MPI Trace Compression Using Event Flow Graphs

Xavier Aguilar[1], Karl Fürlinger[2], and Erwin Laure[1]

[1] KTH Royal Institute of Technology,
High Performance Computing and Visualization Department (HPCViz)
and Swedish e-Science Research Center (SeRC),
Lindstedvägen 5, 10044 Stockholm, Sweden
[2] Ludwig-Maximilians-Universität (LMU) Munich,
Computer Science Department, MNM Team,
Oettingenstr. 67, 80538 Munich, Germany

Abstract. Understanding how parallel applications behave is crucial for using high-performance computing (HPC) resources efficiently. However, the task of performance analysis is becoming increasingly difficult due to the growing complexity of scientific codes and the size of machines. Even though many tools have been developed over the past years to help in this task, current approaches either only offer an overview of the application discarding temporal information, or they generate huge trace files that are often difficult to handle.

In this paper we propose the use of *event flow graphs* for monitoring MPI applications, a new and different approach that balances the low overhead of profiling tools with the abundance of information available from tracers. Event flow graphs are captured with very low overhead, require orders of magnitude less storage than standard trace files, and can still recover the full sequence of events in the application. We test this new approach with the NERSC-8/Trinity Benchmark suite and achieve compression ratios up to 119x.

Keywords: MPI event flow graphs, trace compression, trace reconstruction, performance monitoring.

1 Introduction

Current petascale systems provide massive computing power to run scientific simulations in many disciplines ranging, for example, from weather modeling to protein structure analysis. However, their efficient use requires optimized applications with several levels of parallelism, efficient inter-process communication for complex network topologies and optimized memory access through deep memory hierarchies. Therefore, tools to characterize and better understand the performance behavior of applications are an essential part of the HPC landscape.

Performance tools for HPC systems have been widely studied and developed over the last years. These tools can be divided into two families: profilers and tracers. Profilers generate reports with execution statistics, whereas tracers produce time-stamped event log files. Profilers are less intrusive and more scalable than tracers but profilers do not maintain a record of the structure and sequence of events. In contrast, tracers give the whole picture of what happened during the run time of an application, but are limited in scalability due to the huge amount of data they generate. Current tracing methodologies can produce trace files in the order of gigabytes for only a few minutes of application execution [1]. The size of the trace files also grows drastically as the number of cores used by an application increases. Thus, new scalable methods for performance data collection maintaining sequence and temporal order of the information are needed.

In this paper, we propose a novel approach for application characterization using *event flow graphs* which is designed to combine the advantages of profiling and tracing. This method has the scalability of profiling without discarding the temporal ordering of events performed by the application. We have implemented our solution in the Integrated Performance Monitoring tool (IPM) [2,3], a light-weight and scalable profiling tool for parallel applications. It uses a hash table in memory to store performance data and provides rich metrics about MPI-related events such as MPI timings, communication volume and the communication topology. IPM is open-source and is available freely from `http://www.ipm2.org` under the LGPL license.

The rest of this paper is structured as follows: In Sect. 2 we define and describe our approach for generating MPI event flow graphs. Section 3 shows some experimental results using the NERSC-8/Trinity Benchmark Suite. In Sect. 4 we review some of the related work in trace compression. The paper ends with future work and conclusions in Sect. 5 and Sect. 6, respectively.

2 MPI Event Flow Graphs

In this work, we use and extend the definitions of Fürlinger et al. [4] for a formal treatment of performance monitoring events. We start with an MPI application with n processes (identified by their ranks $[0..n-1]$), where each process i is characterized by a set of events $E_i \subseteq E$ where E represents all the events that happened during the run time of the application. An event can be any action performed by the application, but in this work we restrict ourselves to MPI operations. In other words, an event is an MPI primitive call.

Each event e has a signature $\delta(e)$ that captures the aspects of the events we are interested in. We can think of the signature as a k-tuple of components $\delta(e) = (\delta^1(e), \delta^2(e), ..., \delta^k(e))$ which represent relevant metrics, such as the type of MPI call, communication partner rank, data transfer size, callsite (source code position), or program region. The mapping from event to signature is not necessarily injective and therefore statistics are recorded for each different signature value. Hence, we can conceptually represent the performance behavior of an MPI process as a table where each row is an event indexed by its signature and each column is a different statistic (number of occurrences, minimal duration, maximal duration, etc.).

