dependencies. For this pattern, we define a total number of bytes to transmit (*length*) and, instead of starting with a single 40KB message, a burst of 40 messages of 1KB length is generated. **WF** can be seen as a burst of **W2**s (actually, LU uses SSOR) but using small message, of fixed length. Node 0 starts a burst of messages that flood the network. The simulation ends when the last burst arrives to node $N - 1$. The main characteristic of **WF** is the presence of causality chains. Latencies in the delivery of messages accumulate at the end of the chain. We can consider this pattern heavy because during most of the execution time, the majority of the nodes are injecting messages at once, however it is not as heavy as **BU** or **M2** and **M3** because the causality chains throttle the injection of messages.

5. Experiments and analysis of results

In this section we will evaluate the networks described above. In the initial set of experiments we feed the networks with uniform traffic from independent traffic sources. These preliminary, non-realistic tests could make us think that thin-trees are not a good topological choice. However, a deeper study using a selected mix of workloads that mimic applications behavior tells us a different story: when taking into account that high performance applications' processes synchronize and maintain causal relationships, the load that traverses the upper stages of a tree-like topology is smaller than that traversing the lower stages; therefore, we can build trees that are thinner at the upper stages without adversely affecting the execution time of applications – but with a very positive impact in terms of budget.

5.1. Throughput

The first set of experiments to evaluate the performance of the topologies under test will be by means of a classical throughput measurement for uniform traffic from independent traffic sources. We will plot the accepted load *versus* the offered load to, this way, have a first estimation of how fast the networks reach saturation and what performance could we expect from them under heavy loads. We perform these measurements via simulation using the following methodology. First of all, there is a fixed *warm-up* period of 30K cycles. After that, a *convergence phase* is started. In this phase we measure the average load every 1K cycles. When four consecutive measurements are within a range of 5%, we consider that the convergence phase is finished and that the simulation has reached a *stationary phase*. After this point, the throughput measurement is started. This is the last phase in which 10 batches of 5K cycles each are captured. The result shown is the average accepted load of these ten batches. Note that the standard deviation of the average accepted load of the batches in all experiments is lower than the 0.5% of the average value.

We plot in Fig. 4 the throughput evaluation for the four networks under study. As stated in Chapter 13 of [10], when adding virtual channels the average throughput of the network is increased due to contention reduction. Thus, in order to

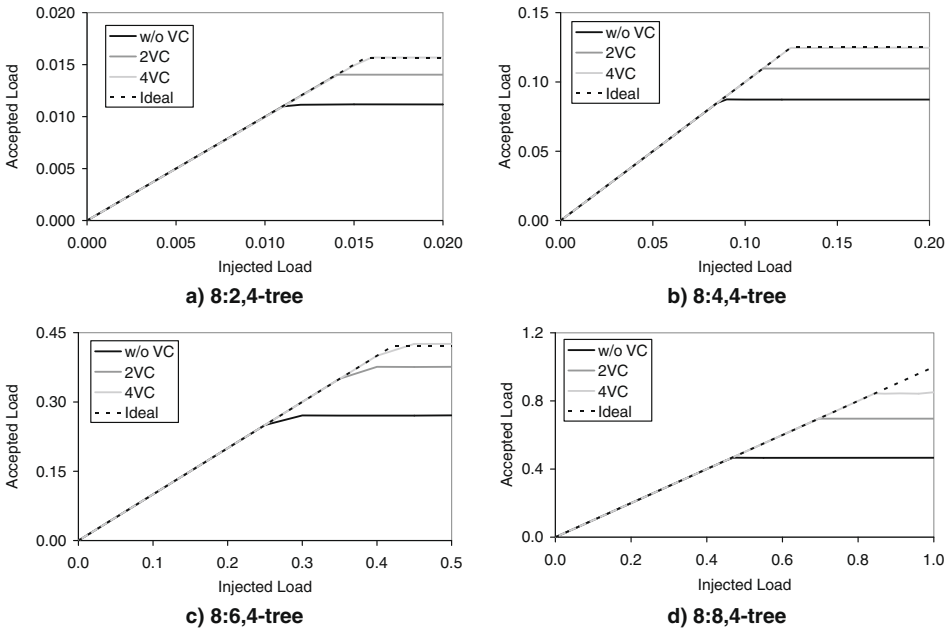


Fig. 4. Relative throughput for networks under study with uniform traffic with independent traffic sources. With 1, 2 or 4 VC and the upper boundary of the throughput. Note the difference in the axis ranges of each topology.

compare and validate the analytical study in Section 3.4, we also plot the results for the same topologies using 2 and 4 virtual channels (denoted as 2VC and 4VC, respectively) and the computed ideal throughput curve.

