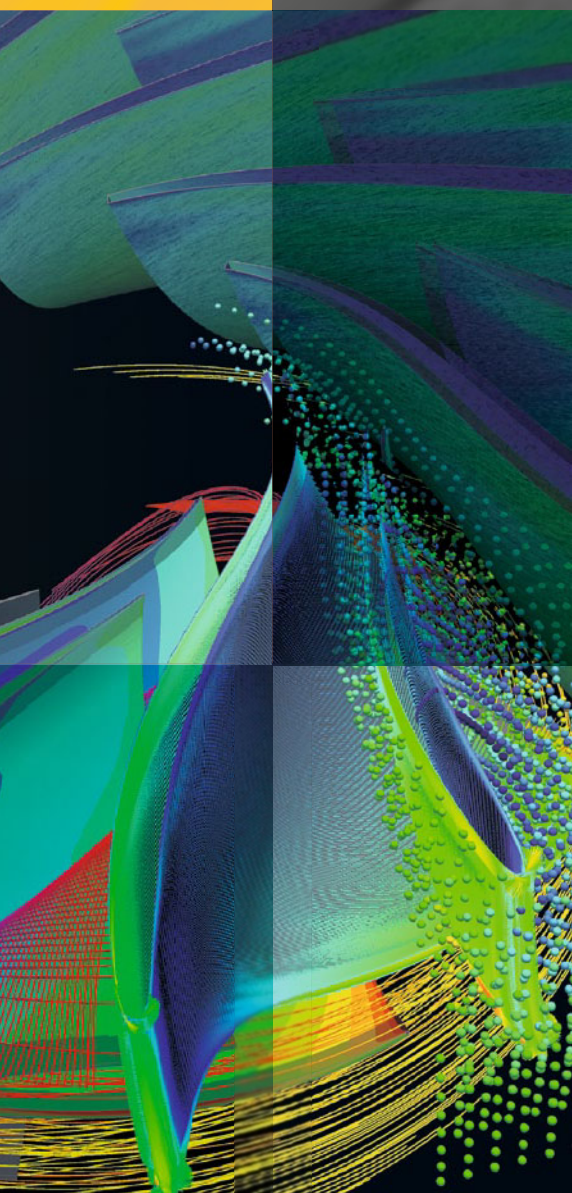


Alexey Cheptsov · Steffen Brinkmann
José Gracia · Michael M. Resch
Wolfgang E. Nagel
Editors

Tools for High Performance Computing 2012



H L R I S

 Springer

Tools for High Performance Computing 2012

Alexey Cheptsov • Steffen Brinkmann
José Gracia • Michael M. Resch
Wolfgang E. Nagel
Editors

Tools for High Performance Computing 2012

 Springer

Editors

Alexey Cheptsov
Steffen Brinkmann
José Gracia
Michael M Resch
Höchstleistungsrechenzentrum,
Stuttgart (HLRS)
Universität Stuttgart
Stuttgart
Germany

Wolfgang E Nagel
Zentrum für Informationsdienste, und
Hochleistungsrechnen (ZIH)
Technische Universität Dresden
Dresden
Germany

Front cover figure: Simulation of waterflow through a Francis-runner. Illustration by Stellba Hydro GmbH & Co KG, Eiffelstr. 4, 89542 Herbrechtingen, Germany.

ISBN 978-3-642-37348-0 ISBN 978-3-642-37349-7 (eBook)
DOI 10.1007/978-3-642-37349-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013940607

Mathematics Subject Classification (2010): 68M20, 65Y20, 65Y05

© Springer-Verlag Berlin Heidelberg 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The latest advances in the high-performance computing (HPC) hardware, such as increased capabilities of a single NUMA node or heterogeneous architectures combining traditional CPU nodes with accelerators, have significantly raised the level of principally available compute performance. At the same time, the growing hardware capabilities of modern supercomputing architectures have caused an increasing complexity of the parallel application development technology. While a number of new programming paradigms, e.g., task-based parallelization and data-driven programming frameworks, have been introduced to fully exploit the available compute resources, very little has been done in terms of tools for performance optimization and debugging for new programming models nor for the latest generation of hardware.

Despite numerous efforts to improve and simplify application development, there is still a lot of manual tuning work required in order to take full advantage of modern HPC architectures. The process of identifying and eliminating performance issues, ranging from simple memory leaks to inefficient design of communication patterns, is very difficult, unless special tools are used. The HPC tools for debugging, performance analysis, and optimization of parallel applications make a major contribution to development of the robust and efficient parallel software.

In order to enable a technology exchange and cross-fertilization in the optimization techniques and development approaches across the HPC tools' developers, the Center for Information Services and High Performance Computing of the University of Dresden (ZIH-TUD)¹ and the High-Performance Computing Center Stuttgart (HLRS)² jointly organize the International Parallel Tools Workshop. The workshop is an annual event, which addresses challenges in parallel software performance assurance and discusses novel trends in HPC tools development.

The workshop has two major goals. The first is serving as discussion forum for tool developers on the latest advances in performance analysis techniques and

¹http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/

²<http://www.hlrs.de/>

software technologies for them. Approaches of eliminating typical performance issues in complex application scenarios by coupling techniques used in different tools were of a special interest for the last workshop's edition. The second goal is to offer the users of parallel tools a unique opportunity to gain a consolidated outlook on state-of-the-art HPC tools. The workshop has proved successful among the application providers, who get an opportunity to have a discussion with the other developers sharing similar performance issues or establish new contacts with tools' developers. On the other hand, the users' feedback helps tools' developers define obstacles to newly raising performance issues and identify engineering or research approaches to overcome them.

This book comprises a continuation of a successful series of publications that started in 2007. It contains contributed papers presented at the 6th International Parallel Tools Workshop,³ held 25–26 September 2012 in Stuttgart, Germany. The workshop's audience represent leading scientific and industrial organizations worldwide. The presentations covered different aspects of the software optimization, ranging from parallel debugging to complex performance data visualization technology. More than ten different tools were addressed in workshop presentations or hands-on tutorials.

Along with the newest features of the well-known tools, such as Vampir (a performance analysis framework for a wide range of parallel applications) or DDT (a debugging framework with a big set of extensive analysis features), the book introduces new tools which were presented for the first time in the Parallel Tools series, e.g., Temanejo (a debugging environment for StarSs) or MemPin (an automatic memory detection tool for MPI applications). The book's material is organized in four sections: Debugging, Automatic Error Detection, Performance Analysis and Optimization, and Performance Data Visualization.

We believe that the presented material offers a comprehensive outlook on the mainstream application analysis and optimization technology in the high-performance computing domain for both categories of readers – parallel tools' developers and developers of parallel applications.

Stuttgart, Germany
February 2013

Alexey Cheptsov
Steffen Brinkmann
José Gracia
Michael M. Resch
Wolfgang E. Nagel

³<http://toolsworkshop.hlr.de/2012/>

Contents

Part I Debugging

Debugging at Scale with Allinea DDT	3
David Lecomber and Patrick Wohlschlegel	
1 Why Scalability Matters for Debugging	3
2 The Ability to Debug at-Scale Changes Everything	4
3 How Allinea DDT Helps to Fix Bugs	5
4 Understanding Multiple Processes	5
5 Simple and Effective Process Control	6
6 Smart Highlighting and Sparklines	7
7 Searching Data Sets	8
8 Visualizing Large Data Sets in Real-Time	9
9 Deadlocks	10
10 Memory Debugging	11
11 Summary	11
12 Further Readings	12
Task Debugging with TEMANEJO	13
Steffen Brinkmann, José Gracia, and Christoph Niethammer	
1 Introduction	13
2 What Debugging Means in the Context of Task-Based Parallelism	15
3 The Debugging Process	16
3.1 Communication	16
3.2 Graph Display	17
3.3 Execution Control	20
4 Conclusion	20
References	21

Part II Automatic Error Detection

MPI Runtime Error Detection with MUST: Advanced Error Reports	25
Joachim Protze, Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, Matthias S. Müller, and Wolfgang E. Nagel	
1 Introduction	25
2 MUST	27
3 Shortcoming of Current Error Views	29
3.1 Example 1: Pinpointing Deadlocks	29
3.2 Example 2: Viewing Datatype Related Problems	31
4 Deadlock View in MUST	31
5 Type Tree View	33
6 Related Work	35
7 Conclusion	36
References	37
Advanced Memory Checking for MPI Parallel Applications Using MemPin	39
Shiqing Fan, Rainer Keller, and Michael Resch	
1 Introduction	40
2 Overview of Intel Pin	41
3 Design and Implementation	42
3.1 MemPin	43
3.2 Integration of MemPin with Open MPI	44
4 Memory Checks in Parallel Application	46
4.1 Pre-communication Checks	46
4.2 Post-communication Checking	47
5 Performance Comparison	48
6 2D Heat Conduction Program with MemPin	49
7 Conclusion	51
References	52

Part III Performance Analysis and Optimization

Generic Support for Remote Memory Access Operations in Score-P and OTF2	57
Andreas Knüpfer, Robert Dietrich, Jens Doleschal, Markus Geimer, Marc-André Hermanns, Christian Rössel, Ronny Tschüter, Bert Wesarg, and Felix Wolf	
1 Introduction	57
2 Overview of Score-P and OTF2	58
2.1 The Score-P Instrumentation and Measurement System	59
2.2 The Open Trace Format 2	59
2.3 Existing Event Record Types	60
2.4 Current and Future Directions	60

- 3 RMA in HPC Parallel Programming 61
 - 3.1 RMA Operations in HPC Parallelization Libraries 61
 - 3.2 Concepts in RMA Parallelization Models 62
- 4 Generic RMA Event Types 64
 - 4.1 RMA Window Handling 64
 - 4.2 Specification of the Passive Side 65
 - 4.3 Get and Put 65
 - 4.4 Atomic RMA Operations 66
 - 4.5 Completion Records 67
 - 4.6 Notification via RMA 67
 - 4.7 Synchronization 68
 - 4.8 Collective Operations and Synchronization 69
 - 4.9 Locking of Resources 69
- 5 Example Cases with RMA Event Types 70
- 6 Conclusions and Outlook 72
- References 73

Cache-Related Performance Analysis Using Rogue Wave Software’s ThreadSpotter 75

Royd Lüdtké and Chris Gottbrath

- 1 Introduction 75
- 2 Basic Overview on Caching 76
 - 2.1 Motivation for Caching 76
 - 2.2 Cache Architectures 76
 - 2.3 Cache Organization 77
 - 2.4 Prefetching 77
 - 2.5 Eviction of Cache Lines (Replacement Policies) 77
 - 2.6 Complexity Added by Coherence 78
 - 2.7 Important Statistics 78
 - 2.8 Optimal Cache Utilization 79
 - 2.9 What Performance Improvement Is Possible When Optimizing an Application’s Cache Utilization? 79
- 3 ThreadSpotter: A Statistical Approach for Cache-Related Profiling 80
 - 3.1 Different Approaches of Cache Related Performance Analysis 80
 - 3.2 What Kind of Data Is ThreadSpotter Looking at in Order to Create a Report? 81
 - 3.3 Sampling an Application 81
 - 3.4 Report Generation 83
 - 3.5 Presenting Optimization Opportunities That ThreadSpotter Discovers 85
- 4 Types of Performance Optimization Opportunities Discovered by ThreadSpotter 89
 - 4.1 What Kind of Cache-Related Opportunities for Performance Optimization Can Be Discovered by ThreadSpotters Statistical Approach? 89
 - 4.2 Reuse 91

4.3	Non-temporal Data	92
4.4	Cache Hot Spots	92
5	Conclusion	93
	References	93
	Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite	95
	Holger Brunst and Matthias Weber	
1	Introduction	95
2	Heat Map Overlay for the Master Timeline	97
3	Customization of Performance Metrics	100
3.1	Delayed MPI Communication	101
3.2	Conditional Floating Point Performance	101
4	Refinement of Invocation Graph	102
4.1	Rules	104
4.2	Examples	105
5	Comparison of Multiple Program Runs	110
5.1	Multiple Program Executions at a Glance	111
5.2	Alignment of Multiple Trace Files	112
6	Conclusion	114
	References	114
	Extending Scalasca's Analysis Features	115
	Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube, and Zoltán Szebenyi	
1	Introduction	115
2	Root Cause Analysis	117
3	Critical Path Analysis	118
4	Time-Series Profiling	120
5	Topologies	122
5.1	Hardware Topologies	123
5.2	Processes X Threads	124
5.3	Runtime Mapping	124
5.4	Algorithm Domain	124
6	Future Work	125
	References	125
	The HOPSA Workflow and Tools	127
	Bernd Mohr, Vladimir Voevodin, Judit Giménez, Erik Hagersten, Andreas Knüpfer, Dmitry A. Nikitenko, Mats Nilsson, Harald Servat, Aamer Shah, Frank Winkler, Felix Wolf, and Ilya Zhukov	
1	Introduction	128
2	The HOPSA Workflow	129
2.1	Overview	129
2.2	Performance Screening	130

- 2.3 Performance Diagnosis 132
- 2.4 The HOPSA Performance Tools 136
- 2.5 Integration Among Performance Analysis Tools 141
- 2.6 Integration of System Data and Performance Analysis Tools 142
- 2.7 Opportunities for System Tuning 144
- 3 Conclusion 145
- References 145

Part IV Performance Data Visualization

- Visualizing More Performance Data Than What Fits on Your Screen..... 149**
- Lucas M. Schnorr and Arnaud Legrand
- 1 Introduction 149
- 2 Motivation and Discussion 151
- 3 Multi-scale Trace Aggregation for Visualization 153
- 4 Visualization Techniques 155
 - 4.1 Squarified Treemap View 155
 - 4.2 Hierarchical Graph View 157
- 5 The Viva Visualization Tool 159
- 6 Conclusion 159
- References 160

Part I

Debugging

Debugging at Scale with Allinea DDT

David Lecomber and Patrick Wohlschlegel

Abstract As core counts in HPC clusters grow, almost every application user is trying to run software at higher scale than before. It is not always an easy task and can end in failure as the limitations of existing software are discovered. Users and developers quickly find new (and old) bugs as scale increases: software can be complex when it seeks to use more threads or processes to exploit the hardware. In this document, we show how using a debugger at the scale of the bug is the most effective way to tackle parallel software problems today. We introduce Allinea DDT – the world’s only scalable parallel debugger – and show how it is fast, capable, and lets you debug your parallel or multithreaded application, no matter how big or small a system you use, easily. Allinea DDT has been setting standards for usability for many years and has torn up scalability records. It is used on the world’s largest systems – debugging over 220,000 processes simultaneously in some cases. Bugs can be fixed easily for all developers – not just those with extreme scale – by using Allinea DDT at your scale.

1 Why Scalability Matters for Debugging

Studies of job failure on larger HPC systems have shown that software problems account for a significant proportion of failures.

Increasingly errors appear at higher scales: exhaustive testing at high scale is often infeasible due to the cost of machine access, or lack of machine access. There are regularly differing development and production environments. This means that errors often occur at scale in production environments – and fixing them is a very high priority.

D. Lecomber · P. Wohlschlegel (✉)
The Innovation Centre, Allinea Software Ltd, Warwick Technology Park, Gallows Hill,
Warwick CV34 6UW, UK
e-mail: patrick@allinea.com

Software developers are used to trying many tricks to find software bugs – and no trick is more common than inserting a quick print statement. This technique can be revealing – but it is one that can scarcely handle a small parallel cluster, let alone 1,000 cores! The difficulty of knowing where to place the print statements in the first place leads to a repetitive task – but the killer problem for debugging larger parallel jobs is making sense of the interleaved print outputs of multiple processes – and this gets much worse as the number of cores increases.

Another approach is to try and make the bug appear at a smaller scale – but this is often an impossible task. Real users have discovered scale-related defects whenever applications are scaled to a higher level of concurrency – and then that it is difficult to observe such defects at reduced scales. Some bugs just do not exist at small scale – or can't be found in smaller datasets. If a stable application suddenly fails when moving to a larger system, it is unlikely that a smaller test will exhibit the issue.

Whilst many bugs are not random, many are and can be the hardest to track down. Developers familiar with array overruns or memory corruption will recognize the random consequences of such errors, and those with multithreaded applications or MPI applications with point-to-point communications or RDMA, will know that timing and ordering of events is a major source of random behaviour.

A bug that is random may fail to occur frequently enough to reproduce it on a job of say, 16 cores, but on 128 cores, it would be more likely to happen just by the cumulative effect of probability or the increase in the number of permutations of event orderings. So, again, reducing the scale of the application doesn't help in fix the problem.

We have seen that scale-related bugs are observed to occur frequently and that two of the tools in the developer's bug-fixing toolbox are just not able to deliver the help required as scale increases.

You might want to spend days or weeks trying to reproduce at a smaller scale or deciphering print output – or you could get straight to the problem by debugging at scale.

2 The Ability to Debug at-Scale Changes Everything

Allinea DDT is a graphical parallel debugger – used by many scientific computing centres, universities and corporations to help in the everyday task of finding and fixing bugs, from single process workstations through to the very largest supercomputers.

It has many features not present in ordinary debuggers – such as memory debugging, data visualization and support for the many MPI and OpenMP implementations that are used by parallel software developers. It also has an interface that makes debugging easy, at any scale.

With Allinea DDT, it's possible to take debugging to as high a scale as you want to take your application – it is there and it is fast! For the first time in

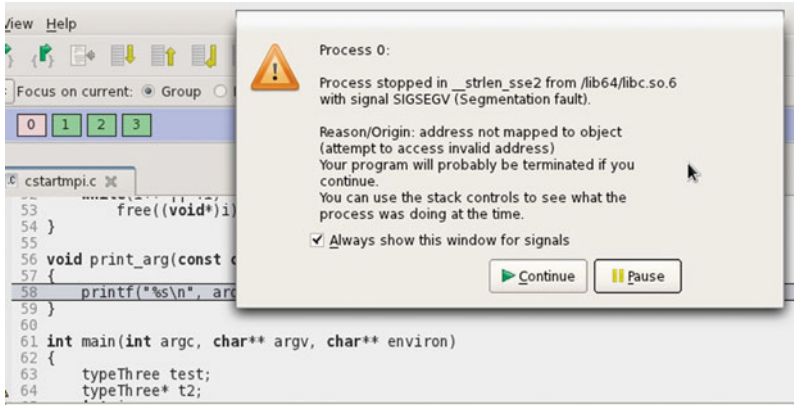


Fig. 1 A typical application crash

history, a debugger with performance logarithmic in the number of cores is giving fraction-of-a-second performance for global (or subset) operations on applications running all the way up from the humble workstation to Petascale.

3 How Allinea DDT Helps to Fix Bugs

Let’s start by looking at how Allinea DDT can help in some of those straightforward cases – like a simple crash. Typically, when an application crashes on a HPC system, very little information is available – the job is cleaned up and all trace of the cause removed. It’s usual not to generate core files – as most filesystems would not want multi-gigabyte core files being written by every process on application crash! With a bit of luck, there would be some output, perhaps some print statement that could explain roughly where the job had reached before it crashed – but beyond that, it would all be guesswork!

With Allinea DDT, the bug fixing is easier. Run the job inside Allinea DDT at the scale of the problem, and when the job crashes, this is what you see (Fig. 1).

It’s immediately clear exactly where the crash happened – to the exact line, and exactly which processes. The error message conveys exactly what the problem is, and the source code is highlighted to tell you where things happened.

4 Understanding Multiple Processes

Sometimes just knowing the source file and line is enough – but often you will need to see the bigger picture. Allinea DDT’s Parallel Stack View enables a rapid identification of where processes and threads are. It displays the stacks of every

Stacks (All)	
Processes	Function
150120	_start
150120	__libc_start_main
150120	main
150120	pop (POP.f90:81)
150120	initialize_pop (initial.f90:119)
150120	init_communicate (communicate.f90:87)
150119	create_ocn_communicator (communicate.f90:300)
1	create_ocn_communicator (communicate.f90:303)

Fig. 2 A parallel stack view of 150,120 cores

process in the application in a tree view – with alike stacks merged together and a process counter shown.

This component does not become any more complex as the process or thread counts increase, as you can see (Fig. 2). This is also one of the bedrocks of hybrid debugging, where it is still of great value even as the number of threads is increasing by many orders of magnitude. It's not unusual to see 20,000 threads in a single GPU device!

One of the great things about Allinea DDT is its speed in giving you this information – Allinea DDT takes a fraction of a second to report where every thread is in a job on the largest machines in the world, which means even if you are using of a few hundred cores you can be confident of incredibly responsive debugging. If a crash like this happens at 16 cores, 128 cores or 200,000 cores, it'll still be quick to fix.

Debugging is not just about the speed and responsiveness of the debugger, it's also about how to let the user see and control what's happening, easily. At scale this is extremely important.

5 Simple and Effective Process Control

Not all bugs are as simple as a crash – there could instead be incorrect output, say. In these cases it is helpful to step through an application and to watch progress unfold by manually controlling the application with a debugger – perhaps running first to a breakpoint in a known good location.

With Allinea DDT, you can quickly jump to a location in the source code, and a simple one click command will let your application run all the processes to that location.

Playing, pausing or stepping groups of processes at scale is scarcely different when working on a workstation or on a cluster. Source code highlighting and the parallel stack view both scale to the task very well.

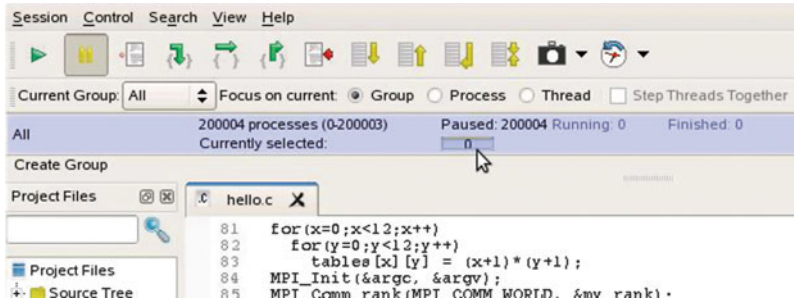


Fig. 3 Summary view of process status

If you’ve seen Allinea DDT before, perhaps running at smaller job sizes, you might remember the graphical tools Allinea DDT used to tell you that processes are paused or which are still playing. But you might ask, how can Allinea DDT tell you such information about a very large job?

Yet again Allinea DDT has the answer: it has a scalable way to show you this – a summary view, as seen on (Fig. 3) – and the GUI automatically mutates to use this when you have more than 64 cores.

In just a glimpse you can see how many processes are busy – and if you hover the mouse over the Paused or Running numbers, Allinea DDT will tell you which processes they are scalably – showing 0,16,21–251, for example.

6 Smart Highlighting and Sparklines

One of the recent innovations in Allinea DDT is Smart Highlighting. Allinea DDT’s raw speed enables variables to be compared across processes automatically in negligible time.

Forgetting to check the error code of function calls within software is a regular source of bugs – it happens too easily – those system call errors seem so unlikely to happen at the time a client function is written – but they bite hard when they do error and the code doesn’t check! Smart Highlighting is ideal for this situation.

With Smart Highlighting, Allinea DDT compares variables across processes automatically every time processes pause after stepping or playing. It will then use colour highlighting when data changes or when different values occur on other processes. This means you see unintrusive hints that may be relevant to why a code has diverged.

Those unexpectedly returned error values we mentioned earlier are a great example of where this feature helps – it would not be practical for a user to repeatedly, manually, check for errors after every step through the code, but with the debugger colour-highlighting a difference, it quickly indicates something is wrong.

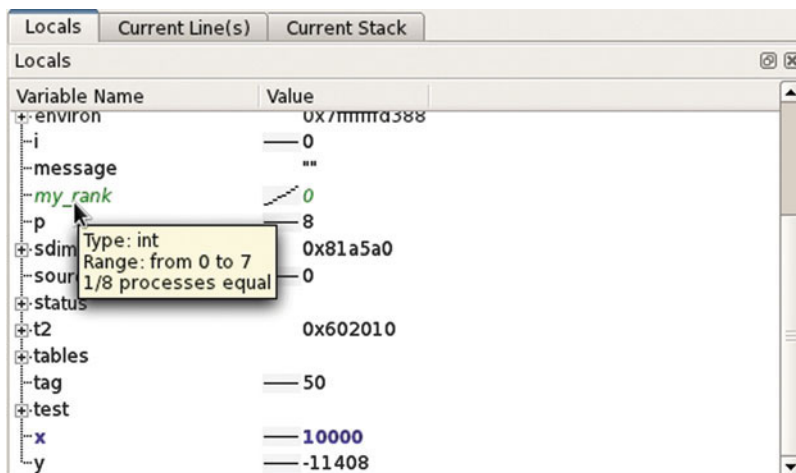


Fig. 4 Smart highlighting and sparklines, showing data change and process differences

We also added a fancy new feature called sparklines, which draws a tiny graph next to each variable in the interface. It compares its value across all the processes, instantly. As a consequence, you don't even need to consider each process one by one – just look at the picture (Fig. 4), and you will intuitively understand if something is going wrong.

7 Searching Data Sets

Large datasets can also be a challenge – with single errors rapidly propagating across the dataset and then across to other processes. For a long time Allinea DDT has given users a spreadsheet like view of arrays, with the ability to filter – to search for NaN or Inf, or some outliers.

This array viewing capability has been extended in Allinea DDT to give access to arrays distributed across multiple processes: It is simple to stitch together arrays distributed over a regular arrangement of processes, such as a 1, 2, or 3D process grid.

This feature lets you search for rogue elements across processes. Filtering data across many processes can be achieved in similar time to on a single processes. This can be seen on (Fig. 5). This is coupled with lazy evaluation where only the visible portion of the array is fetched to the GUI at any time, which means Allinea DDT is always responsive and never swamped.

There is also multi-process multi-dimension array viewing capability – and a built-in export capability to HDF5 and CSV. Exporting data is a theme in Allinea DDT: it is even possible to export, scalably, the stack traces in XML or CSV of

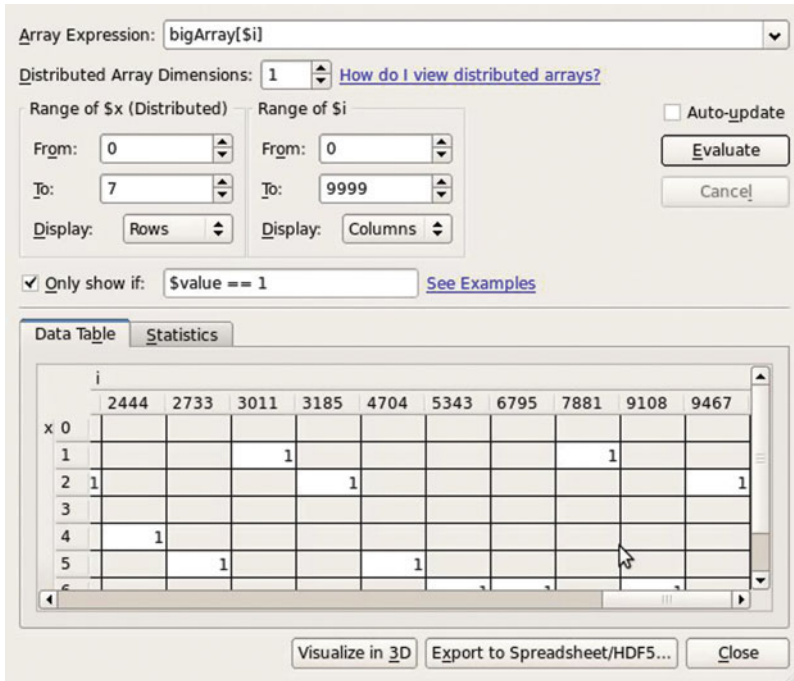


Fig. 5 Filtered search across multiple processes

processes from the Parallel Stack View we saw earlier. This is particularly useful for reporting bugs to other people!

8 Visualizing Large Data Sets in Real-Time

For scientists, Allinea DDT includes the integration of time-based distributed and scalable visualization tool called VisIT. This is illustrated in (Fig. 6). As there is no need to instrument the code or to write complex scripts, this feature allows easier access to data. Moreover, it integrates the ability to click back from observable troublespots to application processes.

Scientific simulations are inherently about the data, and the integration of full-strength scientific visualization within a debugging session is key to exploring – by visualizing – the behavior of data and its interaction with code in an application. VisIt is recognized by the major supercomputing centers as one of the most capable visualization packages for data intensive HPC.

If you are already using VisIt on your cluster, you are aware of how painful it can be to look at data: you either need to store huge data sets on your storage or to instrument your code using an external library to tell VisIt what to do. With this

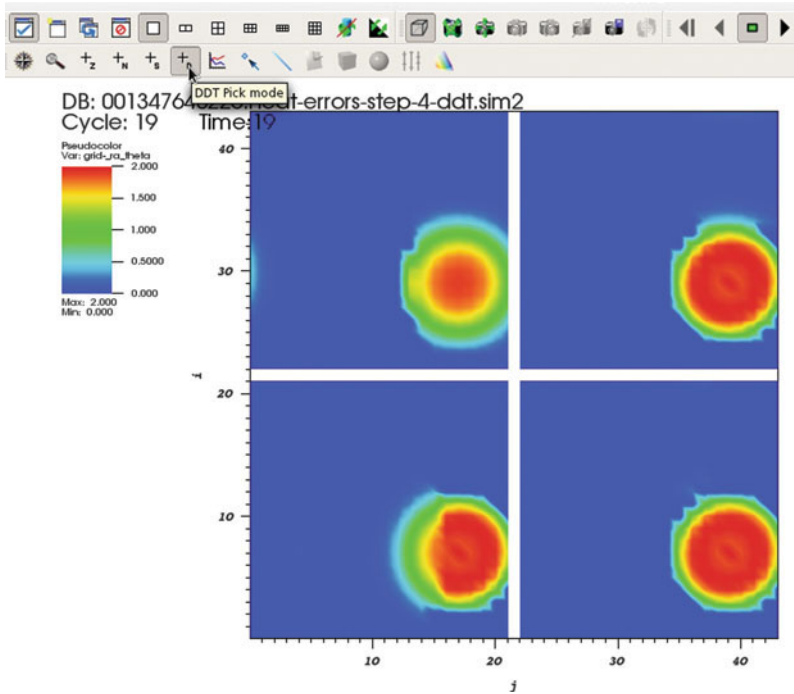


Fig. 6 Visualizing data at runtime and in parallel with VisIt

feature, this era is over: visualize a 1D, 2D or 3D array in real-time just as easily as you would look at it using the Multi-Dimensional Array Viewer. Allinea DDT will instrument the code with VisIt automatically for you!

9 Deadlocks

Deadlock is one of the easier bugs to track down with Allinea DDT. The symptom will be that an application appears to make no progress – perhaps failing to terminate in a timely manner, or the stream of progress output in the job log dries up.

At this point is a dilemma: cancel the job, and waste the cycles so far in the belief that it has locked up – or give the job another “five minutes” in the hope that it will get somewhere. The dilemma is readily solved by just attaching a debugger to the job. Allinea DDT is able to scalably attach to running jobs or subsets of a job – ranks can be specified as a subset such as “1,4,121–140”. Subsets can even be pseudo-randomly selected if the user wishes to just look at (say) one percent of the job.

On attaching, processes are paused and the location of processes, and the data values can be easily seen. This allows a quick decision to be made as to whether the application has hung – and if so where – or whether the application should continue. Detaching from the session will allow it to continue as normal.

10 Memory Debugging

The final class of bug we'll mention today is the memory problem. We don't mean a hardware problem, in this case we're referring to things like memory leaks – forgetting to deallocated memory – or reading beyond the end of an array. Such crashes can often appear to be quite random and can be difficult to detect.

Allinea DDT's memory debugging feature is able to help with these kinds of issues – and fixing this kind of bug is still important at scale. Many aspects of memory debugging translate to large scale very easily – for example checks for out of bound array access or for double deallocation of pointers are entirely parallel operations. Allinea DDT scalably merges and displays error messages when errors happen.

The random nature of these bugs means that being able to have the entire job under the control of a debugger is essential to not missing a crash – yet another reason to have a debugger that can cope with whatever scale you are trying to deploy applications at.

Memory leak detection is also there to help at any scale. Allinea DDT maintains information about the memory usage of the entire job – but crucially it finds the usage issues in a chosen current process, and in the worst processes (in terms of memory usage). This lets you interrogate the problem processes very easily – giving the scalability and focus that is needed to tackle the problem.

11 Summary

The same capabilities that users expect of single process debugging or when working with small clusters are still needed to find bugs at higher scale.

Without debuggers, users often “work blind” – and this is increasingly true as the limitations of print statements and other techniques to fix bugs become evident.

By introducing a debugger that can reach the scale at which users are having problems, Allinea is helping users to scale applications successfully. Features, such as Smart Highlighting help users to work quickly, and smartly at all scales.

With Petascale debugging now a reality, we continue to improve the existing performance and optimize where possible. We are also addressing challenges of debugging large scale hybrid systems of GPUs, and working to ensure that the scalability results so far will carry over as machines head towards Exascale.

12 Further Readings

- “Understanding failures in petascale computers”, B. Schroeder and G. A. Gibson, *Journal of Physics Conf. Ser.* 78, 2007.
- “Vrisha: Using Scaling Properties of Parallel Programs for Bug Detection and Localization”, B. Zhou, M. Kulkarni and S. Bagchi, *Symposium on High Performance Parallel and Distributed Computing (HPDC)*, June 2011.
- “Debugging Exascale: To heck with complexity, full steam ahead!”, B. Feldman, *The Exascale Report*, September 2010.

Task Debugging with TEMANEJO

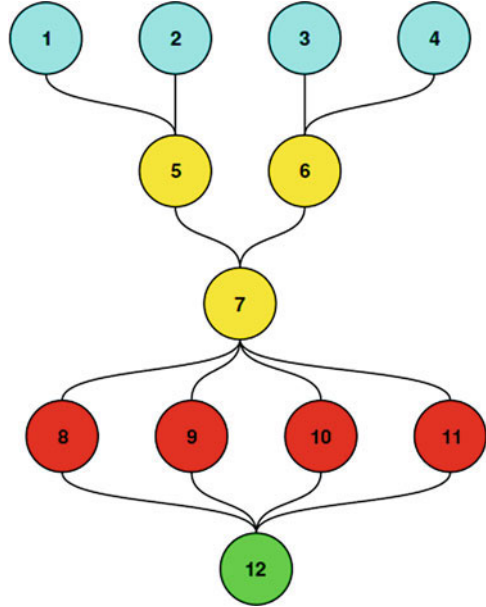
Steffen Brinkmann, José Gracia, and Christoph Niethammer

Abstract In recent years memory layouts have become more and more complex and bandwidth turned out to be the crucial performance parameter. This reflects in new programming paradigms which focus on data flow rather than instruction sequence. A very successful approach is StarSs, where the parallel programme consists of small computing units called *tasks* and dependencies between these tasks which are defined by the programmer. At runtime a dependency graph is created which determines the parallel or sequential execution of the tasks. When it comes to debugging StarSs applications, traditional debuggers such as gdb don't provide enough information and control to uncover shortcomings of the program. We present a new type of debugger which acts on the task level giving the user access to the dependency graph. Information is extracted from the running application with the lightweight library AYUDAME and the information is passed to the remote client TEMANEJO which visualises the dependency graph and passes user requests, such as blocking or prioritising a task, to the application.

1 Introduction

Due to the complexity of parallel programming and the different approaches to parallelism, debugging has become increasingly difficult. The main problems are that (a) threads share resources such as signals, file descriptors and memory address space and (b) threads have to synchronise access to the resources to avoid race conditions. The rules controlling resource access and synchronisation can be explicitly set by the programmer or implicitly generated by the runtime environment

S. Brinkmann (✉) · J. Gracia · C. Niethammer
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart,
70550 Stuttgart, Germany
e-mail: brinkmann@hlrs.de

Fig. 1 Example task graph

(referred to simply as *runtime* hereafter) which controls the parallel execution of the application.

The implicit behaviour of parallel applications often causes problems in application development. What a runtime does is many times per se hard to understand, poorly documented and will almost certainly produce bugs that are hard to find. To avoid these problems many programmers increase the use of explicit synchronisation, e.g. in the form of barriers, which conflicts with the idea of parallel design and will decrease performance.

Many debuggers exist and many of them are capable of debugging multi threaded applications. Nevertheless all of these tools, being originally designed for single threaded programs, work on the base of threads and instructions. The natural unit is the *line of code*. Contrary to that many parallel programming models support so called *task parallelism*. A task in this context is a self contained piece of code with well-define input and output dependencies. Namely the growing family of StarSs compilers and libraries (e.g. SMPSs [1], OmpSs [2], StarPU [3], KAAPI [4] among others) are based on the task as their natural unit.

The tasks generally have implicit and explicit dependencies on other tasks forming a acyclic directed graph (*dependency graph* or *task graph* hereafter) which set the rules for executing the application in parallel (see Fig. 1). Tasks that do not depend on each other can in principle run concurrently. When a given thread runs a specific task is decided by the runtime, which also creates the tasks and dependencies, ergo the task graph.

In order to debug such an application it is necessary to add a new kind of debugger to the developers toolbox. This tool must visualise the task graph and

enable the programmer to interact with the application on the basis of tasks and dependencies.

We present the next version 0.9 of TEMANEJO [5], a tool for debugging task-parallel applications visually. It communicates with the runtime (presently SMPSS is supported fully, StarPU and OmpSs are under development) via the library AYUDAME which was written for that purpose.

In the following we will discuss the new approach to debugging imposed by task parallelism (Sect. 2). Thereafter we will describe how TEMANEJO and AYUDAME cope with these new necessities (Sect. 3) and finally draw our conclusions (Sect. 4).