In practical terms, the performance behavior is recorded in a hash table in memory which is implemented in IPM with the event signatures $\delta(e)$ being used as the hash keys. The values in this hash table are performance metrics such as the number of occurrences and different timings (minimum, maximum and total duration) of each event. This lets us store performance data in the hash keys as well as in the table entries, thereby reducing the monitoring overhead. Notice that if we include the event timestamp as a component of the signature, we have a model for tracing. If the timestamp is omitted, we lose the temporal dimension of the data and instead have a model for profiling since we cannot know the order in which the events happened during the application's run time. However, as we show in this paper, the temporal ordering of events can nevertheless be fully recovered by keeping track of a (very short) history of the event signatures.

Consider again an MPI application with n processes and a set of events $E_i = \{e_0, e_1, ..., e_m\}$ belonging to process with rank i. Let $\delta(e) : E_i \mapsto S_i$ be the signature function at rank i and $s_i^0 \in S_i$ an initial signature value. Then $\delta'(e)$ with

$$\begin{aligned} \delta'(e_0) &= (s_i^0, \delta(e_0)) \\ \delta'(e_i) &= (\delta(e_{i-1}), \delta(e_i)) \qquad \text{if } i > 0 \end{aligned}$$

represents the *signature history* for δ. Then, the directed weighted graph $G = (N_i, L_i, w_i, s_i^0)$ with the event signatures forming the set of nodes N_i and the signature history the set of edges L_i

$$\begin{aligned} N_i &= \{\delta(e_i)\} \qquad e_i \in E_i \\ L_i &= \{\delta'(e_i)\} \qquad e_i \in E_i \\ w_i : L_i \mapsto \mathbb{N} \qquad w_i(l) &= |\{e_i : \delta'(e_i) = l\}| \qquad l \in L_i \end{aligned}$$

is the *event flow graph* for the MPI rank i with s_i^0 as the initial node of the graph. In other words, in the event flow graph the nodes correspond to the MPI calls performed by the application and the edges correspond to the transitions between them. The edge weight (w_i) or edge count is the number of transitions between nodes. Figure 1 depicts a simple MPI application and the corresponding event flow graph for one of its MPI processes, where MPI_Init is the initial node of the graph. Notice that the application has as many graphs as MPI processes.

2.1 Reconstructing Traces from Event Flow Graphs

For the simple example in Fig. 1 we see that the event flow graph completely captures the behavior of an MPI process. It contains all the events performed by the process (nodes of the graph) and the transitions between them (edges between nodes). Therefore, the path $N_i = \{s_i^0, s_i^1 ... s_i^n\}$ from the initial node s_i^0 to the final node s_i^n of the graph corresponds to the event trace for process with rank i. The total number of events (length of the path) in the trace is

$$\sum w_i(l) + 1 \quad \forall l \in L_i$$

```
void main(int argc, char *argv[ ]) {
   MPI_Init(...);
   MPI_Comm_size(...);
   MPI_Comm_rank(..., &myrank);

   for( i = 0; i < 10; i++) {
        if (myrank is even)
            MPI_Send(...);
        else
            MPI_Recv(...);

        MPI_Reduce(...);
   }
   MPI_Finalize( );
}
```

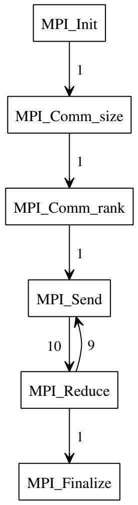


Fig. 1. A simple MPI program and the event flow graph generated for an MPI process with an even rank number

and the number of times that each event e_i appears, also known as node cardinality for the node $\delta(e_i)$, is

$$\sum w_i(in\_edges(\delta(e_i)))$$

In other words, the number of events in the trace is the sum of all edge weights of the graph plus one and each event appears as many times as the total weight of its incoming edges.

