Towards Massively Parallel Simulations of Massively Parallel High-Performance Computing Systems

Robert Birke
bir@zurich.ibm.com

German Rodriguez
rod@zurich.ibm.com

Cyriel Minkenberg
sil@zurich.ibm.com

IBM Research – Zurich, Säumerstrasse 4, 8803 Rüschlikon, Switzerland

ABSTRACT

The power of high-performance computing (HPC) is applied to simulate highly complex systems and processes in many scientific communities, e.g. in particle physics, weather and climate research, bio-sciences, materials science, pharmaceutics, astronomy, or finance.

Current HPC systems are so complex that the design of such a system, including architecture design space exploration and performance prediction, requires HPC-like simulation capabilities. To this end, we developed an Omnest-based simulation environment that enables studying the impact of an HPC machine's communication subsystem on the overall system's performance for specific workloads.

As the scale of current high-end HPC systems is in the range of hundreds of thousands of processing cores, full system simulation—at an abstraction level that still maintains a reasonably high level of detail—is infeasible without resorting to parallel simulation, the main limiting factors being simulation run time and memory footprint.

We describe our experiences in adapting our simulation environment to take advantage of the parallel distributed simulation capabilities provided by Omnest. We present results obtained on a many-core SMP machine as well as a small-scale InfiniBand cluster.

Furthermore, we ported our simulation environment, including Omnest itself, to the massively parallel IBM Blue Gene®/P platform. We report results from initial experiments on this platform using up to 512 cores in parallel.

1. INTRODUCTION

Simulation has become an indispensable tool in many fields of science, both academic and industrial. Rapid advances in the field of high-performance computing (HPC) have enabled tackling ever-larger problems and/or arriving at a solution much faster.

HPC systems are complex amalgamations of various hardware components and software layers: from CPUs, caches, memories, buses, interconnection networks, through operating systems, communication protocols, programming models, up to the parallel applications themselves. Understanding the interactions between these aspects and their implications on system performance is a task that itself can benefit tremendously from HPC capabilities. As HPC systems exhibit very high degrees of parallelism and regularity, they are prime candidates for a parallel discrete event simulation (PDES) approach.

Our research focuses on how the communication subsystem, including protocols, communication libraries, and network hardware, affects system performance as experienced by the user. We therefore opted for a workload-centric evaluation approach, based on execution traces from benchmark programs (for instance from the NAS Parallel Benchmark suite), or specific application codes.

The level of parallelism in today's largest supercomputers is truly massive, with tens of thousands of compute nodes, hundreds of thousands of processor cores (rapidly approaching the 1 million mark) and more than a petabyte worth of memory. The #1 system on the Top 500 list [5] as of November 2011 is Fujitsu's "K" Computer, which achieves more than 10 PF/s of sustained performance using over 700,000 SPARC64 cores.

As a direct consequence, modeling such systems inevitably requires massive parallelism as well. In fact, a very basic problem is that, given current supercomputers of generation x, to simulate a machine of the next-generation $x+1$ in full detail, we would need a machine of generation $x+2$, so we are always lagging two generations in terms of the required simulation capability. This implies that judicious decisions regarding the appropriate levels of abstraction are required to manage the model's complexity.

Although this particular area of research is especially suitable to PDES, many OMNeT++/Omnest user communities could benefit from PDES. As discrete event models generally have a large degree of inherent parallelism, applying PDES should in many cases be fairly straightforward and achieve acceptable speedup factors.

Moreover, as basically every consumer desktop machine has anywhere between 2 and 12 cores nowadays, parallel simulation is no longer an exotic, expensive technique available only to few, but instead a viable, affordable option for most users, especially considering Omnest's built-in support for PDES. With this paper, we would like to share our experiences with the OMNeT++ community and encourage others to give it a try.

The remainder of this paper is organized as follows. Sec. 2 describes our methodology and simulation environment.

OMNeT++ 2012, March 19-23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247685

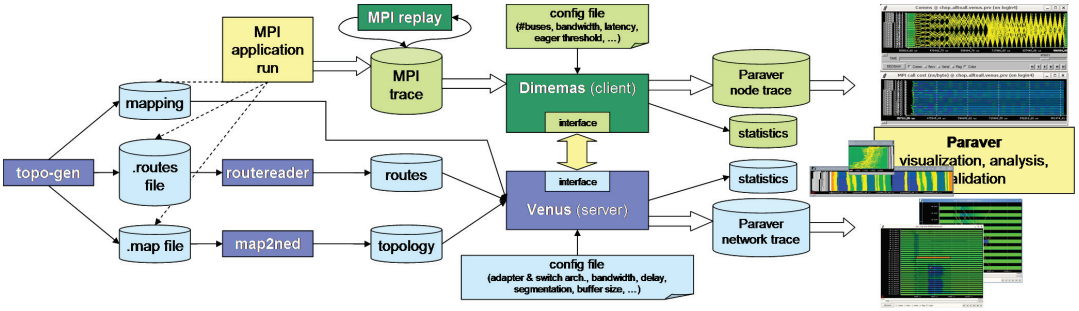


Figure 1: Venus toolchain.

