Automatic Trace-Based Performance Analysis of Metacomputing Applications

Daniel Becker[1,2], Felix Wolf[1,2], Wolfgang Frings[1], Markus Geimer[1],
Brian J.N. Wylie[1], and Bernd Mohr[1]

[1]Forschungszentrum Jülich
John von Neumann Institute for Computing (NIC)
52425 Jülich, Germany

[2]RWTH Aachen University
Department of Computer Science
52056 Aachen, Germany

{d.becker, f.wolf, w.frings, m.geimer, b.wylie, b.mohr}@fz-juelich.de

Abstract

The processing power and memory capacity of independent and heterogeneous parallel machines can be combined to form a single parallel system that is more powerful than any of its constituents. However, achieving satisfactory application performance on such a metacomputer is hard because the high latency of inter-machine communication as well as differences in hardware of constituent machines may introduce various types of wait states. In our earlier work, we have demonstrated that automatic pattern search in event traces can identify the sources of wait states in parallel applications running on a single computer. In this article, we describe how this approach can be extended to metacomputing environments with special emphasis on performance problems related to inter-machine communication. In addition, we demonstrate the benefits of our solution using a real-world multi-physics application.

Keywords: performance tools, grid computing, metacomputing, event tracing

1 Introduction

The solution of compute-intensive problems often requires more processing power than is available on a single parallel system, because the problem cannot be solved within a reasonable time frame on a single machine or because the solution must be calculated under real-time conditions (e.g. weather forecast). For this reason, the processing power and memory capacity of multiple heterogeneous parallel machines can be combined to form a more powerful *metacomputer* [14] that appears to its users as a single transparent parallel machine. Apart from being a pure aggregation of computational power, a metacomputer can also provide a suitable platform for multi-physics simulations, where the different submodels may be optimized for different architectures.

Often, the metacomputer's constituent systems, which are called *metahosts*, are geographically dispersed and may even belong to different organizations. In this sense, a metacomputing environment can be regarded as a special type of computational grid. Due to their distributed nature, the predominant programming model for metacomputers is message passing, which may be combined with multithreading used within the metahosts.

However, although applications can benefit from the increased parallelism offered by a metacomputer, as supported by a recent study by Wong and Goscinski [19], achieving satisfactory application performance is difficult. Often, the network links connecting the different metahosts exhibit high latency. In general, applications have to deal with a hierarchy of varying latencies and bandwidths. Moreover, the heterogeneity of metahost hardware including the differences in internal networks complicate load balancing. Finally, most applications are not designed to distinguish between internal and external communication. Given the fact that performance optimization for a single machine is already a non-trivial task that requires substantial tool support, we argue that this is even more important for metacomputing environments.

In our earlier work [18], we have shown that automatic analysis of event traces is an effective method for identifying complex performance phenomena in parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and searched afterward (i.e., post mortem) for patterns of inefficient behavior. The detected pattern instances are classified by the

1-4244-0910-1/07/$20.00

type of behavior and quantified by their significance for the overall performance. In this article, we describe how this approach can be extended to metacomputing environments consisting of multiple independent parallel computers or clusters. Our extension serves two goals:

- Allow metacomputing applications to take advantage of automatic trace analysis in its present form.
- Formulate new patterns related to metacomputing-specific performance problems, such as wait states during inter-metahost communication.

The first goal requires solutions to the problems of establishing a global view of trace data in the absence of a global file system and synchronizing time stamps across a hierarchy of network links with different latencies. Both goals require a mechanism to identify the metahost a process is running on.

The remainder of this article is organized as follows: After reviewing the state-of-the-art in metacomputing and discussing related work in Section 2, we introduce pattern search in event traces as our performance-analysis method of choice and explain the required infrastructure in Section 3. The extensions applied to use this infrastructure in a metacomputing environment are described in Section 4 along with new metacomputing-specific performance problems supported by this extension. In Section 5, we present experimental results based on a real-world multi-physics application in the metacomputing testbed used for our study. Finally, we conclude the paper and outline future work in Section 6.

2 Related Work

Metacomputing is an active area of research. Several wide-area MPI libraries exist including MPICH-G2, PACX-MPI, and MPICH/Madeleine that permit the execution of MPI applications on a metacomputer. For our experiments, we have chosen the MPI implementation MetaMPICH [3] developed at RWTH Aachen University, but it is worth noting that our approach is independent of a particular MPI implementation. Before running metacomputing applications on a computational grid, several computing resources including the required network links need to be allocated. Bierbaum et al. [2] describe a UNICORE-based infrastructure supporting the co-allocation of the resources making up a metacomputer, with special emphasis on the intricate task of coordinating network allocation with application startup.