2 What Debugging Means in the Context of Task-Based Parallelism

The debugging process of a task parallel application is best divided in three stages: (1) checking the code *without* tasks enabled, (2) checking the code with tasks enabled and executing the tasks in the order of creation on one thread and (3) checking the code with tasks enabled on multiple threads. This scheme must naturally be enhanced as more complex features such as running with various queues or a communicator thread are not taken into account.

For the first and second stage a debugger like `gdb` will suffice enabling the programmer to find bugs of the kind you would have in a serial program. Not so for the third stage. When running separate tasks on different threads two consecutive actions on one thread may (and generally will) be totally independent of each other. Choosing a thread in a `gdb` session for instance can provide valuable information about what is happening inside a task. But when the execution reaches the end of the task, the thread will proceed with a seemingly random and most probably unrelated other task.

Here the problem of the order of execution becomes apparent: **How can the task-parallel application be debugged as a whole?**

Another problem is not as obvious: While one thread is being debugged, i.e. halted and stepped through, what should the other threads do? Or more generically asked: **What is a breakpoint in task-based parallelism?**

The answer to the first question is seeing the dependency graph and stepping task-wise instead of instruction-wise.

The second question is far more challenging and allows for different solutions a few of which are roughly outlined in the following.

- Stop (= Block) one task and let the rest of the application run undisturbed. Arbitrarily many tasks can be marked with such a breakpoint.
- The whole application stops when a marked task is reached. *Stopping* can have two meanings in this context: Actually halting the whole execution, or finishing tasks which are already running but not launching any new tasks. We take the *task* as the smallest unit of an application, therefore we favour the second definition of *stopping*. Again, many tasks can be marked with such a breakpoint.

- The whole application stops due to a user request (e.g. a clicked button or other control widget). The above described ambiguity of *stopping* applies equally.
- The thread reaching a marked task stops. The rest of the program will run undisturbed with one thread less.
- Block a task conditionally, i.e. blocking it until a specific event occurs. Usually the event will be finishing another task. This way one can add dependencies to the application's task graph while running the application.

Each of these strategies can be very useful in some situations and totally meaningless in others. That is why a tool for debugging task-parallel applications must provide not only the graphical interface to the dependency graph but also numerous ways of interacting with the running application through the visual representation of the graph.

In the following section we will describe how TEMANEJO accomplishes this.

3 The Debugging Process

The first and not to be underestimated aid that TEMANEJO can provide is *displaying the graph*. When the dependencies are correct, the programmer will proceed to run the application within TEMANEJO. Both of these parts of the debugging process require communication between the application and the debugger. This is accomplished in two steps (see Fig. 2).

3.1 Communication

The runtime environment which runs the actual application is instrumented with callback routines of the lightweight communication library AYUDAME. As runtimes differ drastically in how tasks are created, ordered and executed and what is more, what a task can be, the instrumentation cannot be done generically. In fact it has to be “tailor-made” for each runtime and both AYUDAME and TEMANEJO grow with each runtime they support.

In the next step AYUDAME passes the information to TEMANEJO via a previously established socket connection using tcp/ip. This way we assure that the application can run on any remote computer while the programmer can debug it from his desktop. The relevant information consists of eight 64-bit unsigned integer which is send as a package to the socket client TEMANEJO. While the first number always identifies the runtime environment and the last one always is a timestamp, the meaning of the other six numbers differs according to the communicated event.

On the other hand the programmer can launch control requests such as executing one or more tasks (*stepping*), setting breakpoints (see Sect. 2), changing the priority level of a task or the number of active threads. These requests are passed from



Fig. 2 Information flow between the application, AYUDAME and TEMANEJO

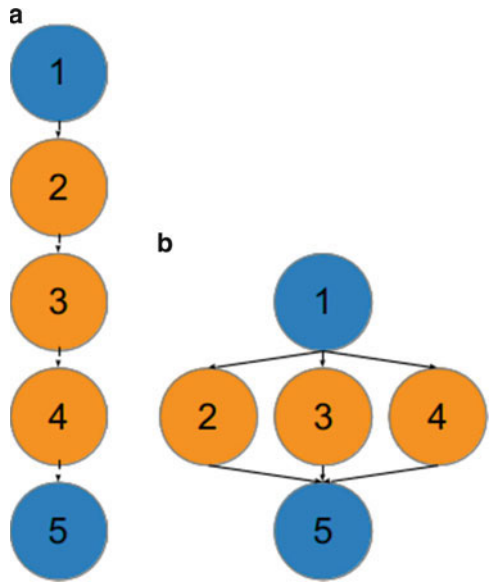


Fig. 3 Typical error as displayed by TEMANEJO. (a) Tasks 2, 3 and 4 implement a reduction but run serial in this program. (b) Corrected version, tasks 2, 3 and 4 run in parallel

TEMANEJO via the tcp/ip socket to AYUDAME which reacts accordingly. This can mean to call a function implemented by the runtime or to set certain variables which in turn can be accessed by the runtime through callback functions.

3.2 Graph Display

TEMANEJO receives the information and displays, analyses and logs the execution of the application. Often times the mere display of the actual graph hints at bugs, unwanted behaviour or even optimisation options not foreseen by the programmer. As an example see Fig. 3.

The information displayed in TEMANEJO are described in the following (see also Fig. 4).

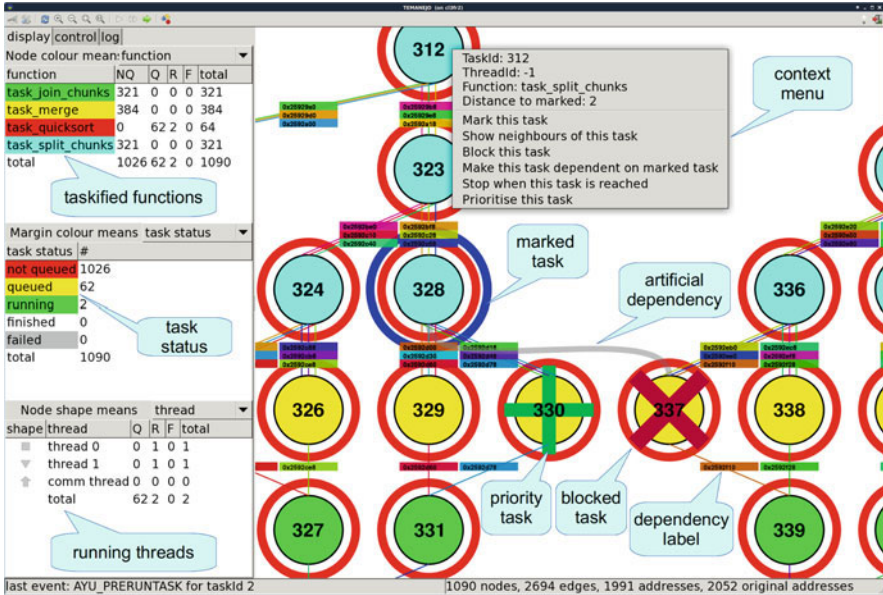


Fig. 4 Annotated screenshot of TEMANEJO

3.2.1 Nodes

The nodes represent tasks which are the natural unit of a task-parallel application. Tasks can consist of a function (subroutine, method), a part of a vectorised loop, a block of code or any other sequence of instructions depending on the runtime. The information displayed by the nodes is:

Label The label is an arbitrary number used to identify the task. It must be unique for each task. Future versions of TEMANEJO will allow for strings to be task labels.

Function By default the fill colour of the node denotes the taskified function or code block. It is communicated by a unique number, the *function id*. These numbers must be consecutive and start with 0. Future versions will allow for arbitrary function ids, consisting of a number or string.

This information is most useful for correctness checking of the structure of the dependency graph.

Task status By default an extra margin is drawn around the nodes which shows the task status. It is red for tasks which still depend on other tasks to finish (*not queued*), yellow for tasks which do not (any more) depend on other tasks but are not yet to be executed because there is no execution resource available (*queued*), green for tasks which are dispatched to run on a specific thread (*running*) and no margin for tasks which ran successfully (*finished*).

The programmer will use this information for checking the basic runtime behaviour of the application. For instance the number of queued tasks (see Sect. 3.2.3) is equivalent to the potential parallelism at a certain point in the execution.

Thread By default the shape of the node indicates the thread on which a task is scheduled to run, is running or has run.

This information can be used by runtime developers to check the correct dispatching of tasks to the threads and by application developers if it is possible to control the dispatch mechanism of the runtime in order to find the optimal setting.

Task duration TEMANEJO offers a rough insight in the execution time of each task. The difference of the cycle counters immediately before and after task execution can be displayed as the node colour. In order to get useful results one has to switch off any type of breakpoint and run the application with the “Fast Forward” button, ignoring at least so many stops as there are threads. The Program will run until the end and the task duration in CPU cycles can be shown as node colour.

This feature does not replace a performance tool but it may give a first hint to tasks which run too long (work balance!) or too short (overhead!).

Distance between tasks When a node is marked using the context menu, the node colours in the graph can be set to indicate the distance to the marked node. As node that are not connected are drawn in white it becomes instantly clear on which part of the graph the task represented by the marked node depends on and which part of the graph depends on this task.

This feature enables the programmer to gain a quick overview over a given application and to analyse the deeper connectivity within the task graph.

3.2.2 Edges

The edges indicate dependencies between tasks. Generally these will consist in data addresses which one task will write to and another will read from. TEMANEJO can indicate the memory address by a label and a colour which uniquely identifies the memory address.

For runtimes which support dependency renaming, a feature equivalent to register renaming which allows the runtime to enhance parallel execution of logically independent tasks working on the same memory, the user can choose to see the original or the renamed memory address.

3.2.3 Other Information

TEMANEJO keeps track of the number of tasks for each function, each thread and the status of each task. Tables indicate how many tasks are queued, running or finished in total and for each thread or function (see Fig. 4). This information can be used

for instance to identify bottlenecks (no queued tasks) or the maximal parallelism at a given stage of execution (number of queued tasks).

Also all messages received from AYUDAME are logged so the exact order of execution can be traced by the programmer.

3.3 Execution Control

By default TEMANEJO will halt the execution before each task. This enables to examine the status of the application while executing it task-wise. On of these steps can consist of one or an arbitrary number of tasks. Halting the application can be switched of in order to run the whole application, e.g. for time measurements.

Individual tasks can be blocked (compare Sect. 2). They are marked with a red cross in the graph display (see Fig. 4). A blocked task is not executed until unblocked. Consequently tasks dependent on a blocked task are not executed.

The mechanism of blocking tasks can be used to add artificial dependencies during the debugging process, i.e. during runtime of the application without recompiling it. When a task is marked (blue margin in the graph display, see Fig. 4) and the programmer right-click on another task, the option “make dependent on marked task” appears active in the context menu. When clicked, the task is blocked and remains so until the previously marked task is finished.

Furthermore it is possible to keep the runtime from running tasks at all (compare Sect. 2). This can be achieved by pressing the button “stop scheduling tasks” in the control tab.

In order to debug a task internally or to debug the main thread it is possible to launch gdb from TEMANEJO. gdb is automatically attached to the applications process and opens with the option `-tui` (terminal/text user interface) in a separate window.

4 Conclusion

We present a debugging toolset for task-parallel applications consisting of the graphical user interface TEMANEJO and the communication library AYUDAME. It has proven very useful to application and runtime developers. With TEMANEJO it is possible to display detailed information about the dependency graph, namely executed functions, state of tasks and threads, duration of tasks, connectivity of subgraphs, memory addresses which cause dependencies, and some statistics.

Moreover the programmer can steer the application by using mechanisms to step task-wise through the application, add dependencies and halt execution at any point in time. The widely used gnu debugger gdb can be launched from TEMANEJO and is automatically attached to the running process.

The library AYUDAME can be used to instrument other runtime environments which use task-parallelism. This way runtime programmer can make debugging with TEMANEJO readily available for application programmers. Presently SMPSSs is fully supported, the instrumentation for OmpSs and StarPU is being developed.

Acknowledgements This work was supported by the European Community's Seventh Framework Programme [FP7-INFRASTRUCTURES-2010-2] project TEXT under grant agreement number 261580. The authors also acknowledge support by the H4H project funded by the German Federal Ministry for Education and Research (grant number 01IS10036B) within the ITEA2 framework (grant number 09011).

References

1. SMPSSs. <http://www.bsc.es/smpss>
2. OMPSSs. <http://pm.bsc.es/ompss>
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, Special Issue: Euro-Par 2009
4. Gautier, Th., Besson, X., Pigeon, L.: KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors. *Parallel Symbolic Computation'07 (PASCO'07)*, (15–23), London, Ontario, Canada, 2007.
5. Brinkmann, S., Gracia, J., Niethammer, Chr., Keller, R.: Temanejo – a debugger for task based parallel programming models. *ParCo* (2011)

Part II
Automatic Error Detection

MPI Runtime Error Detection with MUST: Advanced Error Reports

Joachim Protze, Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz,
Matthias S. Müller, and Wolfgang E. Nagel

Abstract The Message Passing Interface (MPI) is a widely used paradigm for distributed memory programming. Its API is primarily designed for good performance and less for usability; it provides only very limited abstractions that help enforce its correct use. As a result, application developers need tools that aid in the detection and removal of MPI usage errors. Our runtime error detection tool MUST addresses this issue and provides a wide range of automatic correctness checks. MUST uses state-of-the-art approaches to cope with complex MPI semantics like derived datatypes, collective operations, and wildcard receive operations. However, equally important to detecting correctness violations, is that such correctness tools present all details of the violating MPI call(s) required to pinpoint the problem in the source code and to remove the error. In this paper we focus on the error reports presented by MUST and propose a new set of error reports that present complex errors with fine-grained details of the error situation. This includes a deadlock view and a view for usage errors in complex MPI datatypes.

1 Introduction

The development of Message Passing Interface (MPI) [1] applications is a time consuming and complex task. One of the key challenges, aside from achieving high efficiency, is guaranteeing soundness of an application's use of MPI, i.e., its correct usage of the MPI API. While some MPI related errors may directly cause

J. Protze (✉) · T. Hilbrich · M.S. Müller · W.E. Nagel
Center for Information Services and High Performance Computing (ZIH), Technische Universität
Dresden, D-01062 Dresden, Germany
e-mail: joachim.protze@tu-dresden.de

B.R. de Supinski · M. Schulz
Lawrence Livermore National Laboratory, Livermore, CA 94551, USA

Fig. 1 MPI usage error examples. **(a)** Recv-recv deadlock. **(b)** Deadlock resulting from a tag mismatch

Process 0	Process 1
MPI_Recv(from:1, tag:100)	MPI_Recv(from:0, tag:200)
MPI_Isend(to:1, tag:200, &req)	MPI_Isend(to:0, tag:100, &req)
MPI_Wait(&req)	MPI_Wait(&req)
Recv-recv deadlock.	
Process 0	Process 1
MPI_Isend(to:1, tag:200, &req)	MPI_Isend(to:0, tag:100, &req)
MPI_Recv(from:1, tag:200)	MPI_Recv(from:0, tag:100)
MPI_Wait(&req)	MPI_Wait(&req)
Deadlock resulting from a tag mismatch.	

wrong results, application crashes, or hangs, some errors may only manifest on some systems or runs and then in some cases only long after their cause or simply by producing wrong results at the end of the execution. If done manually, finding such problems can be a long and difficult task and developers therefore require tool support that aids in the removal of these errors. Runtime error detection, i.e., detecting errors during an application run, is one tool class that provides this support. We develop the Marmot Umpire Scalable Tool (MUST), named after its predecessor tools Marmot [2] and Umpire [3], for this purpose.

Recent advances in runtime deadlock detection [4] and datatype correctness checks [5] allow MUST to efficiently detect complex errors. However, detecting such errors is only half the solution to the overall problem. Any tool must also present all details about a detected error in a way that helps users understand the erroneous behavior of their codes and help them fix the problem. Consider the following examples that illustrate some potential complexities:

Figure 1 presents two deadlock scenarios with simplified MPI calls. Two processes attempt to send and receive a message from each other using blocking receive and non-blocking send calls. The example in Fig. 1a results in a deadlock, as both processes issue the `MPI_Recv` call without issuing any send calls first. As a result, both processes wait in a cyclic fashion for each other's send call, which is never reached, and hence can't continue execution. MUST's graph-based deadlock detection catches this error and presents the user with a wait-for graph. As no process issued a send call before the receive call, this report includes the key items to understand the error, which in this case are the processes involved in the deadlock and their individual active MPI calls. The situation in Fig. 1b represents a similar communication, which also results in a deadlock, due to a mismatch in the given message tags. MUST's wait-for graph shows the user that both processes are blocked in the `MPI_Recv` call. As both processes have active send calls, the simple criteria used in the example above doesn't hold and the tool user needs to investigate these calls manually in order to determine whether a tag or even a communicator mismatch exists. Different source files that contain active send calls or the use of variables as tag arguments can complicate this further.

In this paper, we present a set of novel output extensions of MUST that provide tool users with the necessary fine-grained and detailed information of such complex error situations, but without overwhelming them with additional unrelated data. In particular, we include:

- A parallel call stack that highlights the processes that MUST determined as the root of a deadlock,
- A condensed message queue that only lists send and receive calls that are meaningful in a deadlock situation, and
- A call-stack based decomposition of the message queue graph to augment a regular message queue graph with source location information, and
- A datatype tree view that highlights error positions in derived datatypes.

We first present an overview of MUST, its correctness checks, and its basic error report in Sect. 2, followed by a summary of MUST’s current deadlock view and datatype usage reports. Afterwards, we present our proposed deadlock view extensions in Sect. 4. Section 5 presents how we can efficiently pinpoint particular error positions in derived datatypes. Finally, we present related work in Sect. 6 and conclude in Sect. 7.

2 MUST

MUST detects MPI usage errors, i.e., usage of MPI calls that are not consistent with restrictions laid out in the MPI standard, during an application run and reports them to the user. Examples for such usage errors are illegal parameters to MPI calls, writes to a send buffer while an asynchronous message transfer is in progress, inconsistent orderings of collective operations, or deadlocks due to improper synchronization. MUST uses the MPI profiling interface to intercept and analyze all MPI calls that an application issues. The tool can be loaded into the application using the *LD_PRELOAD* mechanism. In this case, the usage of the tool becomes as easy as replacing the respective *mpiexec* command with a wrapper command called *mustrun*.

We distinguish two types of correctness checks: local correctness checks and non-local checks. Local checks only require information that is available on a single MPI process and hence don’t require any communication for their execution. As a result, MUST is able to execute local checks inside each application process, or more precisely inside the MUST MPI wrappers used to intercept all MPI calls. Using local checks, we can, e.g., detect whether a datatype that is used in a communication call is committed or whether parameters to MPI calls are out of range. Non-local correctness checks require information from more than one process. Datatype signature matching between sending and receiving communication calls is one such example. The implementation of non-local correctness checks requires additional communication and hence a separate communication mechanism that can forward information about MPI calls to other processes or extra

Rank	Type	Message	From	References
1	Error	Argument 2 (count) has to be a non-negative integer, but is negative (count=-1)	MPI_Send called from: #0 main@Example.c:47 #1 start_main@libc-2.13.so	

Fig. 2 Example MUST error report

resources. MUST uses the Generic Tool Infrastructure (GTI) [6] for this purpose. Currently MUST provides the following classes of correctness checks covering a wide spectrum of possible error cases:

- Local:
 - Integer checks (e.g., restrictions on tags, counts, sizes, and offsets)
 - Integrity checks (e.g., Arrays allocated or communication buffer present)
 - MPI resource surveillance (e.g., use of requests, datatypes, reduce operations, groups, and communicators)
 - Resource leak checks
 - Communication buffer overlap checks
- Non-local:
 - Collective verification (e.g., matching roots and compatible reduce operations)
 - Lost message detection
 - Message type matching (for both point-to-point and collective operations)
 - Deadlock detection

Previous work [4] includes extensive performance results and has shown the feasibility of this approach, including its scalability using an application study on up to 512 processes.

In its initial form, the basic output of MUST is an HTML table that follows the format of Marmot [7]. In Marmot checks had to be implemented for each MPI call, even for the same error conditions, leading to significant code duplication of any error reporting. MUST avoids this redundancy with the use of so-called argument IDs. Figure 2 shows a basic MUST report with an integer usage error. The check that detects the negative count argument in the `MPI_Send` call is mapped to many different calls and argument types. MUST uses the argument IDs to identify the argument number and name, which increases the detail in its output reports. Further, MUST uses the Stackwalker API of the Dyninst project¹ to retrieve call stack information for each MPI call it intercepts.

¹<http://www.dyninst.org/>

3 Shortcoming of Current Error Views

While the initial MUST implementation provided useful information about violated checks, the output format was not optimal and omitted several key pieces of information a user requires to identify the broken code location and to fix it. These shortcomings were introduced because the initial output format was driven by the implementation of the tool and what it naturally collects, without taking the user's needs into account. This is, unfortunately, common for many tools, which flood the user with raw data, but fail to provide some essential details. We illustrate two such problems in the following, using the examples of deadlock detection and problems with complex datatypes. We will first show (in this section) why the existing views are insufficient and (following in the next two sections) how we were able to work around it.

3.1 Example 1: Pinpointing Deadlocks

A key feature of MUST is its graph-based deadlock detection [8]. It creates a wait-for graph and then uses this graph to identify existing deadlock conditions. If such a condition is found, the tool provides the user with a list of processes that are in a deadlocked state as well as their wait-for dependencies that cause them to be deadlocked. This enables MUST to separate processes that cause the deadlock from processes that hang due to waiting for deadlocked processes directly or indirectly.

The graph based approach also has the additional advantage that we can use the graph itself to visualize the deadlock conditions and the wait-for dependencies to the user. As a result, MUST's previous deadlock view provides:

- A textual description of the deadlock situation,
- A wait-for graph of the deadlocked processes, and
- A source location list of the deadlock processes.

In the following we use the erroneous sequence of MPI calls in Fig. 1b as an example to illustrate MUST's previous output. Figure 3a shows the wait-for graph (WFG) that MUST provides for this example. However, this graph along with the source location lists of the deadlocked processes alone is not sufficient to identify the root cause for this error. From our experience, a tool must provide answers to the following questions:

1. Which processes cause the deadlock?
2. What MPI calls are active on these processes?
3. Which control flow led to these active calls?
4. In the case of involved point-to-point operations, which other active communications exist?

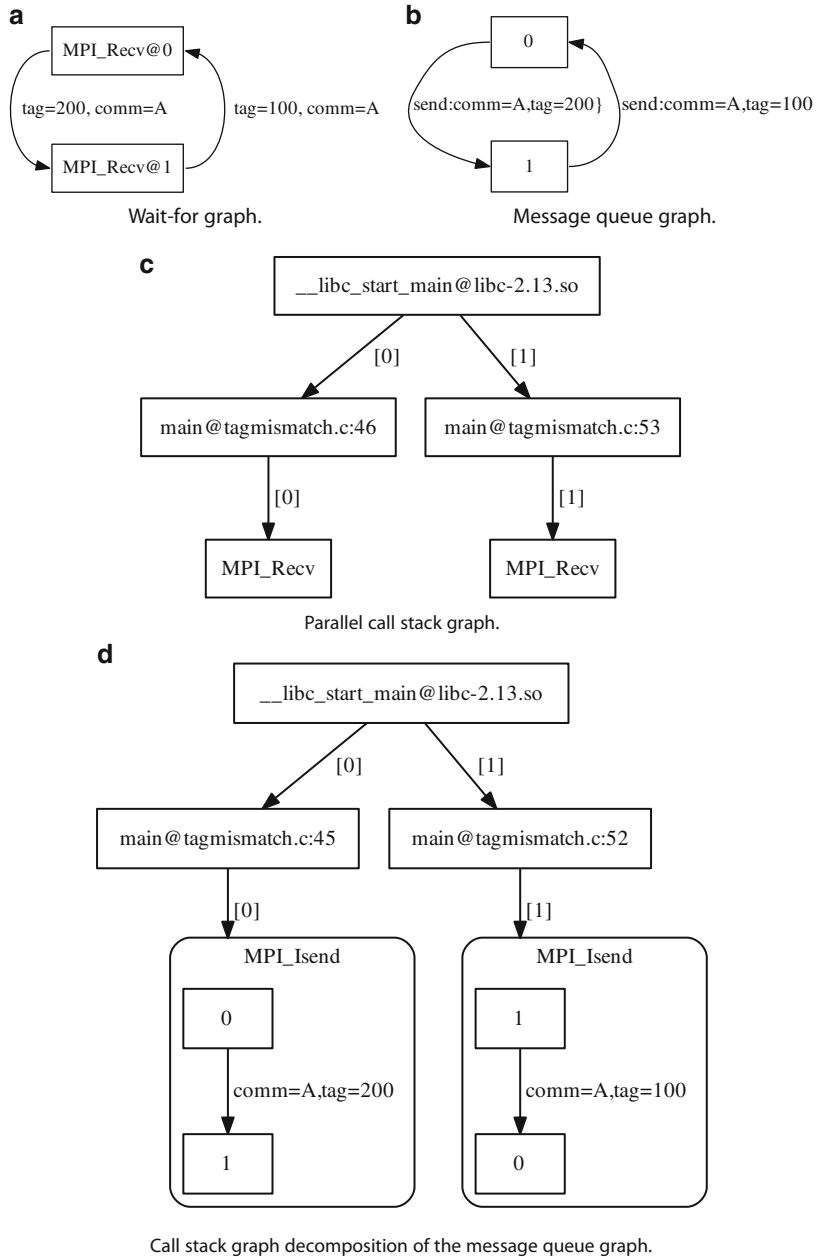


Fig. 3 Deadlock view components for the example in Fig. 1b. (a) Wait-for graph. (b) Message queue graph. (c) Parallel call stack graph. (d) Call stack graph decomposition of the message queue graph

While MUST's previous output provides answers to the first two questions it does not provide information on the latter two. Also, the list of source locations is insufficient for deadlock reports that involve more than a few processes.

3.2 *Example 2: Viewing Datatype Related Problems*

The MPI standard imposes constraints for communication operations. Erroneous usage of MPI datatypes may collide with three of such constraints. In the following we sketch these three referring to version 2.2 of the MPI standard [1]:

- For sending operations, the application may not modify the communication buffer, until the send completes.
- For receiving operations, the application must not access any part of the communication buffer, until the receive completes.
- The type signature of a communication must adhere to matching rules during the following three steps:
 1. MPI types must match programming language types for reads from the application memory (except for the MPI type `MPI_BYTE`),
 2. MPI types must match on receiver and sender sides during transport to receiver, and
 3. MPI types must match programming language types for writes to the application memory (except for the MPI type `MPI_BYTE`).

In MUST we provide checks for overlapping communication buffers handling a subset of clashes with the first two constraints, and for type matching in communication which meets step two of the latter constraint. These checks handle any (derived) datatypes that communication calls may use. We provide no checks for memory manipulation done in application context. Instead, we focus on simultaneous MPI communications that break any of these constraints. If MUST detects such an error, it is crucial that it provides precise information on its source. While the simplest solution would be to provide memory addresses, this provides unsatisfactory details on where the error resides in a communication buffer and its associated MPI datatype. We currently use a path expression approach [5] to pin-point these error locations. An example for this path expression can be found in Sect. 5. While these expressions provide an exact position of the error location within a datatype signature, they require a deep understanding of their format, while losing information about the overall structure of the involved datatype(s).

4 Deadlock View in MUST

As the last section illustrated, MUST's previous deadlock view lacked detail, especially for message mismatch situations, and scalability. To overcome this limitation, we propose a new, dedicated deadlock view that contains the following elements:

- A textual summary,
- A communicator overview,
- The WFG with a legend,
- A parallel call stack,
- A graph representation of the current message queue, and
- A decomposition of the message queue that uses a parallel call stack.

Our new output generator in MUST combines all of these elements in a single HTML page (for better readability, however, we present the individual elements in separate sub-figures). While the textual summary matches our previous outputs, we use the communicator overview to represent each communicator with an upper case letter. In the erroneous sequence of MPI calls in Fig. 1b, which we use as an example throughout this section, the application only uses `MPI_COMM_WORLD`, which we represent as *comm A*. If additional communicators are defined by the application, the communicator summary includes information on the MPI calls that created the communicator. The WFG (Fig. 3a) matches our previous outputs, except that we now use the communicator symbols to also present information on the communicators in use. We also add a legend to this graph as it may contain intermediate nodes to represent complex MPI semantics. Additionally, the new view shows the parallel call stack to provide insights for Question 3 (introduced in Sect. 3) and the last two graphs to provide information for Question 4, which we describe in the following.

Figure 3c shows MUST’s parallel call stack for our example. It helps to illustrate control flow decisions that lead to the deadlock condition. While it is challenging to represent information on the control flow of the individual processes in all details, this limited view provided by call stacks is in most cases sufficient. Additional static source analysis may reveal control flow relevant variables to enrich parallel call stack graphs with further information, as an extension [9] of the STAT [10] tool shows. Further, these graphs scale well with the number of application processes. For our purposes, we limit this call stack graph to only the application processes that are part of the deadlock in order to remove any unnecessary information and provided the most concise representation.

Question 4 addresses situations where point-to-point operations are involved in a deadlock. In this case the root-cause of the error may be a tag or communicator mismatch. In order to understand this situation, the application developer requires information about any active and meaningful point-to-point call, whether it is involved in the actual deadlock condition or not. MUST provides a message queue graph for this purpose. Since MUST detects which processes are part of the deadlock, while it also determines which processes are blocked in point-to-point calls, we can automatically reduce the full message queue graph to only present messages that:

- Were started by a process that is part of the deadlock;
- Have active send operations, which target a process that hangs in a receive operation or a completion that includes a non-blocking receive operation; or

- Have active receive operations, which target a process that hangs in a send operation or a completion that includes a non-blocking send operation.

Using these conditions, we can condense our output to only present relevant point-to-point operations. Figure 3b shows this graph for our example. The graph includes an arc from node 0 (which represents process 0) to node 1 to represent the `MPI_Isend` call that was issued on process 0 before the deadlock manifested. The other arc represents the `MPI_Isend` operation that was started by process 1.

MUST's condensed message queue graph allows application developers to determine whether a potential mismatch exists. In our example, Fig. 3a shows that process 0 waits for a matching send operation of process 1, which uses the tag 200, while Fig. 3b shows that a send operation exits, but with tag 100. If a mismatch exists, the user needs to be able to identify the call and control flow origin of the mismatched operation. We use a parallel call stack to represent all MPI operations that started any operation within MUST's relevant message queue graph. This identifies the call stacks of these operations, but as each operation may use multiple targets, tags, and communicators, we need to highlight which individual parts of the message queue graph result from each leaf of the call stack graph. As a result, we decompose the message queue graph into sub-graphs that represent the components that each MPI operation creates. Figure 3d shows this call-graph-based decomposition for our example. This graph allows the tool user to determine which message might be mismatched, while it contains information about its source location along with limited control flow information.

5 Type Tree View

In this section we will describe a new, more expressive graphical view for datatype related errors.

The code example in Listing 1 sketches a particle simulation where information about a subset of the particles needs to be transferred to a neighbor process. In the application a C struct holds the information about a particle. The set of particles is organized in an array of this struct. Using derived datatypes, MPI enables us to select the subset from the array and send it in a single contiguous operation to the neighbor. To create the fitting datatype, the example uses at first the `MPI_Type_struct` constructor to represent the C struct and then an `MPI_Type_indexed` constructor to select parts of an array of this struct. While the first constructor is correct with respect to type matching, the second one causes a communication buffer overlap when the example issues the `MPI_Sendrecv` call (performed as local operation in this simplified example). MUST's current path expressions calculate to `[0] (INDEXED) [5] [4] (STRUCT) [0] [0] (DOUBLE)` for the sending part and `[0] (INDEXED) [3] [0] (STRUCT) [0] [0] (DOUBLE)` for the receiving part of the `MPI_Sendrecv` call. Figure 4a sketches the overlap within the array (called *cloud*) of the C structure, i.e., the elements that the `MPI_Type_indexed`