We briefly summarize Omnest's built-in support for PDES in Sec. 3, outline the difficulties and pitfalls in parallelizing our simulator in Sec. 4, and report scalability results on SMP and cluster platforms in Sec. 5. Our experiences in porting Omnest and Venus to the massively parallel Blue Gene platform and the corresponding experiments are described in Sec. 6. Finally, we conclude in Sec. 7.

2. SIMULATION ENVIRONMENT

The interconnection network of an HPC system is usually modeled at a high level of abstraction, resorting to discrete-event simulation—as opposed to the cycle-accurate simulation common in processor design—to enable scaling to systems with tens of thousands of network ports. The main purpose of interconnection network simulation is to optimize network topology, switch and adapter architectures and parameters, scheduling and routing policies, flow- and congestion-control mechanisms, as well as aspects related to the software communication stack (protocols, libraries).

2.1 Taking the application view

The common performance evaluation approach taken in network and switch design is of the *Monte Carlo* variety, i.e., the workload applied is of stochastic nature, under the assumption that traffic patterns can be characterized using particular stochastic processes for interarrival time, message size, and destination distributions. This is also referred to as *synthetic* traffic, as opposed to traffic generated by real applications or benchmarking programs.

Although such traffic assumptions may be defensible within a wide-area networking context, where large degrees of statistical multiplexing may reasonably be assumed, they are certainly not applicable within the context of HPC systems.

Trace-driven simulation is one class of approaches to accurately capture the behavior of HPC traffic injection. An application is represented by a "post-mortem" trace, rather than by an exact model of its behavior. Such a trace is collected during a run of the application on a real parallel computer and mainly contains *computation* and *communication* records. Computations are represented by the amount of CPU time they consumed, not by the operations actually performed. Communications are represented by their key parameters, such as source and destination thread, message size, start and end times, communication operation and communication mode. During simulation, a playback engine replays the trace, taking into account the semantics of the communication operations, which depend on the application's parallel programming model. Computation records are treated as delays in between subsequent communications. Communication records are transformed into data messages that are exchanged between tasks via a model of the interconnection network.

As many scientific HPC applications are based on the Message Passing Interface (MPI), tracing MPI calls is a suitable method for characterizing the communication patterns of an important class of HPC workloads. Although our tool is used mostly to analyze MPI applications, it also supports partitioned global address space (PGAS) programming models such as SHMEM and UPC.

2.2 Toolchain

The toolchain used in our research is shown in Fig. 1 and has been described in more detail in [6, 3]. Our Omnest-based network simulator, *Venus*, closely interoperates with the Dimemas and Paraver tools developed at the Barcelona Supercomputing Center (BSC). Dimemas is an MPI simulator that reads the application traces and replays them. For any operation that requires communication among tasks, Dimemas sends a corresponding message via a TCP/IP socket to Venus. Venus simulates the interconnection network, first performing segmentation of messages into packets or frames, then injects these packets into a simulated network of switches, through which they are routed towards their destination nodes, and finally reassembles the packets into messages. The corresponding information received from Dimemas is piggybacked on the simulated messages, and returned to Dimemas (again via a socket interface) when the simulated message has arrived at its destination in its entirety.

To ensure that causality is observed, the socket interface is also used to implement a simple protocol based on the Chandy-Misra-Brandy algorithm [2], also known as the null message algorithm (NMA), to keep the future event lists of both simulators in sync (unlike Venus, Dimemas is not based on Omnest). Both Dimemas and Venus can output traces that can be visualized and inspected in detail with the Paraver visualization tool. In addition, we have developed tools to generate, translate, and import topologies and to compute, optimize, and import source routes.

2.3 Model components

The main components of the Venus simulator are i) workload, ii) compute node, iii) interconnection network, and iv) simulation statistics and control. Workloads can be applied by means of stochastic or deterministic traffic generators, workload models, or application traces.

The compute node is represented by abstract resource

models, which represent node resources such as CPUs (cores), memory (banks, bandwidth, latency), buses (number, bandwidth), and IO (interfaces, bandwidth).

As the focus of our study is on the effect of the system-wide interconnect, the latter is modeled at a suitably fine level of detail. We chose the abstract level of the so-called flow-control digit, or *flit* for short. In essence, this is the atomic unit of transfer across a communication link, which may vary across different networking technologies.

The interconnect model comprises two basic module types: *adapters* as the interface between the compute nodes and the network, and *switches*, forming the network itself.

3. PARALLEL SIMULATION SUPPORT IN OMNEST

Parallel discrete event simulation splits a model into partitions called logical processes (LPs). Each LP is executed on a different processor or host. As each partition is only a part of the original model, it will, in general, have fewer events to simulate and a smaller memory footprint. However, achieving ideal (linear) speedup and memory reductions is difficult. Memory reductions are hampered by the overhead of replicating some common information over all partitions, whereas execution time suffers from a) the overhead introduced by synchronization and communication between LPs and b) intrinsic limitations of the code, such as non-parallelizable code sections.

As each LP has its own future event set and local simulation time, synchronization is needed to prevent violations of the causality of events. Without synchronization, an LP could send an event to another LP with a timestamp that is in the past of the receiving LP, therefore breaking the causality of events in the receiving LP. The overhead itself is due both to the additional messages the LPs need to exchange and process and to the intrinsic overhead of the synchronization algorithm itself. Omnest supports parallel simulation based on conservative synchronization via the NMA. This has been shown to be an efficient approach for large-scale network simulations in [7].

