Jül-4297

Jülich Supercomputing Centre (JSC)

Trace-based performance simulation of large-scale applications

Marc-André Hermanns

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Trace-based performance simulation of large-scale applications

Marc-André Hermanns

Berichte des Forschungszentrums Jülich; 4297
ISSN 0944-2952
Jülich Supercomputing Centre (JSC)
Jül-4297

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL) unter
http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Acknowledgements

This thesis would not have been possible without the support of a lot of different people, some of whom I want to thank in particular.

I am grateful to Prof. Dr. Wolfram Schiffmann of the Fernuniversität Hagen and Prof. Dr. Felix Wolf of the Forschungszentrum Jülich for supervising this thesis and contributing with their valuable comments.

Additionally, I thank Dr. Markus Geimer for all the time we spent in technical discussions on load balancing and performance simulation, and Dr. Brian Wylie for his help in providing measurement data and understanding the analysis data of the XNS application. Their comments have been very helpful to this thesis.

I also express my gratitude towards my departmental managers Dr. Rüdiger Esser and Dr. Norbert Attig, for their flexibility and support during my course of studies.

Furthermore, I thank my parents Brunhild and Helmut for their continuous love and support during my life. Additionally, I thank my parents-in-law Rita and Barthold for their support of me and my family during the sometimes exhausting times of this thesis.

I am indebted to my wife Annette for her unfailing love, support, and endurance during the years of my studies as well as to my daughters Ira and Kara, for sharing me on so many weekends with my studies.

Marc-André Hermanns
Jülich, March 2008

Abstract

The massively parallel computer architectures emerged in the last years create the platform to redefine the limits of todays scientific simulations. To exploit these platforms efficiently, applications need a yet unprecedented scaling behavior to several thousands of processes. The complexity of systems of this magnitude presents a challenge to performance prediction in general. Sometimes it is feasible to extrapolate from small-scale execution behavior to larger scales, however, this is not always the case. Applications and algorithms that change their behavior significantly when executed with a high number of processes will then have to be investigated on the corresponding scale.

Several performance analysis tools exist to help the developer of large-scale simulations to identify performance critical phenomena in their application execution. Choosing the best optimization strategy is still highly depending on the experience of the performance investigator with the application, its algorithms, the analysis tools, and the computer architecture as well as its software. Cost-benefit ratios of specific code optimizations are often hard to estimate precisely. Prediction of a modified application's execution behavior on large scales can help with estimating the cost-benefit ratio of a specific application modification.

This thesis introduces the application performance simulator Silas, which uses an event-trace-based approach to model and predict application execution behavior. Its focus lies on the simulation of hypothetical code optimizations, based on the modification of an existing execution trace. It is embedded in the performance analysis tool Scalasca, which is a scalable set of tools supporting performance investigators in optimizing large-scale applications. Simulated optimizations may include the scaling of region instances, the balancing of parallel region instances, and the deletion of region instances and message transfers from the event trace. A model for trace-based performance simulation as well as the needed event-trace manipulation to simulate optimizations are presented. The implementation of specific parts of the simulator is discussed and test results of Silas investigating synthetic and real-world applications to demonstrate its effectiveness are presented.

Contents

List of Figures

1 Introduction

The performance of parallel applications is mainly influenced by two factors: single core performance of the code as well as its scalability to multiple cores. Even though processor speeds have increased in the past according to Moore's Law [21], the theoretical performance improvements that can be achieved with a single computing core are mostly saturated, leading to the emerging multi-core architectures. But even multi-core architectures still pose very tight constraints on the degree of parallelism possible today. The aggregated performance needed for current scientific simulations therefore calls for massively parallel systems with thousands of nodes, where each node might comprise several cores. In existing supercomputer systems of architectures like the Blue Gene Solution of IBM, a single application might be started on almost 300K cores. This results in very high demands on the applications' scalability. The main threat to application scalability is inter-process communication and synchronization. Applications with no shared resources and no inter-process communication will almost always show perfect scaling, however, not all scientific challenges can be modelled this way. An increase of the problem size often implies a distribution of subparts to different processes, leading to dependencies between the processes. Here, serialization of access to common resources or simple waiting times in communication can easily add up to a significant amount, where Amdahl's Law[1] will lead to an asymptotic maximum of scalability. Scientific projects mostly focus on the science behind the simulation and are therefore interested in new results in this field, mostly ignoring the need for an optimized application. Those applications have provided good results for many years, why would they now not be feasible anymore? With massively parallel symmetric multiprocessor systems, comprising thousands of computational cores, it is already evident today that only few applications are fit to run efficiently on those architectures. The productivity of performance analysis and optimization tools therefore has to reach a level where it becomes feasible to optimize an application and produce fresh scientific results within a rather short time. It is therefore of highest importance to identify wasteful waiting times within the parallel application run, and to improve application performance and scalability.

In this context, event-based performance analysis is a well-known and accepted method for performance analysis of computer applications. Reasoning about an application's performance is based on an event model defining all performance relevant events as well as their semantics and relations. The application run is modelled as a traced stream of events between two reference points, often the application start and end. The use of partial traces, which do not cover the complete execution of the application, is also possible.

The Scalasca tool set is a scalable set of tools, which emerged from the Kojak project and implements a knowledge-based, automatic search for performance relevant communication and synchronization patterns in traces reflecting the application behavior. With this approach, it is possible to automatically expose waiting times on processes linked to another process' behavior. The time lost due to these waiting times defines the severity of a specific pattern. Severities of instances of inefficiency patterns are summarized to present the most severe communication and synchronization patterns to the user. This is defined as the performance bottleneck – the most severe performance problem of the application.

In parallel applications, the symptoms of a performance bottleneck may appear (i) much later than the event causing it, (ii) on a different processor, or (iii) both. The temporal or spatial distance between cause and symptom constitutes a major difficulty in deriving helpful conclusions from a set of performance data.

An example for the first category is load imbalance that creates wait states at the next synchronization point following the imbalance. Since some processes arrive later at this point due to a higher share of the overall load, those arriving earlier have to wait. Since wait states can occur as the result of a superposition of several phenomena, it is hard to determine what the actual contribution of a given imbalance is.

When optimizing an application the outcome of actual code changes can only be estimated. Questions like "What would be the impact on application performance, when I optimize function `foo` to be twice as fast?" or "Is the waiting time in the communication call `x` caused by an imbalanced load in the function `bar`?" can usually be answered only after the change to the code has been made and a run of the optimized code can be measured and analyzed, as they are too complex to answer without some kind of experimental observation of the application performance.

Simulation of hypothetical changes to the code, which are based on a well-defined performance prediction model, can investigate performance implications much faster than actual changes in the application code can do. The simulator introduced in this thesis has the specific goal of helping the user to optimize his application within a given environment, with a focus on scalability of the application. It therefore contributes to productivity in general, which is becoming a major focus in high performance computing application development.

This thesis introduces a trace-based simulator for application performance. It is embedded in the Scalasca tool set and serves as a method of testing hypotheses on possible program behavior. A hypothesis is encoded as a set of functions that manipulate the original trace.

This thesis is organized as follows: Chapter 2 introduces the Message Passing Interface MPI, which is widely used as the communication paradigm in many parallel scientific applications today. Additionally, it gives a short presentation of event-based performance analysis and provides details on the event model defined and used by the Scalasca tool set for the event traces the simulator is processing. Chapter 3 introduces replay-based performance prediction, and describes the proposed optimization hypotheses. A description of a reference implementation of a simulator model is given in detail in Section 4, as well as implications to the simulator quality. Finally, Chapter 5 presents performance prediction results, for both synthetic applications and real world scientific simulations. Chapter 6 summarizes the findings of this work on performance prediction and concludes this thesis.

2 Background

2.1 Introduction

Scientific computing is the third pillar of research, next to theory and experiment. Simulations of physical models give scientists a much more detailed insight into phenomena that cannot be subject to experimentation, be it either large scale, such as the simulation of the creation of galaxies or black holes in astrophysics, or nano-scale with simulations on molecular level. Even engineering sciences and industry applications rely more and more on simulations to gather data on their research projects. Simulations of different aspects and different levels of detail and accuracy will increase the scale at which simulation has to be conducted.

In the past years the power of individual processors was steadily increasing, following *Moore's Law* [21], by doubling the number of transistors on a single chip every two years. This lead to increasingly higher clock rates for the chips, which eventually saturated during the last years, leaving the peak clock rates for the processors at about 3-4 GHz, as the power consumption of higher clock rates was no longer satisfiable. Now, more transistors are brought to several cores on a single chip, which lead to the currently emerging multi-core architectures that will dominate the next parallel architectures. This has direct impact on scientific applications today, as their degree of parallelism will have to adapt to the hardware platforms provided. Applications that remain static in their degree of parallelism will have trouble to compete with others that show a better scaling behavior.

Figure 2.1 shows the aggregated number of cores on the top 20 computer systems in the years 2000 to 2007 of the Top 500 list, which announces the fastest computer systems in the world twice a year. Since the beginning of 2005, the number of cores in the high end computer systems is rapidly increasing. This trend is continuously advancing as the increase in single core performance is levelling off, and overall system performance can only be raised by using more computational cores. As large-scale SMP systems are too expensive to build, most large-scale systems use a massive amount of medium sized SMP nodes to create a larger distributed memory machine.

While massively parallel processor systems (MPPs) have been very popular in the 90's with computers like the Cray T3E, they were replaced by clusters of SMP systems, where hybrid programming models could be applied. These have now evolved to what could be called massively parallel SMP systems, with a large number of SMP nodes. The IBM Blue Gene/P solution supercomputer JUGENE for example, inaugurated at the Jülich Supercomputing Centre in the beginning of 2008, has a total of 16,384 4-way SMP nodes. Currently those architectures can only be utilized efficiently with message passing libraries, where implementations of the Message Passing Interface (MPI) [8, 9] are clearly the most popular among scientific applications. Using threading libraries within a node is attractive with multi-core chips of four and more cores per node. In computational sciences, OpenMP can be seen as one of the most widely used threading interfaces, followed by POSIX threads and others.

Scalability is a key issue in parallel computing and will become more and more important. Applications running on high-end systems in the so-called High Performance Computing (HPC) area will be target of immense optimization towards the emerging architectures, to be able to provide the possibility of simulation at larger scales than today.

In the following sections MPI will be introduced with emphasis on the point-to-point and collective communication routines, as this is the main MPI communication mode used on large-

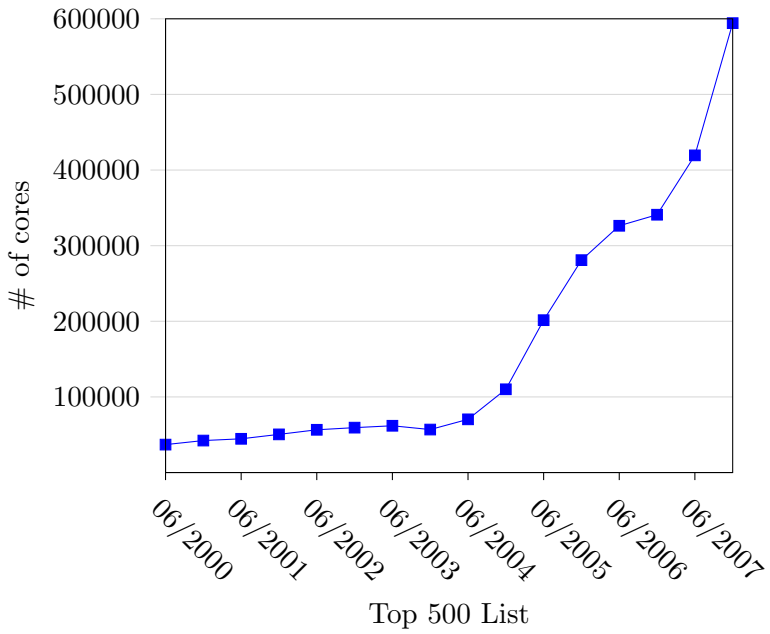


Figure 2.1: Aggregated number of cores on top 20 computer systems (2000-2007).

scale architectures today. Performance analysis, with an emphasis on event-based performance analysis will also be introduced. Concluding this chapter, several other topics related to this thesis will be covered to complete the background knowledge needed for remainder of this thesis.

2.2 The Message Passing Interface (MPI)

As mentioned above, message passing is an explicit process-communication paradigm widely used in scientific computing when running on parallel systems with multiple private address spaces. As message passing itself is a paradigm and not so much a library of any sort, it has been adapted by different vendors of parallel machines in similar ways that are mostly not portable across platforms. This means, developing message passing applications was a highly platform-dependent task, and moving to a new parallel platform resulted in a significant porting effort.

With the goal of providing a widely used standard for writing message-passing programs, the Message Passing Interface Forum, comprised of members from academia as well as industry, created the first version of the MPI standard document in 1994 [8]. It provides a practical, portable, efficient, and flexible standard for message-passing libraries, which has become the most used message passing concept for scientific applications. MPI itself is not a library, but a standardized API defining how a message passing library that implements the standard must be built and behave. This enables vendors of different parallel machines to optimize an implementation for a specific platform, while the API for the application is not changing between different parallel machines, decreasing porting efforts significantly.

In 1997 the first document was revised as MPI 1.1 and extensions were introduced in the MPI 2.0 document [9]. The MPI-2 standard document brought clarifications and new functionality to the existing MPI-1 document, not replacing but extending it with new features. The MPI standard documents cover point-to-point, collective and one-sided communication[1]. However,

[1] introduced with MPI-2

it not only defines communication schemes and their semantics, but also the abstractions needed to provide a portable and efficient communication interface, such as process groups, communication contexts, process topologies, a profiling interface, and environmental management and inquiry functions. Additionally, the bindings for Fortran77, Fortran90 as well as C and C++ are defined.

One of the central data structures, deeply woven into the communication infrastructure of MPI, is the communicator, containing a process group and its context. The context is a property of a communicator that permits further partitioning of the communication space. A message sent in one context cannot be received in another. Contexts are not explicit MPI objects, as they appear only as a part of communicator. This concept allows the existence of two or more different communicators, involving the same group of processes. All communication is triggered within a specific communicator. The global communicator `MPI_COMM_WORLD` is defined by the MPI environment, and exists without any intervention by the user after the call to `MPI_Init` returns. Furthermore, subsets of this communicator can be created by the executing program.

The communicator's process group contains a list of all processes, where each process is assigned a unique identifier called its *rank*. Ranks range from 0 to $n-1$, if n processes are contained in the communicator. A process can then be uniquely identified in the system with a given communicator and rank, and any mapping from local rank to global process can be handled in the background by the communication library.

2.2.1 Point-to-point communication

One of the communication paradigms defined with MPI-1 is point-to-point communication between a single sender and receiver. MPI defines two classes of point-to-point communication modes: blocking and non-blocking[2]. Which refer to the behavior in regard to completion of the function call employed. Blocking communication functions will return when the transfer has at least reached the state that the transfer buffer can be reused. Non-blocking calls return immediately to the user, leaving the transfer buffer in a state in which it must not be touched by the user, until a call to another function completes the transfer and allows the buffer to be used again. While blocking calls keep the user from corrupting transfer buffers while transfers are ongoing, it is not possible to overlap computation and communication. If the communication hardware supports DMA for message transfers, non-blocking calls can perform significantly better.

To track ongoing messages started with non-blocking communication calls, MPI defines communication requests. These requests uniquely identify an ongoing message transfer initiated by a non-blocking communication call. The simulation infrastructure presented in this thesis does not yet support non-blocking communication. The characteristics of non-blocking communication will therefore not be discussed in the following. Both, blocking and non-blocking point-to-point communication calls are provided with four different communication protocols:

`MPI_Ssend` is the synchronous send. It uses a rendezvous protocol with the matching receive operation to handle message exchange. This means that any transfer will only start after the corresponding receive has been started on the destination process. As indicated by the name, the two processes involved in the message transfer will synchronize.

`MPI_Bsend` is the buffered send. Here, the entire message is copied to a user space buffer to be sent in the background, while the application can continue to work on the original buffer. Whether the message is sent synchronously or asynchronously is no longer the issue of the user and dealt with in the background. To enable explicitly buffered

[2] also called *immediate*

message transfers, the user has to provide a buffer with sufficient space via a call to `MPI_Buffer_attach`.

`MPI_Send` is the so-called standard send. Here, the MPI implementation can decide dynamically to either use a synchronous send or a buffered send. As the buffered send would be implicit, it cannot rely on the user providing buffer space. Therefore, the MPI implementation has an internal buffer it can use in this case. As the MPI implementation is competing for memory with the user application, it can typically only reserve a relatively small memory block for internal buffering. The size of this internal buffer influences the threshold that decides whether a message is sent in synchronous or buffered mode.

`MPI_Rsend` is the ready send. This send call relies on the matching receive to be already posted by the destination process. The MPI implementation does not perform any checks whether the receive is actually posted when the send is starting the transfer. As it is sometimes hard to guarantee this in an algorithm, it is not widely used.

In general, a complete message transfer is identified by the sender, the receiver, the communicator this message transfer is using, and a tag. Yet this identification is not unique. Consecutive messages can be sent between two processes using the same set of parameters. In this case, the MPI implementation guarantees that messages with the same communicator and tag between any pair of processes will be received in sending order. The tag can be an arbitrary positive number, within an MPI implementation-specific range of values. On messages with different tags, a receiver can choose to influence the reception order by specifying the tag of the message to be received. Additionally, the MPI standard also defines the wildcards `MPI_ANY_SOURCE` to match messages from any sender and `MPI_ANY_TAG` to match message with any tag.

2.2.2 Collective communication

The second communication paradigm of MPI is collective communication. All processes within a specific communicator collectively perform the communication. Unlike point-to-point communication where a communication can be tagged with a specific number, collective communications cannot be tagged. This means two consecutive calls of the same type on the same communicator cannot be differentiated any further by the process. As all processes of the communicator used in the collective communication have to participate, it is implied that for a specific communicator that each process is calling the same collective communication routines in the same order.

Collective communication calls may be synchronizing, which implies no process leaves the function call until the last process has joined. Only one communication call, the `MPI_Barrier`, is explicitly synchronizing, as this is the purpose of this call. Some communication calls, like `MPI_Allreduce`, are implicitly synchronizing, as they involve message transfers from each process to all others. As it is not possible for a process to receive a message before the sending process started a send, each process has to remain in the communication call until the last process has joined. When dealing with synchronizing collective communication, it is in the programmers' responsibility to ensure that consecutive collective communication calls on different communicators will be called in the same order on each of the participating processes. Otherwise, a deadlock will occur, since both ongoing synchronizing communications will not be able to complete, as each one is waiting for processes engaged in the other call. The function calls that can currently be simulated by the simulator presented in this thesis are described in the following:

`MPI_Bcast` is used to copy (broadcast) data from one process to all other processes in the communicator. This communication call is usually not synchronizing, but the receiving processes will not be able to exit this call until the root process that is sending the data has entered the communication call.

`MPI_Gather` is used to gather data from all processes onto one process. The gathered data is in the receive buffer of the specified root process after call completion. As the root process needs to receive data from every other process involved, it cannot complete the call before every other process has entered the call.

`MPI_Allgather` is a variant of the gather operation. Here, there is no specific root process, as all participating processes will receive the complete gathered buffer.

`MPI_Scatter` is used to send data from one process to all other processes of the communicator. Unlike broadcast, each process receives a different part of the communication buffer. The receiving processes will not be able to complete the call until the specified root process has entered it.

`MPI_Reduce` is used to perform a specified reduction operation on data sent by the processes of the communicator. The reduce operation can be either one of the predefined operations, e.g. `MPI_SUM`, `MPI_MAX`, etc., or a user-defined operator that only has to guarantee associativity. Additionally, it can be specified whether the user defined operation is commutative. This means the order of performed reduction operations does not effect the overall outcome of the function, apart from rounding errors. Rounding errors in floating-point data are mostly inherent to reduction functions when operands differ in several orders of magnitude.

`MPI_Allreduce` is a variant of the reduction function, where the result of the reduction becomes available in the receive buffer of every participating process.

`MPI_Scan` is an ordered partial reduction. The reduction operator is used on the operands in the order of their rank in the communicator that is used, and only up to the rank of the current process. In principle, this means the exit of each process is only dependent on the entering of lower ranks in this communicator, however (naive) implementations might enforce global synchronization (e.g. MPICH).

2.3 Performance optimization

The ways to improve an application's performance are manifold, due to the mapping of specific algorithms on the available hardware. Some developers need to optimize the application performance focusing on the single core performance, with criteria such as cache misses, processor utilization, and similar performance indices, while others need to focus on the parallel nature of a code, and optimize the communication and process-interaction of the code. This chapter will give a short introduction to performance optimization, focusing on techniques of measurement and analysis of the application's behavior that are relevant to this thesis.

2.3.1 The performance optimization cycle

Regardless of optimizing an application for single core performance or optimizing for scalability, the approach is very similar. An application has to have its execution behavior monitored, and this behavior needs to be evaluated to draw conclusions for further improvement. This is an iterative process that can be described by a cycle, the so-called performance optimization cycle.

