Single Node On-Line Simulation of MPI Applications with SMPI

Pierre-Nicolas Clauss*, Mark Stillwell†, Stéphane Genaud‡, Frédéric Suter§, Henri Casanova†, Martin Quinson*

*Nancy University, LORIA, INRIA, Nancy, France
†Department of Information and Computer Sciences, University of Hawai'i at Mānoa, Honolulu, U.S.A.
‡University of Strasbourg, ICPS-LSIIT, Illkirch, France
§IN2P3 Computing Center, CNRS, IN2P3, Lyon-Villeurbanne, France

***Abstract*—Simulation is a popular approach for predicting the performance of MPI applications for platforms that are not at one's disposal. It is also a way to teach the principles of parallel programming and high-performance computing to students without access to a parallel computer. In this work we present SMPI, a simulator for MPI applications that uses on-line simulation, i.e., the application is executed but part of the execution takes place within a simulation component. SMPI simulations account for network contention in a fast and scalable manner. SMPI also implements an original and validated piece-wise linear model for data transfer times between cluster nodes. Finally SMPI simulations of large-scale applications on large-scale platforms can be executed on a single node thanks to techniques to reduce the simulation's compute time and memory footprint. These contributions are validated via a large set of experiments in which SMPI is compared to popular MPI implementations with a view to assess its accuracy, scalability, and speed.**

***Index Terms*—Message Passing Interface; On-line simulation; Performance prediction.**

I. Introduction

This work introduces a new framework for simulating the execution of parallel applications implemented using the Message Passing Interface (MPI) [1] standard on a cluster. Simulation is a popular approach for predicting the performance of an application for a platform that is not available, for example because it is yet to be specified and purchased. Simulations can be used to determine a cost-effective hardware configuration appropriate for the expected application workload. Conversely, simulations can also be used to study the performance behavior of an application by varying the hardware characteristics of an hypothetical platform. In a classroom setting, students without access to a parallel platform could execute applications in simulation on a single node as a way to learn the principles of parallel programming and high-performance computing. Simulation of an application on a platform may also be useful even when the platform is available. For instance, the simulation may bypass actual computations performed by the application, and only simulate the corresponding delays of these computations. In this case the simulated application produces erroneous results, but its performance behavior may be preserved. It is then possible to conduct development activities for performance tuning in simulation only on a small-scale platform, thereby saving time when compared to real executions on a large-scale platform. Furthermore, access to large-scale platforms is typically costly (possible access charges to the user, electrical power consumption). The use of simulation can thus not only save time but also money and resources.

Three challenges for simulating MPI applications are:

1) *Accuracy:* Does the simulation match the real execution?
2) *Scalability:* Is it possible to simulate large applications executing on large-scale platforms?
3) *Speed:* Is it possible to have a low ratio of simulation time to simulated time?

Many simulation frameworks have been developed that attempt to address some or all of these challenges. Simulation approaches fall into two categories: *off-line simulation*, also called trace-based simulation or post-mortem simulation, and *on-line simulation*, also called simulation via direct execution. In off-line simulation a log of a previous execution of the application is "replayed" on a simulated platform. In on-line simulation the application is executed but part of the execution takes place within a simulation component. While both approaches have merit, on-line simulation is more general because the simulation is not tied to a log obtained for particular application and platform configurations.

In this work we present a simulator for MPI applications, SMPI (Simulated MPI), which relies on on-line simulation. SMPI is freely available and provided as part of the SimGrid project [2]. SMPI leverages the simulation existing techniques in SimGrid, but also contributes new ones. More specifically, this work makes these main contributions with respect to the MPI simulation literature:

1) SMPI simulations account for network contention; this is done without resorting to packet-level simulation but rather relying on analytical models, which makes the simulation of contention both fast and scalable;
2) SMPI implements a validated piece-wise linear model for data transfer times between two cluster nodes;
3) SMPI simulations of large-scale applications on large-scale platforms can be executed on a single node thanks to techniques for reducing the simulation's compute time, to techniques for reducing the simulation's memory footprint based on the work in [3], and to the use of a sequential but fast simulation kernel.

4) Experimental results demonstrate that SMPI simulations are accurate, scalable, and fast.

This paper is organized as follows. Section II reviews related work. Section III explains how SMPI simulation can be executed on a single node. Section IV explains how SMPI achieves accurate and fast network simulations. Section V describes SMPI's overall design and highlights key implementation details. Section VI explains how a platform specification can be instantiated for use with SMPI. Section VII presents experimental results. Finally, Section VIII concludes with a summary of results and future research directions.

II. Related Work

One option for simulating the execution of an MPI application is *off-line simulation*. A log, or trace, of MPI communication events (time-stamp, source, destination, data size) is first obtained by running the application on a real-world platform. A simulator then replays the execution of the application as if it were running on a *target platform*, i.e., a platform with hardware characteristics different from those of the platform on which the trace was obtained. This approach is used extensively, as shown by the number of trace-based simulators described in the literature since 2009 [4], [5], [6], [7], [8]. The typical approach is to compute the durations of the time intervals between MPI communication operations, or "CPU bursts". Upon replaying the application, the CPU bursts are modified to account for the performance differential between the platform used to obtain the trace and the target platform, either using simple scaling [4], [6], [7] or using a more sophisticated convolution between the application computational signature and the target platform's hardware signature [9]. Network communications are simulated based on the communication events recorded in the trace and on a simulation model of the network.

A challenge for off-line simulation is the large size of the traces, which can prevent running the simulation on a single node. Mechanisms have been proposed to improve scalability, including compact trace representations [7] and replay of a judiciously selected subset of the traces [8]. Another challenge is that the trace obtained on a given platform contains features that may not occur on another platform, in particular for irregular parallel applications that use asynchronous communications. Even for regular applications, if the application execution is defined by many parameters (e.g., block size, data distribution schemes), a trace may be needed for each parameter configuration. Finally, it is typically necessary to obtain the trace on a platform that has the same scale as the target platform. However, trace extrapolation to larger numbers of nodes than that of the platform used to obtain the trace is feasible in some cases [5], [6]. Most aforementioned off-line simulators run on a single node and, as noted in [6], could benefit from parallelization. The exception is [4], which requires a full-scale cluster but whose objective is closer to detailed performance debugging than to performance prediction.

One approach that avoids these particular challenges, but that comes with challenges of its own, is *on-line* simulation. In this approach, which we use in this work, the actual code of an MPI application, with no or marginal modification, is executed on a *host platform* that attempts to mimic the behavior of the *target platform*. Part of the instruction stream is then intercepted and passed to a simulator. LAPSE is a well-known on-line simulator developed in the early 90's [10] (see therein for references to precursor projects). In LAPSE, the parallel application executes normally but when a communication operation is performed a corresponding communication delay is simulated on the target platform using a simple network model (affine point-to-point communication delay based on link latency and bandwidth). MPI-SIM [11] builds on the same general principles, with the inclusion of I/O subsystem simulation in addition to network simulation. A difference with LAPSE is that MPI processes run as threads, which is enabled by a source code preprocessor (e.g., to privatize global variables). Another project similar in intent and approach is the simulator described in [12]. The BigSim project [13] also builds on similar ideas. However, unlike MPI-SIM, BigSim allows the simulation of computational delays on the target platform. This makes it possible to simulate "what if?" scenarios not only for the network but also for the compute nodes of the target platform. Simulation of computation delays in BigSim is done based either on user-supplied projections for the execution time of each block of code (as done also in [14]), or on scaling execution times measured on the host platform by a factor that accounts for the performance differential between the host and the target platforms, or based on sophisticated execution time prediction techniques such as those developed in [9]. The weakness of such approaches is that since the computational application code is not executed, the computed application data is erroneous. Consequently, application behavior that is data-dependent is lost. This is acceptable for many regular parallel applications, but can make the simulation of irregular applications (e.g., branch-and-bound) questionable at best. Aiming for high accuracy, the work in [15] uses a cycle-accurate hardware simulator of the target platform to simulate computation delays, which leads to a high ratio of simulation time to simulated time.

