Scalable Parallel Trace-Based Performance Analysis

Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr

John von Neumann Institute for Computing (NIC)
Forschungszentrum Jülich, 52425 Jülich, Germany
{m.geimer, f.wolf, b.wylie, b.mohr}@fz-juelich.de

Abstract. Automatic trace analysis is an effective method for identifying complex performance phenomena in parallel applications. However, as the size of parallel systems and the number of processors used by individual applications is continuously raised, the traditional approach of analyzing a single global trace file, as done by KOJAK's EXPERT trace analyzer, becomes increasingly constrained by the large number of events. In this article, we present a scalable version of the EXPERT analysis based on analyzing separate local trace files with a parallel tool which 'replays' the target application's communication behavior. We describe the new parallel analyzer architecture and discuss first empirical results.

1 Introduction

Event tracing is a well-accepted technique for post-mortem performance analysis of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and analyzed afterwards with the help of software tools. For example, graphical trace browsers like VAMPIR [1] and PARAVER [2], allow fine-grained investigation of execution behavior using a zoomable time-line display.

However, in view of the large amounts of data usually generated, automatic off-line trace analyzers, such as the EXPERT tool from the KOJAK toolset [3,4], can provide relevant information more quickly by automatically searching traces for complex patterns of inefficient behavior and quantifying their significance. In addition to usually being faster than a manual analysis performed using trace browsers, this approach is also guaranteed to cover the entire event trace and not to miss any pattern instances.

Unfortunately, sequentially analyzing a single trace file does not scale to applications running on thousands of processors. Even if access locality is exploited, the amount of main memory might not be sufficient to store the current working set of events. Moreover, the amount of trace data might not even fit into a single file, which already suggests to perform the analysis in a more distributed fashion.

In this paper, we describe how the pattern search can be done in a more scalable way by exploiting both distributed memory and parallel processing capabilities available on modern large-scale systems. Instead of sequentially analyzing a single global trace file, we analyze separate local trace files in parallel by *replaying* the original communication on as many CPUs as have been used to execute the target application itself.

We start our discussion with a review of related work in Section 2, followed by an overview of our trace analyzer's new parallel design in Section 3, where it is also compared to the previous sequential design. Then, in Section 4, we discuss the parallel pattern-analysis mechanism in more detail, before we show preliminary experimental results that already demonstrate the improvement over the sequential analysis in Section 5. Finally, in Section 6 we conclude the paper and outline further improvements.

2 Related Work

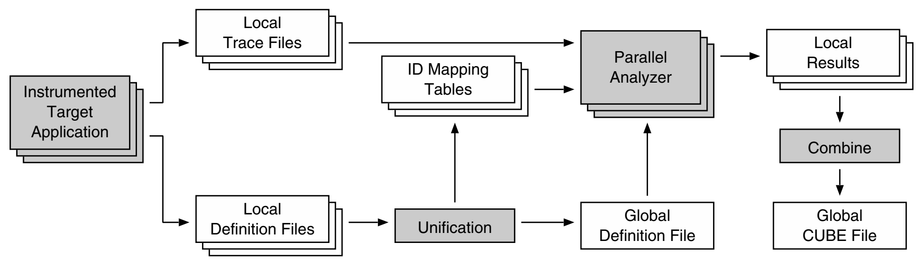
Wolf et al. [5] review a number of approaches addressing scalable trace analysis. Dynamic periodicity detection in OpenMP applications [6] avoids recording redundant performance behavior, while the frame-based SLOG trace-data format [7] supports scalable visualization. Important to our particular approach has been the distributed trace analysis and visualization tool VAMPIR Server[8], which provides parallel trace access mechanisms, albeit targeting a 'serial' human client in front of a graphical trace browser as opposed to fully automatic and parallel trace analysis. A tree-based main memory data structure for event traces called cCCG [9] allows potentially lossy compression of trace data while observing specified deviation bounds.

