UNIVERSITY NAME (IN BLOCK CAPITALS)

Validation and improvement of the SMPI simulation framework for MPI applications

by

Attila Döme Lehóczky

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty Name
Department or School Name

2013. június 27.

Declaration of Authorship

I, AUTHOR NAME, declare that this thesis titled, THESIS TITLE' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

„Write a funny quote here."

If the quote is taken from someone, their name goes here

UNIVERSITY NAME (IN BLOCK CAPITALS)

Abstract

Faculty Name
Department or School Name

Doctor of Philosophy

by Attila Döme Lehóczky

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Abbreviations

LAH **L**ist **A**bbreviations **H**ere

Physical Constants

Speed of Light $c = 2.997\ 924\ 58 \times 10^8\ \mathrm{ms}^{-\mathrm{S}}$ (exact)

Symbols

a	distance	m
P	power	W (Js^{-1})
ω	angular frequency	rads^{-1}

For/Dedicated to/To my...

1. fejezet

Introduction

Distributed computing has become a very important subject in computer science. There are multiple types of large-scale distributed environments that can be used for either production purposes or for research. Parallelism is also used in a lower level, for example in graphics processing, where we can have multiple graphics chips in one computer to do the task.

When doing distributed computing, communication between the processes becomes a very important concern, since it can pose a relatively large overhead compared to sequential problem-solving - to make parallelism worthwhile, we have to make sure that the speedup provided by the distribution of tasks makes up for the communication overhead. To achieve this, task distribution needs to be carefully planned and the communication protocols are needed to be optimized. A widely utilized communication protocol that has been under development for many years is provided by the MPI[1] inter-process, language-independent communication API. MPI itself is just a specification, it has to be implemented. Many such implementations exist, the most widely used ones include OpenMPI[2] and MPICH[3].

Setting up a distributed environment is a complicated endeavour: it needs both human and monetary resources. When doing research on distributed computing, our needs possibly exceed the use of just one single platform: we would like to test how our experiments fare on multiple different environments. Such environments are not always at our disposal. This is why simulation has become a very important field of research. SimGrid[7] is a project providing a wide range of features in this regard: it is a scientific instrument that can be used to simulate large-scale distributed systems in order to study their behavior by evaluating and analyzing the results of parallel experiments on them. As mentioned before, inter-process communication is a very important concern for these experiments. SMPI is a framework that is part of the SimGrid project. This framework makes it possible to simulate the execution of parallel applications that use the MPI

standard. This simulation can be done on a single node.
SMPI is a well-documented and working framework, but also an active project and as such, under constant development. Extensive testing is always needed, as it is very important that the behavior of the application is correctly represented by the simulator. This testing process is currently very time-consuming. This doesn't only limit the number of tests, but also limits the reproducibility of the results that are achieved with SMPI. By constructing a framework that simplifies the testing process, more reliable and verifiable results could be produced, as well as it would make the SMPI project members' lives easier. This thesis discusses how such a framework could be built and provides an implementation, utilizing #TODO.

2. fejezet

Literature Review

2.1. Introduction

In the literature review, we discuss the area of research covered by the thesis, citing relevant papers and articles that serve as a base of ideas for this document.
First, we talk about the Message-Passing Interface (MPI), the parallel programming API used for the purposes of this thesis. We also talk about the different implementations of this API, notably OpenMPI and MPICH. We discuss the methods of modeling and simulation in general. Then, we talk about SimGrid, which is a simulation-based framework, and SMPI, the implementation of MPI that runs on top of SimGrid. While discussing SMPI, we present the idea of a framework that would make it possible to automate running tests, trace collection and post-processing. After that, we also describe StarMPI, a set of MPI communication routines, presenting a technique that can improve the performance of MPI applications.

2.2. MPI

Distributed computing is a very active and important subject of research in computer science, including fields such as cluster computing, grid computing, Cloud computing, or peer-to-peer computing. Communication between the different processes in a distributed application can be implemented in a number of ways. As communication is necessary in most cases, a standardized communication protocol can be a lot of help when developing a distributed program. The Message-Passing Interface (MPI) is a language-independent message-passing library interface specification. It is not a language, but a standard - there exist multiple MPI implementations. Since its take-off, it has become a de facto standard for inter-process communication. The standard provides vendors a clear set of

routines, that they can implement efficiently, or in a way that it suits the hardware they provide.[1]

2.2.1. OpenMPI

OpenMPI is an MPI implementation with the goal of being able to achieve good performance on a wide range of different aspects of high-performance computing. To efficiently support multiple types of parallel machines, high performance "drivers" for all established interconnects are developed. These include TCP/IP, shared memory, Myrinet, Quadrics, and Infiniband. Features for checking data integrity are provided in order to account for network transmission errors. With the utilization of message fragmentation and striping over multiple (potentially heterogeneous) network devices, OpenMPI provides an increased bandwidth to applications, as well as the ability to handle the failure of network devices during runtime.[2] On the Grid'5000 cluster, which is used to most of the research conducted for this thesis, OpenMPI is the default MPI implementation used by the default images.