The complexity of the network simulation model has a high impact on speed and scalability, thus compelling many authors to adopt simplistic network models. One simplification, for instance, is to use monolithic performance models of collective communications rather than simulating them as sets of point-to-point communications [7], [16]. Another simplification used in most aforementioned simulators, whether off-line or on-line, is to ignore network contention because simulating it is known to be costly [17]. The work in [7] proposes the use of simple analytical models of network contention for off-line simulation. An exception is the MPI-NetSim on-line simulator [18], which provides full-fledged contention simulation via a packet-level discrete-event network simulator. As a result, the simulator may run more slowly than the application, which poses time coherence problems for on-line simulation. The solution in [18] is to slow down the entire system (i.e., inserting sleep calls during the application execution) so that the simulator has the time to simulate all network traffic without inducing timing skew. This approach has also been used in the general-purpose simulation environment MicroGrid [19]. Another exception is the PEVPM on-line simulator [14]. PEVPM relies on extensive benchmarks of the target platform that provide probability distributions of communication times, which can in turn be used to model network contention phenomena. Finally, note that two options for general-purpose on-line simulation are to reconfigure the cluster interconnect of the host platform to mimic that of the target platform [20], or to load the host platform with a judiciously chosen synthetic user-level workload [21].

One difficulty faced by most MPI-specific on-line simulators is that the simulation, because done via direct execution of the MPI application, is inherently distributed. Parallel discrete event simulation raises difficult correctness issues pertaining to process synchronization. For the simulation of parallel applications, techniques have been developed to speed up the simulation while preserving correctness (e.g., the asynchronous conservative simulation algorithms in [22], the optimistic simulation protocol in [13]). A way to side-step this difficulty is to run the simulation on a single node.

While in the off-line case it is often possible to run the simulation on a single node [8], doing so proves challenging in the on-line case as the simulation requires large amounts of CPU and RAM resources. For most aforementioned on-line approaches, the resources required to run a simulation of an MPI application are commensurate to those of that application. In some cases, those needs can even be higher (e.g., an extra node to run the network simulation component [18], costly cycle-accurate simulation of the application's code [15]). One way to reduce the CPU needs of the simulation is to avoid executing computational portions of the application and simulate only expected delays on the target platform [13], [14]. Reducing the need for RAM resources is more difficult. For instance, simulations in [14], which run on a single-node, are for applications with small memory footprints. In general, if the target platform is a large cluster, then the host platform must also be a large cluster. However, a solution proposed in [3] consists in removing large data arrays from the simulation with the help of the compiler. When doing so, the modified application produces erroneous results. But, for non-data-dependent applications RAM usage reductions up to 4 orders of magnitudes are reported. In this work we use the same techniques as in [3], which we detail in Section III-B.

III. Single-Node On-line Simulation

One of our goals is for SMPI to be able to simulate an application on a large target platform while using a single node as the host platform. While the motivation for single-node simulation is clear (see Section I), single-node simulation is a challenging proposition for on-line simulation (see Section II). The challenges are the CPU requirements and the RAM requirements of the simulated application. We address both challenges as follows, considering that the host platform is a single cluster node, which we call the "host node."

A. Reducing CPU requirements

In general, the amount of time needed to execute the computational portions of the application's code, or CPU bursts, on a single node is proportional to the number of nodes of the target platform that are used by the application. As in [13], [14], we opt to replace each CPU burst in the simulation by the corresponding expected delay on a target platform node. Thus, unless the application's execution is data-dependent, the computational time of the application when it runs in simulation could be negligible. The main question, however, is how to determine the delay of each CPU burst on the target platform. We allow for the execution of each CPU burst only the first n times the CPU burst occurs, and then using the average delay computed over these n samples as the delay in the simulation for future occurrences of this CPU burst. Using $n > 1$ is useful for CPU bursts that exhibit execution time variations, e.g., due to application data. Since the CPU bursts are measured on the host node, their durations are used directly for simulating a target platform comprised of nodes identical to that of the host node. Otherwise, we simply allow the user to specify a factor by which CPU burst durations can be scaled to account for a performance differential between the host node and the nodes of the target platform. We allow for $n = 0$, in which case the user must supply a number of flops (which is then transformed into a delay using the aforementioned factor) for simulating the corresponding CPU burst on the target platform.

The time to execute each CPU burst $n > 0$ times on the node platform is proportional to the number of nodes in the target platform, since each MPI process executes each CPU burst n times. The scalability of this approach may thus not be acceptable because simulation time increases linearly with the number of simulated nodes. In many parallel applications, computations are regular, meaning that the MPI processes execute identical or similar CPU bursts. This is the case, for instance, for most applications using the SPMD (Single Program Multiple Data) paradigm. Therefore, SMPI allows the measurement of the execution times of the first n CPU bursts on any MPI process. The simulation time of application computation is then independent on the number of nodes in the target platform, and thus scalable. Section V-B provides technical details on how this feature is implemented in SMPI.

For applications that are irregular or data-dependent, replaying previously measured CPU burst durations may not lead to accurate results. In the worst case, all CPU bursts would need to be executed. In this case, single-node simulation would suffer from severe scalability issues. Should these issues endanger the applicability of the approach, one would have to face the challenges of developing a parallel discrete event simulator that can be executed on a cluster, as done for instance in MPI-SIM or MPI-NetSim.

B. Reducing RAM requirements

A problem with single-node on-line simulation is that the memory footprint of the application cannot be accommodated on the host node unless the number of nodes in the target platform is small and/or the application's footprint is small.

In an SMPI simulation, all MPI processes run as threads that share the same address space. In this case, two techniques are proposed in [3] for removing large array references:

Technique #1: Because MPI processes run as threads, references to local arrays can be replaced by references to a single shared array. If the MPI application has m processes that each use an array of size s, then the RAM requirement is reduced from $m \times s$ to s.

Technique #2: Because a CPU burst is simulated by replaying a delay rather than by executing its code, memory references in that code can be removed, which can lead to the removal of potentially large, now unreferenced, arrays.

Both techniques are implemented in SMPI, but the second one can only be used if n, the number of measurements of each CPU burst duration, is 0. In this case, as in [3], CPU burst durations are user-provided and the CPU burst code is effectively removed.

IV. Accurate and Fast Network Simulation

One of the goals of SMPI is to perform accurate simulation of both point-to-point and collective MPI communications. One option is to use a packet-level discrete event network simulator, as in MPI-NetSim [18]. The drawback is that packet-level simulation is neither fast nor scalable. For instance, simulation time grows roughly linearly with message size. Simulation time can then be longer than simulated time, which poses difficulties for preserving the coherence of an on-line simulation. Not surprisingly, most of the simulators reviewed in Section II have opted for not (fully) realistic, but simple, analytical network models. For the same reasons SMPI also uses an analytical network model that can be computed fast and scalably. But this model is more accurate than analytical models used in previously developed MPI simulators.

A. Point-to-Point Model

All on-line MPI simulators reviewed in Section II use the standard affine model defined by a latency and bandwidth parameter. In this model the time to transfer a message of size s in bytes from one node to another is $\alpha + s/\beta$ where α is the network latency in seconds and β the bandwidth in bytes/sec. Unfortunately, this model fails to capture the behavior of real-world cluster interconnects using TCP and popular MPI implementations (e.g., a OpenMPI [23] over a Gigabit Ethernet switch). For instance, a message under 1 KiB fits within an IP frame, in which case the achieved data transfer rate is higher than for larger messages. Also, MPI implementations for `MPI_Send()` typically switch from buffered to synchronous mode above a certain message size. This is seen in OpenMPI or MPICH2 [24], for instance. Consequently, instead of being an affine function of message size, communication time is *piece-wise linear*.

