Automatic On-Line Detection of MPI Application Structure with 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
xaguilar@pdc.kth.se

[2] Computer Science Department, MNM Team, Ludwig-Maximilians-Universität (LMU) Munich, Oettingenstr. 67, 80538 Munich, Germany

Abstract. The deployment of larger and larger HPC systems challenges the scalability of both applications and analysis tools. Performance analysis toolsets provide users with means to spot bottlenecks in their applications by either collecting aggregated statistics or generating lossless time-stamped traces. While obtaining detailed trace information is the best method to examine the behavior of an application in detail, it is infeasible at extreme scales due to the huge volume of data generated.

In this context, knowing the application structure, and particularly the nesting of loops in iterative applications is of great importance as it allows, among other things, to reduce the amount of data collected by focusing on important sections of the code.

In this paper we demonstrate how the loop nesting structure of an MPI application can be extracted on-line from its event flow graph without the need of any explicit source code instrumentation. We show how this knowledge on the application structure can be used to compute post-mortem statistics as well as to reduce the amount of redundant data collected. To that end, we present a usage scenario where this structure information is utilized on-line (while the application runs) to intelligently collect fine-grained data for only a few iterations of an application, considerably reducing the amount of data gathered.

Keywords: Application structure detection · Flow graph analysis · Performance monitoring · Online analysis · Automatic loop detection

1 Introduction

Computer simulations are nowadays an important method of scientific discovery. By using computers, scientists can model processes that would be difficult or impossible to reproduce and study in a real-world scenario. Moreover, the deployment of larger and larger High Performance Computing (HPC) systems provides scientists with an opportunity to solve problems which could not be

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 70–81, 2015.
DOI: 10.1007/978-3-662-48096-0_6

tackled before. However, scientific applications have to be tuned and highly optimized to effectively use all the computational power provided by current HPC infrastructures.

In our previous work we have explored the use of event flow graphs as a novel method for MPI monitoring and analysis, demonstrating that graphs are a good compressed representation of MPI event traces due to the iterative nature of MPI parallel applications [1,8]. Event flow graphs retain the temporal order of the events executed during the lifetime of a program without saving explicit timestamps. Thus, graphs can be used to reconstruct the full ordered sequence of events performed by the application.

In this paper we present how event flow graphs can be used beyond trace compression and reconstruction. Our approach for automatic analysis of event flow graphs sheds light on the inherent structure of parallel applications, for instance, revealing the nesting loop structure present in the program.

Knowledge of the application structure can be very useful both for post-mortem and for on-line performance analysis. On one hand, this structural knowledge can be utilized to automatically generate reports that show the user where and how time is spent among loops, and how the performance characteristics of those loops evolve over the lifetime of an application. This can be done without the need of recompilation, access to the source code, or user involvement at all. On the other hand, knowing the structure of a program while it runs can benefit how data is collected and aggregated. For instance, data can be aggregated at a loop level instead of keeping every event, or redundant information can be reduced by keeping fine-grained data for a few loop iterations only. Furthermore, this structural knowledge can also be used, for example, to help a dynamic runtime system with its decision making process, or to feed an external monitoring tool that decides the grain of the performance data collected.

The contributions of this paper include:

- We develop a simple mechanism to extract the structure of applications with very low overhead and without the need to have access to the source code.
- We present a real usage scenario in which the structure of an application is detected automatically while the application runs to intelligently select the performance data collected.
- We demonstrate that the overall performance behavior of an iterative MPI application is still captured with our approach by selecting only a few representative iterations.

The remainder of this paper is organized as follows: Sect. 2 provides background on our previous work on event flow graphs. Section 3 describes the mechanisms implemented for application structure detection, and its on-line application. Section 4 presents a real usage scenario where the on-line detection of an application's structure is used to intelligently select the amount of performance data generated. Section 5 surveys related work. Finally, Sects. 6 and 7 discuss future work and conclusion, respectively.