In this paper we are only concerned with reconstructing the sequence of events in a trace (time stamps and intervals between events are topics for future work). It is clear that the trace can be easily reconstructed for simple cases such as linear graphs and graphs with a single loop such as the one in Fig. 1. However, there are cases that cannot be reconstructed using flow graphs in this manner. This occurs for applications with conditional branches within a loop, for example, the code snippet in Fig. 2. When reconstructing the trace, we cannot know the order of the calls after the MPI_Barrier across loop iterations.

Thus, we extended our model to cover such cases. Firstly, we added a sequence number to the exit edges of the branch nodes (that is, nodes with more than one exit edge). This new weight is defined as a 2-tuple $< \mathbb{N}, \mathbb{N} >$ where the first element is the sequence number for that edge and the second element is the edge count as defined above. With this extra data, we always know which edge was taken in a branch node when traversing the graph. Figure 3 shows this new extended model.

Secondly, we changed our directed graphs to multidigraphs, that is, directed graphs with more than one edge between the same two nodes. This new graph

```
for( i = 0; i < 10; i++) {
     MPI_Barrier(...);
     if (i < 5)
         MPI_Bcast(...);
     else
         MPI_Gather(...);

     MPI_Reduce(...);
}
```

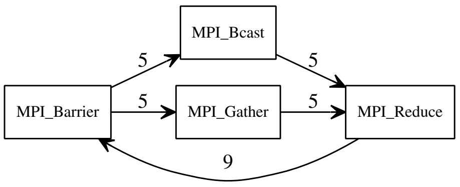


Fig. 2. Example of a conditional branch within a loop and its corresponding event flow graph

model can represent applications in which the conditional branches within a loop vary across loop iterations as depicted in Fig. 3.

```
1: MPI_Comm_rank
2: MPI_Barrier
3: MPI_Send
4: MPI_Barrier
5: MPI_Recv
6: MPI_Barrier
7: MPI_Send
8: MPI_Finalize
```

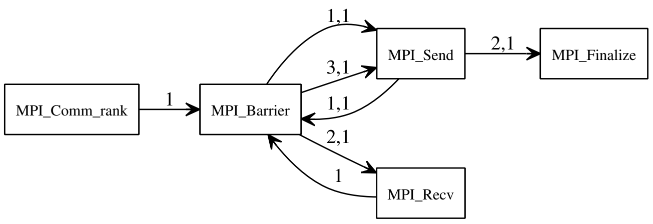


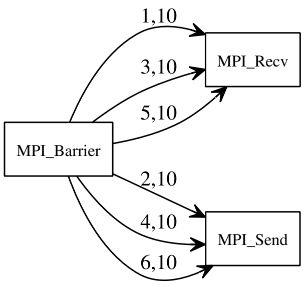
Fig. 3. A sequence of MPI operations and the corresponding multidigraph

2.2 Compressing Edges in Branch Nodes

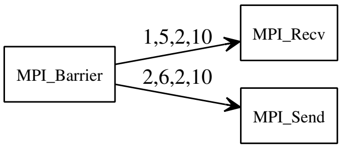
As shown in the previous section, our new event flow graphs are multidigraphs with sequence numbers in edges that have the same source node. Thereby, we can always reconstruct the event trace associated with an application without any loss of temporal order information by traversing the graph edges in ascending order of their sequence number.

However, creating multiple edges between nodes to record the sequence order can lead to huge graphs. In fact, our experiments showed that this situation is quite common among real applications which sometimes have flow graphs with thousands of edges going out from one node. Nevertheless, those graphs usually exhibit repetitive patterns in terms of the multiple edges between nodes as shown in Fig. 4. In that case, the application calls MPI_Barrier followed by MPI_Recv 10 times, then it calls MPI_Barrier followed by MPI_Send 10 times, afterwards it again calls MPI_Barrier followed by MPI_Recv 10 times, and so on, until MPI_Recv and MPI_Send have each been called 30 times.

As we can see in the figure, the sequence numbers for those edges in the event flow graph follow different arithmetic progressions, that is, the difference between two consecutive numbers in the sequence is constant. In such cases, the



(a) Nodes with uncompressed edges.



(b) Nodes with compressed edges.