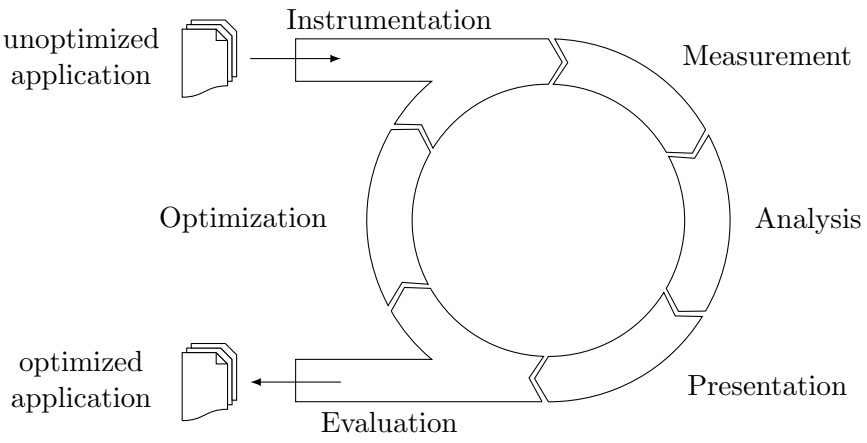


Figure 2.2: The performance optimization cycle.

When broken down into phases, it is comprised of: instrumentation, measurement, analysis, presentation, evaluation and the actual optimization of the code.

As shown in Figure 2.2, the user starts with the original application, which enters the optimization cycle in the instrumentation phase. Instrumentation describes the process of modifying the application code to enable measurement of performance relevant data during the application run. This can be achieved with different mechanisms, such as source code instrumentation, binary instrumentation or linking with pre-instrumented libraries. Instrumentation on the source code level can be done by introducing additional instructions into the source code prior to compilation. When only the compiled binary is available, binary instrumentation can be used to insert the additional instructions into the application. This is done through binary rewrite [19], which inserts the instrumentation directly into the existing binary code. The third method is the use of pre-instrumented libraries, which contain an instrumented version of the relevant library functions. MPI provides a special interface for this kind of instrumentation, the so-called PMPI interface. In most cases, where the compiler supports weak symbols, the MPI library provides the function calls under two symbols different symbols, starting with `MPI_` and `PMPI_`. The first is a weak symbol, which means a tools library can implement a function with the same name, which is called instead of the function provided by the library. To be able to still use the library functionality, the latter symbol can then be used inside the interposing function call. As this interface is defined in the MPI standard, its API is portable and creates an opportunity for tool developers to provide a single portable measurement library for multiple different MPI implementations using so-called wrapper functions. Important MPI functions can be pre-instrumented and linked together with the original MPI library. Listing 2.1 shows an example wrapper from the Epik measurement system used in the Scalasca tool set, which is described in more detail in the following sections.

Finally, instrumentation can also be inserted during runtime execution of the application using dynamic instrumentation [19]. Here, the instrumentation is not inserted statically into the application code, but depending on the history of the running application, instrumentation of certain events can be turned on or off. This can be extremely useful when dealing with long running applications, creating partial event traces. Some dynamic instrumentation tools support the dynamic removal of instrumentation from the code, thus keeping the instrumentation overhead to a minimum. Others query whether an instrumented function needs to be measured or not on each invocation of that function call, which can be costly, as this query adds to the latency of the call. To decide whether a call needs to be monitored or not, blacklist-

```
1  int MPI_Barrier( MPI_Comm comm )
2  {
3    int result;
4
5    if (IS_TRACE_ON)
6      {
7        TRACE_OFF();
8  #ifndef ELG_CSITE_INST
9        elg_enter(elg_mpi_regid[ELG__MPI_BARRIER]);
10 #endif
11       result = PMPI_Barrier(comm);
12
13       elg_mpi_collexit(elg_mpi_regid[ELG__MPI_BARRIER],
14                ELG_NO_ID, ELG_COMM_ID(comm), 0, 0);
15       TRACE_ON();
16     }
17   else
18     {
19       result = PMPI_Barrier(comm);
20     }
21
22   return result;
23 }
```

Listing 2.1: Instrumentation wrapper for `MPI_Barrier` using the MPI profiling interface.

ing or whitelisting can be used. Blacklisting assumes every function needs to be instrumented apart from the function listed in the blacklist. Whitelisting assumes that no function should be instrumented apart from the function listed in the whitelist.

When instrumented code is executed during the measurement phase, performance data is collected. This can be stored as a profile or an event trace, depending on the desired level of information needed. The additional instructions inserted during instrumentation and associated measurement storage require resources: memory as well as CPU time. Therefore the application execution is perturbed to a certain degree. Perturbation by the additional measurement instructions may be small enough to get a fairly accurate view of the application event trace, however, certain application properties like frequently executed regions with extremely small temporal extent, will always lead to a high perturbation. Thus the measurement of those regions must be avoided.

The measurement data can also be analyzed after application execution. The amount of data collected during the application run mainly influences the results of this post-mortem analysis. If a detailed event trace has been collected, more sophisticated dependencies between inter-process events can be investigated, resulting in a more detailed analysis report. Especially inter-process event correlations can usually only be analyzed in such post-mortem analysis. The information needed to analyze these correlations are usually distributed over the processes in question, and transferring the data during normal application runtime would lead to a significant perturbation during measurement, as it would require application resources on the network for this.

After analyzing the collected data, the result needs to be presented in an analysis report. This leads to the next phase in the performance optimization cycle: the presentation phase. At this stage, it is important to reduce the complexity of the performance data to ease evaluation by the user. If the presented data is too abstract, performance critical event patterns might

not be recognized by the user. If it is too detailed from the beginning, the user might drown in too much data. User guidance is therefore the key to productive application optimization.

In the evaluation phase, conclusions are drawn from the presented information, leading to optimization hypotheses. The user proposes optimization strategies for the application, which are then implemented in the optimization phase. Afterwards, the effectiveness of the optimization has to be verified by another pass through the performance optimization cycle. To increase productivity within the optimization process, the user needs assistance in assessing different optimization strategies and their performance impact. When the user is satisfied with the application performance during evaluation and no further optimization is needed, the instrumentation can be disabled, and the performance of the uninstrumented application execution assessed. The result is the optimized application.

2.3.2 Performance indices for parallel applications

Two main indices are currently used for performance classification of applications when dealing with its parallel performance: parallel speedup and parallel efficiency. The speedup expresses the factor of improvement in wall clock time of a parallel execution in comparison to the sequential execution:

$$speedup(n) = \frac{T_{sequential}}{T_{parallel}(n)} \tag{2.1}$$

The parallel efficiency of a code is calculated as the speedup divided by the number of processes or threads. It expresses the factor between efficiency of the reference code and its parallel execution. Efficiency of a sequential program is 1 by definition, as can be derived from Equations 2.1 and 2.2.

$$efficiency(n) = \frac{speedup\,(n)}{n} \tag{2.2}$$

Clearly, when a code has a parallel efficiency of less then 1, more CPU time has to be spent on a single application run, even if the overall wall clock time is less. This has direct impact on two very important factors in scientific computing. First, the intention to parallelize a code is often to reduce the *time to solution*, which expresses the wall clock time needed to receive an answer to the problem posed to the application. With decreasing parallel efficiency, the wall clock time of two runs with different degrees of parallelism converge. This has been expressed in 1967 by Gene Amdahl [1] and is known as *Amdahl's Law*.

$$speedup(n) = \frac{n}{1 + (n-1)\alpha} \tag{2.3}$$

Here, α is the fraction of sequential code in the overall application. Figure 2.3 shows this upper bound for several factors of α against the number of processes ranging from 1 to 65536. It can be derived easily from the plot that with 1 percent of overall execution being serial, the achievable speedup is bound by around 100 for 64k processes. This is less than 0.153 percent parallel efficiency.

However, Amdahl's law only accounts for strong scaling with a fixed problem size. With an increasing number or cores, the time spent in parallel computation is decreasing, whereas the time for serial parts remains constant, eventually dominating the overall time of execution. Gustafson [14, 15] remarks that strong scaling mostly is a subject of academic research, whereas in scientific and industry production simulations weak scaling is used most often. In weak scaling the problem size is scaled with the number of processors, keeping the overall ratio of parallel computation to serial computation during the application runtime constant. When comparing application runs, strong scaling is therefore relevant for solving the same problem faster, and weak scaling is relevant for solving a bigger problem in the same time. This softens the impact

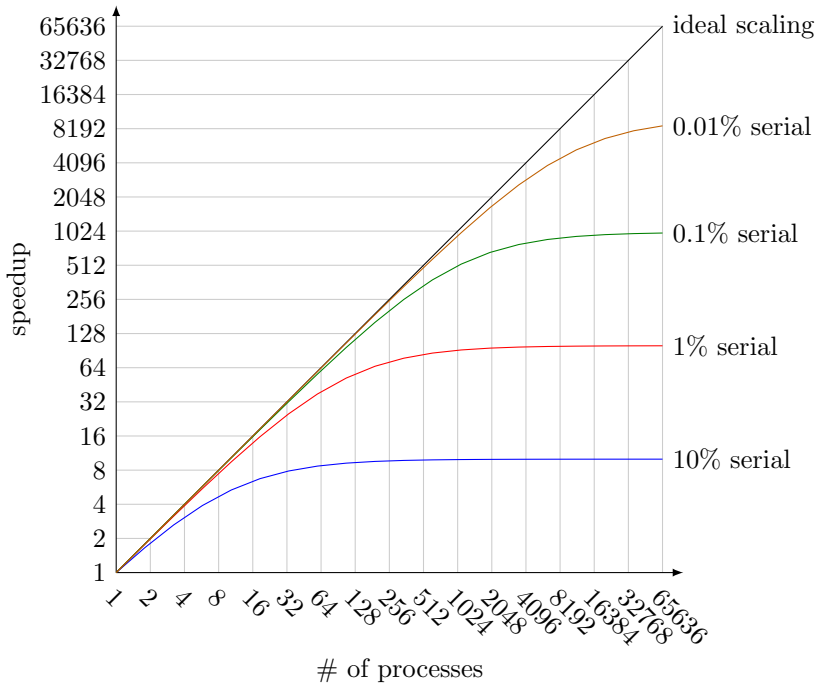


Figure 2.3: Amdahl's Law: Speedup for different factors of α.

of Amdahl's law as a strict upper bound for an applications scalability, and enables current scientific simulations to be conducted efficiently on several thousands of cores.

Both methods of scalability have their place in assessing scaling behavior of an application. However, strong scaling is often considered a better indicator, as it poses harder requirements on the properties of an application. Additionally, there are cases where weak scaling is impossible. This is the case, when an application code is ported to a system with a higher degree of parallelism, without scaling the problem size constraints, too. For example, when comparing the JUMP and JUBL computer systems present at the Jülich Supercomputing Centre today, both systems provide an equal amount of globally available main memory, however the JUBL system comprises more than twelve times the number of cores than JUMP. This means, an application that needs the complete main memory available on the JUMP system, will have to show a good strong scaling behavior to be able to run the same simulation efficiently on the JUBL system.

CPU-time-per-solution is another important factor, when dealing with the performance of parallel applications. It expresses how much CPU time has to be invested to receive the solution. CPU time is the currency of projects in scientific computing, where projects are granted a certain amount of CPU time on a system. With decreasing parallel efficiency, more and more CPU time is needed. A code therefore is considered *scalable* if it still has a high parallel efficiency even at large scales.

Communication plays an important role in an application's parallel efficiency. In distributed memory environments data that is needed on different processes is either sent explicitly, in case of message-passing applications, or implicitly, in case of partitioned global address space languages. The time dedicated to communication is time that is not available for computing the problem's solution. It affects application scaling in much the same way as execution of a sequential part in the application. When parallelizing, communication therefore needs to be kept to a minimum and, when the hardware supports it, overlapped with computation, and

with that hidden from the overall execution time.

When dealing with algorithms that work on a single problem running on distributed memory systems, communication is often inevitable. However, the amount of communication should be reduced to a minimum, thus elimination of unnecessary communication time will automatically improve the scalability of the application.

2.3.3 Methods for obtaining performance data

In performance analysis, there a two main strategies for obtaining performance data: direct and statistical measurement. With direct measurement, the application itself is saving performance relevant information to a so-called event trace at specific points during application execution. This is often done via prior instrumentation of the application code. The events describe a certain action at a specific point in time. Even though the application behavior has to be abstracted, it allows for a consistent tracking of performance relevant available to the measurement system. A certain dilation of the overall application behavior is inherent to this measurement method, as the instrumentation becomes part of the applications execution code. An instrumented version of an application will therefore always have more instructions to process than the uninstrumented application. Direct measurement is typically preferred when individual events are critical to understanding execution performance, e.g. communication and synchronization behavior.

With statistical measurement the application is not instrumented, but observed from the outside. A sample is taken at a specific point in time triggered externally. This trigger can also be seen as an event, but is not to be confused with the events of the internal method, where an event is describing the application state. Here, an event is only the trigger for gathering a sample of the application state, giving this method its name: sampling. The idea is that a statistically relevant set of samples allows inferences to be made about entirety. The amount of samples as well as the moments of taking the samples mainly decide on the quality of the conclusions to be drawn. While a trigger is often time-based to obtain an even sample rate on the application, it can as well be any other hardware event, like a cache miss or similar. This way, application phases with higher cache miss rates are sampled more often, and with that in more detail. In contrast to the direct method, sampling does not have a deterministic outcome. This means that between samples, the application state is mostly unknown, and only a high sample rate will lead to a detailed view of the application behavior. However, even in this case, it will only allow for statistical analysis of the samples. Perturbation of the investigated application is mainly influenced by the frequency the samples are taken and whether the measurement is using hardware support. Each sample also introduces dilation, but not on every event and not at all when sampling is disabled, providing control over measurement quality. Statistical sampling is typically employed when the overhead of processing very large numbers of short execution events would result in overwhelming amounts of measurements and/or unacceptable measurement perturbation, e.g. for cache misses.

When performance data is gathered, it has then to be decided how to process the information. Profiling will aggregate specific performance values online. As performance data is aggregated at runtime, profiling usually needs less memory and disk I/O than tracing, which will be explained later. Even with long running applications, profiling will show a fairly constant memory usage. The performance critical data is aggregated during measurement and a so-called application profile is created at program termination.

As profiling aggregates application execution information on-line and perturbation is kept to a minimum, there is no inter-process communication issued by the measurement system during the application run. This limits the possibilities of analysis of inter-process performance properties.

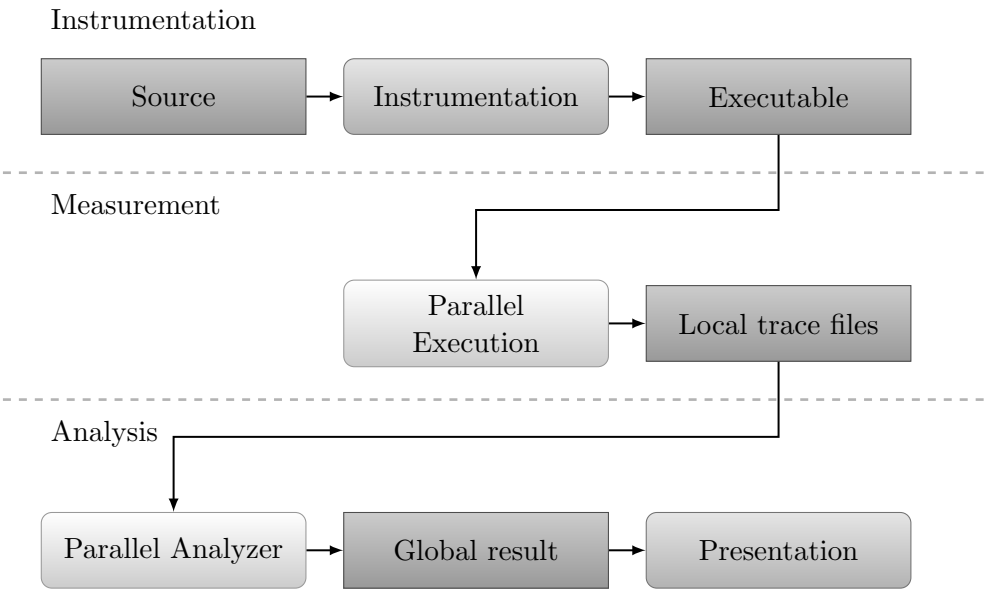


Figure 2.4: The Scalasca analysis workflow.

With tracing, the individual events or samples are preserved and written into a trace. In the following, a trace will refer to event traces gathered by internal measurement. Event traces give detailed insight into the application state at a specific point during program execution. Yet, obtaining an event trace can result in a higher perturbation of the application, as the event trace needs more memory during measurement and might lead to a flush of the event buffers, involving I/O to disk. Traces are analyzed post-mortem, which means after program termination, as the amount of data to be analyzed would lead to significant perturbation, if done online. Offline analysis enables the investigation of more complex performance patterns, including inter-process behavior.

2.4 Event-based performance analysis

2.4.1 Motivation

As shown in the previous sections, scientific simulations need a growing degree of parallelism to cope with the given tasks. With increasing parallelism it becomes more and more complex for the user to get an overview of the global behavior of the application. Tools to analyze parallel application performance give users the opportunity to reason about the behavior of their applications. With a growing number of processes, classic tools for application performance analysis become hard to handle. While measuring and displaying raw data is still feasible on with small numbers of processes or threads, the user will almost certainly get lost in the sheer amount of data presented at large scales. It is therefore evident that the ways information about application performance is presented to the user are subject to a scalable design as well.

To reason about application performance in massively parallel systems, the user will need automatic assistance to filter relevant information from the measured data. Several approaches exist to cope with this problem, each allowing a more or less detailed view on application performance, one of which will be presented below.

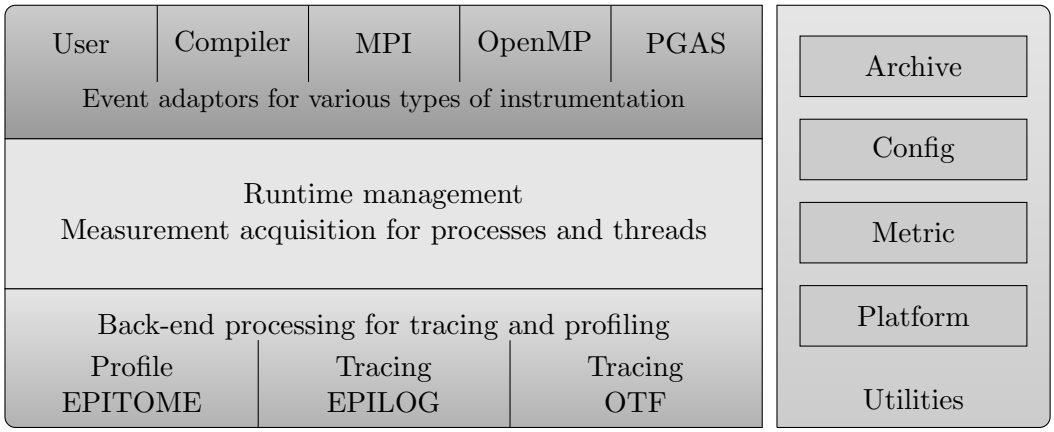


Figure 2.5: The Scalasca measurement system Epik

2.4.2 The Scalasca analysis workflow

The Scalasca tool set, which forms the basis for the performance simulator presented in this thesis, consists of several tools, each covering one of the phases of the performance optimization cycle. The analysis workflow when using Scalasca is displayed in Figure 2.4.

Scalasca uses a combination of the presented instrumentation techniques. While MPI library functions are instrumented by the use of an interposed pre-instrumented library, user code is handled by source code instrumentation. Source instrumentation can be done manually, by inserting instrumentation directives into the code and preprocessing the annotated code by the source-to-source translator Opari [20]. When coverage of a lot of functions is needed, automatic function instrumentation by the compiler can be used, but it is not supported by all compilers. A third method of instrumentation involves automatic instrumentation with blacklisting support via the Tau instrumentor.

The measurement is handled by the Epik measurement infrastructure [31]. Figure 2.5 shows the overall architecture of Epik. It is divided into three layers. The top layer offers several event adaptors for user annotations, computer-generated function instrumentation, MPI library instrumentation, OpenMP instrumentation, and partitioned global address space languages. The middle layer is the runtime management for measurement acquisition for processes and threads, and the bottom layer provides plugins for different forms of output.

The default output module is the profile generator Epitome. The format for the application profile is the Cube file format [7], which was introduced by Wolf [28] and has received several revisions since then: the current version is CUBE3. When using trace file output, the user can choose between two formats: Epilog [30] and OTF [18]. The Epilog format is the standard format for Scalasca traces, which was initially developed by Mohr and Wolf for Scalasca's predecessor Kojak [30]. To improve scalability, Scalasca no longer saves the event trace in a single file, but uses per-process files for the event trace and two additional files for all event definitions in the trace and their mappings. The OTF trace format is a development of TU Dresden, as an open successor for the VTF trace format. It is currently mainly used with VampirServer, the client/server successor of the Vampir performance analysis tool.