For a brief introduction to PDES with Omnest, we refer to Chapter 15 of the Omnest manual [9]. A method to estimate whether a given model will be able to obtain decent speedup using parallel simulation is given in [10]. The key formula is

$$\lambda = \frac{L \cdot E}{\tau \cdot P}, \tag{1}$$

where L is the *lookahead* time between partitions in simulated seconds (determined by the model), E is the number of events per simulated second (as generated by the model), τ is the latency between model partitions (dependent on the communication hardware and software), and P is the number of events processed per (wallclock) second (of a serial execution of the model on a given hardware platform).

If the resulting coupling factor λ is significantly larger than one (by an order of magnitude or two), the model is likely to achieve good speedup using parallel simulation.

The rationale behind this equation is that the model should have a sufficiently large number of events in the lookahead (given by $L \cdot E$) to keep the CPU busy during the communication time, i.e., the number of events processed during the communication time (given by $\tau \cdot P$). The number of partitions mainly affects the event density E.

In principle, almost any Omnest model can be run in parallel without requiring fundamental code changes. The main model constraints are that it should *a)* have no global variables, *b)* communicate only through Omnest's messaging mechanism (not via direct member or method access, nor direct sends), *c)* use only static topologies (i.e., without dynamic module instantiation), and *d)* provide sufficient lookahead L in the form of channel delays. Omnest offers support for different communication libraries, including MPI, making it easy to run on multi-core shared-memory machines as well as on clusters or even on massively parallel processing (MPP) machines such as the IBM Blue Gene architecture.

Partitioning is done by assigning a partition identifier to each module in the model (via Omnest's configuration file). The partitions should be as homogeneous as possible, so that the simulation load is evenly balanced across LPs, because the overall speedup is gated by the slowest LP. The number of partitions should not be too high so that the value of λ_n is not too small. Moreover, the partitioning should maximize the lookahead between LPs to increase λ and minimize the number of events crossing the LP boundaries, thus reducing the communication overhead.

4. PARALLELIZING VENUS

Venus includes support for parallel simulations based on the Omnest framework. In this section, we discuss our experience in parallelizing the Venus simulator.

4.1 Adapting Venus for parallel simulation

In the case of the Venus (and network simulators in general), the lookahead requirement is easy to fulfill because propagation delays on links or through adapters and switches and/or minimum transmission times (flit duration) are natural sources for lookaheads, so requirement d was met. In most of our experiments, we used lookaheads on the order of a single flit time (about 50 ns). Depending on the hardware platform, τ will be on the order of a few to tens of μs.

In addition, as HPC topologies are regular and static, requirement c was met and because Venus communicates between modules by means of messages exclusively, requirement b was also fulfilled.

Typical values for E and P for our model are in the range of 10^{10}–10^{11} and 10^6, respectively. E scales fairly linearly with the number of end nodes and network ports. The lookahead in our simulations is about $5 \cdot 10^{-8}$ s. On a cluster, τ can be expected to be on the order of microseconds. Therefore, λ will be in the range of 100 to 1,000, i.e., sufficiently large to warrant pursuing parallel simulation.

The initial parallelization of Venus was of course not quite as simple as just rebuilding Omnest with MPI,[1] setting the corresponding options in the ini file, and assigning partion IDs. A number of cardinal sins had been committed, including, but not limited to:

- Modifying messages after having sent them out on a channel. This is generally a bad idea, but in a serial execution rarely causes a problem. In parallel mode, however, doing so can lead to obscure bugs and crashes. This was solved by means of additional temporary message buffering.

[1]For development we used OpenMPI, whereas for specific platforms such as Blue Gene, tailor-made MPI implementations can be expected to deliver better performance.

Table 1: Simulated topologies.

	2D mesh	3D mesh	fat tree	h-mesh
configuration	$k = 64$	$k = 16$	$k = 8$	$k_{1,2,3} =$ 16,8,8
	$n = 2$	$n = 3$	$n = 4$	$n = 3$, $p = 4$
end nodes	4,096	4,096	4,096	4,096
switches	4,096	4,096	4,096	1,024
ports/switch	5	7	16	33
links	10,240	14,336	32,768	16,986
diameter	126	45	6	3

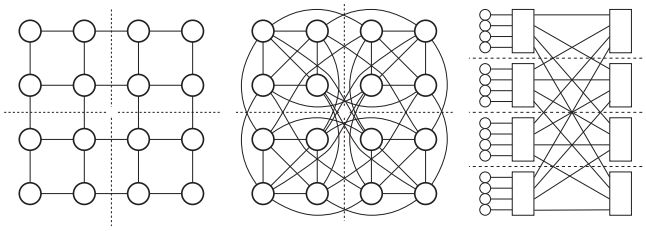


Figure 2: Partitioning examples for a 16-node mesh (left), h-mesh (middle) and fat tree (right) topology into four LPs. The dashed lines delimit each LP.