Gerndt et al. [9] review a number of grid-performance monitoring and evaluation tools among which the following are most relevant to our work: GRM/PROVE [13] are trace-based application performance and monitoring tools for message passing programs. GRM delivers trace data to PROVE, which visualizes trace information on-line during execution of the grid application. The tools target applications running on one grid resource, however, trace collection from several resources (e.g., metacomputing) is also possible. Moreover, VAMPIR, a popular graphical trace browser with a zoomable time-line display that allows the fine-grained investigation of parallel performance behavior, has been extended to support multi-site grid applications [5]. The extension includes a group concept allowing the distinction between different metahosts during an analysis session. VAMPIR has also been successfully integrated with the UNICORE grid middleware [10], allowing the user to create a task with appropriate instrumentation and to retrieve the generated trace files for local visualization after program termination. Another grid-enabled performance monitoring and analysis system is SCALEA-G [16]. It provides an OGSA-based infrastructure for conducting on-line monitoring and performance analysis. Both push and pull models are supported. Source code and dynamic instrumentation are exploited to perform profiling and tracing of grid applications. Finally, Badia et al. [1] report how they have used the prediction tool DIMEMAS to predict the performance on a metacomputer based on execution traces from a single machine in combination with measured network parameters that characterize the communication between different machines.

3 Automatic Trace Analysis

Event tracing is a powerful method for analyzing the performance behavior of parallel applications. For example, graphical trace browsers, such as VAMPIR [12] and Paraver [11], allow the fine-grained investigation of parallel performance behavior and provide statistical summaries. However, in view of the large amounts of data generated on contemporary parallel machines, the depth and coverage of the visual analysis offered by a browser is limited as soon as it targets more complex patterns not included in the statistics generated by such tools.

Trace analysis. By contrast, the KOJAK toolset [18] automatically searches global event traces of parallel programs for patterns of inefficient behavior, classifies detected instances by category, and quantifies the associated performance penalty. This allows developers to study the performance of their applications on a higher level of abstraction, while requiring significantly less time and expertise than a manual analysis. For a detailed description of the pattern analysis including underlying abstraction mechanisms the interested reader may refer to Wolf [17].

To perform the pattern search in a parallel way, KOJAK's successor project SCALASCA exploits both distributed memory and parallel processing capabilities avail-

able on the metacomputer itself. Instead of sequentially analyzing a single global trace file, SCALASCA analyzes separate local trace files in parallel by *replaying* the original communication on the same hardware configuration and the same number of CPUs as the one that has been used to execute the target application. That is we avoid merging local trace files, and, thus, copying large amounts of trace data across the network. A more detailed description of the parallel trace analysis, which was originally introduced to be used on large-scale systems, such as IBM BlueGene/L, can be found in [8].

Trace file organization. To simplify the management of trace files (local and global ones) and analysis reports, all files related to a single experiment are stored in the same archive directory. Although this feature is not essential to perform the intended analysis, compatibility with respect to the development of utilities operating on experiment data has motivated the decision to retain the archive directory in the metacomputing-enabled version.

Event location. The location of an event is specified as a tuple consisting of the following four elements: machine, node, process, and thread. The machine represents a parallel computer or cluster, of which there is only one unless the application runs in a metacomputing environment, as described in the next section. A node is a substructure of a machine and typically corresponds to an SMP node.

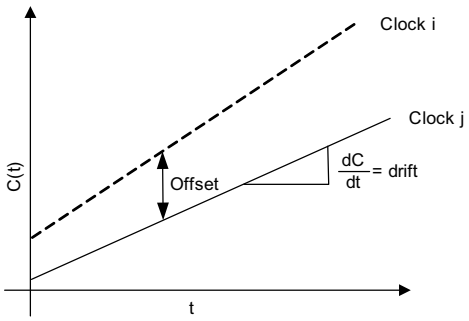


Figure 1. Clocks with both initial offset and different constant drifts.

Synchronization of time stamps. Unfortunately, not all parallel computers provide hardware clock synchronization among different nodes. Instead, their node-local clocks may vary in offset and drift (Figure 1). KOJAK as well as SCALASCA address this problem with software synchronization of time stamps [17] correct the precedence order of distributed events, and, in particular, the causal order of communication events that is known as the *clock condition*.

For this purpose, we perform offset measurements between one master node (without loss of generality the node hosting the process with rank zero) and all the remaining (slave) nodes. We assume that time stamps taken on the same node are already synchronized. The measurements, which are carried out according to the remote clock reading technique [6], are taken at program start and repeated at program end.

Under the assumption, that all clocks have a constant drift and can be described in terms of a linear function, based on an initial offset and a constant slope, it is possible to perform a linear interpolation and calculate the master time m as a function of the slave time s.

4 Trace Analysis on a Metacomputer

A metacomputer consists of several independent and potentially heterogeneous parallel systems (metahosts), which are connected by network links to a single unit. The metahosts' internal network is usually based on a fast interconnect, such as SCI, Myrinet, Infiniband, or a proprietary network. Metahosts belonging to the same organization are typically connected via a local area network. Distant metahosts, which often belong to different organizations, are typically linked by a wide-area interconnection. Figure 2 shows the schematic view of a metacomputer including its external and internal networks.

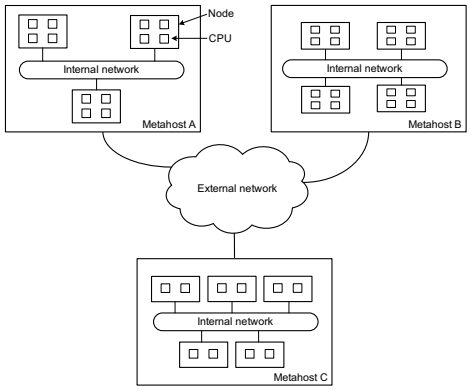


Figure 2. Schematic view of a metacomputer including its external and internal networks.