During the analysis phase, Epilog event traces are processed by the Scout analyzer [11]. It is a parallel analyzer for the automatic search of inefficiency patterns in event traces. Scout uses the same number of processes as the job did that created the event trace. Usually it is executed directly after the measurement run as part of the same batch script, when an allocation of that specific number of processes already exists. The analyzer uses the Pearl high-level

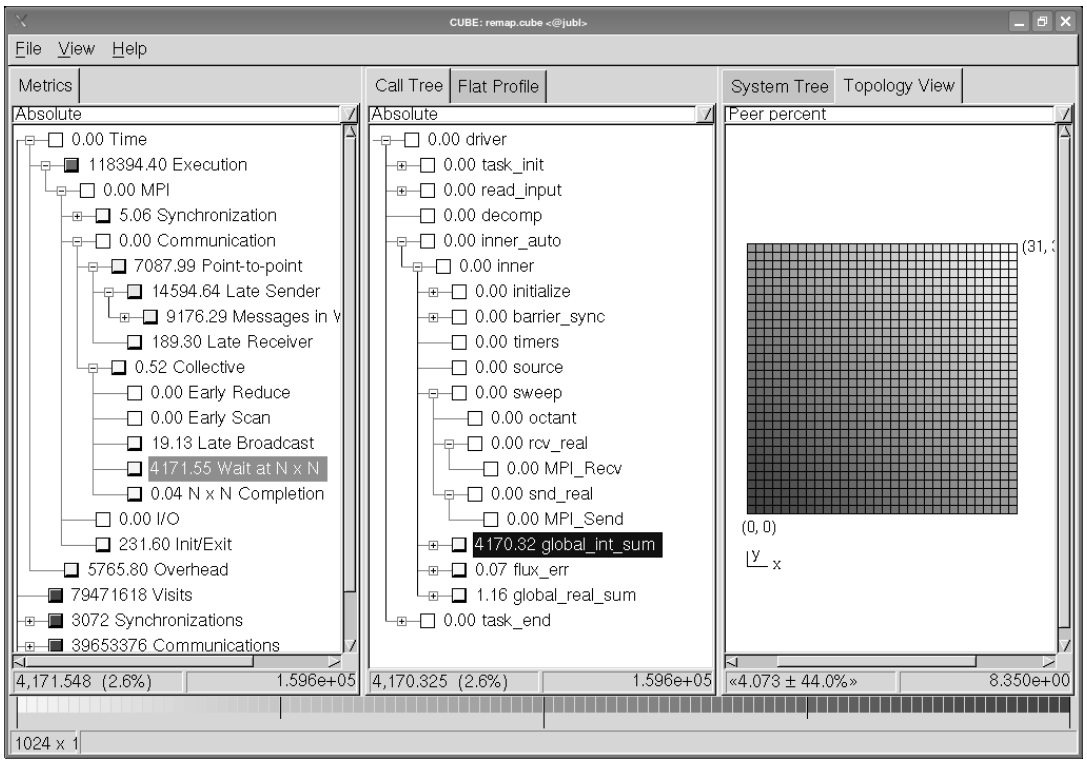


Figure 2.6: The three panes of the CUBE presenter.

trace access library [10] to access the event trace. It provides an interface to a simple event replay, while a callback mechanism supplies the interface to processing the events. Since PEARL forms the basis for the simulator described in this thesis, it will be discussed in more detail in Section 2.4.4.

Like EPITOME for the application profile, SCOUT uses the CUBE file format for its analysis report. The data provided in a CUBE file is a three-dimensional matrix of severities, with the dimensions location, call-path and performance property. While the trace data contains information on individual instances of the performance problems, the CUBE file contains the aggregated severities for each tuple (location, call-tree node, performance property). The file format is self-describing, enabling all names of performance properties to be read in with the data.

The reports can then be displayed with the CUBE presenter [25]. The presenter uses three panes, displaying a tree hierarchy for each dimension (Figure 2.6). The performance property pane is the left-most pane, the call-path is displayed in the middle pane and the location is presented in the right-most pane. The individual panes display inclusion hierarchies using a tree widget. As large process counts are unmanageable to view using a tree hierarchy, the location pane also offers a topology view. If available, the hardware topology is used to visualize the severity on particular locations in a 2D or 3D grid-like fashion. If an application uses an MPI cartesian topology, even this can be used to arrange the individual locations in the pane. Each node in the tree hierarchies of each pane has an associated color coding, visually guiding the user to the highest severity. A pane shows the distribution of the node value selected in its left neighbor pane.

The performance property pane has no left neighbor, thus, the data shown is not relative to a selection. If a node is collapsed, its value as well as the associated color represents an

inclusive value, which is the sum of the values for all child nodes and its own value. If a node is expanded, its value represents an exclusive value, which is only the value associated to that specific node with the values for each child node distributed across each of them.

2.4.3 Event model

In event-based performance analysis, all events in the event trace are defined in an event model. This model specifies all event types as well as their constraints in the traces. This section will introduce the parts of the event model used in Epilog event traces that are relevant to the understanding of this thesis.

Definition 2.1 (event model)
An *event model* defines the nature of events in the system, as well as the semantics of their interaction. ◇

Definition 2.2 (location)
A *location* models a process in multi-process applications and a thread in multi-threaded and hybrid applications. ◇

Definition 2.3 (event)
An *event* describes an atomic action occurring at a distinct location at a distinct point in time. Each event has a set of attributes. The attributes type and time as well as the location are part of every event. Depending on the type, the event can possess additional attributes. Event attributes are annotated with a dot and the name of the attribute, e.g. *e.time*. ◇

Definition 2.4 (event trace)
An *event trace* is an ordered set of events on one or more locations. The events are ordered with ascending time stamps.

$$\forall i, j : i < j \Rightarrow e_i.time \leq e_j.time \tag{2.4}$$

◇

Equation 2.4 ensures first that events are in chronological order and, second that their timestamps are monotonically increasing. Although no two events on a single location physically can occur at the same time, condition 2.4 enforces only monotonically increasing timestamps. Event timestamps written directly by the measurement system are always strictly monotonic, yet in cases where the timer resolution is not sufficient or events cannot be created by the measurement system with the correct time, they have to be shifted to the approximated time. This is the case for modelling RMA transfers, where the communication events need to be shifted to the end of the corresponding synchronization epoch [16].

For a given location the thread of control is modelled by multiple nested region instances. The dynamic extent of each region instance is defined by the time between its ENTER event and its EXIT event. Collective function calls, as defined by MPI, are modelled with a special MPI_COLLEXIT event, which is saved instead of a normal EXIT event for the corresponding region. The MPI_COLLEXIT event possesses additional attributes, which include bytes sent and received during the call, as well as the rank of the associated root process, if applicable.

Point-to-point communication functions are special regions containing additional events concerning communication. This can be an MPI_SEND event or an MPI_RECV event, or in case of `MPI_Sendrecv` and `MPI_Sendrecv_replace` function, even both. The MPI_SEND event is used to model the earliest time the message transfer could have been started, whereas the MPI_RECV event is used to model the latest time a receive operation could have been completed. The

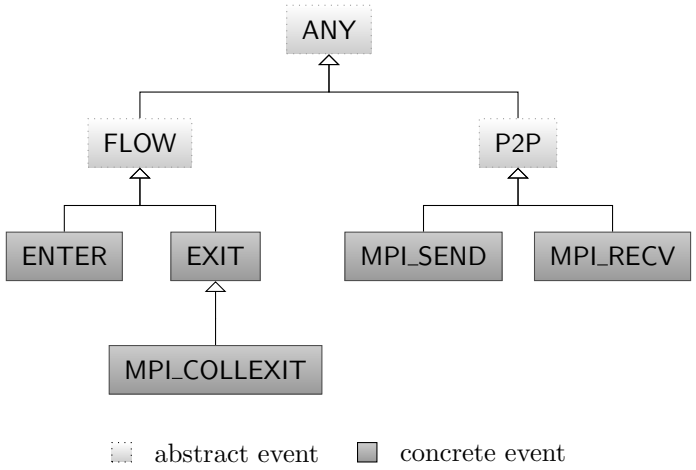


Figure 2.7: PEARL concrete and abstract event types.

information stored with the communication events is used to identify communication partners, as well as the amount of transferred bytes. To define inefficiency patterns for the automatic search, the constraints of the MPI communication are then modelled using these events.

2.4.4 The PEARL library

The high-level trace-access library EARL, which is used in KOJAK, only allows serial access to an event trace. For KOJAK, the event traces measured at each location need to be merged to a single file. For large-scale applications serial access is no longer feasible, as the amount of data that has to be processed is too large, and the time needed to analyze such large traces by a single process is not acceptable. Therefore EARL's functionality as a high-level interface to the EPILOG trace data has been reimplemented for the development of SCOUT, the parallel analyzer of SCALASCA.

PEARL is designed to provide a flexible, transparent and convenient access to the event trace data. It offers a mechanism for iterative processing of the complete event trace, called *event replay*. This replay exists in two flavors: forward replay and backward replay. Whereas the forward replay starts at the first event on each location of the local traces and allows a chronological processing, the backward replay starts at the last event and iterates in reverse chronological order. Heart of the replay is a flexible callback mechanism, where the user can define callback functions to be triggered on special events. Those callbacks can be registered dynamically with a callback manager, which triggers the callback upon the occurrence of the desired event. All concrete events can be used as triggers for the callback. Additionally there are abstract events defined for certain groups of events: ANY, FLOW and P2P. The complete hierarchy is shown in Figure 2.7, where the concrete events are depicted as orange boxes with solid border, and abstract events are shown as blue boxes with dotted border.

When the concrete events are not flexible enough to perform a given task, the user can define additional user events, which can be manually triggered during callback processing. This enables the user to create chains of callback functions with arbitrary complexity to perform a given task.

Another feature of PEARL's replay mechanism is the callback data structure that can be passed to each callback. This data structure can be customized by the user through inheritance and used to pass data between individual nodes of the callback network, without having to rely on global variables at any point. This callback data structure defines at least two methods:

`preprocessing` and `postprocessing`. Those methods are called by the replay before and after the event is processed by the callback network. The initial callbacks, triggered by the replay mechanism for each event, are the callbacks associated with the concrete event type of the current event. Within these callbacks, additional notification of other events can occur.

How this event replay mechanism can be used efficiently to analyze large traces in parallel has been demonstrated by Geimer et al. with the SCOUT parallel trace analyzer [11]. This analyzer uses the target applications' general communication scheme to analyze the communication during replay. This means a point-to-point communication is analyzed by transferring the needed data to the participating process via point-to-point communication. Collective communication is analyzed using collective communication calls to transfer the data to the participating processes. Currently, the entire trace information is loaded into memory in one piece and then replayed, allowing a very time-efficient analysis phase.

Finally, PEARL allows the creation of event traces through its writer library. However, the PEARL library does not process the complete set of events, defined by the EPILOG trace-data format [30] yet. The analysis step currently only supports MPI-1 functionality, thus, events of MPI one-sided communication and OpenMP are ignored while creating the in-memory representation of the local trace and are therefore not written out by the writer.

2.5 Related work

This section briefly introduces previous research on simulation-based performance prediction. While the list may not be exhaustive, the research projects mentioned present some aspects similar to the performance simulation presented in this thesis.

2.5.1 Perturbation compensation with TAU and KOJAK

Any performance measurement will incur an overhead during program execution that will perturb the original application behavior. When the measurement is done by software, the perturbation can only be reduced to a certain amount, as the CPU needs to process the sampling interrupts and measurement instructions where it originally executed user code.

If the level of intrusion can be estimated, Wolf [29] and Shende [24] showed that it is feasible to create an intrusion model and remove the overhead. This intrusion model was demonstrated on a Monte-Carlo simulation based on a Master/Worker scheme. With this, perturbation propagation can be shown to create significant impact on the measured application behavior.

This overhead compensation has been introduced in KOJAK, which is the predecessor of the SCALASCA tool set. Its functionality has not yet been ported to SCALASCA, but the performance prediction presented in this thesis is capable of incorporating the correction model easily. The compensation of overhead can therefore be seen as a possible optimization scenario for the performance simulator presented in this thesis.

2.5.2 DIMEMAS

DIMEMAS [12, 13, 2, 3, 4, 22] is a trace-driven simulator for predicting application behavior of message passing programs. Its development started in 1993 at the Polytechnic University of Catalonia in Barcelona, Spain, and was marketed by Pallas GmbH since 1996. It shows similarities with other trace driven tools, such as the AIMS tool set and even the simulator presented in this thesis. Its initial motivation was to study and predict the behavior of time-sharing message-passing programs, while they are executed concurrently to other applications. Primary objective was the study of different scheduling policies in the presence of a multiprogrammed parallel workload [12]. A question in this context was: "How will the application

behave, if it would not be perturbed by other applications?"

One goal of DIMEMAS was to be able to run on a single workstation to avoid a parallel computer system for generating the simulated traces. The trace generation was initially done via VAMPIRtrace and uses the Paraver-Tracer now. Both trace generation libraries are very similar to the instrumentation and measurement structures provided by KOJAK and SCALASCA. With its focus mainly on the CPU-related issues of computing, time for message passing is explicitly excluded from time measurement. The performance prediction is then based on the timings between two communications to project the behavior on a dedicated machine.

With the prediction of application performance without the influence of other applications on a time-sharing platform, it is also suited to evaluate performance prediction based on trace overhead compensation.

2.5.3 AIMS tool set

The AIMS tool set [33] is a tool set for tuning and predicting the performance of message passing applications in tightly- and loosely-coupled systems. It was developed by NASA Ames Research Center, under the High Performance Computing and Communications Program in the mid 90's. As a performance analysis tool set, it can be seen as predecessor to current performance analysis tool sets, such as TAU, KOJAK or SCALASCA.

One part of the AIMS tool set is concerned with the modelling and prediction of application behavior. Specifically it predicts the scaling behavior of an application when varying the two parameters N and P, where N represents the problem size and P represents the number of processors. Output of the performance prediction model can be directly redirected to symbolic engines like GNUPLOT or MATHEMATICA to plot 2-dimensional and 3-dimensional graphs of expected speed-ups and execution times.

The performance prediction is based on a previously recorded execution trace, which is then extrapolated to higher processor numbers and problem sizes. This extrapolation can yield a good view of the scaling behavior of the current application code. The simulation targets questions like "What if the communication link was twice as fast?" or "What if the performance on a CPU on each node was doubled?". These questions focus more on the application behavior in a modified environment rather then on a modified application in the same environment.

2.5.4 BIGSIM

BIGSIM [23, 35, 34] is a emulation environment for large-scale applications on petaflop architectures. It is based on the CHARM++ environment, which is an object-based portable parallel programming language. It consists of parallel objects that communicate via asynchronous method invocation. It also supports automatic load balancing and migratable objects. A special feature of CHARM++ is a communication system that supports processor virtualization. This enables the use of several MPI processes per physical processor.

Using this process virtualization, BIGSIM is designed to run simulations for several thousands of processes on systems with a significantly lower processor count. The emulation is using a similar amount of main memory as the application would, thus it is only feasible when the emulated architecture is providing much less memory to a process than the host supercomputer systems does. With the Blue Gene architecture this certainly holds true, as in the first generation – the Blue Gene/L – each core had 256 megabytes of main memory at its disposal. Its successor – the Blue Gene/P – doubles that to 512 megabytes per core. Current supercomputing cluster systems with about one thousand cores often have two to four gigabytes of main memory per core.

The primary goal for BigSim is to aid application developers to prepare their applications for this new generation of massively parallel systems, while those are still out of reach for most researchers in computational science.

3 Simulating application behavior

3.1 Introduction

While the automatic analysis within the Scalasca tool set already presents a great profit for productivity when optimizing applications, the evaluation phase and the optimization phase of the performance analysis cycle are mostly based on the expertise of the user in evaluating the presented situation. Even with year long experience, estimations on performance increase are hard to give in parallel computing, as the complexity can be overwhelming, due to the massive parallelism of the systems. Whether the proposed optimization will have the desired impact can often only be estimated roughly, and actual changes to the code can be time consuming. Regarding productivity, the user is interested in getting a positive cost-benefit ratio when changing the application. By simulating the impact of potential alterations, the user will be able to estimate whether the proposed modification yields enough increase in overall performance.

One factor that induces complexity into the estimation of the impact of changes to the application code are correlations between different performance properties in the application behavior. This means the existence of a performance problem may influence other performance problems present in the application behavior. It might even lead to the point that one performance problem is completely canceled out by another one, and execution time savings realized by eliminating the first problem will be annihilated again by the second one, rendering all changes to the code useless in terms of optimizing for minimal computing time when the second performance problem cannot be eliminated as well.

This chapter introduces the foundation for replay-based modification of existing event traces using an execution simulator to predict application performance behavior after the optimization of a specific aspect of the code. As the simulator uses trace-based information to model application performance, a certain degree of abstraction of the application code in investigation is inherent to this approach. Traces are usually not based on instruction level granularity, thus optimization hypotheses cannot be applied on instruction level either. Questions like "What if I can reduce the cache-miss rate of function `foo`?" will have to be redefined to "What if function `foo` performs twice as fast?". The possible optimization strategy of increasing the performance through a more sophisticated memory access pattern is left implicit, as it is not relevant to the simulation process. Yet, it will become important again when the user has to evaluate the cost-benefit ratio based on the simulation result.

When using the simulator to investigate hypothetical changes to the application code, the turnaround time for one pass through the performance optimization cycle can be reduced, as optimizations do not need to be implemented in the application code directly. The increase or decrease in performance due to a proposed optimization in the code can be obtained even before one line of code has been changed. The performance impact as well as the cost-benefit ratio can be estimated more precisely.

3.2 Terms and definitions

This section will introduce the terms and definitions used to describe the capabilities of the performance simulation proposed in this thesis. In the following, the terms event trace and trace are used interchangeably. They are used for streams of events describing application runtime behavior. $\mathcal{T}$ is used for the original trace, and $\mathcal{T}'$ is used for modified traces. For

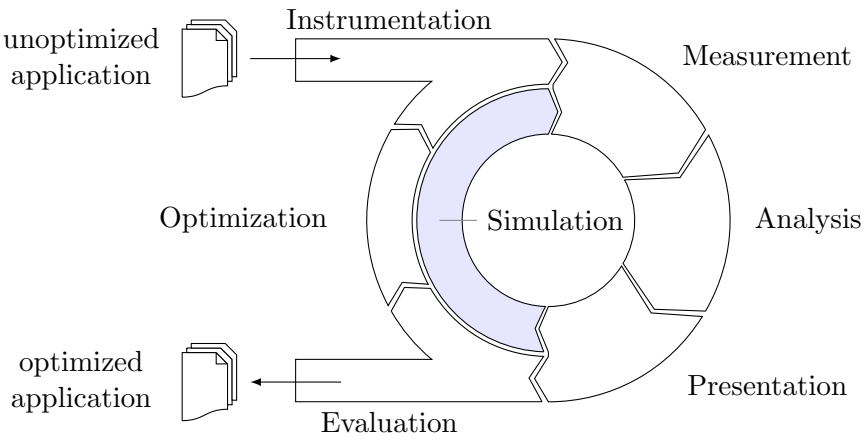


Figure 3.1: Extended performance optimization cycle

further differentiation, indices may appear on the bottom right part, as in $\mathcal{T}'_{id}$, which is used for the resulting trace of the identity simulation. The basis for the performance simulator is a simulation model.

A *simulation model* $\mathcal{M}$ describes the system behavior during simulation. It specifies how event manipulation is carried out during the simulation run. In other words, a simulation model is a set of rules that define how specific events have to be shifted by updating their timestamps. While the simulation model is used as a complete atomic set, the simulation hypothesis is assembled at runtime by several *partial hypotheses*.

A partial hypothesis is the building block for the simulation hypothesis, which describes the complete modification of the event trace. It may describe the modification of the temporal extent of a region instance, the balancing of region instances across several processes, or the elimination of specific events. The *optimization hypothesis* $\mathcal{H}$ is a set of partial hypotheses, which model the proposed modification. The hypothesis, which contains no partial hypothesis, is called the *empty hypothesis* $\mathcal{H}_0$. The simulation hypothesis is a way to encode abstract changes to the code to be considered in a simulation run.

Definition 3.1 (simulation)
Let $\mathcal{T}$ be the original event trace, and $\mathcal{T}'$ the simulated event trace. Then

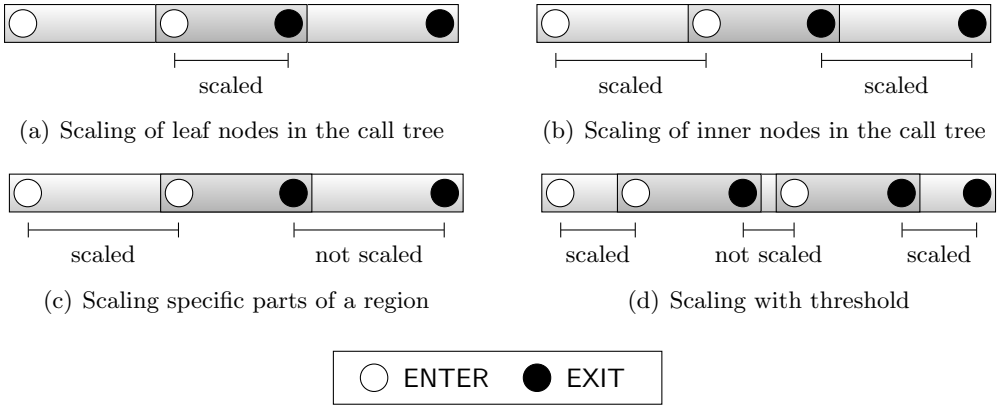
$$sim : \mathcal{T} \times \mathcal{M} \times \mathcal{H} \to \mathcal{T}' \tag{3.1}$$

is called the *simulation* of $\mathcal{T}'$. If $\mathcal{H} = \mathcal{H}_0$, *sim* is called the *identity simulation*. ◇

3.3 Simulation and performance prediction

Performance prediction tools assist the user in estimating application behavior under the assumption of changing parameters. A parameter can be either a change in problem size, number of processors executing the application, a changing execution environment, or even a changing application. The goal of this thesis is to provide a system to predict application behavior after a code modification, based on an event trace of the unmodified application.

