Interconnection network simulation using traces of MPI applications

Technical Report EHU-KAT-IK-01-08

J. Miguel-Alonso*, J. Navaridas, F.J. Ridruejo,
{j.miguel, javier.navaridas, franciscojavier.ridruejo}@ehu.es
Dep. of Computer Architecture and Technology, The University of the Basque Country,
P.O. Box 649, 20080 San Sebastian, Spain

Abstract

We discuss a collection of techniques, included in our INSEE simulation environment but applicable to other contexts, designed to do simulation-based performance studies of parallel computing systems using 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 (namely, lack of accuracy in timing information and invisibility of message interchanges in collective operations), and explain the way we circumvent some of those limitations. After that, we describe in detail the way traces are 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, and also how this mechanism can be extended to carry out performance predictions for different compute nodes or networks. To illustrate the usefulness of these techniques, INSEE is used to carry out two simple, example studies using traces from actual applications.

Keywords

Interconnection network simulation; traces of parallel applications; message passing interface

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

* Contact author. Tel. +34 943 018019, Fax +34 943 015590

being evaluated (which 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, because 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 [17, 25]. 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, a cluster of 10 PCs can be used to generate a trace of a parallel application running on 200 nodes—we only need to run twenty 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 packets are valid, and the distribution of packet sizes is valid too.

In this paper we discuss the mechanisms included in INSEE [24], an in-house developed simulation environment for the evaluation of interconnection networks, to perform simulation-based performance studies of parallel computing systems. INSEE is able to accept many kinds of workloads for the simulation (from synthetically generated messages to full system simulation),

but we will focus on the utilization of 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 are 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.

The rest of this paper is organized as follows. Section 2 introduce 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 6 we use INSEE to make two performance-related studies. Section 7 reviews some related work. Conclusions of the paper are summarized in Section 8.

2 Evaluation of networks using simulation: INSEE

In this section we briefly introduce INSEE [24], 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 [23].

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 causes 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 [18], 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 [17]. 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) [13], 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 tandem. 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 of 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 in a given state, the second when the process exits from it. States are defined in 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 of 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" node (destination for a send, source for a receive)
  - A message tag
  - Message size

From now on, we will use this way of referencing trace records: **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 [30], distributed with MPE. Fig. 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 ones.

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 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.