We can observe that the *k*-ary *n*-tree topology is not able to reach the ideal limit, not even when using 4 VC. Note that the ideal limit of 1 means that all nodes send and receive one phit per cycle which is the maximum allowed by any link attaching a node to the network. Any form of contention, including that at destination nodes, will reduce the actual throughput. The utilization of multiple VCs can help reducing unnecessary contention, but it cannot be completely eliminated. In the case of the thin-trees (8:6,4-tree 8:4,4-tree and 8:2,4-tree), nodes can still inject up to one phit per cycle, which is much higher that the theoretical throughput limit. Consumption is never a bottleneck, and the utilization of a few virtual channels allows the network to reach the ideal throughput.

The reader should note the difference in the axis range for the three experiments. This is because the ideal throughput for the networks under study is 1 for the 8:8,4-tree, ∼0.42 for the 8:6,4-tree, 0.125 for the 8:4,4-tree and ∼1.6E-2 for the 8:2,4-tree.

This first performance evaluation say us that thin-trees are not a good idea if we have to deal with random traffic from independent sources. However, in the next subsections we will study the networks with workloads that *mimic* actual parallel applications.

5.2. Experiments with same size networks

In this set of experiments we have gathered the time (in simulation cycles) used by the networks to deliver all the messages of each of the application-kernels. As these times differ widely, due to the characteristics of the patterns, we have normalized them, using the times for the complete trees as the reference. These normalized times are represented in Fig. 5. Note that we have truncated the results to 10, in order to allow a clear depiction of the values. When a measured value exceeds this value, the actual value is plotted close to the corresponding bar.

The reader can observe that most thin-tree networks perform acceptably with light traffic patterns (**BI**, **W2** and **W3**). With them, the *k*-ary *n*-tree cannot take advantage of its high bandwidth and path diversity, just because the network occupancy is low and so is the probability of two packets competing for an output port in a switch. In contrast, under heavy loads, the high bandwidth of the *k*-ary *n*-tree topology is able to handle the high amount of packets inside the network. In the slimmed networks, still, there is too much contention due to the bandwidth reduction between each level so the packet delivery is slower. This is especially noticeable in the large-scale configuration in which delivery of the messages may suffer a slowdown of up to 41.

Regarding the slimming factors, the topologies with the 8:7 slimming factor show delays in therms of the delivery times that were always below 13% and in most cases below 5% compared to those obtained by the complete tree. 8:6 slimmed topologies were always below 21% and in most cases below 10%. 8:5 slimmed topologies were always below 50% and in most