The simulator is used during the evaluation phase of the performance optimization cycle shown in Figure 2.2 on page 8. Instead of moving to the optimization phase, the simulation phase will shortcut the cycle directly to a new pass providing the input to a new analysis phase, as shown in Figure 3.1.



(a) Scaling of leaf nodes in the call tree

(b) Scaling of inner nodes in the call tree

(c) Scaling specific parts of a region

(d) Scaling with threshold

Figure 3.2: Different scaling strategies

The evaluation phase yields the proposition of optimization hypotheses. A trace resembling these optimizations is produced in the simulation phase. This trace can then be processed further in the original cycle, which leads to a new evaluation phase. This *inner cycle* is traversed until the user is satisfied with the estimated performance increase and the concrete code changes can be proposed. The changes to the application code will then have to pass once more through the cycle to prove their effectiveness.

3.4 Optimization Hypotheses

In the context of this thesis, the user has several possibilities to influence how the existing event trace can be manipulated. The manipulations are only possible on an abstract level. "Switch the order of two instructions" is not within the scope of a tool working on an event trace, as the event trace itself is only an abstract view of the application behavior. Still, manipulations on this higher level of abstraction provided in an event trace can be very powerful. The following sections will introduce optimization hypotheses, which can be used in the proposed replay-based simulation of application performance presented in this thesis.

3.4.1 Scaling of regions

Performance of a specific user function can be measured by the time it needs to complete. Thus, changes in the performance can be modelled by scaling the runtime of this function. Region instances are described in the event model by the time between entering and leaving the function. When reasoning about elapsed time in a function, two different times have to be taken into account: the inclusive time and the exclusive time. The inclusive time describes the overall time a function call needs to complete. When the function itself is calling other functions, the time spent in the callees is also attributed to the caller. It is therefore desirable to obtain the exclusive time, which defines the overall time needed for completion, excluding the time spent in child nodes in the call tree.

When scaling a function `foo`, i.e., modifying the performance of `foo`, the scaling therefore only applies to the parts where code of `foo` is executed. Thus, scaling applies only to the exclusive time.

If a function call is a leaf node in the call tree, as shown in Figure 3.2(a), its inclusive and exclusive time are equal by definition. As a result, functions of this kind can be scaled directly by adjusting the timestamp of the EXIT event and all following events. When a function call

is an inner node in the call tree, it is interrupted by a function call at least once, as depicted in Figure 3.2(b). One way to scale these kind of functions is to apply the scaling factor to each part where the function is active. As shown in Figure 3.2(c), it might also be desirable to scale only certain parts of this function. This can model improvements made only before or after an intermediate function call. The function call interrupting the execution of the caller increases the level of granularity in this case. Finally, between an EXIT and an immediately following ENTER event that model two consecutive function calls, the EXIT event of the first call and the ENTER event of the second call have a time distance of a minimal latency. When scaling a function, this latency must not be scaled, as this can usually not be influenced by the user. Thus, scaling should only be applied if the time difference is above a certain threshold, as shown in Figure 3.2(d).

3.4.2 Balancing of regions

Load imbalance is one of the most severe performance problems when dealing with massively parallel systems. Minimal waiting times on individual processes can easily add up to a wait states in communication or synchronization region instances. Therefore, balancing parallel parts of the application is very much desirable.

In the current analysis performed by the Scalasca tool set, imbalances in the application run will show up as significant severities of inefficiency patterns on process-synchronizing events. However, these synchronization points are usually not the original cause of the waiting time. Instead, the waiting time is calculated with respect to the semantics of the synchronization point, such as a message, which cannot be received before it is sent. If a receiving process is entering the receive call before the sender is entering the corresponding send call, the time between the two corresponding ENTER events is the waiting time.

Definition 3.2 (load imbalance)
Load imbalance is the difference in work on two or more processes between two of their synchronization points. A common reference time span, like the overall average, is chosen for all processes and the absolute value of the individual time difference is added to the complete imbalance of this process group. ◇

This definition is grasping the core of the matter, yet, it is too general to be handled easily in performance analysis. A load imbalance may be comprised of several region instances. Each of these region instances can have different execution times on the corresponding locations. Additionally, the imbalanced timespan on the locations is not necessarily composed of the same regions. This is quite unwieldy when identifying the cause of the imbalance. Which of the region instances in the timespan would be the cause of the load imbalance of the two processes? Which region should be modified? To reduce the complexity of these questions, load imbalance needs to be evaluated on a per-region-instance basis.

Definition 3.3 (regional load imbalance)
Regional load imbalance is the difference in execution time of the correlating region instances on two or more processes. ◇

As with the general load imbalance, the regional load imbalance in regard to a common reference point is aggregated when more than two processes are involved. It assumes that the investigated scientific simulation uses a domain decomposition algorithm to distribute work, where all processes having a similar call tree work on different parts of the data.

Whereas the effects of load imbalances turn up as waiting time in the application event trace and can identified by the automatic analysis, the cause of the imbalance is currently still to be manually identified by the user. To support the user in the deeper investigation by narrowing

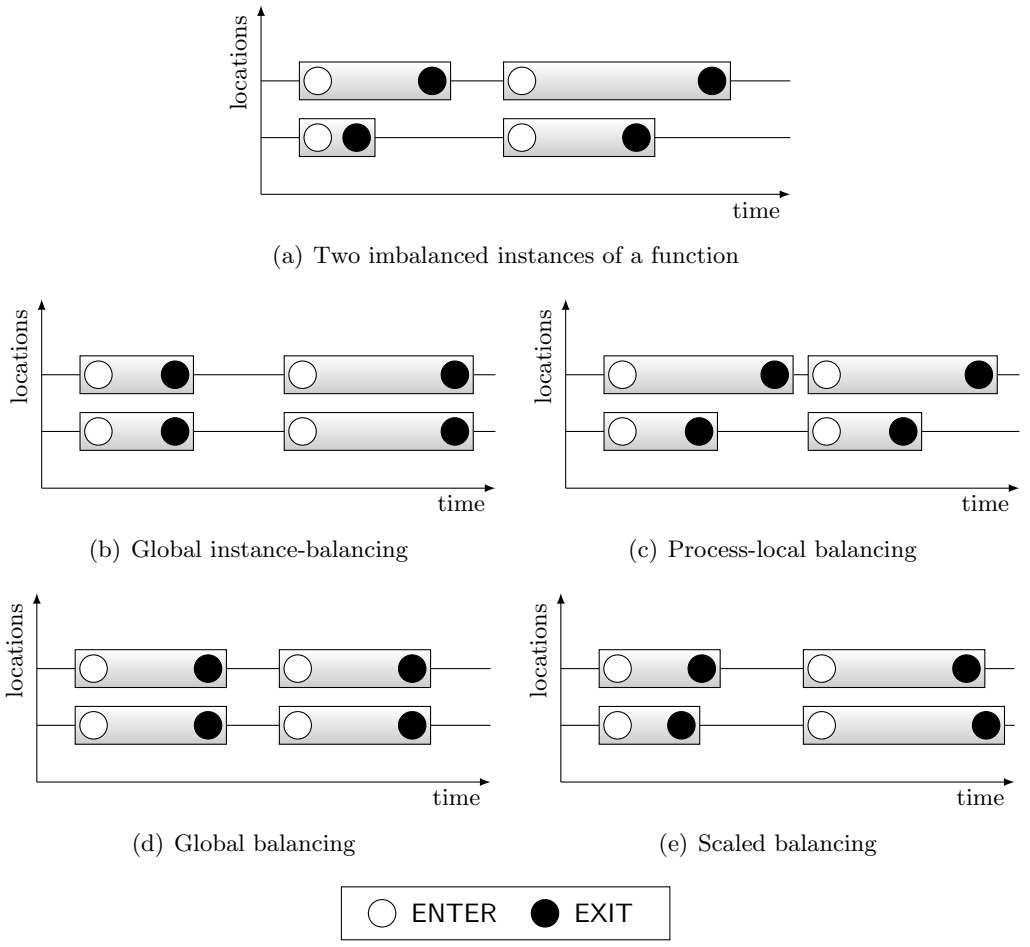


Figure 3.3: Different balancing strategies

down on specific locations, load imbalances can be characterized by classifying the affected process in above and below the reference time span. By definition, an imbalance is caused by the minority of the processes, which means the class with the lower cardinality is defined as the causing set of processes. If both classes have the same cardinality, as with a single point-to-point transfer, it is up to the user to define one of the classes as the cause. In two inefficiency patterns of Scalasca defined for point-to-point communication, namely the Late Sender and Late Receiver, the cause of the waiting time is defined to be on the process that the other is waiting for.

When a specific instance of a regional load imbalance is identified as the cause of the waiting time, the elimination of it in the application code may still be very complex. For instance, it might require reimplementation of a new domain decomposition strategy, which can be very intricate to realize. A simulator, however, can balance function calls easily, as it involves only the manipulation of timestamps, due to the simulation being based on the event trace and not the initial domain decomposition. The prediction of application behavior after the modification can then be used to verify the cause of the waiting time, and give an estimation of the possible performance impact as well.

When balancing a function call, different strategies can be used, as depicted in Figure 3.3. The original load imbalance is shown in Figure 3.3(a). The optimal solution to fix this imbalance is to balance each instance of the function call, as shown in Figure 3.3(b) as global instance-

balancing. Unfortunately, this type of balancing imposes very strict rules on the properties of the function to be balanced. The simulator has no information on which region instances on the different processes correspond to the parallel instance of that call. Thus, it will have to assume that the nth instance of `foo` on process x corresponds to the nth instance of `foo` on process y. Therefore, this type of balancing only works on functions that have exactly the same *shape*. The shape in this context defines the position in the call tree and the functions that are called within the function itself. This restriction can often be met in applications that work with parallel algorithms that have a similar structure on each process, in particular SPMD (Single Program Multiple Data) applications, where the same executable is started in parallel to work on independent parts of the same problem.

However, these restrictions cannot be fulfilled in every application, thus, a means to approximate the *correct* balancing has to be found. Figure 3.3(c) shows an example of balancing a function, not across processes but on the same processor. This is called process-local balancing. While this balancing strategy is very lean in the restrictions it imposes on the shape of `foo`, balancing within one process to overcome an imbalance across processes, might not lead to the desired result.

The third balancing strategy, shown in Figure 3.3(d), is called global balancing. Here, the function `foo` is balanced globally over all instances on all locations. This means all instances on all locations where the function is called will have the same length. This is suited for function calls that should theoretically have equal length on each invocation, but are not balanced due to some runtime parameter. An example would be globally distributed parallel computation phases in iterations. When each iteration corresponds to one function call, balancing across the phases will yield a good balancing for each instance.

If the different phases have significantly different lengths, and this difference is part of the algorithm, global balancing will eliminate this characteristic from the event trace. Scaled balancing is bridging this gap, aggregating the time used for the function per process, and calculating the balancing upon these aggregated values. The absolute increase or decrease in time on the process is distributed with respect to the percentage an instance added to the aggregated time. This way, the process-local imbalance in the call is somewhat conserved, while a global balancing is still taking place. Please note in Figure 3.3(e) that this strategy might lead to inverting a regional load imbalance instance when the process-local variation of region instances' temporal extent differs significantly. Whereas in the original imbalance shown in Figure 3.3(a) the upper location spends more time in the corresponding region instances than the lower location. This changed after the scaled balancing in Figure 3.3(e) on the second region instance, where execution time on the upper location is less than on the lower location.

With all presented types of balancing, the concrete change to an individual region instance on a single location is subject to the same modification strategies used for region-instance scaling.

3.4.3 Elimination of regions

One of the advantages of the proposed form of application performance prediction is its abstraction from the original algorithm during the computational phases. This means that single phases of the application run can be investigated in isolation. For example, the preconditioning of a matrix will not be necessary, since during the performance simulation no actual matrix is being processed.

The process of isolating such phases for further investigation needs the elimination of surrounding events that are not of interest. While it is easy to manage the local deletion of events, it must be guaranteed that corresponding events on remote processes are deleted too, to ensure a consistent state for the global event trace. This means, if an MPI_SEND event is deleted, the corresponding MPI_RECV event needs to be deleted in the event trace of the receiving process.

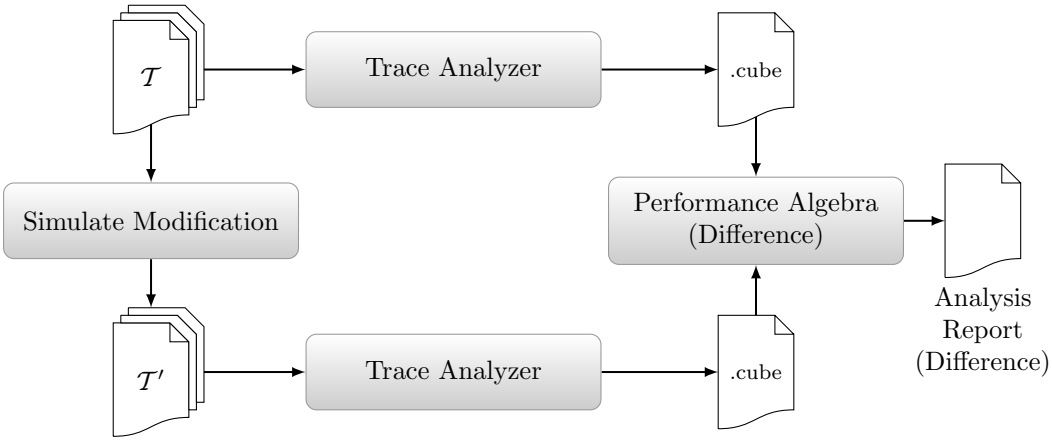


Figure 3.4: Testing correlation between regional instances and performance problems.

If a collective call is eliminated on one process, it must be ensured that it is deleted on all other processes involved in this call, accordingly.

Yet, elimination of regions is not only useful to reduce the simulation space that is being processed, but can also be used to explicitly test the influence of one or more regions on following regions. To test the correlation, as shown in Figure 3.4, the severities of two runs – one with and one without the regions of interest – can be compared. When the severity of a performance problem after the deletion of a region is decreased, it proves the correlation of its occurrence in the trace and a following communication inefficiency. The degree of influence on the performance problem is equivalent to the change in severity of that specific performance problem.

3.4.4 Elimination of messages

Messages or communication between processes in general are synchronization points between processes, thus, directly influencing the scaling behavior of an application. Zero-sized messages are an example where communication may be explicitly used for process synchronization. Even though the message itself has no payload, the receiving process has to wait for the sender to send the message. In practice, these kind of messages can be used for token passing, as a barrier-less synchronization over several processes, where a certain order in processing is needed.

However, these kind of messages can also appear in the system inadvertently, when the size of the message is calculated automatically, and the actual communication is executed without regard of the buffer size or when the receiver doesn't know the size of the message to receive in advance. When happening in all-to-all communication patterns, it can lead to significant waiting times. As a direct result of the synchronizing effect described earlier, messages with payload might have to wait for reception until several other zero-sized messages have been received. Whereas these waiting times are unintentional here, it has to be emphasized that this behavior can sometimes be desired.

Additionally, the elimination of messages with certain properties can give insight to specific behavior of an application. Small messages bring a high latency cost in regard to the transferred bytes. The impact of these latencies can be investigated by comparing the application behavior with and without those messages. Messages can be specified for deletion by size, tag or its occurring region.

4 Implementation

4.1 Introduction

This chapter will introduce the application performance simulator Silas. Its implementation is based on the performance simulation model introduced in Chapter 3. First, the overall architecture of the simulator will be discussed, proposing a trace-based event replay approach for parallel performance prediction, which uses the event trace interface library Pearl. As part of the twofold architecture with simulation model and hypothesis as the core elements, the *reenact model* as well as the corresponding optimization hypotheses are introduced. As Silas does not yet fully implement all proposed optimization hypotheses, this chapter will conclude with future work for extending the simulator's capabilities.

4.2 Architecture

The simulator's main building block is a parallel event replay of the event trace, which is provided by the Pearl library. The Pearl library offers a high-level access to the event trace and provides additional interfaces to easily perform replay-based processing. The heart of the event replay is a callback mechanism that allows the definition and registration of callback handlers for specific native as well as user-defined events.

Figure 4.1 shows the overall architecture of Silas, as well as its external interfaces. The green blocks with dashed border show modules that were developed for Silas, and the orange blocks with dotted border shows the usage of external libraries, here the Pearl library. The original trace is read by the corresponding parts of the Pearl library, while the simulation model and optimization hypothesis are defined in an external configuration file. Trace, model and hypothesis are used by the replay facility of the simulator, which is using a special callback data structure to aid the event trace manipulation. After the simulation is finished, the event trace is handed over to the Pearl writer facility to write the modified trace, resembling the application behavior of the application modified according to the optimization hypothesis. As described earlier, this modified trace can then be analyzed again to evaluate the effectiveness of the applied hypotheses.

The model as well as the optimization hypotheses are encoded using the callback mechanism of Pearl. This means, callback functions are defined and registered for specific events where they should get invoked. The model describes the overall behavior of the simulator, the number of replays used, as well as the techniques used for altering the event trace. A replay can be either local or global. Local replays do not involve inter-process communication other than possibly on start and end of the replay, whereas global replays involve inter-process communication also during the event replay itself. The hypotheses describe change patterns, which are to be applied to the event trace by the simulator.

The simulator supports an arbitrary number of replays of the event trace. This number is determined by the simulation model, as it defines how modifications to the code are performed. Complex modifications might need multiple replays to obtain a good performance prediction. The interface to a model's runtime configuration is given by the model's virtual member function `get_run_configuration()`, which is overridden by each model implementation to fit the model's needs. The configuration is returned as a vector that stores the direction of replay as well as the name. The overall size of the vector determines the number of replays performed.

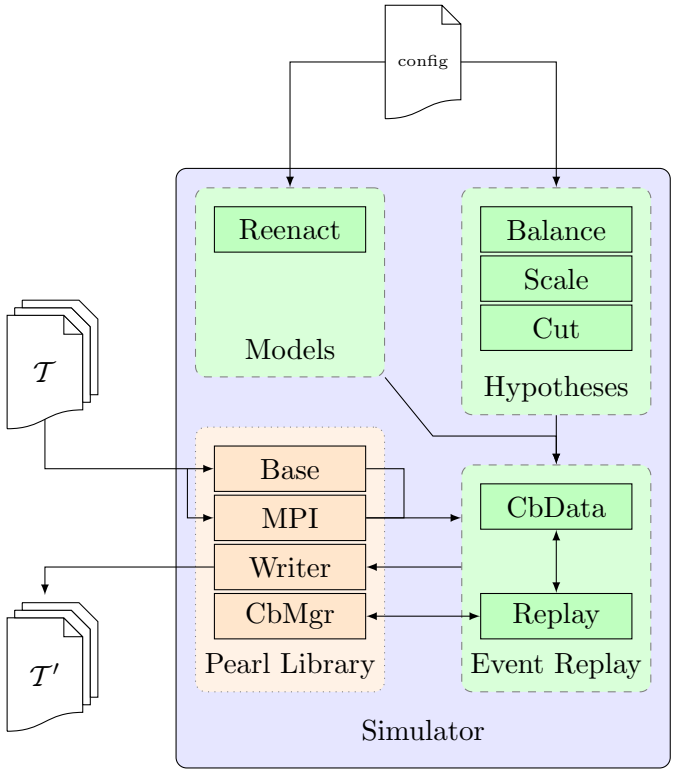


Figure 4.1: Main components of the SILAS simulator

The simulation model as well as the performance hypotheses use callbacks to perform the modifications on the event trace. Even though an hypothesis' modifications to the event trace is independent of the model in general, it has to register its callback methods during specific runs.

The simulation run is then an iteration over all configured runs, with the following phases:

1. Create a callback manager object and register model and hypothesis callbacks for the current run.
2. Signal the START user event.
3. Start a forward or backward replay, according to the configured direction.
4. Signal the FINISHED user event after the replay processed the last event.
5. Start over again, with the next configured replay run or terminate.

After the last replay, the PEARL writer is used to write the modified event trace to disk. A difference in format to the original event trace is only that any attributes with local scope in the original trace have already been modified to the global scope. Applications using this trace do not need to perform this part of the preprocessing again, and can directly use the event trace files with its global identifiers.