- Use of global variables and static class members (requirement *a*). Eliminating global variables is generally a good programming practice. However, static members have some legitimate uses, and completely eliminating them would not have been feasible. By keeping in mind that each partition will have its private copy of any such variable, these issues could be solved.
- Topology discovery: when using the `cTopology` class, only modules present in the local partition will be discovered. Therefore, for full topology discovery, additional explicit message exchanges were required.

We also discovered some problems with partitioning of the model. Placing different submodules of the same compound module on different partitions led to failed assertions in the initialization of the parallel simulation; apparently, proxy gates were not properly connected in certain cases. One very important aspect to keep in mind is that when a compound module is distributed over multiple partitions, the partition ID *must* specify *all* partitions in which any of its submodules reside (by means of a comma-separated list).

We also discovered an Omnest bug that caused zero lookahead to be assumed for any two partitions that were not connected to each other, as well as an issue in the use of MPI in the parsim implementation. All of the above bug fixes have been incorporated into Omnest version 4.2; for version 4.1, patches are available.

Another issue was proper implementation of message packing and unpacking for message classes that make use of the `customize` feature, because the msg-compiler can't always produce a correct implementation in such cases. A further challenge was the correct and efficient construction of a global address book (i.e., a mapping of task ranks to network addresses and vice versa), which required implementing explicit announcements among the partitions to communicate the task ranks and network addresses present in each individual partition to all of the others. Finally, the MPI buffers allocated for the buffered sends had to be increased, depending on the number of partitions, by increasing the Omnest parameter `parsim-mpicommunications-mpibuffer`.

4.2 Model partitioning

Having obtained a working parallel version of our simulator, the next important step was deciding on a suitable partitioning of the model.

Initial tests revealed that the model's *Statistics* module was a major bottleneck. The model had only one such module, and *every single* packet had to be routed through it for statistics collection. The Statistics module performs aggregation of packet- and message-level statistics and forwards these aggregates to the *Control* module, which collects them and, when the simulation finishes, produces a detailed statistics report. Because the partition that contained the Statistics module received much more messages than the others, it proceeded at a much slower pace, holding back the overall simulation speed. The key to obtaining good speedup was instantiating one Statistics module in every partition (responsible for all nodes in that partition), which effectively distributed its load equally over all partitions, thus achieving a good load balance.

Venus supports a broad range of network topologies, each requiring a specific partitioning strategy. Here, we focus on three regular topologies: the mesh, the hierarchical full mesh (referred to as h-mesh), and the fat tree [8]. To facilitate comparison, we used a configuration connecting 4,096 end nodes for each network type. Specifically, we considered a 2D 64×64 mesh, 3D 16×16×16 mesh, a 3-level h-mesh (with 16, 8, 8 groups per level and 4 nodes per switch), and a 4-level 8-radix fat tree (Table 1). All links had the same delays (lookahead), which was set to the flit duration of 51.2 ns (64B @ 10 Gb/s). In all experiments, we ran the model for 1 ms of simulated time, applying uniform random Bernoulli arrivals at 100% load.

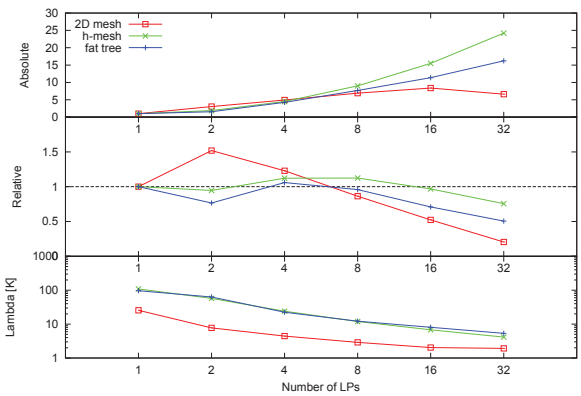
Figure 2 presents examples of how to partition these topologies into LPs. For clarity, the examples are given for a 16-node configuration rather than for the full 4,096-node configuration. In the figures for mesh and h-mesh, each circle represents one switch and the nodes directly attached to it, whereas in the fat-tree representation hosts and switches are shown separately.

For optimal load balancing, each partition should be assigned about the same number of hosts and switches. This avoids having one slow LP dragging down the performance of the entire simulation. Moreover, the partitioning should minimize the number of links crossing LP boundaries, so that the amount of traffic that stays within the local partition is maximized; traffic with source and destination in the same LP should not be forced to cross an LP boundary, to avoid the additional communication overhead. Finally, the partitioning should maximize the lookahead values between LPs to reduce the synchronization overhead, but as all lookaheads were equal, there was no room for optimization in this respect.

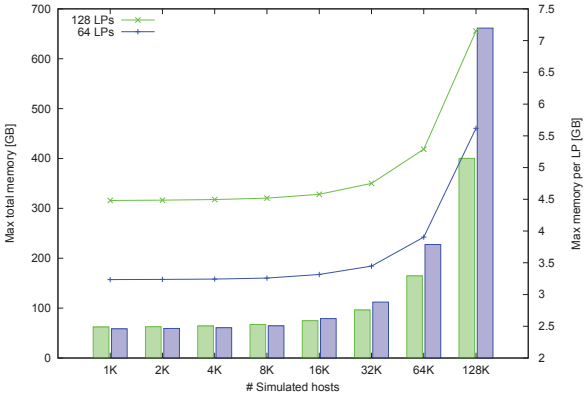
5. SCALABILITY

We simulated all three network models using Omnest v4.1 and OpenMPI v1.4 on one high-end server equipped with 4 Intel[2] Xeon[2] X7560@2.27 GHz CPUs (32 cores total) and

[2]IBM, Blue Gene, PowerPC are trademarks of International



(a) Absolute (top), relative (middle) speedups, and λ (bottom) for the three reference topologies.