Listing 1 Example for a communication buffer overlap

```

    double velocity[3]; double spin[3]; char charge;
    double radius; double mass; };

    struct particle cloud[112];
    MPI_Datatype structtype, indexedtype;

    int blocklens[7] = {3, 3, 3, 3, 1, 1, 1};
    MPI_Datatype types[7] = {MPI_DOUBLE, MPI_INT, MPI_DOUBLE,
        MPI_DOUBLE, MPI_CHAR, MPI_DOUBLE, MPI_DOUBLE};
    // displs derived from c-struct by MPI_Get_address ()
    MPI_Aint displs[7] = {0, 24, 40, 64, 88, 96, 104, 112};
    MPI_Type_struct (7, blocklens, displs, types,
        &structtype);

    int array_of_blocklens[8] = {3, 2, 1, 2, 4, 8, 1, 3};
    int array_of_displs[8] = {3, 13, 23, 34, 44, 55, 65, 76};
    MPI_Type_indexed (8, array_of_blocklens, array_of_displs,
        structtype, &indexedtype);
    MPI_Type_commit (&indexedtype);

    MPI_Sendrecv(cloud, 1, indexedtype, 0, 42, cloud + 25, 1,
        indexedtype, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```

constructor selects from the array. While this representation highlights the overlap, this display loses information about the internal datatype structure. To combine the expressiveness of the path expression and the overview of such a memory map, we propose an overlap graph. This graph visualizes the two path expressions that cause the overlap along with a sketched structure of the datatypes in use. Figure 4b shows this graph for the example in Listing 1. We represent the path expressions of the overlap in red in this graph. For overlaps the trees of the colliding communication operations will either join at a node of the same basic MPI type and absolute offset, as in our example, or we use a compound node if the overlap occurs for two different types/offsets. We join further tree nodes if they compare to equal sub-types, as for the `MPI_Type_struct` in our example. We compute this by recursing the type trees from the leaf towards its root.

An example for a type mismatch can be derived from the above example by mixing up the struct entries for `charge` and `radius` at one of the neighbor processes. The current path expression for this situation calculates to `[0] (INDEXED) [0] [0] (STRUCT) [4] [0] (CHAR)` and `[0] (INDEXED) [0] [0] (STRUCT) [4] [0] (DOUBLE)`, indicating that an `MPI_CHAR` mismatches with an `MPI_DOUBLE`. To display the mismatch we create a tree for both involved datatypes where we skip nodes apart from the (red) error path while we keep a few basic MPI types near the mismatch position to have a more detailed context of the mismatch. To derive a smaller graph we merge similar nodes of both trees. Figure 5 provides the resulting view for the sketched mismatch situation.

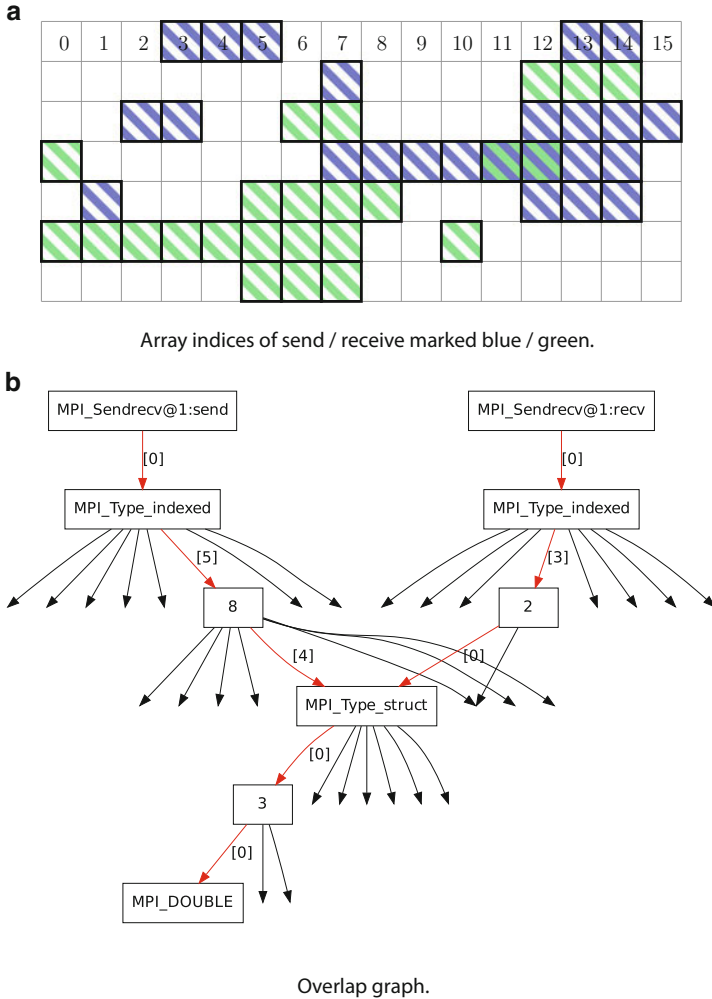


Fig. 4 Overlap view for the example in Listing 1. (a) Array indices of send/receive marked blue/green. (b) Overlap graph

6 Related Work

This work directly relates to other runtime error detection approaches for MPI applications, which include Marmot [2], Umpire [3], ISP [11], MPI-Check [12], and Intel’s approach [13]. While MUST as successor of both Marmot and Umpire identifies deadlocks with a graph-based approach, the MPI-Check tool and Intel’s approach use a timeout-based deadlock detection. As a result, these tools only provide a list of all active MPI calls when the presence of a deadlock is suspected.

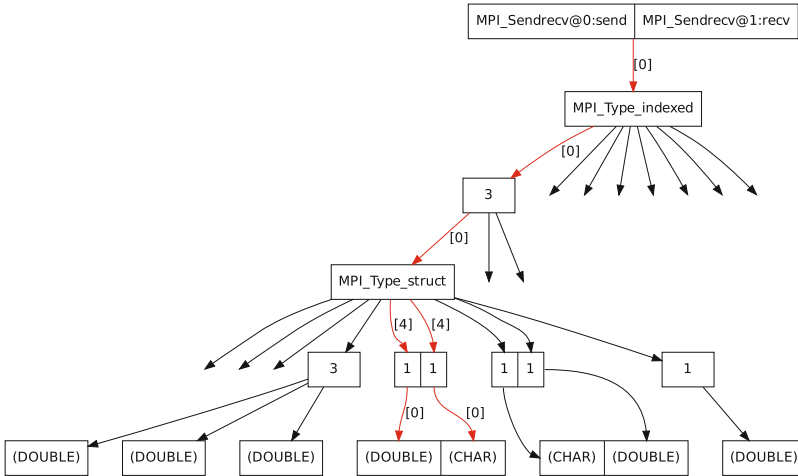


Fig. 5 Type mismatch view for a confusion in the type definition

ISP runs a replay based investigation of all possible interleavings of an MPI application. As a result, this tool can detect some deadlocks that MUST would not detect in a certain application run. ISP’s deadlock output includes a trace of all MPI calls that each process issued, as well as their matching decisions. While very detailed, this output will get overly complex, especially for longer application runs with more than a few processes. While our output contains no complete history of all issued MPI calls, we provide the user with a more scalable deadlock view that condenses relevant history information with the use of a reduced message queue graph.

The STAT [10] tool and debuggers like DDT and Totalview use parallel call stack graphs and/or message queue graphs. Debuggers use interfaces to the MPI library [14] to retrieve message queue information, whereas MUST tracks all MPI calls during the whole application run. Existing integrations of runtime error detection tools with debuggers, e.g. DDT and Marmot [15], could be extended to provide debuggers with information on which processes cause a deadlock. Debuggers could then condense message queue graphs as in our approach. Also, the representation of derived datatypes with trees is based on ideas of the flattening on the fly technique [16].

7 Conclusion

We present the MUST runtime error detection tool for MPI applications along with extensions of its error reports. Our previous output for deadlock situations failed to capture information on active point-to-point messages, which is crucial

in the detection of message mismatch situations. We use message queue graphs to present these active operations. MUST's graph-based deadlock detection yields a set of processes that cause the deadlock, which allows us to condense parallel call stacks and message queues to only include relevant information. In order to add call location information to the message queue graph representation, we propose an extended parallel call stack graph that includes a decomposition of the message queue graphs in their leaves. While these representations allow us to present relevant information for the removal of deadlocks at moderate scale, we still need to investigate their practicability for thousands or more processes. While our approach allows us to visualize deadlocks that only involve a few processes, it may fail for complex deadlocks that involve all or most application processes. This especially affects the size of the WFG and the message queue graphs.

Our second error view provides a detailed output for errors that involve derived datatypes. This includes communication buffer overlaps, and type mismatches between point-to-point or collective MPI operations. The removal of these errors requires a precise understanding of which part in a derived datatype causes the error. As a result, we use a narrowed type tree representation that highlights the position in the datatype that causes the error, while it sketches the structure of the involved datatypes at the same time.

Acknowledgements Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-586816). This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2. <http://www.mpi-forum.org/docs/mpi22-report.pdf> (April 2009)
2. Krammer, B., Müller, M.S.: MPI Application Development with MARMOT. In: PARCO. Volume 33., Central Institute for Applied Mathematics, Jülich, Germany (2005) 893–900
3. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. Supercomputing, ACM/IEEE 2000 Conference (04–10 Nov. 2000) 51–51
4. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: Mpi runtime error detection with must: Advances in deadlock detection. In: Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '12, New York, NY, USA, ACM (2012)
5. Protze, J., Hilbrich, T., Knüpfer, A., de Supinski, B.R., Müller, M.S.: Holistic Debugging of MPI Derived Datatypes. In: IPDPS 2012: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium. (2012)
6. Hilbrich, T., Müller, M.S., de Supinski, B.R., Schulz, M., Nagel, W.E.: GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems. In: IPDPS 2012: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium. (2012)
7. Krammer, B., Hilbrich, T., Himmler, V., Czink, B., Dichev, K., Müller, M.S.: MPI Correctness Checking with Marmot. In: Parallel Tools Workshop'08. (2008) 61–78

8. Hilbrich, T., de Supinski, B.R., Schulz, M., Müller, M.S.: A Graph Based Approach for MPI Deadlock Detection. In: ICS '09: Proceedings of the 23rd International Conference on Supercomputing, New York, NY, USA, ACM (2009) 296–305
9. Ahn, D.H., de Supinski, B.R., Laguna, I., Lee, G.L., Liblit, B., Miller, B.P., Schulz, M.: Scalable temporal order analysis for large scale debugging. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. SC '09, New York, NY, USA, ACM (2009) 44:1–44:11
10. Arnold, D., Ahn, D., de Supinski, B., Lee, G., Miller, B., Schulz, M.: Stack Trace Analysis for Large Scale Debugging. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. (march 2007) 1–10
11. Vakkalanka, S.S., Sharma, S., Gopalakrishnan, G., Kirby, R.M.: ISP: A Tool for Model Checking MPI Programs. In: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2008) 285–286
12. Luecke, G.R., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., Zou, Y.: MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs. *Concurrency and Computation: Practice and Experience* **15**(2) (2003) 93–100
13. Desouza, J., Kuhn, B., Supinski, B.R.D.: Automated, Scalable Debugging of MPI Programs with Intel Message Checker. In: In Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS). (2005)
14. Cownie, J.: A standard interface for debugger access to message queue information in MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 1697 of Lecture Notes in Computer Science, Springer Verlag (1999) 51–58
15. Krammer, B., Himmler, V., Lecomber, D.: Coupling DDT and Marmot for debugging of MPI applications. In: PARCO'07. (2007) 653–660
16. Träff, J.L., Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the Fly: Efficient Handling of MPI Derived Datatypes. In: Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, Springer-Verlag (1999) 109–116

Advanced Memory Checking for MPI Parallel Applications Using MemPin

Shiqing Fan, Rainer Keller, and Michael Resch

Abstract In this paper, we describe the implementation of memory checking functionality that is based on instrumentation tools. The combination of instrumentation based checking functions and the MPI-implementation offers superior debugging functionalities, for errors that otherwise are not possible to detect with comparable MPI-debugging tools. Our implementation contains three parts: first, a memory callback extension that is implemented on top of the Valgrind Memcheck tool for advanced memory checking in parallel applications; second, a new instrumentation tool was developed based on the Intel Pin framework, which provides similar functionality as Memcheck. It can be used in Windows environments that have no access to the Valgrind suite; third, all the checking functionalities are integrated as the so-called memchecker framework within Open MPI. This will also allow other memory debuggers that offer a similar API to be integrated. The tight control of the user's memory passed to Open MPI, allows us to detect application errors and to track bugs within Open MPI itself. The extension of the callback mechanism targets communication buffer checks in both pre- and post-communication phases, in order to analyze the usage of the received data, e.g. whether the received data has been overwritten before it is used in an computation or whether the data is never used. We describe our actual checks, how memory buffers are being handled internally, show errors actually found in user's code, and the performance improvement of our instrumentation.

S. Fan (✉) · M. Resch
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart,
70550 Stuttgart, Germany
e-mail: fan@hlrs.de

R. Keller
Hochschule für Technik Stuttgart, Schellingstr. 24, 70174 Stuttgart, Germany

1 Introduction

Parallel programming with the Message Passing Interface MPI [7] as a distributed memory paradigm is an error-prone process. Great effort has been put into parallelizing libraries and applications using MPI. However when it comes to maintaining software, optimizing for new hardware, or even porting codes to other platforms and other MPI implementations, developers will face additional difficulties [2]. They may experience errors due to hard-to-track interleaving dependent bugs, deadlocks due to communication characteristics, and MPI-implementation defined or even hardware dependent behavior. One class of bugs that are hard-to-track are memory errors, specifically in non-blocking and one-sided communication.

In the beginning of previous work, we introduced new implementations of the `memchecker` framework within Open MPI to check for memory problems in parallel applications [3, 11]. It extends and integrates the Valgrind [10] Memcheck tool in `memchecker` to observe communication buffers during the communication as well as user specified parameters. We also introduced a newly developed memory checking tool, MemPin, which has similar functionalities as Memcheck and its extension. This new tool provide more flexibilities to be integrated in the MPI libraries. It supports both Linux and Windows platforms, and more new memory check functionalities may be implemented. On the other hand, the two phase MPI communication checks, i. e. pre- and post-communication checks have been defined in previous work. Performance implications based on these checks were also discussed, which showed that the introduced overhead by the debugging feature is minimum when the user application is not running with the debugger.

MemPin has been integrated into Open MPI for pre- and post-communication checks. The communication buffers errors, such as accessing buffers of active non-blocking operations, writing communicated buffer before reading it, or transferring unused data are being checked and reported. This kind of functionalities would otherwise not be detectable within traditional MPI-debuggers based on the PMPI-interface. In this paper, we continue the work on MemPin. Details of the MemPin implementation will be discussed. The integration of the tool and Open MPI is based on pre- and post-communication checks for non-blocking and collective communications. We will take a closer look into how MemPin and Open MPI work together, as well as how new checks may be implemented. Furthermore, several MPI parallel applications will be introduced to run with our memory checking tool. Problems, bugs and critical issues in these applications, which have been found using the debugging tool, will be mainly focused on.

The structure of this paper is as follows: Sect. 2 shows the basic idea and functionalities of Intel PIN; Sect. 3 first introduces the functionalities and architecture of MemPin, then it gives a detailed description on the integration the tool in Open MPI libraries; Sect. 4 shows details of the pre- and post-communication checks for parallel applications; then in Sect. 5 we give more performance

improvement results based on previous work; finally; in Sect. 6, we introduce a 2D heat conduction application, and discuss issues found by running the application with MemPin; in Sect. 7, we make a comparison with other available tools and conclude the paper with an outlook of our future work.

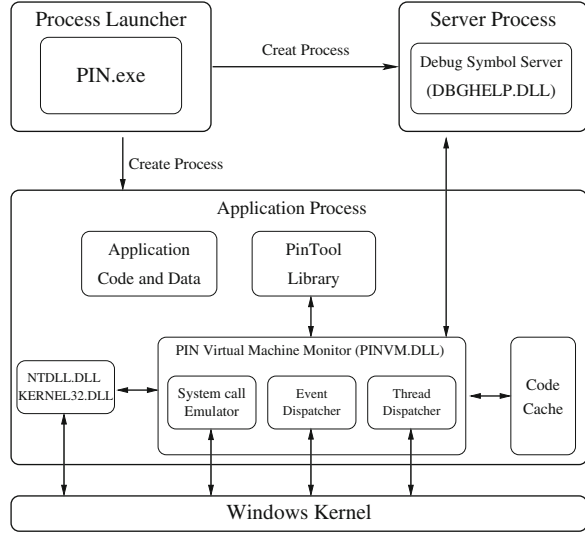
2 Overview of Intel Pin

Intel Pin [6] is a framework for building robust and powerful software instrumentation tools, such as profiling, performance evaluation, and error detection tools. Intel Pin is capable of analyzing and instrumenting application code, and it is easy-to-use, portable, transparent, and efficient for building instrumentation tools (Pintools) written in C/C++. The Pin framework follows the ATOM [13] model that allows the Pintools to analyze the application at the instruction level without detailed knowledge of the underlying instruction set. The framework API is designed to be platform independent in order to make the Pintools compatible across different architectures. It can provide architecture specific details when necessary. The instrumentation process is transparent as both the application and the Pintool observe the original application code. Pin uses techniques like inlining, register re-allocation, and instruction scheduling, in order to run more efficiently. The basic overhead of the Pin framework is approximately 10–20%, and extra overhead might be caused by the Pintool.

Figure 1 shows a basic runtime architecture of Pin on Windows. It consists of three main processes, the launcher process, the server process and the instrumented process. The launcher process creates the other two processes, injects the Pin modules, and instrumented code into the instrumented process, then it waits for the process to terminate. The server process provides services for managing symbol information, injecting Pin, or communicating with the instrumented process via shared memory. The instrumented process includes the Pin Virtual Machine Monitor (VMM), the user defined Pintool library, and a copy of the application executable and libraries. The VMM is the core engine of the entire instrumented process that includes a system call emulator, event and thread dispatchers, and also a (Just In Time) JIT compiler. After Pin takes over the control of the application, the VMM coordinates its execution. The JIT compiler instruments the code and passes it to the dispatcher, which launches the execution. The compiled code is stored in the code cache. The Pin tool contains the instrumentation and analysis routines. It is a plug-in and linked with the Pin library, which allows it to communicate with the Pin VMM.

The Pin JIT compiler recompiles and instruments small chunks of binary code immediately before executing them. The modified instructions are stored in a software code cache. It allows code regions to be generated once and reused for the remainder of program execution, in order to reduce the costs of recompilation. The overhead introduced by the compilation is highly dependent on the application and workload [6].

Fig. 1 Overview of program execution with an Intel Pin tool on Windows



In this paper, we will describe the implementation of Memcheck extension and the integration of a newly developed Pintool as the second memchecker component in Open MPI, which may help MPI application and Open MPI developers to track erroneous use of memory, such as reading or writing buffers of active non-blocking receive operations, writing to buffers of active one-sided get operations as well as erroneous use of communicated data, such as writing the communicated buffer before reading.

3 Design and Implementation

In previous work, we have taken advantage of an instrumentation API offered by Memcheck to find MPI-related hard-to-track bugs in applications (and within Open MPI). In order to allow other kinds of memory-debuggers, such as `bcheck` [1] or Totalview's memory debugging features [15], we have implemented the functionality as a module within Open MPI's Modular Component Architecture [17]. The module is therefore called `memchecker` and may be enabled with the configure-option `--enable-memchecker`.

However, that did not meet all the requirement of advanced MPI semantic memory checking, and the current functionalities of Valgrind Memcheck do not suffice. For example, the future MPI standard will change behavior compared to the current version of the standard in that it allows read access to send buffers of non-blocking operations. In order to detect send buffer rewrite errors, Memcheck has to know which memory region is readable or writable. On the other hand, checking the communicated buffer is also important, for example, writing the

received buffer before reading may cause computation errors. And for performance concerns, data transferred but never used (read) might also be necessary to detect. The extensions for Memcheck have been implemented for this purpose.

3.1 MemPin

As Open MPI is supported on Windows platforms, a debugging tool for checking memory errors is also necessary. A new tool named MemPin has been designed on top of Intel Pin framework to meet this need.¹

The MemPin tool uses Intel Pin's instrumentation API to provide the same callback functionalities as the Memcheck extension for the user application. Furthermore it may be used to perform the basic functionalities of Memcheck, such as make memory readable or inaccessible, but through a different approach (see Sect. 4.2). The available interfaces and descriptions are:

- `MEMPIN_RUNNING_WITH_PIN`
Checks whether the user application is running under Pin and Pintool.
- `MEMPIN_REG_MEM_WATCH`
Registers the memory entry for specific memory operation.
- `MEMPIN_UPDATE_MEM_WATCH`
Updates the memory entry parameters for the specific memory operation.
- `MEMPIN_UNREG_MEM_WATCH`
Deregisters one memory entry.
- `MEMPIN_SEARCH_MEM_INDEX`
Returns the memory entry index from the memory address storage.
- `MEMPIN_PRINT_CALLSTACK`
Prints the current callstack to standard output or a file.

The user may use the MemPin API to register memory regions with specific callback function and parameter pointers. When the user application is not running with the Pintool, all MemPin calls will be taken as empty macros, and add no overhead. But if running with MemPin tool, Pin first reads the entire executable, and all the MemPin calls will be replaced with the corresponding function calls that are defined inside MemPin. The generated instrumented codes will be then executed and MemPin will observe and respond to the behavior of the user application.

¹The MemPin tool in this work is developed only targeting at Windows platforms, although it may be used under Linux too.

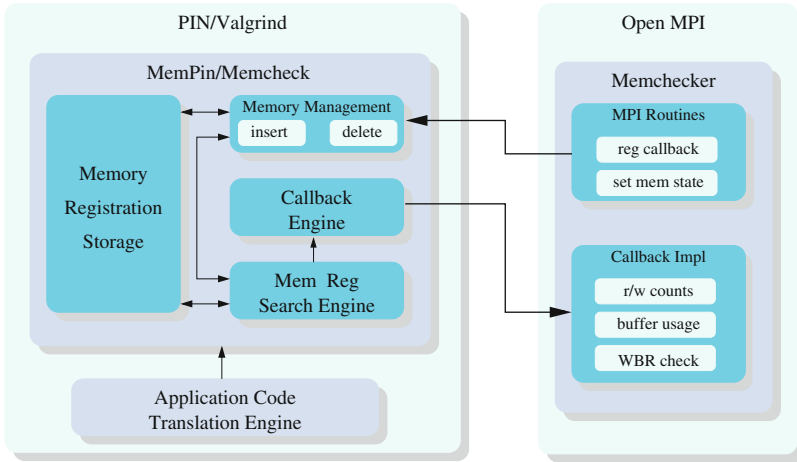


Fig. 2 Run-time structure of MemPin

MemPin uses image and trace instrumentations in user applications. The image instrumentation is done when the image is loaded. In this stage, all the MemPin calls used in the user application will be replaced, and the main entry function, like `main` will be instrumented for starting the trace engine and the callstack log of MemPin.

The next stage will mainly take care of the memory access, callback functions and the user application callstack. The trace instrumentation is analyzed according to each Basic Block (BBL), and every memory operation in the BBL is checked. When the memory is read or written, the single instruction of the memory operation is instrumented with an analysis function with memory information as operands. For generating useful information of where exactly the memory operation has happened, a callstack log engine is instrumented also in this stage. The callstack engine is implemented using a simple C++ stack structure, which stores only the necessary historical instruction addresses of the application and translates the addresses into source information when required. The new function entry address from the caller will be pushed onto the stack, and it is popped off at the end of the callee. To achieve this goal, the tail instruction of each BBL has to be analyzed. More precisely, every “call” and “return” instruction are instrumented for pushing and popping the instruction address stack.

3.2 Integration of MemPin with Open MPI

MemPin has been successfully integrated into Open MPI as an MCA component, in order to achieve the parallel memory checking discussed in previous sections. Figure 2 shows the basic architecture of MemPin and how it can work with

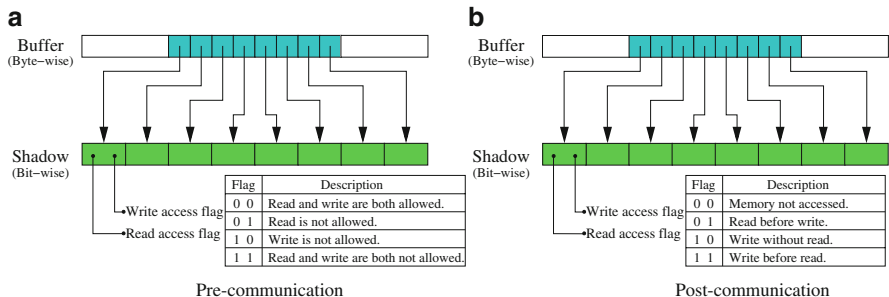


Fig. 3 Shadow memory for pre- and post-communication using MemPin. (a) Pre-communication. (b) Post-communication

Open MPI. memchecker component can directly communicate with MemPin at runtime. When a communication is initialized, memchecker send a request to MemPin to record or update the memory information. The Memory Management component is in charge of inserting, deleting or updating memory entries. An memory entry contains information of a communication buffer, including starting address, size, callback function pointer and memory operation flag. All memory entries are stored in the Memory Registration Storage, which is a multiple map data structure. The Search Engine intercepts the translated application code from Intel Pin, and find the match access according to the storage. If a match is found, it will send corresponding memory information to the Callback Engine, which then will directly call the callback function registered with the memory entry.

MemPin has no information about whether the memory is readable or writable. But similar functionalities for making memory readable and writable have been implemented in Open MPI using the callback scenario of MemPin. In the callback function defined in Open MPI, for both pre- and post-communication checks, the same memory states Bits are used. For pre-communication checks, the two Bits of memory state are used for marking the memory readable or writable, as shown in Fig. 3a. If the first Bit is set to “1”, the memory is marked as not readable. The second Bit is the writable Bit, which means a “1” is for not writable. When both Bits are set to “1”, then the memory is marked as not accessible at all.

For post-communication checks, the same Bit table will be used in order to save the storage. But the two bits have different meanings, see Fig. 3b. The first Bit indicates whether the byte of memory has been read or not. The second bit is for whether the byte of memory has been written or not. The callback function will check for each registered receive buffer, whether they are read before written, otherwise reporting a Write Before Read error. Furthermore, in the MPI_Finalize call, all memory state Bits are checked for buffer that are not used after communication.

These two phases of memory checks in MPI communication may change automatically to the other phase. If a parallel application has several communications, whenever the communication is started, for example calling the MPI_Isend, the pre-communication check will be enabled. When the communication is finished,

for example `MPI_Wait` is called, the post-communication check will be executed. The shadow memory for both checks does not need to be reallocated, as they have the same format but different meaning of the Bits. All the registered memory checks will be cleared in the `MPI_Finalize`, which is the end of the parallel computation.

4 Memory Checks in Parallel Application

4.1 Pre-communication Checks

In Open MPI objects such as communicators, types and requests are declared as pointers to structures. These objects when passed to MPI-calls are being immediately checked for definedness and together with `MPI_Status` are checked upon exit.² Memory being passed to send operations is being checked for accessibility and definedness, while pointers in receive operations are checked for accessibility, only.

Reading or writing to buffers of active, non-blocking receive operations and writing to buffers of active, non-blocking Send-operations are obvious bugs. Buffers being passed to non-blocking operations (after the above checking) are being set to undefined within the MPI-layer of Open MPI until the corresponding completion operation is issued. This setting of the visibility is being set independent of non-blocking `MPI_Isend` or `MPI_Irecv` function. When the application touches the corresponding part in memory before the completion with `MPI_Wait`, `MPI_Test` or multiple completion calls, an error message will be issued. In order to allow the lower-level MPI-functionality to send the user-buffer as fragment, the lower-layer BTLs (Byte Transfer Layers) are adapted to set the fragment in question to accessible and defined, as may be seen in Fig. 4. Care has been taken to handle derived datatypes and its implications. Complex datatypes are checked according to their definitions, which means gaps will be ignored to avoid false positive messages.

For send operations, the MPI-1 standard also defines, that the application may not access the send-buffer at all (see [7], p. 30). Many applications do not obey this strict policy, domain-decomposition based applications that communicate ghost-cells, still read from the send-buffer. To the authors' knowledge, no existing implementation requires this policy, therefore the setting to undefined on the Send side is only done when strict-checking is enabled.

For one-sided communications, MPI-2 standard defines that, any conflicting accesses to the same memory location in a window are erroneous (see [8], p. 112). If a location is updated by a put or an accumulate operation, then this location cannot be accessed by a load or another RMA operation until the updating operation is

²For example this showed up uninitialized data in derived objects, e.g. communicators created using `MPI_Comm_dup`.

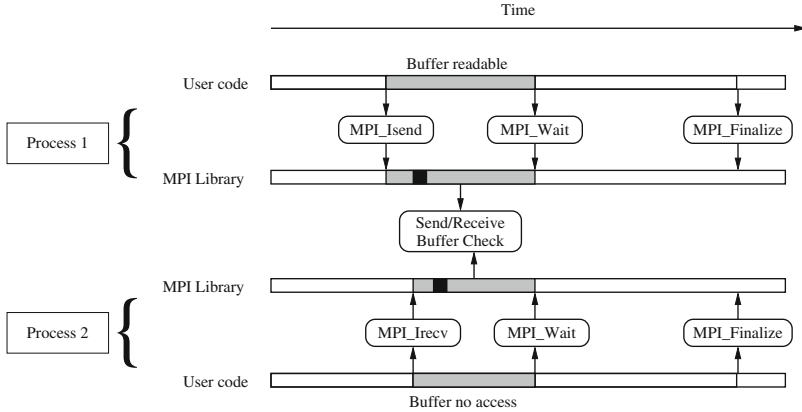


Fig. 4 An example of non-blocking communication using pre-communication checks with fragment handling to set accessibility and definedness

completed on the target. If a location is fetched by a get operation, this location cannot be accessed by other operations as well. When a synchronization call starts, the local communication buffer of an RMA call and a get call should not be updated until it is finished. User buffer of MPI_Put or MPI_Accumulate, for instance, are set not accessible when these operations are initiated, until the completion operation finished. Valgrind will produce an error message, if there is any read or write to the memory area of the user buffer before corresponding completion operation terminates.

In Open MPI, there are two One-sided communication modules, point-to-point and RDMA. Similar checks have been implemented for MPI_Get, MPI_Put, MPI_Fence and MPI_Accumulate in point-to-point module.

For the above communication buffer checking, the original features of Valgrind are used and integrated in Open MPI on Linux. On the other hand, in order to implement the same checks on Windows, the callback scenario of MemPin is used. When the communication starts, for example, the MPI_Isend and MPI_Irecv are called, all the communication buffers are registered, and all read and write access on the buffers will be checked whether they are legal according to the standards. If an illegal access is found, a warning message with sufficient callstack information will be generated for the user. However, in order to make the MemPin efficient and lightweight, memory checks for accessibility and definedness cannot be easily implemented due to complex and large shadow memory consumption.

4.2 Post-communication Checking

For more detailed memory checking, one may further implement an interface to encode read and write accesses of previously communicated data. This new

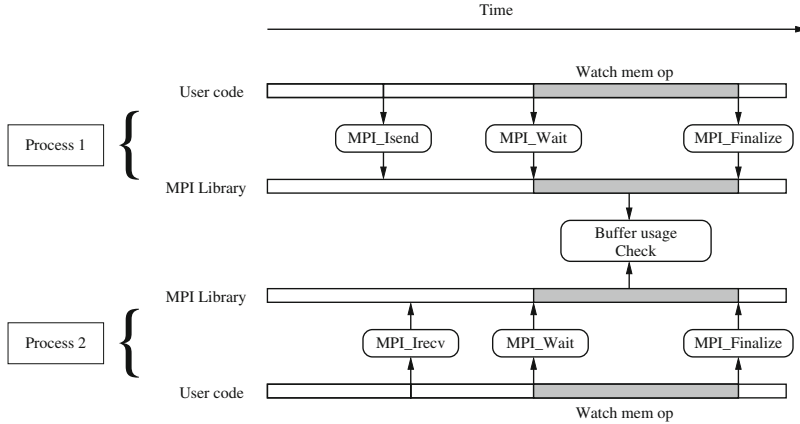


Fig. 5 An example of non-blocking communication using post-communication checks

analysis mechanism may help user applications to detect whether data has been sent needlessly, i. e. data that has been sent but might be overwritten before reading or may be never accessed in the receiving process.

More precisely, read accesses on the received data is counted as meaningful, while the first write access is not, for the communicated buffer is overwritten before any actual use. To achieve this goal, we make use of the tools introduced in Sect. 3.1, in order to notify Open MPI of the corresponding tasks based on the type of operations, i.e. read or write to the received buffer.

All the communication buffers are registered when the communication finishes. For example, in Fig. 5, when the `MPI_Wait` is called for non-blocking communication, every read and write access on the received buffer will be processed by the callback implementation in Open MPI, and a write before read is marked as illegal. In the finalization phase of the parallel application, all registered memory will be checked for whether there are communicated data but never used, i. e. whether there are buffers without a read access.

5 Performance Comparison

In previous work [12], we showed the performance when the benchmarks were not run with debuggers. The results proved that the overhead introduced by the debugging frameworks are neglectable. However, when the benchmarks are run with debuggers, there will be approximately 30–50 % slowdown.

Figure 6 shows a comparison between running with Valgrind and MemPin using NetPIPE. NetPIPE was run under supervision of Valgrind and MemPin respectively using two compute nodes interconnected with InfinBand on Nehalem cluster at HLRS. The latency (in Fig. 6a) of using MemPin is nearly 50 % less than

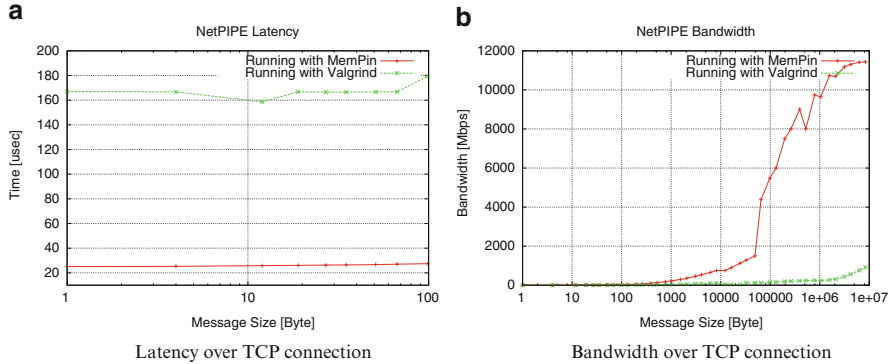


Fig. 6 NetPIPE latency (*left*) and bandwidth (*right*) comparison of Open MPI run with the memchecker framework over InfiniBand. **(a)** Latency over TCP connection. **(b)** Bandwidth over TCP connection

the latency using Valgrind. The difference of bandwidth between the two tests increases largely when the message size increases, as shown in Fig. 6b. Using the memchecker framework based on MemPin has a better performance than using framework based on Valgrind.

6 2D Heat Conduction Program with MemPin

During the course of development, several software packages have been tested with the memchecker functionality. Among them problems showed up in Open MPI itself (failed in initialization of fields of the status copied to user-space), an MPI testsuite [4], where tests for the MPI_ERROR triggered an error. In order to reduce the number of false positives in Infiniband networks, the `ibverbs` library of the OFED stack [14] was extended with instrumentation for buffer passed back from kernel-space.

A 2d heat conduction algorithm has been used for running with the new implemented memory checking framework on both Linux and Windows. The algorithm is based on Parallel CFD Test Case [9]. It solves the partial differential equation for unsteady heat conduction over a square domain. It was firstly run with two processes under control of MemPin, where warnings about 112 bytes of communicated but not used data were reported. A small amount of data on two processes may be not critical to the communication time. But when running with large number of processes with the application, it may differ.

In the 2D domain decomposition algorithm, it requires calculating elements from their horizontal and vertical neighbor elements, but the whole border element arrays are updated from neighbor sub-domain. This results that, for the border update in every sub-domain, there will be four corner elements that will never be used for

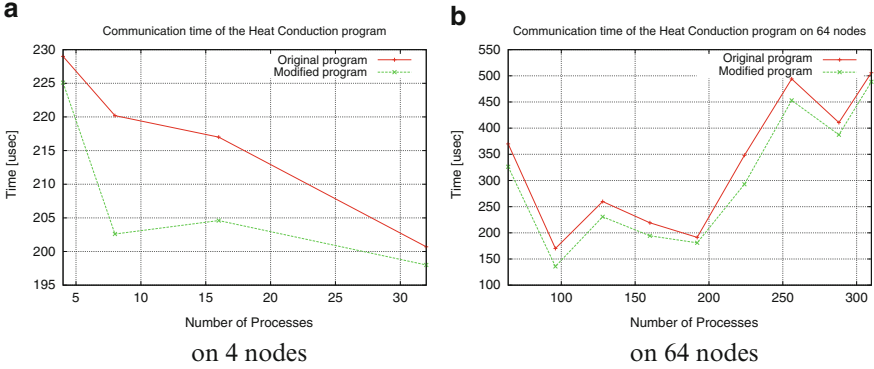


Fig. 7 Comparison of the communication time between the original and modified heat conduction program on Nehalem cluster. (a) On 4 nodes. (b) On 64 nodes

calculation (without periodic boundary condition, the virtual border elements are not taken into account in the example). This might be no harm for the calculation result of the algorithm. But when decomposing the entire problem into a large number of sub-domains, the total amount of transferred but unused data may be high, and as consequence communication might require more time.

Take a 4×4 domain decomposition as an example, where every element calculation requires horizontal and vertical neighbor elements. In this specific condition, there will be 72 elements transferred but not used (36 corner elements transferred two times). When scaling this code by doubling the number of processors used to compute this domain, the number of elements communicated but not used increases dramatically. In the case of a 8×8 domain decomposition, that has 392 elements (196 corner elements transferred two times) might not be communicated. Assuming we have a $M \times N$ domain decomposition, the total amount of such elements are described by:

$$(M - 1) \times (N - 1) \times 4 \times 2 \quad (1)$$

It is obvious that, the number of unnecessary communicated data grows superlinearly with the domain decomposition.

The heat program has further been tested with more processes on different number of nodes on BWGrid and Nehalem Cluster at HLRS, in order to discover the relationship between the communicated but unused corner elements and the communication performance. For the first test, the heat program was set to a $1,500 \times 1,500$ domain, and parallelized with different number of processes over four compute node (eight cores on each node) on Nehalem cluster. A modified version of the heat program was also used for the test, which does not send any unused corner elements. The average communication time and overall run time are measured based on five executions on different number of processes, as shown in Fig. 7a. When not

oversubscribing the nodes (each core has no more than one process), the modified version is generally better than the original version ranging from 3 to 7 %. As we can see, communicating the corner elements of the sub-domains will indeed affect the communication time of the program.

Another test on 64 nodes was made on Nehalem cluster to start large number of processes without oversubscribing the compute node cores. The processes are assigned using round-robin algorithm among the nodes, in order to achieve a better load balancing for the simulation. Figure 7b shows the communication time of running the same simulation with different number of processes on the cluster. It presents the communication time for different number of processes (64–310). The modified version has a shorter communication time on average, which is 10 % better. The best case is even 20 % better than the original version.

The communication time does not increase or decrease linearly with the number of processes, because the domain decomposition will influence the communication efficiency. Assuming we have eight bytes data in each corner element, for a 96 processes run (12×8 decomposition), the number of border exchange is $(12 - 1) \times (8 - 1) \times 4 \times 2$, which is 616 times. This results to 4,928 bytes of communicated but never used data. For the same configuration, if running with 128 processes (16×8 decomposition), the size of each border element is halved, i. e. four bytes. But the number of border exchange is now $(16 - 1) \times (8 - 1) \times 4 \times 2$, which is 840 with 3,360 bytes in total. One may argue that the total size of transferred data is smaller with high resolution of domain decomposition, the communication speed should increase. However, this is not true. The overall communication speed is highly determined by the number of communication but not the data size that is transferred. In Open MPI, for blocking and non-blocking communication, there are two transmission protocols, i.e. Eager and Rendezvous. When the data size is small than 12 kB, the data will be sent in one package (Eager protocol). But when the data size is larger than 12 kB, the data will be divided into smaller packages (Rendezvous protocol), so there is not only one send and receive operation on this data. When the data size does not exceed the limit, the number of the communication will determine the overall communication speed. This also explains why the communication time is larger when running with 64 processes. In this case, the corner data is much larger than 12 kB, so the number of communication is doubled or even tripled.

7 Conclusion

We have presented an implementation of memory debugging features into Open MPI, using the instrumentation of the `valgrind` and a newly developed Intel Pintool, and the performance implication of using the instrumentation with several benchmarks. This allows detection of hard-to-find bugs in MPI parallel applications, libraries and Open MPI itself [2]. Up to now, no other debugger is able to find these kinds of errors.

With regard to related work, debuggers such as Umpire [16], Marmot [5] or the Intel Trace Analyzer and Collector [2], actually any other debugger based on the Profiling Interface of MPI, may detect bugs regarding non-standard access to buffers used in active, non-blocking communication without hiding false positives of the MPI-library itself.

Acknowledgements This work was funded by project Int. EU. Grid (Interactive European Grid) with EU-Contract Number 031857, by project DORII (Deployment of Remote Instrumentation Infrastructure) with EU-Contract Number 213110 and by Microsoft Technical Computing Initiative (TCI) since March 2007.

We would like to thank Julian Seward and the open source community for `valgrind`, which has proven invaluable in many software projects.

References

1. bcheck Man Page from SUN developers Website. Internet (2011). <http://developers.sun.com/sunstudio/documentation/ss11/mr/man1/bcheck.1.html>
2. DeSouza, J., Kuhn, B., de Supinski, B.R.: Automated, scalable debugging of MPI programs with Intel message checker. In: Proceedings of the 2nd International Workshop on Software engineering for high performance computing system applications, vol. 4, pp. 78–82. ACM Press, NY, USA (2005)
3. Keller, R., Fan, S., Resch, M.: Memory debugging of MPI-parallel Applications in Open MPI. In: G. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bucker, P. Gibbon, B. Mohr (eds.) Proceedings of ParCo’07. Julich, Germany (2007)
4. Keller, R., Resch, M.: Testing the Correctness of MPI implementations. In: Proceedings of the 5th Int. Symp. on Parallel and Distributed Computing conference, pp. 291–295. Timisoara, Romania (2006)
5. Krammer, B., Mller, M.S., Resch, M.M.: Runtime checking of MPI applications with Marmot. Malaga, Spain (2005)
6. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190–200. ACM
7. Message Passing Interface Forum: MPI: A Message Passing Interface Standard (1995). <http://www.mpi-forum.org>
8. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (1997). <http://www.mpi-forum.org>
9. Resch, M., Sander, B., Loebich, I.: A comparison of OpenMP and MPI for the parallel CFD test case. In: Proc. of the First European Workshop on OpenMP, pp. 71–75 (1999)
10. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX’05 Annual Technical Conference. Anaheim, CA, USA (2005)
11. Shiqing Fan, R.K., Resch, M.: Enhanced memory debugging of mpi-parallel applications in open mpi. In: 4th Parallel Tools Workshop (2010)
12. Shiqing Fan, R.K., Resch, M.: Advanced memory checking frameworks for mpi parallel applications in open mpi. In: 5th Parallel Tools Workshop (2011)
13. Srivastava, A., Eustace, A.: Atom: A system for building customized program analysis tools. pp. 196–205. ACM (1994)
14. The Open Fabrics project webpage. WWW (2007). <https://www.openfabrics.org>

15. Totalview Memory Debugging capabilities. WWW. <http://www.etnus.com/TotalView/Memory.html>
16. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. In: Proceedings of Supercomputing (SC) (2000). <http://www.sc2000.org/proceedings/techpaper/index.htm>
17. Woodall, T., Graham, R., Castain, R., Daniel, D., Sukalski, M., Fagg, G., Gabriel, E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A.: Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 3241, pp. 105–111. Springer, Budapest, Hungary (2004)

Part III
Performance Analysis and Optimization

Generic Support for Remote Memory Access Operations in Score-P and OTF2

Andreas Knüpfer, Robert Dietrich, Jens Doleschal, Markus Geimer, Marc-André Hermanns, Christian Rössel, Ronny Tschüter, Bert Wesarg, and Felix Wolf

Abstract Remote memory access (RMA) describes the ability of a process to access all or parts of the memory belonging to a remote process directly, without explicit participation of the remote side. There are a number of parallel programming models based on RMA operations that are relevant for High Performance Computing (HPC). On the one hand, Partitioned Global Address Space (PGAS) language extensions use RMA operations as underlying communication substrate, e.g. Co-Array Fortran and UPC. On the other hand, RMA programming APIs provide so called one-sided data transfer primitives as an alternative to the classic two-sided message passing. In this paper, we describe how Score-P, a scalable performance measurement infrastructure for parallel applications, is extended to support trace-based performance analyses of RMA parallelization models. Emphasis is given to the generic event model we designed to record RMA operations in the OTF2 trace format across a range of one-sided APIs and libraries.

1 Introduction

In high-performance computing (HPC), the prevalent programming paradigm has been the Message Passing Interface (MPI) for many years. Although MPI defines three communication paradigms, point-to-point, collective, and one-sided communication, the former two are widely used, whereas the latter is scarcely present in current HPC applications. The reasons for this imbalance can be manifold, as the one-sided communication interface was added with MPI 2.0 a year after the initial release of MPI, and many developers perceive the interface semantics to be unwieldy and complicated. Despite its minor role in current HPC applications, with

A. Knüpfer (✉) · R. Dietrich · J. Doleschal · M. Geimer · M.-A Hermanns · C. Rössel · R. Tschüter · B. Wesarg · F. Wolf
Center for Information Services and HPC (ZIH), TU Dresden, 01062 Dresden, Germany
e-mail: andreas.knuepfer@tu-dresden.de

the complexity of petascale computing systems in use and exascale systems on the horizon, the interest in one-sided communication is beginning to rise again. This is much due to Partitioned Global Address Space (PGAS) libraries and languages entering the HPC world, promising to reduce programming complexity while retaining performance levels of current applications. In the PGAS programming model, the developer can directly address variables on remote processes through a global shared memory view. Thus, the developer does not have to deal explicitly with data transfers. Instead, the PGAS runtime system manages data transfers among remote portions of the memory transparently to the user, reducing the source code complexity. The most prevalent PGAS representatives today are Unified Parallel C (UPC) [18] and Co-Array Fortran (CAF) [15].

Although showing promising results in terms of performance, all lack support for incremental parallelization starting from the prevalent HPC programming languages C and Fortran. However, with HPC application codes relying on code bases with several decades of history, the adoption of the PGAS concept is slow. As alternative approaches, one-sided parallelization models were developed which require neither language extensions nor special compilers. Examples are GASNet [2], ARMCI [13], GlobalArrays (GA) [12], SHMEM [14], GPI [11], Cray's DMAPP [19], and MPI 3.0 [17]. All provide APIs plus libraries for remote memory access (RMA) operations in a similar way as MPI provides point-to-point and collective communication operations. The RMA APIs promise an easier adoption by the HPC community compared to PGAS language extensions, yet there is no established quasi-standard dominating the field like MPI is for point-to-point and collective communication.

When various RMA parallelization models gain momentum in the HPC field, tool support for performance analysis and optimization becomes crucial. Yet, it is impractical to develop individual tools for all promising RMA libraries. Therefore, this paper presents a first step towards a generic performance tool infrastructure for RMA parallelization models. This is feasible because all build around the same fundamental semantics even though they provide different APIs. This leads to a number of common concepts of HPC RMA models and a generic event model for RMA operations together with its representation in the Open Trace Format Version 2 (OTF2). Performance tools will be able to build on top of this abstract model to provide abstract analysis functionality.

In the following section, we give a short overview about the Score-P/OTF2 event-based performance analysis infrastructure. After that, Sect. 3 introduces the common concepts in RMA libraries and Sect. 4 presents the design of the new generic RMA event types in OTF2. The paper ends with an outlook and conclusion.

2 Overview of Score-P and OTF2

The Score-P project started in 2009, funded by the German BMBF and the US DOE, and the entire performance measurement infrastructure is available as Open Source under the New BSD License.

2.1 *The Score-P Instrumentation and Measurement System*

The Score-P performance measurement system [10] is a highly scalable and user-friendly tool suite for profiling, event tracing, and online analysis of HPC applications. It is designed as a joint instrumentation and run-time data collection infrastructure for a number of performance analysis tools. Currently, Score-P supports the well-established analysis tools Periscope [1], Scalasca [7], VAMPIR [9], and TAU [16] and is open for other tools.

To collect performance data, e.g., times, visits, or communication metrics, measurement probes are inserted into the application code. These probes will collect performance-related data whenever they are triggered during measurement runs. Currently, Score-P supports the following instrumentation variants for C/C++ and Fortran codes:

- Compiler instrumentation,
- MPI library interposition,
- OpenMP source code instrumentation using OPARI2,
- Source code instrumentation using the TAU instrumentor,
- Binary instrumentation via COBI, and
- User instrumentation using convenient macros.

Score-P targets programs parallelized via MPI 2.1 and OpenMP 3.0 (incl. tied tasking) as well as hybrid combinations.

Besides the post-mortem analysis of the gathered performance data, Score-P offers an interface for on-line analysis. Using this interface, analysis tools can re-configure measurement parameters, retrieve profile data, and also interrupt and resume the application's execution.

2.2 *The Open Trace Format 2*

The Open Trace Format 2 [5] is a highly scalable, memory efficient event trace data format and support library. It is the common successor format for the Open Trace Format (OTF) [8] and the EPILOG trace format [20]. The analysis tools Scalasca, VAMPIR, and TAU already use OTF2 as their standard trace format. Nevertheless, OTF2 is open to support other tools.

The most important innovation of OTF2 over OTF and EPILOG is its scalability. Using a combination of online event-data and zlib compression reduces the trace file size considerably and thus defers potential buffer flushes. Leveraging the scalable parallel I/O library SIONlib [6], multiple task-local result files can be mapped onto one single or a small number of physical files, which results in reduced pressure on metadata servers during file creation. Finally, OTF2 avoids copying trace data during unification of parallel event streams.

2.3 Existing Event Record Types

The OTF2 format is based on two basic record types. On the one hand, there are definition records containing general information like subroutine names and processes/threads used. On the other hand, OTF2 provides event records. Currently, the following types of event records are supported:

Program flow: Events of this category are used to record program flow, e.g. entering or leaving subroutine calls.

Communication: The second category contains events like message transfers via point-to-point or collective communication.

OpenMP-related events: For OpenMP parallel programs OFT2 provides records to track creation and termination of threads as well as records associated with OpenMP locking routines. In order to support OpenMP 3.0 tasking, events for tracking task creation, completion, and task switches are provided.

Measurement control: We do not always need performance data of the full program run to narrow down performance bottlenecks. To reduce the amount of storage required, the user can therefore limit event recording to performance critical parts of code. For this purpose, the Score-P measurement system provides the option of temporarily switching off recording, which will later be indicated by corresponding records in the trace file.

Additional information: Often the information obtained from the previously mentioned event types is not sufficient for comprehensive performance analysis. For example, information on floating-point performance or cache misses helps to unveil unfavorable data access patterns. Sources of such information are hardware performance counters, whose values can be stored in metric records.

2.4 Current and Future Directions

Emerging new trends and architectures in HPC are continuously addressed by Score-P. New instrumentation and measurement features are being integrated during the continuous evolution of the trace and profile formats. However, to keep changes to the output formats moderate, we strive for generic event abstractions that can serve several programming models. Future Score-P releases will feature

- Sampling as alternative to instrumentation when obtaining performance data,
- A plugin-architecture to connect to a wide range of external metric sources,
- Improved threading support—the OpenMP support will be generalized and extended to address Pthread-, OmpSS-, and HMPP-based programming models,
- The compression of time-series call-path profiles to record the detailed performance dynamics even of long-running codes,
- And the reduction of I/O-demands of hybrid programming models on ever-growing HPC-systems through extended SIONlib support.

3 RMA in HPC Parallel Programming

Below, the most relevant one-sided communication interfaces in the HPC field will be introduced, followed by an overview about common terms and concepts.

3.1 RMA Operations in HPC Parallelization Libraries

The following RMA parallelization models were the starting points for the common concepts. PGAS language extensions are postponed to the outlook in Sect. 6.

MPI 2.0 and 3.0