2 Background: IPM and Event Flow Graphs

The work presented in this paper builds on top of the Integrated Performance Monitoring (IPM) tool [13]. In [1,8], IPM was extended to capture and generate event flow graphs of MPI parallel applications. Upon program termination, IPM generates for each MPI process a weighted directed graph in which nodes are the different[1] MPI calls performed by that process, and edges are the transitions between those calls. In other words, edges are the computational parts between two MPI calls. Therefore, event flow graphs keep the temporal order of the events performed by the application. In addition, metrics such as timers and hardware counters can be associated with the nodes and edges of the graph, increasing thereby the usability of such graphs.

3 Automatic Analysis of Event Flow Graphs

3.1 Loops in Event Flow Graphs

It is commonly accepted conventional wisdom that the vast majority of HPC scientific parallel codes are iterative and spend most of their time in loops. These scientific applications are usually composed of a large outer loop, which controls the simulation time-steps, and which contains several inner loops with different nesting levels. Since most of the application time is spent in loops, they become one of the main targets when analyzing and optimizing programs.

Most MPI parallel programs contain MPI operations in some of their loops, as data needs to be shared among processes across loop iterations. In those cases, the generated event flow graphs will contain cycles. Thus, by detecting those cycles, we are detecting the actual loops that drive the simulation process in the application. Loops without MPI calls are not detected with this approach, however, their behavior gets captured in the edges of the graph as these loops are just pure computational parts between two MPI calls.

Figure 1 shows the basic cycle shapes that can appear in our event flow graphs. Each one of the loops is accompanied with a source code example that generates such a loop structure. Calls to *A, B, C* and *D* represent any MPI routine. As can be seen in the picture, loops can range from single node cycles, through several nesting structures, to cycles with multiple tails. In addition, all these basic loop structures can be combined to form more complex ones. At the moment, our work focuses in reducible loops [23], that is, loops with just one entry point. Formally, given a loop L with header h (h dominates all the nodes in the loop L) and an edge $< u, v >$, if $u \notin L$ and $v \in L - \{h\}$, then v is a re-entry point and the loop is irreducible.

Irreducibility in our event flow graphs can be caused by two different factors: the application's source code structure, and the event signatures used for the

[1] What constitutes different MPI calls for recording our event flow graphs is governed by a configurable event signature in IPM. The signature usually consists of the name of the call and its call site and can optionally also include the communication partner rank and the transfer size.

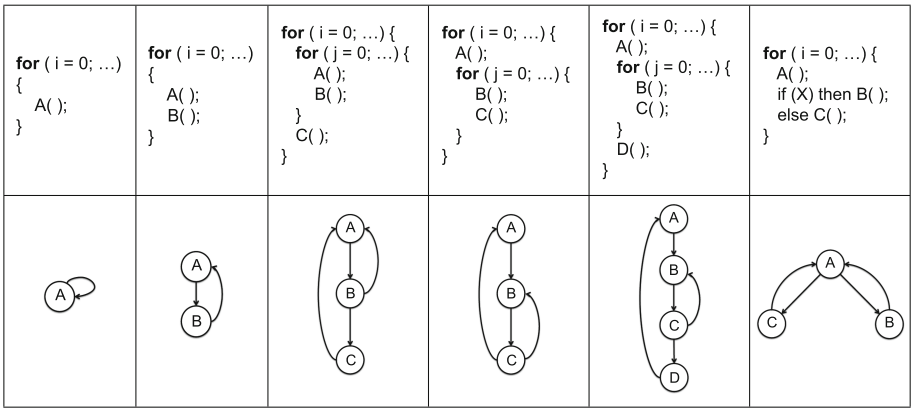


Fig. 1. Different cycle shapes in event flow graphs.

nodes of the graph. Irreducibility caused by unstructured programming (e.g., the use of `goto`), however, is nowadays rare and will become even rarer in the future due to the adoption of more structured programming practices [22]. On the other hand, irreducibility caused by the event signature used can always be solved by changing the signature. If transfer size is used in the signature for example, one single MPI call in the source code can be translated into different nodes in the graph if such a call has different transfer sizes at runtime. Thereby, generating sometimes irreducible cycles. However, this situation is also rare, and the graph usually becomes reducible again by using the call name and call site as event signature, because then each graph node maps exclusively to only one MPI call in the source code.