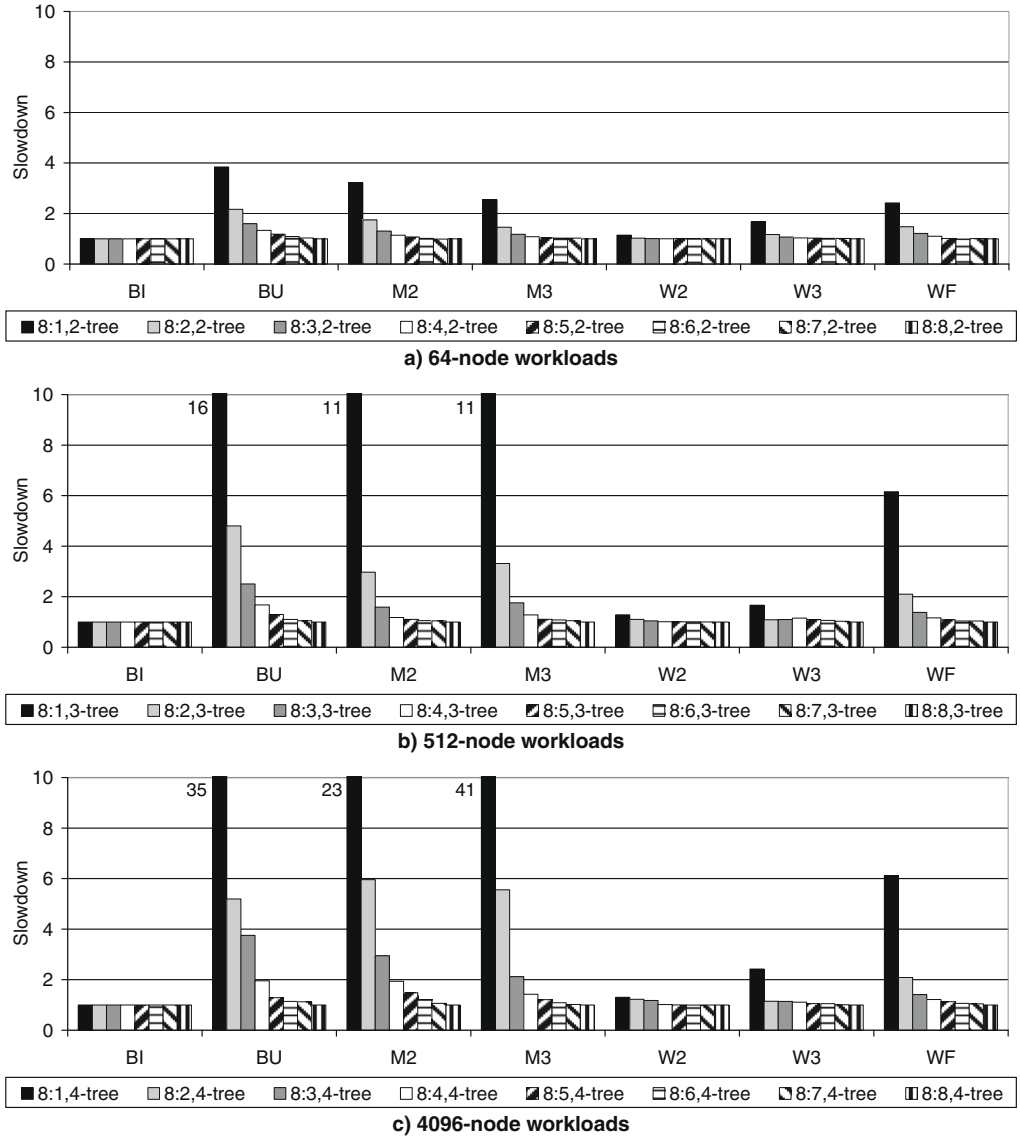


Fig. 5. Normalized time to perform all communications of each traffic pattern in the same-sized networks.

cases below 30%. The 8:4 slimmed topologies are shown to be the inflection point of performance; the slowdown values of the networks with more aggressive slimming factors rocket to values that may be intolerable.

Obviously, the smaller is the network, the less noticeable is the effect of thinning the topology, but, as we will see later, the reduction in terms of cost are also smaller. It is also noticeable that the behaviour of the networks does not scale with the size of the network.

We can conclude from this set of experiments that the *k*-ary *n*-tree is the best-performing topology in almost all experiments (combinations of pattern and message length). Nonetheless, in many cases thin-trees with reduced slimming factors (up to 8:5 or 8:4, depending on the workload) performs equally well. Additional slimming causes excessive network contention, so results are noticeably worse (over 40 times in the largest configuration).

5.3. Experiments with same radix switches

In this subsection we analyze the results of the experiments comparing topologies built with switches with the same radix (12 ports). Results are plot in Fig. 6. In general, they confirm what we learned from the previous set of experiments.