4.3 The reenact model

The first model implemented for the simulation is the *reenact* model. The name was chosen as the model is remeasuring the events' timestamps, as the application behavior is reenacted in real time. However, event traces are an abstract description of the application's behavior. Thus, the simulation can neither perform exactly the same computations nor transfer exactly the same

buffers that were transferred during original measurement. This has to be approximated by the model. The reenact model uses real communication with pre-allocated buffers and a waiting function to model the applications behavior during the remeasurement phase.

The events in the event trace can be categorized in two classes: those involving communication and those not. The reenact model uses a look-ahead algorithm to process the event trace, which means, the current event is one event ahead of the actions that have to be taken. For example, a collective communication call is reenacted when the corresponding MPI_COLLEXIT event is processed. In other words, when the evolution of the simulation on a process p reaches event e_i, which means the timestamp of event e_i has just been written, the replay is processing event e_{i+1} to decide what to do with the time span between e_i and e_{i+1}.

The communication callbacks are notified on native events of type EXIT for point-to-point send calls, RECV on point-to-point receive calls as well as the `MPI_Sendrecv`, which implements a deadlock-free, combined send and receive call. The collective communication callbacks are notified on MPI_COLLEXIT events. The MPI_SEND event, for example, is not used as a communication trigger, as it is written to the event trace *before* the transfer begins. As the replay is using the described look-ahead mechanism, the following event needs to be the trigger, which is the EXIT event of the corresponding region. Clearly, the events alone cannot be the indicator for the notification of the callbacks, as a communication would then be triggered on each exit. The region an event belongs to is also taken into account. For example, the point-to-point send is triggered only on EXIT events of the corresponding MPI regions of `MPI_Send`, `MPI_Ssend`, and `MPI_Bsend` calls.

4.3.1 Simulating CPU time

If an event does not notify a communication callback, it triggers the simulation of the time span between itself and the preceding event. This is needed to simulate the correct temporal offsets between the communicating processes.

On UNIX-like systems, waiting a specific time span can usually be accomplished by calling library functions like `usleep` or `nanosleep`. These functions implement an interrupt driven idling, having the operating system wake up the process after the timer has exceeded the specified time. Modern massively parallel systems, like the Cray XT series or the IBM Blue Gene family of supercomputers, often run a reduced kernel on the compute nodes, which do not offer all standard services. For the Blue Gene/L, this includes the timing functions `usleep` and `nanosleep` mentioned above. Thus, a different mechanism for waiting needs to be found. As interrupt-driven waiting is not an option, a busy-waiting scheme is the algorithm of choice. The Blue Gene/L systems provide a very fast and very accurate timer function to query the so-called wall clock time, `rts_get_timebase`. It will return the time in seconds since the partition has been booted. Its accuracy is around 10 nanoseconds.

This enables a time span simulation routine as shown in Listing 4.1. With the time span parameter and a configurable overhead compensation, the end time is calculated in line 5. After this initialization, a loop continuously queries the timing function to check whether the end time is reached. The function is used by the callback function that is registered for so-called internal events, which do not involve communication. Despite this busy-waiting implementation, the terms *idle* and *idling* will be used henceforth to describe the time span simulation. The timing function to retrieve the wall clock time is called `get_wtime` and is also shown in Listing 4.1. It directly calls the runtime system function whose result has to be scaled to the clock speed of the processor. To minimize function call overhead, both function calls are inlined by the compiler.

```
1  inline void ReenactModel::simulate_timespan(double time)
2  {
3    /* calculate the end time for this call */
4    pearl::timestamp_t end_time =
5        get_wtime() + time - idle_overhead;
6
7    while (get_wtime() < end_time)
8        ; /* busy waiting: do nothing else, but checking the time */
9  }
10
11 inline double ReenactModel::get_wtime()
12 {
13   /* Use high resolution timer of the runtime system */
14   return ( rts_get_timebase() * clockspeed );
15 }
```

Listing 4.1: Timing and time span simulation routines for Blue Gene/L

4.3.2 Simulating communication

Reenacting the given communication in an application is a fast and accurate way to deal with several unknowns in the communication library implementation. The measurement of a real transfer transparently models latency costs, link bandwidth, and network congestion.

The event trace provided by the PEARL library layer gives access to the required information, such as the number of bytes transferred, the name of the function call, and other MPI specific information, like the root process of collective operations, where applicable. It can use this information to reenact the message transfer. The communication callbacks used by the reenact model to simulate MPI communication are built much like the wrappers of the measurement system tracing library, yet, as the simulator is not an interposed library, but an MPI application itself, it does not need to use the PMPI-interface.

Listing 4.2 shows an example of a point-to-point callback, here, reenacting an `MPI_Ssend` call. To record the communication time as accurately as possible, the measurement calls directly wrap the call to the corresponding MPI call.

A collective operation is modelled by a normal ENTER event and a special exit event, MPI_COLLEXIT. The collective communication is taking place between these two events, and the parameters to this call are saved with the exit event. Listing 4.3 displays a typical callback for a collective operation, here `MPI_Allreduce`. The handling of timestamp modification is comparable to the point-to-point communication callbacks. Line 21 and 24 show the modification of the ENTER and MPI_COLLEXIT event.

As collective operations are modelled by only two events and the callback management is done in lines 15-18 prior to the first time measurement. The overhead of the measurement wrapper that was initially part of the time span of the collective operation, is eliminated, while the overhead of the callback handler is attributed to the region instance preceding the collective operation. This results in a slightly smaller time attributed to the collective operation.

The communication callback for `MPI_Allreduce` was deliberately chosen to show another aspect concerning simulation accuracy. MPI reduction operations have an associated reduction operator which defines how the communicated data is reduced. The event trace, however, only stores information on how many bytes have been transferred in the call, and does not include the reduction operator. In the simulation callback, this is approximated by using the binary-AND operator. Depending on the MPI implementation, this might lead to discrepancies between the

```
1  void ReenactModel::cb_mpi_ssend(
2          const pearl::CallbackManager& cbman,
3          int user_event, const pearl::Event& event,
4          pearl::CallbackData* cdata)
5  {
6    silas::CallbackData* data =
7        static_cast<silas::CallbackData*>(cdata);
8
9    // cache pointer to preceding (send) event
10   pearl::Event prev = event.prev();
11
12   MpiComm* comm = prev->get_comm();
13   void *buf     = data->get_send_buffer();
14   int count     = prev->get_sent();
15   int dest      = comm->get_rank(prev->get_dest()->get_process());
16   int tag       = prev->get_tag();
17
18   MPI_Comm mpicomm  = comm->get_comm();
19
20   // set send event timestamp
21   prev->set_time(get_wtime() - reference_timestamp);
22   // send message
23   MPI_Ssend(buf, count, MPI_BYTE, dest, tag, mpicomm);
24   // set exit event timestamp
25   event->set_time(get_wtime() - reference_timestamp);
26 }
```

Listing 4.2: Simulation callback for `MPI_Ssend`

time needed for the original communication call and the simulation, yet, testing with real world examples has shown this to be reasonably accurate on the Blue Gene/L platform. This might be due to the hardware support for reductions on the MPI_COMM_WORLD communicator present on the Blue Gene architectures.

Not all MPI communication functions are currently supported by this model. Non-blocking point-to-point communication functions as well as collective calls involving variable send and receive counts are not in the scope of this model yet. Both classes of functions are excluded from the model due to missing information in the event trace. While some of the limitations with non-blocking communication calls could be overcome (see Section 4.7), the collective communication involving variable send and receive counts can only be overcome using either external information provided to the simulator or additional information stored in the event trace.

4.3.3 Improving accuracy

One of the most important issues in performance prediction is the accuracy of the result. When the difference between the simulated result and the real result that is measured after successful code optimization is exceeding a certain level, it will no longer be feasible to use it as a basis for estimating the cost-benefit ration prior to code alteration. It is therefore of absolute importance to reduce the prediction deviation to a minimum.

```
1  void ReenactModel::cb_mpi_allreduce(
2          const pearl::CallbackManager& cbman,
3          int user_event, const pearl::Event& event,
4          pearl::CallbackData* cdata)
5  {
6    /* INFO:
7     * The operation and datatype used are not available from the trace
8     * data. This implies that reenactment of the reduce operation may
9     * deviate from the 'real' reduce used by the application, as an
10    * operation on bytes might take longer than on ints, etc.
11    */
12   silas::CallbackData* data =
13       static_cast<silas::CallbackData*>(cdata);
14
15   int count        = event->get_received();
16   void *sbuf       = data->get_send_buffer(count);
17   void *rbuf       = data->get_recv_buffer(count);
18   MPI_Comm mpicomm = event->get_comm()->get_comm();
19
20   // set enter event timestamp
21   event.prev()->set_time(get_wtime() - reference_timestamp);
22   MPI_Allreduce(sbuf, rbuf, count, MPI_BYTE, MPI_BAND, mpicomm);
23   // set exit event timestamp
24   event->set_time(get_wtime() - reference_timestamp);
25 }
```

Listing 4.3: Simulation callback for `MPI_Allreduce`

Action list

As the reenact model is based on actual measurement of time spans, it is important to reduce the overhead during the simulation run. As shown in one of the previous sections, the notification of a communication function is not only depending on the event type, but also on the enclosing region. A comparison of region identifiers during the simulation can be very costly and distort the result. The reenact model therefore introduces a mechanism to preevaluate the critical parameters that lead to the decision on which callback needs to be triggered, the action list. It stores the user-defined event types that need to be triggered for all events in the event trace. The user-defined event types stored in the action list are called *actions*.

Definition 4.1 (action)
An *action* is a user-defined Pearl event, defined by Silas, to be used directly to trigger a simulation callback. Only one callback per action is allowed to be registered, but several actions are allowed to trigger the same callback.

◇

The action list is assembled prior to the actual simulation to enable a very fast triggering of callbacks. The following actions are defined by Silas:

SKIP is used in the idle time aggregation described later in this section. It will cause the replay to proceed directly to the next event, without calling the pre- and postprocessing methods of the callback data structure.

DELETE behaves much like the SKIP action, yet, it additionally indicates that this event is not considered in idle time aggregation and especially will not be written to the simulated

trace. This action is needed, as the PEARL event trace interface currently does not allow the insertion and deletion of events into or from the event trace.

SKIP_END is a special form of the SKIP action. It indicates the last instance of several consecutive SKIP actions, which has to be handled differently.

INIT is used to initialize the action list. The callback preparing the action list for simulation can subscribe to this action.

BEGIN is used for the first event in the event trace.

END is used for the last event in the event trace.

NOOP is an action that is not associated with a callback, but may still be triggered. The time needed to trigger the event is higher than that of a SKIP and less than IDLE. Details follow later in this section.

IDLE triggers the time span simulation function.

There are also actions for each type of communication call, where the action is named like the corresponding MPI call, without the preceding MPI_ and written all in capital letters. For example, SEND is used as a registration target for the callback initiating an `MPI_Send` call. Other supported communication actions are SSEND, BSEND, RECV, SENDRECV, BARRIER, BCAST, ALLREDUCE, ALLGATHER, and SCAN.

SILAS also defines PEARL user events that will not appear in the action list, but are used to create the network of callbacks to handle the performance prediction and event trace manipulation. These are:

START is a special action that is triggered before each replay run is started. It can be used to setup data structures and initialize the replay, which is started right after it.

FINISHED is the counterpart of START. It is signaled after the last event has been processed.

ANY is a convenience target to enable registration of callbacks on any action.

ATOMIC_REGION_EXIT is used to trigger actions on leaf nodes in the call tree. The term *atomic region* refers to its property of not being interrupted by another function call.

SEND_EXIT is a special trigger that is signaled when a cut hypothesis is active. At the end of a blocking MPI point-to-point send call, it is then decided, whether this call needs to be eliminated.

RECV_EXIT is the counterpart to the SEND_EXIT for a blocking MPI point-to-point receive call.

Simulation of very small time spans

The time span simulation routine naturally has a minimal time span it can accurately simulate. Any time span smaller than this threshold will receive an absolute error of the difference of its original time to this threshold. These time spans are of very small scale, yet, often present in event traces, as they model temporal distances between two consecutive function calls. Time spans higher than this threshold, will be handled by the IDLE action. The reenact model also defines two additional actions: NOOP and SKIP. Those enable the modelling of time spans smaller than the minimum simulation time span. However, both actions have a constant time associated with them. NOOP uses the overhead of the callback manager to model the time

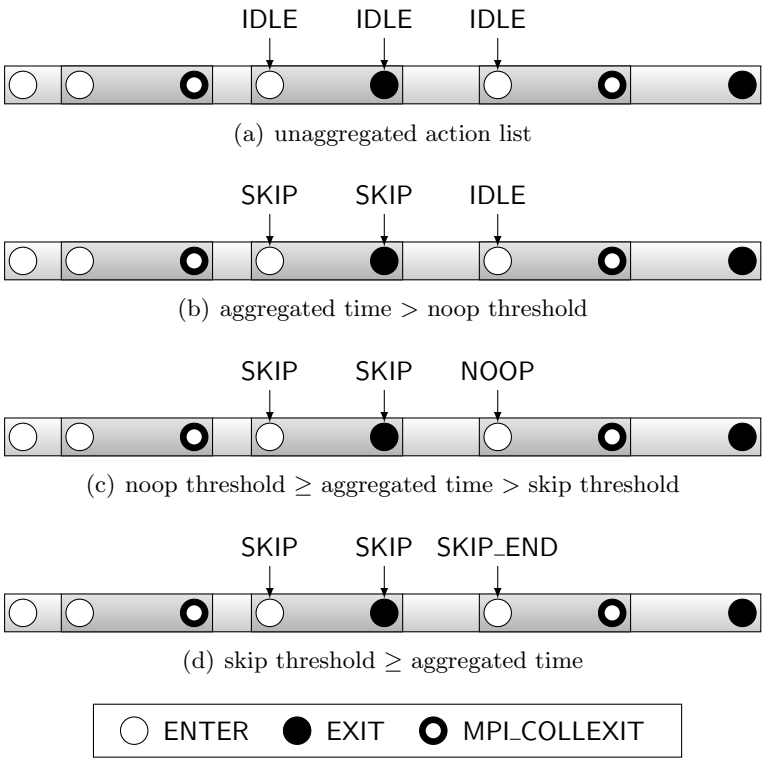


Figure 4.2: Aggregation of consecutive idle actions

span, as it does not receive a callback registration. Thus, the callback manager is signalling the NOOP action, but no additional code will be executed. This overhead is usually lower than the minimal time span that can be accurately simulated by the time span simulation routine. If the time span in question is even smaller than the time modelled by the NOOP action, the SKIP action can be used. This will circumvent any further processing of the event, and will proceed to the next event in the trace. This will result in the smallest overhead possible.

Idle aggregation

When reenacting the relative temporal offset between the processes, special care has to be taken to reduce the overhead to a minimum. While the traces are collected on the same system, their timestamps are subject to the same accuracy. However, very small time spans between events can lead to a significant relative error. As with every individual simulation of a time span an absolute error within a specific range is introduced, one can reduce the resulting relative error, by aggregating several consecutive waiting time spans to a single one, and adjusting the timestamps of the aggregated events after the actual simulation.

As it can easily be deduced from Listing 4.1, the current time span simulation routine will never return before the real time has elapsed, if it is assumed that the parameter *idle overhead* is between zero and the real overhead. The overhead itself can mostly only be estimated, leaving a certain inaccuracy on each invocation of the call.

Usually, an application will have several consecutive events that are all triggering the timespan simulation routine. With each call to the routine, a certain error will be introduced to the measurement. With its *idle aggregation* mode, the reenact model allows the simulation of consecutive timespans with a single call, and recalculates the timestamps of the aggregated events later. To enable this, two additional local replay runs are defined, directly wrapping

the simulation run. As they only process their local event trace and do not reenact the events, those replays are usually much faster than the simulation replay. During the replay before the simulation run, aggregatable time spans are combined and marked in the action list, while the replay after the simulation run enables the calculation of the correct timestamps of the events belonging to the aggregated time spans.

In the event trace a sequence of aggregatable time spans is usually interrupted by a communication time span. Henceforth, the first will be called *idle epoch* and the latter *communication epoch*. Those two epochs alternate in the event trace. This means no two successive communication epochs will ever occur, while two successive idle epochs will always be aggregatable to a single one.

With one single call for several aggregated time spans, the inaccuracy of this single call is spread over several regions, rendering the individual error much smaller than with separate time span simulation calls. Additionally, aggregation of time spans will increase the chance that even very small time frames can be simulated precisely, as they can then be part of an epoch whose dimensions are better suited for the time span simulation routine, rather than one of the approximating actions NOOP or SKIP, which then occur less often. In conjunction with the techniques for the simulation of very small time spans, idle aggregation can significantly reduce overhead in the simulation phase and increase accuracy.

Figure 4.2 shows different modes when aggregating consecutive idle actions. In the situation depicted in Figure 4.2(a), three consecutive IDLE actions are enclosed by two collective exits. These exits have a communication associated with them and therefore form the boundary for the aggregation. The different modes of aggregation only differ in the action used for the last event. All aggregated events but the last one will receive a SKIP action. Deleted events will keep their DELETE action and are not part of the aggregated time frame. The last action defines the action to be used for the aggregated time span. To determine which action is used, the aggregation process is consulting two thresholds introduced in the previous section, which can be specified in the simulator's configuration (see Section 4.5: the *noop threshold* and the *skip threshold*. The *noop threshold* defines the minimal time span that can be efficiently simulated with the waiting time function. If the time span to be simulated is larger than this threshold, the last action will be set to IDLE, which means the simulator will initiate the time span simulation function for the aggregated amount of time, as shown in Figure 4.2(b). If the aggregated time span is less than the *noop threshold* but higher than the *skip threshold*, the NOOP action will be used, as done in Figure 4.2(c). This action is more or less empty, but still produces some overhead, whose temporal extent lies between the two thresholds. This overhead is used to simulate the time spans. If the aggregated time is even less than the *skip threshold*, the action will be set to SKIP_END, as displayed in Figure 4.2(d). Here, all events between the two enclosing communications will be skipped, resulting in the smallest amount of overhead currently possible with this simulation model.

When idle aggregation is enabled, the complete simulation is comprised of alternating communication and idle actions. This enables the measurement of the timestamps to be done entirely by the communication callbacks, as shown in the lines 21 and 25 of Listing 4.2. When idle aggregation is not enabled, the individual idle actions will set the corresponding timestamp on the associated events individually.

When a set of time spans is aggregated, the individual percentage of each time span of the aggregated time is saved. The time for the aggregated time span, measured in the simulation replay, is then distributed to the individual time spans of the idle epoch in respect to their original percentage, and the timestamps of the corresponding events are updated accordingly. As only one time span simulation call is issued, the overall dilation is reduced, and the remaining error is distributed over all time spans of the idle epoch.

Overhead compensation

The anatomy of send and receive regions is always very similar. On sending events, the ENTER and the MPI_SEND event are very close together, as the instrumentation first writes the ENTER event, then possibly does some small calculations and writes the MPI_SEND event, directly before engaging in the transfer. The time between those events is naturally very small and, if seen in context with overhead compensation, pure overhead. On receiving events, this layout is very similar, except the communication event MPI_RECV is coupled very close to the EXIT event, and the time between the ENTER and the MPI_RECV event models the actual transfer. In the combined functions `MPI_Sendrecv` and `MPI_Sendrecv_replace` the MPI_SEND will occur close to the ENTER and MPI_RECV close to the EXIT event.

In relation to the work of Wolf et al. on tracing overhead compensation [29] introduced in Section 2.5.1, this overhead can be reduced to a minimal minimal time span that models the time needed to start the transfer after entering the function. While the time span is just reduced to a minimal temporal extent, it is still part of the idle epoch, which might be subject to idle aggregation. In the aggregated time span the percentage of this overhead is then much less than in the original trace.

Perturbation compensation is a hypothesis by itself, however, always with global scope. It will not make sense to enable this overhead compensation only for a subset of the events, as all events are subject to tracing overhead. Therefore, it is not encoded as a partial hypothesis, but can be enabled or disabled at runtime by a special configuration flag. When compensating tracing overhead, no other optimization hypotheses should be enabled, as it will be hard to interpret the resulting performance increase.

As perturbation compensation of an event trace itself was not the focus of this thesis, is was not fully implemented. However, it its shown to be a valid target for replay-based performance prediction, and might be addressed by future work.

Replay overhead minimization

The original replay mechanism of Pearl involves a flexible interface which allows the subscription to native as well as user events. As the simulator is entirely using user-defined events, this flexibility of Pearl bares an inherent overhead to the simulation. During the simulation replay, first a callback is notified that will then itself do nothing but signal the corresponding action. To overcome this double notification problem, the replay mechanism was modified in two ways. First, the replay was adapted to use the action list of the callback data structure directly as the point of entry into the callback network, rather than the native type of the event. Secondly, with the notification using the action list concept, skipping of certain events was implemented at the core to ensure a very low-cost event skip needed for aggregation and deletion of events.