(b) Maximum total and per-LP h-mesh memory footprint with increasing number of hosts.

Figure 3: (a) Speedup values on SMP as a function of number of LPs (constant model size), (b) memory usage (aggregate (lines) and per-LP (bars)) on cluster as a function of model size for 64 and 128 LPs.

768 GB RAM.

Figure 3a shows the results of simulation runs using $N \in [1, 2, 4, 8, 16, 32]$ LPs. The upper and middle panels show the speedups achieved, in absolute (serial run time (1 LP) divided by run time for N LPs) and relative (absolute speedup divided by N) terms. The bottom panel presents the corresponding λ values measured. All simulations behave similarly, and all have high λ values, indicating that the models are sufficiently large and complex to benefit from parallel simulation, as evidenced by the speedup values achieved. However, we observed some differences between the models. As expected, as N increased, the relative speedup first increased up to a peak and then decreased. The peak indicates the optimum tradeoff between the gain from parallel simulation and the overhead incurred. The mesh topology attained the peak earlier than the fat tree and h-mesh do. Hence, fat trees and h-meshes can achieve better speedups with a high number of LPs.

Surprisingly, we obtained close to or even better than linear speedup for certain values of N, i.e., relative speedups greater than one, which may be a consequence of the reduced overhead in the simulator itself because of the lower event density. As operations on the future event set (inserting, removing events) often account for a significant share of CPU time and as these operations get faster as the list becomes shorter, partitioning the model can benefit from this compound effect. In addition, as each core has its own local cache, overall more cache memory is available, resulting in (on average) faster memory accesses.

In addition to achieving speedup, parallel simulations can also help to overcome memory constraints. Figure 3b shows the memory footprint of different h-meshes with increasing numbers of nodes run on a 64-node cluster with two Intel® Xeon® X5670@2.93 GHz CPUs per node, interconnected by an InfiniBand network. As the number of hosts increased, the total peak memory footprint rapidly grew to hundreds of GBytes. In these simulations, the scale of the simulated system was clearly limited by the available RAM. However, the maximum memory footprint per partition was much more reasonable. This enabled simulation of up to 128K nodes, which would not have been feasible on the SMP machine.

6. PORTING VENUS TO BLUE GENE

6.1 Blue Gene

IBM's Blue Gene project [4, 1] set out to design a massively parallel computer architecture that was initially specifically tailored to biomolecular studies concerning, for instance, protein folding. At the same time, Blue Gene was also intended to foster innovation in parallel computer hardware architecture and software, aiming to achieve new levels of parallelism and overall computing power (in GF/s), and, perhaps even more importantly, to attain very high-density packaging and vastly improved GFlops per Watt ratios.

Blue Gene systems are generally based on relatively "slow" processors, with clock rates that are significantly below the limit of their respective technology generations. This design decision enabled significantly better power-efficiency and, consequently, denser packaging and smaller footprint, which are all crucial aspects in reducing the total cost of owning and operating such an HPC system.

Up to now, two generations of the Blue Gene architecture have been made available, namely Blue Gene/L and Blue Gene/P, whereas the third generation, Blue Gene/Q, has recently been announced. BG/L and BG/P employ a 3D Torus network for point-to-point communications, augmented by a dedicated network for collective communications (e.g., broadcasts and reductions) and a global interrupt network for fast barrier synchronization.

As their operating system, the BG/P compute nodes have a lightweight proprietary kernel, whereas the front end and service nodes use a SLES-based Linux variant. Application developers have the IBM XL suite of compilers and tools

and the GNU toolchain at their disposal.

6.2 Porting Omnest and Venus

The GNU toolchain enables easy porting of application across GNU/Linux platforms, and relatively simple porting of applications to other systems, provided that they have installed some basic tools and that the source code has been made portable using the *autotools* tools and properly uses pre-processor directives (e.g. `#define`) to selectively include the appropriate code for each platform.

To install Omnest and Venus on BG/P we had to solve several portability issues in both codes. The issues that we describe here could be considered for inclusion in the Omnet code to facilitate porting not only to other GNU/Linux platforms, but also to AIX platforms using the IBM XL compilers. This section summarized the main changes required to successfully compile our simulator.

6.2.1 GNU compiler

To port Venus to the BG architecture, we first ported the core Omnest library (excluding the GUI). We started from Omnest version 4.1 with the appropriate patches applied. Porting Omnest using the GNU toolchain was quite straightforward; it mainly required setting up `configure.user` correctly. Apart from setting up the paths pointing to the correct libraries (i.e., the versions compiled for the BG/P back-end; we had to additionally port the zlib library), we added explicit definitions of the CC, CXX and AR variables pointing to the cross-compilers needed by the BG/P back-end, and we disabled the `SHARED_LIBS` option to produce static libraries. Furthermore, we removed the graphical interface by setting the `NO_TCL` variable, thus removing the dependency on the Tcl libraries, because no graphical environment is available on the back-end.