Our motivation to transfer the methodology described in the previous section to metacomputing environments is twofold: Our first goal was to allow metacomputing applications to take advantage of this performance analysis approach, which implies to simply make it work on a metacomputer. Our second goal was to support the analysis of metacomputing-specific performance problems. These two goals lead to the following requirements:

- Facilitate automatic trace analysis in the absence of a file system shared by all processes. The trace file of a process can only be written to a file system the process has access to. In a metacomputing environment, such as the one used for our experiments, metahosts may be owned by different organizations; therefore, the existence of a shared file system cannot be assumed. The previous approach depends on a global file system because merging local trace files is performed inside the same archive directory. Copying potentially large trace files across the network is in principle possible, but introduces undesired overhead especially if the application was executed on a large number of processors. Also, in the absence of a global file system, the aforementioned archive directory would not be visible by every process, which would result in erroneous behavior.
- Adapt the mechanism for the synchronization of time stamps such that it can cope with a hierarchy of latencies found in metacomputer interconnects. The previous approach is inaccurate because of the network links connecting different metahosts, whose latencies may be an order of magnitude larger than those of the internal networks. As a consequence, offset measurements across these links are less accurate in absolute terms than those across the internal networks, which our measurements in Section 5 confirm. When processes living on different nodes of the same metahost measure their offset relative to a master process living on another metahost, they might be well-synchronized relative to the master because the accuracy of the offset is sufficient in relation to the message latency of the external network. However, the offset relative to each other, which is calculated by subtracting their offsets relative to the master, might be inaccurate at an unacceptable scale when compared to the latency of the internal network between them.
- Formulate patterns that refer to metacomputing-specific performance problems, such as load balancing. This requires the ability to identify the metahost a process is living on and to distinguish between different metahosts during analysis. Later we will see that this subrequirement is also a prerequisite for an improved synchronization of time stamps.

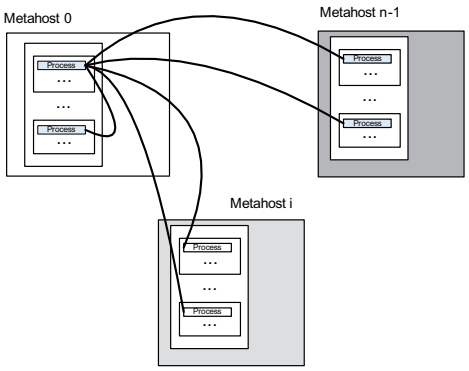
The first two requirements address goal one, while the third requirement addresses both goals one and two. In the remainder of this section, we describe the pieces needed to create a metacomputing-enabled trace-analysis infrastructure that satisfies the requirements listed above.

Metahost identification. The ability to identify the metahost a process is running on is required for both the improved time synchronization and the formulation of metacomputing-specific patterns. To correctly recognize the metahost at runtime, the user has to set two environment variables on each metahost that specify an unique numeric identifier as well as a human-readable metahost name. The numeric identifier is used during all internal operations of trace generation and analysis, whereas the human-readable name is used for the presentation of analysis results (see Figure 6, tree in the right panel).

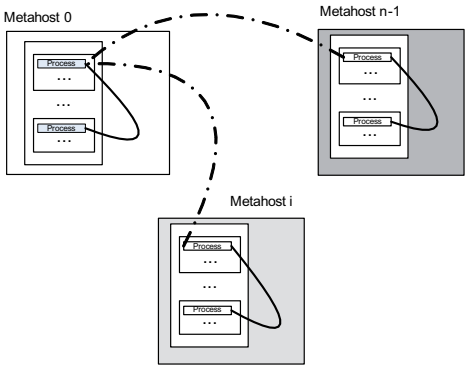
Hierarchical synchronization of time stamps. The previous synchronization of time stamps followed a centralized approach that is based on the transitivity of offset relations. All slave processes measure their offset relative to the master and it is assumed that the offsets relative to each other can be derived from their master offsets (Figure 3 (a)). The error of the offset measurement between two processes at a given moment (derived or measured) should be smaller than the message latency between them to ensure the clock condition. As explained above, this requirement may be violated if the offset between processes connected by a low-latency link is derived from offsets between processes connected by a high-latency link because it is assumed that the error of offset measurements grows with the latency.

The previous offset measurement is *flat* in that all slaves measure their offset by contacting the master directly without taking the hierarchy of network latencies between them into account. In contrast, our new scheme follows a *hierarchical* approach (Figure 3 (b)): Using the metahost identification mechanism, each metahost determines a local master. After that, one metamaster is chosen among all local masters. Now all local masters measure their offset relative to the metamaster. After this has been done, all slave processes exchange ping-pongs with their local master to determine the offset relative to the local master. In the case that a metahost already provides a global clock, this second step is omitted. Finally, the offset to the metamaster is calculated by adding the two measured offset values. Since all slaves within the same metahost now use the same inter-metahost offset measurement, their relative offset remains unaffected. An experimental validation of the new approach is presented in Section 5.