2.2.2. MPICH

MPICH was originally developed during the MPI standards process starting in 1992 to provide feedback to the MPI Forum on implementation and usability issues. This original implementation was based on the Chameleon portability system to provide a light-weight implementation layer (hence the name MPICH from MPI over CHameleon). Around August 2001, development begun on a new implementation called MPICH2.[3] This implementation introduced improvements on collective communication operations by using multiple algorithms, choosing between them depending on certain variables - for example the message size.[4] Another important result during the development of MPICH2 was the design of the Nemesis communication subsystem and the porting of MPICH2 on that system. The efficient implementation of shared-memory communication helped Nemesis MPICH2 achieve low latency and high bandwidth.[5]
Starting with November 2012, the project is renamed to MPICH, with version number 3.0.[3]

2.3. Modelling and Simulation

In distributed computing, modelling means creating an abstraction of a real system by taking only the aspects of it that are relevant to the system's behavior into account. Once

constructed, such a model becomes a tool with which we can investigate the behavior of the system.[6]

2.3.1. Advantages of Modeling

Modelling and simulation techniques have been used extensively in parallel computing and is an ongoing research topic, with new challenges continuously arising. There are various reasons for its importance.
Conducting experiments on real-world systems can be infeasible because experimenting would disrupt the service that is provided by the system. For example, in the case of a mail server, experiments or monitoring could cause delay, or maybe even data loss. Service disruption can sometimes be even dangerous, in addition to being an inconvenience: in the case of a nuclear reactor, delay or loss of data can prove fatal. Timeliness can be as important in such systems as correctness. However, performance analysis and monitoring might be crucial to draw conclusions about maintenance, for example. Another problem with direct experimentation is that the information we are looking for may not be available, or may be complicated to get. For example, in most operating systems, it is difficult to obtain the exact timing of instruction-level events.[6] Also, when conducting experiments on a real-world system, results are often non-reproducible, due to resource dynamics.[7] Another argument on the side of modelling is that it provides the ability of experimenting on different configurations. Investing in a large-scale computer cluster, or the setup of a distributed grid environment is an expensive and tedious process. Investors want to make sure that they get what they want: they impose performance constraints on the system. This means that they want to know how the system will behave before buying it and setting it up. To predict the behavior, experiments are needed to be conducted. We need to do these experiments on different setups, before finding out which one is the best in the current situation. Changing the hardware or software configuration parameters on a real-world system is very inconvenient - in most cases, it's not doable, because of time and money constraints. Thus, the solution is to simulate the desired system, and run the experiments there. This way, changing the configuration is simple and costless.[6] Another great benefit of simulation is that in a classroom setting, students can learn the principles of high-performance and distributed computing without actual access to a parallel platform.[8]

2.3.2. Analytical and Simulation Models

The accuracy of a model can vary: we can make an analytical, or qualitative model, in which all definite values are abstracted away - in this case, we get a representation of

the system, which can be analysed mathematically to deduce its behavior. When using this method, no experiments can be conducted, we solely rely on theoretical analysis. In contrast, a simulation model is a stochastic model, which is an algorithmic abstraction of the real-world system that can be executed to reproduce the system's behavior. This model is also called a quantitative model, as we can get estimates of the modeled system's quantitative attributes, such as response time or throughput. In other words, we can use a simulation model to conduct performance analysis on a system, without actually having the actual system at our disposal.[6][9]
When wanting to get a prediction about how a specific system would perform, a theoretical model, in most cases, produces unreliable and unrealistic results - it's not feasible for such accurate predictions. The vast majority of research results are obtained via empirical evaluation of experiments.[7] For these reasons, we use the simulation model in this thesis. As we stated before, such a model can be executed, which is called simulation. During simulation, the model is supposed to behave like the real system would. It is hard to produce a 100% accurate simulation, but more and more reliable solutions are being developed. The simulation model contains more aspects of the real system compared to the theoretical model, in order to accurately represent the system, while still avoiding unnecessary detail.[6] Creating and executing a simulation model is complicated, computationally expensive and poses a number of challenges, thus, a good simulation framework (such as SMPI) can prove to be of much help when conducting experiments.

2.4. Off-line and partial on-line simulation

Full simulation - including CPU and network emulation - of a parallel application can be, in many cases, even more resource-intensive than running real-world experiments. This contradicts the fact that one of the most prominent goals of simulation is to observe the behavior of such large-scale platforms that aren't available. Thus, there is much interest in more efficient simulation approaches. [10] The most widely used of such approaches fall into two categories: off-line simulation, which is also called trace-based or post-mortem simulation and on-line simulation, which is simulation via direct execution.[8] As in the subject of this thesis, we are interested only in the simulation of MPI applications, we describe the two different simulation approaches concentrating specifically on that subject.

2.4.1. Off-line simulation

For conducting off-line simulation, logs or traces are needed to be collected of an execution of the MPI application to be simulated, taking place on a real-world platform. This is necessary because the obtained traces are used as an input for the simulator, which then replays the execution traces as if the application was running on the target platform. This platform's characteristics may differ from the one's that we obtained the traces from, since we may want to use the simulator to predict the application's performance on a different system. Thus, there is a need to calculate how the target platform would execute the application, based on the traces we got on the other platform. The typical approach to this problem is to first compute the time intervals between the MPI communication operations. During these intervals, local computations were conducted, that's why we call these "CPU bursts". During simulation, we have to account for the differences between the performance of the platforms by modifying the time these CPU bursts take. This can be done by simply scaling the time intervals, or by using more sophisticated methods, by calculating exactly how the application's computational signature and the platform's hardware signature relate.[8] Communication operations, of course, also need to be simulated. This is done based on the events recorded on the trace, and on the network model of the simulated platform.[8]
As mentioned in [8], there are multiple downsides and challenges to the off-line approach. On such downside is that when wanting to simulate a relatively larger-scale application, the size of the obtained traces can be so large, that running the simulation on a single node might become a problem. Methods in order to overcome this obstacle include a compact representation of the traces in order to reduce its size. Another solution is to only consider a carefully selected subset of the obtained traces. A big disadvantage when using off-line simulation is that because we use the traces as an input to the simulator in order to replay the execution of the application, the simulation is dependant on the platform we collect the traces on. This means that, for example, there can be features in the obtained traces that might not be available on the target platform. In most cases, it is also necessary that the two platforms have the same number of nodes to run the experiment on. Although there has been a good amount of research done in the area, MPI itself and also the application might alter its behavior depending on problem and message size. Because of this, simulating the scaling of an application is a very hard, if not impossible task.[10]