After these steps, we could compile Omnest "out of the box" with the GNU toolchain. Having successfully compiled Omnest, porting Venus (or any other application based on Omnest) was simple because the `opp_makemake` script will import the basic setup from Omnest.

However, to get more performance out of the BG architecture, it is recommended to use IBM's XL toolchain, which can perform better code optimization by taking advantage of architecture-specific hardware features.

6.2.2 IBM XL compiler

To add support for the XL compiler we had to change `configure.user` again to point to the corresponding tools, as well as the `./configure.in` script to use new variables that were needed for compiler options incompatible with the XL toolchain. In particular, the original script uses the *-fno-stack-protector* option, which is not supported by the XL compilers. Also, during the execution of the configuration script the correct variable to test the version in XL compilers is *-qversion* instead of *–version*. The flag *-pthread* is not supported by the XL toolchain either. We used the `bgxlc_r` and `bgxlc++_r` compilers for C and C++, and as a static library archiver we used the modified GNU `powerpc-bgp-linux-ar`. We also supplied the correct flags and library paths for the BG-specific MPI libraries.

The most important code modifications were as follows. Omnest includes several yacc `(*.y)` files that, after conversion to C code by means of yacc, caused a compilation problem, because of the double definition of memory allocation functions in both `stdlib.h` and within the generated code. This was easily resolved by indicating that `stdlib.h` is already included (by putting `#define YYINCLUDED_STDLIB_H` at the head of the yacc source files).

Some modifications were necessitated by the original code not strictly adhering to the C++ standard. For instance, the use of the `static` keyword to specify internal linkage (limited to file scope) is deprecated in C++, and should be replaced by an *unnamed* namespace.

The XL compiler requires that function macros (of the type of `#define func(x)`) that perform class registration for the Object Factories end with a semicolon. This also required changes to scripts (e.g. `opp_msgc`) that generate C code that uses these macros. Some functions that are available for the C library (e.g. `trunc()`) had to be replaced with the equivalent C++ function (`ctrunc()`). Also, a buffer overflow had to be corrected in `cexception.cc`: the buffers should be one element larger than `BUFLEN`.

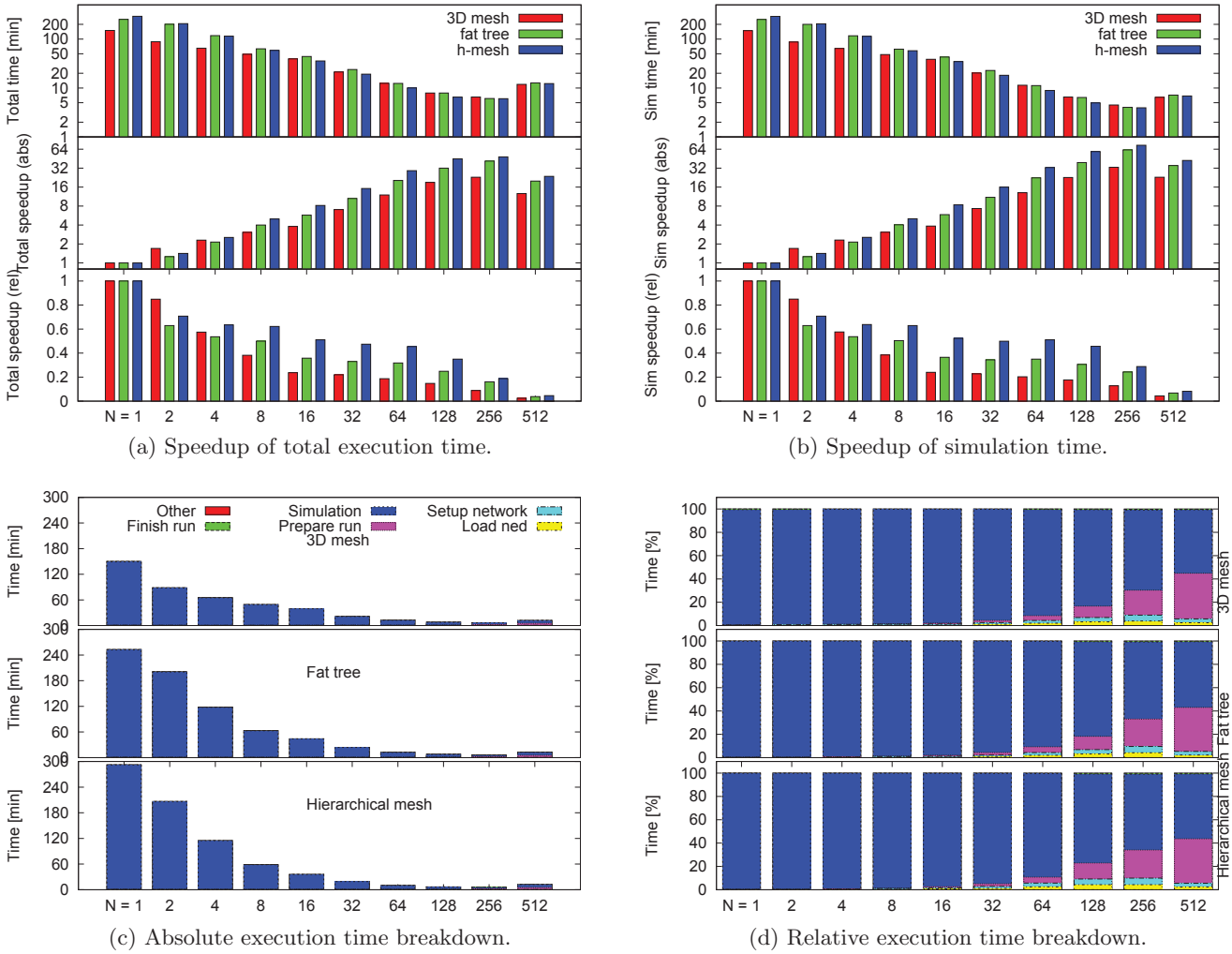
The root of the main problem we encountered were the Object Factory classes, which produced a runtime error about classes not being found, although they were properly linked in. The automatic class registration makes use of the names given to the classes by the compiler, and checks the list of registered classes whenever the model wants to instantiate an object of a registered class. Usually, a C++ compiler *mangles* class, data member and method names, embedding extra information regarding namespace, types, parameters, etc. The demangling worked for the GNU compiler but the required code was not compiled in when using the XL compiler. Fortunately, the mangling used by both compilers is very similar, so that we could reuse the demangling code by simply modifying the corresponding checks, i.e., by adding `__IBMCPP__` to the `#ifdef`s in `sim/util.cc`.