SMPI models point-to-point communication times with a piece-wise linear model with an arbitrary number of linear segments. Each segment is obtained using linear regression on a set of real measurements. The number of segments and the segments boundaries are chosen such that the product of the correlation coefficients is maximized. In practice, we find that the model should be instantiated for 3 segments, leading to 8 parameters defining the model (2 for defining the boundaries of the 3 segments, and one latency and bandwidth parameter for each segment). Some of these parameters are actually dependent on each other and it would be possible to define the model using only 6 parameters. Regardless, the number of parameters is much higher than that for the simplistic affine model, making model instantiation more challenging. We discuss model instantiation in Section VI.

B. Contention Model

Among the works reviewed in Section II, only [7] mentions an analytical model for network contention. However, few details are given and all results therein are for a simple model that does not account for contention. In this work we reuse the analytical network contention model implemented as part of the SimGrid [2] simulation framework. This model was developed for arbitrary network topologies with end-points that use standard network protocols, such as TCP/IP, and are connected via multi-hop paths. This model is thus applicable to networks ranging from simple switches in clusters to wide-area networks. Instead of being packet-based, the model is flow-based, meaning that at each instant the bandwidth allocated to an active flow (i.e., a data transfer occurring between two end-points) is computed analytically given the topology of the network and all currently active flows. This model is described and validated via comparison to the GTNetS packet-level simulator in [25], [26].

We have implemented the piece-wise linear point-to-point model described earlier in SimGrid so that it can be combined with its network contention model. This leads to an immediate simulation model for collective communication operations. Just like in any MPI implementation, collective communications are implemented in SMPI as sets of point-to-point communications that may experience network contention among themselves. This is to be contrasted with monolithic modeling of collective communications [7], [16].

V. SMPI Design and Implementation

A. Overall Design

SMPI is implemented as one of SimGrid's APIs, and as such is built on top of SimGrid's internal simulation API, called SIMIX. SIMIX provides access to the simulation kernel, SURF, in which simulation models are implemented. SMPI supports MPI applications written in C, which must be linked to the SMPI library in order to execute in simulation. This software organization is depicted in Figure 1. In its current implementation SMPI implements the following subset of the MPI standard:

- error codes, predefined datatypes, and predefined and user-defined operators;
- process groups, communicators, and their operations (except Comm_split);
- these point-to-point communication primitives: Send_Init, Recv_Init, Start, Startall, Isend, Irecv, Send, Recv,

Sendrecv, Test, Testany, Wait, Waitany, Waitall, and Waitsome;
- these collective communication primitives: Broadcast, Barrier, Gather, Gatherv, Allgather, Allgatherv, Scatter, Scatterv, Reduce, Allreduce, Scan, Reduce_scatter, Alltoall, and Alltoallv.

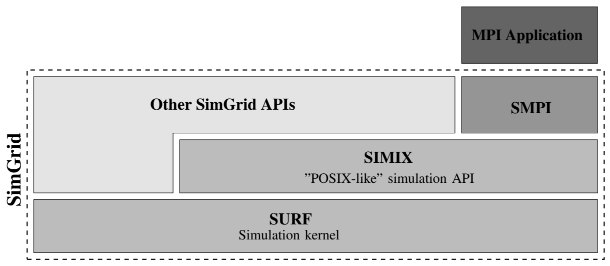

Figure 1. SIMGRID internal structure.

```
1  ...
2  double *data =
3      (double*)SMPI_SHARED_MALLOC(...);
4  MPI_Init(...);
5  SMPI_SAMPLE_LOCAL(10)
6  {
7    < Some computation (A) >
8  }
9  SMPI_SAMPLE_GLOBAL(10)
10 {
11   < Some computation (B) >
12 }
13 SMPI_SAMPLE_DELAY(1048576)
14 {
15   < Some computation (C) >
16 }
17 < Some computation (D) >
18 MPI_Finalize();
19 SMPI_FREE(a);
20 ...
```

Figure 2. SMPI macro usage example.

An SMPI simulation runs in a single process, with each MPI process running in its own thread. However, these threads run sequentially, under the control of the SIMGRID simulation kernel. This simulation kernel is thus fully sequential. This is a design choice due to the known challenges of parallel discrete event simulation [13], [22]. The potential drawback of a sequential kernel is that simulation time may increase drastically with the scale of the simulation. However, SIMGRID relies on the analytical simulation models implemented in SURF that can be computed quickly, leading to scalable simulation capabilities.

B. Application Source Code Modifications

Some modifications to the code of the MPI application may be needed for execution with SMPI. A well-known problem for on-line simulation when running MPI processes as threads is the management of global variables. These variables need to be privatized so that each thread has its own copy, which is addressed in [11] via the use of a source code preprocessor. The solutions described in Section III-A to reduce CPU requirements and in Section III-B to reduce RAM requirements also require modification of the application's code. For CPU reduction, the code for a CPU burst must be bypassed after n executions of it, either within a single thread or over all threads. This can be enabled easily by a code preprocessor that takes n as input and inserts global counters and if-then-else statements around the code for each CPU bursts. For RAM reduction, some array references must be equated so that fewer individual arrays are referenced. This is more challenging, but a compiler-based approach is proposed and developed in [3]. Consequently, all aforementioned source code modifications can be handled by the compiler (e.g., by a modified mpicc).

Our current focus is on simulation accuracy and scalability. Improved usability through the development of a pre-processor or compiler will be addressed in a later phase of this work, building on the results in [3], [11]. For the time being, we let the user handle global variable privatization by hand, and provide macros designed for the standard C preprocessor for CPU and RAM requirement reductions. Figure 2 shows a source code sketch that utilizes these macros. At line 2 array `data` is allocated using the `SMPI_SHARED_MALLOC` macro. This macro allows the array to be allocated only once and to be shared by all simulated MPI processes. Similarly, at line 19, the `SMPI_FREE` macro is used so that the array is freed only once. At line 5, the `SMPI_SAMPLE_LOCAL` macro is used to indicate that the following CPU burst (in between curly braces) should be executed and timed 10 times by *each* MPI process, and subsequently bypassed and replaced by a simulation of a delay equal to the average of the 10 measured execution times. At line 9, the `SMPI_SAMPLE_GLOBAL` macro is similar but the CPU burst is measured only 10 times in total (possibly when executed by 10 different MPI processes), before its execution is bypassed and replaced by an average delay. At line 13, the block of code following the `SMPI_SAMPLE_DELAY` is never executed but instead replaced in the simulation by the given amount of flops. All these macros are expanded into one or more calls to various functions that look up and update hash tables where each entry contains a unique identifier (based on source file name and line number), execution counters, reference counters, and/or pointers to user arrays.

C. Which MPI Implementation?

One important question when designing an on-line MPI simulator is that of the MPI implementation simulated. Different MPI implementations employ different algorithms, with notable differences including various data distribution schemes for collective communications and different algorithmic choices depending on message length. One option is to build a simulation back-end that can be used directly by an existing MPI implementation. For instance, in [18], the simulator is implemented as an MPICH device for the MPICH2 implementation of MPI [24]. Similarly, an MPICH device could be developed on top of SIMIX (the generic SIMGRID internal simulation API shown in Figure 1). The advantage is that the simulator should be easily evolvable to accommodate future releases of the MPI implementation. This approach may be applicable to other MPI implementations, such as OpenMPI [23], but we are not aware of any simulator implemented as a back-end to OpenMPI.

Another option is to copy the implementation of each MPI primitives and modify it to integrate it in the simulator. This "cut-modify-and-paste" approach is feasible only if the implementation of each MPI primitive is relatively self-contained. After examining the MPICH2 and the OpenMPI implementations, we have found this approach to be particularly straightforward using MPICH2. For each collective communication implementation, the integration of the implementation within the simulator can be done in a few minutes. However, SMPI also implements algorithms implemented in OpenMPI (for many-to-many operations, mainly). It is worthwhile to note that there is no unique algorithm for any collectives operations, each variant being best in particular settings. SMPI currently only implements one variant for each operation. Future versions will provide multiple variants, letting users choose which ones to use in the simulation.