When the extensions to the Message Passing Interface were introduced with MPI 2.0 in 1995, they included an interface for one-sided communication. As a third communication paradigm, it complemented the existing point-to-point and collective communication interfaces. Adoption in the user community as a direct user-level interface was very slow. Its interface was often regarded by users as too complicated, and Bonachea and Duell [3] showed that several aspects in the interface disqualify it for use as a runtime backend for PGAS languages. With MPI 3.0, the MPI Forum addressed these shortcomings to enable the interface to be used as a portable interface for PGAS compiler and library providers.

SHMEM and OpenSHMEM

The SHMEM interface was originally introduced by Cray as a shared memory programming interface for their supercomputing platforms. Although initially a proprietary interface, it has since been ported to different platforms. With OpenSHMEM [4] an open interface was introduced in 2010 that was explicitly designed to be used as the one-sided backend for PGAS compilers, such as the Open64 Co-Array Fortran compiler.

ARMCI and GlobalArrays

The Aggregate Remote Memory Copy Interface (ARMCI) [13] was designed and is still used as the one-sided communication backend for the Global Arrays Toolkit (GA) [12], a library for PGAS-like computation on distributed matrices.

GPI and GASPI

The Global Address Space Programming Interface (GPI) is a proprietary library for RMA parallelization developed and commercially distributed by the Fraunhofer Institut für Techno-und-Wirtschaftsmathematik (ITWM). Together with academical and industrial partners, they started a community-driven project in 2011 called The Global Address Space Programming Interface (GASPI). On the one hand, it strives for a standardization of RMA APIs for HPC parallelization. On the other hand, it will provide an Open Source reference implementation.

CUDA and OpenCL

The Compute Unified Device Architecture (CUDA) is a programming model developed by NVIDIA for their line of GPGPUs. The Open Computing Language (OpenCL) is an open standard for parallel programming for heterogeneous platforms, including GPU devices and other accelerator devices.

Both do not belong to the area of one-sided communication interfaces. Yet, they employ direct memory access (DMA) operations for the data transfer between hosts and devices which are very similar to the RMA operations relevant for this paper. It turns out, they can be covered with the same event model. Thus, this selected aspect of their APIs is also covered here, whereas all other aspects are left out.

3.2 Concepts in RMA Parallelization Models

The following concepts have been collected across all RMA models covered.

Communication Contexts

A communication context is the frame in which the remote side of an RMA operation (i.e. the target for write operations or the source for read operations) is specified. One could also call it a name space for the remote processes. Furthermore, synchronization and interaction between RMA operations are enclosed in communication contexts, e.g. the preservation of the order of operations. The best known example in the HPC field are MPI communicators, where the specification or the remote rank is relative to the communicator and the same rank number in different communicators may refer to different processes. Most RMA parallelization models have only a single global context but there are a few exceptions: First MPI, which uses its communicator concept also for one-sided operations. Second GPI and GASPI which have so called *communication queues*. In each queue the operations stay in-order, across queues this is not guaranteed. Finally, for CUDA and OpenCL there are only local contexts within every host in which the first, second, third, . . . local accelerator device can be addressed. With separate contexts, one can distinguish between all accelerator devices across multiple hosts.

Memory Windows

In most RMA models, only parts of the address space may be accessed by RMA operations. Such address ranges are called *memory windows*.

There are models that offer symmetric windows (SHMEM), where the start address of the window is the same on all processes. Others have non-symmetric ones (MPI, ARMCI). Some models allow only a single static window where all data for communication needs to be placed (GPI, GASPI) while others allow many fine-granular and dynamic windows (MPI, SHMEM). Finally, not all models have such a concept (CUDA).

Remote Memory Access Operations

The *get* and *put* primitives are the very essence of the RMA concept. They allow access to a remote memory location without the active participation from the other side.¹ Some interfaces explicitly distinguish between non-blocking calls, which just issue an operation, and blocking variants, which wait for local completion of the operation. Here, local completion means that following local operations cannot interfere with an operation in flight, e.g. by overwriting the source location. Remote completion can only be inferred via synchronization operations, see below.

Besides the basic get and put operations, *atomic* operations allow more sophisticated operations on remote memory. Sometimes they are called *read-modify-write* operations. Examples are *compare-and-swap* and *fetch-and-increment*. Atomic operations guarantee atomicity for the entire operation, which does not hold for successive get and put operations that imitate the same behavior on the same remote memory location. The same rules for non-blocking vs. blocking operations apply.

Synchronization

The completion of a remote memory access operation usually has only local scope. Therefore, additional synchronization primitives are provided to ensure local and remote completion. Unlike for P2P communication, there are two separate aspects of synchronization for RMA communication. One is the *execution synchronization*, which waits for remote operations to be finished before proceeding locally. The other is *memory synchronization* which waits for local or remote memory windows to be accessible again. Both can be used separately or combined.

The synchronization methods provided by the different RMA models fall into four categories: collective synchronization, group synchronization, notification, and locking. Collective synchronization is performed among all

¹Sometimes they are called *read* and *write* instead, but *get* and *put* are the typical terms. To avoid confusion, *load* and *store* are used explicitly for local memory accesses.

processes of the communication context. Examples include `MPI.Win_fence` and `ARMCI.Barrier`. Note that all processes in the communication context need to participate, regardless of whether they issued RMA operations themselves (i.e. active ones) or whether they were target of any (i.e. passive ones).

Synchronization among a sub-group of a communication context is much more lightweight than collective synchronization in the entire context. The sub-groups don't need to be collectively defined, but again all active and passive processes need to participate. This kind of synchronization is sometimes used to define so called *phases* in which the access to a memory window is restricted to either only local or only remote access.

Notification is another flavor which is useful for pairwise synchronization. The typical procedure is that a process waits for a change at a local memory address which is eventually triggered by an RMA operation from a remote process.

Finally, synchronization via locking is a method to coordinate mutual exclusive access to either entire memory windows or parts thereof. There are normal locking models where the acquisition of a lock may be successful, unsuccessful, or block until it succeeds. In addition, MPI has a special mode of issuing a locking request, which is guaranteed to be fulfilled in the future. Afterwards, non-blocking RMA operations may be issued and are automatically postponed until the lock is available.

Collective Communication

Collective communication operations combine data from a group of processes in predefined or self-defined ways. These are, in particular, reduction operations such as sum, maximum, or minimum. Such operations could be implemented manually using either basic get and put operations or using point-to-point send and receive operations. Therefore, this is no inherent RMA concept. Still, it is covered with new event types below, because collective operations influence the synchronization within RMA based parallel applications.

4 Generic RMA Event Types

This section presents and explains the RMA event record types distilled out of the set of RMA programming models previously covered. They relate directly to the common concepts identified before.

4.1 RMA Window Handling

The memory window concept is directly represented by a definition record and two event records that mark the creation and destruction of the window.

The *OTF2_GlobalDefRmaWin* record defines a new window with a unique identifier, a free form name, and a reference to a communicator. The communicator has to be defined earlier and references all participating processes or threads as well as the actual RMA model (MPI, GASPI, ...).

OTF2_GlobalDefRmaWin

OTF2_RmaWinRef	self	the new window ID being defined
OTF2_StringRef	name	a free form name, e.g. "MPI window" or "Gfx Card 1"
OTF2_CommRef	comm	underlying communicator

With this definition, the *OTF2_RmaWinCreate* or *OTF2_RmaWinDestroy* events can be generated. They mark the location and time of the creation resp. destruction of the window on all participating processes/threads and thus enclose all operations related to this window.

OTF2_RmaWin(Create|Destroy)

OTF2_LocationRef	location	Process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window

4.2 Specification of the Passive Side

RMA event records are usually recorded at the active process or thread, i.e. where they are issued with a RMA API call. In an event record the active side is referenced with an *OTF2_LocationRef* entry. This is not possible in the same way for the passive side, because that are entire address spaces instead of processes or threads.

As a flexible solution, the RMA event records represent the passive sides of RMA operations as a pair of an index *target* and a memory window *window*. The *window* refers to a communicator definition which in turn refers to all processes or threads in the communicator. The *index* can be used to identify one or all threads that belong to the passive side's address space. Usually, it is the same number that was passed to API call which issued the operation. This scheme is used for all following records.

4.3 Get and Put

The *get* and *put* operations access remote memory addresses. The corresponding *get* and *put* records mark when they are issued. The actual start and the completion may happen later.

OTF2_Rma(Put|Get)

OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint32_t	target	rank of target in context of window
uint64_t	size	number of bytes transferred
uint64_t	matching	matching number

The matching number allows to reference the point of completion of the operation. It will reappear in a completion record on the same process or thread (see Sect. 4.5).

This can mean different things: For *put* operations, the local completion means that the data is sent off and the source address range may be modified again without affecting the RMA operation. Remote completion means that the new data is ensured to be visible when read on the passive target process. Completion of *get* operations means that the data is available at the local target address, i.e. the fetched data is safe to be used. Remote completion is implied in this case.

4.4 Atomic RMA Operations

The atomic RMA operations are similar to the *get* and *put* operations. As an additional field they provide the type of operation from the following set, which may be extended in the future.

OTF2_RmaAtomicType

OTF2_RMA_ATOMIC_TYPE_ACCUMULATE	accumulate
OTF2_RMA_ATOMIC_TYPE_FETCH_AND_INCREMENT	fetch and add
OTF2_RMA_ATOMIC_TYPE_TEST_AND_SET	test and set
OTF2_RMA_ATOMIC_TYPE_COMPARE_AND_SWAP	compare and swap

Depending on the type, data may be received, sent, or both, therefore, the sizes are specified separately. Matching the local and optionally remote completion works the same way as for *get* and *put* operations.

OTF2_RmaAtomic

OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	window
uint32_t	target	rank of target in context of window
OTF2_RmaAtomicType	type	type of atomic operation
uint64_t	size_sent	number of bytes transferred to target
uint64_t	size_received	number of bytes transferred from target
uint64_t	matching	matching number

4.5 Completion Records

The completion records mark the end of RMA operations. Local completion for every RMA operation (get, put, or atomic operation) always has to be marked with either *OTF2_RmaOpCompleteBlocking* or *OTF2_RmaOpCompleteNonBlocking* using the same matching number as the RMA operation record. An RMA operation is blocking when the operation completes locally before leaving the call, for non-blocking operations local completion has to be ensured by a subsequent call.

OTF2_RmaOp(Test)|(Complete(Blocking|NonBlocking|Remote))

OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint64_t	matching	matching number

The *OTF2_RmaOpTest* record indicates a test for completion. This is only useful for non-blocking RMA calls where the API supports such a test. The test record stands for a negative outcome, otherwise a completion record is written. An optional remote completion point can be specified with *OTF2_RmaOpCompleteRemote*. The *OTF2_RmaOpCompleteRemote* record is recorded on the same process as the RMA operation itself. Again, multiple RMA operations may map to the same *OTF2_RmaOpCompleteRemote*. The target processes are not explicitly specified but implicitly as all those that were referenced in matching RMA operations. There is no counterpart for *OTF2_RmaOpTest* for remote completion, because no RMA model provides such a primitive. A completion record marks all RMA operation records (get, put atomic operation) that used the same matching number as completed in the respective scope. Using the distinct matching numbers for every operation implies a 1:1 relationship between operation and completion records. Where semantics permit it the same matching number can be used for subsequent operations, leading to an n:1 relationship between operation and completion records, effectively reducing the number of records in the trace.

4.6 Notification via RMA

The *OTF2_RmaWaitChange* event marks a synchronization point that blocks until a remote operation modifies a given memory field (see ‘notification’ in Sect. 3.2).

OTF2_RmaWaitChange

OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window

This event marks the beginning of the waiting period. The memory field in question is part of the specified window. Its address and length is not stored

because different RMA models specify them differently. It can be added as optional model-specific key-value attributes. The source process/thread of the modification cannot be determined in general and is therefore omitted.

4.7 Synchronization

Synchronization plays an important role in all RMA models and can mean two different things: On the one hand, the execution of processes/threads are synchronized, such that they cannot proceed before all processes arrive. This is the notion of a barrier for parallel execution. On the other hand, memory areas of RMA operations “in flight” are synchronized. This does not necessarily imply that the processes/threads are synchronized. (Yet, synchronization records do not replace completion records which must be written in addition.)

The following four synchronization levels cover all combinations of the two:

enum OTF2_RmaSyncLevel	
OTF2_RMA_SYNC_LEVEL_NONE	no process synchronization or access completion, e.g. MPI.Win.post/start
OTF2_RMA_SYNC_LEVEL_PROCESS	synchronize processes, e.g. MPI.Win.create/free
OTF2_RMA_SYNC_LEVEL_MEMORY	complete memory accesses, e.g. MPI.Win.complete/wait
OTF2_RMA_SYNC_LEVEL_ALL	complete memory accesses and synchronize processes, e.g. MPI.Win.fence

The first synchronization record type is for simple pairwise synchronization:

OTF2_RmaSync		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint32_t	target	rank of target in context of window
OTF2_RmaSyncLevel	sync_level	synchronization level

The second record type synchronizes a sub-group of the processes associated with the given memory window. It needs to be recorded for all participants.

OTF2_RmaSyncGroup		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaSyncLevel	sync_level	synchronization level
OTF2_RmaWinRef	window	memory window
OTF2_GroupDef	group	group of participating processes or threads

Further means of synchronization are RMA collective operations, see below.

4.8 Collective Operations and Synchronization

Many RMA models provide collective operations. They are similar to the MPI collectives and as such not specific to the RMA concept. But they are also part of the RMA models and affect the synchronization with RMA operations. Therefore, separate records are defined which refer to the synchronization level introduced before.

The following event records for collective RMA operations must be generated on all participating members of the communicator that is referenced from the memory window. On all processes, a *OTF2_RmaCollectiveBegin* event record with only the location and time must be followed by a *OTF2_RmaCollectiveEnd* event record with all details. It is invalid to intermix or nest begin and end records of different collective operations, but local or remote completion records may be placed in between. If there is a root process, it must be specified in *root* which is a process rank relative to *window*. Finally, the amount of data send or received is given.

OTF2_RmaCollectiveBegin

OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp

OTF2_RmaCollectiveEnd

OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaSyncLevel	sync_level	synchronization level
OTF2_RmaWinRef	window	memory window
uint32_t	root	root process/rank if there is one
uint64_t	size_sent	number of bytes sent
uint64_t	size_received	number of bytes received

4.9 Locking of Resources

Another basic concept is locking and unlocking of shared resources in order to ensure data consistency during concurrent accesses. A lock can be exclusive, i.e., only one participant may hold it at a time. This is necessary for competing write accesses. Or it can be shared, i.e., many participants may hold it. This can be allowed for simultaneous read access. The following lock types are defined in OTF2 to distinguish the two.

OTF2_LockType

OTF2_LOCK_TYPE_EXCLUSIVE	only one lock allowed at the same time, e.g., write-lock, mutex, MPI exclusive lock
OTF2_LOCK_TYPE_SHARED	multiple shared locks allowed at the same time, e.g., read-lock, MPI shared lock

For the actual locking and unlocking, four record types are defined. All are associated to a memory window on a target process, which either is the subject of the lock or contains it. In addition, a *lock_id* can be specified to describe different targets in the same window. The *lock_id* is simply a number with no semantics and there is no definition record for it.

For the actual lock event, there are three separate record types with slightly different semantics. First, the *OTF2_AquireLock* marks the time that a lock is granted. This is the typical situation. It has to be followed by a matching *OTF2_ReleaseLock* record later on. Second, an attempt to acquire a lock which turns out negative can be marked with *OTF2_TryLock*. In this case, no release record may follow. With this a series of unsuccessful locking attempts can be identified. If an lock attempt is successful, it is marked with *OTF2_AquireLock* right away instead of a pair of *OTF2_TryLock* and *OTF2_AquireLock*. And third, the *OTF2_RequestLock* record marks the time that a request for a lock is issued where the RMA model ensures that the lock is granted eventually without further notification. As of now this is specific for MPI. In this case, the *OTF2_AquireLock* event is not present.

Finally, the *OTF2_ReleaseLock* marks the time the lock is freed. It contains all fields that are necessary to match it to either an earlier *OTF2_AquireLock* or *OTF2_RequestLock* event and is required to follow either of the two.

OTF2_(Aquire Try Request)Lock		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	memory window
uint32_t	target	rank of target in context of window
OTF2_RmaLockType	lock_type	Type of lock (shared vs. exclusive)
uint64_t	lock_id	lock id in context of window
OTF2_ReleaseLock		
OTF2_LocationRef	location	process or thread of execution
OTF2_TimeStamp	time	time stamp
OTF2_RmaWinRef	window	
uint32_t	target	rank of target in context of window
uint64_t	lock_id	lock id in context of window

5 Example Cases with RMA Event Types

In the following, a number of examples illustrate how typical situations with selected RMA parallelization models would be captured in an OTF2 event trace.

Figure 1 shows three aspects with an example scenario for MPI. The first aspect is the creation of a memory window in the beginning and the destruction in the end. For both activities, the MPI call is recorded (*MPI_Win_create* and *MPI_Win_free*) on every process with Enter (E) and Leave (L) events (which contain the time, the process/thread, and the function being called). Inside the *CollectiveBegin* (Cb) and

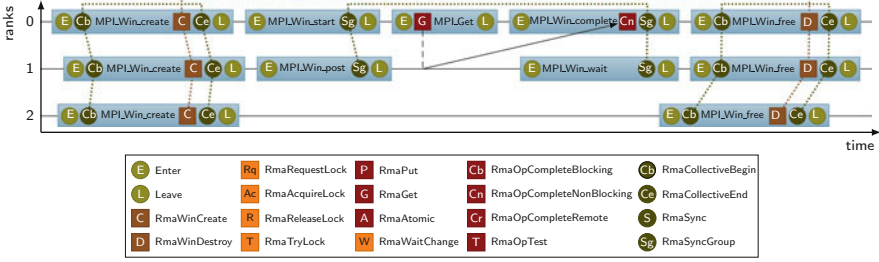


Fig. 1 Timeline example for *OTF2_RmaWinCreate* and *OTF2_RmaWinDestroy* in MPI

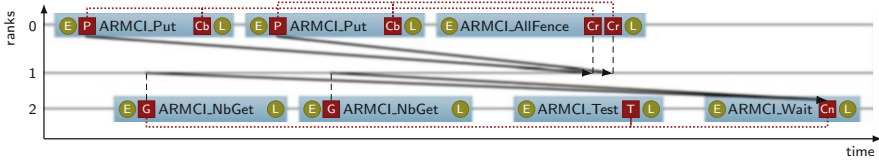


Fig. 2 Example of the use of completion events for blocking and non-blocking RMA operations in ARMCI. It uses the same legend as Fig. 1

CollectiveEnd (Ce) events mark them as collective calls and the actual *WinCreate* (C) or *WinDestroy* (D) events are the earliest or latest points when the defined memory window may be used, respectively. The second aspect shown in Fig. 1 is an example of MPI’s “general active target synchronization” [17]. Rank 0 plays the active role using *MPI_Win_start* and *MPI_Win_complete* and rank 1 is the passive side which still needs to take part in the synchronization with *MPI_Win_post* and *MPI_Win_wait*. All record group synchronization events (Sg). The third aspect is the connection between a RMA *get* event (G) inside the *MPI_Get* call and its completion point (Cn) inside *MPI_Win_complete*. Both are only present on the origin process.

The second example in Fig. 2 presents typical put and get events, here with the ARMCI RMA model. On rank 0 two blocking put (P) operations are issued. Their local completion (Cb) is recorded inside the same API call. Their remote completion points (Cr) are recorded in the *ARMCI_AllFence* call later on—this is the time where the arrows for the data transfer should end. On rank 2 two non-blocking get operations (G) are issued followed by a (negative) test (T) for the completion. Both get operations are connected to the same local completion event (Cn). Here, no remote completion event is necessary.

The third example in Fig. 3 shows locking of memory windows with SHMEM (top) and MPI (bottom)—Cray SHMEM actually allows mixing with MPI and uses the ranks from *MPI_COMM_WORLD*. The SHMEM case shows how acquiring (Ac) and releasing (R) a lock is serialized over ranks 0 and 1 as one would expect. The MPI case shows the special variant for MPI requesting a lock (Rq) which can happen (almost) simultaneously (ranks 2 and 3). The actual lock will be assigned

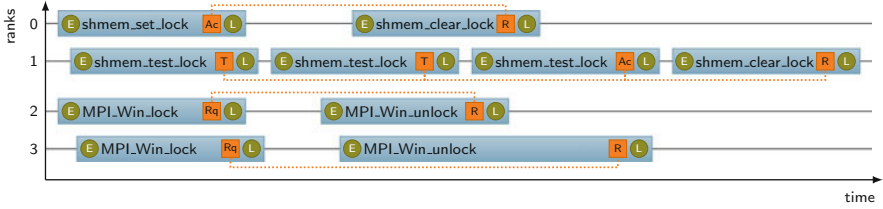


Fig. 3 Example of the different uses of lock events for SHMEM (top) and MPI (bottom)

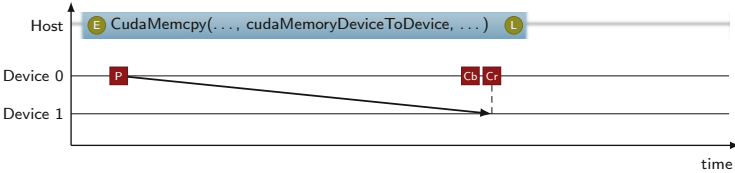


Fig. 4 Example of the a device-to-device memory transfer in Cuda

to the ranks later without a general way to observe when this happens. The only indication of a serialized execution on the locked memory window is the difference in the time when the associated release events (R) take place.

The final example in Fig. 4 shows how the generic RMA event types are also used for the CUDA data transfers. All operations are issued by a host process. A typical host-to-device or device-to-host transfer can be recorded like a normal put or get event similar to the example in Fig. 2. The scenario shown includes three sides, the active host issuing a device-to-device transfer and two passive devices which are the source resp. the target of the operation. Since the host is recording all events for the devices, it is feasible that the active side (Host) places events in one of the passive sides (Device 0). The recommended way is to record a put event (P) together with completion events (Cb,Cr) for the source as shown in Fig. 4. As an alternative, the target could record a get event.

6 Conclusions and Outlook

In this paper, we presented a generic representation of parallelization models based on the remote memory access (RMA) paradigm in the event tracing mode of the Score-P measurement infrastructure. It focuses only on RMA-models relevant in the field of high-performance computing. Starting from the concepts common to all those models, a set of event record types was designed that covers all relevant RMA models and they were illustrated with four examples.

With this background, the performance analysis of RMA patterns will become possible on a generic level, i.e without having to know the specific underlying RMA model. Examples are the evaluation of data transfer volumes between

processes/threads and the total waiting time at collective synchronization points due to load imbalances. The presented event model is not bound to one of the RMA models, unlike the representation of point-to-point communication which is inherently MPI specific—which is less critical because it is the only relevant model for HPC.

The following steps of the ongoing collaborative development will be to implement the instrumentation layers for the RMA libraries in the order of actual demand, starting with CUDA/OpenCL followed by GASPI or MPI 3.0 depending on when their official versions will be available. Afterwards, RMA-specific analysis functionality will be implemented for the analysis tools based on Score-P. In particular, this includes the graphical representation and the visual analysis of VAMPIR and the automatic replay-based analysis of Scalasca.

A topic of future work will be how to incorporate PGAS language extensions with the same event model. Even though this is generally feasible, it poses two challenges. First, the instrumentation may be more difficult because PGAS compilers insert RMA operations in a transparent fashion. This can be solved rather easily in all cases where RMA operations are mapped to an underlying library such as GASNet or Cray's libonesided. Second, the occurring RMA operations are more difficult to be related to the source code because there are no explicit RMA statements. Furthermore, there is no direct way to predict how a PGAS compiler will interpret a modified source code after a change (optimization) due to the analysis of the original code. In particular with optimizing PGAS compilers, this may result in complex behavioral changes in the application's execution.

References

1. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: An online-based distributed performance analysis tool. In: M.S. Miller, M.M. Resch, A. Schulz, W.E. Nagel (eds.) *Tools for High Performance Computing 2009*, pp. 1–16. Springer, Berlin/Heidelberg (2010). URL http://dx.doi.org/10.1007/978-3-642-11261-4_1
2. Bonachea, D.: GASNet Specification, v1.1. Tech. rep., University of California, Berkeley (2002). URL <http://techreports.lib.berkeley.edu/accessPages/CSD-02-1207>
3. Bonachea, D., Duell, J.: Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking* **1**(1–3), 91–99 (2004). DOI 10.1504/IJHPCN.2004.007569. URL <http://portal.acm.org/citation.cfm?id=1359705>
4. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10*, pp. 2:1–2:3. ACM, New York, NY, USA (2010). DOI 10.1145/2020373.2020375. URL <http://doi.acm.org/10.1145/2020373.2020375>
5. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open Trace Format 2 – The next generation of scalable trace formats and support libraries. In: *Proc. of the Intl. Conference on Parallel Computing (ParCo)*, Ghent, Belgium, August 30–September 2, 2011, *Advances in Parallel Computing*, vol. 22, pp. 481–490. IOS Press (2012). DOI 10.3233/978-1-61499-041-3-481

6. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA. ACM (2009). DOI 10.1145/1654059.1654077
7. Geimer, M., Wolf, F., Wylie, B.J., Abraham, E., Becker, D., Mohr, B.: The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience* **22**(6), 702–719 (2010). DOI 10.1002/cpe.1556
8. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Computational Science ICCS 2006: 6th International Conference, LNCS 3992. Springer, Reading, UK (2006)
9. 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)
10. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmid, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proc. of 5th Parallel Tools Workshop, Dresden, Germany (2011)
11. Machado, R., Lojewski, C., Abreu, S., Pfreundt, F.J.: Unbalanced tree search on a manycore system using the gpi programming model. *Computer Science – R&D* **26**(3–4), 229–236 (2011). URL <http://dblp.uni-trier.de/db/journals/ife/ife26.html#MachadoLAP11>
12. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global Arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.* **10**, 169–189 (1996). URL <http://portal.acm.org/citation.cfm?id=243179.243182>
13. Nieplocha, J., Tipparaju, V., Krishnan, M., Panda, D.K.: High performance remote memory access communication: The ARMCI approach. *Int. J. High Perform. Comput. Appl.* **20**, 233–253 (2006). DOI 10.1177/1094342006064504. URL <http://portal.acm.org/citation.cfm?id=1125980.1125986>
14. Poole, S.W., Hernandez, O., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: Openshmem – toward a unified rma model. In: D.A. Padua (ed.) *Encyclopedia of Parallel Computing*, pp. 1379–1391. Springer (2011). URL <http://dblp.uni-trier.de/db/reference/parallel/parallel2011.html#PooleHKSCF11>
15. Reid, J.: Coarrays in the next fortran standard. *SIGPLAN Fortran Forum* **29**, 10–27 (2010). DOI 10.1145/1837137.1837138
16. Shende, S., Malony, A.D.: The TAU Parallel Performance System, SAGE Publications. *International Journal of High Performance Computing Applications* **20**(2), 287–331 (2006)
17. The Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.0 (Draft Aug. 2012). Tech. rep. (2012). Aug. 2012
18. UPC Consortium: UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab (2005). URL <http://www.gwu.edu/~upc/publications/LBNL-59208.pdf>
19. Vishnu, A., ten Bruggencate, M., Olson, R.: Evaluating the potential of cray gemini interconnect for pgas communication runtime systems. In: High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on, pp. 70–77 (2011). DOI 10.1109/hoti.2011.19
20. Wolf, F., Mohr, B.: EPILOG Binary Trace-Data Format. Tech. Rep. FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich (2004)

Cache-Related Performance Analysis Using Rogue Wave Software's ThreadSpotter

Royd Lüdtkke and Chris Gottbrath

Abstract Modern processors cannot deliver high performance without applying caching mechanisms. However, the cache-conscious programming requires from the developer quite a deep knowledge about the underlying processor's hardware architecture and is thus very hard to be adopted by the software codes. The cache-aware application optimization is getting even more challenging for the parallel (multi-threaded) applications running in multi-processor and/or multi-core environments. We introduce the Rogue Wave Software's ThreadSpotter performance analysis tool, which is designed to simplify the cache-aware application development by leveraging the unique performance optimization techniques. Following an original statistical approach, ThreadSpotter enables the in-depth application analysis on the wide range of hardware platforms.

1 Introduction

Application optimization is always challenging, in particular when considering such hardware-related features as caching. This paper focuses on the impact of applying the cache utilization technique on the application performance optimization. As an easy-to-use profiling tool, ThreadSpotter, developed by Rogue Wave Software, offers a lot of opportunities for cache-related performance optimization. The paper starts by giving a brief overview on the caching technology in Sect. 2, also including the discussion on the relevant metrics. Section 3 discusses ThreadSpotter's statistical approach for the applications analysis and introduces the basics of the tool's usage. Section 4 discusses the performance issues identified by ThreadSpotter. Section 5 concludes the paper.

R. Lüdtkke (✉) · C. Gottbrath
Rogue Wave Software GmbH, Robert-Bosch-Strasse 5c, 63303 Dreieich, Germany
e-mail: luedtke@roguewave.com

2 Basic Overview on Caching

This section is dedicated to the discussion on the major quality metrics of the application's cache utilization.

2.1 Motivation for Caching

From a processors point of view, high-performance processing of a program is strongly dependent on memory access time. In turn, that time is related to memory type, memory size, and a machines architecture. There are two main kinds of memory that are used in computers. Each has different characteristics that make them ideal for playing different roles in the memory system as a whole.

For the main memory purposes, DRAM's are used because of their low space requirement due to a high integration level. Unfortunately, their use of capacitors for storage causes a relatively long access time. In contrast to DRAM, SRAM's are based on bistable multivibrators (flip-flops). These act much faster, but require a considerably larger square footage on the chip as they consist of about six transistors per bit ($\sim 140 F^2$), compared to the one transistor and one capacitor per bit ($\sim 6-10 F^2$) on which DRAM's are based. Regardless of a much higher cost of production, as well as non-economical power consumption, SRAM's of several GB would take much more space on the motherboard. The longer average distance to the processor would compensate for the advantage of faster access time.

Another factor on the main memory's response time is the computer's architecture. von Neumann architectures are based on a bus system, which in general suffers from limited bandwidth (von Neumann bottleneck). Therefore buffering (caching) of often used data in very small, but very fast memory is a good compromise. Cache memory is usually based on SRAM technology, ideally having a small size in the range of a few kB and located directly on the processor's chip, in order to allow very fast accesses.

2.2 Cache Architectures

Most systems have a cascading set of two or three cache buffers of ascending sizes. The first level cache (L1\$) is the smallest and fastest. Usually the majority of reused data (e.g. in program loops) does not exceed the size of L1\$. However, if this happens, the next higher level cache will take over. The caches L2\$ and L3\$ (if available) are slower because their access times are related to their size, but accessing them is still much faster than accessing main memory.

The typical cache sizes are:

- L1\$: $\sim 4-256$ kB per core;
- L2\$: $\sim 64-512$ kB per core;
- L3\$: $\sim 2-32$ MB shared by all cores.

2.3 *Cache Organization*

Caches store chunks of data called cache lines. A cache line is the smallest unit of data a cache is able to address. Cache line sizes depend on the processor type. Common sizes are 64 and 128 bytes. Since the number of memory blocks in main memory is much larger than the number of cache lines, cache memory must be organized in such a way that every memory block can be mapped on one or more cache lines. The choice of the mapping defines the so-called cache associativity. That means that cache lines are labeled by tags, which are referring to addresses of main memory. This architecture offers a very fast way for finding out whether requested data is already cached or not.

2.4 *Prefetching*

The decision on what kind of data has to be copied from main memory to the cache is made by the algorithm called the prefetcher. The prefetcher predicts the memory addresses the processor will most likely access in the near future based on constant-stride patterns it has identified. This works well in the case when strides have a fixed length, e.g. if an array will be populated element by element, in ascending order, or inside of a loop. If the application accesses data randomly, or in a way the prefetcher interprets as random, it would be prevented from working correctly. This could even lead to caches filling up with unusable data. The likelihood of cache misses will increase which will result in time consuming accesses to main memory.

Modern processors have hardware prefetching algorithms already implemented on chips. Another common method is controlling prefetching by software where the developer or the compiler places prefetch instructions in the code. Usually it is recommended to rely on the processors hardware prefetching algorithms. Only if the application needs to access data irregularly does it make sense to assist the prefetcher by using software prefetch instructions.

2.5 *Eviction of Cache Lines (Replacement Policies)*

There are a variety of more or less effective strategies for selecting cache lines in order to evict them from cache. Below is a list of the most common strategies:

- LRU (Least Recently Used): The least recently used cache line will be evicted. This is the most common strategy used in modern processors,
- Random: A random cache line will be replaced,
- FIFO (First In First Out): The oldest entry will be removed,
- LFU (Least Frequently Used): The least requested cache line will be evicted.

2.6 *Complexity Added by Coherence*

If multithreaded applications are executed on multicore or multiprocessor environments, caching becomes more complex. Each thread uses its own cache in order to achieve optimal concurrency. This means that if one thread changes data in its own cache by initiating a write operation, all other caches that have stored the same data in their private caches will need to be informed that their data has become invalid. The most common technique modern microprocessors leverage for keeping all caches synchronized is the so-called MESI coherence protocol (MSI, MOSI, MOESI, MERSI, MESIF operate similarly). Each cache line will get a state described by flags.

The MESI Cache Line States are as follows:

- **Modified:** the data in this cache has been modified. There are no copies in other caches. In case it has to be evicted or requested by another thread, it needs to be written back to main memory.
- **Exclusive:** only one copy of the cache line exists, which is the one located in this cache. The data has not been modified.
- **Shared:** the cache line is still in an unmodified state. There might be copies in other caches.
- **Invalid:** the cache line contains invalid data.

It is obvious that such a technique requires a lot of inter-cache communication and data interchange. A detailed description of how the MESI protocol works would go beyond the scope of this article.

2.7 *Important Statistics*

The following statistical metrics describe the quality of an applications cache utilization:

- **Miss ratio** – the percentage of memory accesses that causes a miss in the cache.
- **Fetch ratio**¹ – the percentage of the applications memory access statements that causes an access to main memory. Due to prefetching, the miss ratio will usually be lower than the fetch ratio. The fetch ratio is a direct rate for the applications bandwidth requirement concerning read accesses.
- **Fetch utilization**¹ – the average percentage of data the cache lines use before the cache lines are evicted from cache. A low fetch utilization means the prefetcher is loading a lot of unused data from main memory into cache and is consuming bandwidth unnecessarily.

¹This statistical metric is introduced by Rogue Wave Software.

- Writeback utilization¹ – the fraction of a cache line which is written back to main memory that has actually been changed.
- Communication ratio – the percentage of the applications memory accesses that cause communication between threads.
- Communication utilization¹ (of a multithreaded application) – means that only a small fraction of a cache line sent by a thread is really used by a receiver (consumer) thread. It can be seen as a rate for multithreading efficiency.
- Upgrade ratio – the number of upgrades to a cache lines state in comparison to the whole number of memory accesses. Upgrade means that a memory access will cause a cache line to change its state either from “Shared” to “Exclusive” or “Modified”.

Assume a thread loads a cache line into its cache that would get a Shared state, because a copy is already stored in another cache. If that thread modifies the data in the cache line by performing a write access, the state of the cache line will be upgraded. On the other hand, a cache line that is in an Exclusive or Modified state would be downgraded to a Shared state if another thread performs a read request on that data. It would get upgraded again if the first thread writes to that cache line.

2.8 Optimal Cache Utilization

The remarks above provide a brief idea of how caching works in general. In order to take the best advantage of the cache system and run an application with maximum performance, programmers need to keep the cache “hot” by following three simple rules:

- Try to let the processor get as much data as possible from the cache and not from main memory (assist the prefetcher, don't confuse it).
- Try to ensure that every fetched cache line will have 100 % of its data used.
- When possible, let each thread do work that makes use of data already stored in its local cache (and avoid using data that resides in another thread's cache).

As simple as these rules are, it is complex to implement programs that will not disregard them in some cases. Therefore performance analysis tools like ThreadSpotter can assist programmers in successfully finding places in their code that present opportunities for optimization (aka slow spots).

2.9 What Performance Improvement Is Possible When Optimizing an Application's Cache Utilization?

This question cannot be answered in general. Even huge applications which process small amounts of data will not benefit from optimizations if the whole data set fits into the cache's size. Likewise, very small improvements would be seen if

an application mostly deals with input and output, without performing repeated accesses to the same data (e.g. load balancers). The influence of caching in this case is marginal.

On the other hand, applications that process huge amounts of data by performing many r/w memory accesses could have improved throughputs ranging from 2 to 7 times faster than before optimization; sometimes just by modifying a small piece of code. With regard to those applications, cache related performance analysis tools are always a valuable investment.

3 ThreadSpotter: A Statistical Approach for Cache-Related Profiling

This section will focus on different approaches profiling tools use for benchmarking an application. In addition, the technique that ThreadSpotter utilizes will be discussed, including a brief description of its usage.

3.1 Different Approaches of Cache Related Performance Analysis

In general, there are two approaches profiling tools use in order to gather data and create reports. The first, very common approach is reading hardware counters. Tools that use this method usually have a very low runtime overhead. Unfortunately, they are not very flexible because the data they take as a basis for creating reports is strongly dependant on the hardware parameters of the system from which they have retrieved the data. The second approach is using simulators. Simulators are not bound to specific hardware and therefore, they are very flexible. The flip side is that depending on the amount of data they have to process, they are usually slow and their results are only as reliable as the underlying model.

In contrast to both these approaches, ThreadSpotter utilizes statistical models based on samples of the running processes' memory access behavior. Depending on the sample rate, the performance overhead can be very low. Based on the gathered data (fingerprint of the sampled application), ThreadSpotter is able to create a report containing information on where performance optimizations might exist. This approach provides the most flexibility because it is possible to create reports for different target systems based on the same fingerprint (e.g. different number of processors, cores, caches, and cache sizes).

Since only the binary is analysed, ThreadSpotter is usable for all applications based on compiled languages.

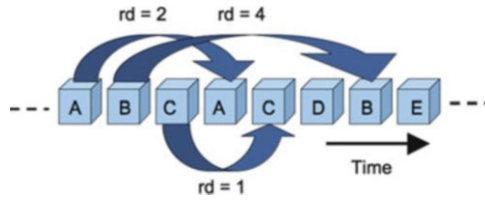


Fig. 1 The reuse distance d is the number of interim data accesses until a cache line sized data pattern will be reused (Adopted from [1])

3.2 What Kind of Data Is ThreadSpotter Looking at in Order to Create a Report?

ThreadSpotter looks at the accesses to memory a program performs during execution. It tries to identify patterns of the same data that is requested several times while sampling the address stream. It also takes metrics regarding the distances of occurrence, see Fig. 1.

If a datum “A” is fetched a second time and in between the first and second fetch, two fetches to other data occurs, the reuse distance would be $rd = 2$. Based on the detected reuse distances combined with cache properties like size and cache line length, ThreadSpotter is able to calculate the miss ratio. The fetch ratio can be derived from the miss ratio by considering the effect of prefetching. Realistic models of the target system’s cache architecture enable ThreadSpotter to create surveys about the number of cache misses, percentage of cache line’s data utilization, and other cache relevant metrics.

3.3 Sampling an Application

ThreadSpotter was designed to be simple and intuitive. It provides a graphical user interface, but the tool’s functionality can be invoked via command line as well. In order to execute the application and to start sampling, the sampler simply takes the name and arguments of the application to be sampled as its arguments. It is also possible to attach the sampler to an already running process (see Fig. 2).

In some cases it will only be necessary to just sample a portion of the whole application. This is made possible by providing the sampler with several start and stop conditions (see Fig. 3). The sample rate will automatically be adjusted to what works well in most cases.

In order to meet special requirements, e.g. for very short running applications, it is also possible to provide a sample period (see Fig. 4).

Thread Spotter also provides special scripts for sampling MPI applications.

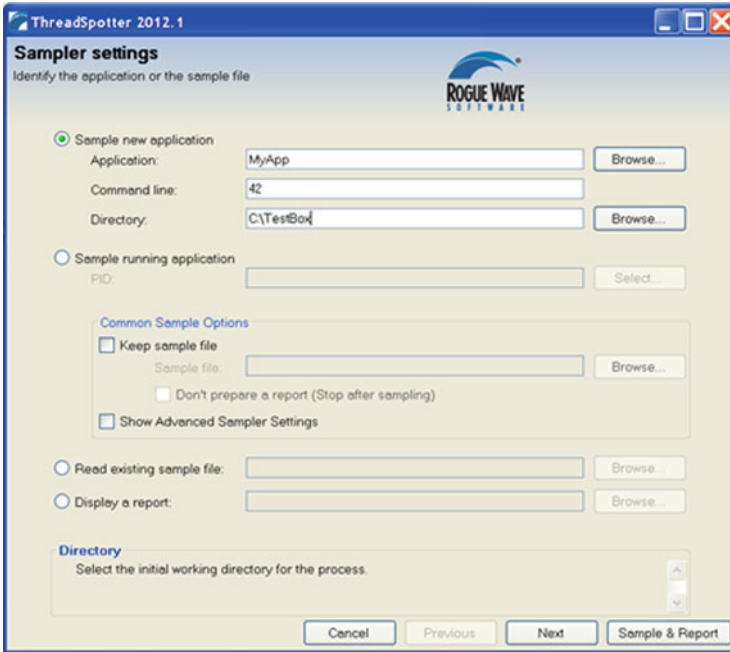


Fig. 2 ThreadSpotter's sampler settings

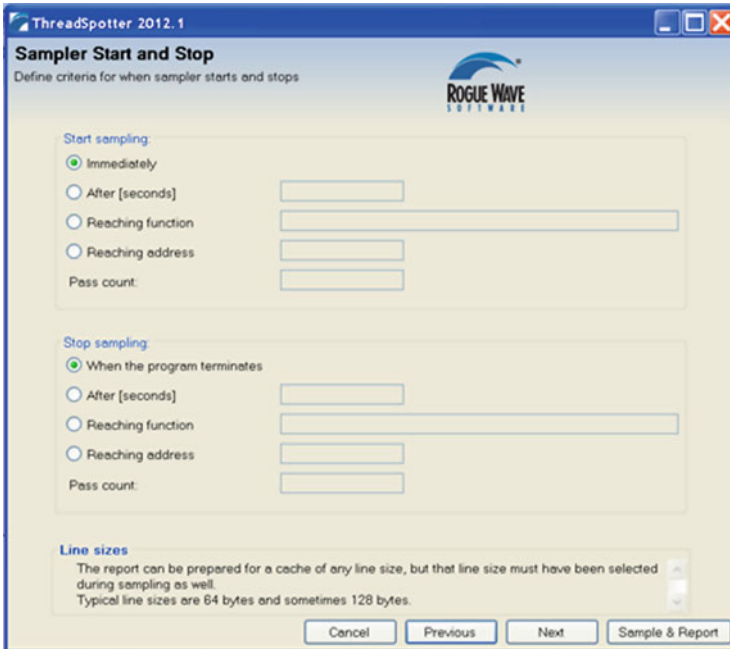


Fig. 3 ThreadSpotter's sampler allows defining start and stop criteria in order to sample slices of an application

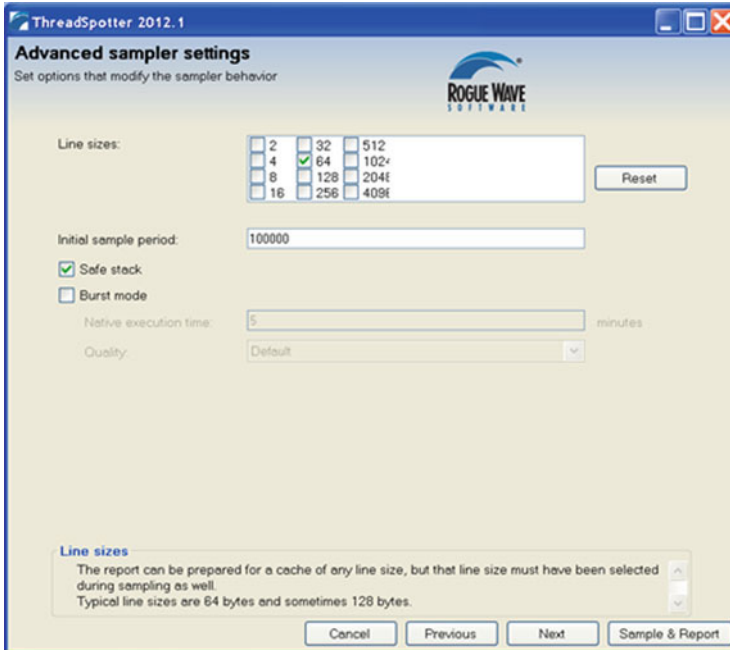


Fig. 4 The sample period can be modified to take enough samples for short running applications

3.4 Report Generation

ThreadSpotter generates platform independent reports based on fingerprint files that were previously established by the sampler. These reports do not necessarily have to be created for the same environment and processor architecture on which the sampling was made. Such flexibility is very useful because the production machines will often times differ from the development machines.

If a developer has implemented an application on his laptop (e.g. Intel CPU T2500, 2 cores, 2×32kB L1\$, 2MB L2\$), they may need to optimize this application to perform as well on an Intel Core i7 server (4 cores, 4×32kB L1\$, 4×256kB L2\$, 8 MB L3\$) without a need for creating new fingerprints on the target machine.

The settings for generating reports are just as simple as for sampling. The user can easily choose the cache level for which the report has to be generated. By default, the highest cache level will be selected (see Fig. 5).

If not explicitly specified, the generator will take the CPU's architecture for the current machine as a template for the underlying cache model. However, it is possible to choose the target CPU from a comprehensive list (see Fig. 6) or by providing its parameters regarding to number of caches, cache sizes, and replacement policy explicitly (see Fig. 7).

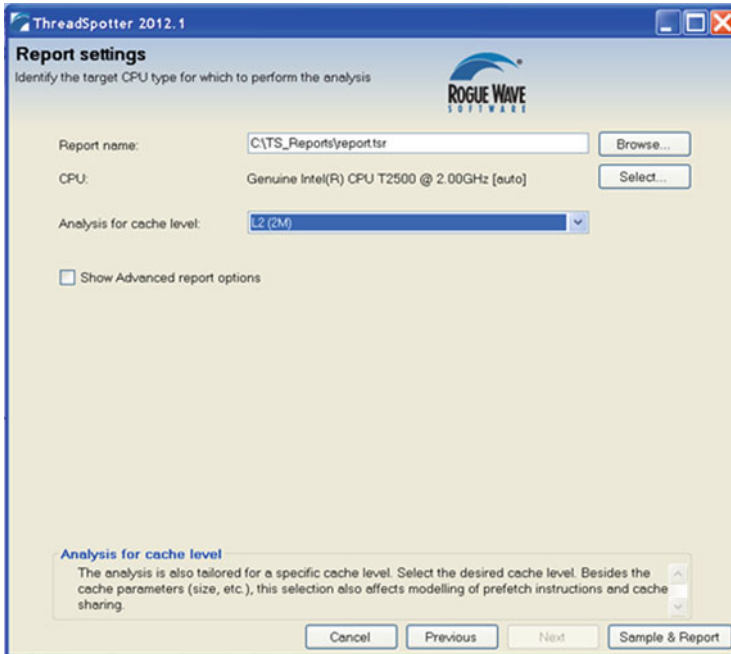


Fig. 5 ThreadSpotter’s settings for report generation

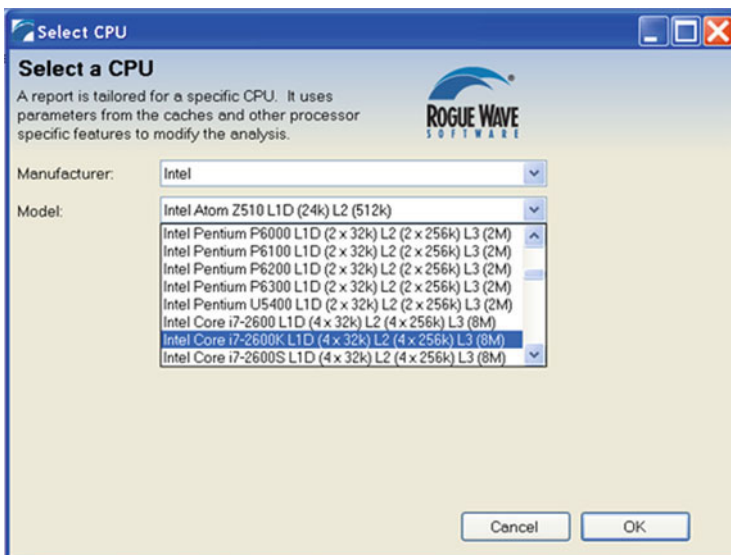


Fig. 6 CPU model selection interface

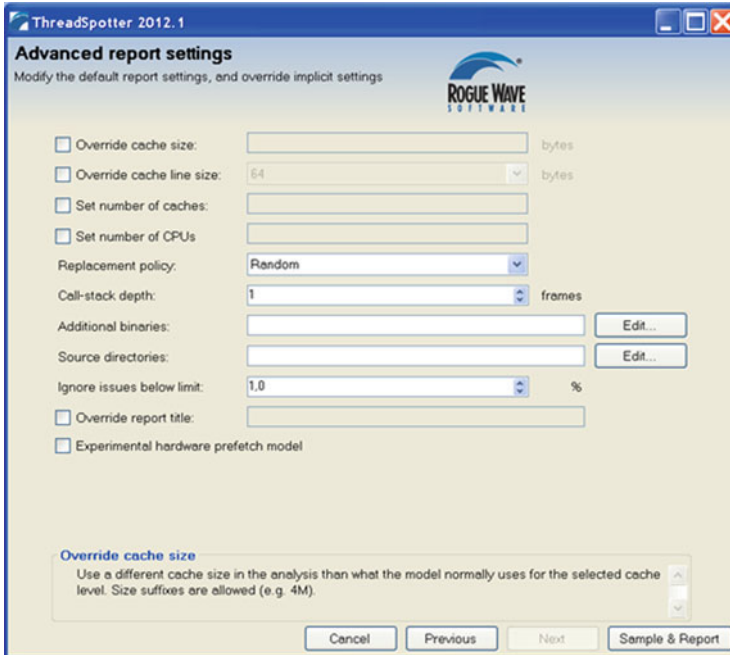


Fig. 7 Possible adaptations of the model

The generated reports can be displayed by ThreadSpotter's viewer application, supported on both Linux and Windows platforms. The viewers act as web servers, allowing the use of web browsers for presenting the content.

3.5 Presenting Optimization Opportunities That ThreadSpotter Discovers

Presenting results in a clearly arranged way is always a challenge. The approach ThreadSpotter utilizes to present the opportunities for optimization, as well as the underlying statistics inside a browser window, allows for establishing hierarchical structures. The user gets the information needed, but is always able to dive into a deeper level. Comprehensive help messages are linked to most of the displayed issues, and metrics which will then pop up with a mouse click on the referring item.

Every report includes a quick overview page (see Fig. 8) which outlines the necessity of optimization. Four indicators give an idea whether the application suffers from bandwidth, latency, data locality, or thread interaction issues. During the development process, this page can be helpful for quick, intermediate checks.

The main report page (see Fig. 9) is divided into three sections:


- Section 1: A tabbed area called a summary frame that contains all metrics, diagrams, and found issues listed in order of severity;

ThreadSpotter™

ThreadSpotter™ is a tool to quickly analyze an application for a range of performance problems, particularly related to multicore optimization.


[Read more...](#) [Manual](#)

Open the Report




Your application


Application: C:\CacheDemo\TestBed.exe 201




Memory Bandwidth
The memory bus transports data between the main memory and the processor. The capacity of the memory bus is limited. Abuse of this resource limits application scalability.
[Manual](#) [Bandwidth](#)



Memory Latency
The regularity of the application's memory accesses affects the efficiency of the hardware prefetcher. Irregular accesses causes cache misses, which forces the processor to wait a lot for data to arrive.
[Manual](#) [Cache miss](#) [Manual](#) [Prefetching](#)



Data Locality
Failure to pay attention to data locality has several negative effects. Caches will be filled with unused data, and the memory bandwidth will waste transporting unused data.
[Manual](#) [Locality](#)



Thread Communication / Interaction
Several threads contending over ownership of data in their respective caches causes the different processor cores to stall.
[Manual](#) [Multithreading](#)

This means that your application shows opportunities to:
Avoid major processor stalls and a congested memory bus due to poor data usage.

[Read more...](#)

Next Steps

The prepared report is divided into sections.

- Select the tab **Summary** to see global statistics for the entire application.
- Select the tabs **Bandwidth Issues**, **Latency Issues** and **MT Issues** to browse through the detected problems.
- Select the tab **Loops** to browse through statistics and detected problems loop by loop.

The Issue and Source windows contain details and annotated source code for the detected problems.

Summary

Source

Resources

[Manual](#)

[Table of Contents](#)

[Optimization Workflow](#)

[Read the Report](#)

[Rogue Wave Software Web Site](#)

[Rogue Wave Web Site](#)

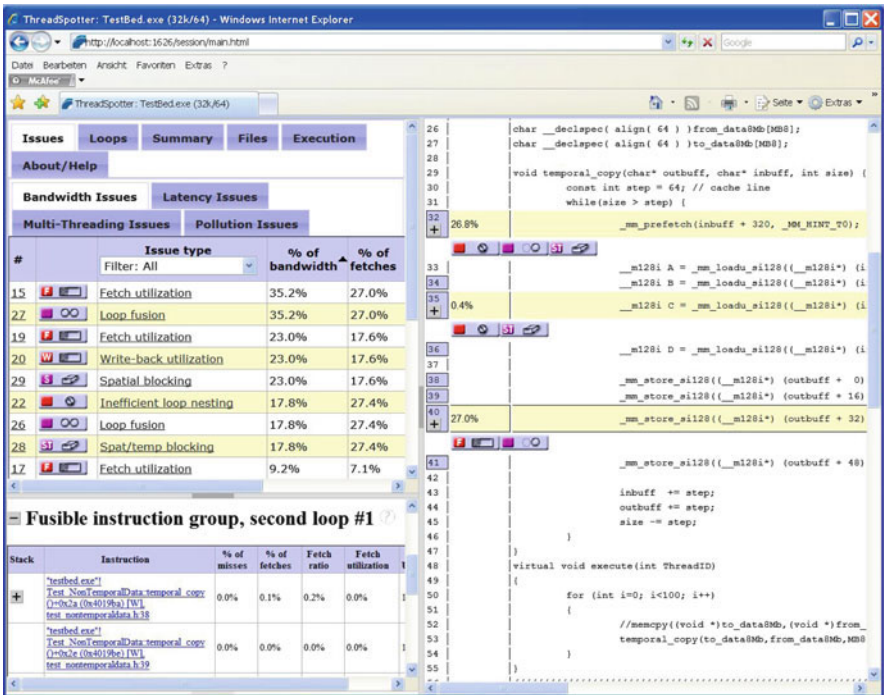
[Overview](#)

[Concepts](#)

[Issue Reference](#)

[Tutorials](#)

Fig. 8 ThreadSpotter quick overview page



The screenshot shows the ThreadSpotter main report page. The top navigation bar includes 'Issues', 'Loops', 'Summary', 'Files', and 'Execution'. The 'Issues' tab is active, showing a table of detected issues. Below the table, there is a section for 'Fusible instruction group, second loop #1' with a detailed instruction table. The right side of the page displays the source code for the selected issue, with line numbers and annotations.

#	Issue type	% of bandwidth	% of fetches
15	Fetch utilization	35.2%	27.0%
27	Loop fusion	35.2%	27.0%
19	Fetch utilization	23.0%	17.6%
20	Write-back utilization	23.0%	17.6%
29	Spatial blocking	23.0%	17.6%
22	Inefficient loop nesting	17.8%	27.4%
26	Loop fusion	17.8%	27.4%
28	Spat/temp blocking	17.8%	27.4%
17	Fetch utilization	9.2%	7.1%

Stack	Instruction	% of misses	% of fetches	Fetch ratio	Fetch utilization
"testbed.exe"	Test_NonTemporalData_temporal_copy	0.0%	0.1%	0.2%	0.0%
"testbed.exe"	Test_NonTemporalData_h38	0.0%	0.0%	0.0%	0.0%
"testbed.exe"	Test_NonTemporalData_temporal_copy	0.0%	0.0%	0.0%	0.0%
"testbed.exe"	Test_NonTemporalData_h39	0.0%	0.0%	0.0%	0.0%