6.3 Results

Figure 4 shows results obtained on BG/P, each subfigure containing three graphs (top to bottom) for the run time, absolute speedup, and relative speedup, all as a function of the number of LPs N ranging from 1 (serial execution) to 512. Figure 4a corresponds to the total execution time, whereas Fig. 4b corresponds to just the simulation part of the run, excluding overhead such as ned-file loading, parsim lookahead discovery, model initialization, and finalization. Each graph contains one set of bars per topology (3D mesh, hierarchical full mesh, and fat tree).

Speedup improved up to 256 LPs; beyond that, the added overhead of adding more LPs was greater than the benefit. As N increased, the total speedup started lagging the simulation speedup, e.g., for the h-mesh at 256 LPs the corresponding speedup factors are 48 (total) to 73 (simulation). The cause for this discrepancy is illustrated by Figs. 4(c, d), which break the total execution time of each run down into its components (expressed by their absolute (c) and relative (d) contributions): load ned files (yellow), setup network (cyan), prepare run (pink), simulation (blue), finish run (green), and other (red). Most notably, the relative contribution of the "prepare run" part grew significantly with increasing N, accounting for close to 40% of the total run time with 512 LPs.

For large N by far the largest contributor to the initialization time was the lookahead discovery, i.e., exchanging messages among LPs to discover inter-LP lookaheads, which depends strongly on N. To highlight this, Fig. 5a shows the rel-



(a) Speedup of total execution time.

(b) Speedup of simulation time.

(c) Absolute execution time breakdown.

(d) Relative execution time breakdown.

Figure 4: Strong scaling results for BG/P for networks with 4,096 end nodes. In all graphs the x-axis represents the number of LPs N.

ative breakdown comparing fat tree and h-mesh topologies with 1,024, 4,096, and 16,384 end nodes simulated using 256 LPs. Run preparation also includes invoking `initialize()` on all modules. In the worst case, run preparation consumed almost two-thirds of the total execution time! This effect seriously degrades the overall speedup; for instance, for the 16K-node cases, the pure simulation part achieved superlinear speedup of 2.1× for the h-mesh and even 2.7× for the fat tree by doubling the number of LPs from 128 to 256, whereas the corresponding overall speedup factor was just 1.3× in both cases.

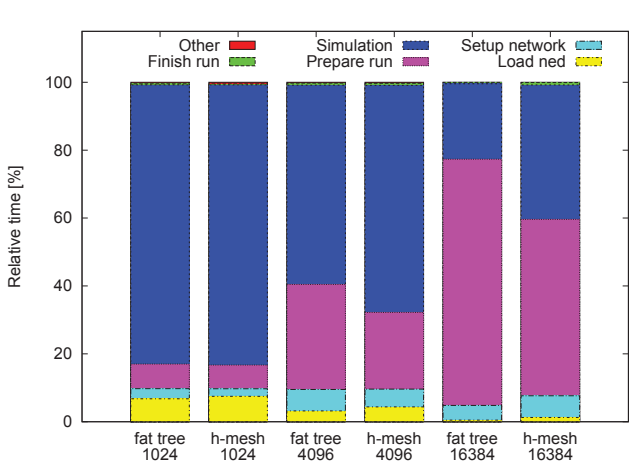
Other contributors to the initialization time were network setup and ned file loading, but these are independent of N.

We quite consistently obtained best speedup results for the hierachical mesh topology, followed by the fat tree, with the 3D mesh coming in last. A possible explanation is that the diameter of the mesh is significanty larger than that of the other networks, implying that the mean number of hops per packet for uniform traffic is also proportionally larger. This translates to a larger number of LP boundary crossings and therefore a higher communication to computation ratio. The top graph of Fig. 5b shows the time spent in MPI communications for all three topologies, indicating the minimum, median, and maximum across all LPs for each run. Most striking are the enormous spreads for the mesh topology, which indicates a severe communication imbalance.