VI. Target Platform Instantiation

An SMPI simulation takes as input the number of MPI processes, their command-line arguments, and a specification of the target platform. This specification is written in XML using SimGrid's Document Type Definition (DTD). In the context of SMPI, the specification contains descriptions of the cluster nodes, including a performance indicator measured in Flop/s. Then, the performance of the host node is given as argument to the simulation program, allowing to scale the timings obtained on the host node to what would be experienced on the nodes of the target platforms. The specification also lists network elements, which are on paths between cluster nodes. As explained in Section IV-A, the performance of point-to-point communications on these links is described by 8 parameters. While the values of these parameters can be chosen arbitrarily by the SMPI user, it is likely difficult to simply pick reasonable values. This is why we *calibrate* the SMPI simulation by automatically instantiating these parameters based on point-to-point experiments executed on two nodes of one or more real-world clusters. It is then possible to modify this instantiation to run simulations for "what if?" scenarios (e.g., simulate a network that achieves 30% higher data transfer rate for large messages).

We use the freely available SKaMPI [27] benchmarking framework to perform simulation calibration. Using the simple ping-pong MPI benchmark provided by SKaMPI, we obtain data transfer times achieved for a wide range of message sizes. We can then automatically fit the experimental data to a piece-wise linear model, thereby obtaining an instantiation of the required parameters. A user can easily perform such instantiation when wanting to simulate a particular cluster deployment. Alternatively, this instantiation can be conducted by a third party, for a range of typical cluster deployments, and made publicly available. SMPI users can then reuse these instantiations, or modify them to explore reasonable "what if?" scenarios.

VII. Evaluation

In this section we evaluate the accuracy, scalability and speed of SMPI simulations for scenarios ranging from simple point-to-point communication to more complex communication benchmarks. All experiments were conducted using SMPI implemented within SimGrid v3.5-r8210. To compare simulation results to real-world measurements we ran all the MPI applications described hereafter on the following clusters of the Grid'5000 platform: *griffon* and *gdx*. The *griffon* cluster comprises 92 2.5 GHz Dual-Proc, Quad-Core, Intel Xeon L5420 nodes. These nodes are divided into three cabinets that contain 33, 27, and 32 nodes respectively. Each cabinet has its own switch and these switches are then interconnected through a 10 Gigabit second-level switch. The *gdx* cluster comprises 312 2.0 GHz Dual-Proc AMD Opteron 246 scattered across 36 cabinets. Two cabinets share a common switch and all these switches are connected to a single second level switch through Ethernet 1 Gigabit links. Consequently a communication between two nodes located in two distant cabinets goes through three different switches.

A. Accuracy of SMPI Simulations

We measured errors in our experiments using the logarithmic error, as introduced in [26]. This new metric was proposed to overcome biases in the relative error metric. Given a reference value R and an experimental value X, the relative error is given by

$$Err = \frac{X - R}{R} \ .$$

It is easy to see that this metric is not symmetric: having X twice as large as R yields a relative error of 100%, while having X half as small as R yields a relative error of -50%. The logarithmic error is given as

$$Log_{Err} = |\ln X - \ln R| = |\ln R - \ln X| \ .$$

This metric fixes the previously observed bias because it is symmetric. It can also be used with additive aggregation operation (e.g., maximum, mean, variance). Finally, the logarithmic error value has to be taken out of the log-space to be interpreted as a regular percentage value:

$$Err = e^{Log_{Err}} - 1 \ .$$

1) Point-to-Point Communications: The goal of our first set of experiments is to validate the piece-wise linear network model described in Section IV-A. First we compare the results achieved by SKaMPI (using OpenMPI) and by SMPI for a simple ping-pong test between two machines on the cluster used to calibrate the simulation, i.e., the *griffon* cluster.

Figure 3 shows communication time vs. message size, using a logarithmic scale for both axes. We display three sets of SMPI results. The results "Default Affine" are obtained for an affine model calibrated on the cluster using the time to send a 1-byte message for the latency and the maximum achievable bandwidth

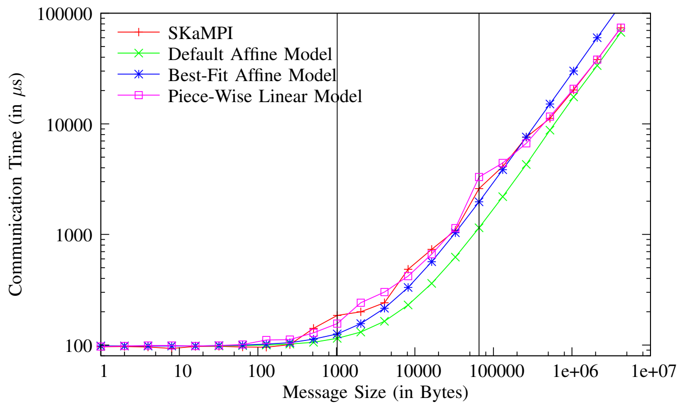

Figure 3. Comparison between a SKaMPI run and SMPI (default affine, best-fit affine, and piece-wise linear models) for a ping-pong operation between two machines of the calibration cluster (*griffon*).

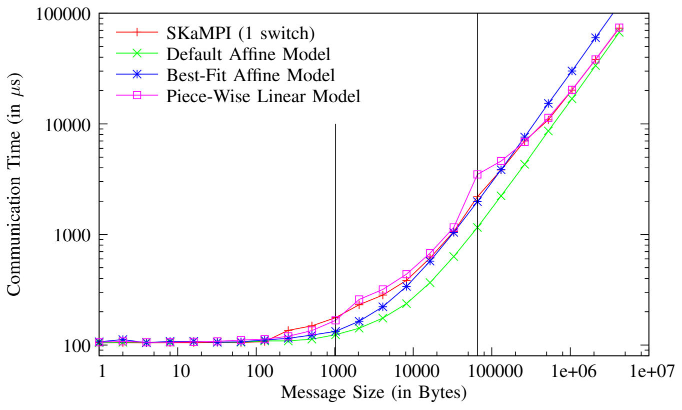

Figure 4. Comparison between a SKaMPI run and SMPI (default affine, best-fit affine and piece-wise linear models) for a ping-pong operation between two machines of the *gdx* cluster using the calibration made on the *griffon* cluster.

using the TCP/IP protocol (i.e., approximately 92% of the peak bandwidth). This is the standard method for instantiating the affine model, and corresponds to the approach taken by many of the MPI simulators reviewed in Section II. The "Best-Fit Affine" results are for an affine model instantiated using the latency and bandwidth values that minimize the average logarithmic error with respect to the SKaMPI results. We include these results to see whether a linear model could be inherently inaccurate. Finally, the "Piece-Wise Linear" results are for the piece-wise linear model described in Section IV-A and instantiated as described in Section VI.

We see that the piece-wise linear model matches the real-world results very well (at most a 8.63% average error overall, with worst case at 27%). By contrast, both affine models fail to capture the entire real-world behavior. The Default Affine model is accurate for small and big messages, but inaccurate in between (for a 32.1% average error overall, with worst case at 127%). The Best-Fit Affine model performs better for medium-sized messages, but overestimates communication time for big messages (for a 18.5% average error overall, with worst case at 62.6%). These results demonstrate that a piece-wise linear model is necessary for accurate simulation of MPI communication on a cluster.

Figure 4 presents results from a similar experiment conducted on a cluster with different compute nodes but a similar interconnect. The goal of this experiment is to demonstrate both that the instantiation of the network model in SMPI is decoupled from the characteristics of the compute nodes, and that calibration done on one cluster can be used to simulate a different cluster.