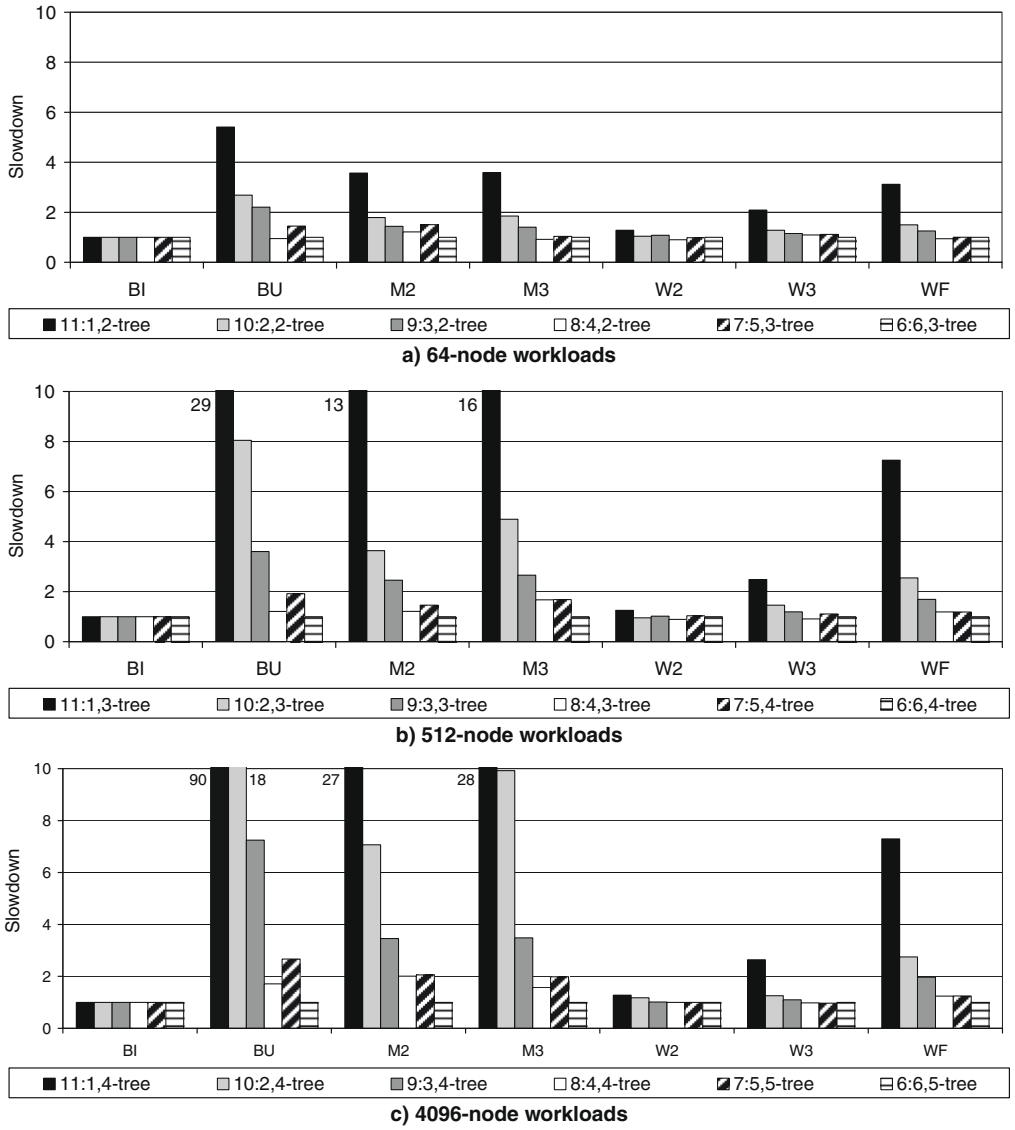


Fig. 6. Normalized time to perform all communications of each traffic pattern in the networks with radix-12 switches.

Nevertheless, in this case, the slimmed topologies have an additional advantage: the increased ability to exploit locality in communication. A switch in a 6:6,5-tree uses 6 ports as downward ports, and 6 ports as upward ports. In contrast, the same switch in a 8:4,4-thin-tree has 8 downward and 4 upward ports. In other words, in the slimmed topologies the unused upward ports are rearranged to work as downward ports. To a certain extent, this compensates the reduction of links and switches in the upper levels. The result is that slimmed topologies may outperform the complete trees. For example, the **BU** pattern is one of the heaviest workloads, and the 8:4,2-tree is able to deliver it in less time than the 6:6,3-tree.

The **W2** and **W3** patterns require specific attention. Note the excellent performance of thin-trees. The high causality of this pattern does not allow the utilization of all the resources of the complete trees, but allows for a productive exploitation of additional levels of locality – for instance, 7:5,5-tree and 8:4,4-tree deliver both workloads faster than the 6:6,5-tree.

Again, the slimming factor should not go very far. The performance of the thinnest topologies (9:3,4-tree and thinner) is too low for the heavier workloads, reaching slowdowns of up to 90 times in the largest configuration.

To summarize this section, we can state that for the lighter workloads, there is almost no difference between topologies – again, ignoring cost. For heavy loads, networks with 2:1 slimming factor, perform as well as, or even better, than those with 1:1. Growing up to higher slimming factors is counter-productive.