Runtime archive management. In the previous single-machine version, all local trace files are written to the same archive directory, which therefore must be visible from all processes. The archive directory is a container simplifying the management of all files related to a single experiment. Even if the advantage of a single archive cannot be trivially retained in the absence of a shared file system, correct operation requires that every process has access to an archive directory to which the trace data can be written and where they can be accessed later during the analysis.



(a) Flat synchronization of time stamps with offset measurements between all slave processes and the same master process.



(b) Hierarchical synchronization of time stamps with offset measurements between all local slave processes and their local master.

Figure 3. Comparison of the previous flat synchronization approach and the new hierarchical synchronization.

To guarantee the existence of an archive directory on each metahost, we apply the following hierarchical scheme again utilizing the metahost identification mechanism: First, rank zero attempts to create a single archive directory and broadcasts the outcome to all other processes that only continue if the creation was successful. Then, similar to the hierarchical synchronization, each metahost appoints a local master process that checks whether it can see the directory. If there is no directory because it resides on a different file system, the local master creates another one. Finally, all processes check whether they can see an archive directory. The results are exchanged between all processes using an all-reduce operation. If all processes can see a directory, the measurement is resumed, otherwise the application is aborted. This procedure offers a high degree of scalability because it avoids a larger number of simultaneous attempts to create the same directory.

Parallel trace analysis. The major advantage of SCALASCA's parallel trace analysis in metacomputing environments is that each analysis process needs only access to the corresponding local trace file. Because of our initial assumption that we use the same hardware configuration for both the target application and the trace analysis, each analysis process will automatically have access to the trace data it needs. Note that the parallel analysis exchanges trace data across the network, but that the amount of data transferred per process is significantly smaller than the entire trace file belonging to that process.

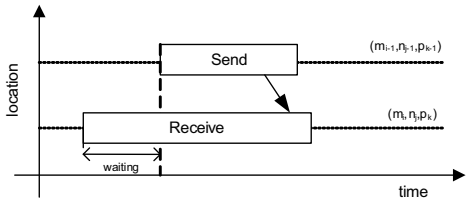
Metacomputing patterns. Our pattern analysis identifies wait states that occur when processes reach synchronization points at different moments. In MPI-1 applications, such synchronization points can either have the form of synchronous message exchanges between two processes in point-to-point mode or of synchronous group communications in collective mode. A full description of the single-machine patterns for MPI-1 has been given by Wolf and Mohr [18].

When developing efficient MPI applications for metacomputers, a major difficulty arises from load balancing. As illustrated in Figure 2, the metahosts making up a metacomputer may differ in the number of nodes, in the number of CPUs per node, in the type of CPU and operating system, and in the characteristics of their internal networks. In addition, the external network connecting them may suffer from high latency and, if it is not a dedicated network link, from interference with unrelated traffic. Since load imbalance often manifests itself as processes arriving untimely at synchronization points, the general concept behind our pattern analysis is well suited to guide application developers in recognizing problems of this kind.

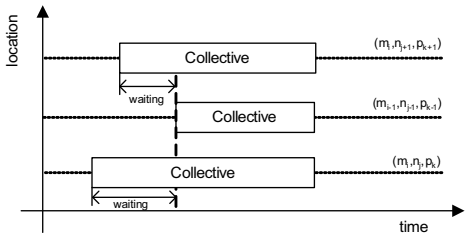
To distinguish pattern instances that result from processes on different metahosts waiting for each other, we have created special "grid" versions of most of the already existing patterns. In the case of point-to-point communication, the analysis recognizes whether sender and receiver reside on different metahosts. In the case of collective communication, the entire communicator is searched for processes differing in their machine (i.e., metahost) location component. Below, we discuss two representative examples, the *Late Sender* and the *Wait at N × N* pattern.

A point-to-point message can only be received after it has been sent. Late Sender (Figure 4 (a)) refers to the situation, in which a process is waiting in a blocking receive operation (e.g, `MPI_Recv() or MPI_Wait()`) that is posted earlier than the corresponding send operation.

A related phenomenon can be observed during certain types of collective communication. Collective operations that send data from n processes to n processes (e.g., `MPI_Allreduce()`) exhibit an inherent synchroniza-



(a) Late Sender pattern.



(b) Wait at N *times* N pattern.

Figure 4. Exemplary point-to-point and collective metacomputing patterns.

tion among all participants, that is, no process can finish the operation until the last process has started it. Wait at N×N (Figure 4 (b)) covers the time spent in *n*-to-*n* operations until all processes have reached it. Waiting at an MPI barrier can be considered a variant of this pattern and is called *Wait at Barrier*.

The grid versions of these patterns (*Grid Late Sender* and *Grid Wait at N × N*) simply check whether communication across different metahosts has taken place. Our graphical browser organizes the grid patterns in a hierarchy, which allows a convenient in-depth study of the performance behavior at varying levels of granularity. The hierarchy mirrors the hierarchy used for the non-grid versions of our patterns. In the following section, we show that these patterns can indeed provide valuable information on the performance behavior of metacomputing applications.

5 Experimental Results

In this section, we present experimental results that prove the feasibility of our approach and demonstrate the additional value created for the performance analysis of metacomputing applications. We start with a description of the metacomputing testbed used for our experiments, followed by an experimental validation of the hierarchical time-stamp synchronization scheme. Finally, we discuss metacomputing-specific performance problems of a real-world multi-physics application that have been identified using our tool set.