The results are very similar to those of Figure 3 with the piece-wise linear model being the most realistic (7.88% average error overall, with worst case at 59.1%). This shows that it is not necessary to perform a calibration step on each target platform to obtain accurate results. Both Default and Best-Fit Affine models still exhibits the same drawback for medium messages (for 28.1% and 16.4% average error overall, and worst case at 89.6% and 63.8%, respectively.)

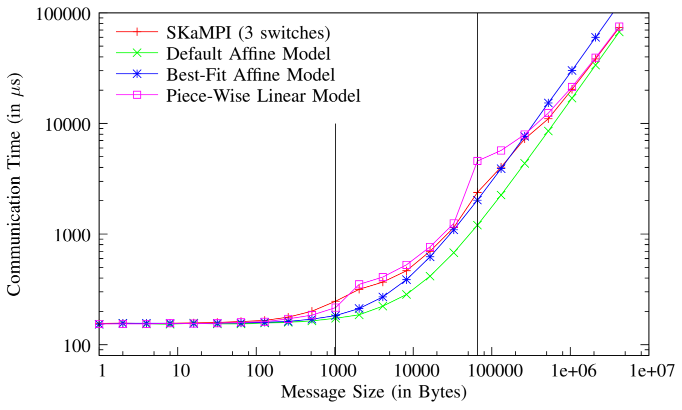

Figure 5. Comparison between a SKaMPI run and SMPI (default affine, best-fit affine and piece-wise linear models) for a ping-pong operation between two machines interconnected by three switches on the *gdx* cluster using the calibration made on the *griffon* cluster.

The calibration procedure described in Section VI is usually performed between two machines connected to the same switch. However, large compute clusters are often divided in several cabinets. The network interconnect is thus hierarchical and a point-to-point communication between two machines can go through two or more switches. Figure 5 shows the accuracy of SMPI in such a setting, i.e., a ping-pong between two nodes of the *gdx* cluster located on distant switches. The route connecting these two nodes goes through three different switches. The average error of our piece-wise linear model is 9.94% overall (with worst case at 92.2%), which is comparable to the accuracy shown in Figure 3. The bigger mis-estimations in Figures 4 and 5 (respectively 1.29 and 2.19 ms) appear at 64 KiB, which corresponds to a linear segment boundary of the model, and also to the message size at which the MPI implementation used switches from the eager to rendezvous protocol. Although the prediction at this point is worse than predictions of the other models, this segment's slope better fits the real data than other models as soon as message size reaches 256 KiB. By contrast, the best model at 64 KiB, namely Best-Fit Affine, makes an error of 46 ms at 4 MiB while our piece-wise linear model is only 1.6 ms away from the real-world value.

2) One-to-Many and Many-to-One Communications: We now assess the accuracy of SMPI using more complex communication operations. Among all the available One-to-Many and Many-to-One collective operations, we focus on the `MPI_Scatter` function. The scatter operation consists in distributing chunks of a single buffer of size S located on a *root process* among the P processes belonging to the MPI communicator. At the end of this operation, each process owns a chunk of size S/P. Depending on the message sizes and number of processes, MPICH2 and OpenMPI use several algorithms to implement this operation. One of them relies on a binomial tree, which is depicted in Figure 6 for 16 processes. Note that the volumes of data sent along each edge of this communication graph are different. For instance, process 0 will send 8 chunks of data to process 8 (that will then be scattered in the subtree rooted in 8) while only one chunk of data will be sent to process 1.

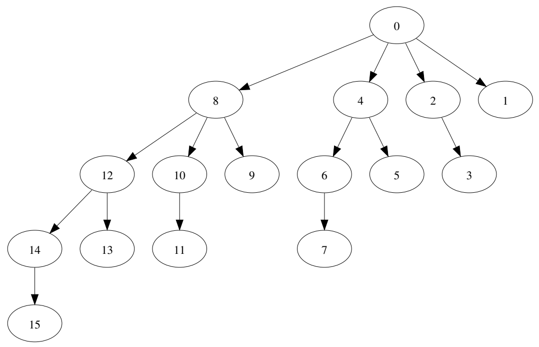

Figure 6. Communication scheme of a binomial tree based `MPI_Scatter` with 16 processes.

In Figure 7 we compare the execution times, on a per-process basis, of a binomial tree based scatter operation respectively achieved by OpenMPI and MPICH2 implementations and by SMPI. To ensure that OpenMPI and MPICH2 use the binomial tree algorithm, we do not call directly `MPI_Scatter`, but use a manual implementation of this algorithm. This is realistic since this algorithm is used in most cases by the implementations. In a future version of SMPI, we plan to implement other existing algorithms and detect which algorithm to use based on the message size and number of processes, just as real implementations like OpenMPI and MPICH2 do.

In this experiment, a 64 MiB buffer initially held by process 0 is scattered across 16 processes. The size of the receive buffers at the leaves of the binomial tree is thus 4 MiB. For the SMPI version, two execution times are displayed. The black bar shows the times to complete the scatter operation when network links suffer from contention. The white bar shows the results obtained on an equivalent platform whose network links do not induce contention. In other words, each communication going through one of these links will get the maximal bandwidth, i.e., 1 Gigabit per second, whatever the number of concurrent communications. This no-contention scenario is intended to mimic the behavior of most MPI simulators reviewed in Section II, which do not take contention into account, for comparison purposes.

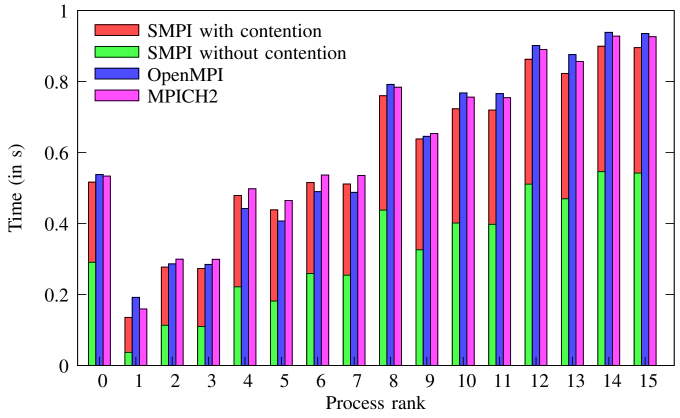

Figure 7. Per-process timing results for a binomial tree based scatter operation with 4 MiB messages.

We see that the network model without contention always underestimates the completion time of a scatter operation. Consequently, we claim that most of the MPI simulators reviewed in Section II would lead to similar underestimations. Conversely our piece-wise linear model with contention leads to simulated execution times that are very close to the performance of MPI implementations. On average, the difference between SMPI and MPICH2 is almost the same as the difference between OpenMPI and MPICH2 (around 5.3% overall, with worst case at 17.6% and 20.2% respectively).

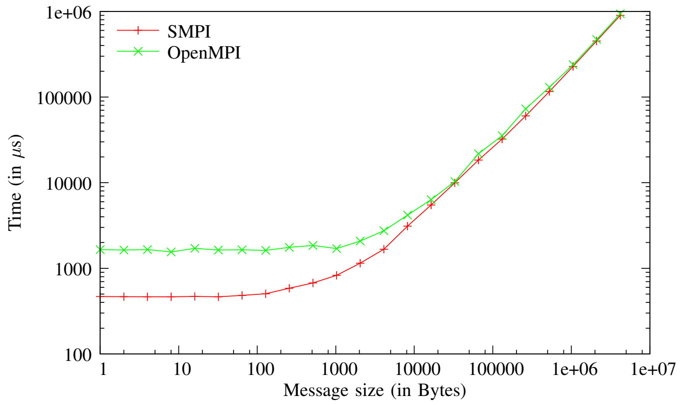

Figure 8. Impact of message size on accuracy for a binomial tree scatter operation with 16 processes.