Algorithms for graph cycle detection have been studied and used in the field of compilers for years [12,20,21]. Our framework for graph analysis implements the algorithm from [24]. This algorithm traverses the graph using a depth-first search (DFS) and runs in almost linear time. It does not require any complicated data structures as other cycle detection algorithms, and thus, it is much easier to implement. After running the algorithm, loop header nodes (entry node in a graph cycle) are identified and all loop nodes are labeled with their corresponding header. If multi-entry loops (irreducible loops) are found, the graph is marked as irreducible and the process ends.

Once nodes have been labeled, our framework knows for every graph node to which loop it belongs, which loops are outermost, which are nested, etc. Thereby, our analysis tool can provide detailed exclusive and inclusive loop metrics such as percentage of MPI time over total loop time. Figure 2 shows the percentage of MPI time across ranks in the main simulation loop for MiniFE, a finite-element code. The MPI time is the inclusive total time for this outermost loop, that is, the time for its nested loops is also included. The picture shows that MiniFE suffers of imbalance in this loop as some processes spend around 20 % of their time in MPI whereas some others less than 5 %. It is important to

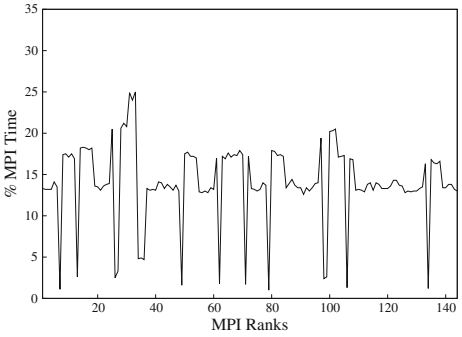


Fig. 2. Percentage of MPI time across ranks in the main simulation loop of MiniFE.

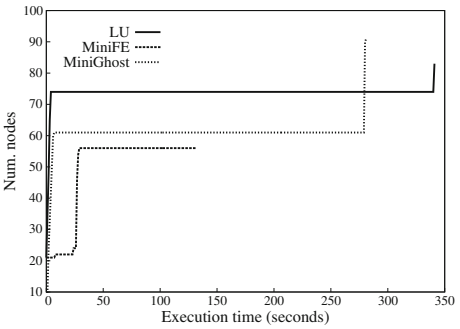


Fig. 3. Change in the number of nodes in graphs from several applications during their execution time.

remark that the statistics on loops provided by our approach are automatically obtained without any user involvement or source code modification. Our solution utilizes the PMPI interface to intercept MPI calls, and the libunwind library to determine their call sites.

The use of event flow graphs together with automatic loop detection opens many possibilities for post-mortem performance analysis of MPI parallel applications. However, this topic is out of the scope of this article. More details in the use of graphs for visual performance analysis of MPI applications can be found in [2].

3.2 Runtime Loop Detection

The previous section has focused on the automatic post-mortem analysis of graphs to detect the structure of an application, however, our mechanism can also be used in real-time while applications run.

In order to minimize the amount of overhead introduced into the application, our on-line loop detection mechanism is performed only once when the application has reached a stable state. The application is considered stable when it enters into an iterative phase in which its performance behavior presents minor fluctuations. Most scientific applications arrive into this state when they start executing their main simulation loop, which is executed for most of the running time. In our case, this situation is reflected in the number of nodes in the event flow graph. In other words, once the application reaches an iterative stable state, the number of nodes in the graph does not change since the same MPI calls are repeated over and over again. Figure 3 shows the number of nodes in the graph during application execution for one process of different MPI applications. It can be seen, for instance with MiniGhost, that after some short initialization time, the number of nodes in the graph does not change during most of the execution time since the application has entered its stable state. At the end of its execution, the number of nodes in the graph increases again as the application exits the main loop and performs some new MPI calls before finalizing. However, these

Table 1. Overhead introduced by IPM over total application running time.

Metrics	MiniGhost	MiniFE	GTC	MiniMD	BT	LU
Ranks	96	144	64	192	144	128
% Overhead	0.9 %	0.65 %	1.06 %	1.10 %	0.81 %	1.2 %

new final nodes represent a minimal percentage over the total running time, and thus, they are not important for performance analysis purposes.