Non-trace-based on-line performance tools, such as Paradyn [10] or Periscope [11], that analyze performance data in real-time address scalability by employing hierarchical networks for efficient reduction and broadcast operations between back-end processes and the tool front-end. The particular way patterns are specified and implemented in EXPERT was stimulated by the APART Specification Language (ASL) [12], which provides a formal notation to describe performance properties of parallel applications. Other ASL-inspired work includes JavaPSL [13], a Java version of ASL, and the aforementioned Periscope tool. KappaPI 2 [14] sequentially searches trace files of message-passing applications for patterns very similar to those used in our approach, but in KappaPI 2 emphasis is put on generating recommendations on how to improve the performance using knowledge of bottleneck use cases.

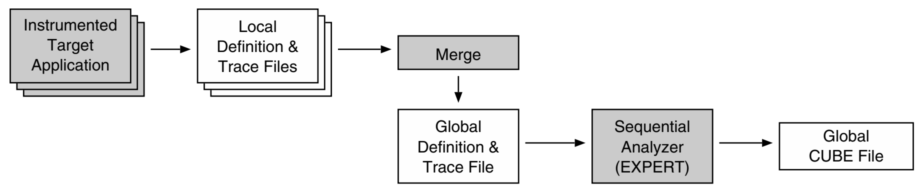
3 Overview of Parallel Trace Analysis

Instead of sequentially analyzing a single and potentially large global trace file, we analyze multiple local trace files in parallel based on the same parallel programming paradigm as the one used by the target application. For the sake of simplicity, we currently have restricted ourselves to handle only single-threaded MPI-1 applications, which implies that our parallel analyzer is an MPI-1-based program as well. The analyzer is executed on as many CPUs as have been allocated for the target application, allowing to run it within the same batch job as the application itself. Using an allocation with a different (smaller) number of CPUs for the analysis would require a separate batch job introducing typically significant additional waiting time in the performance analysis workflow. Figure 1 depicts the analysis workflow along with responsible components in comparison to the sequential analysis implemented by EXPERT.

The parallel analyzer itself uses a distributed memory approach, where each process reads only the trace data that was recorded for the corresponding process of the target application. This specifically addresses scalability with respect to wider traces, this is, those from larger numbers of processes. Since longer traces can be handled by selective tracing — i.e., by recording events only for code regions of particular interest — we



(a) New replay-based parallel approach.


Instrumented Target Application
Local Definition & Trace Files
Merge
Global Definition & Trace File
Sequential Analyzer (EXPERT)
Global CUBE File

(b) Sequential analysis using EXPERT.

Fig. 1. Schematic overview of the new parallel analysis work flow (a) in comparison to the previous sequential analysis (b). Stacked rectangles denote multiple instances of files or applications executed in parallel.

assume that the local trace data can be completely held in the main memory of the compute nodes. This has the advantage of having efficient random-access to individual events, whereas this is often not the case when dealing with a global trace file.

The actual analysis can then be accomplished by performing a *parallel replay* of the application's communication behavior. The central idea behind this replay-based analysis approach is to analyze a communication operation using an operation of the same type. For example, to analyze a point-to-point message, the event data necessary to analyze this communication is also exchanged in point-to-point mode between the corresponding analysis processes. To do this, the new analysis traverses local traces in parallel and meets at the synchronization points of the target application by replaying the original communication. How this idea can be used to search for complex patterns of inefficient behavior will be described in more detail in Section 4.

The event records stored in the individual per-process trace files use local identifiers to refer to static program entities, such as source-code regions or MPI communicators. Therefore, these local identifiers are mapped onto unique, global identifiers for the exchange of trace data between analysis processes. In the sequential analysis this mapping is part of the *Merge* step. In the parallel approach, this is similarly accomplished by performing a preprocessing step using a separate program that sequentially unifies the definitions of the per-process traces and generates a global definitions file that is shared between all analysis processes. To avoid reading the entire local trace files to extract definition records, we have modified the KOJAK measurement system to write definition and event records into separate files. The *Unification* step also creates a set of mapping tables that the analysis processes use to convert local into global identifiers while reading their local event data.