The VIOLA metacomputer. VIOLA [4] is a project funded by the German Ministry for Education and Research, which provides a testbed for advanced optical network technology. A major focus is the enhancement and test of new advanced grid applications. The network topology of the testbed section used in our study is illustrated in Figure 5.

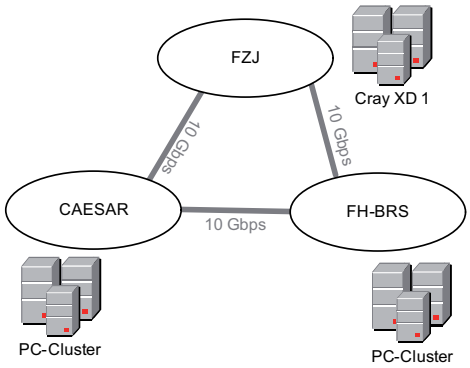


Figure 5. Network topology of the VIOLA testbed section used for our experiments.

The section comprises three sites, the Center of Advanced European Studies and Research (CAESAR), FH Bonn-Rhein-Sieg Stankt Augustin (FH-BRS), and Forschungszentrum Jülich (FZJ), which are connected via high-speed optical links offering a bandwidth of 10 Gbps between each pair of sites. The three sites lie between 20 and 100 km apart. The metacomputer used for our purposes includes three metahosts, one at each site:

- A PC Linux cluster with 32 2-way Intel Xeon SMP nodes at 2.6 GHz with a Gigabit Ethernet interconnect located at CAESAR.
- A PC Linux cluster with 6 4-way AMD Opteron SMP nodes at 2 GHz with a usock over Myrinet interconnect located at FH-BRS.
- A Cray XD1 Linux cluster with 60 2-way AMD Opteron SMP nodes at 2.2 GHz with a usock over RapidArray interconnect located at FZJ.

The three metahosts provide a separate Gigabit Ethernet network interface adapter for each node so that a direct connection to the external network can be established from each process. MetaMPICH, the MPICH-based MPI-implementation chosen for this testbed, supports this direct external connection through its multi-device architecture [3] that allows communication between processes across the external network without the involvement of dedicated router processes that would be needed otherwise.

Table 1. Latencies of the internal and external networks in VIOLA.

	mean [μs]	std. deviation [μs]
FZJ - FH-BRS (external network)	9.88E+02	3.86E+00
FZJ (internal network)	2.15E+01	8.14E-01
FH-BRS (internal network)	4.44E+01	3.60E-01

Synchronization of time stamps. In our configuration, the latency of the external network exceeds the latency of the internal network by two orders of magnitude. Also, the latencies of the internal networks differ significantly. Table 1 shows the latencies measured using MetaMPICH. The standard deviation is an indicator for the precision of offset measurements across these links, which confirms our assumption that offset measurements across the external network are much less accurate than those across the internal networks.

The accuracy of the hierarchical synchronization scheme was verified using a benchmark that has been specifically designed to exchange a large number of short messages between varying pairs of processes. This way, the benchmark produces pairs of send and receive events that are chronologically close to each other. Also, the parallel analyzer has been extended to report violations of the clock condition. Table 2 shows the number of violations found in traces from this benchmark for synchronization based on (i) a single flat offset measurement without compensation for drift, (ii) two flat offset measurements and subsequent linear interpolation (our previous method), and (iii) two hierarchical offset measurements and subsequent linear interpolation. As one can see in this experiment, the hierarchical scheme was able to significantly reduce the number of clock condition violations.

Table 2. Number of clock condition violations recognized by the parallel analyzer.

Measurement	clock condition violations
single flat offset	7560
two flat offsets	2179
two hierarchical offsets	0

Analysis of a metacomputing application. To demonstrate that automatic trace analysis can help identify typical performance problems related to metacomputing, we analyzed the performance of a multi-physics application called MetaTrace [7], which simulates solute transport in heterogeneous soil-aquifer systems to study the spread of groundwater pollutants. MetaTrace consists of two submodels: Trace calculates the velocity field of water flow in variably saturated media, whereas Partrace calculates the solute transport in a given velocity field (i.e., the one provided by Trace). Trace applies a three-dimensional domain decomposition with nearest-neighbor communication. The algorithm is based on a parallel version of the conjugate gradient (CG) method. In contrast, Partrace tracks individual particles. Every 10-15 seconds, Trace sends the velocity field in a chunk of 200 MB in parallel to Partrace for further processing. In addition, Partrace sends steering information to Trace. The entire simulation is provided as a single executable that integrates the two submodels written in FORTRAN and C++ using a wrapper written in C. It is worth noting that our toolset can, in principle, also handle separate executables. The application was instrumented by inserting directives which were automatically translated into tracing API calls by a preprocessor.

Table 3. Detailed configurations of the three-metahost and one-metahost experiments.

	Experiment 1	Experiment 2
Partrace	XD1: 8 nodes 2 processes/node	IBM AIX POWER: 1 node 16 processes/node
Trace	FH-BRS: 2 nodes 4 processes/node CAESAR: 4 nodes 2 processes/node	IBM AIX POWER: 1 node 16 processes/node

To facilitate a comparison between a heterogeneous and a homogeneous cluster, we performed two experiments, one using three metahosts and one using a single metahost. The total number of processes was 32 in both cases. The detailed configurations of these experiments are listed in Table 3. Note that we assigned the same number of processors to Trace and Partrace.