Fig. 4. Branch compression of multiple edges between nodes

set of edges can be compressed into a single one as long as their edge count is the same. Using this approach, the new weight for the compressed edges is a 4-tuple $< \mathbb{N}, \mathbb{N}, \mathbb{N}, \mathbb{N} >$ where the first element is the first number of the sequence, the second element is the last number of the sequence, the third element is the stride and the last element of the tuple is the edge count. For instance, the set of edges $[1, 10], [3, 10], [5, 10]$ in Fig. 4 from the MPI_Barrier to the MPI_Send node can be compressed into a single edge with weight $< 1, 5, 2, 10 >$. Hereby, we increase the readability of the graphs and reduce the space needed to store them. For irregular patterns without a clear stride no compression is possible and individual edges need to be stored.

2.3 Implementation in IPM

We have extended IPM to generate MPI event flow graphs as described in the previous section. IPM maintains event statistics such as the total duration, the minimum and maximum time and the number of occurrences for all MPI calls. These statistics are stored in a hash table using the event signatures described in Sect. 2 as the hash key for each event.

To record the transitions between events, we introduced a second hash table that contains pairs of event signatures. This "history" hash table keeps information on transition pairs of event signatures $(\delta(e_{i-1}), \delta(e_i))$. IPM keeps track of the last event signature by storing it in a variable and updating it each time there is a new insertion into the transition hash table. Moreover, IPM also keeps track of branches within loops by checking if there are two pairs in the transition hash table $(< \delta(e_i), \delta(e_j) >, < \delta(e_i), \delta(e_k) >)$ where $\delta(e_j) \neq \delta(e_k)$. If that is so, each element is given a sequence number indicating their arrival order. IPM also joins elements in the transition hash table to compress the number of edges between nodes as described in Sect. 2.2. It keeps track of the old branches for each node. When a branch finishes, IPM checks if the sequence number of the branch follows an arithmetic progression in relation to any of the older branches of that particular node. If that is the case and if both branches have the same edge count, the two branches are compressed into a single branch. Upon program termination, IPM constructs the event flow graph for each MPI process by matching pairs of event edges.

3 Experiments

In order to test our approach for trace reconstruction from MPI event flow graphs, we used the following mini-applications from the NERSC-8/Trinity Benchmarks suite [5]: AMG, an algebraic multigrid solver for linear systems on unstructured grids; GTC, a 3D Particle-in-cell code (PIC) with a non-spectral Poisson solver used for gyrokinetic particle simulation of turbulent transport in burning plasma; MILC, a code for simulating four dimensional SU(3) lattice gauge theory to study quantum chromodynamics (QCD); SNAP, a proxy application that models the performance of a modern discrete ordinates neutral particle transport application, PARTISN [6]; MiniDFT, a plane-wave DFT mini-kernel that computes self-consistent solutions for the Kohn-Sham equations; MiniFE, a mini-application that implements different kernels representative of implicit finite-element applications; MiniGhost, a mini-application that implements a difference stencil across a homogenous three dimensional domain.

The experiments were performed on a Cray XE6 with 2 twelve-core AMD MagnyCours at 2.1 GHz per node. The nodes are interconnected through a Cray Gemini Network, each of them having a total of 32 GB DDR3 memory. The benchmarks were compiled with Intel 12.1.5 and run using the small test case that is provided for each one of them.

3.1 Overhead

Figure 5 shows for each benchmark the percentage of overhead introduced by IPM over their total running time (writing the graph files to disk is also included in the percentage). These experiments were run using strong scaling except for SNAP, MILC and GTC. As depicted in the figure, the overhead introduced to generate the *event flow graphs* is almost negligible, being always below 2%.