```

26 char __declspec( align( 64 ) )from_data0MB(MB8);
27 char __declspec( align( 64 ) )to_data0MB(MB8);
28
29 void temporal_copy(char* outbuff, char* inbuff, int size) {
30     const int step = 64; // cache line
31     while(size > step) {
32         + 26.8%      __mm_prefetch(inbuff + 320, _MM_HINT_T0);
33
34         __m28i A = __mm_loadu_si128((__m28i*)(i
35         __m28i B = __mm_loadu_si128((__m28i*)(i
36         + 0.4%      __m28i C = __mm_loadu_si128((__m28i*)(i
37
38         __m28i D = __mm_loadu_si128((__m28i*)(i
39
40         __mm_store_si128((__m28i*)( outbuff + 0)
41         __mm_store_si128((__m28i*)( outbuff + 16)
42         + 27.0%      __mm_store_si128((__m28i*)( outbuff + 32)
43
44         __mm_store_si128((__m28i*)( outbuff + 48)
45
46         inbuff += step;
47         outbuff += step;
48         size -= step;
49     }
50 }
51 virtual void execute(int threadID)
52 {
53     for (int i=0; i<100; i++)
54     {
55         //memcpy((void *)to_data0MB, (void *)from
56         temporal_copy(to_data0MB, from_data0MB, MB8)
57     }
58 }

```

Fig. 9 ThreadSpotter main report page

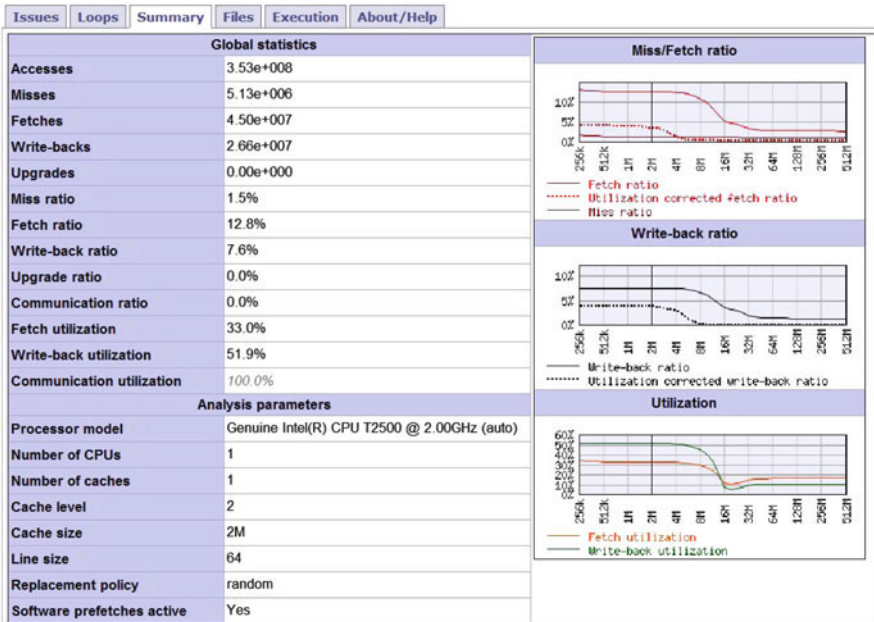


Fig. 10 ThreadSpotter summary tab

- Section 2: An issue frame, located below the summary frame, showing detailed information referring to selected issues or loops;
- Section 3: An area for selected issues (“slowspots”) that are referenced to responsible lines of the source code when the binary is built using the -g compiler option.

The “Summary” tab of the summary frame (Fig. 10) shows the global statistics based on the sampled data of the application. The most important numbers for qualifying a sampled application at a glance are the miss ratio, fetch ratio, fetch utilization, and the communication utilization (for multithreaded applications).

Additional diagrams provide the statistics for miss/fetch ratio, write-back ratio, and utilization related to the cache size. Achievable improvements that can be gained by amending the cache utilization are represented in the diagrams by the additional curves called utilization corrected fetch ratio and utilization corrected write-back ratio.

In the same way the global statistics are represented, there are also similar frames for loops and instruction groups related to detected issues. Underneath the “Issues” tab (see Fig. 11), all detected issues are separated by “Bandwidth Issues”, “Latency Issues”, “Multi-Threading Issues”, and “Pollution Issues”, ordered by severity.

The listed issues icon will indicate the type of issue. Red icons are generally used for labeling issues that are responsible for decreasing performance. Purple icons are used to identify opportunities where fixing an issue could improve

Issues Loops Summary Files Execution About/Help						
Bandwidth Issues		Latency Issues		Multi-Threading Issues		Pollution Issues
#	Issue type	% of bandwidth	% of fetches	% of write-backs	Fetch utilization	Write-back utilization
5	Fetch utilization	35.9%	28.6%	48.3%	0.0%	100.0%
6	Loop fusion	35.9%	28.6%	48.3%	0.0%	100.0%
7	Non-temporal store possible	35.9%	28.6%	48.3%	0.0%	100.0%
12	Fetch utilization	26.4%	21.0%	35.5%	0.2%	0.2%
13	Write back utilization	26.4%	21.0%	35.5%	0.2%	0.2%
14	Spatial blocking	26.4%	21.0%	35.5%	0.2%	0.2%
15	Loop fusion	26.4%	21.0%	35.5%	0.2%	0.2%
1	Fetch hot-spot	18.2%	28.9%	0.0%	100.0%	100.0%
2	Spat/temp blocking	18.2%	28.9%	0.0%	100.0%	100.0%
3	Loop fusion	18.2%	28.9%	0.0%	100.0%	100.0%
9	Fetch utilization	10.0%	7.9%	13.4%	1.0%	1.0%
10	Write back utilization	10.0%	7.9%	13.4%	1.0%	1.0%
11	Fetch utilization	4.9%	7.8%	0.0%	16.0%	100.0%
18	Fetch utilization	3.2%	3.5%	2.7%	26.0%	62.7%
20	Fetch hot-spot	1.2%	2.0%	0.0%	49.2%	100.0%
17	Fetch hot-spot	0.2%	0.2%	0.0%	100.0%	100.0%

Fig. 11 ThreadSpotter issues tab

Issue #19: Fetch utilization

This instruction group also shows symptoms of  Fetch hot-spot,  Write-back hot-spot.

-  **Statistics for instructions of this issue** 
-  **Instructions involved in this issue** 
-  **Loop statistics** 
-  **Loop instructions** 

Fig. 12 ThreadSpotter issue frame

performance. Blue icons are indicators for high cache activity. Every listed issue can be expanded in order to obtain more detailed information, such as individual statistics, instructions involved in this issue (stack frame), and assistance from a comprehensive help system (see Fig. 12).

If the application was built using the `-g` compiler flag and the source code files were available during report generation, ThreadSpotter’s report includes the source code related to every listed issue and references each issue to the responsible line of code (see Fig. 13).

Detailed statistics on the significant machine instructions and icons representing the issues will appear when the expand button is pressed. Based on this information, programmers are able to look for opportunities for code improvements.

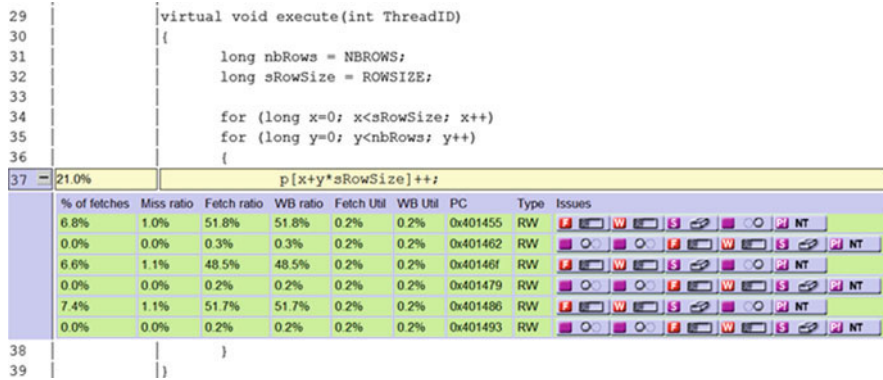


Fig. 13 ThreadSpotter source code frame

4 Types of Performance Optimization Opportunities Discovered by ThreadSpotter

ThreadSpotter was designed for assisting developers in optimizing applications regarding the performance. This section outlines the opportunities for optimization Thread-Spotter is able to discover.

4.1 What Kind of Cache-Related Opportunities for Performance Optimization Can Be Discovered by ThreadSpotters Statistical Approach?

ThreadSpotter is able to reliably detect the cache related performance issues by examining the binary and mark the responsible line(s) in the source code, grouped in the following categories.

Poor Fetch Utilization

The content of a fetched cache line will only partially be used. This issue is typical in object oriented programming. A good example is a program traversing an array of structures while accessing only one of multiple member variables of every structure. This fills up the cache with unnecessary data that will increase the likelihood a fetch will end up having to access the slower main memory. Similar effects could occur in the case where member variables of structures are unfavorably aligned.

Poor Write-Back Utilization

Writing data will first take effect to the referring cache line. The main memory will not be updated until the cache line is evicted from cache. At that time the content of the entire cache line has to be written back. When the application is updating only a subset of the whole cache line, even unchanged data will be written back to main memory, which increases the bandwidth usage of the system.

Poor Communication Utilization

In multithreaded environments, threads very often need to interchange cache lines between their private caches. This procedure is always time consuming and is accompanied by some inter-thread communication. ThreadSpotter sets the amount of communication between two threads in relation to the cache line utilization of the receiving thread. A high percentage of communication utilization is an indicator of the efficiency of the multithreaded application.

False Sharing

False sharing is also an issue that only occurs in multithreaded environments. Sometimes two threads have to access different data independently that are unfortunately located in the same cache line. That could mean that although the two threads are working on different data, they have to update the same cache line in their private caches, which produces unnecessary communication and data exchange overhead. A solution could be to distribute the referring data into two different cache lines.

Inefficient Loop Nesting

If multidimensional arrays are traversed in the wrong direction inside a loop (e.g. column wise instead of row wise in C/C++ applications), only one data element will be used from each cache line. Depending on the size of the cache, this could mean that cache lines have already been evicted from cache until they have to be used the next time. This ends up in bad fetch/write-back utilization issues.

Random Access Pattern

If the application does not access memory addresses in a regular manner, the hardware prefetcher is not able to identify constant-stride access patterns in order to work efficiently. This will usually result in an increase of cache misses. The application will require more bandwidth because the processor has to access main memory more often. Two strategies could help in this case. It will either be necessary

to change the algorithms of the application so they have regular data access patterns or add software prefetch statements to the code in order to assist the hardware prefetcher.

Unnecessary Prefetch

This refers to a software prefetch statement that just takes execution time without providing any benefit because the data was already cached (e.g. by a smart hardware prefetcher).

Prefetch Too Distant

If a software prefetch statement points to a too distant data address referring to the actual stride pattern, it is most likely that a cache line which would contain the predictive data has already been evicted before it can be requested.

Prefetch Too Close

This occurs when the software prefetch statement points to data which is located too close to data that was recently fetched. That means that the access time for receiving the data from main memory is too long in order to have that data copied to cache before it will be used.

4.2 Reuse

Two following reuse opportunities can be detected by ThreadSpotter.

Spatial/Temporal Blocking

Blocking means optimizing a memory-consuming algorithm by breaking it down so it will process the whole amount of data contained in the cache. An example for optimizing matrix multiplication by blocking is shown in Fig. 14.

Loop Fusion

The reuse opportunity loop fusion will be pointed out if different loops are iterating over the same data and an effective reuse of that data will be prevented because the cache lines are already evicted from cache before the second loop will request them.

Fig. 14 Matrix multiplication optimized by blocking