Each parallel analysis process only calculates a subset of the overall analysis report. Therefore, these local reports have to be combined into a single output file after the analysis has completed. In our current prototype, the individual analysis processes write their results to local files, which are then merged into a global CUBE output file [15] during a separate postprocessing *Combine* step.

These sequential pre- and postprocessing steps can be optimized in several ways, among which the most promising option is their integration into the analyzer and concomitant parallelization to minimize costly file I/O operations. However, detailed discussion of these optimizations is beyond the scope of this paper.

4 Message Passing Pattern Analysis

The replay-based analysis approach can be used to search for a large number of inefficiency patterns. Our current prototype supports the full range of MPI-1 performance metrics offered by the original sequential EXPERT tool, with the exception of *Late Receiver, Messages in Wrong Order* that is rarely significant in practice. A representative subset of these patterns is diagrammed in Figure 2. Their detection algorithms will be used to illustrate the parallel analysis mechanism below.

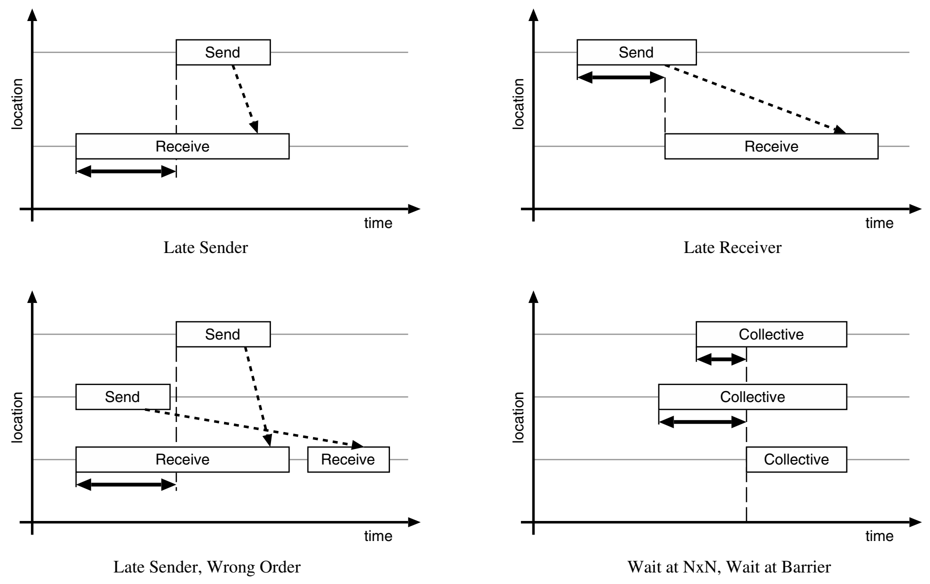


Fig. 2. Patterns of inefficient behavior

4.1 Point-to-Point Communication

As an example for inefficient point-to-point communication, we consider the so-called *Late Sender* pattern. Here, a receive operation is entered by one process before the corresponding send operation has been started by the other. The time lost due to this pattern is therefore the difference between the timestamps of the enter events of the MPI function instances which contain the corresponding message send and receive events. The complete Late Sender pattern consists of four events, specifically the two enter events and the respective message send and receive events.

During the parallel replay, the detection of this performance problem is triggered by the point-to-point communication events involved (i.e., send and receive). That is, when a send event is found by one of the processes, a message containing this event as well as the associated enter event is created. This message is then sent to the process representing the receiver using a point-to-point operation. To ensure the correct matching of send and receive events, we use equivalent tag and communicator information to perform the communication.