Figure 8 shows a global view of the accuracy of SMPI for such a binomial tree based scatter operation with respect to the message size. We see that the scatter simulation with messages over 10 KiB is reasonably accurate (under 10% error). However, with smaller messages, the simulation underestimates the real-world execution time. The root cause of this error lays in the analytical network simulation model used by SMPI, as implemented as part of SIMGRID's simulation kernel (see Section V-A). This model computes the bandwidth allocated to each flow among those that contend with each other on a network path. This computation is based on the number of flows, the bandwidth of the bottleneck link on the path, and the end-to-end latencies of the flows. This model is continuous, meaning that conceptually all flows make progress simultaneously during each infinitesimal time unit. This corresponds to a continuous approximation of a discrete phenomenon in which the transmission of individual physical packets is serialized and packets are sent out in an interleaved manner. While the approximation error is amortized over large number of packets, i.e., for large messages, the approximation is optimistic in the case of small messages.

Figure 9 shows the evolution of the execution time of a scatter operation with regard to the number of involved processes. Here the size of the receive buffer is constant and of size 4 MiB while the size of the data to scatter increases linearly with the number of processes.

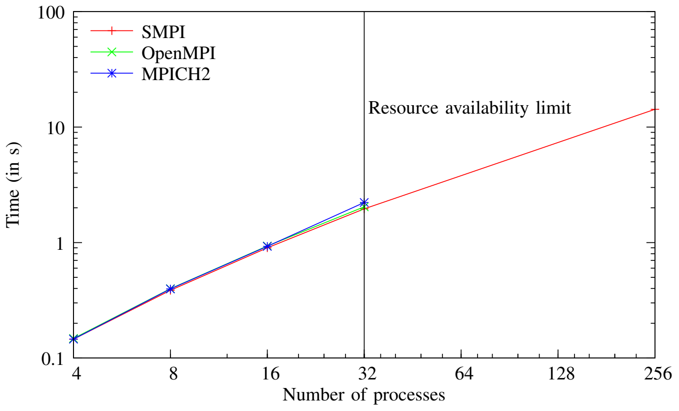

Figure 9. Binomial tree scatter operation with varying number of processes and 4 MiB receive buffers.

The performance of SMPI is very consistent with both MPI implementations for this message size. While the size of the actual cluster and its heavy load prevented us to make experiments with OpenMPI and MPICH for more than 32 processors, we pushed the scalability further with SMPI (up to 256 processes). We can see that the predicted execution times evolves in the same trend. Similar behavior is observed for receive buffers as small as 100 KiB (results not included). For smaller messages, SMPI underestimates the communication time as seen in Figure 8 and for the same reasons.

3) Many-to-Many Communications: The `MPI_Alltoall` function implements a collective operation in which each process sends distinct data to each of the receiver. The messages exchanged by each pair of processes are of the same size. As for scatter before, several algorithms exist to perform this collective operation, as the implementation is not enforced by the MPI specification. The pairwise All-to-All algorithm is one of the options for implementing this operation, and is used by OpenMPI and MPICH2 under some conditions on the message sizes and number of processes. This algorithm can be decomposed in as many steps as there are processes. At each step, each process sends and receives data to and from a unique distinct remote process. Figure 10 shows the communication scheme for each of these steps with 4 processes.

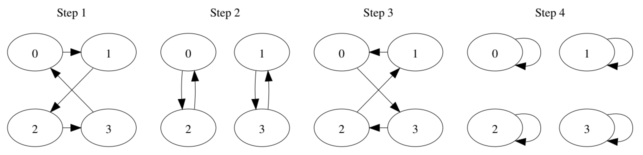

Figure 10. Communication scheme of a pairwise all-to-all with 4 processes.

We compare the accuracy of SMPI to that of a manual implementation of the pairwise algorithm with OpenMPI. As in Figure 7, we show the execution times achieved with a network model that ignores contention. Figure 11 shows that this simple model, depicted by the white bars, induces a logarithmic error of 78% that is consistent for all the 16 processes, for an all-to-all with 4 MiB messages. By contrast, the SMPI version that relies on the piece-wise linear model is accurate (less than 1% error) when accounting for contention.

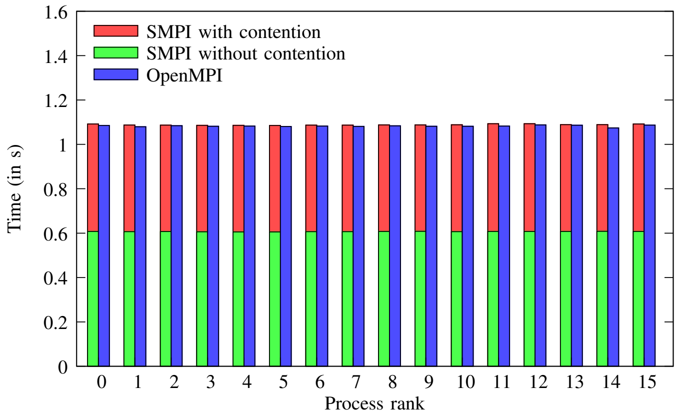

Figure 11. Per-process timing results for a pairwise all-to-all with 4 MiB messages.

Figure 12, which is for 16 processes, shows that simulation accuracy varies depending on the message size. SMPI leads to the same underestimation as with one-to-many or many-to-one operations for small messages. We have explained this behavior for the scatter simulation results in Figure 8, and the same explanation holds here as well. The overall average error in this all-to-all experiment is 28.7%, with worst case at 80%.

4) NAS DT Benchmark: The Data Traffic (DT) application is part of the NAS Parallel Benchmarks (NPB) suite. NPB is a set of programs commonly used to assess the performance of parallel platforms. Each kernel can be executed for 7 different *classes*, denoting different problem sizes: S (the smallest), W, A, B, C, D, and E (the largest). For instance, a class D instance corresponds to approximately 20 times as much work and a data set almost 16 as large as a class C problem.

In the case of the Data Traffic benchmark used in this experiment, the classes also denote the number of communicating processes in addition to the data size. Moreover

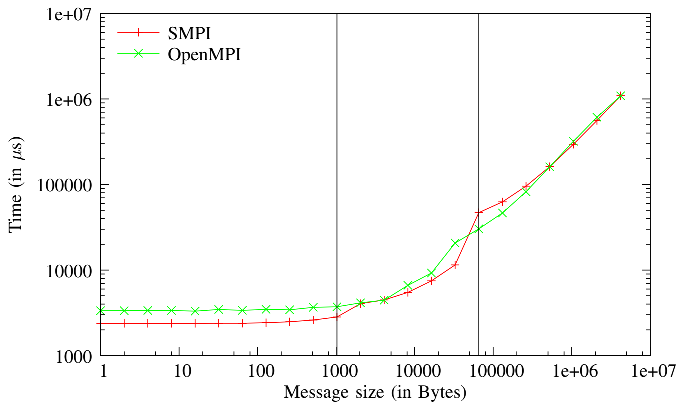

Figure 12. Impact of message size on accuracy for a pairwise all-to-all with 16 processes.

three communication schemes can be tested. A Black Hole graph (BH) collects data from multiple sources in a single sink, as shown in Figure 13. Conversely a White Hole graph (WH) distributes data from a single source to multiple consuming nodes, with a dual communication scheme than the one of Figure 13. The last communication graph is a Shuffle (SH) that arranges the processes in different layers and shuffles data from the top layer down to the bottom layer. Classes A, B and C, respectively involve 21, 43, and 85 processes for the White Hole and Black Hole graphs, and 80, 192 and 448 processes for the Shuffle graph.

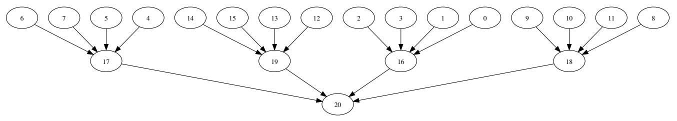

Figure 13. Communication scheme for the BH graph in DT Class A problem.