4.3.4 Replay definition

With the functionality described in the preceding sections, the reenact model now defines four replay steps: hypothesis application, idle aggregation, simulation and postprocessing. The first two replays are local replays to prepare the actual reenactment. The simulation replay is the only global reenacting replay of this model, followed by a last local replay to postprocess some of the events. Before each replay run, the special action START is triggered. This is used in the reenact model to initialize some values in the callback data structure prior to the run. After the last event of the event trace is processed, the special action FINISHED is triggered. Here, some collective data exchange functions are used to exchange inter-process information.

Hypothesis application

During this first replay, as the name suggests, the defined optimization hypotheses are applied to the events in the event trace, by notification of the corresponding abstract events. This means, events are either not touched at all, shifted to a new timestamp, or marked as deleted. The shifting of events is done through an attribute of the callback data structure, Δt. The first callback called on an event in this phase directly shifts the timestamp of the current event by Δt. This applies any shifting due to the shift of prior events to be done first. Then, any further processing decides on how Δt is adjusted before the next event is processed.

During this phase, the event trace as well as internal parameters stored in the callback data structure are set up. This setup includes the tracking of the maximal buffer sizes used for communication and pre-allocation of communication buffers for the simulation phase. For each event, the appropriate action is chosen and saved to the action list also present in the callback data structure.

Idle aggregation

This phase is entirely local to a process, and is dedicated to perform the idle aggregation described earlier. As it is involving only local information and modifications, its execution time is only influenced by the number of events to be processed, as no inter-process communication or synchronization needs to be done.

Simulation

To enable maximal accuracy, the reenact model uses a dedicated replay for measuring the new timestamps that reduces management overhead to a minimum. It builds upon the action list, which is set up in the preceding replay runs. In the simulation replay, the action list is used as a lookup table to minimize the overhead during the simulation phase when deciding on the appropriate action to trigger. During this phase, a maximum of one callback is executed per simulated event and no callbacks are called for skipped and deleted events. The simulation of MPI communication functions is done with individual callbacks for each communication function. This eliminates overhead that would otherwise be introduced by determining the correct communication function during the simulation run. The decision on the appropriate callback is made in advance during the non-time-critical phases prior to simulation.

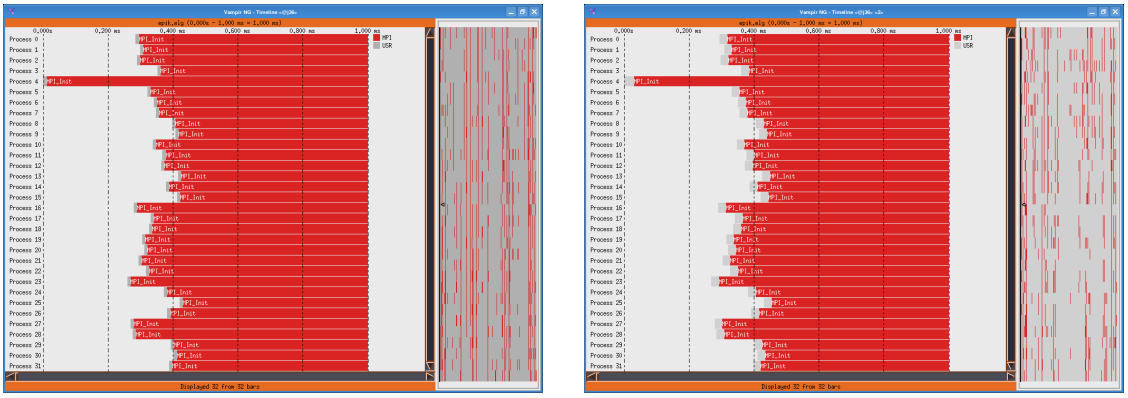
The time in non-communication functions is sent waiting by the individual processes to create the correct temporal offsets between the communication partners. This enables the communication functions to deliver a very precise image of the communication performance to be expected.

Postprocessing

The last replay, which updates all timestamps of events skipped due to idle aggregation, is a local replay again. To adjust the timestamps of the skipped events, the time span between the event associated with the IDLE , NOOP or SKIP_END action and the preceding communication event is determined and distributed on the individual events, according to their share on the original distribution that was saved during the aggregation phase.

4.3.5 Synchronizing the Simulation Startup

As already emphasized, the correct temporal offset between the individual processes is one of the core elements concerning the accuracy of the reenact model. This includes the recreation of the exact temporal evolution of events on process startup. However, the startup of MPI processes



(a) Original trace (b) Simulated trace

Figure 4.3: Original and simulated startup of Sweep3d benchmark.

is not deterministic. It is neither guaranteed that the order of initializing the processes is in process rank order, nor that processes will start in the same order on consecutive runs. If the simulation replay would start from a barrier synchronized state, it would certainly not reflect the original application startup, hence, would produce inaccuracies in the timestamp modification up to the first global synchronization in the application. If the application has no global, synchronizing communication, the complete run would be affected, degrading the prediction accuracy. The simulation replay therefore has to simulate the startup sequence itself, to keep process synchronization as close to the original trace as possible.

In the reenact model of Silas, this is achieved through a specific startup callback that is triggered directly before the replay starts. Here, the timestamps of the first event on each process trace are exchanged to search the minimal timestamp. This timestamp will then work as the *zero-point* for the simulated trace. Then, each process determines its local time span between its first event and the zero-point. All processes are then synchronized by a barrier, after which they wait this determined time span. This ensures that the simulated trace will show the processes start up in the order of the original trace with a minimum of inaccuracies. Figure 4.3 shows the original application trace and the simulated trace in the first millisecond of execution of Sweep3d [27]. Even though the resolution of the figure does not allow a quantitative verification of the accuracy of the simulation, the congruence of the two startup sequences can still be observed on a qualitative level. Simulation accuracy will be discussed in depth in Chapter 5.

This method of temporal synchronization of the processes can in principle also be used to adjourn the simulation run at some point, for example to perform some internal calculations, and then resume the simulation without affecting the prediction accuracy. This may become important when the overall trace size is limiting the amount of additional data the simulator can precalculate. A simulation in several steps would then become possible through consecutive partial simulations of the complete trace.

4.4 Optimization Hypotheses

Several of the partial hypotheses discussed in Chapter 3 have been implemented for the reenact model to prove the effectiveness of the simulation. A full implementation of all optimization hypotheses is beyond the scope of this thesis, but will be addressed in future work.

4.4.1 Scaling of regions

The scaling of regions is currently only available for atomic region instances, which are not interrupted by other flow events. To scale an atomic region, the time span between the ENTER and EXIT event is multiplied with the associated scaling factor. The exit event is shifted appropriately, and Δt is updated to enable successive events to be adapted to the new offset.

4.4.2 Balancing of regions

Similar to the scaling of regions, balancing is currently only available for atomic regions that fulfill the restrictions for the global instance balancing strategy. The average time span is determined by a global reduction operation using the `MPI_SUM` operator, dividing the result by the number of participating processes. The resulting time span is then used to calculate the new timestamp of the EXIT event, and Δt is updated correspondingly.

4.4.3 Elimination of regions

The elimination of regions involves three parts. For events on the local process, it can easily be decided whether an event has to be marked for deletion. Additionally, the inter-process connections with point-to-point and collective communication have to be checked as well. For point-to-point communication, a ping-pong scheme is used to exchange the deletion flag for this transfer between the two communication partners. If either of the flags is indicating a deletion, both processes will delete the events corresponding to this transfer. This has to be done for consistency reasons. For collective communication functions this can be dealt with via a reduction of all local flags using the maximum-operator. If any single flag is set, the reduction will lead to the deletion of the collective communication call. After eliminating a region, all successive events on a process are shifted by the time spanned by the deleted region on that process. The action for the deleted events is set to DELETE. Additionally, the timestamps of deleted events are invalidated by setting them to a negative timestamp, enabling the Pearl writer library to stop the export of these events.

The Pearl writer library uses the replay mechanism to iterate over the event trace and write the events to a new trace file. As the writer is encapsulated in a library, it does not use the callback data structure defined for Silas, which contains the action list. Therefore, the deleted events cannot be identified by a corresponding action in the action list, but need to be marked in existing attributes. As deleted events should not appear in the written trace, it was decided to use the timestamp attribute of the event to indicate deletion. By definition, negative timestamps are now considered invalid, and the writer library was modified to discard such invalidated events, accordingly.

4.4.4 Elimination of messages

The elimination of messages is concerned with deleting specific messages from the event trace. These messages can be identified either by size or by tag. If the hypothesis is referring to the size, the data is only present on the sending process, thus, it has to decide whether the send matches the criteria and notify the receiver via a message transfer. All three events of a point-to-point operation are associated with a DELETE action in the action list and their timestamps are invalidated.

<Configuration>	::=	<Model> <Hypotheses>
<Model>	::=	MODEL <String> <Options>
<Options>	::=	[[<Options> ,] <Option>]
<Option>	::=	OPTION <String> <String>
	|	OPTION <String> <Number>
	|	OPTION <String> <Boolean>
<Hypotheses>	::=	[[<Hypotheses> ,] <Hypothesis>]
<Hypothesis>	::=	<Balance> | <Cut> | <Scale>
<Region>	::=	REGION <String>
<Balance>	::=	BALANCE <Region> <Options>
<Cut>	::=	CUT <Region> <Options>
	|	CUT <Message> <Options>
<Message>	::=	MESSAGE <MessageOption>
<Relation>	::=	== | != | < | <= | > | >=
<MessageOption>	::=	SIZE <Relation> UNSIGNED
	|	TAG <Relation> UNSIGNED
<Scale>	::=	SCALE <Region> <Number>
<Number>	::=	UNSIGNED | INTEGER | REAL
<Boolean>	::=	TRUE | FALSE
<String>	::=	" ? Printable ASCII characters ? "

Table 4.1: The configuration file grammar in EBNF

4.5 Simulator Configuration

To allow a convenient interface to the simulation, the user can influence the model and hypothesis through a configuration file, which is read by the simulator at startup. The grammar of the configuration file is shown in Table 4.1.

The terminal symbols UNSIGNED, INTEGER and REAL describe the corresponding number type and allow different notations, such as omitting the sign on positive integers and floating-point numbers, and using the scientific notation for specifying a floating-point value.

The overall configuration file layout is quite simple. It has two parts, defining the model and its options as well as zero, one or more optimization hypotheses and their corresponding options.

The model options are given as key-value pairs with a preceding keyword OPTION, where keys are string, and values can be either strings, numbers or boolean values. Options with boolean values are referred to as flags. Strings always have to be enclosed in quotes. This approach for configuration enables a fast and flexible way to add new model options without having to change the configuration grammar and with that keeping it simple. All options in the configuration are passed on to the model, which then decides to either interpret the option or to ignore it completely. As the simulator provides an interface for several different models, it is more flexible to keep model specific keywords out of the formal description of the configuration.

The reenact model has several options that can be specified in the runtime configuration:

compensate perturbation enables or disables the perturbation compensation features of the model. Currently, this compensation is implemented only for point-to-point communication events.

aggregate idle can be of value *enabled* or *disabled*. When enabled, the idle aggregation described

```
# Example configuration for SILAS
MODEL "Reenact"
  OPTION "idle overhead" 5e-06
BALANCE REGION "foo"
SCALE REGION "bar" 0.5
CUT MESSAGE SIZE == 0
```

Listing 4.4: An example configuration file

in Section 4.3.3 is activated.

noop threshold indicates the minimal time that can be modelled with the IDLE action. Below this threshold, the NOOP action will be used to simulate time spans.

skip threshold indicates the minimal time that can be modelled with the NOOP action. Below this threshold, the SKIP_END action will be used to simulate time spans.

The hypotheses are constructed by a left-recursive rule, which enables the correct number of hypotheses to be matched by the parser. Currently, the configuration recognizes three different types of hypotheses: balance, cut and scale. These correspond to the definitions of optimization hypotheses in Chapter 3. Their implementation will be discussed in more detail below. Each hypothesis entry starts with a keyword identifying its type: BALANCE, CUT, or SCALE.

The balance hypothesis currently only takes one additional parameter, which specifies the region that should be balanced by its name. Additionally, options to the balancer can be specified to identify the specific balancing mode to be used with this region, as discussed in Section 3.4.2. An option key that is prepared to be interpreted by the simulator is:

mode Describes which balancing mode to be used for this partial hypothesis. Valid values are *global instance*, *process-local*, *global* and *scaled*[1], corresponding to the balancing modes described in Section 3.4.2.

The cut hypothesis is available in two flavors: region and message. The first is specified with CUT REGION, and enables the elimination of a region, including all enclosed regions as well as message transfers and their corresponding events. Collective operations within this region will also be eliminated on remote processes, even if they are not lying within a deleted region on the remote process. This ensures that the processes, where this communication would otherwise not be deleted, do not enter a deadlock, as some processes on the collective communication will never call this function.

Cutting a message transfer applies to point-to-point communication only, and is specified with CUT MESSAGE. Currently, cutting messages of a certain type is a global operation, meaning it is not restricted to certain regions or call paths. The criteria to delete a message can relate to either the size or the tag of the message. Currently, the parameters cannot be combined to refer to a message of size x with tag y.

The scale hypothesis can be applied to a region. The scaling factor is described by a number, which can be given in any number format that is convertible to a floating point number in C/C++.

Comments to the configuration file, which allow the annotation of specific configuration rules as well as quick enabling and disabling, are introduced by the hash character, and reach to the end of the current line. An example configuration file for Silas is shown in Listing 4.4.

[1] currently only global instance balancing is fully implemented

4.6 Limitations

As mentioned earlier, the simulator does not perform execution behavior predictions with instruction-level granularity. Optimization hypothesis are applied on region instances, where each region instance usually contains several instructions. Additionally, SILAS does not take memory bandwidth or cache behavior into account. On shared-memory nodes, several cores compete for the available memory bandwidth on a single node and sometimes share caches. Changes in the inter-process behavior may lead to changes in the memory access patterns on that node. During the execution of the real application this may have an effect on the execution of a specific function, either lengthening the time span used for that region instance or shortening it. However, the simulator cannot reeanct this memory access pattern, and therefore will only simulate the same local execution time spans, as observed during the original application execution. Furthermore, when dealing with load balancing, computational region instances and preceding or succeeding communication calls correlate. Load imbalances based on different amounts of workload, as opposed to equal workload that needs different time spans for processing, may have a similar imbalance in the size of transfer buffers that represent these workloads. This correlation is not expressed by simulation hypotheses yet. Simulation support for non-blocking MPI communication calls can also not be provided by the simulation until the event trace contains the required request tracking information. Special non-blocking function calls like `MPI_Iprobe` or `MPI_Test*`, where the number of region instances present in the event trace is dependent on the relative order of concurrent activities on different processes, would require on-the-fly insertion or deletion of events from the event trace. Additionally, the complex communication algorithms possible with these functions tend to be out of the scope of the proposed simulation model.

4.7 Future Work

As described earlier, not all optimization hypotheses introduced in Chapter 3 have been implemented yet. Therefore, one focus of future work lies on extending the simulator presented in this thesis to support the missing hypotheses. This applies mainly to the scaling and balancing of non-atomic regions. Additionally, the existing features can be subject to a more fine-grained specification. This includes eliminating messages and regions with respect to their enclosing region, as well as allowing multiple options on a single cut hypothesis. This will enable the specification of hypotheses like "Delete all instances of `foo` within `bar`." or "Delete all messages with tag x in region `foo`."

To enable a more sophisticated modification of the event trace, real deletion as well as insertion of events into the trace will be a feature that opens new possibilities for extended optimization hypotheses, like "Replace the call to `MPI_Sendrecv` with two independent calls to `MPI_Send` and `MPI_Recv`" or even, once non-blocking communication can be handled, "Replace the blocking communication in `foo` with non-blocking calls and place the `MPI_Wait` call after `bar`." However, to enable this within the underlying event trace access layer PEARL, the internal data structures have to be changed substantially. Additionally, the PEARL interface to the trace information will have to be publicized, as currently only the timestamp can be modified on an event.

One of the drawbacks of the reenact model is the fact that it needs as much time for the simulation as the original application, while most of the time is spent waiting for the correct moment to engage in communication. As the simulator provides a flexible interface for pluggable models, a model that is based entirely on computation of new timestamps could be created and investigated. As no idling would then be involved to recreate the communication behavior in the trace, it might increase simulation speed. However, the MPI standard explicitly leaves a lot

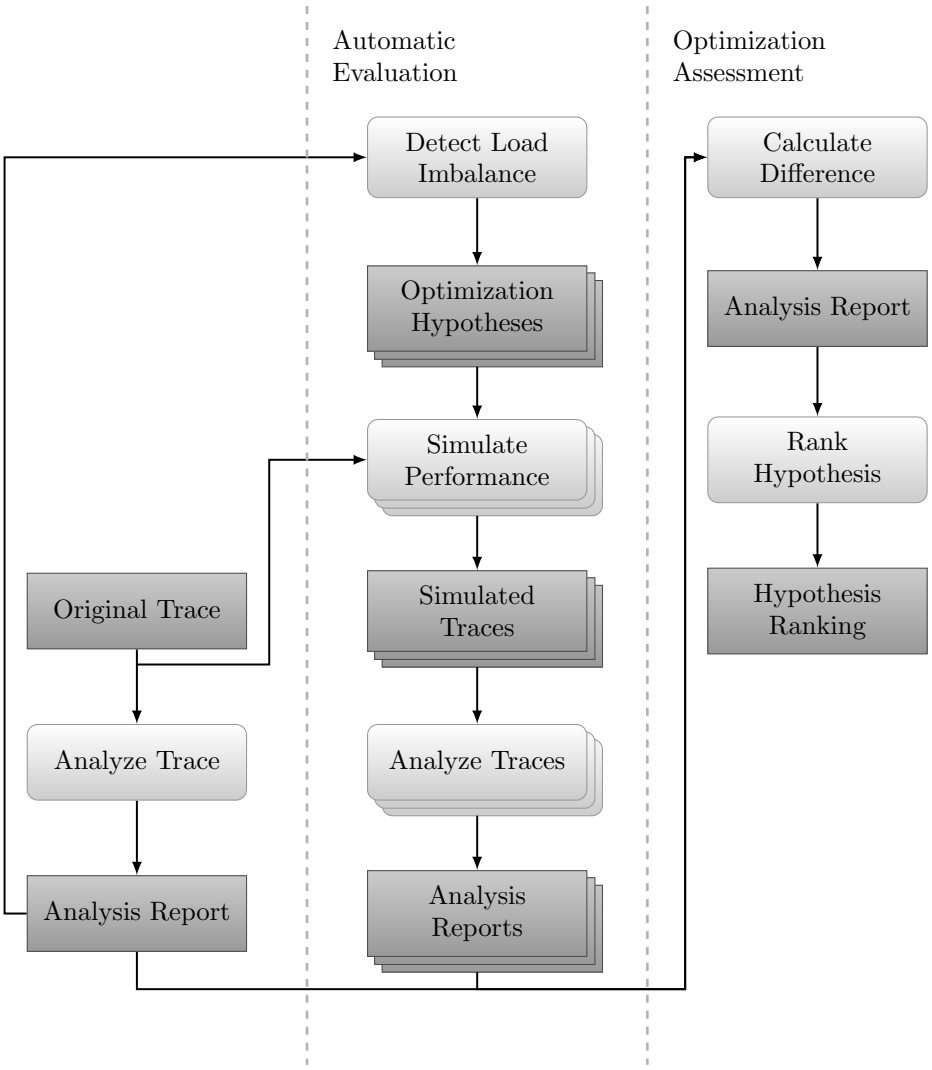


Figure 4.4: Performance simulation in an automated analysis environment.

of freedom for implementors to implement communication calls efficiently. Thus, the model will need to be highly and flexibly configurable to adapt to the MPI implementation in use, and will need a possibility to obtain all influencing parameters, like eager limits, latencies, throughput and similar. If this cannot be obtained, accuracy of the prediction will degrade significantly. In this context, further investigation is needed to show its feasibility.

Finally, performance prediction is still only a tool for further automatic assistance to the user in analyzing and optimizing large codes. Figure 4.4 displays a sketch of how Silas can be integrated in the automatic performance analysis tool set, such as Scalasca.

After the original trace has been analyzed, the analysis report is investigated by a tool to detect load imbalances. According to the imbalances found, different optimization hypotheses are proposed, which serve as input for the performance simulator. The simulator uses the original trace and the specified hypothesis to create a simulated trace, which in turn is subject to performance analysis. While the analysis report of the simulated trace serves as feedback to the load balance detection, it is also used to assess the quality of the hypothesis. The feedback loop in simulation is needed to verify that no additional imbalances have been introduced by the simulation. These are then subject to new optimization hypotheses. The optimization

assessment will create a list of possible optimization strategies and their predicted performance improvement. Now the user can judge on concrete code optimizations more easily and precisely.

5 Results

To demonstrate the effectiveness of the proposed performance prediction, the SILAS simulator was tested in several application scenarios. The results obtained from investigations on synthetic as well as real-world applications are discussed in this chapter. All applications have been investigated on the Blue Gene/L computer system JUBL at the Jülich Supercomputing Centre of the Research Centre Jülich. It consists of 8192 dual-core nodes, interconnected via a 3D torus network for point-to-point communication, a tree-network for collective communication and a signaling network, which is used for process synchronization. The dual-core nodes can be used in two different execution modes: co-processor (CO) and virtual-node mode (VN). In co-processor mode, the first core of each node is executing the application code, while the second core is dedicated to handle the application's communication calls. In virtual-node mode, both cores host a process of the application code, and each core handles both application code and communication calls. The total memory available on a single compute node is 512 megabytes, which is either available to the single process running in co-processor mode, or has to be shared between two processes running in virtual node mode, with 256 megabytes available to each process.