To detect when an application becomes stable, IPM checks at regular intervals if the graph has changed since the last time it was checked. When the graph remains the same for a certain number of times, the graph is considered stable and the loop detection mechanisms are triggered. This graph sampling interval used by IPM as well as the number of times the graph has to remain identical are configured by the user.

4 Experiments

In this section we demonstrate the ability of our system to automatically identify the structure of an application while it runs. Moreover, we demonstrate how this knowledge can be used to reduce the amount of tracing data collected by only keeping information from a few representative iterations.

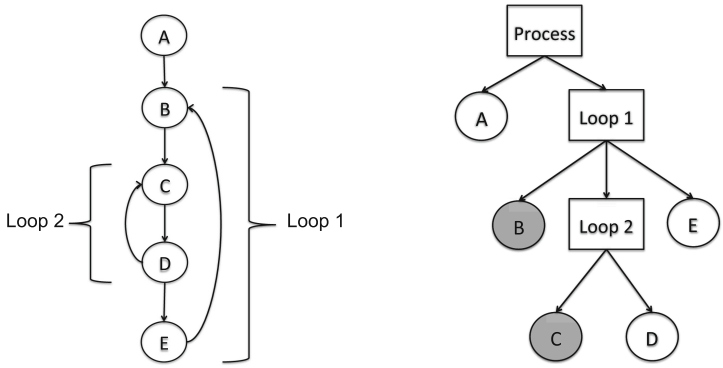
To this end, we run six different applications that represent typical scientific codes: MiniGhost, MiniFE and MiniMD from the Mantevo project [17]; BT and LU from the NAS Benchmarks [4]; and the GTC code [15]. The applications were run in a Cray XE6 machine with 2 twelve-core AMD MagnyCours CPUs at 2.1 GHz per node. The nodes had a total of 32 GB DDR3 memory and were interconnected through a Cray Gemini network. The benchmarks were compiled using Intel 12.1.5.

4.1 Overhead

Table 1 shows for each benchmark the percentage of overhead introduced by IPM over the total application running time. This overhead includes intercepting every MPI call, building the graph, and detecting the cycles in it. As can be seen in the table, the applications are not perturbed much since the overhead is very small, being always under 2 %. In addition, this overhead does not increase with the number of cores used as the graph creation and structure detection mechanism are performed locally without any inter-process communication required.

4.2 Usage Scenario

Collecting fine-grained information with tracing tools avoids the loss of microscopic information that can occur with other summarization methods such as



(a) Graph with two loops, one of them nested.

(b) Hierarchical loop structure tree. Grey nodes are loop headers.

Fig. 4. Event flow graph and its hierarchical tree representation after loop detection.

profiling. However, detailed trace-based analysis for the whole lifetime of an application is infeasible due to the scalability problems caused by the amount of data generated.

Nevertheless, most MPI scientific applications are usually iterative algorithms that repeat the same operations over time as the simulation evolves. Given this iterative nature, applications exhibit a similar performance behavior across iterations during their lifetime. Thus, keeping information on only a few iterations of such a stable region should be sufficient to capture the overall application behavior.

By means of detecting the application structure, we aim to identify at runtime the repetitive pattern of the program, thereby, collecting information on a few representative iterations only. This process works as follows. First, the corresponding event flow graphs are built when the application reaches a stable state. In other words, when the number of graph nodes remains stable and does not change. Once the event flow graph has been built for each process, each graph is analyzed to automatically detect and label its cycles. Then, IPM creates for each graph one tree that depicts the hierarchical relation of the loops detected. The root of the tree represents the process and every internal tree node corresponds to a loop or an event in such a process. If two loops are nested, they will be parent and child in the tree. These trees model naturally the hierarchical relations among loops and allow IPM to check easily, among other things, if an event belong to a certain loop, if a loop is nested within another, or if two loops are nested within the same loop. Figure 4 presents an event flow graph with two loops and how this information would be represented within IPM after the loop detection.