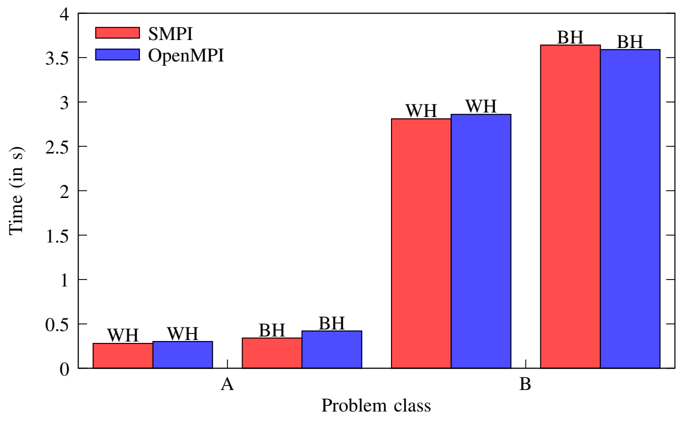

Figure 14. Execution time of the DT benchmark for classes A and B.

Figure 14 shows the comparison between SMPI and an OpenMPI implementation for the WH and BH variants of the DT benchmarks for classes A and B. The behavior of this complete benchmark is correctly predicted by SMPI, with an average error of 8.11% and a maximal error of 23.5% (for Class A and BH graph). SMPI is sufficiently accurate for predicting the correct trend, i.e., that the BH variant takes more time than the WH variant, with strong confidence. Recall that although obtaining this kind of performance information with OpenMPI requires access to up to 43 nodes, SMPI provides it using a single node. Due to the access policy on the *griffon* cluster, we could not run real-world experiments with more than 43 nodes, thus preventing us from running the SH variant and class C instances of the WH and BH variants.

B. Scalability

We now present SMPI results obtained by applying the techniques described in Section III-B, which allow us to scale the simulations up to the 448 processors needed for class C of the SH variant of the DT benchmark. This is well beyond what we could run on the *griffon* cluster (see the previous section).

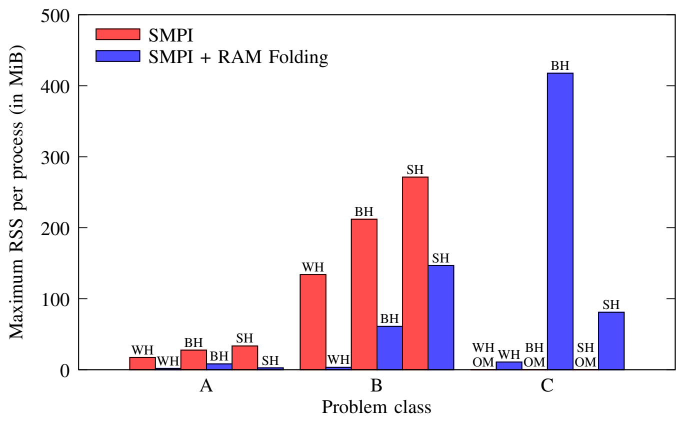

Figure 15. Memory consumption of DT with and without RAM footprint reduction.

Figure 15 shows the effect of using RAM footprint reduction techniques on the per-process maximum Resident Set Size (RSS). Memory consumption is drastically reduced when using such techniques, and it becomes possible to simulate applications that would otherwise overcome memory (these are represented by ”OM” — Out-of-Memory — labels in the figure). In these experiments, SMPI's memory consumption has been reduced by a factor 11.9 on average, and up to 40.5 for the WH graph in class B.

A side effect of RAM footprint reduction is that less CPU time is devoted to memory allocations, which is not captured by SMPI at the moment. The simulated execution times obtained are thus slightly lower than those obtained with the full memory footprint. When comparing SMPI with RAM footprint reduction techniques to OpenMPI, for those DT benchmarks that we were able to run on our cluster (WH and BH in classes A and B), the average error of the simulated execution time is increased to 18.5%, with a worst case at 42.1%.

C. Speed

One of the attractive aspects of on-line simulation is that simulation time can be shorter than simulated time. This time reduction can come from the simulation of communication operations. Figure 16 presents results from an experiment in which we measure the time needed for performing a scatter of messages of increasing sizes on a real-world platform with OpenMPI and in simulation with SMPI.

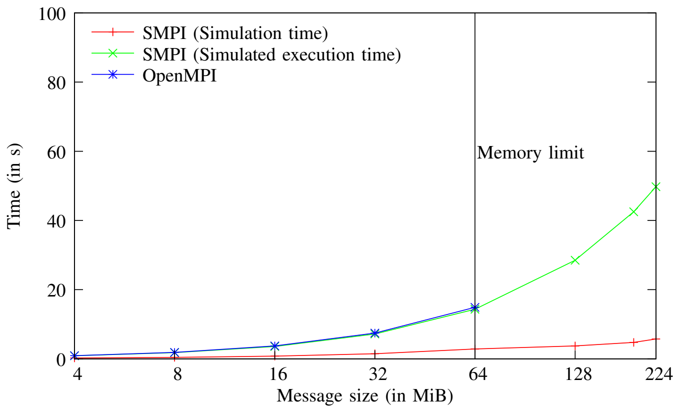

Figure 16. Simulation time versus experimental time for a binomial tree scatter operation with 16 processes with messages of increasing sizes.

Note that OpenMPI becomes limited by the memory as soon as the buffer size reaches 64MiB. With SMPI and its RAM folding technique, we were able to break this limit and obtain values up to 224 MiB. Again the obtained results are consistent with the observed trend while the simulation time remains low.

The gain offered by SMPI in terms of execution time increases with the message size. For medium-sized messages of 4 MiB, SMPI leads to a good estimation (maximum error of 4%) of the OpenMPI execution time while running 3.58 faster. For larger messages, this factor reaches 5.25.

The speed advantage over real execution that comes from the simulation of communications may be hindered by the execution of the computational part of an application (which is done on a single node). Indeed all computations are serialized in the simulation, thus increasing the simulation time by a factor equal to the number of simulated processes in the worst case. To minimize this effect, SMPI relies on the CPU sampling approach detailed in Section V-B. Figure 17 presents results that make it possible to evaluate the efficiency of this approach, more precisely of the `SMPI_SAMPLE_LOCAL` macro, in terms of reduction of the simulation time. For this experiment we used the Embarrassingly Parallel (EP) application from the NAS Parallel Benchmark suite. This application simply distributes a large computation among the processes. Each process computes its share of the computation without any further communication. The results shown in Figure 17 are for a class B problem on four processes. We observed similar results for other classes and numbers of processes. The x-axis is the sampling ratio, i.e., which fraction of the iteration space was actually executed. For instance, a sampling ratio of 25% means that only the first 25% of the iterations are executed, while the remaining 75% are replaced by the average computation time of the first iterations. The left y-axis shows the simulation time and the right y-axis shows the simulated execution time of the benchmark.

Expectedly, the simulation time decreases linearly as the sampling ratio decreases. When only one fourth of the iterations are actually executed (1024 instead of 4096 in this case), the simulation time is also divided by a factor four. More interestingly this reduction of the simulation time is not at the expense of accuracy. The dashed lines in Figure 17 show that the sampling factor has almost no impact on the simulated execution time and that accuracy is constant with respect to the OpenMPI implementation. This phenomenon is application dependent. The impact on accuracy would be zero for perfectly regular data-independent applications while it could be large (and thus unreasonable) for irregular data-dependent applications.