5.1 Synthetic Examples

As predicting the effect of load balancing is one of the central features of the simulator, two synthetic applications have been created. Synthetic applications provide a good means for controlled test cases for the simulation software, tailored to isolate the use of specific features of the simulator. This explicit control can then be used to compare the simulation results with measurements of a real application run of a modified version.

5.1.1 LB-COLL

The synthetic example LB-COLL creates a classic example for the WAIT AT NXN inefficiency pattern. In this pattern a mutual dependency between each pair of the N processes creates waiting times on the individual locations when others join the operation late. In the tested

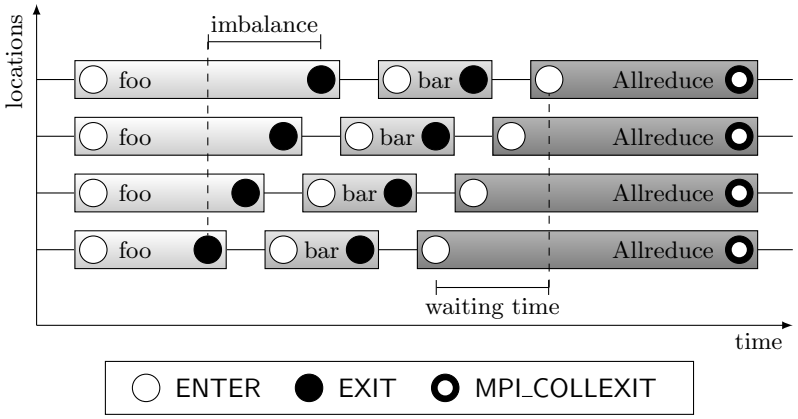
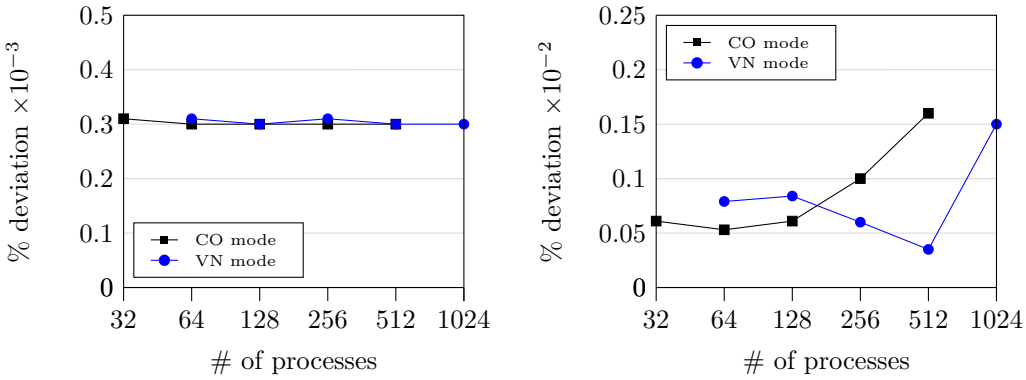


Figure 5.1: Load imbalance causing waiting time in the collective synchronization



(a) identity simulation

(b) global instance balancing of `foo`

Figure 5.2: Percent deviation on LB-COLL simulation

application, the individual processes are entering the synchronizing collective communication call at different times, depending on their rank. As no process can complete the call before all other processes have entered the call, the difference in the time of the ENTER event of a process and the time of the ENTER event of the last process to join is counted as waiting time.

While the time of the EXIT event of a region depends on the computation or communication done inside of the function call, the timestamp of the corresponding ENTER event is dependent only on the history of the process up to that time. This means that the cause for waiting time can not only lie in a load imbalance within the specific region instance, but also in some other region instances preceding the ENTER event of the imbalanced region instance.

Figure 5.1 shows one possible incarnation of this process time line pattern, as it is realized in LB-COLL. The first region instances on the individual locations model the imbalanced calls of function `foo`. The middle region instance of the displayed triple on each location models the call of function `bar`, which is balanced and therefore neither contributing to the imbalance nor decreasing it. The last region instances on each location model the synchronizing collective MPI communication. In the case of LB-COLL the communication is an `MPI_Allreduce`. As this communication scheme involves an all-to-all communication pattern, it possesses an implicit synchronization of all participating processes. In the LB-COLL application, this triple of function calls is repeated 100 times.

To demonstrate that the waiting time within a communication function is not necessarily caused by the directly preceding function call, a call to `bar` is interposed between the imbalanced function call `foo` and the communication routine. Although the call to `bar` is triggered on the individual processes at different points in time, their instances are independent of each other and need the same time to complete on each location. Therefore, it does not influence the imbalance introduced by `foo`.

Figure 5.2 presents the deviation of the simulated performance to the measured application performance. Figure 5.2(a) displays the deviation on identity simulations in co-processor as well as virtual node mode on 32 to 512 nodes on JUBL. Both modes show a very similar pattern on the identity simulation, with less than $0.3 \cdot 10^{-3}$ percent deviation. The event trace was then balanced using the *global-instance balancing* scheme. The deviation of the simulated balancing from the measured performance of the manually balanced code proved to be very small with values less than $0.3 \cdot 10^{-2}$ percent.

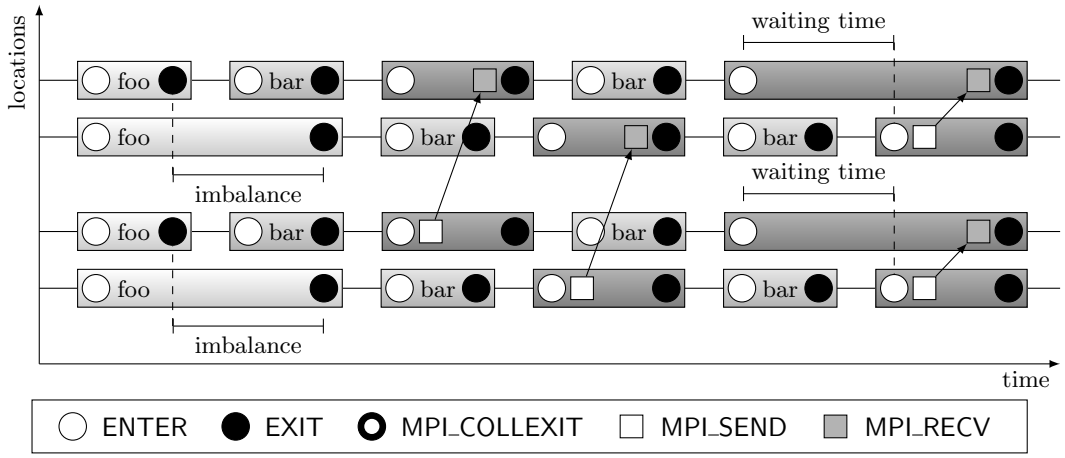
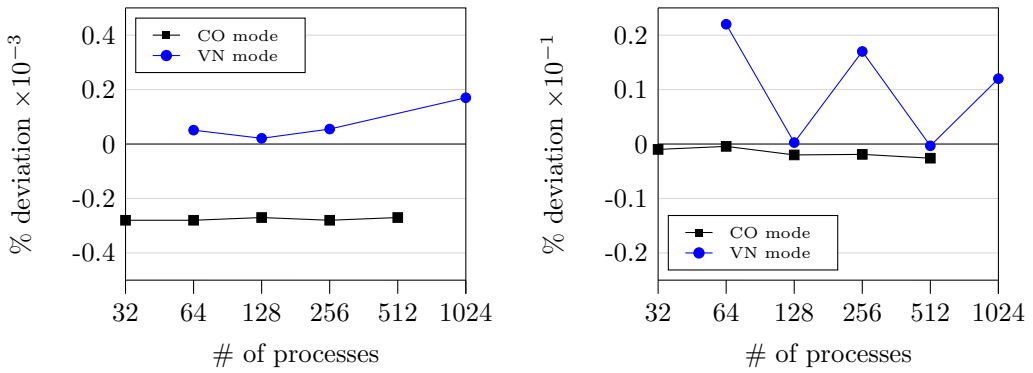


Figure 5.3: Load imbalance causing waiting time in point-to-point communication



(a) identity simulation

(b) global instance balancing of `foo`

Figure 5.4: Percent deviation on LB-P2P simulation

5.1.2 LB-P2P

LB-P2P is a synthetic application creating controlled waiting time in a point-to-point communication scenario. A schematic view of the pattern is shown in Figure 5.3. Again the application calls two user functions: `foo` with variable time per process, and `bar` with constant time per process. In point-to-point communication, the synchronization is evidently only between the two processes involved in the communication. The communication pattern in LB-P2P demonstrates this by creating a load imbalance between odd and even ranks. Then, two point-to-point transfers are started. The first transfer is issued between two odd or two even processes, where both processes have the same offset in regard to the global imbalance. Therefore, the `MPI_Send` on the sender as well as the `MPI_Recv` on the receiver are started at the same time, and no additional waiting time is created. The second transfer is issued between one odd and one even rank, which still have a time offset to each other. The `MPI_Recv` is started on the receiver much earlier than the `MPI_Send` on the sender, creating waiting time, which is detected by the LATE SENDER inefficiency pattern. This function call pattern is repeated 100 times.

The simulated balancing of the region `foo` eliminates the majority of the LATE SENDER pattern, and the overall prediction is again very close to what can be measured when changing the original application, as it can be seen in Figure 5.4(b).

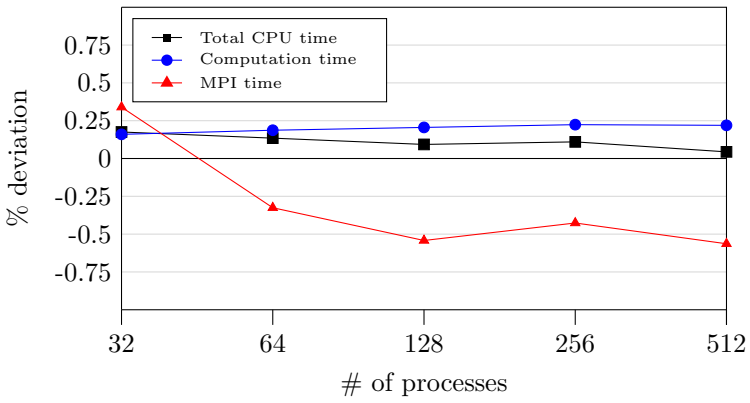


Figure 5.5: Percent deviation of identity simulation of Sweep3D

5.2 Real World Applications

After the feasibility of the performance simulation was proven with the synthetic examples, real world applications were analyzed and the resulting traces used for performance simulation. The simulation of real world applications poses a greater challenge on the accuracy of the simulator, as the overall CPU time, the event frequency as well as the number of events are much larger compared to the synthetic test applications.

5.2.1 Sweep3D

The benchmark code Sweep3D [27] is an MPI program performing the core computation of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian geometry neutron transport problem by calculating the flux of neutrons through each cell of a three-dimensional grid (i, j, k) along several possible directions (angles) of travel. The angles are split into eight octants, each corresponding to one of the eight directed diagonals of the grid.

To exploit parallelism, Sweep3D maps the (i, j) planes of the three-dimensional domain onto a two-dimensional grid of processes. The parallel computation follows a pipelined wavefront process that propagates data along diagonal lines through the grid.

Responsible for the wavefront computation in the code is a subroutine called `sweep`, which alternately initiates wavefronts from all four corners of the two-dimensional grid of processes. The wavefronts are pipelined to enable multiple wavefronts to follow each other along the same direction simultaneously. Thus, the parallelization in Sweep3D is based on concurrency among algorithmically independent processes and pipelining among algorithmically dependent processes.

The Sweep3D code is written in Fortran77 with the addition of automatic arrays and the usage of a C timer routine. For the simulation this is irrelevant, as the simulator is working only on the measured event traces. However, for an application to be traceable with Scalasca it must be written in Fortran77 and newer, C/C++ or a combination of those languages.

Due to its specific communication patterns and the waiting times implied, Sweep3D has been subject to performance investigations in the past [6, 17]. Though the waiting time is inherent to the communication pattern, and can only be optimized by completely changing the communication pattern, which currently cannot be simulated with Silas, it proved a good candidate for testing identity simulation, as it involves point-to-point as well as collective communication, and is scalable to large numbers of processes.

Figure 5.5 displays the percentage of deviation to the original trace for the aggregated CPU time, the communication and computation time. Communication time is the overall time spent in MPI communication and synchronization functions. Computation time is the remaining time of the aggregated CPU time without the communication time. The computation time shows a small positive absolute error to the simulation of timespans, while that of the communication time of the simulation is negative. This is potentially caused by two factors: the approximation of reduction functions described in Section 4.3.2 causing the communication to perform faster as well as a different temporal process offset while reenacting the communication due to the execution time simulation dilation. Still the overall accuracy of identity simulation is very high, with an overall CPU time deviation of less than 0.1 percent.

5.2.2 XNS

XNS is an academic computational fluid dynamics code (CFD) for simulation of unsteady fluid flows [5]. Its simulation capabilities include micro-structured liquids in situations with significant deformation of the computational domain. It is based on finite-element methods, using unstructured meshes and iterative solution strategies.

It consists of more than 32 thousand lines of Fortran90 code distributed over 66 files. The EWD substrate library is used to encapsulate the use of BLAS routines. XNS is parallelized via the use of message passing libraries, and is portable between a wide range of computer architectures. It was also successfully ported to the Blue Gene architecture, but until the first Jülich Scaling Workshop it failed to scale to several thousands of processes [32].

The simulation studied at that time was a test-case of a 3-dimensional space-time simulation of the MicroMed DeBakey axial ventricular assist blood pump. The mesh used in the simulation had a resolution of 3,714,611 elements. Domain decomposition was done using the METIS graph partitioner to create element sets, which form contiguous subdomains. These subdomains were then assigned to the processes.

XNS uses an iterative solver, and several tests with longer running simulations showed that deviation of time between individual iterations was very small and that the first iteration could be chosen as a representative in analyzing the iteration step. The application was therefore configured to perform only a single iteration.

Whereas the computational parts scaled well to higher number of processors, initial performance analysis showed that the communication routines performing scatter and gather operations between the processes became increasingly dominant for the overall performance. As the computational work per process decreased with increasing processor counts (strong scaling), this behavior is not uncommon for distributed-memory parallelizations.

Further analysis showed that calls to `ewdgather1` and `ewdscatter2` would eventually dominate the simulation performance, as they were issuing an increasing amount of communication calls using `MPI_Sendrecv`. Both function calls are part of the EWD substrace library and are used to exchange simulation data between the subdomains. The payload of the individual transfers would vary substantially, however, as the communication is used to exchange boundary information between the processes it is not uncommon to be like this, either. Additionally, the data exchange was done for every processor pair, with a growing number of transfers being issued with an empty payload.

During optimization of the application, the communication routines using `MPI_Sendrev` were modified to use separate calls to `MPI_Send` and `MPI_Recv`. Then, the message transfers with empty payload were suppressed, resulting in a significant performance gain and a radical increase in scalability. In this example, the elimination of messages was easy. Since a static partitioning is used, all processes could determine the number of elements linked to each partition, and with that the amount of data to be transferred between a pair of processes.

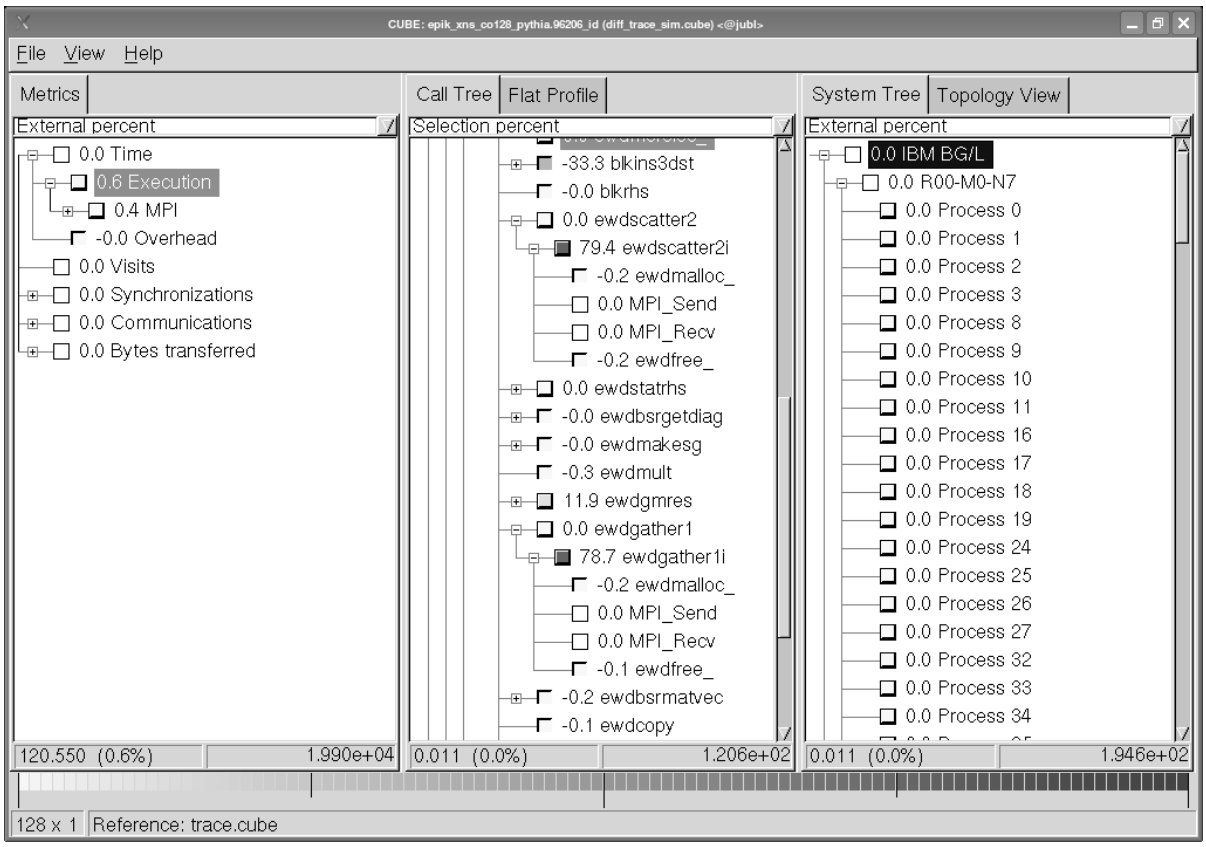


Figure 5.6: Percent deviation on Xns identity simulation on 128 nodes

Xns was chosen as a real-world application test case for optimization hypotheses of Silas, as it is an example where performance improvements were achieved by optimizing the original application, and results of the corresponding simulation could be verified. The domination of the application's runtime clearly showed only on problem sizes involving more than 900 processes. However, handling of traces of this size from long running applications is not trivial. The amount of events written to the event trace will lead to a significant dilation of the measurement or inhibit the measurement completely. The largest trace that was obtained by measuring the unoptimized application was a 1024 process run in co-processor mode on JUBL. The overall trace size is 68 gigabytes with a total of 8,798,565,118 events and an average count of 8,592,349 events per process. Memory limitations of Silas currently prevent simulations of this size, therefore, the basis for performance simulation was a 128-node trace of an application run. Event though performance improvements between the unoptimized and optimized version of Xns are not so evident as with process counts higher than 1000, it presents a valid first test case.

The design of the function calls `ewdscatter2` and `ewdgather1`, however, poses a significant challenge to simulation accuracy. In those function calls individual communication calls are issued in a for-loop. In addition to very small payloads, the event rate in this part of the simulation is extremely high, with alternation between very small message transfers and short computational times. This means, the resulting idle and communication epochs are of very small scale, which leads to a high relative error rate, even if the absolute error rate is still very small. Figure 5.6 shows the analysis report of the identity simulation of an Xns trace. The analysis

report viewed was created using the CUBE_DIFF utility, which uses a so-called performance algebra [26, 25] to calculate the difference between two experiments in the form of CUBE analysis reports for each individual node of the hierarchies being displayed. The reports used for this particular difference were the measured trace of the unoptimized XNS application on one hand and the identity simulation of this trace on the other. This file is then viewed using the "External percent" display mode of CUBE, where percentage values are displayed in relation to another analysis report. Here, the referenced analysis report is the measured trace of the unoptimized application. In this setup positive numbers will indicate parts of the applications behavior where the simulation needed more CPU time than the original unoptimized application, while negative values indicate the simulator being faster than the measurement of the unoptimized code. All values of the left pane, which displays the performance properties of XNS, are therefore to be read as the percentage of dilation of the simulation run. Additionally, the call-tree pane of CUBE is set to show "Selection Percent", which in turn shows the distribution of the node and value selected in the left pane in the call tree.