When the receiver reaches the receive event, the aforementioned message containing the *remote constituents* of the pattern is received. Together with the locally available constituents (i.e., the receive and the enter events), a Late Sender situation can be detected by comparing the timestamps of the two enter events and calculating the time spent waiting for the sender. This approach relies on the availability of a synchronized clock: otherwise linear interpolation of timestamps [16] is used, but alternative methods of time correction are being considered.

The detection of the *Late Receiver* pattern is very similar and straightforward to implement. However, to avoid sending redundant messages while executing the detection algorithms for the different performance problems related to point-to-point communication, we exploit specialization relationships between patterns and reuse results obtained on higher levels of the hierarchy. This is implemented using a sophisticated event notification and call-back mechanism similar to the publish-and-subscribe approach presented in [4]. For this pattern the severity is calculated by the receiver but attributed to the sender's location. To avoid the additional overhead of transferring the calculated waiting time back to the sender, it is stored as a *remote result* at the receiving process.

By contrast, detecting the *Late Sender, Messages in Wrong Order* pattern is more difficult. This pattern describes the situation that during a Late Sender pattern, another message is waiting to be received by the same destination but which was sent earlier. To detect it, we would need a global view of the messages currently in transit while assessing the Late Sender situation, which is not available in a parallel implementation. Therefore, each analysis process keeps track of the last occurrences of the Late Sender pattern found in its local trace using a ring buffer. If a receive event is encountered during the replay, we compare the timestamps of the corresponding send event and those of the buffered Late Sender occurrences. If the Late Sender's send operation starts after the send event associated with the current receive, the Late Sender instance is classified as a Wrong Order situation and removed from the buffer. Note that this approach does not guarantee to find all occurrences of this pattern, although empirical results suggest that the coverage of our method is sufficient in practice.

4.2 Collective Communication and Synchronization Operations

The second important type of communication operations are MPI collective operations. As an example of a related performance problem, we discuss the detection of the *Wait* *at N×N* pattern, which quantifies the waiting time due to the inherent synchronization in N-to-N operations, such as `MPI_Allreduce`.

While traversing the local trace data, all processes involved in a collective operation will eventually reach their corresponding collective exit events. After verifying that it relates to an N-to-N operation, accomplished by examining the associated region identifier, the analyzer invokes the detection algorithm, which determines the latest of the corresponding enter events using an `MPI_Allreduce` operation. After that, each process calculates the local waiting time by subtracting the timestamp of the local enter event from the timestamp of the enter event obtained through the reduction operation. The group of ranks involved in the analysis of the collective operation is easily determined by re-using the communicator of the original collective operation.

Very similar algorithms can be used to implement patterns related to 1-to-N, N-to-1 and barrier operations. As with point-to-point operations, a single MPI call is used to calculate the asscociated waiting times. Only barrier operations, for which the analyzer also calculates asymmetries that occur when leaving the operation, require two calls.

5 Results

To evaluate the effectiveness of parallel analysis based on a replay of the target application's communication behavior, a number of experiments with our current prototype implementation have been performed at a range of scales and compared with the sequential EXPERT tool. To facilitate a fair comparison, a restricted version of EXPERT was used that provides only the functionality of our parallel prototype, i.e., support for MPI-2, OpenMP, and SHMEM pattern analysis was disabled.

Measurements were taken on the IBM BlueGene/L system at Forschungszentrum Jülich (JUBL), which consists of 8,192 dual-core 700 MHz PowerPC 440 compute nodes (each with 512 MBytes of memory), 288 I/O nodes, and p720 service and login nodes each with eight 1.6 GHz Power5 processors [17]. The system was running the V1R2 software release with GPFS parallel filesystem configured with 4 servers. A dedicated partition consisting of all of the compute nodes was used for the parallel analyses, whereas the sequential programs (pre- and postprocessing, and EXPERT) ran on the lightly-loaded login node. Two applications with quite different execution and performance characteristics have been selected for detailed comparison.

