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.

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.

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

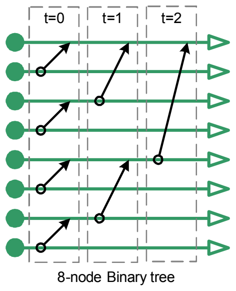

Fig. 1. Algorithmic and graphical definitions of Binary Tree pattern.

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

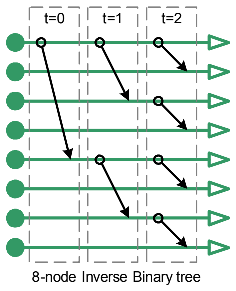

Fig. 2. Algorithmic and graphical definitions of Inverse Binary Tree pattern.

```
$\forall node\ in\ [0,N):$
$Butterfly\ (node,S):$
    $for\ t\ in\ [\ 0, \log_2 N\ )$
        $Computation()$
        $if\ \left(\left\lfloor node/2^t \right\rfloor \bmod 2\right) == 0\ then$
            $Send\_to(node+2^t, length(t,S))$
            $Wait\_from(node+2^t, length(t,S))$
        $else$
            $Send\_to(node-2^t, length(t,S))$
            $Wait\_from(node-2^t, length(t,S))$
        $endif$
    $endfor$
```

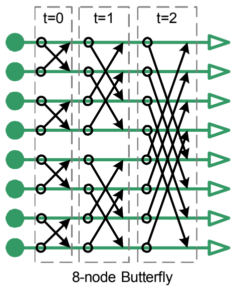

Fig. 3. Algorithmic and graphical definitions of Butterfly pattern.

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.

```
$\forall node\ in\ [0,N)$:
$Wavefront\ (node,S,D)$:
  $Computation()$
  $for\ d\ in\ [0,D)$
    $Wait\_from(Neighbor(node,d,'-'),S)$
  $end\ for$
  $for\ d\ in\ [0,D)$
    $Send\_to(Neighbor(node,d,'+'),S)$
  $end\ for$
```

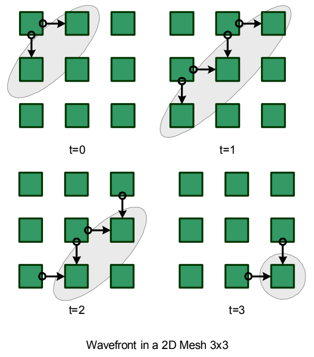

Fig. 4. Algorithmic and graphical definitions of 2D Wave-Front pattern.

```
$\forall node\ in\ [0,N)$:
$Distribution\ (node,S,D)$:
  $Computation()$
  $for\ d\ in\ [0,D)$
    $for\ w\ in\ \{'+','-'\}$
      $Send\_to(Neighbor(node,d,w),S)$
    $end\ for$
  $end\ for$
  $for\ d\ in\ [0,D)$
    $for\ w\ in\ \{'+','-'\}$
      $Wait\_from(Neighbor(node,d,w),S)$
    $end\ for$
  $end\ for$
```

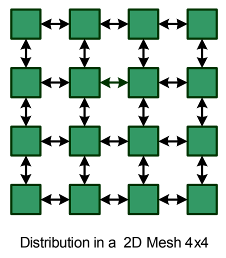

Fig. 5. Algorithmic and graphical definitions of 2D Distribution pattern.

```
$\forall node\ in\ [0,N)$:
$Direction\_Distribution\ (node,S,D)$:
  $Computation()$
  $for\ d\ in\ [0,D)$
    $for\ w\ in\ \{'+','-'\}$
      $Send\_to(Neighbor(node,d,w),S)$
      $Wait\_from(Neighbor(node,d,w),S)$
    $end\ for$
  $end\ for$
```

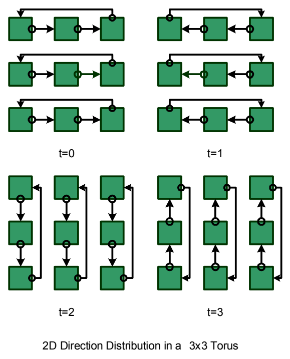

Fig. 6. Algorithmic and graphical definitions of 2D Dir. Dist. pattern.