As previously stated, the loop detection mechanisms are triggered once the application enters into a stable state, that is, when the application starts iterating over its main simulation loop. Therefore, in order to monitor some iterations of

Table 2. Size comparisons between full traces and traces with 10 selected iterations.

Metric	MiniGhost	MiniFE	GTC	MiniMD	BT	LU
Ranks	96	144	64	192	144	128
Total iterations	60	200	200	2000	250	300
Total trace size	26 MB	77 MB	48 MB	555 MB	717 MB	7.7 GB
10 iterations size	4.4 MB	4.1 MB	1.3 MB	788 KB	29 MB	267 MB
% reduced	83 %	94.7 %	97.3 %	99.8 %	96 %	96.53 %

that main loop, IPM only has to wait for the event that is the loop header of the current outermost loop being executed. In other words, an event that is the header of a loop that hangs from the root of the constructed tree. Once this event is intercepted, IPM starts the tracing to collect detailed information for a certain number of iterations defined by the user. Afterwards, IPM stops the tracing and the application continues its execution normally.

Table 2 shows a comparison of sizes between a full trace and a trace with a few selected iterations for our various test applications. The table contains the number of processes used, the total number of iterations for the full test case, the total trace size for the full test, the trace size when tracing automatically only 10 iterations, and the percentage of trace size reduction achieved. As can be observed in the table, by keeping information on only a few iterations, we can reduce the final trace size up to several orders of magnitude.

The current approach leaves room for some improvements though. For instance, IPM could take into account some loop performance metrics before turning on the tracing. Checking metrics such as instructions per cycle (IPC) across iterations of the outermost loop could guarantee even more that the application has reached its stable state. At the moment, the loop detection mechanism and the selective tracing are triggered only once during the whole lifetime of the application. Therefore, in cases where applications have several phases or various outermost loops, our methodology will trace only one of them. It is planned in our future work to solve this issue by triggering the selective tracing few times during the execution, as well as providing the possibility to trigger the loop detection explicitly with an API.

Although tracing just a few iterations provides detailed information while reducing the amount of data collected, it always comes with an inevitable data loss. Specially in punctual variations between iterations. Therefore, we performed several experiments to measure the quality of our results, that is, we examined how representative from the overall execution are the iterations automatically selected by IPM. With that in mind, we used the CrayPat performance tool to collect several statistics about the most important functions in MiniGhost. Then we computed the same statistics from the reduced trace that contained only 10 iterations. Those statistics are the percentage of time spent in each call, and the average of instructions and cycles per call. Table 3 compares the measurements obtained with Craypat for the whole run with the measurements obtained from

Table 3. Statistics per call for the most relevant functions in MiniGhost.

	CrayPat			IPM Trace		
Function name	Time %	Kinstr	KCycles	Time %	Kinstr	KCycles
MG_BSPMA_DIAGS	5.70 %	401,476	558,737	5.42 %	394,140	552,064
MG_STENCIL_3D27PT	80.8 %	120,886	199,126	79 %	120,864	199,397
MG_ALLREDUCE_SUM	12.10 %	14,052	29,822	12.96 %	14,432	29,818

the automatically reduced trace. As can be seen in the table, the differences are very small (always under 2 %) and could be explained due to the different overheads introduced by both tools, or by small variances between executions or even across iterations. In any case, the results demonstrate that the trace containing only a few selected iterations is representative of the overall behavior of the application.

5 Related Work

Detection and analysis of parallel application structure is the topic of several related works. The ScalaTrace [18] framework provides on-the-fly lossless trace compression of MPI communication traces by detecting loops, or repeating events, and encoding them using RSDs [11]. Our approach differs in the fact that whereas ScalaTrace detects loops for trace compression, our on-line loop detection has more general purposes, from statistic aggregation to data filtering. Our solution is highly customizable, allowing the generation of compressed full traces and small uncompressed fine-grained traces.