The ASC benchmark SMG2000 [18] is a parallel semi-coarsening multigrid solver, which uses a complex communication pattern. The MPI version performs a lot of non-nearest-neighbor point-to-point communication operations (and only a negligible number of collective communication operations) and can be considered to be a stress-test for the memory and network subsystems of a machine. To investigate *weak scaling* behavior, a fixed $64 \times 64 \times 32$ problem size per process with five solver iterations was configured, resulting in a nearly constant application run-time as additional CPUs were used. Because the number of events traced for each process increases with the total number of processes, the aggregate trace volume increases faster than linearly.

The second case, PEPC-B [19], uses a locally-developed parallel tree code for computing long-range forces in N-body particle systems applied in this case to beam-plasma interactions. With a fixed problem size consisting of one million charged particles updated for 10 steps, increasing the number of CPUs reduces overall run-time as a demonstration of *strong scaling* behavior. By contrast to the SMG2000 benchmark, it uses a significant proportion of collective communication and synchronization operations.

Figure 3 charts wall-clock execution times for the uninstrumented applications and their analysis with a range of process numbers on JUBL. The 8-fold doubling of process numbers necessitates a log–log scale to show the corresponding range of times, particularly for the old sequential analysis (which furthermore becomes impractical for the largest traces). The figure shows the total time needed for the parallel analysis including the aforementioned sequential steps, the time taken by the parallel analysis without sequential steps, and the time taken by the parallel replay itself without file I/O. Due to the often considerable variation in the time for file I/O (e.g., depending on overall filesystem load) the times reported are the best of several measurements.

While the set of execution traces from 1,024 PEPC-B processes only reached 400 MBytes aggregate size (56 million events in total), the corresponding execution traces from 1,024 SMG2000 processes were 10 GBytes (a total of 1,886 million events). The largest set of execution traces from 16,384 SMG2000 processes amounted to 230 GBytes (over 40,000 million events in total). Both applications have communication characteristics that result in individual process traces being considerably smaller or larger than the average.

File I/O can be seen to command increasing proportions of the analysis time, however, future versions of the parallel analysis will reduce this overhead by parallelizing