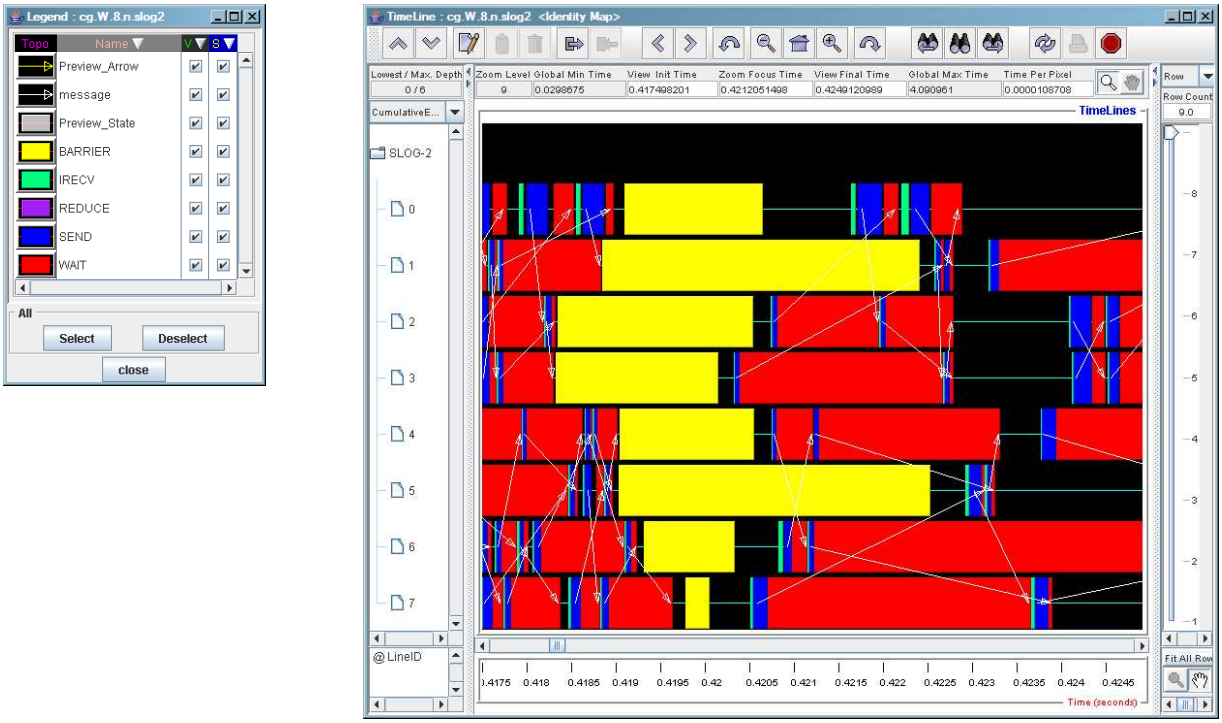

Fig. 1. Screenshot of Jumpshot visualizing a trace file generated by CG.W.8

Regarding collective operations, they 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 information 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 internals of the default implementation of collectives is 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.

With the modified MPICH, the generated, extended trace files follow the scheme described before, but they are longer because they include more detail. When visualized with 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 green boxes inside the yellow ones represent the way

MPI_Barrier is implemented 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. Regarding the second limitation of normal trace files (information of message interchange spread into state and message records), it is still applicable to extended trace files.

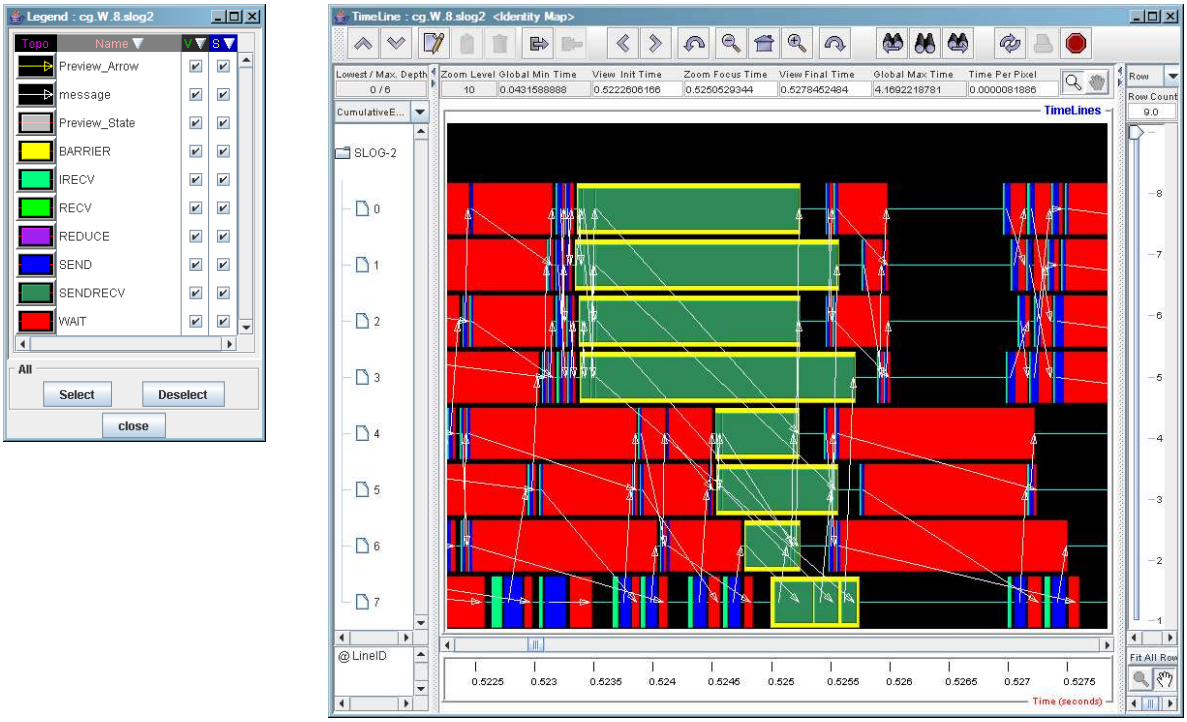

Fig. 2. Screenshot of Jumpshot visualizing an extended trace file generated by CG.W.8.