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-

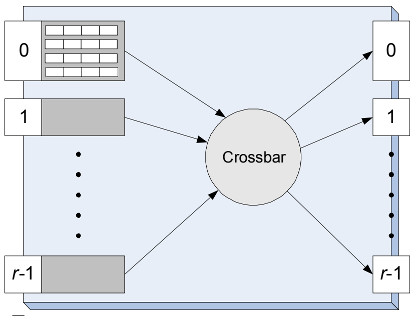

Fig. 7. Model of the switch (radix *r*). Four virtual channels sharing port 0 are shown.

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.

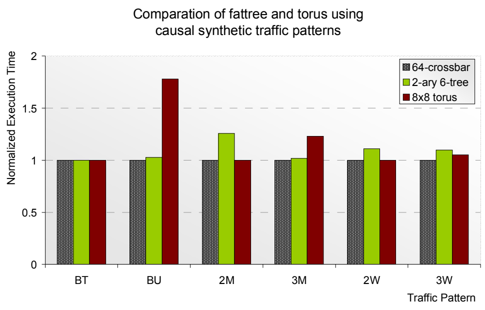

Fig. 8 Times to deliver all the messages for each topology and traffic pattern (*normalized* to crossbar)

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**).

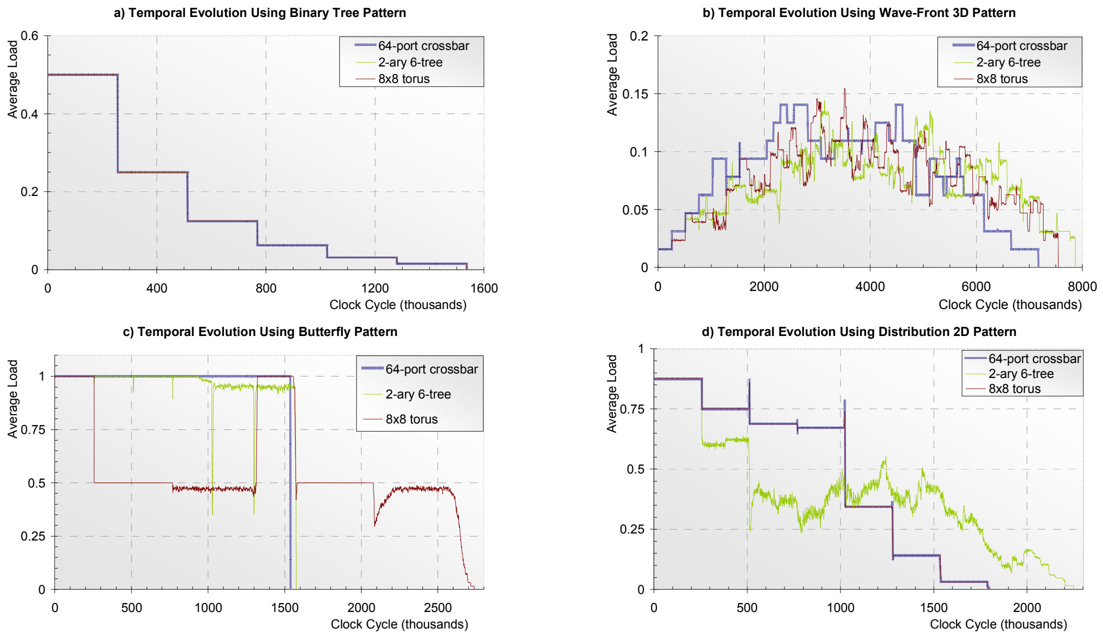

Fig. 9. Temporal evolution of the average consumed load measured in phit/cycle/node for different traffic patterns: a) **BT** pattern. b) **3W** pattern. c) **BU** pattern. d) **2M** pattern.

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