3.2 Compression Ratios

Table 1 shows the compression ratio for each benchmark in terms of file size between our flow graph file and a standard trace file for that application generated by IPM, in other words, it shows how many times smaller the event flow graph file is compared to the standard trace file. It is important to be aware that both files contain exactly the same amount of information for each MPI call: call name, bytes sent or received, communication partner and callsite. As our current implementation generates one flow graph per MPI process, the table shows statistics for the minimum, maximum and average compression ratios for all processes within each application. The results in the table demonstrate that the compression depends on the nature of the benchmark. For instance, we have

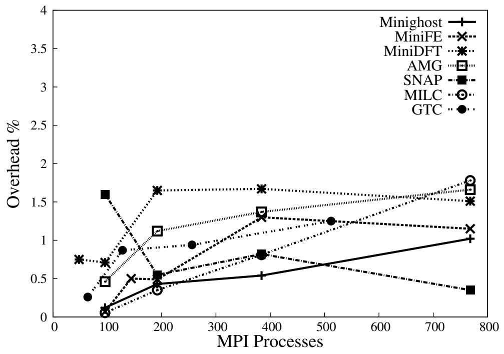


Fig. 5. Percentage of overhead over total running time introduced in the NERSC-8/Trinity benchmarks when generating their event flow graphs

applications such as SNAP with flow graph files 119 times smaller than the standard linear trace whereas in other applications such as AMG the compression ratio is 1.76. In terms of file size, the amount of disk space required to store the traces for a run with 96 cores of SNAP is 1.1 GB whereas the space required for the *event flow graphs* is only 10 MB.

In order to explain this variance in the compression we need to look into some graph metrics. Table 2 gives statistics for the number of nodes, the number of links and the average cardinality of nodes in the graphs. Remember that the node cardinality is the number of instances an event $\delta(e_i)$ happened during the run time of the application as explained in Sect. 2.1. The figures in the tables show that low compression ratios are related to graphs with a large number of nodes with low cardinality such as AMG or MiniDFT. In contrast, graphs consisting of a few nodes with high node cardinality exhibit very good compression ratios.

As explained in Sect. 3, each event is identified uniquely using a signature defined by several metrics. Furthermore, each one of these events is eventually converted into a node in the event flow graph. Therefore, the metrics used as signature elements have an important role in the cardinality of the graph. In our experiments, the event signature was composed of the MPI call name, the MPI rank, the number of bytes associated with the call and the call site. Thus, it is not surprising that applications with huge graphs (such as AMG) have a large number of different call sites and message sizes - this was confirmed by our experiments. The variability in the number of call sites and the sizes of messages leads to a greater number of signatures, and consequently more nodes in the resulting graph.

Finally, we performed another set of experiments with some of the NERSC-8 benchmarks and a five-point stencil code computing a wave 2D equation [7] to measure the increase ratio in file size as we increase the number of simulation time steps. Figure 6 shows that standard trace files increase linearly with the number of simulation steps whereas the event flow graph (EFG) files do not. For most of the benchmarks, the small increment in the graph file size is caused by the addition of new edges to the graphs due to the execution of different call paths as the number of simulation steps increases. (For GTC the number of nodes also increases due to a variation in the size of messages.) However, applications that execute the same loop over time such as the 5-stencil code have constant event flow graph size irrespective of the number of simulation steps. For applications like that, the only difference between graphs from runs with different simulation times is their node cardinality.

Table 1. File compression ratios

Benchmark	Ranks	Min	Max	Avg
AMG	96	1.70	1.85	1.76
GTC	64	37.95	47.65	46.60
MILC	96	38.67	39.44	39.03
SNAP	96	75.37	210.88	119.23
MiniDFT	40	3.14	8.39	4.33
MiniFE	144	15.23	22.25	19.93
Minighost	96	3.84	5.72	4.85

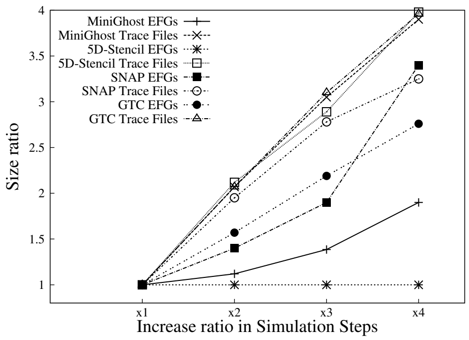


Fig. 6. Increase in file size when increasing simulation steps

Table 2. Number of nodes, edges and cardinality of nodes in the event flow graphs