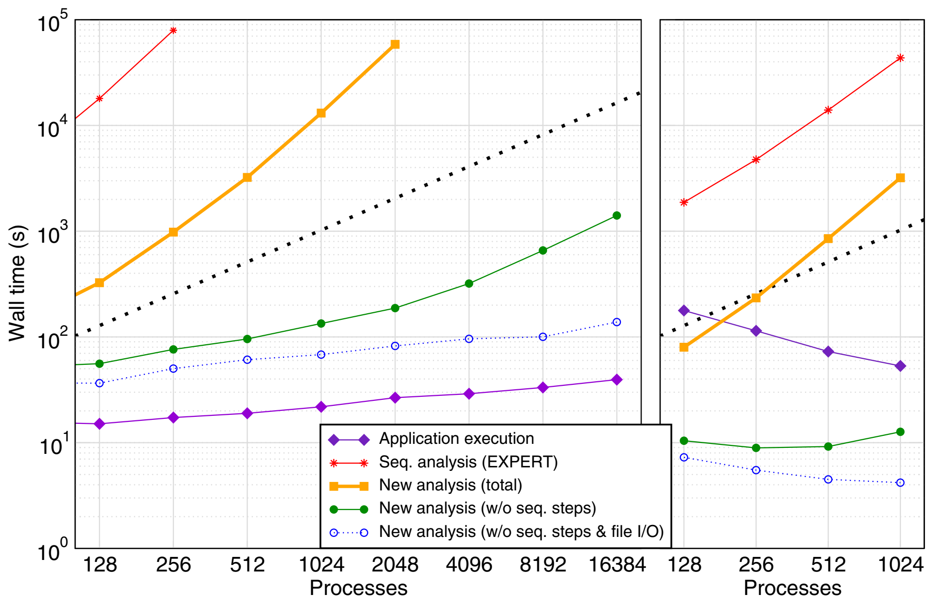
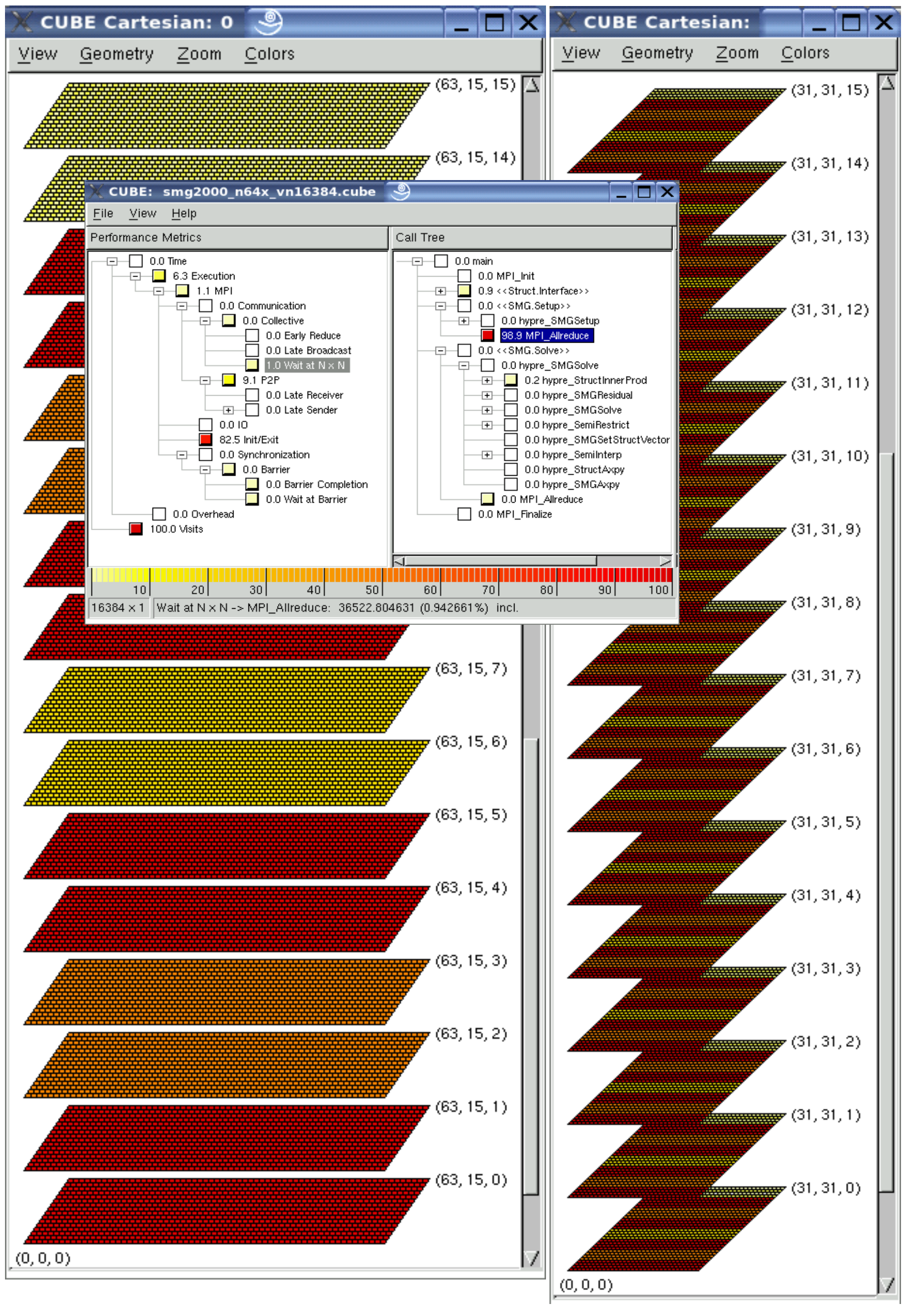