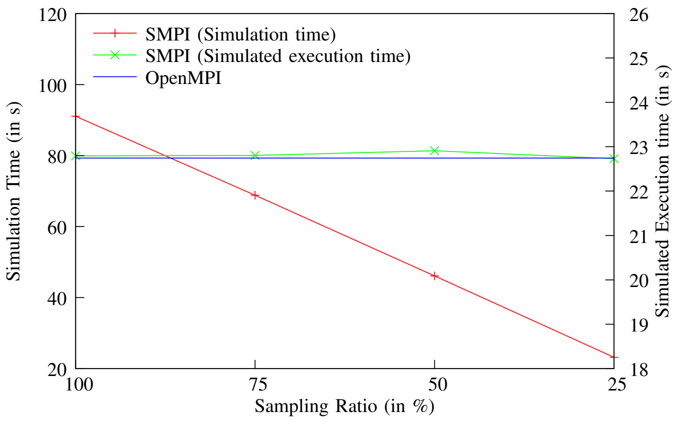

Figure 17. Impact of CPU sampling on the simulation time and accuracy.

VIII. Conclusion and Future Work

In this paper, we have proposed a new simulator, SMPI, for the on-line simulation of MPI applications. This simulator is built on top of the simulation kernel of the SimGrid toolkit and thus benefits of its fast, scalable, and validated network models. SMPI extends the existing models with an original piece-wise linear model. This new model takes into account the specifics of the cluster interconnect to perform accurate estimations of communication times. A salient property of SMPI is its capacity to simulate MPI applications on a single node. This allows SMPI users to assess the performance of applications and to explore "what if?" scenarios without needing access to a parallel computer. This is made possible by techniques to reduce both memory consumption and CPU requirements. In our experiments we have demonstrated the accuracy, scalability, and speed of SMPI over a wide range of applicative scenarios.

The simulation capabilities of SMPI can be extended in many directions. In terms of network simulation, the network model should be enhanced to better handle short messages, (see Section VII-C). Furthermore, this model is developed and validated only for TCP-based cluster interconnects, such as Gigabit Ethernet switches. Other interconnects including Myrinet or InfiniBand are currently not supported, and corresponding models need to be developed. In terms of simulation of CPU bursts, one interesting idea would be to automate the sampling technique described in Section III-A to run enough iterations to obtain accurate results without resorting to a user-provided value (much like the SKaMPI tool does for running micro-benchmarks). A long-term goal is to allow users to run SMPI simulations for specified real-world MPI implementations (e.g. OpenMPI or MPICH2). This feature would either require a careful (and automated) analysis of these implementations or interfacing SMPI directly with existing implementation, which would in turn provide parameters for instantiating the MPI implementation model in SMPI. Another long-term goal is for SMPI to simulate I/O resources and I/O operations, such as those implemented in MPI-IO. While the MPI-SIM project [11] has already provided I/O simulation capabilities, recent work has also tackled I/O simulation for MPI application [6]. Similar techniques, or models implemented as part of general-purpose I/O system simulators such as that in [28], could be integrated in SimGrid and SMPI.

Other future directions pertain to the usability of SMPI. At the moment, technical details in the internal design of SimGrid only allow SMPI to interface with applications written in C, while many MPI applications are written in Fortran. We expect to address this issue in the short term so that users can run any MPI application with SMPI. As explained in Section V, SMPI still requires user intervention to modify the application code (to privatize global variables, and add macros for increasing scalability and speed on a single node). Previous work shows that all these modifications can be done automatically via a compiler [3], [11], the development of which is a longer-term goal of this work.

SMPI is available at: http://simgrid.gforge.inria.fr

Acknowledgments

This work is partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022) and the CNRS PICS N° 5473. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd ed., ser. Scientific And Engineering Computation Series. MIT Press, 1999.

[2] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *Proc. of the 10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[3] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou, "Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures," *Journal of Parallel and Distributed Computing*, vol. 62, no. 3, pp. 393–426, 2002.

[4] M.-A. Hermanns, M. Geimer, F. Wolf, and B. J. N. Wylie, "Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications," in *Proc. of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 78–84.

[5] T. Hoefler, C. Siebert, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, 2010.

[6] A. Núñez, J. Fernández, J. Garcia, F. Garcia, and J. Carretero, "New Techniques for Simulating High Performance MPI Applications on Large Storage Networks," *Journal of Supercomputing*, vol. 51, no. 1, pp. 40–57, 2010.

[7] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications," in *Proc. of the 15th International Euro-Par Conference on Parallel Processing*, 2009, pp. 135–148.

[8] J. Zhai, W. Chen, and W. Zheng, "PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node," in *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan. 2010, pp. 305–314.

[9] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Application Performance Modeling and Prediction," in *Proc. of the International Conference for High Performance Computing and Communications (SC)*, Nov. 2002.

[10] P. Dickens, P. Heidelberger, and D. Nicol, "Parallelized Direct Execution Simulation of Message-Passing Parallel Programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1090–1105, 1996.

[11] R. Bagrodia, E. Deelman, and T. Phan, "Parallel Simulation of Large-Scale Parallel Applications," *International Journal of High Performance Computing Applications*, vol. 15, no. 1, pp. 3–12, 2001.

[12] R. Riesen, "A Hybrid MPI Simulator," in *Proc. of the IEEE International Conference on Cluster Computing*, Sep. 2006, pp. 1–9.

[13] G. Zheng, G. Kakulapati, and L. V. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *Proc. of the 18th International Parallel and Distributed Processing Symposium*, Apr. 2004.

[14] D. A. Grove and P. D. Coddington, "Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers," *Journal of Supercomputing*, vol. 34, no. 2, pp. 201–217, 2005.

[15] E. León, R. Riesen, and A. Maccabe, "Instruction-Level Simulation of a Cluster at Scale," in *Proc. of the International Conference for High Performance Computing and Communications (SC)*, Nov. 2009.

[16] R. Badia, J. Labarta, J. Gimnez, and F. Escal, "Dimemas: Predicting MPI applications behavior in Grid environments," in *Proc. of the Workshop on Grid Applications and Programming Tools*, 2003.

[17] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. Kalé, "Simulation-Based Performance Prediction for Large Parallel Machines," *International Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 183–207, 2005.

[18] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler, "MPI-NetSim: A network simulation module for MPI," in *Proc. of the 15th International Conference on Parallel and Distributed Systems*, 2009.

[19] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, "The MicroGrid: a scientific tool for modeling computational grids," in *Proc. of the ACM/IEEE conference on Supercomputing*, 2000.

[20] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, "Scalability and Accuracy in a Large-Scale Network Emulator," in *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[21] L.-C. Canon, O. Dubuisson, J. Gustedt, and E. Jeannot, "Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool," *Journal of Systems and Software*, vol. 83, no. 5, pp. 786–802, May 2010.

[22] S. Prakash, E. Deelman, and R. Bagrodia, "Asynchronous Parallel Simulation of Parallel Programs," *IEEE Trans. on Software Engineering*, vol. 26, no. 5, pp. 385–400, 2000.

[23] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proc. of the 11th European PVM/MPI Users' Group Meeting*, Sep. 2004, pp. 97–104.

[24] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in computer Science, vol. 2474. Linz, Austria: Springer, Oct. 2002, p. 7.

[25] K. Fujiwara and H. Casanova, "Speed and Accuracy of Network Simulation in the SimGrid Framework," in *Proc. of the 2nd International Conference on Performance Evaluation Methodologies and Tools*, 2007.

[26] P. Velho and A. Legrand, "Accuracy Study and Improvement of Network Simulation in the SimGrid Framework," in *Proc. of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[27] R. Reussner, P. Sanders, and J. L. Träff, "SKaMPI: a Comprehensive Benchmark for Public Benchmarking of MPI," *Scientific Programming*, vol. 10, no. 1, pp. 55–65, 2002.

[28] P. Berenbrink, A. Brinkmann, and C. Scheideler, "SIMLAB: A Simulation Environment for Storage Area Networks," in *Proc. of the 9th Euromicro Workshop on Parallel and Distributed Processing*, Feb. 2001, pp. 227–234.