6. Performance/cost analysis

In the previous section we have carried out a comparison of topologies taking into account only their raw performance. We have ignored the costs of the networks under evaluation. These costs differ widely from network to network, and *must* be taken into account if we want to make a fair comparison. We propose a methodology to compare the networks. It takes into account that it is necessary to measure the performance of the network by evaluating it with appropriate workloads.

6.1. Characterizing performance

If we had unlimited (financial) resources we could just select the *best-performing* option, but that option may not be the most *cost-effective*. Here we propose a means to measure the effectiveness of a network that takes into account the workloads using it.

Actual workloads vary widely from site to site, depending on the applications in use. In this work we are not using actual applications, but a collection of synthetic – but representative – workloads. We describe a network-efficiency function in the context of these workloads that can be extended with further workload types.

For each given workload simulation reported a (relative) time T_W. For example, we have a certain execution time T_{BU} for butterfly. Note that these values are relative to the complete trees: consequently they are always 1 for this topology. Depending on the application mix of interest in a particular computing center, we may apply a weighting factor to each workload w_W. This weight should be large for those applications that are used often. For a given network, we define its performance ϕ as follows:

$$\phi = \frac{1}{\sum_W w_W \cdot T_W}$$

Note that for a given application mix (and a set of weights) a higher value of ϕ represents a better-performing network. As in this work we can neither identify all representative application mixes nor even use actual applications, we decided to use a constant value of one for all the weights, with the solely purpose of illustrating the proposed methodology. With this constant weight, the denominator in our efficiency value is just the addition of the (relative) times obtained in the experiments. This yields a value of $\phi = 1/7$ for the k-ary n-trees. We further normalize this value to be in the range [0, 1]. Tables 4 and 5 show the normalized performance values for the two sets of experiments. Note how, using this criterion, the best performing networks are the complete trees, however some configurations of thin-trees have similar ϕ value.

Table 4
Performance of the same size networks.

	ϕ
(a)	
8:1,2-tree	0.4419
8:2,2-tree	0.6970
8:3,2-tree	0.8354
8:4,2-tree	0.9094
8:5,2-tree	0.9539
8:6,2-tree	0.9791
8:7,2-tree	0.9900
8:8,2-tree	1.0000
(b)	
8:1,3-tree	0.1410
8:2,3-tree	0.4272
8:3,3-tree	0.6746
8:4,3-tree	0.8273
8:5,3-tree	0.9088
8:6,3-tree	0.9523
8:7,3-tree	0.9695
8:8,3-tree	1.0000
(c)	
8:1,4-tree	0.0628
8:2,4-tree	0.3157
8:3,4-tree	0.5164
8:4,4-tree	0.7243
8:5,4-tree	0.8569
8:6,4-tree	0.9276
8:7,4-tree	0.9647
8:8,4-tree	1.0000

(a) 64-node systems. (b) 512-node systems. (c) 4096-node systems.

Table 5
Performance of the same radix networks.

	ϕ
(a)	
11:1,2-tree	0.3491
10:2,2-tree	0.6277
9:3,2-tree	0.7341
8:4,2-tree	0.9969
7:5,3-tree	0.8644
6:6,3-tree	1.0000
(b)	
11:1,3-tree	0.0980
10:2,3-tree	0.3105
9:3,3-tree	0.5135
8:4,3-tree	0.8653
7:5,4-tree	0.7453
6:6,4-tree	1.0000
(c)	
11:1,4-tree	0.0441
10:2,4-tree	0.1667
9:3,4-tree	0.3633
8:4,4-tree	0.7359
7:5,5-tree	0.6411
6:6,5-tree	1.0000

(a) 64-node systems. (b) 512-node systems. (c) 4096-node systems.

6.2. Cost of the networks

Performing an exhaustive cost analysis of a complete system is, clearly, a difficult task that requires the knowledge of a large number of parameters, including the choice of technologies and physical placement of the elements of the system (nodes, racks). A proper cost evaluation should take into account both deployment and maintenance costs. Deployment cost must consider the number of switches and links, that may, and probably will, have different characteristics – for instance, the use of wires of different length would be needed for most plant organizations. Maintenance costs include the power consumption and the heat dissipation.

At any rate, all this concerns are outside of the scope of this paper. For this reason, we will consider three simple functions to compute the cost of each network in order to be able to carry out a performance/cost comparison. Note that these functions bear in mind some different aspects of the design of a system and, consequently the actual cost function may be a mixture of these three.