Fig. 3. Execution times for SMG2000 (left) and PEPC-B (right) and their analysis using the sequential EXPERT and new prototype at a range of scales. Linear scaling is the bold dotted line.



```
Performance Metrics
0.0 Time
  6.3 Execution
    1.1 MPI
      0.0 Communication
        0.0 Collective
          0.0 Early Reduce
          0.0 Late Broadcast
          1.0 Wait at N x N
        9.1 P2P
          0.0 Late Receiver
          0.0 Late Sender
      0.0 IO
      82.5 Init/Exit
      0.0 Synchronization
        0.0 Barrier
          0.0 Barrier Completion
          0.0 Wait at Barrier
  0.0 Overhead
100.0 Visits

Call Tree
0.0 main
  0.0 MPI_Init
  0.9 <<Struct.Interface>>
  0.0 <<SMG.Setup>>
    0.0 hypre_SMGSetup
    98.9 MPI_Allreduce
  0.0 <<SMG.Solve>>
    0.0 hypre_SMGSolve
      0.2 hypre_StructInnerProd
      0.0 hypre_SMGResidual
      0.0 hypre_SMGSolve
      0.0 hypre_SemiRestrict
      0.0 hypre_SMGSetStructVector
      0.0 hypre_SemiInterp
      0.0 hypre_StructAxpy
      0.0 hypre_SMGAxpy
    0.0 MPI_Allreduce
  0.0 MPI_Finalize

16384 x 1 | Wait at N x N -> MPI_Allreduce: 36522.804631 (0.942661%) incl.
```

Fig. 4. Analysis report for ASC SMG2000 on 16,384 processors of BlueGene/L highlighting the distribution of the *Wait at N x N* performance metric in the SMG.Setup `MPI_Allreduce` on the physical machine topology distribution (left) and MPI process topological distribution (right).

the currently sequential pre- and postprocessing steps and thereby eliminating the need to read and write intermediate data files. By contrast, the actual procedure of replaying and analyzing the event traces, the focus of this paper, exhibits a satisfactory scaling behavior up to very large configurations. On account of its replay-based nature, the time needed for this part of the analysis procedure depends on the communication behavior of the target application. Since communication is a key factor in the scaling behavior of the target application as well, similarities can be seen in the way both curves evolve as the number of processes increases.

Notably, the total time for the new analysis approach is already more than one order of magnitude faster than the sequential analysis based on EXPERT, which makes it possible to examine wider (and longer) parallel traces in a reasonable time.

While SMG2000 is a reasonable test case for examining the scaling behavior of performance analysis to large scales, as a well-optimized benchmark application, the analysis results are of little interest (see Figure 4). On the other hand, PEPC-B is a relatively new application which has recently been scaled in size and the performance report shows that communication and load imbalance have become increasingly important issues.

6 Conclusion and Future Work

We have presented a novel approach for automatically analyzing event traces of large-scale applications based on exploiting the distributed memory capacity and the parallel processing capabilities of modern supercomputing systems. Instead of sequentially analyzing a single and potentially large global event trace file, we analyze separate local trace files with an analyzer, that is a parallel application in its own right, replaying the target application's communication behavior. This approach has been elaborated to implement the detection algorithms for a variety of performance problems related to the use of the MPI-1 parallel programming interface. In the future, we plan to add support for additional APIs, such as OpenMP and MPI-2, and will investigate using a smaller number of processes for the replay analysis than were used for the measurement, to provide greater analysis flexibility.

To evaluate the scalability of our approach, we have performed experiments with different applications using our prototype implementation on up to 16,384 CPUs. Although the overall analysis time is currently dominated by the sequential parts of the procedure and associated file I/O, the new approach is already more than one order of magnitude faster than the sequential analysis carried out by the EXPERT tool, thereby enabling analyses at scales that have been previously inaccessible.