Parallel trace analysis. Figure 6 shows screen shots of trace analysis results for the three-metahost case. In each subfigure, the tree in the left panel displays the grid-specific patterns discussed earlier, which are arranged in a specialization hierarchy. The numbers left of the pattern names indicate the total execution time penalty in percent. In addition, the color of the little square provides a visual clue of

the percentage to quickly guide the user to the most severe performance problems. The middle tree shows the distribution of the selected pattern across the call tree. Finally, the right tree shows the distribution of the selected pattern at the selected call path across the hierarchy of metahosts, nodes, and processes.

Apparently, the application suffers from both *Late Sender* and *Wait at Barrier*, when communicating or synchronizing across metahost boundaries. As the displays indicate, the grid-specific *Late Sender* version (Figure 6 (a)) and the grid-specific *Wait at Barrier* version (Figure 6 (b)) consume 9.3 % and 23.1 % of the overall execution time, respectively.

In the three-metahost case, Partrace ran on the XD1, while Trace was distributed across the FH-BRS cluster and the CAESAR cluster. Functions in Trace that did not call any MPI routines (e.g. `finelassdt()`) were executed about two times faster on the FH-BRS cluster than they were on the CAESAR cluster, although the algorithm assigns the same portion of work to every process in Trace. A major fraction of the *Late Sender* pattern is concentrated in `cgiteration()`, a function communicating only inside Trace according to a nearest neighbor scheme with most of the waiting time occurring on the faster FH-BRS cluster. Given that both FH-BRS and CAESAR supply the same number of CPUs to process the same amount of work, the result suggests that the speed differences mentioned above can be held responsible.

Moreover, most of the *Wait at Barrier* problem occurs inside the Partrace function `ReadVelFieldFromTrace()`, a function that only does synchronous communication with Trace. More precisely, Trace waits at the barrier in function `printtolink()` until all processes in Partrace reach the corresponding barrier in function `ReadVelFieldFromTrace()`, before Trace unidirectionally sends the velocity field to Partrace for further processing. The bigger share of the barrier waiting time in this experiment could be attributed to Partrace, which indicates another imbalance, this time between the two component models. However, only based on this analysis result it is difficult to judge whether the imbalance is caused by the heterogeneity of the cluster (including varying network characteristics) or by the application itself (e.g., algorithm or input data).

It can be observed that running the application on the homogeneous cluster IBM AIX POWER (Figure 7) leads to a significant decrease of the barrier waiting time inside the Partrace function `ReadVelFieldFromTrace()` and also of the waiting time in the receive operation inside `cgiteration()` mentioned above. In spite of an overall performance improvement, a *Late Sender* problem occurring inside another function that involves communication of currently unused steering information from Partrace back to Trace experienced a significant increase, indicating that now Trace mostly waits for Partrace. This large impact of the heterogeneous hardware configuration on the performance behavior is typical for metacomputing applications. In the case of MetaTrace, a dynamic load balancing scheme might be advisable.

6 Conclusion

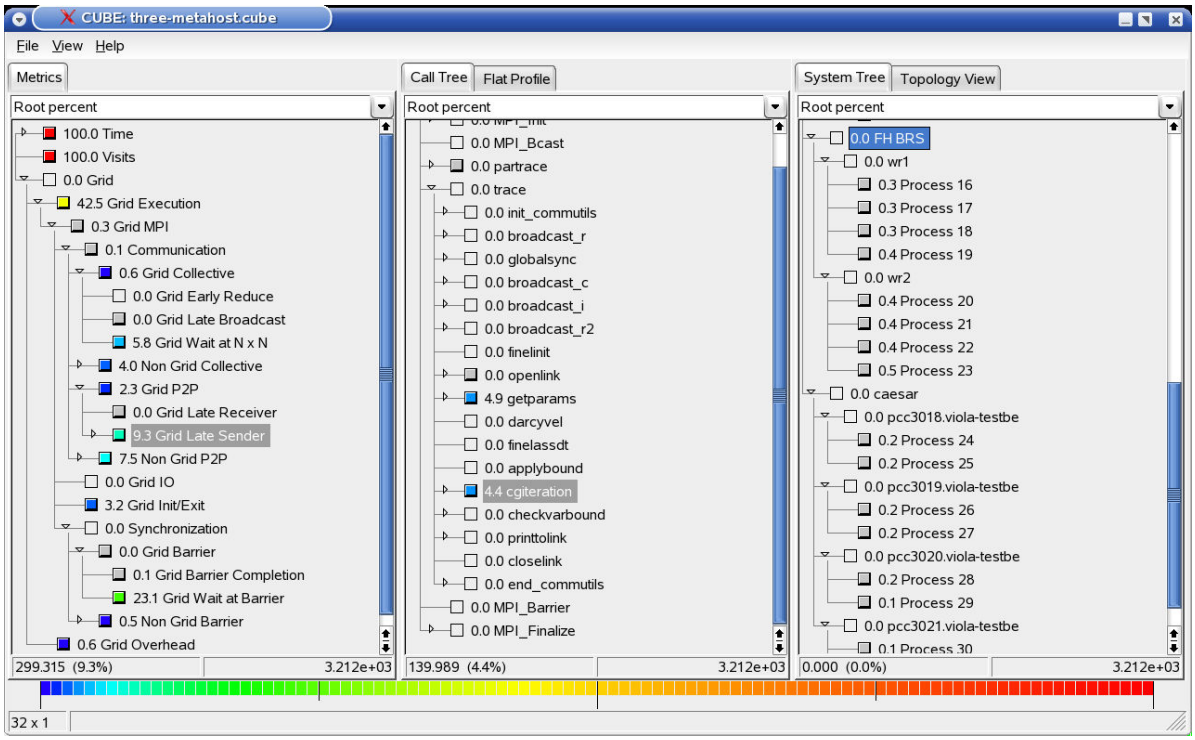
The contribution of this work is twofold. First, we have made the performance analysis technique of automatic pattern search in event traces available to metacomputing environments. For this purpose, we have developed an extension to the SCALASCA toolkit consisting of the following components:

1. Extended runtime configuration management capable of identifying the metahost an application process is running on, a prerequisite needed for the remaining items.
2. A hierarchical mechanism for synchronizing event timings that is able to deal with substantially different latencies in external and internal networks.
3. Extended trace archive management permitting the creation of multiple partial trace archives, which is required in the absence of a shared file system between metahosts.

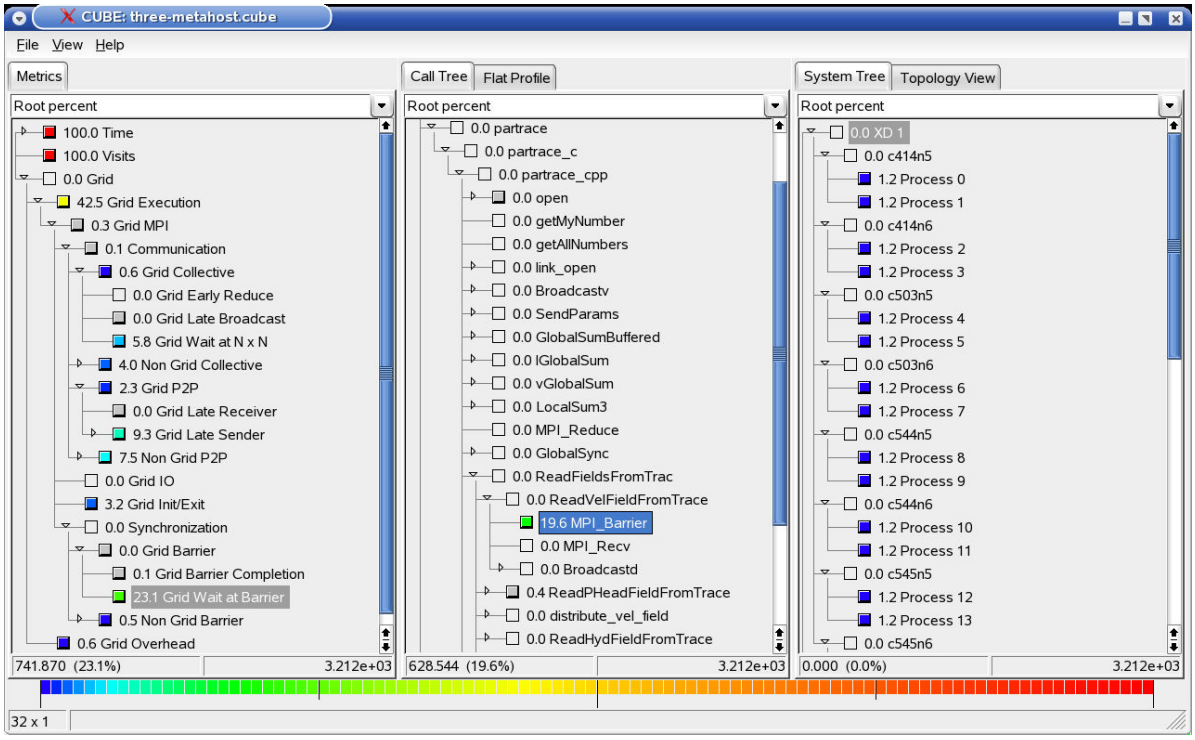
In addition, we have shown that the parallel trace algorithm used in SCALASCA is not only more scalable, but also avoids costly copying of trace data between metahosts in the absence of a shared file system.

Second, we have defined metacomputing-specific patterns that target wait states occurring during communication between different metahosts. As our experiments have shown, these wait states provide useful hints pointing the application developer to potential sources of load imbalance. However, from a single experiment it is difficult to judge whether the load imbalance is caused by the heterogeneity of the cluster (including varying network characteristics) or by the application itself (e.g., algorithm or input data). The value of our trace analysis is increased by the comparison with measurements on a homogeneous cluster.

This type of comparative analysis could be effectively supported by the algebra utilities developed by Song et al. [15], which we plan to make available in a version compatible to the parallel analyzer. Also, it is worth mentioning that our analysis does not yet use all the information available in our trace data. For example, the current grid patterns only distinguish between internal and external communication without differentiating between different combinations of metahosts. Here, a more fine-grained classification would be desirable.