2.4.1.1. Time-independent traces

Another link that ties the produced trace to the host platform occurs when we use timed traces, meaning that each traced event is associated to a time-stamp. Since the time

delays between the events are specific to the platform specification, the simulator has to apply a correction factor to these delays when running the simulation on the target platform. Thus, the simulator has to know precisely the specification of both the host and the target platform, in order to be able to calculate this correction factor. Another difficulty regarding that comes up regarding this problem is that actually calculating the correction factor is a tedious process. It can take a considerable amount of time, depending on how similar the host and the target platforms are.[11]
In [11], a solution to this problem is proposed: time-independent traces. Acquiring time-independent traces means that the traces won't contain any timestamps, breaking this link between the acquisition and the replay of the traces. In these type of traces, for each computation or communication operation, we log the volume of the operation (in number of floating point operations or bytes) instead of the time the execution took. This type of information, in most cases, does not vary depending on the platform the experiment is run on. The exceptions are the adaptive MPI applications that modify their execution path according to the execution platform.
[12] contains a guide describing how to acquire such traces on the Grid5000 platform. The guide was used to serve as a base for the process on how to acquire traces. Since the work in this thesis is mostly related to producing traces for validating on-line simulation, in which case time-stamps don't have an influence on the process (neither in a positive, nor a negative way), the extraction of time-independent information from the traces can be omitted in our case.

2.4.2. Partial on-line simulation

Partial on-line simulation is a different approach. Here, we execute the program with no or very little modification on a host platform, that tries to mimic the behavior of the target platform.[8] Computational tasks are executed on the hardware, but the timing and the delivery of the messages is calculated by the simulation environment. Thus, the simulator is responsible for mainaining the correct order of the events, both computational and communicational.[10]
A downside of the on-line approach is that since we actually execute the code, the resource needs for running the simulation is about as high or even higher (in case of needing an extra node to run the simulation component, for example) than it is for the actual experiment. Techniques have been implemented in order to help alleviate this problem. The basic idea is that the actual results of the experiments (for example, the result of multiplying two matrices) might not be important in our case: we are only interested in the *time* it takes to get those results on the target platform. This is why methods can be employed which trade off accuracy for performance. This idea might

not be feasible for experiments where data-dependent application behavior is vital, but a large portion of benchmarks can be indeed simulated this way, providing a reasonably accurate execution profile.
Although slower, on-line simulation is more general than the off-line approach, as it does not, in any way depend on some other platform - whereas in the case of off-line simulation, as we mentioned before, the trace is acquired on a different platform, with maybe specific application configurations, thus inevitably bringing dependencies.

2.5. SimGrid

For reasons mentioned before, simulation techniques have historically been widely utilised in several areas of computer science, e.g. microprocessor design, network protocol design. Due to this, a lot of effort went into developing the technology and as a result, widely used and reliable simulation frameworks have been developed in these areas. However, there hasn't been a well-developed standard simulation tool for what we talk about in this thesis: execution of distributed applications on distributed computing platforms. Rather, there has only been a number of in-house developed, highly specialized tools to satisfy the need of the community. SimGrid is a more generic simulation framework that is being developed to be one of the acknowledged and widespread tools for simulation in large-scale distributed computing.[7]
SimGrid's key features include:[7]

- A scalable and extensible simulation engine that implements several validated simulation models, and that makes it possible to simulate arbitrary network topologies, dynamic computational and network resource availabilities, as well as resource failures;
- High-level user interfaces for researchers (who are not necessarily computer science experts, but rather experts on their own field of research) to quickly assemble simulation prototypes in either C or Java;
- APIs for distributed computing developers to create distributed applications that can run seamlessly in either "simulation mode" or "real-world mode", in order to be able to test it on the simulated environment before actually deploying it.

SimGrid is a very active project, both in terms of research and in terms of development. It is a favored tool by researchers, which is proven by the increasing number of papers written where the research was conducted using SimGrid as a scientific instrument. In terms of development, the developer team envisions a number of directions for future

work: addition of a model for disk resources; extention of scalability to improve usability in the P2P domain; ability to dispatch simulated nodes over several physical machines.[7] Another important field of research for the SimGrid team is the implementation of the API that has already been mentioned: the Message-Passing Interface (MPI).