4 Using traces to feed simulations

In this section we describe how trace files can be used to provide realistic communication workloads to simulators of interconnection networks. In this discussion, we make these assumptions:

- Extended trace files are available.
- 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 interchange of packets. Applications generate/consume messages of variable sizes, which need to be split into fixed or variable-sized packets. 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.

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 in the network the messages of its list, as fast as the network accepts them. Network backpressure is used to modulate the injection of load into the network.

As we can see, timestamps are ignored except 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 implementations, 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 wanted 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 unaffordable.

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 in reproducing the *temporal* pattern, which should respect message causality and reflect the actual way message interchanges are interleaved, as required by the application.

4.2 Second approach: follow causal order

As we have just stated, 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 for the reception of a new message. Process execution is only resumed when the expected message arrives. We may emulate this behavior following this approach:

1. Ignore all state records – in other words, use only MS and MR records.
2. Split the trace file in one list per simulated node, in timestamp order. 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 ignored, except to order records.
3. Create, at each node, a reception 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 the corresponding message (matching origin, destination, tag and size) is in the reception list. If it is there, remove both entries. Otherwise, do nothing.
    c. When the simulator delivers a message, put it in the reception 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 #1, even if a message from #2 has been received already. In contrast, node #1 can advance because the required message from #0 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.

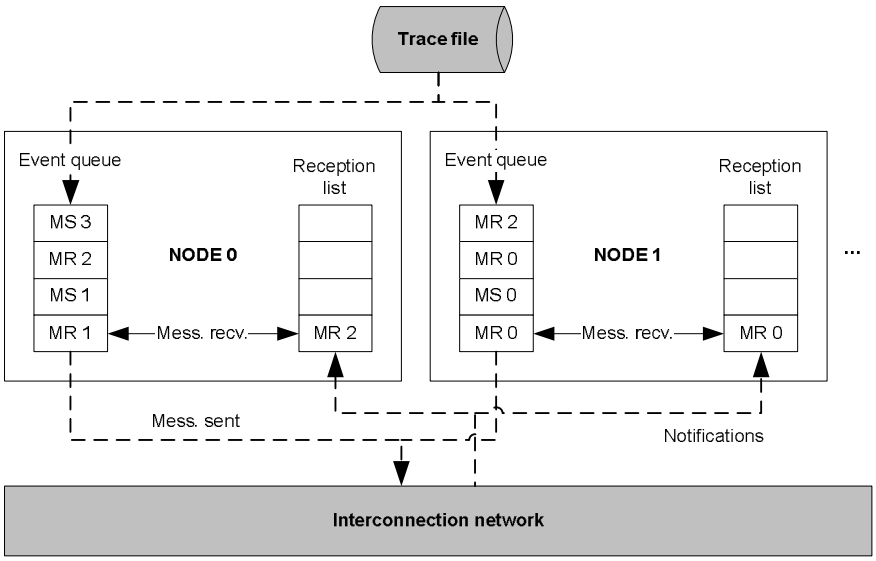

Fig. 3. Interface between the trace file and the network simulator. In the boxes, "MS *n*" means that a message is sent to node #*n*, and "MR *n*" means that a message is expected from node #*n*.

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 #2, it is possible to send a message to #3, and it happens that the message has been received already. However, node #0 is stalled (waiting for a message from #1). We may wonder if application semantics is adequately emulated. Is it *really* necessary to receive the message from #1 before advancing?

There is not a single answer to this question: it has to be discussed in 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 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*." [29]. We further discuss this issue in Section 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 the 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 Fig. 1 and Fig. 2. In the figures we can see "empty" spaces between MPI states. These spaces represent the time spent by processes *outside* MPI calls. We can assume that, during that time, the processes are doing useful CPU work, although this is not totally accurate. This assumption is only valid if there is a one-to-one relationship between process and CPU, and we ignore the side-effects of other activities performed by the CPU – otherwise the empty states may correspond to times in which processes have been context-switched and do not have access to the CPU. Under these assumptions, a simulator can be fed with the traces and a set of parameters that define the CPU characteristics, as well as the network characteristics.

The simulator runs as described in Section 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 do 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. 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 messing up with unnecessary details. We consider that this subset is still valid, 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 a process #*n* invokes the MPI_*X* function; an <*n ts* SE MPI_*X*> record is generated when this function returns. MS and MR records (about 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 reception are not logged when they happen*.