In these functions, S represents the total number of switching elements of the network and R their radix. Note that in our topological model the upward ports of the topmost stage are unplugged, and therefore the topmost level of all trees is formed by switches with a smaller radix. However, for the sake of simplicity, we will consider that all the switches have the same radix. Furthermore, to simplify these functions, the number of links is not taken into account as it depends on the number of switches and ports.

The considered cost functions are the following:

- In the first function c_C, the cost of the switch is constant regardless of its radix, $c_C = S$. Several aspects of the manufacture of the network scale linearly with the number of switches independently of their radix, such as the cost related to the plant area, the rack space or the packaging of the switch.
- In the second function c_L, the cost of the switch depends linearly on the radix, $c_L = S \cdot R$. For instance the number of links scales linearly with the radix, as well as the cost of the hardware associated to each port.
- In the third function c_Q, the cost increases quadratically with the radix, $c_Q = S \cdot R^2$. Note that the heart of a switch (the crossbar) scales quadratically with the number of ports [13].

Using the characteristics of the networks previously introduced in Table 2, we have computed the cost of the same size systems using each of the three cost functions. Furthermore, using the performance figures shown in Table 4 we have shown the cost-efficiency of the systems, calculated as the performance divided by the cost, and normalized to that of the complete tree. All these values are collected in Table 6. Note that measuring the cost of the same radix networks would be a bit tricky – and probably unfair – because every configuration has a different number of nodes. For this reason we will restrict the evaluation of the performance/cost efficiency to the same size networks.

For the small-scale configuration, we can see how the performance/cost efficiency of the 8:2,2-tree is the highest for the linear and quadratic cost functions, being 8:3,2-tree the most efficient when considering the constant cost function. This is because its performance is good, but the cost is reduced almost to one half compared with the complete tree. In this case all

Table 6
Performance/cost efficiency of the same size networks.

(a)	8:1,2-tree	8:2,2-tree	8:3,2-tree	8:4,2-tree	8:5,2-tree	8:6,2-tree	8:7,2-tree	8:8,2-tree
c_C	9	10	11	12	13	14	15	16
ϕ/c_C	0.7857	1.1152	**1.2152**	1.2125	1.1740	1.1190	1.0560	1.0000
c_L	81	100	121	144	169	196	225	256
ϕ/c_L	1.3967	**1.7843**	1.7675	1.6166	1.4449	1.2789	1.1264	1.0000
c_Q	729	1000	1331	1728	2197	2744	3375	4096
ϕ/c_Q	2.4831	**2.8549**	2.5709	2.1555	1.7783	1.4616	1.2015	1.0000
(b)	8:1,3-tree	8:2,3-tree	8:3,3-tree	8:4,3-tree	8:5,3-tree	8:6,3-tree	8:7,3-tree	8:8,3-tree
c_C	73	84	97	112	129	148	169	192
ϕ/c_C	0.3708	0.9765	1.3352	**1.4182**	1.3527	1.2354	1.1014	1.0000
c_L	657	840	1067	1344	1677	2072	2535	3072
ϕ/c_L	0.6592	1.5623	**1.9421**	1.8910	1.6648	1.4119	1.1748	1.0000
c_Q	5913	8400	11737	16128	21801	29008	38025	49152
ϕ/c_Q	1.1719	2.4997	**2.8249**	2.5213	2.0490	1.6136	1.2531	1.0000
(c)	8:1,4-tree	8:2,4-tree	8:3,4-tree	8:4,4-tree	8:5,4-tree	8:6,4-tree	8:7,4-tree	8:8,4-tree
c_C	585	680	803	960	1157	1400	1695	2048
ϕ/c_C	0.2197	0.9509	1.3171	**1.5452**	1.5167	1.3569	1.1656	1.0000
c_L	5265	6800	8833	11520	15041	19600	25425	32768
ϕ/c_L	0.3906	1.5214	1.9158	**2.0602**	1.8668	1.5508	1.2434	1.0000
c_Q	47385	68000	97163	138240	195533	274400	381375	524288
ϕ/c_Q	0.6944	2.4342	**2.7866**	2.7470	2.2976	1.7723	1.3262	1.0000

(a) 64-node systems. (b) 512-node systems. (c) 4096-node systems.

the configurations have better efficiency than the complete tree regardless of the used cost function. The only exception is the 8:1,2-tree with the constant cost function, whose loss in terms of performance is not compensated by the reduction in number of switches.