2.6. SMPI

As stated before, MPI is one of the most widely used APIs for communication between nodes in distributed computing. SMPI is a framework for simulating on a single node the execution of parallel applications implemented using the MPI standard. It is part of the SimGrid project and as such, it is built on the SimGrid simulation kernel, benefiting from its fast, scalable and validated network models. SMPI also extends the existing model with other techniques, such as a validated piece-wise linear model for data transfer times between cluster nodes. SMPI simulations also account for network contention - timing and delivery of the messages are determined using the network model of SimGrid.[8] A current limitation in SMPI is that it is unable to simulate high-performance networking hardware such as Infiniband. Thus, when wanting to compare simulation to real-life results, we have to make sure those results were gatheres using Gigabit ethernet.
Three of the main challenges for simulating an MPI application are:

- Accuracy: The prediction of the real-world execution time (the "simulated time") needs to be as accurate as possible, so that reasonable conclusions can be drawn from the experiments.
- Scalability: We want to be able to simulate large-scale applications within a reasonable timescale.
- Speed: It would be advantageous if the simulation time (the actual time of running the simulation) would be as low as possible, compared to the simulated time (the predicted execution time of the real-world application).

As for simulation methods, SMPI can be used for both off-line and on-line simulation, although the emphasis is more on the on-line approach, since it's actually a partial implementation of the MPI standard in itself, thus making it feasible for executing MPI experiments. More specifically, in SMPI, the goal is to be able to make such simulations on a single node. The most prominent challenges when doing this are the large CPU and memory requirements. SMPI provides some special techniques that help overcoming these challenges. The basic idea about trading off accuracy for performance has already been described in the previous section about on-line simulation. SMPI implements

multiple such techniques, allowing to run experiments with such high resource requirements that would otherwise be impossible to fulfill. Such a method in order to reduce CPU usage is to run the benchmark only on a subset of all the nodes, while in place of running the code on the others as well, we just insert the computation time that we got previously. Apart from CPU usage, we need to also account for the need for memory. A technique for that is "RAM folding": here, multiple simulated processes, that in SMPI are, in fact, simple threads, use the same reserved memory location, thus overwriting each other's data structures. Also, another implemented solution is to remove large data array references from the code, with the help of the compiler which can result in the complete removal of potentially large, now unreferenced arrays. Again, this obviously corrupts the results that the experiment program gives, but in the same time helps to simulate applications that would use such an amount of memory that just wouldn't be physically possible to provide in our testing environment, while still providing a reliable estimate of the performance.[10] These features are disabled by default, they have to be explicitly enabled by the user.

Extensive testing was conducted in [8] to verify the previously mentioned qualities of the framework. In these tests, the OpenMPI and MPICH implementations were used to serve as verification benchmarks: the same experiments were run using both MPI implementations, as well as simulated with SMPI. The results show that SMPI predicted the execution time of OpenMPI and MPICH applications for point-to-point, one-to-many and many-to-many applications with an average error value of under 10% in each cases. Using the aforementioned techniques to reduce the memory footprint, SMPI tests were successfully conducted on a scale of up to 448 processors. The results showed that the predicted execution times were underestimates with an average error value of 18.5%, which is higher than in previous experiments without these techniques. We have to note here, though, that certain tests weren't successful without the RAM-folding techniques, due to an out-of-memory error. This shows, that although it poses difficulties, reducing the memory usage is vital in SMPI.

As SMPI is an actively developed project alongside SimGrid, there are a number of research directions. One major development to the project would be a testing framework that would aim to lessen the burdens of testing as much as possible. The goal is to provide a unified method to set up experiments across different environments and to do it with as little necessary adjustments on user part as possible. Another direction is related to the optimization of the framework's performance, with the utilisation of ideas from the STAR-MPI project.

2.7. STAR-MPI

Self-Tuned Adaptive Routines for MPI Collective Operations (STAR-MPI) is a set of MPI collective communication routines that are capable of dynamically adapting to system architecture and application workload. The main idea lays in a technique called "delayed finalization of MPI collective communication routines" (DF). For each operation, STAR-MPI maintains a set of communication algorithms. The aim is to postpone the decision of which algorithm to use until after the platform and/or the application are known. This technique bears the potential of platform-specific or application-specific optimalization of an MPI application.[13]
A development idea for SMPI is to apply the same technique there, thus, in a sense, to "implement" STAR-MPI in SMPI. A set of potentially choosable algorithms could be implemented alongside a set of selector mechanisms. With the right mechanisms, the performance of SMPI could greatly improve.

3. fejezet

Problem Description

3.1. Reproducible research

New scientific ideas, developments and results are only useful when they are documented and published. It is vital that results are announced, so others can be aware of the latest developments on their field of research. This helps in creating a linked data cloud, used by scientists to incorporate various output of other research into their own, using previous results as "stepping stones" to achieve something new.[14] But simply publishing results is not enough in order for others to make use of them. Besides announcing the achievements, the other goal of scientific publications is to convince the readers that the results it presents are correct. Besides theoretical reasoning, papers in experimental science should provide a documented methodology describing how the author has gotten to those results.[15] The methodology has to be detailed and precise enough so other researchers can repeat the same steps, thus reproducing the same results. This is vital in order to provide the possibility to verify those results and to fully understand them. Reproducing the results also makes for a starting point for further development, as the described methods used for reproduction can be extended to achieve something more or something different in the same area of research, or repurposed to gain useful results in a completely different area. This subject is relevant to SMPI and one of the main goals of the testing and validation framework is to make developments in the area.