```

#define SIZE_MATRIX 256
#define BLOCK_I 16
#define BLOCK_K 16
#define BLOCK_J 16

void multiply(double a[SIZE_MATRIX][SIZE_MATRIX],
             double b[SIZE_MATRIX][SIZE_MATRIX],
             double c[SIZE_MATRIX][SIZE_MATRIX])
{
    for (int i = 0; i < SIZE_MATRIX; i++)
        for (int j = 0; j < SIZE_MATRIX; j++)
            c[i][j] = 0;

    for (int ii = 0; ii < SIZE_MATRIX; ii += BLOCK_I)
        for (int kk = 0; kk < SIZE_MATRIX; kk += BLOCK_K)
            for (int jj = 0; jj < SIZE_MATRIX; jj += BLOCK_J)
                for (int i = ii; i < ii + BLOCK_I && i < SIZE_MATRIX; i++)
                    for (int k = kk; k < kk + BLOCK_K && k < SIZE_MATRIX; k++)
                        for (int j = jj; j < jj + BLOCK_J && j < SIZE_MATRIX; j++)
                            c[i][j] += a[i][k] * b[k][j];
}

```

In these cases, moving these loops closer together or even fusing them may improve reusability, and as a result, performance. ThreadSpotter explicitly references the fusible instruction groups belonging to each other.

4.3 Non-temporal Data

If cached data will never be reused, it is obvious that the cache lines just occupy valuable space in the cache without providing any performance benefit. Other threads or processes might make better use of that cache space. Depending on the compiler and hardware, software prefetch statements could avoid caching that data. Typical examples for processing non-temporal data are I/O operations.

4.4 Cache Hot Spots

In addition to the caching issues mentioned above, which can be qualified as related to inefficient coding, ThreadSpotter points out parts in the code that are responsible for exceptional cache activities:

- Fetch Hot-Spots
- Write-Back Hot-Spots
- Communication Hot-Spots

These hot-spots are not necessarily caused by improper code patterns. Sometimes they are simply an unavoidable side effect of the algorithms used. However, it is always useful to have a closer look at the referring parts in the code. Referring to the majority of identified slow spots in the application, ThreadSpotter assists the programmer by discovering opportunities for optimizing performance.

5 Conclusion

This article has reflected on caching mechanisms and cache related performance issues that can be detected and analyzed by a statistical approach, upon which Rogue Wave Software's ThreadSpotter is based. The introduction of new profiling concepts should help make programmers aware of potential cache related optimizations. However, the cache-related performance analysis and the optimization of applications are topics that are much more substantial than this article may suggest. For more information please refer to the manual [2]. Further reading on these subjects is therefore recommended.

References

1. Berg, Hakan and Hagersten. A Statistical Multiprocessor Cache Model by Erik Berg, Håkan Zeffner, and Erik Hagersten. In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006), Austin, Texas, USA, March 2006.
2. Rogue Wave Software. ThreadSpotter Manual Version 2012.1 Boulder, CO, USA. 2012

Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite

Holger Brunst and Matthias Weber

Abstract The development and maintenance of scalable, state-of-the-art applications in High Performance Computing (HPC) is complex and error-prone. Today, performance debuggers and monitors are mandatory in the software development chain and well established. Like the applications, the tools themselves have to keep track of the developments in system and software engineering. Prominent developments in this regard are for example hybrid, accelerated, and energy aware computing. The ever increasing system complexity requires tools that can be adjusted and focused to user specific interests and questions. This article explains how the performance tool Vampir can be used to detect and highlight user-defined hot spots in HPC applications. This includes the customization and derivation of performance metrics, highly configurable performance data filters and a powerful comparison mode for multiple program runs. The latter allows to keep track of the performance improvements of an application during its evolution.

1 Introduction

Today, High Performance Computing (HPC) is implemented by means of many thousand to millions of processing elements working together at the same task [4]. Writing software for systems of this scale is cumbersome and involves hybrid programming models, accelerated computing [2], and even energy considerations [3]. An iterative performance design, verification, and optimization process [5, 7] is needed to exploit the vast resources efficiently. This article describes recent

H. Brunst (✉) · M. Weber
Center for Information Services and HPC (ZIH), Technische Universität Dresden,
01062 Dresden, Germany
e-mail: holger.brunst@tu-dresden.de

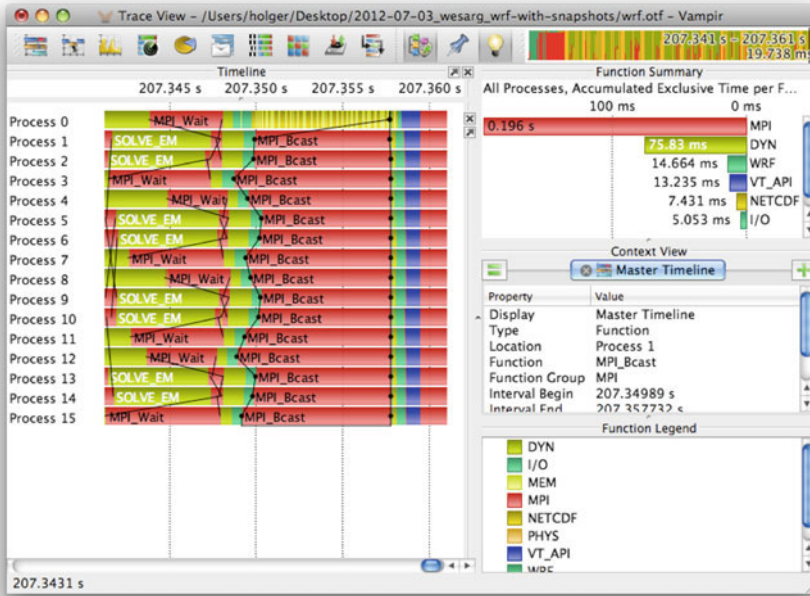


Fig. 1 Default performance overview at initial Vampir startup

developments implemented in the Vampir [1] performance tool suite to customize the verification and optimization process to specific user interests. First of all, a new hybrid flow diagram is introduced that visualizes performance metrics in the context of a program’s procedural invocation graph. Thereafter, the intuitive graphical definition of derived custom metrics is discussed and demonstrated with examples. Subsequently, the customization and filtering of complex invocation graphs by means of linked rule sets is presented. Finally, the detailed structural comparison of multiple program runs using manual graphical event data alignment is described.

When opening a program’s event log (trace file) for the first time, Vampir’s main *Trace View* window displays a default set of performance charts (see Fig. 1), which are called *Master Timeline* (left), *Function Summary* (top right), *Context View* (center right), and *Function Legend* (bottom right). The Master Timeline reveals the exact procedural program behavior, i.e. *what* is executed *when* and by *whom*, for arbitrary program phases, while the *Function Summary* provides corresponding accumulated performance metrics, e.g. the inclusive time spent in procedures.

Multiple timeline charts are aligned horizontally. This alignment ensures that the temporal relationship of events is preserved across the charts. For chart objects that have been clicked with the left mouse button, supplemental detail information is provided in the Context View. Color encoding is customizable and typically refers

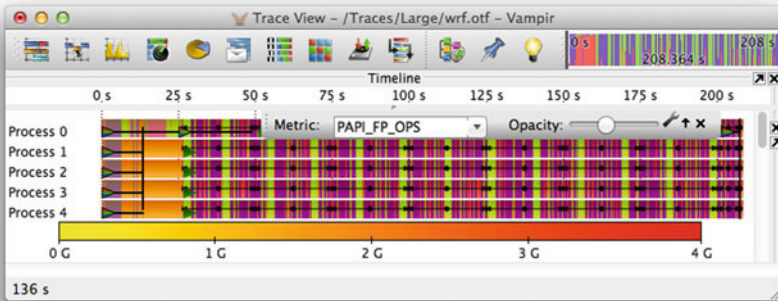


Fig. 2 Master Timeline with semitransparent heat map overlay and opacity set to 40 %

to the type of program activity, e.g. communication = red, computation = green, as shown in the Function Legend. On top of all charts a tool bar is available which allows to add further charts to the Trace View (left) and to navigate through the presented performance log, i.e. refine the time interval of interest (right).

2 Heat Map Overlay for the Master Timeline

Vampir 7.5 features the highlighting of arbitrary program phases that match user-defined performance conditions in combination with the recorded application flow. For this purpose, a new overlay mode has been introduced to Vampir's Master Timeline chart. This overlay mode is fully compatible with the Performance Radar chart known from previous releases. It is capable of highlighting user-defined hot spots on top of the procedural information provided in the Master Timeline chart. Figure 2 gives an overview of the new overlay mode. It is activated via the chart's context menu under *Options* → *Performance Data*. When the overlay mode is active, a tool bar appears at the top of the Master Timeline chart. On the left hand side, it allows to select the performance metric of interest. On the right hand side, a slider that controls the opacity of the overlay is provided. The overlay is capable of displaying all metrics available in the Performance Radar and the Counter Data Timeline chart.

The selected metric PAPI_FP.OPS (floating point operations per second) is shown in a color-coded fashion. The overlay mode combines the visualization capabilities of the Master Timeline, i.e. showing the application's chain of events, and the Performance Radar, which shows the development of arbitrary performance metrics over time. To fully benefit from this combination, the opacity slider of the overlay control window needs to be used as depicted in Fig. 3. The slider allows to

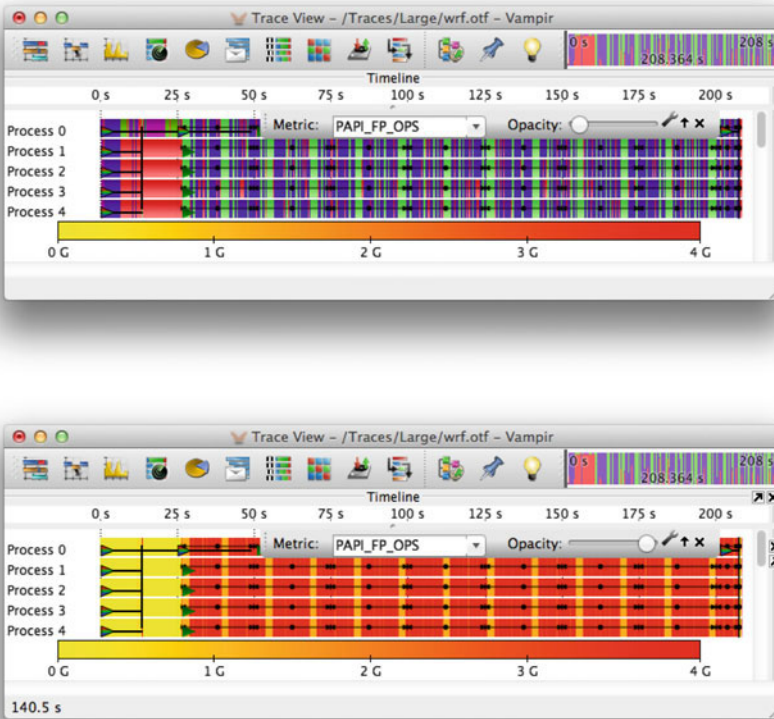


Fig. 3 Master Timeline with fully transparent (top) and opaque heat map overlay (bottom)

quickly manipulate the opacity of the overlay making the underlying application context visible. This is particularly useful for pinpointing performance relevant areas and subsequently analyzing these areas in detail.

The color scale of the performance data overlay is freely customizable. Clicking the wrench icon in the overlay control window opens the color scale options dialog as depicted in Fig. 4. The color scale provides three standard modes: *Default*, *Highlight*, and *Find*. Additionally, the mode *Custom* allows to configure the color scale to personal preferences.

To identify program phases with high or low floating point performance the display range of the metric can be adjusted as depicted in Fig. 5. The adjustment has been done by dragging the edges of the colored area of the scale to the desired minimum and maximum values. As a result, only performance values within the given range appear color-coded in the chart. Values outside the given range are grayed out.

Figure 5 highlights program phases with bad floating point performance. The color scale has been set to a range between 0.1 and 1.6 GFLOPS. The minimum

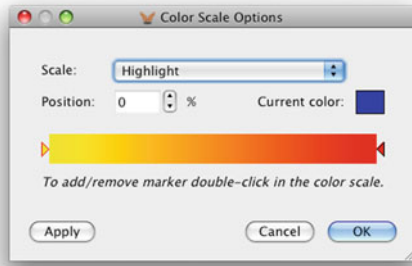


Fig. 4 Dialog for color scale customization

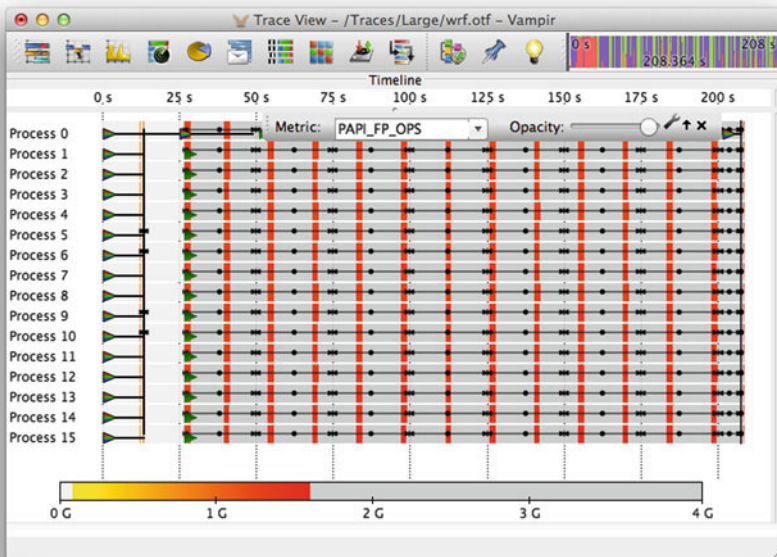


Fig. 5 Highlighting of program phases with low performance

value is set to 0.1 GFLOPS to completely hide non-computing program activity like *MPI_Init* and *MPI_Send*. As a result, only computational program phases with low floating point performance are highlighted now. In this example these areas represent procedures at the beginning of each iteration. Likewise, program phases with high floating point performance can be highlighted accordingly by moving the colored slider to the right boundary of the scale.

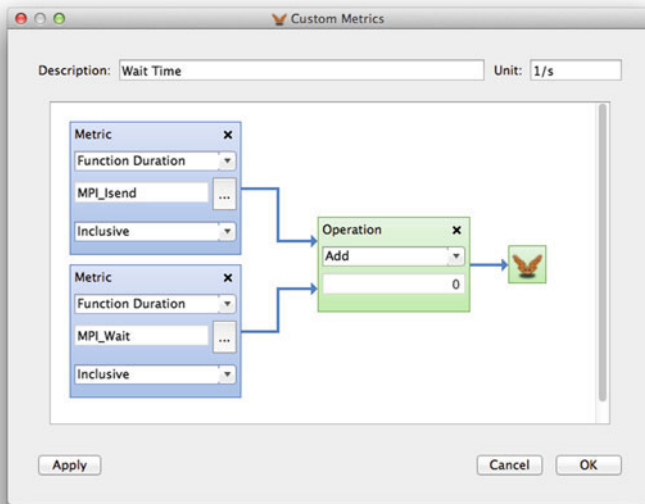


Fig. 6 Custom metric editor showing a simple custom metric called *Wait Time*. The metric adds up the duration of the functions *MPI_Isend* and *MPI_Wait*

3 Customization of Performance Metrics

The *Custom Metrics Editor* allows to derive custom metrics from recorded metrics and events. This is particularly useful for refining and narrowing the effect under investigation. The editor is accessible via the context menu entry *Customize Metrics ...* in the Performance Radar or the Counter Data Timeline charts. A basic example of a custom metric called *Wait Time* is shown in Fig. 6. This metric combines the time spent in the functions *MPI_Irecv* and *MPI_Wait*. Custom metrics are built from input metrics that are re-combined by using arithmetic and logical operations. In the editor the context menu accessible via the right mouse button allows to add new input metrics and operations. New custom metrics become available in the metric selection lists of the respective charts. Custom metrics can be exported and imported in order to use them in multiple trace files.

The following examples illustrate the usage of custom metrics in combination with the heat map overlay explained in Sect. 2. The depicted trace file data originates from a parallel program run of the Open Source *Weather Research & Forecasting Model* (WRF) [6] on 16 CPU cores.

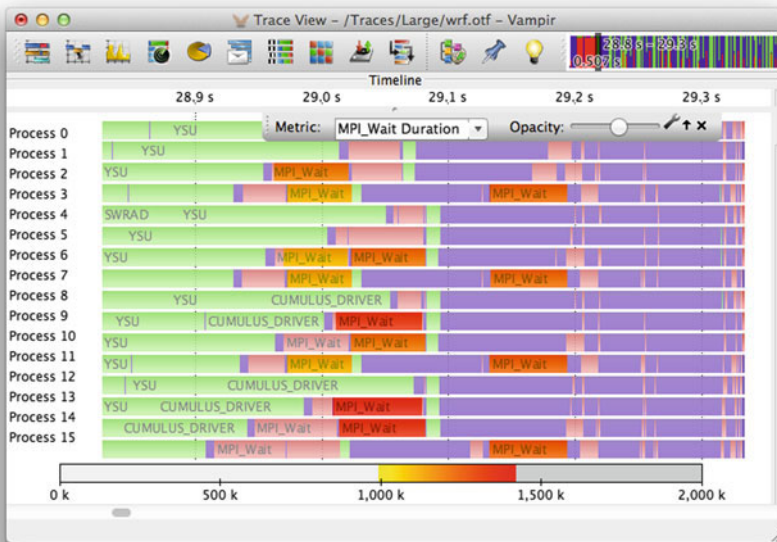


Fig. 7 Identification of delayed MPI.Wait invocations (highlighted in *bright orange* and *red*) in combination with the overall application flow (dimmed areas)

3.1 Delayed MPI Communication

Delayed invocations of the function MPI.Wait are investigated in the following example. The first step is to construct a custom metric showing the MPI.Wait duration time as explained above. Subsequently, the heat map overlay is used to show the custom metric in the Master Timeline as depicted in Fig. 7. The color scale is configured to show delayed MPI.Wait invocations (longer than one million clock cycles). Program phases that fulfill the given condition are highlighted in deep red. Zooming into detail eventually reveals individual MPI.Wait invocations. The invocation context of these calls becomes visible in the Master Timeline by setting the opacity slider to 50%.

3.2 Conditional Floating Point Performance

Vampir also allows to search for invocations of individual functions below or above a custom performance threshold. In this example invocations of a given function (SOLVE_EM) with a floating point performance above 150 MFLOPS are searched. The first step is to construct a custom metric providing the floating point performance for the given function only. The custom metric definition is depicted

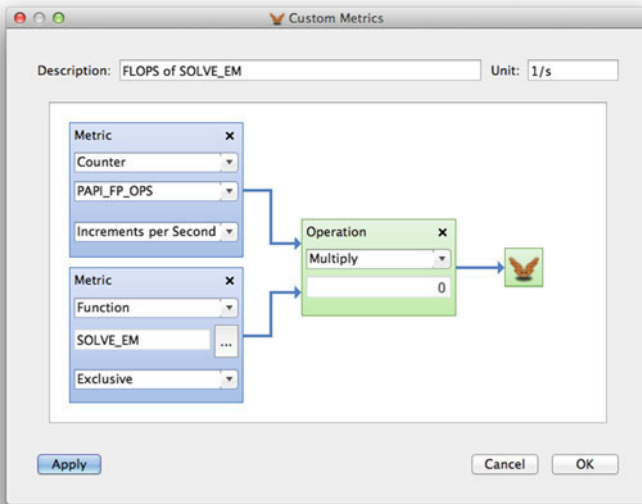


Fig. 8 Custom metric definition which computes the FLOPS metric for function SOLVE_EM only

in Fig. 8. It is a combination of the recorded hardware metric flop/s and a program state mask which is one whenever the program resides in the function of interest and zero otherwise. The multiplication of both metrics results in flop/s readings greater zero whenever in the given function and zero otherwise.

Figure 9 shows the resulting metric in the heat map overlay of the Master Timeline. The color scale is configured to highlight the given function if its performance is above 150 MFLOPS. When zooming into an area of interest the opacity slider can be used to reveal individual function invocations in the timeline.

4 Refinement of Invocation Graph

The conceptual and procedural structure of a parallel program is defined by the respective program developers. Yet, standard building blocks like STL, Boost, MPI and OpenMP add structural elements to the procedural invocation structure of an HPC application. The resulting overall structure is hard to study and often beyond the interests of application developers who want to focus on particular aspects of a program only. Vampir has the ability to filter the recorded procedural invocation history per individual event. The filtering is controlled from the *Function Filter* dialog depicted in Fig. 10. This dialog can be accessed from the main menu under *Filter* → *Functions* ... The filter affects all performance charts that deal with the procedural invocation structure.

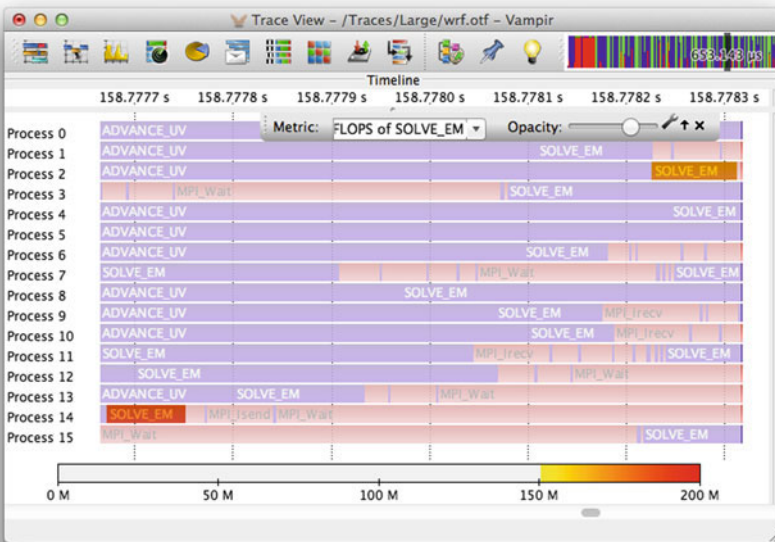


Fig. 9 Conditional FLOPS metric depicted as transparent heat map overlay in Master Timeline. Please note that only some (those who perform at least at 150 MFLOPS) of the SOLVE_EM invocations are highlighted

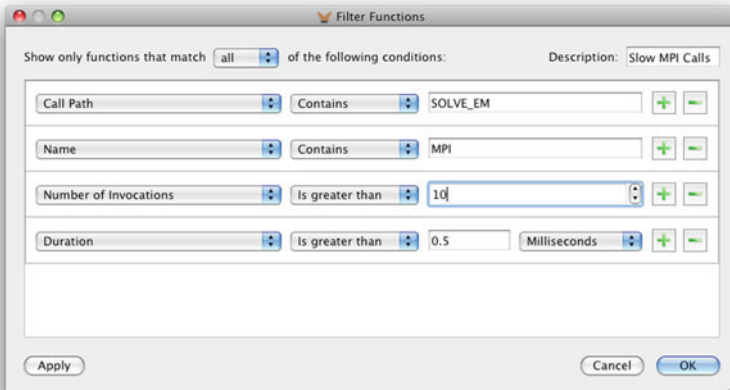


Fig. 10 Procedural Event Filter Dialog with example rules

A custom filter is defined by combining a set of rules. The different types of rules are explained below. The header of the dialog entitled *Show only functions that match ... of the following conditions:* defines how multiple rules are evaluated. One possibility is to build up the filter in a way that combines the filter rules with an *and* relation. To enable this mode, *all* must be selected at the top of the dialog. This means that all rules must apply. An alternative way is to combine the rules with an *or* relation. To enable this mode, *any* must be selected. In this case each rule is applied individually to the recorded performance data. An arbitrary number of conditions can be defined as rules. This example shows four common rules, which limit the invocation graph to MPI procedures called from the procedure *SOLVE_EM* occurring more frequently than 10 times and taking longer than 0.5 ms.

4.1 Rules

4.1.1 Event Matching by Name

A straight-forward way of filtering the procedural invocation graph is by procedure name. This condition checks procedure invocations against the given reference name with the following alternative ways of matching:

- **Contains:** The given string must occur in the procedure name.
- **Does not contain:** The given string must not occur in the procedure name.
- **Is equal to:** The given string must be the same as the procedure name.
- **Is not equal to:** The given string must not be the same as the procedure name.
- **Begins with:** The procedure name must start with the given string.
- **Ends with:** The procedure name must end with the given string.

The matching is not case sensitive. Alternatively, a complete list of procedure names allows to directly select a set of procedures to be displayed.

4.1.2 Event Matching by Duration

Procedure invocations can also be filtered by their duration. Duration of a procedure refers to the time spent in this procedure from entry to exit. There are two options available:

- **Is greater than:** The invocations must take longer than the given duration.
- **Is less than:** The invocations must be shorter than the given duration.

4.1.3 Event Matching by Invocation Number

The number of invocations of a procedure can also be used for filtering. This criteria refers to how often a procedure is invoked during an application run. There are two possible parameter interpretations in this mode:

- **Is greater than:** Procedure must be called at least N times.
- **Is less than:** Procedure must not be called more frequently than N .

Number of Invocations refers to the total number of invocations in the whole application run. *Number of Invocations per Process* refers to the individual number of invocations per process. Hence, if the number of invocations of a procedure varies across processes, a procedure might be shown for some processes and filtered for others.

4.2 Examples

The following examples explain the usage of the procedure invocation filter. They enable the reader to understand the basic principles of procedure invocation filtering at a glance.

4.2.1 Showing MPI Procedure Invocations Only

In this example only procedures that contain the string *mpi* (not case sensitive) in their name are shown. Since all MPI procedures are prefixed with *MPI* this filter setting shows all MPI invocations and hides all others as depicted in Fig. 11.

4.2.2 Showing Procedure Invocations Longer than 250 ms

This example demonstrates the filtering of procedures by duration. In this case only long procedure invocations with a minimum duration time of 250 ms are shown as depicted in Fig. 12. All other procedure invocations remain hidden. This filter is effective whenever fine grained program behavior – for example calls to the STL – is of no interest.

4.2.3 Combining Rules

This example combines the two previously introduced rules. The relation *any* is used. The resulting performance charts in Fig. 13 depict all procedure invocations that are at least 250 ms long and additionally all *MPI* invocations (also the short ones).

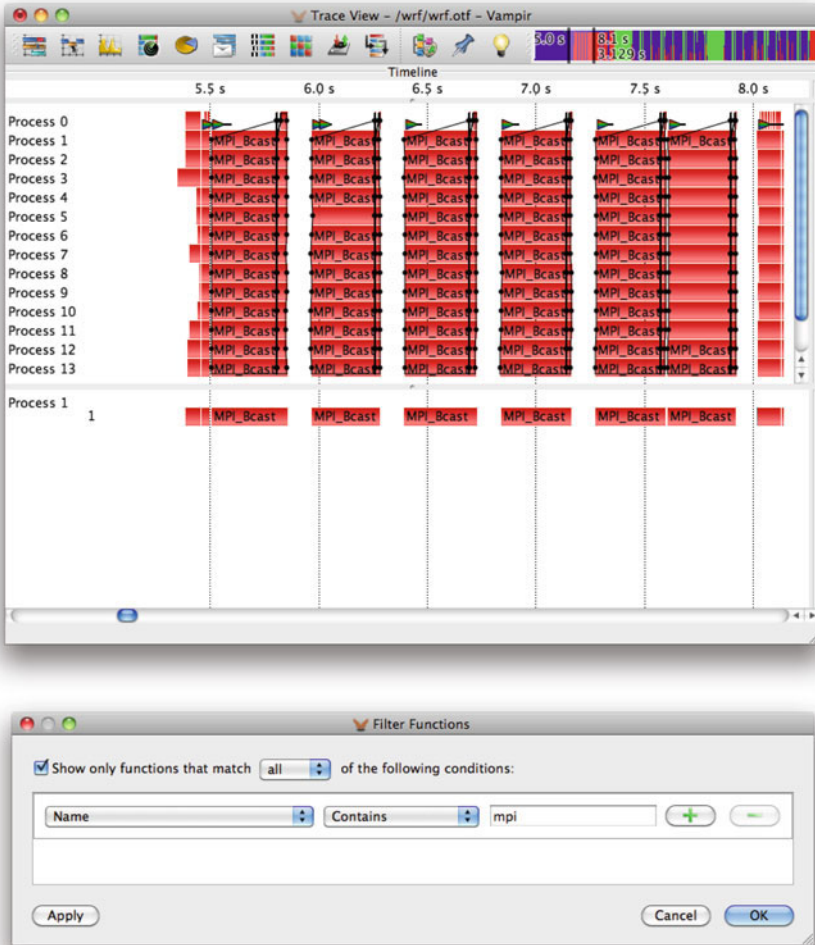


Fig. 11 Master Timeline, Process Timeline (top), and filter (bottom) showing MPI activity only

4.2.4 Building Value Ranges

The combination of rules enables the filtering of procedures which fulfill a specific criteria range. The following filter setup shows all procedure invocations whose number of invocations is in the the range of 100–15,000 as depicted in Fig. 14.

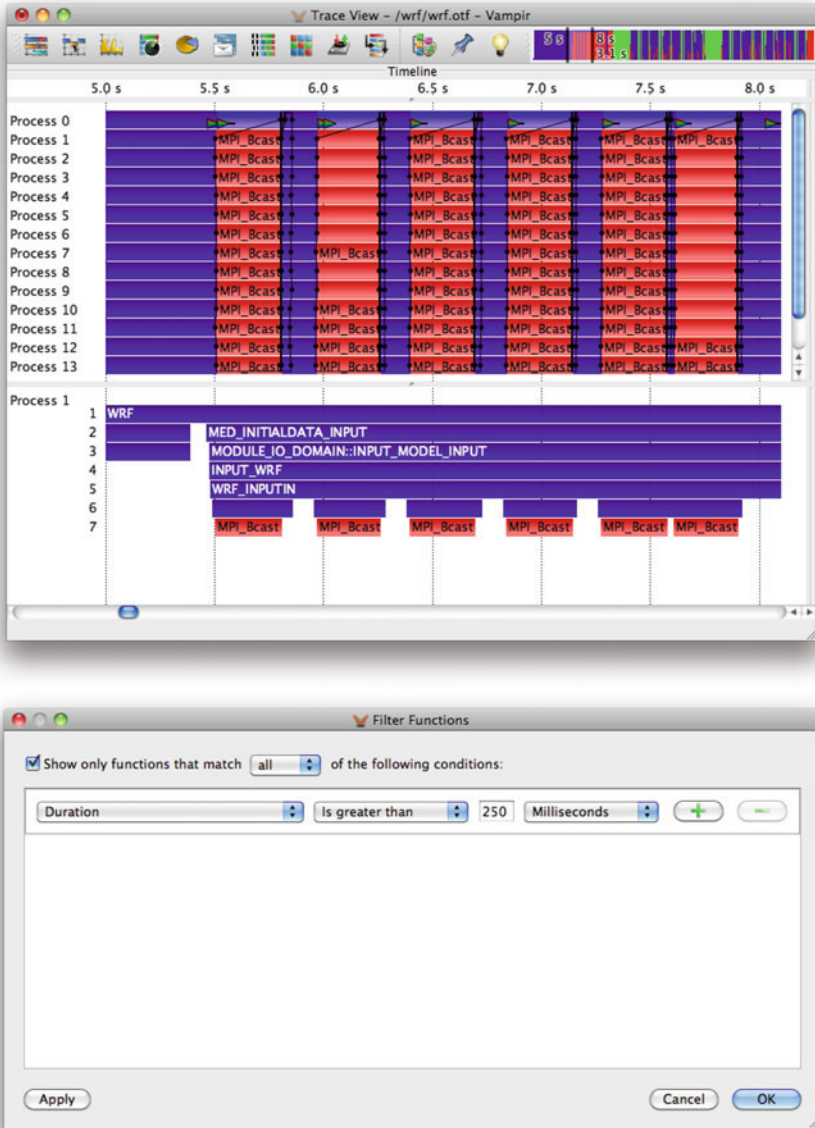


Fig. 12 Showing procedure invocations that are longer than 250 ms

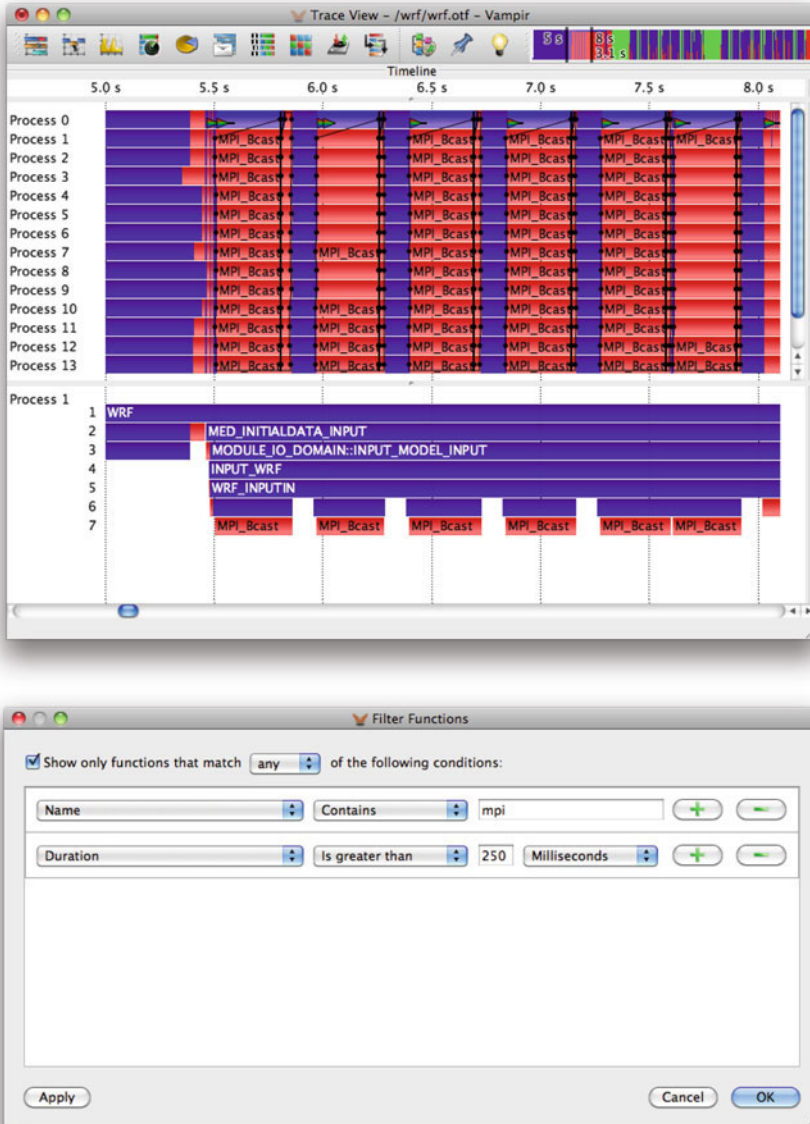


Fig. 13 Combination of rules that reveals all MPI invocations together with non-MPI activity taking longer than 250 ms

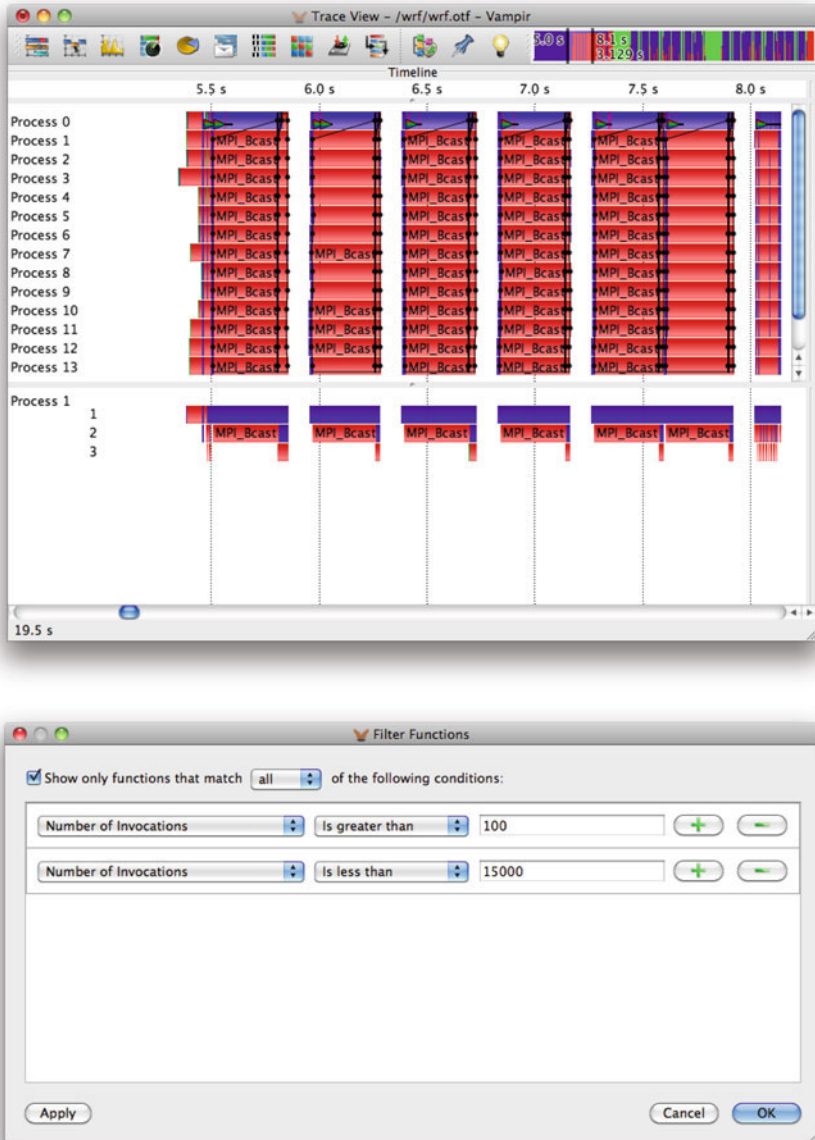


Fig. 14 Procedures that have been called more than 100 times and fewer than 15,000 times

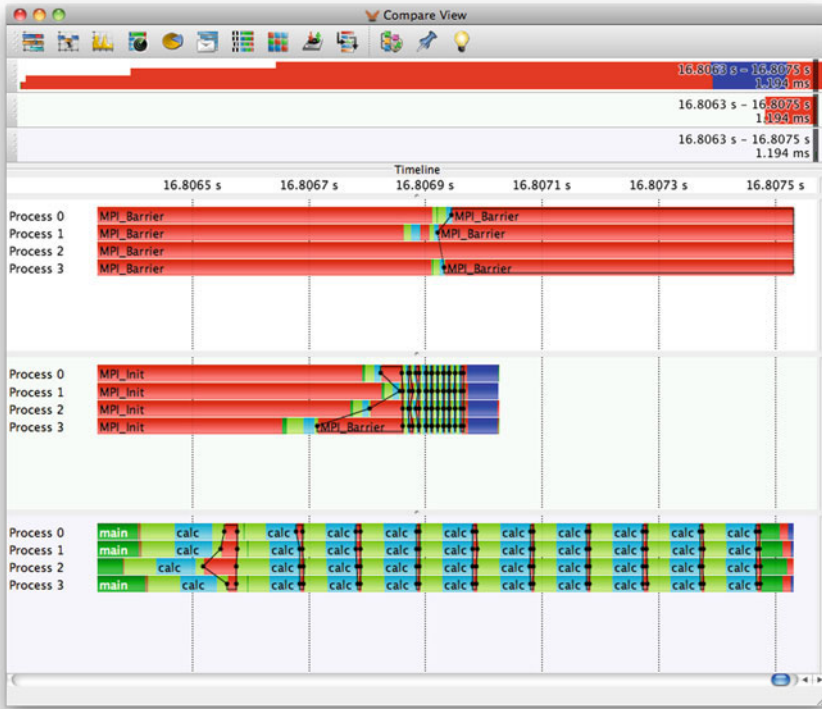


Fig. 15 Compare View with combined zoom and alignment toolbar (top) and timeline charts (center) for identical program runs on three different computer systems

5 Comparison of Multiple Program Runs

The comparison of multiple program runs has been integrated seamlessly into Vampir, which allows software developers to track the performance evolution of a software project. For comparison of performance characteristics all existing displays are supported. In order to effectively compare multiple trace files, their time needs to be coupled and synchronized. Furthermore, selected areas in the trace files need to be shifted freely on the time scale. This allows for arbitrary alignments of the trace files, and thus, enables comparison of user selected sections in the trace data. All comparison features are provided in a new window entitled *Compare View*. Figure 15 depicts the aligned computation part of three program execution logs as Master Timelines. They represent the measurement of one test application on three different computer systems. The test application consists of an initialization phase, ten iterations of computation, and a finalization phase. All three Master Timeline charts share the same time scale and zoom.

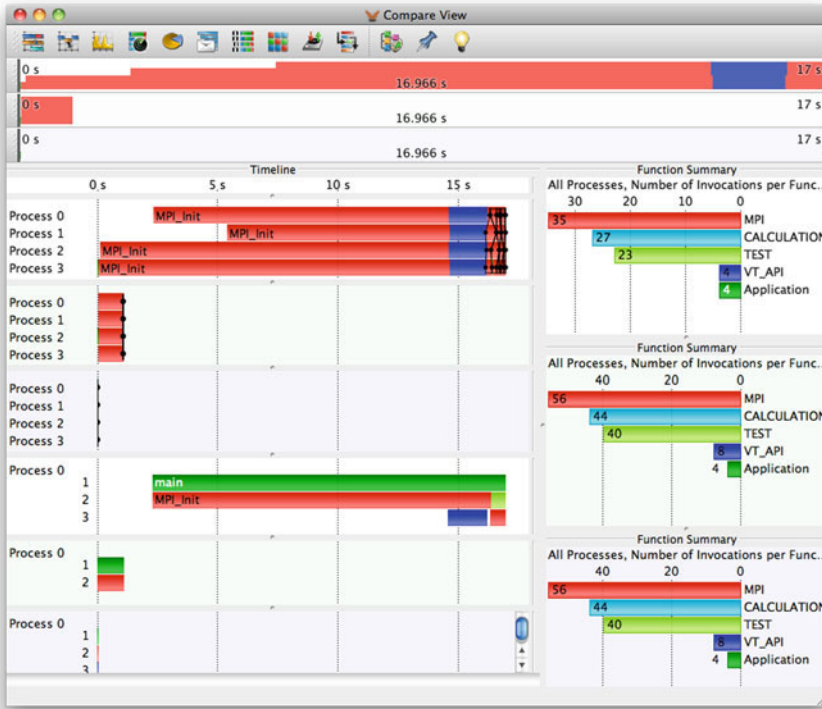


Fig. 16 Compare View with Master Timeline (*upper left*), Function Summary (*right*), and Process Timeline (*bottom left*) for identical program runs on three different computer systems

5.1 Multiple Program Executions at a Glance

The Compare View supports all available performance charts of *Vampir*. In contrast to the single file mode, multiple chart instances are opened whenever the user clicks on a chart toolbar icon, i.e. with three input files, a click on the Master Timeline toolbar icon opens three aligned Master Timeline charts in the same view instead of just one. To distinguish between the trace files depicted in the charts, all charts belonging to the same trace file have an individual background color. Figure 16 depicts a Compare View with Master Timeline, Process Timeline, and Function Summary charts. Due to the fact that the Compare View automatically couples the time range of all trace files, the displays can be used to compare performance characteristics. As can be seen in Fig. 16, trace A (white background) has the longest overall execution time. The execution time of trace C (blue background) is very short and almost invisible. Zooming into the computation phase of trace C reveals its details but hides the corresponding computation phases of trace A and B (green background) as depicted in Fig. 17. This behavior is clearly not desirable. In order

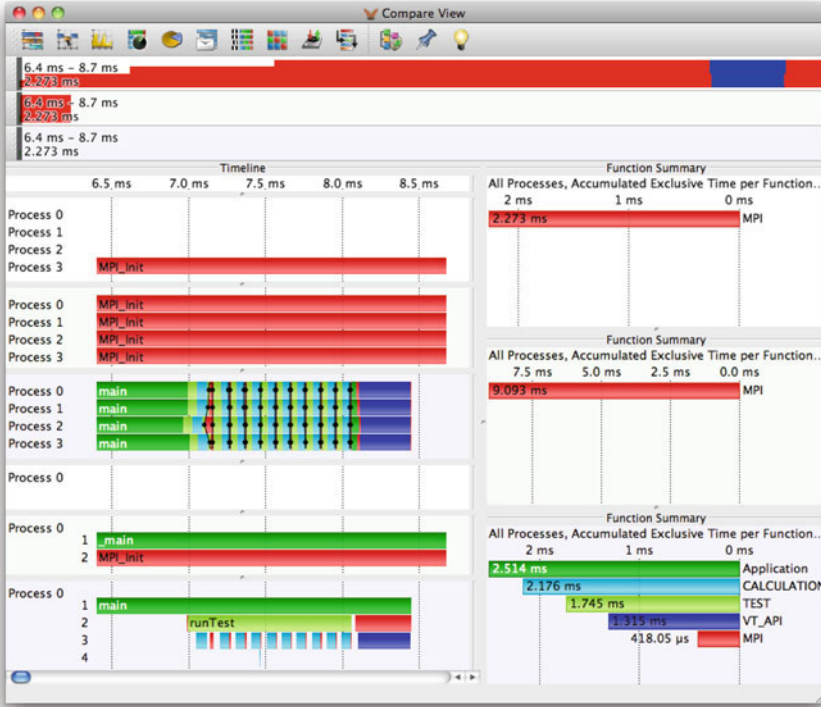


Fig. 17 Unaligned view of computation phase (ten iterations) of program run C (blue background)

to compare the computation phases across multiple traces, the trace files need to be aligned to the starting points of the computation phases.

5.2 Alignment of Multiple Trace Files

The Compare View functionality to shift individual trace files in time allows to compare areas of the data that did not occur at the same time. For instance, in order to compare the iterations of the three example trace files these areas need to be aligned to each other. This is required due to the fact that the initialization of the application took different times on the three computer systems.

There are several ways to shift the trace files in time. One way is to use the context menu of the *Navigation Toolbar*. A right click on the toolbar displays the context menu. The entry *Set Time Offset* allows to manually set an arbitrary time offset for the trace file.

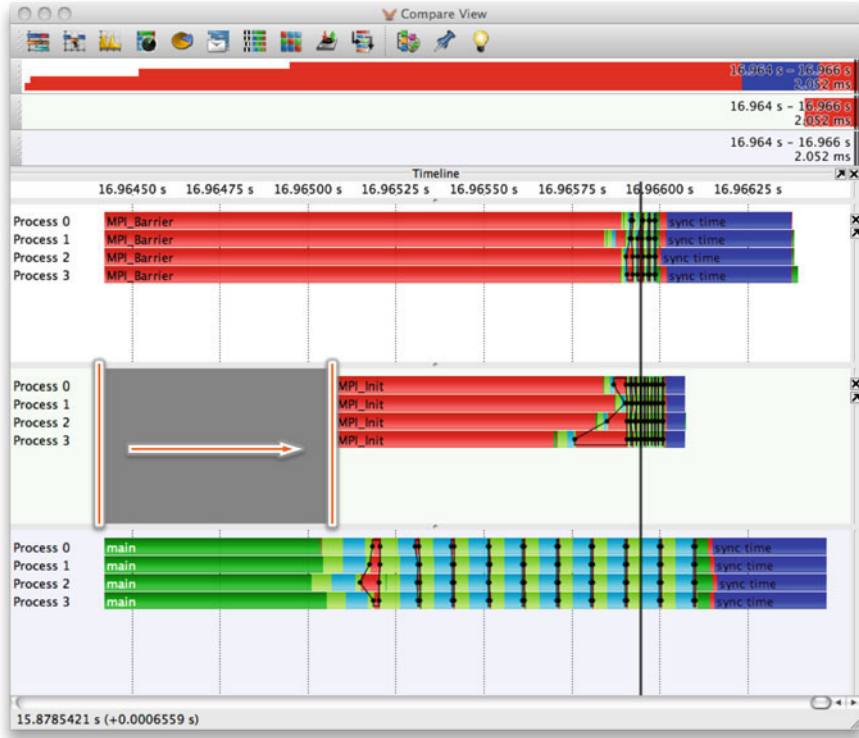


Fig. 18 Alignment process of trace A and B. While holding down the *Ctrl* modifier key, the trace can be dragged with the left mouse button

The easiest way to achieve a coarse alignment is to drag the trace file in the Navigation Toolbar itself. Holding down the *Ctrl* modifier key and the left mouse button, the trace can be moved to the desired position in time. After the coarse shifting has been carried out for all three traces, a finer alignment can be done in the Master Timeline itself. First of all the user needs to zoom into the area to be aligned and compared. While keeping the *Ctrl* modifier key pressed, the trace can be dragged with the left mouse button. Figure 18 depicts the process of dragging trace B to the beginning of trace A's iterations. Once the traces are aligned it becomes quite obvious that although the initialization phase of run A took the longest, this system was the fastest in computing the actual iterations. While this is just a minor finding for a basic example, real world load distribution issues and arbitrary temporal deviations can be documented, compared, and presented efficiently for the entire evolution of a given program.

6 Conclusion

This article introduces new graphical approaches to customize event-based performance analysis of HPC software. A semi-transparent heat map overlay layer now allows to analyze both structural and performance behavior of any given HPC application at the same time. The intuitive graphical editor for derived performance metrics presented in this article enables users to study custom effects specific to their application, which is particularly useful for refining and narrowing the investigation process. Today's complex procedural invocation hierarchies are addressed by means of combinable filter rules, which allow to adapt the performance visualization (timelines and profiles) to the relevant parts of a given program. Finally, a new powerful compare- and alignment-mode is outlined, which allows to compare multiple program executions in detail, i.e. step by step per procedure invocation. Detailed examples demonstrate the above approaches and techniques in the Vampir tool suite.

References

1. Brunst, H., Knüpfer, A.: Vampir. In: D. Padua (ed.) *Encyclopedia of Parallel Computing*, pp. 2125–2129. Springer US (2011). DOI 10.1007/978-0-387-09766-4_60
2. Bura, H., Widera, R., Hönig, W., Juckeland, G., Debus, A., Kluge, T., Schramm, U., Cowan, T., Sauerbrey, R., Bussmann, M.: PIconGPU: A fully relativistic particle-in-cell code for a GPU cluster. *Plasma Science, IEEE Transactions on* **38**(10), 2831–2839 (2010). DOI 10.1109/TPS.2010.2064310
3. Hackenberg, D., Schöne, R., Molka, D., Müller, M., Knüpfer, A.: Quantifying power consumption variations of HPC systems using SPEC MPI benchmarks. *Computer Science - Research and Development* **25**, 155–163 (2010). URL <http://dx.doi.org/10.1007/s00450-010-0118-0>