In the case of the medium-scale configuration, the 8:4,3-tree is the best contender when using the constant cost function, while the 8:3,3-tree wins for the other two cost functions. In this case, not all the thinned topologies are able to beat the complete tree, 8:1,3-tree only wins with the quadratic cost function.

If we focus on the large-scale configuration, we can notice that things are not as clear as before. The 8:4,4-tree is the most cost-effective for the linear and constant cost functions, but its performance is reduced considerably, around 30% lower than that of the complete tree. The 8:5,4-tree may be a better option due to a noticeable boost in terms of performance, even when its cost-efficiency is not as high as the 8:4,4-tree's. Finally, it is remarkable that, even an awful-performing configuration, the 8:2,4-tree, with a measured performance of 0.32, overtakes the complete tree when using the quadratic cost function.

The reader should note that the network is only a part of the system, so that the execution time depends (greatly) on the behavior of the other components, and the interactions between all of them. In other words, the advantages or disadvantages of a given network might not be as clear as shown in our evaluations. This is an argument against the better-performing, more-expensive networks, because in real set-ups the benefit of using them will be diluted. The extent of this dilution depends on the applications and their data-sets, as well as on the architecture of the system. Furthermore, the collection of workloads used in this analysis might not be representative – and probably is not—of all actual workloads used at supercomputing centers. A thorough study should be customized for a particular site, taking into account their applications and fine-tuning their relative weights.

7. Conclusions and future work

In this paper we have described and characterized a slimmed version of k-ary n-trees: the k:k'-ary n-thin-trees. A thin-tree can be seen as a k-ary n-tree after removing some links and switches. A thin-tree costs a fraction of the price of a complete tree, in terms of deployment and maintenance. In terms of performance, thin-trees with low slimming factor perform as well as comparable k-ary n-trees. Excessive slimming (beyond 2:1 in our workbench) results in bad performance results. A qualitative comparison of performance/cost ratios turns out to be very favorable for the thin-trees in the studied cases.

Thin-trees are obviously cheaper than the complete tree, but their bandwidth in the upper levels is greatly reduced. After removing links and switches in the upper levels, performance can be maintained (or even increased) due to an effective exploitation of locality. Using fixed-radix switches, a thin-tree devotes more ports to downward links, so more nodes can communicate without using the upper levels.

At least three super-clusters that are in positions #2 (RoadRunner), #56 (Tsubame) and #85 (Thunderbird) of November 2009 edition of Top500 list were built with InfiniBand networks arranged as thin trees—actually, narrowed spines. In Thunderbird the slimming factor is 2:1; RoadRunner and Tsubame goes further, to 4:1 and 5:1, respectively. We have not found any evaluation work providing the rationale behind those decisions. However, in this work we have shown that, compared to

full-fledged *k*-ary *n*-trees, thinner topologies provide comparable performances at much lower cost. The savings in the number of networking elements could be invested in other ways to improve the system (faster CPUs, better-performing networking technology, enlarge the system, etc.).

This study opens several lines for future work. We want to further explore the workload generation mechanism, to make it mimic the characteristics of parallel applications as accurately as possible. One interesting place to start is [4], were 13 "dwarfs" are identified as representative scientific applications (an extension of the original "seven dwarfs" introduced by Phil Colella).

The *scheduling* problem is also of interest. The advantage of slimmed topologies is obtained through an efficient exploitation of locality – something that is impossible to achieve under the flat-network assumption in commonly used schedulers. As in the case of the BlueGene [3], we need to make scheduler's topology-aware.

Acknowledgments

This work has been supported by the Ministry of Education and Science (Spain), Grant TIN2007-68023-C02-02, and by Grant IT-242-07 from the Basque Government. Mr. Javier Navaridas is supported by a doctoral grant of The University of the Basque Country.