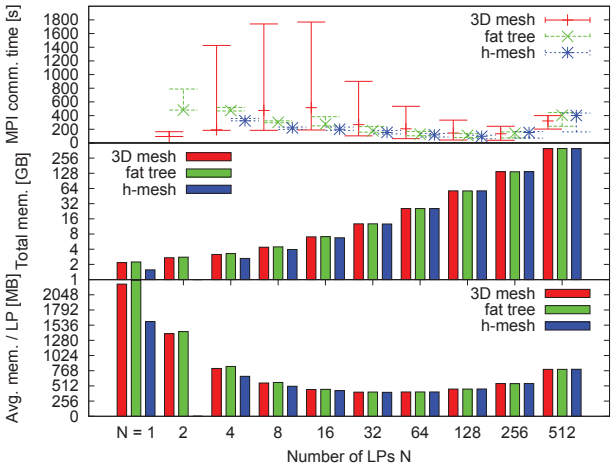
Finally, the middle and bottom graphs of Fig. 5b show total and per-LP memory usage as a function of N for all three topologies with 4,096 end nodes. As we increased N, the per-LP memory usage first clearly decreased (up to $N = 8$), then remained fairly constant up to about 64 LPs, and beyond 64 increased again. Nevertheless, the per-LP memory footprint was well below 1 GB for all runs. A large part of this memory was consumed by the MPI communication buffer, which we had to enlarge as N increased to prevent buffer overflows from crashing the simulation.

7. CONCLUSIONS

As the demand for computational power grows and technology advances, HPC systems and their interconnection networks are becoming larger and more complex. To study the performance of such systems, discrete event simulation



(a) Execution time breakdown for 256 LPs for 6 different topologies.



(b) MPI communication time (top), total (middle, log-scale) and per-LP (bottom) memory usage.

Figure 5: Simulation time, communication, and memory overheads.

is an important tool. Nevertheless, the need to simulate ever larger and more complex models puts new emphasis on the scalability of such tools.

We used Omnest's PDES capabilities to simulate large networks in parallel on three different platforms. On an SMP machine, we obtained excellent speedup results, but were ultimately limited by the available RAM. On a cluster, we not only achieved good speedups, but also achieved scalability to beyond 100'000 end nodes. Finally, we ported Omnest and Venus to the IBM Blue Gene platform, overcoming various portability and compiler issues, and obtained promising initial results that demonstrated performance improvements up to 256 LPs even for a relatively small problem size.

We intend to further investigate the scaling issues that became apparent with the current Omnest/Venus implementation on BG, in particular the per-LP memory overhead (including MPI buffer), the excessive initialization times, the MPI communication overhead, and achieving deterministic event handling order indepedent of the number of LPs.

Acknowledgments

The authors would like to express their sincerest appreciation for the Omnest team and Andras Varga in particular for their excellent support and high responsiveness to our questions and problem reports.

This work was funded in part by the U.S. Department of Defense and used elements at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory and funded by the U.S. Department of Defense.

8. REFERENCES

[1] Adiga, N.R., *et al.*. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, Vol. 49, No. 2/3, March/May 2005, pp. 265-276.

[2] Chandy, M., Misra, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, vol. 5, pp. 440–452.

[3] Denzel, W. E., Li, J., Walker, P., Jin, Y. A framework for end-to-end simulation of high-performance computing systems. In *Proc. of the First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008)*, Marseille, France, March 3–7, 2008, article No. 21..

[4] Gara, A., *et al.*. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, Vol. 49, No. 2/3, March/May 2005, pp. 195-212.

[5] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H. TOP500 Supercomputer Sites, 2011 (accessed Nov. 22, 2011). `http://www.top500.org`.

[6] Minkenberg, C., Rodriguez, G. Trace-driven co-simulation of high-performance computing systems using OMNeT++. In *Proc. of the SIMUTools 2nd International Workshop on OMNeT++ (OMNeT++ 2009)*, Rome, Italy, Mar. 6, 2009.

[7] Park, A., Fujimoto, R.M., Perumalla, K.S. Conservative synchronization of large-scale network simulations. In *Proc. of the Workshop on Parallel and Distributed Simulation (PADS), ACM/IEEE/SCS*, May 16–19, 2004, Kufstein, Austria, pp. 153–161.

[8] Petrini, F., Vanneschi, M. k-ary n-trees: High-performance networks for massively parallel architectures. In *Proc. of the 11th International Symposium on Parallel Processing (IPPS '97)*, Apr. 1–5, 1997, Geneva, Switzerland, pp. 87–93.

[9] Varga, A. OMNeT++ User Manual (accessed Nov. 23, 2011). `http://www.omnetpp.org/doc/omnetpp/manual/usman.html`

[10] Varga, A., Sekercioglu, Y. A., Egan, G. K. A practical efficiency criterion for the null message algorithm. In *Proc. of the European Simulation Symposium (ESS 2003)*, Oct. 26–29, 2003, Delft, The Netherlands.