3.2. Testing framework

SMPI is an actively developed project and as such, a lot of tests are run and a lot of measurements are taken. Previous papers ([8] [10]) have shown, amongst other results, how

accurate the performance predictions SMPI makes are and how the time of the simulation can be lowered, while getting very little differences in simulated time (which means the predicted performance). These results are obtained through extensive testing and as development continues, more and more test data is needed for verification purposes. In order to get results, both real-life (RL) tests and on-line simulation tests using SimGrid (SG) are needed to be obtained and then the results need to be visualized, analyzed and compared. Comparison is very important, since this is how we can validate SMPI. All this needs to be done in as many different kinds of environments as possible. Currently, this is a tedious task that needs a lot of configuration by hand, as there is no unified methodology or automation for setting up experiments and obtaining results on different kinds of distributed environments and for different kinds of MPI implementations - one has to find his/her own way to make it work. Documentation only exists for specific systems, for example in [12], there is a guide about how to produce time-independent RL traces on the Grid'5000 testbed. Obviously, platform-specific guides don't always help with problems arising in an other environment.
Below comes a more detailed description of the process of acquiring traces.

3.3. Obtaining traces

3.3.1. On-line SG traces

In order to conduct tests on the simulator, the first step is to create the simulated environment. This is done by creating a platform file that can be fine-tuned to model the desired system. #maybe detail how a platform file looks like?#

3.3.2. RL traces

As talked about in detail in [12], conducting RL tests involves multiple steps. RL test data collection is done by collecting traces of MPI benchmarks. Currently, the favored tool in trace collection in the project is Tuning and Analisys Utilities (TAU)[16], which is a well-established profiling tool, that also provides tracing features. Thus, on the system where we are running the benchmarks, TAU has to be deployed and configured, alongside other software that TAU depends on. One is PAPI[17][18], an interface which provides us with the possibility to get access to low-level hardware counters (to trace the number of instructions at processor level). We also need the Program Database Toolkit (PDT)[19], which provides the ability of automatic performance instrumentation. In order for TAU to collect the traces we need, these toolkits have to be deployed and correctly linked with TAU. TAU has its own compiler scripts for MPI programs for both

Fortran, C and C++. After compiling a benchmark using one of these scripts, they will generate TAU trace files upon execution. One trace file (.trc) and one event file (.edf) is generated for each MPI process.
It is possible to visualize the traces, in order to compare the RL and SG results more easily. Paje is a visualization tool that can be used for this purpose. It has its own trace format that it can comprehend, thus the TAU traces have to be converted to that. Also, we need to merge the traces into one file that we can give to Paje after the conversion. There is a TAU script that is able to do this, creating one trace and one event file. We now only have the task of converting the TAU trace to a Paje trace file. Another MPI tracing library, Akypuera provides this possibility, having its own tau-to-paje conversion script. Once done, we finally have a trace file that Paje can read and display to us.
In the future, it is very likely that more and more tests will have to be run in order to verify old and newly implemented features, as well as various experiments will be conducted using SMPI to test it against various MPI platforms. The main reason of importance of developing an automated way for testing and validation is that it could make the previously discussed tedious trace-gathering process a lot smoother and faster, with less user interaction. If the process could be incorporated into a single workflow, that would make it much easier for the user to procure test results. This way, proportionally more tests could be run, providing more reliable results with less effort than before. Apart from making extensive testing and experimentation more straightforward, a functioning test and validation framework would provide a well-documented method, which could be used by other researchers to reproduce the achieved results, the importance of which is discussed above.

3.4. Tracing-related problems

Apart from wanting to simplify the previously described, fairly convoluted process of getting results, there are other problems related to tracing that need to be addressed. Since we need to compare RL and SG traces, it is very important that the traces we get are accurate, otherwise we could be lead to false conclusions. Below are the most prominent traps that could potentially compromise our results. Note that these are problems that come up in a real environment, thus only related to real-life traces.

3.4.1. Multiple cores on one node

Nowadays, it is common for a computer to have multiple processors (cores). Because of this, applications have been optimized to increase performance by utilizing more than one core. MPI implementations have also been optimized to migrate threads between

cores in certain cases. The problem with this is that SMPI is not configured to account for the speedup that the utilization of multi-core processors can cause. So if we compare the running time of an MPI application that was run on multi-core nodes and compare the running time to our simulation's running time that was done by using SMPI, even if we simulated the used platform correctly, we will likely see that SMPI overestimated the running time, since it didn't take into account the performance increase caused by using multiple cores.
A solution to this problem is to explicitly disable all cores but one on every node before running the real-life experiment.[12] The downsides to doing this are that we need to know the platform-specific instructions on the nodes which have to be given to disable cores, as well as we most likely need root privileges. But if we can use it, this method is a simplistic and sure way to solve this problem.
Another possible way to make sure that MPI doesn't make use of having multiple cores at its disposal is to specifically bind MPI processes to cores. For example for OpenMPI 1.4 and above, this can be done by using the processor affinity instruction parameter *--bind-to-core* when running the application.

3.4.2. Impact of instrumentation

In the context of the collection of traces, "instrumenting" an application means specifying what kind of data we need and which part of the program we need it from. This can be done either directly (by inserting function calls or macros that record the data we want), or by using an overlay library for this purpose. For example, in the case of TAU, there is a feature called selective instrumentation, which provides us with the possibility to specify which functions we want to be traced.
When instrumenting the application, it is important to know at what degree we want to do it. We want to collect the data we need, but the greater the degree of performance instrumentation in the program, the higher the likelihood that the performance measurements will alter the program's performance, an outcome called *performance perturbation*.[16] In the case of most performance tools, TAU included, this is a concern that the developers try to address by reducing the overhead of the performance measurements as much as possible. It is worth noting though that although the overhead the measurements cause might be reduced, they can't be completely eliminated, since the tracing operations have to be handled by the processor as well. Because of this, the user has to be very careful when instrumenting an application. There are two main concerns that come up in our case.