- An MS record is logged *after* the process has entered into a state in which it request a message being sent (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 different reasons: semantic of MPI operations (immediate operations), previous messages queued, network congestion, decision of the kernel's scheduler, etc.

- An MR record is logged *once* 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 *before* the process exits from that state. It may happen that a message has been received from the network interface long ago, but this reception is not logged until the receiving process has entered a waiting state.

This behavior is clearly visible in Fig. 1 and Fig. 2. Note that MPI_Irecv states can be safely ignored, because the actual reception of a message is recorded in a subsequent MPI_Wait; there are no arrows arriving to the light-green 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 declared in the trace file, and the actual reception of a message may have taken place before the time indicated by the record timestamp – sometimes, long before.

5.2 Receptions from MPI_ANY_SOURCE

A *connoisseur* of MPI programming 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 #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 reception is from process #3, while the other one may contain any (valid) process identifier. Note that state records do not contain any indication of the message source, so from now on we will ignore them.

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, in such a way that 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 programmer could program 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:

```
…
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.

Code Version A forces an unnecessary reception order, which may delay program progress. For example it may happen that a message from slave #2 is already buffered, but no one from slave #1 has been received yet. Program is stalled in the first sentence of the loop, even when the third (reception of a job request from #2) and fourth (sending a task to #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 reception side would allow 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 **`Trace a`** of Fig. 5. It is easy to understand that a record reflecting that #1 sends a message to #0 (<1 *ts1* MS 0>) must be recorded in 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 anywhere before record *ta4*, to match with record *ta3*. Note, again, that the actual reception from node #2 could have happened before the reception from #1, but the trace file would not reflect this circumstance.

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.

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 reception. 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.

We claim that we should not use wildcards in the trace files, and must stick to the sequence of events actually stored. This may not be the only valid sequence of events but, at least, we know that this is a semantically valid one.

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 [17] we cross-validated the trace-driven simulation with an execution-driven environment based on Simics; and in [26] we study 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) in 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 of a cluster.

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 interest of using adaptive routing. Fig. 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 X+0, X+1, X+2.

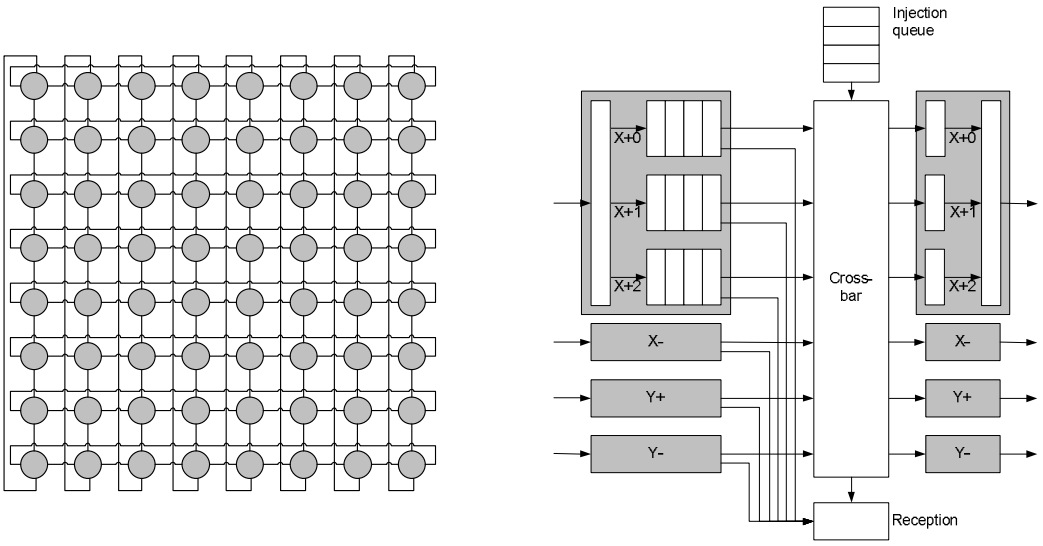

Fig. 6. Left: an 8-ary 2-cube (2D torus with 64 nodes). Right: model of the router simulated by FSIN.

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 the column where the destination is. Then, it has to move in the Y axis (up or down in the column of the destination) until reaching the node where it has to be consumed.

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 [22] 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 different 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 in Fig. 7 relative values; 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.

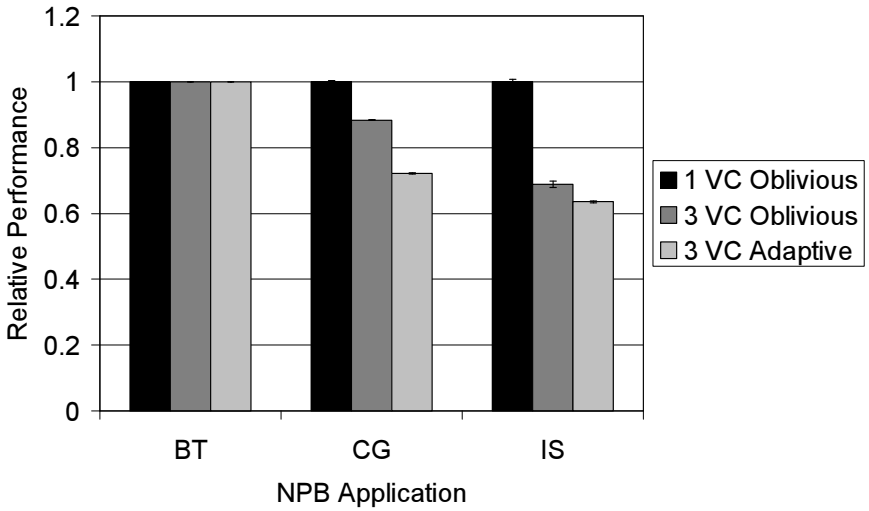

Fig. 7. Effects of using 3 VC per physical link, and of adding adaptivity. Results relative to the base case (Oblivious 1 VC). Average of 10 simulation runs, and 99% confidence intervals.