References

[1] R. Ansaloni, The cray XT4 programming environment. <www.csc.fi/english/csc/courses/programming_environment>.
[2] Y. Aoyama, J. Nakano, RS/6000 SP: practical MPI programming, IBM Red Books SG24-5380-00, Chapter 4, ISBN:0738413658. August, 1999, pp. 99–153. <http://www.redbooks.ibm.com/abstracts/sg245380.html>.
[3] Y. Aridor et al., Resource allocation and utilization in the Blue Gene/L supercomputer, IBM J. Res. & Dev. vol. 49 No. 2/3 March/May 2005, <http://www.research.ibm.com/journal/rd/492/aridor.pdf>.
[4] K. Asanovic et al., The landscape of parallel computing research: a view from berkeley, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183. December 18, 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>.
[5] Barcelona Supercomputing Center Mare Nostrum. <http://www.bsc.es/>.
[6] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, S. Weeratunga, Solution of regular, sparse triangular linear systems on vector and distributed-memory multiprocessors, Technical Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA, 94035-1000, April 1993.
[7] C. Clos, A study of non-blocking switching networks, Bell System Technical Journal (1953) 406–424.
[8] Cluster Resources, Moab Workload Manager Administrator's Guide. <http://www.clusterresources.com/moabdocs/MoabAdminGuide510.pdf>.
[9] Cray Inc., Cray XD1 Overview. <http://www.cray.com/products/xd1/>.
[10] W.J. Dally, B. Towles, Principles and Practices of Interconnection Networks, Morgan-Kaufmann, 2004.
[11] W.J. Dally et al., The BlackWindow high-radix Clos network, in: Proceedings of the 33rd annual international symposium on Computer Architecture, June 17–21, 2006, pp. 16–28.
[12] J.J. Dongarra, H.W. Meuer, E. Strohmaier, Top500 Supercomputer sites, 2007 edition. <http://www.top500.org/>.
[13] H. El-Rewini, M. Abd-El-Barr, Advanced Computer Architecture and Parallel Processing, Wiley, 2005. ISBN 978-0-471-46740-3.
[14] IBM, IBM LoadLeveler for AIX 5L: Using and Administering. <http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IBMp690/IBM/usr/lpp/LoadL/html/am2ugmst02.html>.
[15] Infiniband Trade Association, Infiniband® Trade Association. <http://www.infinibandta.org>.
[16] S. Labour, MPICH-G2 collective operations, performance evaluation, optimizations. <http://www-unix.mcs.anl.gov/~lacour/argonne2001/report.ps>.
[17] C.E. Leiserson, Fat-trees: universal networks for hardware efficient supercomputing, IEEE Transactions on Computers C-34 (10) (1985) 892–901.
[18] C.E. Leiserson et al., The network architecture of the connection machine CM-5, in: Symposium on Parallel Algorithms and Architectures (April 1992).
[19] Myricom, Myrinet home page. <http://www.myri.com/>.
[20] J. Navaridas, J. Miguel-Alonso, F.J. Ridruejo, On synthesizing workloads emulating MPI applications, in: The Ninth IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08), Miami, Florida, USA, April 14–18, 2008.
[21] F. Petrini, W. Feng, A. Hoisie, S, Coll, E. Frachtenberg, The quadrics network: high-performance clustering technology, in: IEEE Micro 22, 1 (Jan. 2002), pp. 46–57. http://dx.doi.org/10.1109/40.988689.
[22] F, Petrini, M. Vanneschi, k-ary n-trees: high performance networks for massively parallel architectures, in: Proceedings of the 11th International Parallel Processing Symposium, IPPS'97, Geneva, Switzerland, 1997, pp. 87–93.
[23] F.J. Ridruejo, J. Miguel-Alonso, INSEE: an interconnection network simulation and evaluation environment, in: Proceedings of the Euro-Par 2005, Lecture Notes in Computer Science, vol. 3648, 2005, pp. 1014–1023.
[24] F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso , C. Izu, Realistic evaluation of interconnection network performance at high loads, in: 8th International Conference on Parallel and Distributed Computing Applications and Technologies – PDCAT 2007, Adelaide, Australia, 3–6 December 2007.
[25] G. Rodriguez, R. Beivide, C. Minkenberg, J. Labarta, M. Valero, Exploring pattern-aware routing in generalized fat tree networks, in: Proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY, USA, 2009, pp. 276–285, doi:10.1145/1542275.1542316.
[26] Sandia National Labs, "ASCI Red". <http://www.sandia.gov/ASCI/Red/>.
[27] Sun Microsystems, Inc. "N1 Grid Engine 6 User's Guide". <http://docs.sun.com/app/docs/coll/1017.3>.
[28] R. Thakur, W. Gropp, Improving the performance of collective operations in MPICH. <http://www-unix.mcs.anl.gov/~thakur/papers/mpi-coll.pdf>.