Time overhead. Instrumentation can have a sizeable negative impact on the performance of the application. If the instrumentation causes a noticeable increase in running time, the simulator's prediction might be off, since it doesn't take into account the instrumentation overhead (as instrumentation is not part of the experiment).

Impact on hardware counter values. As mentioned before, when collecting time-independent traces, we use hardware counters to measure the volume of each operation. The hardware counter doesn't distinguish between events related to the experiment and tracing operations, thus all of them are taken into account. The result is that the collected trace represents the traced experiment, while we want information from just the experiment, without the traces. If the difference is too big, this can make our simulations look inaccurate. The impact it can have when using off-line simulation, where the simulator replays the traces corrupted with this overhead is shown in [20].

In [20], the authors propose an instrumentation as a correction to their previous work in [11]. The problem was that there was a sizeable overhead caused mainly by TAU building the whole call path of the instrumented application. While it can be very useful when trying to find bottlenecks in the application, this information is not needed for simulation purposes. In the proposed method, we tell TAU to exclude all of the source files from the instrumentation. This way, instrumentation becomes minimal, while still covering our specific needs: the hardware counter will be triggered at each MPI call to measure the number of executed instructions in the operations. All of the information related to MPI calls, i.e. the id of the process that made the call, the name of the function and its parameters are traced. All this while both the time overhead and the hardware counter value discrepancies are considerably reduced, as shown in [20].

3.4.3. Clock synchronization

Another obstacle that comes up when wanting to analyze trace data generated in a distributed system is related to clock synchronization. When running a parallel experiment on a distributed system that uses multiple nodes, all the used nodes produce separate trace streams independently of each other. The problem is that between the local processor clocks of different nodes, there almost always exists some amount of discrepancy, mostly related to the temperature of the processor. No matter how little this discrepancy is, it can accumulate over time, as well as it can change the logical event order, which requires a message to be received only after being sent from the other node. This is also referred to as the *clock condition*.[21][22] Such inaccuracy in the traces can lead to false conclusions when doing the performance analysis. Even though in our work we

want to use time-independent traces, violations of the clock condition is still a problem we have to be able to handle.
The problem could be easily avoided if every process would use a global clock instead of the local clock of the node it's running on. The problem with this approach is that accessing the global clock can be much more expensive than accessing the local clock, thus causing performance issues, as well as it's not available on every platform. In [21], the authors use an IBM switch adapter's globally synchronized clock's register to periodically collect global clock records for each node, thus being able to correct the local clock discrepancies after the experiment finished.
The main problem with this is, as already mentioned, that although some systems do offer a relatively accurate global clock, many other systems are only equipped with processor-local clocks, in which case the problem has to be approached from another direction. There exist clock synchronization protocols, such as NTP [23], which provides a widely used solution to align the clocks to a certain degree by adjusting local clocks at regular intervals to a globally accessible time server. Unfortunately, due to varying network latencies, this method still leaves an error rate of about 1 ms when synchronizing, which is not good enough for our purposes. TAU handles this problem with another post-processing approach, using an extended and parallelized version of the *controlled logical clock* (CLC) algorithm, which is described in [22]. CLC retroactively corrects clock condition violations in event traces of message-passing applications by shifting message events in time. When making such a correction, CLC makes certain precautions, since after the modification of individual timestamps, the length of intervals between events in the immediate vicinity of the affected event might change, as well as new clock violations might be introduced.

3.5. Environment

3.5.1. The importance of testbeds

Most Grids that are deployed at a large scale are production platforms inappropriate for research: such Grids mostly have an environment that's been set up specifically for the owner's purposes. Such a system would most probably need some amount of reconfiguration in order to make it feasible for what we would like to do, which might influence the behavior of the system. Another concern is that running experiments on the platform might cause delays, or even disruption in the service the Grid is originally used for. Obviously, this is most probably not acceptable for the owner of the Grid. This is why it is important to make a distinction between production Grids and Grids that are made for testing purposes, or "testbeds". Because testbeds are specifically

made for researchers to run experiments on, using up resources is not that much of a concern as it would be on a production platform. Since other people might be using the same platform, there are of course still some limits as to how much resources one user can utilize, but these limits are not so strict and are much more prone to negotiation. As previously mentioned, another important factor is that the nodes we are working on should be reconfigurable: we need to be able to make customizations to set up our testing environment. We need to be able to install and uninstall programs. Root access should not be necessary when doing tests, but it can make things easier. Deep control and monitoring mechanisms are also needed (not just in one node, but across multiple nodes) in order to be able to track our experiments.
Simulation with SMPI can be done on any system, since it only needs one node but in order to run RL experiments, such a testbed is needed. Most of the work related to this thesis has been done on the Grid'5000 platform.[24]

3.5.2. Architecture of Grid'5000

Below, we discuss some of the architecture aspects of Grid'5000 to show how it fulfills our previously mentioned needs, as well as how it addresses some other concerns as well. Description details are taken from [24].