From the obtained results, we can find out 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. 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 it 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.

6.2 Estimating execution times

In Subsection 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, whose interconnection network is a fat-tree implemented with Myrinet-2000 adapters and switches. This network operates at 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, 8x8 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 these three target architectures, are summarized 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	8x8 torus 100 Mb/s	8x8 torus 1 Gb/s	8x8 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 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 10K nodes of this computer. The 64 CPUs used in the experiment were not consecutive, but located in different portions of the tree's

leaves. 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 interest. 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 [14] 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 capturing 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. This approach would provide better timing information. An example of this approach is Sun's Dtrace [19].

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 [28] 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 they are actually available to the corresponding application processes. 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): as 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 Extended vs. regular traces

In Section 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. A fair assessment of a computer with support for collectives should be done using regular trace files. A well-known machine *with* this support is the BlueGene [1,4]. In contrast, clusters built around Myrinet [15] networks *do not* include this 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 [16]) 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.

7.3 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 a much high level of 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 – 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 CPU, when they are blocked by other tasks (when sharing CPU) and when they are stalled for I/O operations.

7.4 Other simulation tools

We end this section with a review of simulation tools for interconnection networks. We start with SICOSYS [20], developed at the University of Cantabria. It 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 feed with synthetic workloads and application traces, and can also be integrated with other tools to perform full system simulation.

The Flexim 1.2 simulator [27], 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 detail of the involved mechanism 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 to INSEE, SICOSYS, or Flexim. It works alongside with BigSim, a system emulator able to run applications specifically compiled for it. The emulator captures a collection of tasks 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 which can be used to analyze its performance on the target machine.

8 Conclusions

In this paper we have introduced a collection of tools and techniques necessary 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 incorporate 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 relationship 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 example of 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 in a collection of target architectures using traces captured in a real machine.

For future work, we plan to investigate further into how to integrate application semantics into simulation, to allow different (but valid) event orderings. Meanwhile, the best way of dealing with this issue is the utilization of execution-driven evaluation environments—but only if it is powerful enough to allow for the simulation of networks of interest.

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 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. G. Almási 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, 2005). ICS '05. ACM Press, New York, NY, 253-262.
2. D. H. Bailey, T. Harris, R. Van der Wigngaart, W. Saphir, A. Woo, and M. Yarrow. "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