Since the remaining sequential overhead can be reduced by integrating and parallelizing the pre- and postprocessing parts to eliminate the need to read and write intermediate data files, these early results point to further improvements that can be realized based on the new approach, as we focus on these parts of the analysis work flow. The all-in-memory analysis (perhaps using cCCGs) will also be explored for opportunities to facilitate the detection of new and more complex performance problems.

References

1. Nagel, W., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer **63, XII**(1) (1996) 69–80
2. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP : A Parallel Program Development Environment. In: Proc. 2nd Int'l Euro-Par Conf. (Lyon, France), Springer (1996)
3. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture **49**(10-11) (2003) 421–439
4. Wolf, F., Mohr, B., Dongarra, J., Moore, S.: Efficient Pattern Search in Large Traces through Successive Refinement. In: Proc. European Conf. on Parallel Computing (Euro-Par, Pisa, Italy), Springer (2004)
5. Wolf, F., Freitag, F., Mohr, B., Moore, S., Wylie, B.: Large Event Traces in Parallel Performance Analysis. In: Proc. 8th Workshop on Parallel Systems and Algorithms (PASA, Frankfurt/Main, Germany). Lecture Notes in Informatics, Gesellschaft für Informatik (2006)
6. Freitag, F., Caubet, J., Labarta, J.: On the Scalability of Tracing Mechanisms. In: Proc. European Conference on Parallel Computing (Euro-Par, Paderborn, Germany). Lecture Notes in Computer Science 2400, Springer (2002)
7. Wu, C.E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., Gropp, W.: From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In: Proc. SC2000 (Dallas, TX, USA). (2000)
8. Brunst, H., Nagel, W.E.: Scalable Performance Analysis of Parallel Systems: Concepts and Experiences. In: Parallel Computing: Software Technology, Algorithms, Architectures and Applications, Elsevier (2004) 737–744
9. Knüpfer, A., Nagel, W.E.: Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis. In: Proc. of the International Conference on Parallel Processing (ICPP, Oslo, Norway), IEEE Computer Society (2005) 165–172
10. Roth, P.C., Miller, B.P.: On-line automated performance diagnosis on thousands of processes. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06, New York City, NY, USA). (2006)
11. Fürlinger, K., Gerndt, M.: Distributed Application Monitoring for Clustered SMP Architectures. In: Proc. 9th Int'l Euro-Par Conf. (Klagenfurt, Austria), Springer (2003)
12. Fahringer, T., Gerndt, M., Mohr, B., Wolf, F., Riley, G., Träff, J.L.: Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrum Jülich (2001) Revised version.
13. Fahringer, T., Seragiotto, Jr., C.: Modelling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In: Proc. SC2001 (Denver, CO, USA). (2001)
14. Jorba, J., Margalef, T., Luque, E.: Performance Analysis of Parallel Applications with KappaPI 2. In: Proc. Parallel Computing 2005 (ParCo, Málaga, Spain). (2006)
15. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An Algebra for Cross-Experiment Performance Analysis. In: Proc. Int'l Conf. on Parallel Processing (ICPP, Montreal, Canada), IEEE Computer Society (2004)
16. Wolf, F.: Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich (2003) ISBN 3-00-010003-2.
17. The BlueGene/L Team at IBM and LLNL: An overview of the BlueGene/L supercomputer. In: Proc. SC2002 (Baltimore, MD, USA), IEEE Computer Society (2002)
18. Advanced Simulation and Computing Program: The ASC SMG2000 Benchmark Code. http://www.llnl.gov/asc/purple/benchmarks/limited/smg/ (2001)
19. Gibbon, P.: PEPC: A Multi-Purpose Parallel Tree-Code. http://www.fz-juelich.de/zam/pepc/ (2005)