3.5.2.1. Networking

Grid'5000 is a platform currently consisting of 5000 CPUs, distributed across 9 different sites in France, connected by high speed network. These sites host their own clusters and they are connected through the Internet. It is very important with regard to the experiments that inter-site communication and inter-node communication are unrestricted and don't weigh any overhead on the experiments. Thus, all communication can be done without any constraints between sites. But as for security, we have to take into account the following: if a node is fully reconfigurable by the researcher, that means we can't make any assumptions about the configuration of the security mechanisms on an allocated node, thus, we have to assume that they are unprotected. This is the reasoning behind the decision that sites themselves (thus, of course, the nodes they are hosting) are not directly connected to the Internet, making Grid'5000 an isolated domain. This way, the sites are protected against DoS attacks.
It is possible to open restricted routes through the Internet to external clusters, which provides the possibility of doing multiplatform experiments.

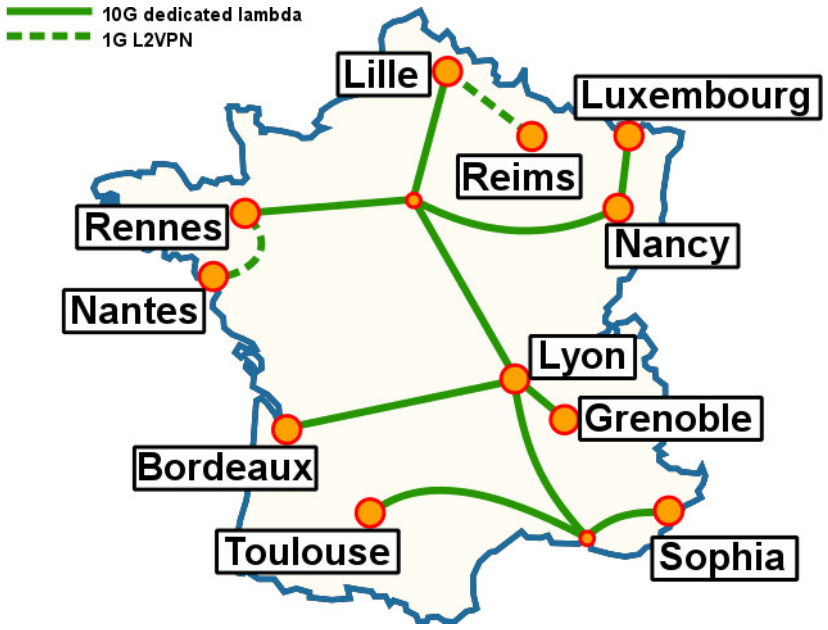

Figure 3.1. The sites of Grid'5000

3.5.2.2. User view and data management

As previously mentioned, communication is done with minimal restrictions between Grid'5000 machines, meaning that authentication procedures in such cases is also minimal: a user has a single account across the whole platform. An LDAP directory is installed to provide this in a reliable way. Every site runs an LDAP server. These servers have the same root and there is a branch for every site. On a given site, the local administrator has read-write access, as well as is able to manage user accounts.
A user has access to all of the Grid'5000 sites and services (monitoring tools, wiki, deployment, etc.). The user also has an independent home directory at every site as well. Synchronization can be done with any of the standard tools, such as *rsync*, *scp*, or *sftp*. Data transfers to the outside world are restricted - it can only be done via secure tools such as *scp* in order to prevent identity spoofing. Authentication is done via a user-generated public key, in order to prevent brute-force attacks.

3.5.2.3. Experiment scheduling

At cluster level, the OAR[25] resource management system is used to handle the scheduling of experiments and resource allocation. Large-scale operations such as parallel task launching or monitoring is handled by a specialized parallel launching tool, Taktuk[26]. Taktuk is a handy tool which can be used from the console to, for example, perform a certain set of tasks on multiple nodes.

A simple grid broker is handling resource management at grid level, allowing co-allocation of nodes on multiple clusters. Co-allocation is a very simple process for the user who, after submitting an experiment needing several sets of nodes across different clusters, receives an identifier from the broker which can be used to retrieve all necessary information about the allocated nodes.
Node reconfiguration, talked about in more detail below, co-operates with the resource management system at certain points. Such a point is that when a user submits an experiment that requires node reconfiguration, the job submission is registered in a queue. Also, in the prologue script that runs before the actual experiment, deployment rights are given to the user which gives him/her the capability to deploy system images on the allocated nodes. An epilogue script runs after the experiment, revoking these rights. After the experiment is finished, all of the allocated nodes are rebooted, deploying a default environment, to provide a constant, unified system to run experiments on that don't need node reconfiguration.

3.5.2.4. Node reconfiguration

Node reconfiguration on Grid'5000 is handled in a very user-friendly way, using a tool called Kadeploy3[27]. For every user, a set of default environments is available at start. After starting an interactive job (that is, a job that requires node reconfiguration), the user can deploy any of these environments by providing the chosen image's name to Kadeploy3. Deployment usually takes only a few minutes to complete - deployment time increases if we do the deployment on more nodes. After this, the nodes are rebooted and the user can log onto any of the nodes where an image was successfully deployed. When logged in, the user can freely customize the environment: he/she can install or remove software, modify configuration files, etc. - root password is given to Grid'5000 users as well to provide more possibilities. After reconfiguring the environment, the user has the the possibility of saving the now customized image. This image includes all software layers from OS to application levels, just as it was for the default environments. The home directory is independent of the image. After successfully saving an image, the user can deploy it on the allocated nodes the same way he/she did for the default images. This way, an environment tailored for the specific needs of the user only needs to be created once, then it can be freely reused on any other node on the site at a later time. When trying to port an image created on one site to another site, there can be compatibility issues due to inter-site differences. Modifications to the customized image can be done with ease.