(a) Analysis results for the three-metahost case: Grid Late Sender problem inside the Trace function `cgiteration()` distributed across the FH-BRS and the CAESAR clusters.



(b) Analysis results for the three-metahost case: Grid Wait at Barrier problem inside the Partrace function `ReadVelFieldFromTrace()` on the Cray XD1 at FZJ.

Figure 6. Analysis results of the multi-physics application MetaTrace showing the metric hierarchy, the call tree, and the hierarchy of system resources.

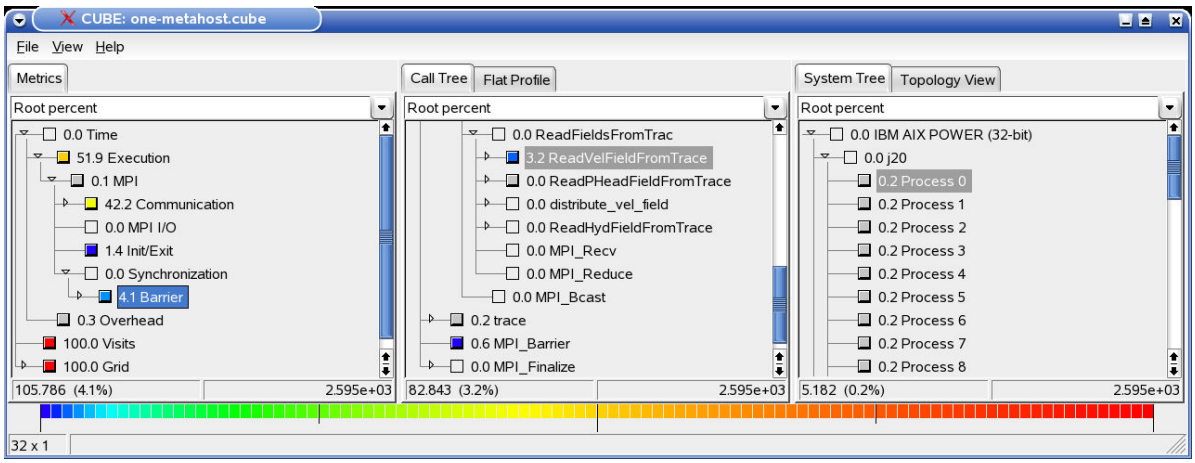


Figure 7. Analysis results for the one-metahost case: Wait at Barrier problem inside the Partrace function `ReadVelFieldFromTrace()` on the IBM AIX Power cluster at FZJ.

References

[1] R. M. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and M. Müller. Performance prediction in a grid environment. In *Proc. of the 1st European Across Grid Conference*, Santiago de Compostella, Spain, February 2003.

[2] B. Bierbaum, C. Clauss, T. Eickermann, L. Kirtchakova, A. Krechel, S. Springstubbe, O. Wäldrich, and W. Ziegler. Orchestration of distributed MPI-applications in a UNICORE-based grid with metampich and metascheduling. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.

[3] B. Bierbaum, C. Clauss, M. Pöppe, S. Lankes, and T. Bemmerl. The new multidevice architecture of MetaMPICH in the context of other approaches to grid-enabled MPI. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.

[4] BMBF (Ministry for Education and Research). *Vertically Integrated Optical Testbed for Large Applications in DFN (VIOLA)*. `http://www.viola-testbed.de/`.

[5] H. Brunst, E. Gabriel, M. Lange, M. S. Müller, W. E. Nagel, and M. M. Resch. Performance analysis of a parallel application in the grid. In *Proc. of the International Conference on Computational Science (ICCS)*. Springer, 2003.

[6] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1998. Springer Verlag.

[7] Forschungszentrum Jülich. *Solute Transport in Heterogeneous Soil-Aquifer Systems*. `http://www.fz-juelich.de/icg/icg-iv/modeling`.

[8] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.

[9] M. Gerndt, R. Wismüller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. Performance tools for the grid: State of the art and future. Technical report, APART White Paper, 2004. `http://www.lpds.sztaki.hu/~zsnemeth/apart/repository/gridtools.pdf`.

[10] S. Haubold, H. Mix, W. E. Nagel, and M. Romberg. The UNICORE grid and its options for performance analysis. pages 275–288, 2004.

[11] J. Labarta, S. Girona, V. .Pillet, T. Cortes, and L. Gregoris. DiP : A parallel program development environment. In *Proc. of the 2th International Euro-Par Conference*, Lyon, France, August 1996. Springer.

[12] W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[13] N. Podhorszki and P. Kacsuk. Presentation and analysis of grid performance data. In *Proc. of the 9th International Euro-Par Conference*, Klagenfurt, Austria, 2003. Springer.

[14] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6), June 1992.

[15] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004. IEEE Computer Society.

[16] H.-L. Truong and T. Fahringer. SCALEA-G: A unified monitoring and performance analysis system for the grid. *Scientific Programming*, 12(4):225–237, 2004. IOS Press.

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

[18] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, Nov. 2003.

[19] A. Wong and A. Goscinski. Using an enterprise grid for execution of MPI parallel applications - a case study. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.