4. Kogge, P.: The tops in flops. *IEEE Spectrum* **48**, 48–54 (2011). DOI 10.1109/MSPEC.2011.5693074. URL <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5693074>
5. Pennycook, S.J., Hammond, S.D., Jarvis, S.A., Mudalige, G.R.: Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. *SIGMETRICS Perform. Eval. Rev.* **38**(4), 23–29 (2011). DOI 10.1145/1964218.1964223. URL <http://doi.acm.org/10.1145/1964218.1964223>
6. Skamarock, W.C., Klemp, J.B.: A time-split nonhydrostatic atmospheric model for weather research and forecasting applications. *Journal of Computational Physics* **227**(7), 3465–3485 (2008). DOI 10.1016/j.jcp.2007.01.037
7. William, T., Berry, D.K., Henschel, R.: Analysis and optimization of a molecular dynamics code using PAPI and the Vampir toolchain. In: *Cray User Group 2012*. CUG, Stuttgart (2012). URL <https://portal.futuregrid.org/sites/default/files/2012-CUG-pap142.pdf>

Extending Scalasca's Analysis Features

Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube,
and Zoltán Szebenyi

Abstract Scalasca is a performance analysis tool, which parses the trace of an application run for certain patterns that indicate performance inefficiencies. In this paper, we present recently developed new features in Scalasca. In particular, we describe two newly implemented analysis methods: the root cause analysis which tries to identify the cause of a delay and the critical path analysis, which analyses the path of execution that determines the application runtime. Furthermore, we present time-series profiling, a method that allows to explore time-dependent behavior of an application. Finally, we extended the means of Scalasca and its output format CUBE to define and display topologies.

1 Introduction

Today, high performance computers provide the computing power which is required by the complexity of many scientific computations. However, providing larger and more powerful computer systems is useless if the applications do not make efficient use of the available resources. Especially since the clock rate will no longer continue to grow, the performance increase is due to increasing the parallelism of the computer systems.

However, the complexity of parallel programs is much higher than sequential programs. Furthermore, the hardware is more complex, too. Instead of a single processor, the programmer must deal with e.g. networks, data transfer rates and hierarchical memory. Thus, the optimization becomes much more difficult.

D. Lorenz (✉) · B. Mohr · A. Strube

Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany
e-mail: d.lorenz@fz-juelich.de

D. Böhme · Z. Szebenyi

German Research School of Simulation Science GmbH, Aachen, Germany

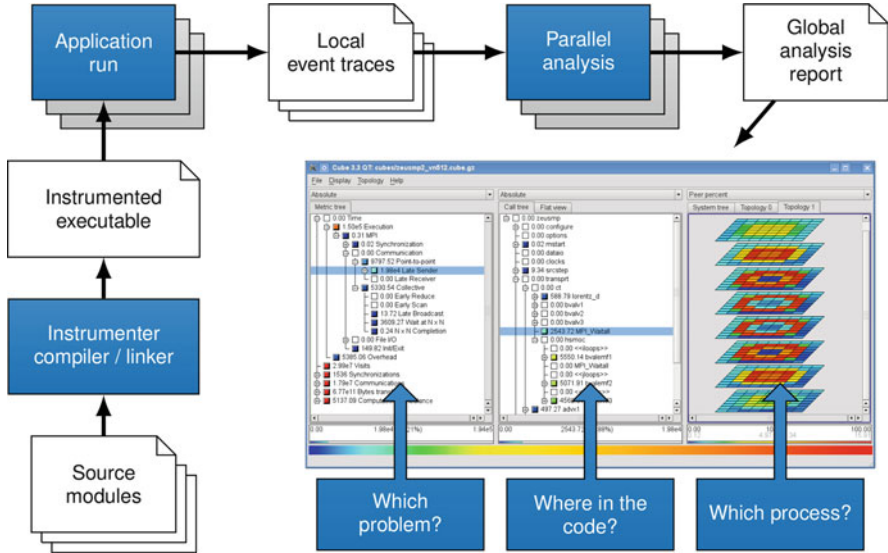


Fig. 1 The performance analysis workflow with Scalasca

Before a programmer can optimize the performance of his program, he must understand the performance behavior of his program and identify causes of performance reduction. In order to analyze the performance of a program, various performance analysis tools have been developed, like Scalasca [6], Vampir [8], TAU [13], Periscope [7] and Paraver [11]. Although each of the tools has its specific features and methods for analysis and display of results, there are some common techniques.

Some tools [1, 6, 13] can create a so called *profile*. It consists of aggregated statistics of performance metrics like runtime and/or number of visits for every function or call path and/or for every thread. These statistics give an overview over the execution.

Another approach is to record all events, e.g. function entry/exit, and its associated performance data like timestamps in a so called *trace*. This allows a fine grained analysis of the application. The user can visualize the trace directly, e.g., with Vampir [8] or Paraver [11]. However, manually searching the whole trace is very tedious. Another possibility, which is used by Scalasca [6], is to automatically examine an event trace for certain patterns of inefficient behavior. This search guarantees to cover the whole trace.

Figure 1 shows the performance analysis workflow with Scalasca. To instrument his application, a user must prefix the original compile and link command with the scalasca instrumenter. During application run, the events are recorded into a trace file. The Scalasca parallel trace analyzer analyses the trace and writes an analysis output which can be interactively explored in the CUBE graphical user interface (GUI). The CUBE GUI has three panes. The left panes allows to select a metric,

e.g. time waited, due to a late sender. The middle pane shows a call tree which provides information where in the code, a particular issue appeared. The right pane shows the distribution of values over all processes for the selected call path.

Scalasca uses a highly scalable event-trace replay mechanism for its analysis. Hereby, the trace for every process is stored in a separate file. The trace analyzer is launched as a separate program after the target application finished and runs on the same number of processes as the application ran. Every analysis process traverses the trace of one application process. Whenever the application process communicated with another process, Scalasca exchanges information, too. Furthermore, a backward replay is also possible.

In this paper, we will present extensions and new features to Scalasca. First, we added two new analysis methods to our automatic trace analysis. The first new method is called *root-cause analysis*, which identifies the root causes of wait states in MPI synchronization points. It is described in Sect. 2. Afterwards, Sect. 3 describes the second new analysis method, called *critical-path analysis*. It is able to detect inefficiencies that otherwise may be hidden through data aggregation.

Usually, profiling tool designs assume that the iterations of an iterative application behave basically the same. However, this is not generally true. Section 4 presents a new feature to analyze time dependent behavior.

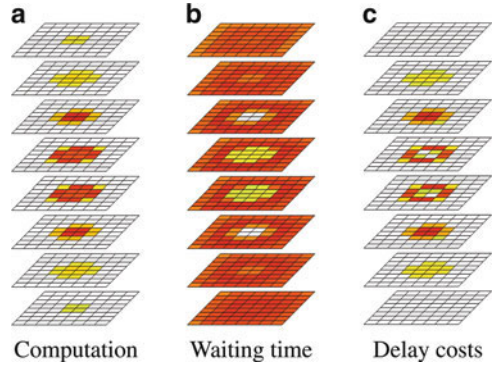
Section 5 describes the enhancements of the possibilities to define and display topologies.

2 Root Cause Analysis

So far, Scalasca's trace analysis could identify wait states at MPI synchronization points. Wait states, which are intervals through which a process is idle while waiting for a delayed process, are a primary symptom of load imbalance in parallel programs. The new root-cause analysis [2] also identifies the root causes of these wait states and calculates the costs of delays in terms of the waiting time that they induce.

A *delay* is the original source of a wait state, that is, an interval that causes a process to arrive belatedly at a synchronization point, causing one or more other processes to wait. Besides simple computational overload, delays may include a variety of behaviors such as serial operations or centralized coordination activities that are performed only by a designated process. The *costs* of a delay are the total amount of wait states it causes. Wait states can also themselves delay subsequent communication operations and produce further indirect wait states. This *propagation effect* does not only add to the total costs of the original delay, but also creates a potentially large temporal and spatial distance in between a wait state and its original root cause. The root-cause analysis closes this gap by mapping the costs of a delay onto the call paths and processes where the delay occurs, offering a high degree of guidance in identifying promising targets for load or communication balancing. Together with the analysis of wait-state propagation effects, the delay

Fig. 2 Distribution of computation time, waiting time, and total delay costs in Zeus-MP/2 across the $8 \times 8 \times 8$ three-dimensional computational domain. *Red colors* indicate high values. (a) Computation. (b) Waiting time. (c) Delay costs



costs enable a precise understanding of the root causes and the formation of wait states in parallel programs.

We applied the delay analysis to a variety of real-world MPI programs. One example is the astrophysics code Zeus-MP/2, where we studied the formation of wait states in a simulation of a 3D blast wave over 100 time steps on 512 processes. Around 12.5 % of the program’s total CPU allocation time is waiting time. Scalasca’s report browser can visualize the Cartesian process topology of a program, which we use in Fig. 2 to illustrate the relation between waiting and delaying processes in terms of their position within the computational domain. Obviously, there is a computational load imbalance between the central and outer ranks of the domain. Accordingly, the underloaded processes exhibit a significant amount of waiting time (Fig. 2b). Our analysis shows that about 70 % of the waiting time was indirectly caused by wait-state propagation. Examining the delay costs reveals that almost all the delay originates from the border processes of the central, overloaded region (Fig. 2c). The distribution of the workload explains this observation: Within the central and outer regions, the workload is relatively well balanced. Therefore, communication within the same region is not significantly delayed. In contrast, the difference in computation time between the central and outer region causes wait states at synchronization points along the border, which subsequently propagate towards the outer domain border. By pinpointing one subroutine and three computational loops with particularly high delay costs, the delay analysis also helped isolating the imbalanced source-code regions that lead to the wait states.

3 Critical Path Analysis

Our search for compact yet powerful means to uncover inefficiencies in parallel programs has led us to revisit the critical path as a key performance structure. Although the power and expressiveness of the critical path has been demonstrated in previous work, critical-path techniques only play a minor role in current

performance analysis tools. This arises partly from the difficulty in isolating the critical path, but also from the inability to extract intuitively accessible insight from the available information. Our work [3] addresses both issues. We leverage Scalasca's parallel trace replay technique to isolate the critical path in a highly scalable way. Also, instead of exposing the lengthy critical-path structure to the user in its entirety, we use the critical path to derive a set of compact performance indicators, which provide intuitive guidance about load-balance characteristics and quickly draw attention to potentially inefficient code regions.

The critical path is the longest path through a program activity graph that does not include wait states. Thus, it determines the length of program execution. Prolonging activities on the critical path increases program runtime, whereas shortening them (usually) reduces it. In contrast, optimizing an activity not on the critical path only increases waiting time, but does not affect the overall runtime.

Our critical-path analysis produces a critical-path profile, which represents the time an activity spends on the critical path. In addition, we combine the critical-path profile with per-process time profiles to create a *critical-path imbalance* performance indicator. This critical-path imbalance corresponds to the time that is lost due to inefficient parallelization in comparison with a perfectly balanced program. As such, it provides similar guidance as prior profile-based load imbalance metrics (e.g., the difference of maximum and average aggregate workload per process), but the critical-path imbalance indicator can draw a more accurate picture. The critical path retains dynamic effects in the program execution, such as shifting of imbalance between processes over time, which per-process profiles simply cannot capture. Because of this, purely profile-based imbalance metrics regularly underestimate the actual performance impact of a given load imbalance. As an extreme example, consider a program like the example in Fig. 3, where a function is serialized across all processes but runs for the same amount of time on each. Purely per-process profile based metrics would not show any load imbalance at all, whereas the critical-path imbalance indicator correctly characterizes the functions serialized execution as a performance bottleneck.

Both delay analysis and critical-path analysis are implemented as extensions to the automatic wait-state detection of the Scalasca performance analysis toolset, leveraging its scalable, post-mortem event-trace analysis. The analyzer traverses the traces in parallel, iterating over each process-local trace, and exchanges data required for the performance analysis at each recorded synchronization point using a communication operation similar to the one originally used by the program.

Other than the pure wait-state analysis, the delay and critical-path analysis require an additional, backward replay over the trace. A backward replay processes a trace backwards in time, from its end to its beginning, and reverses the role of senders and receivers. Overall, the analysis now consists of two stages: (1) a parallel forward replay that performs the wait state analysis and annotates communication events with information on synchronization points and waiting time incurred; and (2) a parallel backward replay that identifies the delays causing each of the wait states detected during forward replay, calculates their costs, and extracts the critical path. Starting at the endmost wait states, the backward replay allows delay costs

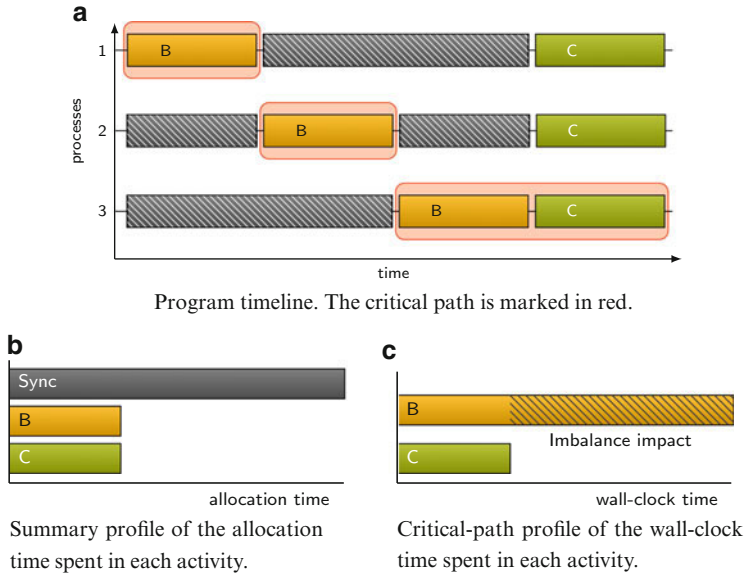


Fig. 3 Analysis of dynamic performance effects: The serialized execution of function B, as seen in the timeline (a), goes unnoticed in a summary profile (b), but is correctly identified as a performance bottleneck by the critical-path imbalance indicator (c)

to travel from the point where they materialize in the form of wait states back to the place where they are caused by delays. The backward replay also facilitates the critical-path analysis, since the route of the critical path through the program cannot be determined without knowing the end of the execution. For MPI programs, the critical path runs between `MPI.Init` and `MPI.Finalize`. Our critical-path search begins by determining the MPI rank that entered `MPI.Finalize` last, which marks the endpoint of the critical path, and then exploits the lack of wait states on the critical path: whenever a wait state is found on the currently active path, the search proceeds on the MPI rank that caused the wait state. This way, we follow the entire critical path backwards through the trace.

4 Time-Series Profiling

Call path profiling accumulate multiple visits of the same call path. If all visits of the same call path behave basically the same, all visits are well represented by the resulting statistics. For iterative applications, the user usually assumes very similar behavior of each iteration. However, this assumption is not always true. Szebebyi et al. [15] have shown on examples from the SPEC MPI benchmarks (see Fig. 4) and on the coulomb solver PEPC [16] (see Fig. 5) that some applications change their behavior for different iterations.

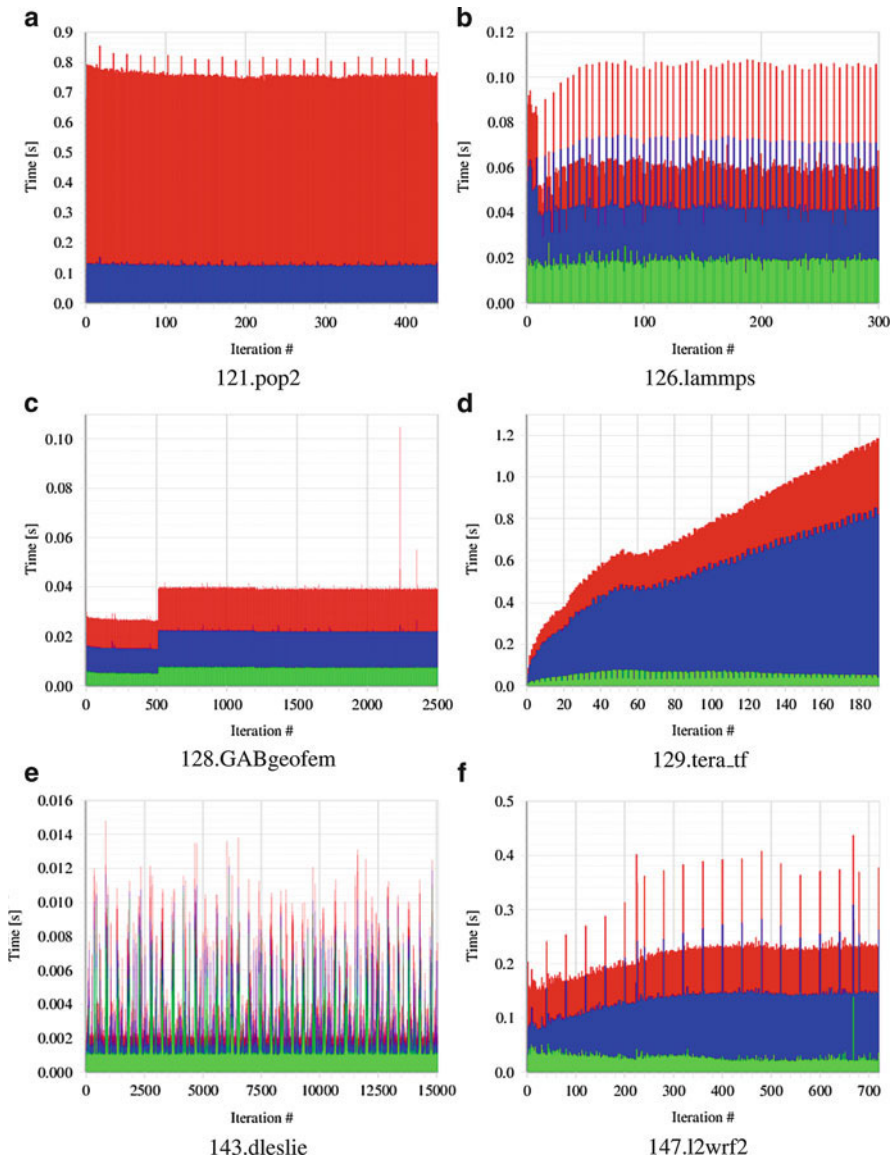
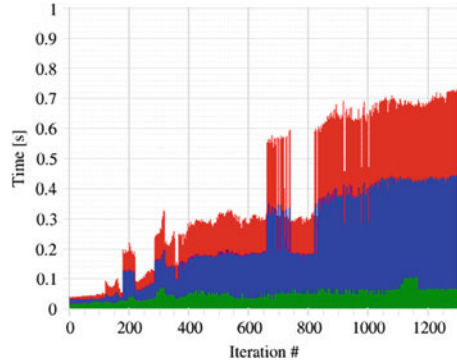


Fig. 4 Runtime of iterations of the SPEC MPI benchmark codes 121.pop, 126.lammps, 128.GABgeofem, 129.tera_tf, 143.dleslie, and 147.l2wrf2

To display time-dependent behavior of iterative applications, the Scalasca measurement system can now record separate profiles for every iteration of the main loop. For this purpose, a user can manually instrument the body of the main loop using the EPIK API. TAU [13] and Score-P [10] provide similar concepts named dynamic timer, or dynamic region, respectively.

Fig. 5 Point-to point communication time of the PEPC code for each program iteration



However, for applications with many iterations such a time-series profile may become very large. To reduce the memory requirements of the profile, we added a mechanism which clusters similar iterations to a single profile sub-tree instance [14]. The maximum number of clusters is configurable. However, we only cluster iterations with a similar structure. Iterations which contain different call paths are never clustered together. The time dynamics can be reconstructed from a mapping table, which stores the cluster associated with each iteration.

In case of the SPEC MPI benchmarks, the clustered profiles match the profiles without clustering very well, even if only 64 clusters are used. Figure 6 shows the comparison between the runtimes of iterations with clustering and without clustering. In the case of clustering the more complex PEPC coulomb solver, count based metrics show already a good match when using 64 clusters. However, time based metrics require more clusters for a good match [14].

5 Topologies

Scalasca obtains performance data for every process/thread of the application. In many cases, the hardware, the network, the communication system (e.g. MPI), or the application define neighborhood/dependency relationships and/or structure on these processes/threads, called topologies. However, network structure, hardware hierarchy, and application neighborhood relationships may significantly influence the performance of an application. Thus, we extended Scalasca's capabilities to record and display topologies. The previous capabilities of Scalasca to represent topologies were limited to 3-dimensional Cartesian topologies, i.e. each element has an unique set of coordinates in a 3-dimensional realm. Now, the topologies defined in Scalasca can have any number of dimensions. Furthermore, an application can define more than one topology, e.g., to compare the results for hardware topology and an algorithms domain topology.

Scalasca can work with the following types of Cartesian topologies:

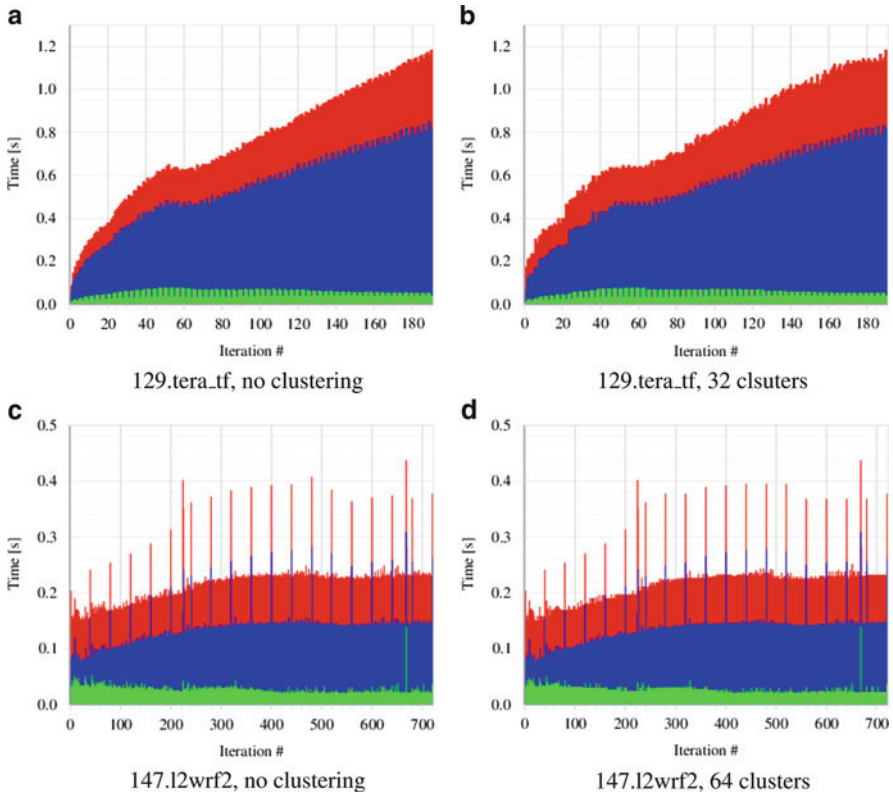


Fig. 6 comparison between the runtimes of each iteration without clustering and with clustering for the SPEC MPI benchmarks 129.tera_tf, and 147.l2wrf2. (a) 129.tera_tf, no clustering. (b) 129.tera_tf, 32 clusters. (c) 147.l2wrf2, no clustering. (d) 147.l2wrf2, 64 clusters

5.1 Hardware Topologies

Supercomputers, such as the IBM Blue Gene series, can report where on the machine processes run. This is useful in tightly-coupled programs, where distance information can provide insights on communication delays. This information is collected automatically by Scalasca during measurement.

Going further with the example of the Blue Gene series, the Blue Gene/Q model of supercomputer has a five-dimensional torus network, which is fully supported by Scalasca. Besides the support of topologies with more than 3 dimensions on measurement and analysis, Scalasca now has a “folding” mechanism which allows the user to select three-dimensional slices of the topology for visualization. Scalasca now also supports names for all the topologies and their dimensions, thus, the topologies can be easily identified. The names of the dimensions also are helpful to identify the individual processes/threads in complex hardware topologies such as the Blue Gene/Q’s (see Fig. 7).

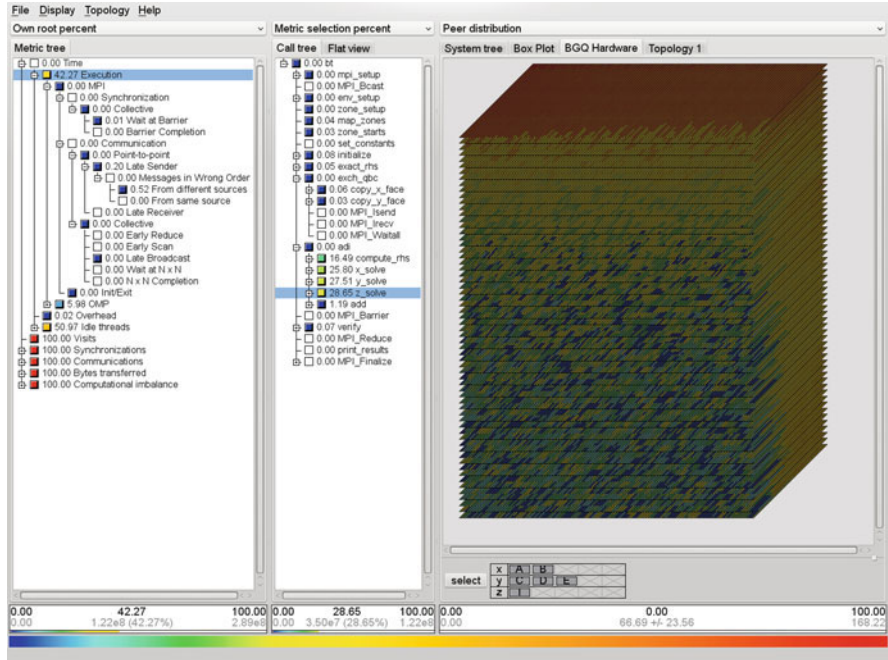


Fig. 7 CUBE showing the Blue Gene/Q's 5D Hardware Topology. The screenshot shows 524,288 threads on the Jülich BlueGene/Q machine

5.2 Processes X Threads

Scalasca now automatically creates a two-dimensional virtual topology that shows all OpenMP threads belonging to each process.

5.3 Runtime Mapping

On MPI programs, for instance, Scalasca can show how the processes are distributed in ranks and the relationship between them. This information is also collected automatically by Scalasca during measurement. Scalasca now supports any number of dimensions. In the future, topologies whose communicator was named using the `MPI_Comm_set_name` function will have this name shown in the topology's tab.

5.4 Algorithm Domain

The MPI specification does not enforce a strict mapping between neighbor ranks and the hardware, neither between ranks and the problem decomposition, which is

application specific. In some specific cases, the user might benefit from creating a virtual topology that is independent from the hardware and from the MPI mapping, being specific to the application. These topologies can be created manually by the user, using our API.

6 Future Work

As computational science evolves and new programming paradigms emerge, new challenges for performance analysis appear. Thus, we will continue to improve Scalasca. This section describes some of the ongoing developments and planned features for Scalasca.

First, we will replace Scalasca's native instrumentation and measurement system by Score-P [10]. Score-P is a common instrumentation and measurement system, initially used by Periscope [7], Scalasca [6], TAU [13], and Vampir [8]. This implies that these tools will also share common data formats for tracing and profiling. Scalasca will remain as a pure trace analyzer for traces written in the Score-P OTF2 [4] trace format. Score-P profiles and Scalasca trace analysis reports will be written in the CUBE4 format, which is the more scalable successor of the current CUBE3.

The common Score-P measurement stack improves the interoperability of the tools. Furthermore, it reduces the need for each tool to maintain its own instrumentation and measurement system, and thus, allows tool developers to focus on the specific analysis strengths of their tools.

With respect to future trace analysis enhancements, we plan to extend the current OpenMP analysis of Scalasca with the analysis of OpenMP tasks. Score-P can already record task events [9, 12]. However, we must extend Scalasca's profile construction algorithm and we want to add some task specific patterns to its analysis.

At the Barcelona Supercomputing Centre, a new tasking system was developed, named OmpSs [5]. We want to support measurements of applications using OmpSs. Therefore, we will implement instrumentation and measurement support for OmpSs in Score-P. Furthermore, we want to create the task analysis general enough that it covers also OmpSs tasks. A third new analysis feature that we are working on is the analysis of one-sided communication and PGAS languages.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* **22**, 685–701 (2010). DOI <http://dx.doi.org/10.1002/cpe.v22:6>

2. Böhme, D., Geimer, M., Wolf, F., Arnold, L.: Identifying the root causes of wait states in large-scale parallel applications. In: Proc. of the 39th International Conference on Parallel Processing (ICPP, San Diego, CA, USA), pp. 90–100. IEEE Computer Society (2010). DOI 10.1109/ICPP.2010.18
3. Böhme, D., de Supinski, B.R., Geimer, M., Schulz, M., Wolf, F.: Scalable critical-path based performance analysis. In: Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China (2012)
4. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W., Wolf, F.: Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In: Proc. of the Intl. Conference on Parallel Computing (ParCo) 2011, Ghent, Belgium, *Advances in Parallel Computing*, vol. 22, pp. 481–490. IOS press (2012). DOI 10.3233/978-1-61499-041-3-481
5. Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia, R., Ayguade, E., Labarta, J.: Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In: Languages and Compilers for Parallel Computing, *LNCSC*, vol. 6548, pp. 215–229 (2011). URL http://dx.doi.org/10.1007/978-3-642-19595-2_15
6. Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* **22**(6), 702–719 (2010). DOI 10.1002/cpe.1556
7. Gerndt, M., Ott, M.: Automatic performance analysis with Periscope. *Concurrency and Computation: Pract. Exper.* **22**(6), 736–748 (2010). DOI 10.1002/cpe.1551
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M., Nagel, W.: The Vampir performance analysis tool set. In: Tools for High Performance Computing, pp. 139–155. Springer (2008)
9. Lorenz, D., Philippen, P., Schmidl, D., Wolf, F.: Profiling of OpenMP tasks with Score-P. In: Proc. of Third International Workshop of Parallel Software Tools and Tool Infrastructures (PSTI 2012), Pittsburgh, PA, USA (2012)
10. an Mey, D., Biersdorff, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S.S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A unified performance measurement system for petascale applications. In: Competence in High Performance Computing 2010 (CtHPC), pp. 85–97. Gauß-Allianz, Springer (2012). DOI 10.1007/978-3-642-24025-6_8
11. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A tool to visualize and analyze parallel code. In: WoTUG-18: Transputer and occam Developments, pp. 17–31 (1995)
12. Schmidl, D., Philippen, P., Lorenz, D., Rössel, C., Geimer, M., an Mey, D., Mohr, B., Wolf, F.: Performance analysis techniques for task-based OpenMP applications. In: 8th Int. Workshop of OpenMP (IWOMP), *LNCSC*, vol. 7312, pp. 196–209. Springer, Berlin / Heidelberg (2012)
13. Shende, S., Malony, A.D.: The TAU parallel performance system. *International Journal of High Performance Computing Applications* **20**(2), 287–331 (2006)
14. Szebenyi, Z., Wolf, F., Wylie, B.J.N.: Space-efficient time-series call-path profiling of parallel applications. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA. ACM (2009). DOI 10.1145/1654059.1654097
15. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proc. of the 1st SPEC International Performance Evaluation Workshop (SIPEW), Darmstadt, Germany, *Lecture Notes in Computer Science*, vol. 5119, pp. 99–123. Springer (2008). DOI 10.1007/978-3-540-69814-2_8
16. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: Scalasca parallel performance analyses of PEPC. In: Proc. of the 1st Workshop on Productivity and Performance (PROPER) in conjunction with Euro-Par 2008, Las Palmas de Gran Canaria, Spain, *Lecture Notes in Computer Science*, vol. 5415, pp. 305–314. Springer (2009). DOI 10.1007/978-3-642-00955-6_35

The HOPSA Workflow and Tools

**Bernd Mohr, Vladimir Voevodin, Judit Giménez, Erik Hagersten,
Andreas Knüpfer, Dmitry A. Nikitenko, Mats Nilsson, Harald Servat,
Aamer Shah, Frank Winkler, Felix Wolf, and Ilya Zhukov**