The overall deviation in computation time is mainly attributed to the two function calls `ewdscatter2` and `ewdgather1`, which can be identified in the middle pane by their large severity and dark squares. The figure shows that those two functions are contributing significantly to the overall deviation. With the selected view and "Execution" selected in the performance property pane, it can be seen, that the computation time is deviated by 0.6 percent of the original runtime of the trace. `ewdscatter2` and `ewdgather1` both show a positive deviation of roughly 80 percent of the selected performance property, which would account for more than the 0.6 percent of computation time. This is possible as the value in the performance property pane is aggregated over all call-tree nodes, with positive and negative deviation cancelling each other, resulting in the presented example.

The functions in question implement the data distribution between the steps of the iterations. While on a small scale, the individual processes have fairly large domains, with common borders to some of the other processes, this rapidly changes when increasing the process count, as the subdomains become smaller and the amount of adjacent subdomain to a single subdomain is a lot smaller than the overall count of subdomains. With the many-to-many communication pattern of gathering and scattering data between the processes, the count of zero-sized messages is increasingly disproportionate to the count of messages with payload, rendering most of the communication without any use to the simulation itself. As mentioned, the optimization of XNS involved the elimination of these zero-sized messages, to make transfers between processes only if data actually needs to be exchanged.

As the PEARL event trace interface provides only read access to the event information except the event timestamp, this modification could not be simulated, but had to be done by the user. Thus, the simulation could only start on the intermediate step of individual send and receive calls. Figure 5.7 shows the difference in analysis data of measurements of the optimized and unoptimized code versions. Figure 5.8 is displaying the predicted difference in application performance based on the unoptimized version, simulating the corresponding code modification. Even with the difficulties of accurately simulating minimal time spans, the overall prediction of time savings in those two function calls is yielding good results. The function calls of `ewdscatter2` and `ewdgather1` can easily be identified in the figures, with their higher severity than the surrounding calls in the call tree, shown in the middle pane. The time savings of `ewdscatter2` in this part of the call tree amounts to 363 seconds, while its prediction yielded 375 seconds. The measured performance increase of `ewdgather1` in this part of the code is yielding 332 seconds, while the prediction is 311 seconds.

Most of the predicted application behavior outside of the investigated call tree shown in Figure 5.8 is within the accuracy bounds already shown in the identity simulation of the unoptimized code, which was discussed earlier. There is a deviation of computational time that is due to non-

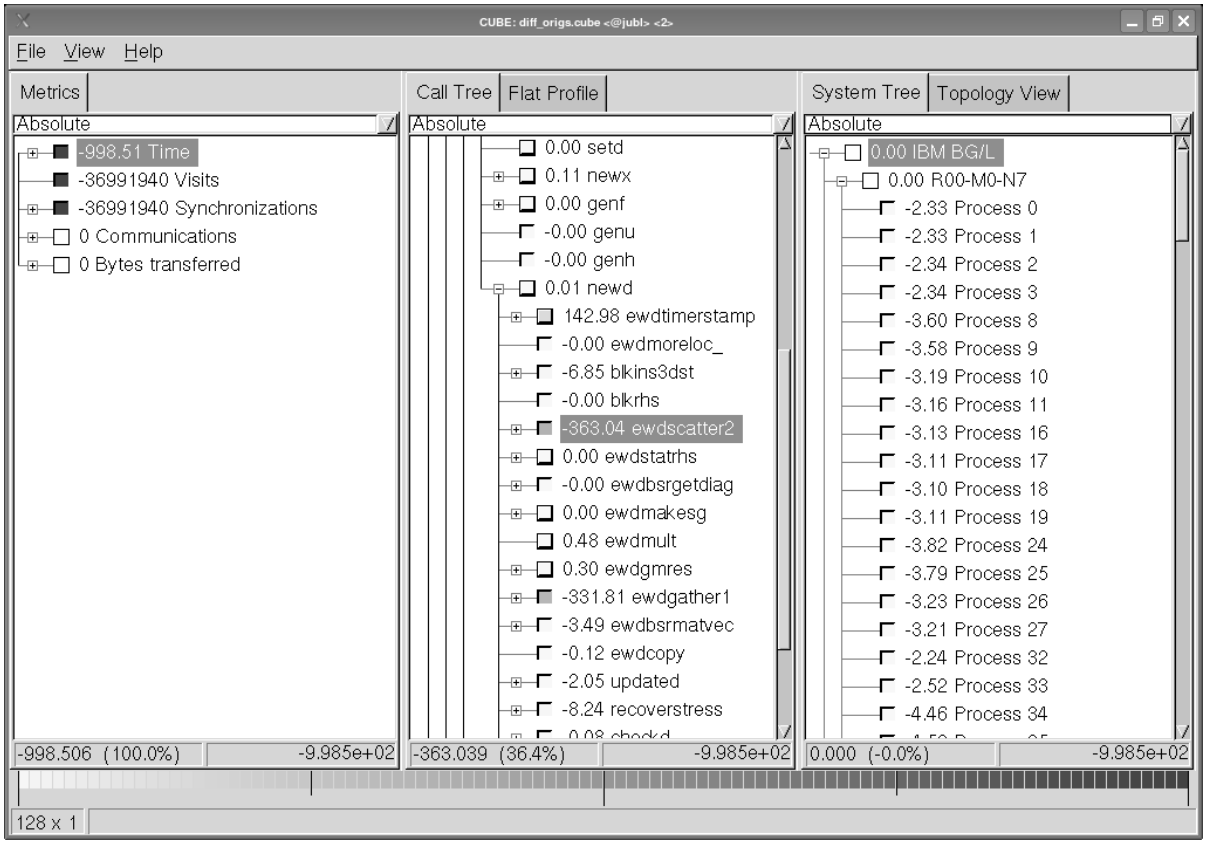


Figure 5.7: Measured performance improvements of Xns on 128 nodes

deterministic behavior of function calls involving disk I/O. With varying load on the file system, these calls will take a different amount of time to perform the same task. The simulation of the optimization hypothesis and the measurement of the optimized application are two individual runs, and while the simulation is simulating the exact same time needed for those functions as in the run of the unoptimized application, the run of the optimized application shows a different performance. However, not all deviation of the predicted performance is attributed to computational regions. The calls to `ewdscatter2` and `ewdgather1` within the function call `genbc` show a significant difference in communication times, however, the cause for this could not yet be identified. Further investigation on this is the subject of future work on Silas.

5.3 Future Work

The investigations of the Xns application and early prototypes of a load imbalance pattern in the analysis tool Scout led to the discovery of multiple instances of load imbalance. Some of the imbalances are located on inner nodes of the call-tree, thus the support for balancing inner nodes has to be implemented prior to their investigation. One occurrence that was located in a leaf node of the call tree could be hypothetically balanced by Silas, which yielded a six percent performance gain, if eliminated completely. In collaboration with the developers of Xns, it has to be investigated whether a corresponding code optimization is feasible.

Current investigations of Silas simulations of Xns mainly focus on the execution on 128 nodes in co-processor mode. Larger Xns event traces than the one investigated in this thesis

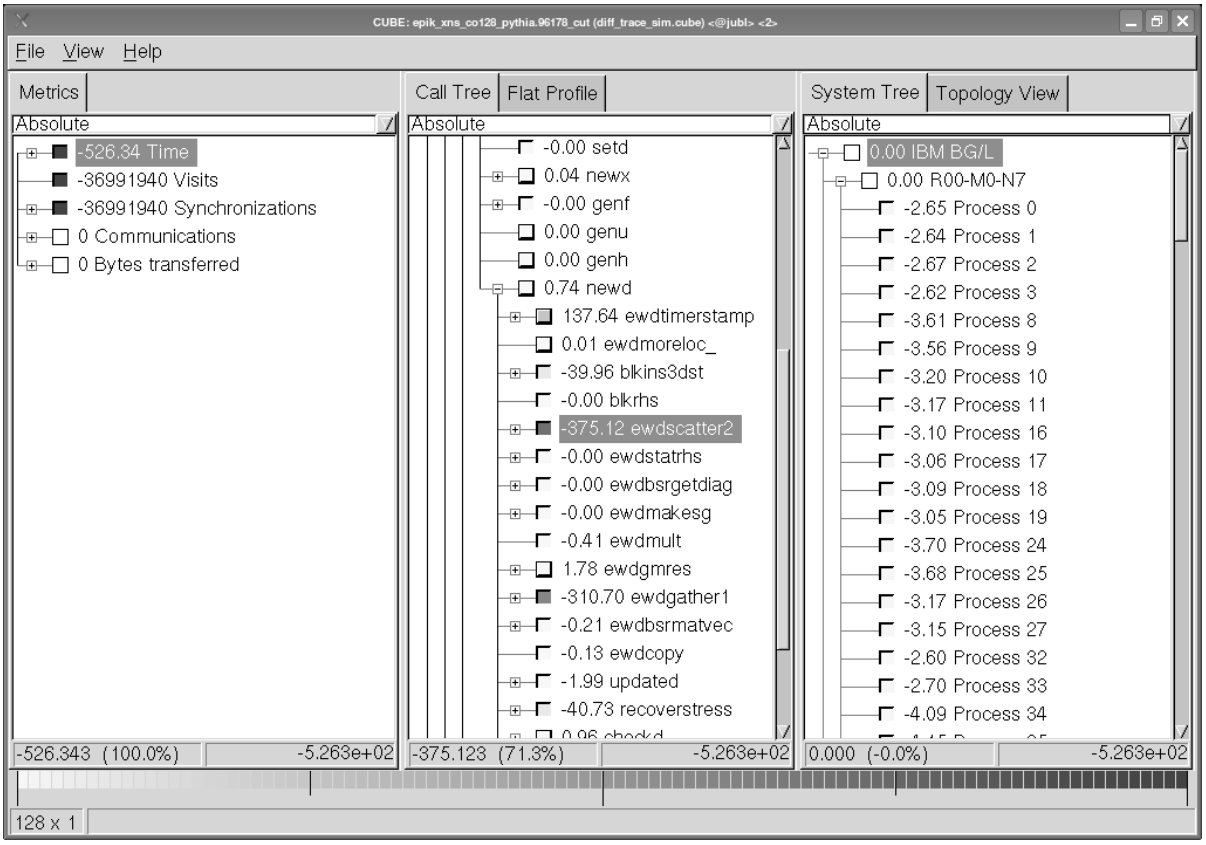


Figure 5.8: Predicted performance improvements of Xns on 128 nodes

have been created, such as a 1024 node trace in co-processor mode with a total size of 67 gigabytes. Simulation on this event trace posed a new challenge to the simulation infrastructure, and memory limitations led to significant inaccuracies, as the idle time aggregation had to be disabled on these large simulations. In parallel to the development of Silas, the Pearl infrastructure was improved in its memory consumption behavior. To enable the simulation of those large traces, the modifications of the Pearl trace interface have to be incorporated into the code base of the simulator. Additional support for large traces could be obtained through iterative simulation of the event trace, which was already outlined in Section 4.3.5. Also, a pure computational simulation model may yield a smaller memory footprint.

6 Conclusion

In scientific computing, applications face a growing number of cores available in the supercomputing systems they run on. With the newly emerging massively parallel architectures comprising several thousands of cores, the new frontier for large-scale applications will be the efficient use of the provided computational power to expand the possibilities of simulation as the third pillar of research. Whether a code is fit to run on several thousands of cores is not easily predictable from data of runs on small scales, as some algorithms show their weaknesses only on this very large scale. With performance evaluation of those algorithms conducted on production scale, the application developer can get detailed insight to the complex behavior of his large-scale application.

Modern supercomputing environments allow applications to run efficiently with several hundreds of thousands of processes. However, only few application codes fulfill all requirements to run at this scale. One of the central issues in optimizing large-scale codes is the complexity of the parallel system that needs to be handled. Not only is it sometimes hard to estimate the behavior of a software system running at large scales, it is usually also a challenge to work through the vast amounts of data gathered when monitoring such a large-scale run. To increase the productivity of the optimization process, the user needs assistance in reducing this complexity.

Scalasca is a set of tools for automatically searching for inefficiency patterns in event traces. It provides mostly automatic user assistance through several steps in the performance optimization cycle, as introduced in Chapter 2, ending in the presentation of a distilled performance report. The presentation of this performance report guides the investigator to the most performance-relevant points in the application. Yet, the evaluation of this report still needs significant intervention by the user.

This thesis presents the application performance simulator Silas, an event-replay-based performance prediction tool to aid the user in estimating the effects on the application behavior after specific changes in the application. The basis for performance prediction is an abstract application behavior model provided by an event trace. The event trace interface for the simulator is provided by the Pearl library, which is part of Scalasca. It provides efficient access to the distributed event trace and supports parallel iterative processing using a callback mechanism. The simulator uses this callback mechanism to define actions to be taken on certain events in the event trace. Those callback functions can be divided into two classes: those belonging to the model and those belonging to the simulated optimization hypothesis. The model comprises all callbacks that deal with the actual performance prediction whereas the optimization hypothesis contains all callbacks that modify the trace data to resemble the specified code changes. The simulator performs one or more event replays that are defined by the model to reach a final state where the event trace in memory corresponds to the application's behavior after the proposed optimization. The result is a new event trace that can then be analyzed and compared to the original one by the use of the existing trace analysis, performance algebra and presentation tools.

The presented simulator focusses on the prediction of large-scale application behavior. It simulates changes in the inter-process behavior after modifications of the code. It will not answer the question of which order is best for the execution of consecutive local instructions, for example to improve cache usage, but will give a prospect of the global performance change that can be expected when a certain part of the application runs faster.

The accuracy of the proposed performance prediction using the reenact model was shown in Chapter 5 on the synthetic applications LB-P2P and LB-COLL. Both applications create a controlled load imbalance scenario, one involving point-to-point communication and the other involving collective communication. The simulator proved to be very precise in the prediction of the behavior of these synthetic applications, with mostly less than 0.1 percent deviation from the measured performance of an optimized application version.

Additionally, two real world applications were tested with the current implementation of the simulator: SWEEP3D and XNS. The ASCI benchmark application SWEEP3D was used as a test subject for the identity simulation to verify simulation accuracy. Although the absolute deviation was higher than on the synthetic applications before, the results still confirm an accurate performance prediction. The fluid dynamics application XNS was the second application under investigation. The fact that for a specific prior optimization code versions with and without optimization are available, created an excellent test case to verify whether the performance gain achieved would also have been predicted by the simulation. XNS proved to be a hard test case, as it contained a large number of very small time spans between events, which largely influences the simulation precision of the proposed reenact model. Still, the results that could be obtained by simulating the optimization show a deviation of less than three percent in CPU time from the simulated optimization to the measurement of the optimized code version.

As some aspects of the simulator exceed the scope of this thesis, complete support for the proposed optimization hypotheses is the subject of future work. Additional work will also include increasing the prediction accuracy, and reducing the memory footprint of the simulator. As such, the XNS application code will be subject to further load balance simulation and general deeper investigation.

Bibliography

[1] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, number 30, pages 483–485, 1967.

[2] Rosa M. Badia, Francesc Escale, Edgar Gabriel, Judit Gimenez, Rainer Keller, Jesús Labarta, and Matthias S. Müller. Performance Prediction in a Grid Environment. In *1st European Across Grids Conference*, Santiago de Compostela, Spain, 2003.

[3] Rosa M. Badia, Jesús Labarta, Judit Gimenez, and Francesc Escale. DIMEMAS: Predicting MPI applications behavior in Grid environments. In *Workshop on Grid Applications and Programming Tools (GGF8)*, 2003.

[4] Rosa M. Badia, G. Rodriguez, and Jesús Labarta. Deriving Analytical Models from a Limited Number of Runs. In *Minisymposium on Performance Analysis, ParCo 2003*, 2003.

[5] M. Behr, D. Arora, O. Coronado, and M. Pasquali. Models and finite element techniques for blood flow simulation. *International Journal of Computational Fluid Dynamics*, 20:175–181, 2006.

[6] N. Bhatia, F. Song, F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Automatic Experimental Analysis of Communication Patterns in Virtual Topologies. In *Proc. of the International Conference on Parallel Processing (ICPP)*, pages 465–472, Oslo, Norway, June 2005. IEEE Society.

[7] CUBE3 XML Schema. cube3.xsd, provided as part of the SCALASCA distribution. `http://www.scalasca.org`.

[8] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennesse, Knoxville, 1994. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[9] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. University of Tennessee, Knoxville, 1996. `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`.

[10] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. N. Wylie. A Parallel Trace-Data Interface for Scalable Performance Analysis. In *Proc. 8th Workshop on State-of-the-art in Scientific and Parallel Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 398–408. Springer, June 2006.

[11] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference*, volume 4192 of *Lecture Notes in Computer Science*, pages 303–312, Bonn, Germany, 2006. Springer.

[12] Sergi Girona and Jesús Labarta. Sensitivity of Performance Prediction of Message Passing Programs. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Monte Carlo Resort, Las Vegas, USA, July 1999.

[13] Sergi Girona, Jesús Labarta, and Rosa M. Badia. Validation of Dimemas communication model for MPI collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI User's Group Meeting*, Balatonfüred, Lake Balaton, Hungary, September 2000.

[14] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[15] John L. Gustafson. The Scaled-Sized Model: A Revision of Amdhal's Law. In *Proceedings of the 3rd International Conference on Supercomputing (ICS)*, volume II, pages 130–133, 1988.

[16] Marc-André Hermanns. Ereignisbasierte Leistungsanalyse von Remote-Memory-Access-Operationen. Technical Report ZAMIB-2005-15, Research Centre Jülich, December 2004.

[17] Kevin A. Huck, Allen D. Malony, and Alan Morris. Design and implementation of a parallel performance data management framework. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 473–482, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Andreas Knüpfer, Holger Brunst, Allen D. Malony, and Sameer Shende. *Open Trace Format API Specification Version 1.1.* TU Dresden and University of Oregon, November 2006.

[19] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[20] B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. In *2nd Annual Los Alamos Computer Science Institute Symposium (LACSI 2001)*, Santa Fe, New Mexico, October 2001.

[21] G. Moore. Progress in Digital Integrated Electronics. In *Proceedings of IEEE Digital Integrated Electronic Device Meeting*, page 11, 1975.

[22] G. Rodriguez, Rosa M. Badia, and Jesús Labarta. Generation of Simple Analytical Models for Message Passing Applications. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 183–188. Springer, 2004.

[23] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and L. V. Kalé. Emulating Petaflops Machines and Blue Gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

[24] S. Shende, A. Malony, A. Morris, and F. Wolf. Performance Profiling Overhead Compensation for MPI Programs. In *In Proc. of the 12th European Parallel Virtual Machine and Message Passing Interface Conference (EUROPVMMPI)*, volume 3666 of *Lecture Notes in Computer Science*, pages 359–367, Sorrento, Italy, September 2005. Springer.

[25] Fengguang Song, Felix Wolf, Farzona Pulatova, Markus Geimer, Daniel Becker, and Brian Wylie. *CUBE3 - User Manual.* University of Tennessee and Research Centre Jülich, February 2007.

[26] Fenguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An algebra for cross-experiment performance analysis. In *International Conference on Parallel Processing (ICPP)*, volume 1, pages 63 – 72. IEEE Computer Society, August 2004.

[27] The Sweep3D benchmark. distributed as part of the ASCI (Accellerated Strategic Computing Initiative) benchmark suite.

[28] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. PhD thesis, RWTH Aachen, 2003. ISBN 3-00-010003-2.

[29] F. Wolf, A. Malony, S. Shende, and A. Morris. Trace-Based Parallel Performance Overhead Compensation. In *Proc. of the International Conference on High Performance Computing and Communications (HPCC)*, volume 3726 of *Lecture Notes in Computer Science*, pages 617–628, Sorrento, Italy, September 2005. Springer.

[30] Felix Wolf, Bernd Mohr, Nikhil Bhatia, Marc-André Hermanns, and Markus Geimer. *The EPILOG Binary Trace-Data Format*. Jülich Supercomputing Centre.

[31] Brian J. N. Wylie, Felix Wolf, Bernd Mohr, and Markus Geimer. Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. In *Proc. 8th Workshop on State-of-the-art in Scientific and Parallel Computing*, number 4699 in LNCS, pages 460–469. Springer, June 2006.

[32] Brian J.N. Wylie, Markus Geimer, Mike Nicolai, and Markus Probst. Performance analysis and tuning of the XNS CFD solver on BlueGene/L. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Proceedings of the 14th European PVM/MPI User's Group Meeting)*, volume 4757 of *Lecture Notes in Computer Science (LNCS)*, pages 107–116. Springer, 2007.

[33] Jerry Yan, Sekhar Sarukkai, and Pankaj Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software – Practice and Experience*, 25(4):429–461, 1995.

[34] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.

[35] Gengbin Zheng, Terry Wilmarth, Orion Sky Lawlor, Laxmikant V. Kalé, Sarita Adve, David Padua, and Philippe Geubelle. Performance Modeling and Programming Environments for Petaflops Computers and the Blue Gene Machine. In *NSF Next Generation Systems Program Workshop, 18th International Parallel and Distributed Processing Symposium(IPDPS)*, page 197, Santa Fe, New Mexico, April 2004. IEEE Press.

Jül-4297
Mai 2009
ISSN 0944-2952