4. R. Brightwell. "A Comparison of Three MPI Implementations for Red Storm". 12th European PVM/MPI Conference. September 2005, Lecture Notes in Computer Science, Volume 3666, Oct 2005, Pages 425 – 432,
5. Cámara J., Moretó M., Vallejo E., Beivide R., Martinez C, Miguel J. and Navaridas J. "Mixed-radix Twisted Torus Interconnection Networks". Proc. 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., and 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. W.J. Dally, B. Towles. "Principles and Practices of Interconnection Networks". Morgan-Kaufmann, 2004.
8. European Center of Parallelism of Barcelona / Technical University of Catalonia. Dimemas tool. http://www.cepba.upc.edu/dimemas/

9. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. “A high-performance, portable implementation of the MPI Message-Passing Interface standard”. *Parallel Computing*, 22(6):789-828, 1996.
10. E. Karrels and E. Lusk, “Performance Analysis of MPI Programs,” Proc. of the Workshop on Environments and Tools for Parallel Scientific Computing, 1994.
11. P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hillberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” Computer, vol. 35, no. 2, pp. 50-58, Feb. 2002.
12. D.A. Menasce and L.A. Barroso, “A methodology for performance evaluation of parallel applications on multiprocessors,” Journal of Parallel and Distributed Computing, vol. 14, pp. 1-14, 1992.
13. Message Passing Interface Forum. “MPI: A Message-Passing Interface Standard”. University of Tennessee.
14. S. Moore, D. Cronk, K. London, and J. Dongarra, “Review of Performance Analysis Tools for MPI Parallel Programs”, pp 241-248, 8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131, Editors, Yiannis Cotronis and Jack Dongarra, Springer Verlag, Berlin, 2001.
15. Myricom Home Page. Available at http://www.myri.com
16. 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
17. 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, March 26-28, 2007.
18. J. Navaridas, F. J. Ridruejo, J. Miguel-Alonso. "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.
19. OpenSolaris Community: Dtrace. Available at http://opensolaris.org/os/community/dtrace/
20. V. Puente, J.A. Gregorio, R. Beivide, "SICOSYS: An Integrated Framework for studying Interconnection Network Performance in Multiprocessor Systems," Proc. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP 2002), 2002
21. V. Puente, J.M. Prellezo, C. Izu, J.A. Gregorio and R. Beivide, “A Case Study of Trace-driven Simulation for Analyzing Interconnection Networks: cc-NUMAs with ILP Processors”, Proceedings of the IEEE 8th Euromicro Workshop on Parallel and Distributed Processing. Rhodes, Greece. January 2000.
22. V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo and J.M. Prellezo, The Adaptative Bubble Router, Journal of Parallel and Distributed Computing. Vol 61 - nº 9, September 2001
23. F.J. Ridruejo, A. Gonzalez, J. Miguel-Alonso. "TrGen: a Traffic Generation System for Interconnection Network Simulators". 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.
24. F.J. Ridruejo, J. Miguel-Alonso. "INSEE: an Interconnection Network Simulation and Evaluation Environment". Lecture Notes in Computer Science, Vol. 3648 / 2005 (Proc. Euro-Par 2005), Pages 1014-1023.
25. F.J. Ridruejo, J. Miguel-Alonso, J. Navaridas. "Concepts and Components of Full-System Simulation of Distributed Memory Parallel Computers". Proc. HPDC’07, June 25–29, 2007, Monterey, California, USA.

26. F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso, Cruz Izu. “Realistic Evaluation of Interconnection Network Performance at High Loads”. Proc. 8th Int. Conf. on Parallel and Distributed Computing Applications and Technologies - PDCAT 2007, Adelaide, Australia, 3-6 December 2007.
27. SMART group at the U. of Southern California. Information on FlexSim1.2. Available at http://ceng.usc.edu/smart/FlexSim/flexsim.html
28. Wireshark home page. Available at http://www.wireshark.org/
29. F. Wolf, A. Malony, S. Shende, A. Morris. “Trace-Based Parallel Performance Overhead Compensation”, Proc. Int. Conf. on High Performance Computing and Communications (HPCC), Sorrento, Italy, Sept., 2005.
30. O. Zaki, E. Lusk, W. Gropp, and D. Swider. “Toward scalable performance visualization with Jumpshot”. High Performance Computing Applications, 13(2):277-288, Fall 1999.