4. fejezet

Implementation

5. fejezet

Evaluation Plan

6. fejezet

Results

7. fejezet

Conclusion

A. Függelék

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] Message Passing Interface, Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, September 2012.

[2] Gabriel, E., Fagg G., E., Bosilca, G., Angskun, T., Dongarra J., J., Squyres J., M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain R., H., Daniel D., J., Graham R., L., and Woodall T., S. Open mpi: Goals and concept, and design of a next generation mpi implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[3] , MPICH. Mpich overview, 2012. URL http://www.mpich.org/about/overview/.

[4] Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in mpich. *Int'l Journal of High Performance Computing Applications*, pages 49–66, 2005.

[5] Buntinas D. Mercier, D. and Gropp, W. Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem. *Parallel Computing*, 33:634–644, September 2007. URL http://linkinghub.elsevier.com/retrieve/pii/S0167819107000786.

[6] Hillston, J. Performance modelling and lecture 1 ("modelling and simulation"), 2012. URL http://www.inf.ed.ac.uk/teaching/courses/pm/PM-lecture1.pdf.

[7] Casanova, H., Legrand, A., and Quinson, M. Simgrid: a generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Event on Computer Modeling and Simulation*, pages 126–131, 2008.

[8] Clauss, P.-N., Stillwell, M., Genaud, S., Suter, F., Casanova, H., and Quinson, M. Single node on-line simulation of mpi applications with smpi. In *International Parallel & Distributed Processing Symposium*, pages 664–675, May 2011.

[9] Hillston, J. Performance modelling and lecture 13 ("simulation models: Introduction and motivation"), 2012. URL http://www.inf.ed.ac.uk/teaching/courses/pm/PM-lecture1.pdf.

[10] Bédaride, P., A., Degomme, Genaud, S., Legrand, A., Markomanolis G., S., Quinson, M., Stillwell, M., Suter, F., and Videau, B. Improving simulations of mpi applications using a hybrid network model with topology and contention support. 2013.

[11] Desprez, F., Markomanolis G., S., Suter, F., and Quinson, M. Assessing the performance of mpi applications through time-independent trace replay. In *Proceedings of the 2nd International Workshop of Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 467–476, Taipei, Taiwan, September 2011.

[12] G. S., Markomanolis and F., Suter. Time-independent trace acquisition framework – a grid'5000 how-to. Technical report, Institut National de Recherche en Informatique et en Automatique, 2011. URL http://hal.inria.fr/inria-00593842.

[13] Faraj, A., Yuan, X., and Lowenthal D., K. Star-mpi: Self tuned adaptive routines for mpi collective operations. In *The 20th ACM International Conference on Supercomputing*, pages 199–208, June 2006.

[14] Bechhofer, S., Ainsworth, J., Bhagat, J., Buchan, I., Couch, P., Cruickshank, D., Roure D., D., Delderfield, M., Dunlop, I., and Gamble, M. Why linked data is not enough for scientists. In *e-Science and 2010 IEEE Sixth International Conference*, pages 300–307, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5693931.

[15] Mesirov J., P. Accessible reproducible research. *Science Magazine*, 327:415–416, 2010. URL http://statlab.bio5.org/foswiki/pub/Main/PapersForClassCPH685/science-reproducible-research.pdf.

[16] S. S., Shende and A. D., Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20:287–311, 2006. URL http://statlab.bio5.org/foswiki/pub/Main/PapersForClassCPH685/science-reproducible-research.pdf.

[17] Mucci P., J., Browne, S., Deane, C., and Ho, G. Papi: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, California, USA, June 1999.

[18] London, K., Moore, S., Mucci P., J., Seymour, K., and Luczak, R. The papi crossplaftorm interface to hardware performance counters. In *Department of Defense Users' Group Conference Proceedings*, Biloxi, Mississippi, USA, June 2001.

[19] Lindlan K., A., Cuny, J., Malony A., D., Shende, S., Mohr, B., Rivenburgh, R., and Rasmussen, C. A tool framework for static and dynamic analysis of object-oriented

software with templates. In *Supercomputing, ACM/IEEE 2000 Conference*, Dallas, Texas, USA, 2000.

[20] Desprez, F., Markomanolis G., S., and Suter, F. Improving the accuracy and efficiency of time-independent trace replay. Technical report, Institut National de Recherche en Informatique et en Automatique, October 2012.

[21] Wu. C., E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., and Gropp, W. From trace generation to visualization: A performance framework for distributed parallel systems. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 50–67, Dallas, Texas, USA, 2000.

[22] Becker, D., Rabenseifner, R., Wolf, F., and Linford J., C. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing*, 35: 595–607, 2009.

[23] Mills D., L. Network time protocol (version 3). Technical report, The Internet Engineering Task Force - Network Working Group, March 1992.

[24] Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E., and Touche, I. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20:481–494, November 2006.

[25] Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., and Richard, O. A batch scheduler with high level components. In *IEEE International Symposium on Cluster Computing and the Grid*, volume 2, Cardiff, Wales, UK, May 2005.

[26] Claudel, B., Huard, G., and Richard, O. Taktuk, adaptive deployment of remote executions. In *Proceedings of the International Symposium of High Performance Distributed Computing (HPDC)*, Munich, Germany, June 2009.

[27] Jeanvoine, E., Sarzyniec, L., and Nussbaum, L. Kadeploy3: Efficient and scalable operating system provisioning. *Usenix*, 38:38–44, February 2013.