		Num. of nodes			Num. of edges			Node Cardinality		
Benchmark	Ranks	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
AMG	96	4973	15115	9,348.94	5652	17287	10,586.47	4.44	4.83	4.59
GTC	64	114	130	114.50	120	151	121.20	96.52	109.53	109.10
MILC	96	6330	6347	6330.18	97426	97443	97426.18	1653.01	1657.31	1657.27
SNAP	96	22	28	24.77	340	1729	1,120.26	7,007.50	17,805.91	14,149.22
MiniDFT	40	512	1087	690.30	873	5851	1,980.38	12.39	63.01	27.29
MiniFE	144	73	280	161.08	75	282	163.08	33.86	50.35	45.10
Minighost	96	89	95	92.33	91	135	111.04	12.13	13.89	13.13

4 Related Work

Performance tools for HPC systems have been studied and developed for years. Extrae and Paraver [8,9], and also ScoreP with Vampir [10,11], are tracing toolsets used to visualize the behavior of MPI applications over time. They provide lossless traces that include all the events that happened while the application was running. However, these traces are huge and their size increases linearly with the number of MPI processes. Therefore, the use of such toolsets is limited by scalability constraints. In contrast, our current work with *event flow* *graphs* shows that we can capture the events and their temporal order as tracers do while storing it in files that are a few orders of magnitude smaller. However, our approach is still in an early stage and more work is needed to reach the same level of usability and information granularity as that provided by current tracing tools, for example, including continuous data such as timestamps or hardware performance counters in the trace.

Our work is also related to lightweight profiling tools such as mpiP [12] or Gprof [13]. These tools generate profiles of aggregated information with very low overhead. Although these tools can provide a good overview of the performance problems for a particular application, they lack the temporal order of data needed for in-depth performance analysis. In contrast, IPM can provide temporal order in the performance data using *event flow graphs.* Additionally, IPM also provides standard reports with aggregated statistics.

Scalatrace [14] is a tracing framework that provides on-the-fly lossless trace compression of MPI communication traces. It implements intra-node compression describing single loops with RSDs [15] and using techniques such as callpath compression. Scalatrace also implements inter-node compression at the end of the run when each process trace is merged into a single one for the whole application. Scalatrace comes with a replay mechanism for a later analysis of those traces. Our work differs from Scalatrace in the sense that we do not compress series of events, but instead record the behavior of an application using graphs. We believe this approach has better compression ratios and much less overhead as discussed in Sect. 3. Furthermore, our approach also makes it possible to replay traces later for the purposes of performance analysis. Nevertheless, our current implementation still lacks inter-node compression, generating one file per process. This is subject to future work though.

Krishnamoorthy et al. use SEQUITUR to compress traces creating context-free grammars from the sequence of MPI calls [16]. In order to achieve better compression, the trace is not compressed at an event level, but instead every call argument is compressed in a different stream. This loses any program structure in the resulting trace and makes it unreadable. In contrast, our approach keeps the program structure, thus allowing us to easily visualize the traces.

Knüpfer et al. use Complete Call Graphs (CCGs) to compress post-mortem traces according to the call stack [17]. This approach builds a call graph and replaces similar repeated sub-trees with a reference to a single instance. Therefore, CCGs can be very useful for trace analysis tools, reducing their memory footprint and allowing them to deal with bigger traces. However, this method does not eliminate the burden of generating large traces while the application runs.

Flow graphs have been widely used in other areas of computer science such as code generation and analysis. In those contexts, compilers generate flow graphs from their intermediate representation (IR) where nodes are code blocks and edges are branches that a program may take. Our work differs in the sense that the nodes in our graphs are communication events instead of code blocks. In addition, the edges of our event flow graphs are not possible branches but rather transitions that actually happened during the execution.

5 Future Work

Using *event flow graphs* in the analysis of MPI parallel applications opens up many possibilities such as developing new tools to visualize, navigate and interact with graphs. Possible visualization features could be graph coloring depending on different metrics or highlighting differences among graphs to detect load imbalance among processes. The graph approach also allows the use of different algorithms and techniques for automatic graph analysis, for instance, detecting loops in the graph and time spent in those loops. Furthermore, these new performance tools could provide trace reconstruction features for just some sections of the graph or a couple of iterations of a graph cycle.