The work of Gonzalez et al. [9,10] works on two-dimensional hardware counter data derived from computational bursts, and employs a density based clustering approach to identify the SPMD structure of the application. Although this approach allows to reduce the size of traces by collecting only relevant information from a few iterations, it has no precise control over which part of the code corresponds to a certain traced region. In contrast, our approach provides fine-grained precision in delimiting loops within the application. In addition, the use of burst clustering demands a more complex parallel software infrastructure to be used in an on-line scenario [16].

The work of Casas et al. [6,7] utilizes spectral analysis techniques such as wavelets to unveil the inner structure of parallel programs in performance traces. Thereby, generating sub-traces that only contain a few representative iterations. In addition, the tool can also find regions within the trace that are not usable due to tool perturbation, e.g. flushing of tracing buffers to disk. Although this is a good approach that helps the user to focus in relevant parts of the application while reducing the trace size, it still requires the whole original post-mortem trace. In contrast, our approach can be performed on-line while the application runs.

An approach that puts more focus on the communication structure is followed by [3,19]. Repeated communication patterns are here first identified locally (on a single process) and then grown globally by using string processing techniques such as n-gram detection and suffix trees. The work of Alawneh et al. [3] further attempts to group repeated patterns into homogeneous phases using information theory concepts.

AutomaDeD [5,14] has similarities to our approach in that the application execution is also represented as a set of states and the transitions between them. However, AutomaDeD focuses on debugging and only records transition propabilities between the states to create a Semi-Markov Model (SMM) of the application execution. In contrast, our event flow graphs record the actual program execution, allowing us to reproduce exactly the full sequence of events ordered in time.

6 Future Work

Our current implementation captures information from several consecutive iterations only once during the lifetime of an application. Nevertheless, this mechanism can be easily extended to acquire information with more advanced strategies. For instance, every time a certain condition such as the variation of a particular performance parameter is fulfilled. IPM could keep track of metrics such as instructions per cycle (IPC) on an iteration basis, and then trigger the tracing every time there is a noticeable change of such a metric. Thereby, if an application degrades during a long job we can have fine-grained snapshots at several points in time. Furthermore, we want to extend the iteration selection mechanism in order to automatically detect when an application has irregular loop behavior or combined repetitive loop patterns. That is, the sequence of events executed by the application is not regular and it changes from time to time regarding current loop iteration, program state, or simulation phase. Our current solution for selective tracing generates always a fixed number of consecutive iterations, therefore, we can lose irregular loop patterns if their frequency of appearance is smaller than the fixed number of iterations traced.

In the present work, we have shown how we discover the structure of an application across the time dimension, that is, detecting patterns (loops) in the sequence of events performed by each process. However, our ongoing work is also directed towards investigating the structure of applications across the process dimension. We are studying the utilization of graphs to build a process signature that could be clustered to detect processes with the same program behavior. Thereby, we could reduce even more the amount of data collected as only data from a few representative tasks could be kept.

Our current event flow graphs are focused in pure MPI applications, however, with the increase in the number of cores within computer nodes, hybrid approaches such as MPI+OpenMP or MPI+PGAS are becoming more usual. Thus, we want to provide our graphs with extensions to model such situations, for instance, having new graph nodes that represent OpenMP regions,

or PGAS operations. Moreover, we want to study the utilization of graphs with non-iterative applications, for instance, recursive codes or applications with task-based parallelism.

7 Conclusion

This paper presents the use of event flow graphs together with cycle detection algorithms to automatically detect the loop nesting structure of MPI parallel applications. This loop structure can be extracted from any MPI program without recompilation or modification of the source code.

We demonstrate how our work can be used, for instance, to automatically compute post-mortem statistics that help users to better understand their applications, e.g., the distribution of time across loops, or the percentage of MPI time spent in a certain loop. Nevertheless, the greatest strength of our structure detection approach is that it can be performed with very low overhead while the application runs. To that end, we present a test case where the structure of a stencil code is extracted on-line while the program runs to intelligently filter the performance data collected. By knowing the loop structure of an application, our framework traces automatically only a small fraction of representative iterations, reducing considerably the amount of data collected while keeping the overall performance behavior of the application. Furthermore, the overhead introduced by our mechanism is very small, being always under 2 % in our experiments.