Our current implementation of the *event flow graphs* in IPM does not keep any time information on call duration in the graph. Thus, trace reconstruction with timestamps is not possible yet. Therefore, we are looking into methods for trace reconstruction that include time information. Furthermore, we want to apply those methods for the reconstruction of any continuous data in the trace, for example, hardware performance counters.

Finally, another aspect we want to explore in the future is inter-node trace compression across ranks. Our current version always generates one flow graph per process. However, it is usual in parallel application that a set of processes has similar or identical behavior. In such cases, the graphs generated by those processes will be similar as well, and thus, they can be compressed into a single graph that could be used to describe that whole set of processes with similar execution.

6 Conclusion

Performance analysis through tracing is the best method to understand the behavior of applications. However, tracing techniques have scalability limitations due to the amount of information that is generated. In this paper we have presented a disruptive approach for performance tracing of MPI parallel applications using *event flow graphs*. This new method combines the scalability and low overhead of profiling methods with the lossless information capabilities of tracing tools. We evaluated our implementation using several mini-applications from the NERSC-8/Trinity Benchmark Suite. The experiments showed promising results, achieving file compression ratios up to 119 with overheads below 2%. Furthermore, the use of applications with longer simulations would allow even better compression ratios because the same paths in the application are executed more times. Although our work is still at an early stage, we believe it has strong potential to be a way towards developing performance analysis tools that are effective at an exascale level.

References

1. Labarta, J., Gimenez, J., Martinez, E., González, P., Servat, H., Llort, G., Aguilar, X.: Scalability of visualization and tracing tools. In: Proc. 11th Parallel Computing Conf. (ParCo 2005), pp. 869–876 (2005)

2. Fuerlinger, K., Wright, N.J., Skinner, D.: Effective performance measurement at petascale using ipm. In: 2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS), pp. 373–380. IEEE (2010)
3. Aguilar, X., Fürlinger, K., Laure, E.: Online performance data introspection with ipm. In: The 15th IEEE International Conference on High Performance Computing and Communications (2013) (to be published)
4. Fürlinger, K., Skinner, D.: Capturing and visualizing event flow graphs of mpi applications. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009 Workshops 2009. LNCS, vol. 6043, pp. 218–227. Springer, Heidelberg (2010)
5. NERSC-8 / Trinity Benchmarks WWW site, `http://www.nersc.gov/systems/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/`
6. Alcouffe, R.E., Baker, R.S., Dahl, J.A., Turner, S.A., Ward, R.: Partisn: A time-dependent, parallel neutral particle transport code system. Los Alamos National Laboratory, LA-UR-05-3925 (May 2005)
7. MPICH wiki, `http://wiki.mpich.org/mpich/images/1/17/Wave2d.cpp.txt`
8. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: Transputer and Occam Developments, vol. 44, pp. 17–31 (1995)
9. Servat, H., Llort, G., Huck, K., Giménez, J., Labarta, J.: Framework for a productive performance optimization. Parallel Computing 39(8), 336–353 (2013)
10. Knüpfer, A., Rössel, C., Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al.: Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In: Tools for High Performance Computing 2011, pp. 79–91. Springer (2012)
11. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The vampir performance analysis tool-set. In: Tools for High Performance Computing, pp. 139–155. Springer (2008)
12. Vetter, J.S., McCracken, M.O.: Statistical scalability analysis of communication operations in distributed applications. In: ACM SIGPLAN Notices, vol. 36, pp. 123–132. ACM (2001)
13. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. ACM Sigplan Notices 17(6), 120–126 (1982)
14. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: Scalatrace: Scalable compression and replay of communication traces for high-performance computing. Journal of Parallel and Distributed Computing 69(8), 696–710 (2009)
15. Havlak, P., Kennedy, K.: An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems 2(3), 350–360 (1991)
16. Krishnamoorthy, S., Agarwal, K.: Scalable communication trace compression. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 408–417. IEEE Computer Society (2010)
17. Knupfer, A., Nagel, W.E.: Construction and compression of complete call graphs for post-mortem program trace analysis. In: International Conference on Parallel Processing, ICPP 2005, pp. 165–172. IEEE (2005)