References

1. Aguilar, X., Fürlinger, K., Laure, E.: MPI trace compression using event flow graphs. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 1–12. Springer, Heidelberg (2014)
2. Aguilar, X., Fürlinger, K., Laure, E.: Visual MPI performance analysis using event flow graphs. Procedia Comput. Sci. **51**, 1353–1362 (2015). International Conference On Computational Science, ICCS 2015 Computational Science at the Gates of Nature
3. Alawneh, L., Hamou-Lhadj, A.: Identifying computational phases from inter-process communication traces of HPC applications. In: 2012 IEEE 20th International Conference on Program Comprehension (ICPC), June 2012, pp. 133–142 (2012)
4. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The NAS parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (1991)
5. Bronevetsky, G., Laguna, I., Bagchi, S., de Supinski, B.R., Ahn, D.H., Schulz, M.: AutomaDeD: automata-based debugging for dissimilar parallel tasks. In: Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, 28 June - 1 July 2010, pp. 231–240 (2010)
6. Casas, M., Badia, R.M., Labarta, J.: Automatic structure extraction from MPI applications tracefiles. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 3–12. Springer, Heidelberg (2007)

7. Casas, M., Badia, R.M., Labarta, J.: Automatic phase detection and structure extraction of MPI applications. Int. J. High Perform. Comput. Appl. **24**(3), 335–360 (2010)
8. 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. LNCS, vol. 6043, pp. 218–227. Springer, Heidelberg (2010)
9. Gonzalez, J., Gimenez, J., Labarta, J.: Automatic detection of parallel applications computation phases. In: IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009, May 2009, pp. 1–11 (2009)
10. Gonzalez, J., Huck, K., Gimenez, J., Labarta, J.: Automatic refinement of parallel applications structure detection. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops Ph.D. Forum (IPDPSW), May 2012, pp. 1680–1687 (2012)
11. Havlak, P., Kennedy, K.: An implementation of interprocedural bounded regular section analysis. IEEE Trans. Parallel Distrib. Syst. **2**, 350–360 (1991)
12. Havlak, P.: Nesting of reducible and irreducible loops. ACM Trans. Program. Lang. Syst. (TOPLAS) **19**(4), 557–567 (1997)
13. IPM WWW site: http://www.ipm2.org
14. Laguna, I., Gamblin, T., de Supinski, B.R., Bagchi, S., Bronevetsky, G., Anh, D.H., Schulz, M., Rountree, B.: Large scale debugging of parallel tasks with AutomaDeD. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 50:1–50:10. ACM, New York (2011)
15. Lin, Z., Hahm, T.S., Lee, W., Tang, W.M., White, R.B.: Turbulent transport reduction by zonal flows: Massively parallel simulations. Science **281**(5384), 1835–1837 (1998)
16. Llort, G., Gonzalez, J., Servat, H., Gimenez, J., Labarta, J.: On-line detection of large-scale parallel application's structure. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–10. IEEE (2010)
17. Mantevo Project: http://mantevo.org
18. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: ScalaTrace: scalable compression and replay of communication traces for high-performance computing. J. Parallel Distrib. Comput. **69**(8), 696–710 (2009)
19. Preissl, R., Kockerbauer, T., Schulz, M., Kranzlmuller, D., Supinski, B., Quinlan, D.: Detecting patterns in MPI communication traces. In: 37th International Conference on Parallel Processing, 2008. ICPP 2008, September 2008, pp. 230–237 (2008)
20. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. (TOPLAS) **21**(2), 175–188 (1999)
21. Sreedhar, V.C., Gao, G.R., Lee, Y.F.: Identifying loops using DJ graphs. ACM Trans. Program. Lang. Syst. (TOPLAS) **18**(6), 649–658 (1996)
22. Stanier, J., Watson, D.: A study of irreducibility in C programs. Softw. Pract. Experience **42**(1), 117–130 (2012)
23. Tarjan, R.: Testing flow graph reducibility. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, pp. 96–107. ACM (1973)
24. Wei, T., Mao, J., Zou, W., Chen, Y.: A new algorithm for identifying loops in decompilation. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 170–183. Springer, Heidelberg (2007)