Abstract To maximise the scientific output of a high-performance computing system, different stakeholders pursue different strategies. While individual application developers are trying to shorten the time to solution by optimising their codes, system administrators are tuning the configuration of the overall system to increase its throughput. Yet, the complexity of today’s machines with their strong interrelationship between application and system performance presents serious challenges to achieving these goals. The HOPSA project (HOListic Performance System Analysis) therefore sets out to create an integrated diagnostic infrastructure for combined application and system-level tuning – with the former provided by the EU and the latter by the Russian project partners. Starting from system-wide basic performance screening of individual jobs, an automated workflow routes findings on potential bottlenecks either to application developers or system administrators with recommendations on how to identify their root cause using more powerful diagnostic tools. Developers can choose from a variety of mature performance-analysis tools developed by our consortium. Within this project, the tools will be

B. Mohr (✉) · I. Zhukov
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Juelich, Germany
e-mail: b.mohr@fz-juelich.de

V. Voevodin · D.A. Nikitenko
Moscow State University, RCC, Moscow, Russia

J. Giménez · H. Servat
Barcelona Supercomputing Centre, Barcelona, Spain

F. Wolf · A. Shah
German Research School for Simulation Sciences GmbH / RWTH Aachen University,
Aachen, Germany

E. Hagersten · M. Nilsson
Rogue Wave Software AB, Uppsala, Sweden

A. Knüpfer · F. Winkler
Technical University Dresden, Dresden, Germany

further integrated and enhanced with respect to scalability, depth of analysis, and support for asynchronous tasking, a node-level paradigm playing an increasingly important role in hybrid programs on emerging hierarchical and heterogeneous systems.

1 Introduction

To maximise the scientific and commercial output of a high-performance computing system, different stakeholders pursue different strategies. While individual application developers are trying to shorten the time to solution by optimising their codes, system administrators are tuning the configuration of the overall system to increase its throughput. Yet, the complexity of today's machines with their strong interrelationship between application and system-level performance demands an interrelation of application and system programming.

The HOPSA project (HOListic Performance System Analysis) therefore sets out for the first time to develop an integrated diagnostic infrastructure for combined application and system-level tuning. Using more powerful diagnostic tools application developers and system administrators will easier identify the root causes of their respective bottlenecks. With the HOPSA infrastructure, it is more effective to optimise codes running on HPC systems. More efficient codes mean either getting results faster or being able to get higher quality or more results in the same time.

The work in HOPSA is carried out by two coordinated projects funded by the EU under call FP7-ICT- 2011-EU-Russia and the Russian Ministry of Education and Science, respectively. Its objective is the new innovative integration of application tuning with overall system diagnosis and tuning to maximise the scientific output of our HPC infrastructures. While the Russian consortium will focus on the system aspect, the EU consortium will focus on the application aspect. At the interface between these two facets of our holistic approach, which is illustrated in Fig. 1, is the system-wide performance screening of individual jobs, pointing at both inefficiencies of individual applications and system-related performance issues.

This article describes the overall workflow of performance analysis of parallel programs using the HOPSA infrastructure, introduces the individual tools developed inside the project consortium, and shows how to use the tools in a complementary way. For detailed information, please see the user guides of the individual performance-analysis tools.

At the centre of this workflow is the so-called lightweight measurement module (LWM²). It is responsible for the first step in the workflow, the system-wide mandatory collection of basic performance data. For each execution on the cluster, LWM² produces a so-called job digest. The metrics listed in this compact report indicate whether an application suffers from an inherent performance problem or whether application interference may have been at the root of dissatisfactory behaviour. They also provide a first assessment regarding the nature of a potential performance problem and help to decide on further diagnostic steps using any of the

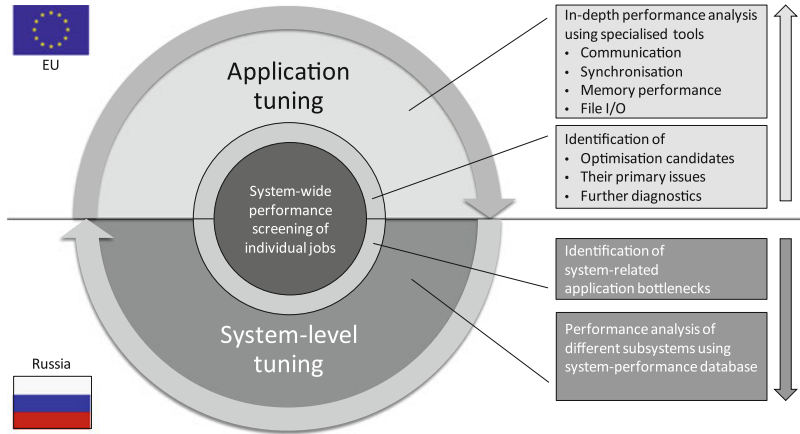


Fig. 1 System-level tuning (*bottom*), application-level tuning (*top*), and system-wide performance screening (*centre*) use common interfaces for exchanging performance properties

more powerful performance-analysis tools. For each of those tools, a short summary is given with information on the most important questions it can help to answer. Moreover, the document covers Score-P [5], a common measurement infrastructure shared by some of the tools. The performance data types supported by Score-P form a natural refinement hierarchy that can be followed to track down and represent even complex bottleneck situations at increasing levels of granularity. Finally, a brief excursion on system-level tuning explains how system providers can leverage the data collected by LWM² to identify a suboptimal system configuration or faulty components.

2 The HOPSA Workflow

2.1 Overview

The performance-analysis workflow (Fig. 2) consists of two basic steps. During the first step, we identify all those applications running on the system that may suffer from inefficiencies. This is done via system-wide job screening supported by a lightweight measurement module (LWM²) dynamically linked to every executable. The screening output identifies potential problem areas such as communication, memory, or file I/O, and issues recommendations on which diagnostic tools can be used to explore the issue further. Available application performance analysis tools include Paraver/Dimemas [1, 12], Scalasca [2], ThreadSpotter [3], and Vampir [4]. The data collected by LWM² is also fed into the Clustrx.Watch hierarchical cluster monitoring system [13] which combines it with system and hardware data and

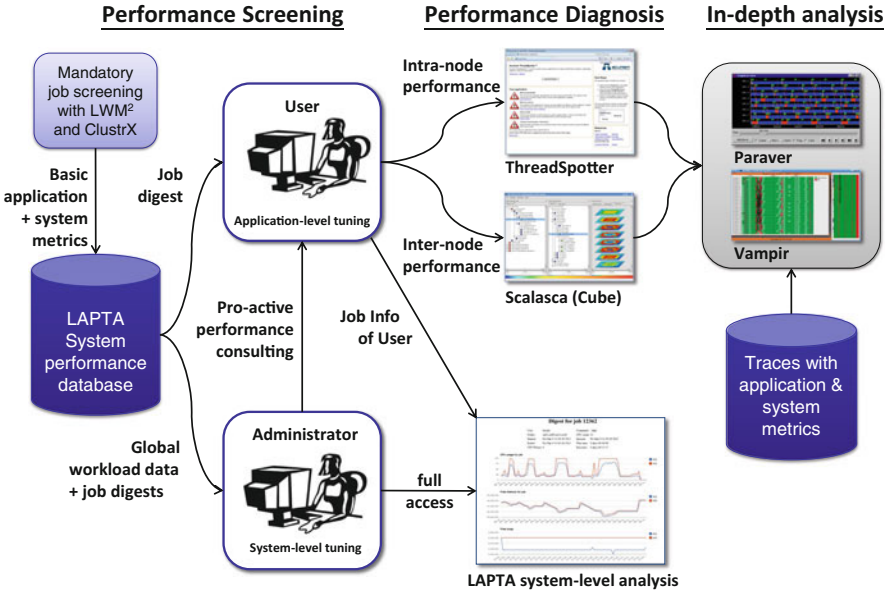


Fig. 2 Overview of the performance analysis workflow

forwards it to the LAPTA cluster monitoring and analysis system [14] for further analysis by system administrators.

In general, the workflow successively narrows the analysis focus and increases the level of detail at which performance data are collected. At the same time, the measurement configuration is optimised to keep intrusion low and limit the amount of data that needs to be stored. To distinguish between system and application-related performance problems, some of the tools allow also system-level data to be retrieved and displayed. The system administrator, in contrast, has access to global performance data. He can use this data to identify potential system performance bottlenecks and to optimise the system configuration based on current workload needs. In addition, the administrator can identify applications that continuously underperform and proactively offer performance-consulting services. In this way, it becomes possible to reduce the unnecessary waste of expensive system resources.

2.2 Performance Screening

This step decides whether an application behaves inefficiently. On the side of the user, nothing has to be done except running the application as usual. Upon application start, LWM² is automatically and transparently linked to the executable through library pre-loading. At runtime, the module collects basic performance data with very low overhead. The performance data characterise various aspects such as

sequential performance, parallel performance, and file I/O. At the end of execution, the user receives a job digest that contains the most important performance metrics. The digest also recommends further diagnostics in the case certain key metrics show unexpected values, which may often be indicative of a performance problem. If needed, the user can disable LWM², for example, to avoid interference with the analysis tools used in subsequent stages of the tuning process.

2.2.1 The Lightweight Measurement Module LWM²

The lightweight measurement module LWM² collects basic performance data for every process of a parallel application. It supports applications based on MPI and multithreaded applications based on POSIX Threads or any higher-level model implemented on top of it, which usually includes OpenMP. Multithreaded MPI applications and applications that additionally use CUDA are supported as well. To keep the overhead at a minimum, the module applies a combination of sampling and careful direct instrumentation via interposition wrappers. Direct instrumentation is needed to track the state of a thread (e.g., whether it executes inside or outside an MPI function) and to access relevant communication or I/O parameters such as the number of bytes sent or written to disk. Based on the state tracking performed by the instrumentation, sampling partitions the execution time into different components such as computation, communication, or I/O. LWM² refrains from direct time measurements as far as possible. Hardware counters deliver basic information on single-node performance. To save storage space, the performance data of individual threads are folded into per-process metrics such as the average number of threads.

In addition to collecting performance data separately for each process, LWM² divides the time axis into disjoint slices, recording selected metrics related to the use of shared resources at this finer granularity. The slices have a length of 10s and are synchronized across the entire machine. Together with the location of each process on the cluster, which LWM² records along with the performance data, LWM² provides performance data for each active cell of a cluster-wide time-space grid. The discretised time axis constitutes the first dimension, the nodes of the system the second one.

The purpose of organising the performance data in this way is threefold: First, by comparing the data of different jobs that were active during the same time slice, it becomes possible to see signs of interference between applications. Examples include reduced communication performance due to overall network saturation or low I/O bandwidth due to concurrent I/O requests from other jobs. Second, by looking at the performance data of the same node across a larger number of jobs and comparing it to the performance of other nodes during the same period, anomalies can be detected that would otherwise be hidden when analysing performance data only on a per-job basis. Third, collecting synchronised performance data from all the jobs running on a given system will open the way for new directions in the development of job scheduling algorithms that take the performance characteristics of individual jobs into account. For example, to avoid file-server contention and

waiting time that may occur in its wake, it might be wiser not to co-schedule I/O-intensive applications. In this way, overall system utilisation may be further improved.

After the expiration of every time slice, LWM² passes the data of the current time slice on to Clustrx.Watch, a system-monitoring infrastructure running on each node. Clustrx augments these data with system data collected using various sensors and forwards them to the LAPTA system performance database.

2.2.2 LAPTA

LAPTA is a pAckage for Performance moniTORing and Analysis. The software is aimed at providing flexible, scalable and extendable infrastructure for system-level performance analysis. It includes special tools and interfaces for data collection supporting various data collectors (Clustrx, Ganglia, LWM², etc.), data storage supporting wide range of databases (MongoDB, Cassandra, etc.) and both stored and streamed data access and analysis. LAPTA provides interfaces to access the collected system monitoring data for both query models: post mortem and on-the-fly. For example LAPTA serves as the basis for Job Digest generation based on system-level performance monitoring data. The screening of general job behavior through Job Digest is very useful for users and tuners to understand the possible bottlenecks that can be seen at a glance (like network overload, bad data locality, inefficient memory usage, too intensive I/O, etc.). Also, performance data of the same application collected over an extended period of time will document the tuning and scaling history of this application allowing to make even more detailed analysis of the dynamic application behavior further. Studying the performance behaviour of the entire job mix will allow to make conclusions on the optimal system configuration for the given workload. For example, system providers will learn whether requirements to amount of physical memory available, I/O or network bandwidth and other system hardware requirements were over- or underestimated.

2.3 *Performance Diagnosis*

This step decides why an application behaves inefficiently. It is only needed if the screening identifies a potential performance problem. Depending on the recommendation made by LWM², the user chooses one or more of the performance-analysis tools offered by the HOPSA tool environment. The general strategy of the diagnosis is to start with an overview and then to go deeper as more information on the problem's root cause becomes available.

Table 1 Classification of tools based on problem class and level of detail

	Intra-node performance	Inter-node performance	I/O
Overview	ThreadSpotter	Score-P profile + cube	Scalasca(Cube)
In-depth analysis	ThreadSpotter, Paraver, Vampir	Scalasca trace analyzer + Cube, Paraver, Vampir	Paraver, Vampir

2.3.1 Overview of the Performance Analysis Tool Suite

An overview of the HOPSA performance analysis tool suite is presented in Table 1.

For the analysis of intra-node performance, ThreadSpotter is the primary tool, with the possibility of more detailed analyses using Paraver. For investigating inter-node performance, looking at a performance profile using Scalasca's Cube browser is a good starting point. For even more detailed analyses, the results of the Scalasca trace-analyser can be displayed in Cube, or the Vampir and Paraver/Dimemas tools can be used for a detailed visual exploration of the traces. For understanding I/O-related issues, profiles displayed in the Cube browser give a good overview, while Vampir can be used for more in-depth analysis.

2.3.2 The Score-P Instrumentation and Measurement System

The Score-P [5] measurement infrastructure is a highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications. It collects performance data that can be analysed using the HOPSA tools Scalasca and Vampir. In addition, it supports the performance tools Persicope [6] and TAU [7] developed outside the HOPSA project. Score-P has been created in the projects SILC and PRIMA funded by the German Ministry of Education and Research and the US Department of Energy, respectively. It will be maintained and further enhanced in a number of follow-up projects including HOPSA.

The main performance data formats produced by Score-P are CUBE-4 [8] for profiles and OTF2 [9] for event traces. Profiles provide a compact performance overview, while event traces allow the in-depth analysis of parallel performance phenomena. While classic profiles aggregate performance metrics across the entire execution, time-series profiles treat individual iterations of the application's main loop separately, which allows studying the temporal evolution of the performance behaviour. They provide less detail than event traces, but can cover longer executions. Together, the above-mentioned options form a hierarchy of performance data types with increasing level of detail. The main advantage of Score-P is that a user needs to become familiar with only one set of instrumentation commands to produce all these data types, which can be analysed using the majority of the tools listed Table 1. Figure 3 provides an overview of the different performance data types supported by Score-P and the tools that can be used to analyse them. Below we cover the individual data types in more detail.

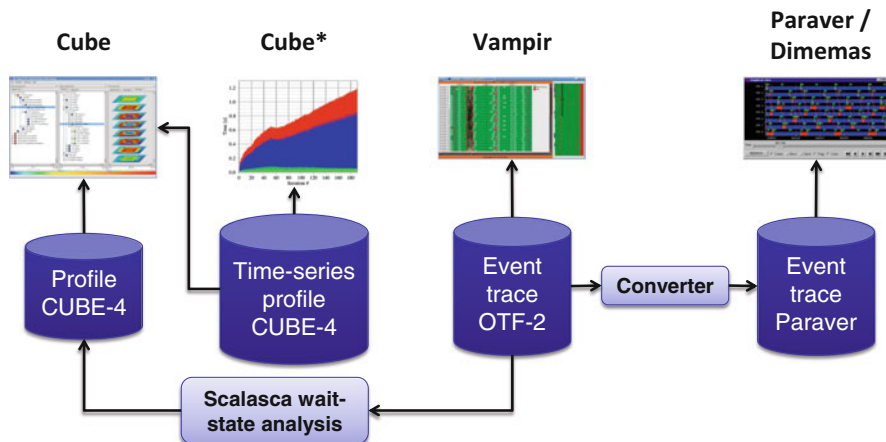


Fig. 3 Performance data types supported by Score-P and the tools that can be used to analyse them. The * next to the second mentioning of Cube indicates a display type that will be provided in a future version

Profiles Profiles in the CUBE-4 format map a set of performance metrics such as the time spent on some activity or the number of messages sent or received onto pairs of call paths and processes (or threads in multithreaded applications). Metrics with a specialization (i.e., subset) relationship can be arranged and displayed in a hierarchy. The call-path dimension forms the natural call-tree hierarchy. Processes and threads are also arranged in an inclusion hierarchy together with hardware components such as the nodes they reside on. In addition, it is possible to define Cartesian process topologies to represent network or virtual topologies. Profiles can be visually explored using the Cube browser. Compared to its predecessor CUBE-3, CUBE-4 files have been optimized for fast writing by storing the metric values in a binary file.

Time-Series Profiles Time-series profiles are like normal CUBE-4 profiles except that they maintain a separate sub-tree in the call tree for each iteration of the time-step loop. This allows the user to distinguish individual iterations and to observe the evolution of the performance behaviour along the time axis. Time-series profiles are created by annotating the body of the time-step loop with special instrumentation, which tells Score-P when an iteration ends and when a new one begins. They can be analysed using the normal Cube display. A future version of Cube (to be completed after this project ends) will provide special iteration diagrams that offer an easy way to judge how the performance changes over time. To avoid that profiling data exceeds the available buffer space, future versions of Score-P will support the dynamic compression of time-series profile data using an online clustering algorithm [15].

Event Traces Event traces include all events of an application run that are of interest for later examination, together with the time they occurred and a number of event-type-specific attributes. Typical events are entering and leaving of

functions or sending and receiving of messages. Event traces produced by Score-P are stored in the Open Trace Format Version 2 (OTF-2), a new trace format whose design is based on the experiences with the two predecessor formats OTF [10] and EPILOG [11], the former native formats of Vampir and Scalasca, respectively. The main characteristics of OTF-2 are similar to other record-based parallel event trace formats. It contains events and definitions and distributes data storage over multiple files. In addition, it is more memory efficient, offering the possibility to achieve measurements with less perturbation due to memory buffer flushes. In contrast to OTF, the event traces are stored in a binary format, which reduces the size of the trace files without the need for a separate compression step. OTF-2 traces are the foundation for further analysis. Vampir can display OTF-2 traces visually using different kinds of displays, including a zoomable timeline. The Scalasca trace analyser identifies wait states and their root causes, producing a CUBE-4 file that provides a higher-level view of the application performance data. This is typically recommended to get an idea of key performance issues before visually exploring the traces directly using a trace browser. Moreover, there is on-going work to convert the traces to the Paraver format so that they can be analysed using Paraver (visual exploration) and Dimemas (what-if analysis).

Overhead Minimisation

Another important aspect is the quality of the collected performance data in terms of intrusion and their size. To keep both intrusion and data size small, the Score-P measurement system offers a systematic approach of expanding the level of detail while at the same time narrowing the measurement focus:

1. Generate a summary profile with generous instrumentation while measuring the overhead. If the overhead is too large ($>10\%$), reduce instrumentation, for example, through the application of filter lists. Measure overhead again and iterate until the overhead is satisfactory.
2. Generate a new summary profile with acceptable overhead. This provides an overview of the performance behaviour across the entire execution time and allows the identification of suspicious call paths and processes.
3. Generate a time-series profile, which provides a separate summary profile for every iteration of the time-step loop. This shows to which degree the performance behaviour changes as the simulation progresses and allows the identification of iterations that warrant deeper analysis. A semantic compression algorithm will ensure that the size of time-series profiles stays within reasonable limits.
4. For the identified iterations, generate event traces. Event traces provide the highest level of detail and offer a number of interesting analysis options including automatic wait-state analysis and visual exploration.

2.4 *The HOPSA Performance Tools*

This section introduces the various HOPSA performance tools.

2.4.1 **Dimemas**

Dimemas [12] is a performance prediction tool for message-passing programs. The Dimemas simulator reconstructs the time behaviour of a parallel application using as input an event trace that captures the time resource demands (CPU and network) of a parallel application. The target machine is modeled by a reduced set of key factors influencing the performance that model linear components like the point-to-point transfer time as well as non-linear factors like resources contention or synchronisation. Using a simple model, Dimemas allows performing parametric studies in a very short time frame. The supported target architecture is a cloud of parallel machines, each one with multiple nodes and multiples CPUs per node allowing the evaluation of a very wide range of alternatives, despite the most common environment is a computing cluster. Dimemas can generate as part of its output a Paraver trace file, enabling the user to conveniently examine the simulated run and understand the application behaviour.

Typical Questions Dimemas Helps to Answer

- How would my application perform in a future system?
- Can increasing the network bandwidth improve the application performance?
- Would my application benefit from asynchronous communication?
- Is my application limited by the network or by serialisation and dependency chains in my code?
- What is the sensitivity of my application to different system parameters?
- What would be the impact of accelerating specific regions of my code?

2.4.2 **Paraver**

Paraver [1] is a very flexible data browser that is part of the CEPBA-Tools toolkit. Its analysis power is based on two main pillars. First, its trace format has no semantics; extending the tool to support new performance data or new programming models requires no changes to the visualiser – just capturing such data in a Paraver trace. The second pillar is that the metrics are not hardwired in the tool but can be programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two timelines. This approach allows displaying a huge number of metrics with the available data. To capture the expert's knowledge, any view or set of views can be saved as a Paraver configuration file. After that, re-computing the view with new data is as simple as loading the saved

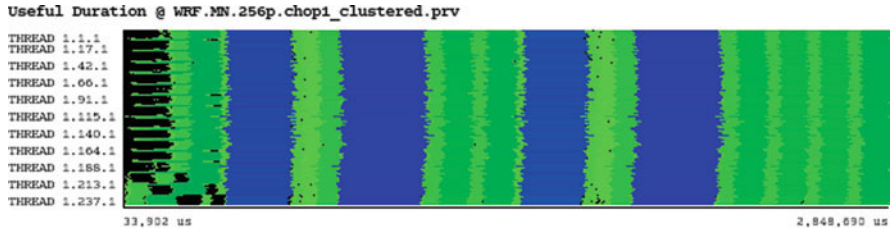


Fig. 4 Paraver timeline display

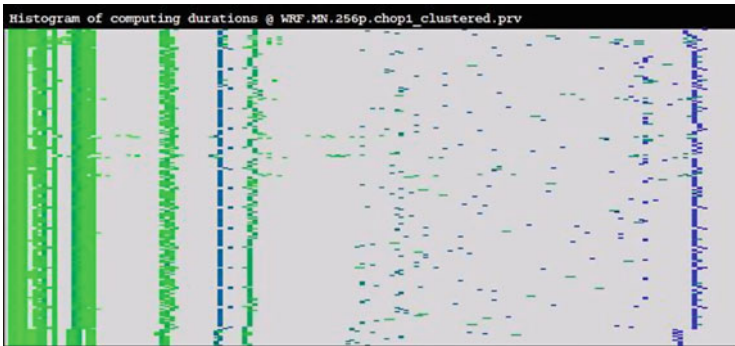


Fig. 5 Paraver histogram display

file. The tool has been demonstrated to be very useful for performance analysis studies, giving much more details about the application behaviour than most other performance tools.

Performance information in Paraver is presented with two main displays that provide qualitatively different types of information. The *timeline display* represents the behaviour of the application along time and processes, in a way that easily conveys to the user a general understanding of the application behaviour and simple identification of phases and patterns. The *statistics display* provides numerical analysis of the data that can be applied to any user-selected region, helping to draw conclusions on where and how to focus the optimisation effort. See Figs. 4 and 5 for an example of Paraver’s main displays.

Typical Questions Paraver Helps to Answer

- What is the parallelisation efficiency and the performance of communication?
- What are the differences that can be observed between two different executions?
- Does the behaviour of the application change over time?
- Are performance or workload variations the cause of load imbalances in computation?
- Which performance issues do the microprocessor’s hardware counters reflect?

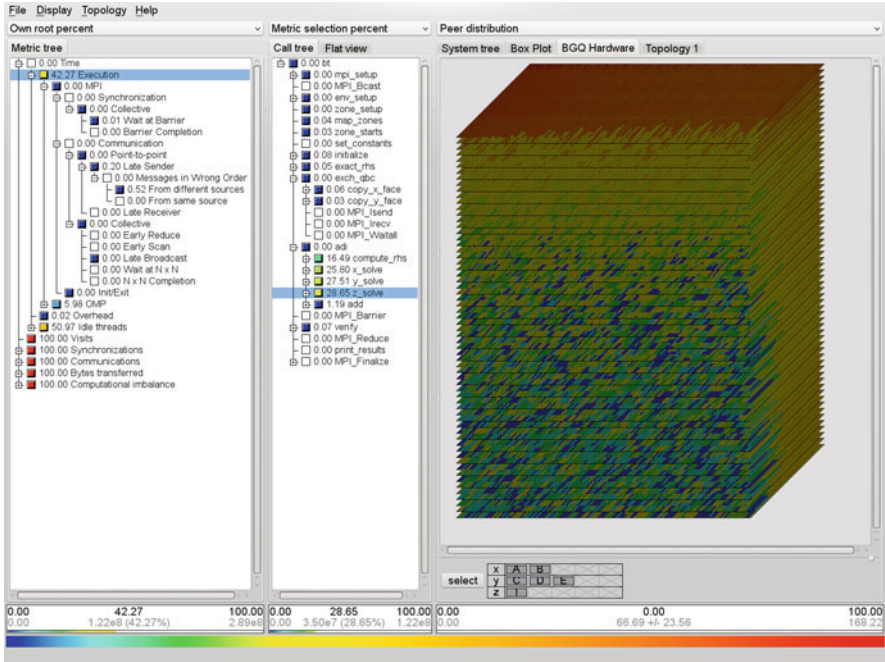


Fig. 6 Interactive exploration of performance behaviour in Scalasca along the dimensions performance metric (*left*), call tree (*middle*), and process topology (*right*). Screenshot shows result of a 524,288 threads run on the Jülich BlueGene/Q machine

2.4.3 Scalasca

Scalasca [2] is a free software tool that supports the performance optimisation of parallel programs by measuring and analysing their runtime behaviour. The tool has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XE, but is also well suited for small- and medium-scale HPC platforms. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes.

The user of Scalasca can choose between two different analysis modes: (i) performance overview on the call-path level via profiling and (ii) the analysis of wait-state formation via event tracing. Wait states often occur in the wake of load imbalance and are serious obstacles to achieving satisfactory performance. Performance-analysis results are presented to the user in an interactive explorer called Cube (Fig. 6) that allows the investigation of the performance behaviour on different levels of granularity along the dimensions performance problem, call path, and process. The software has been installed at numerous sites in the world and has been successfully used to optimise academic and industrial simulation codes.

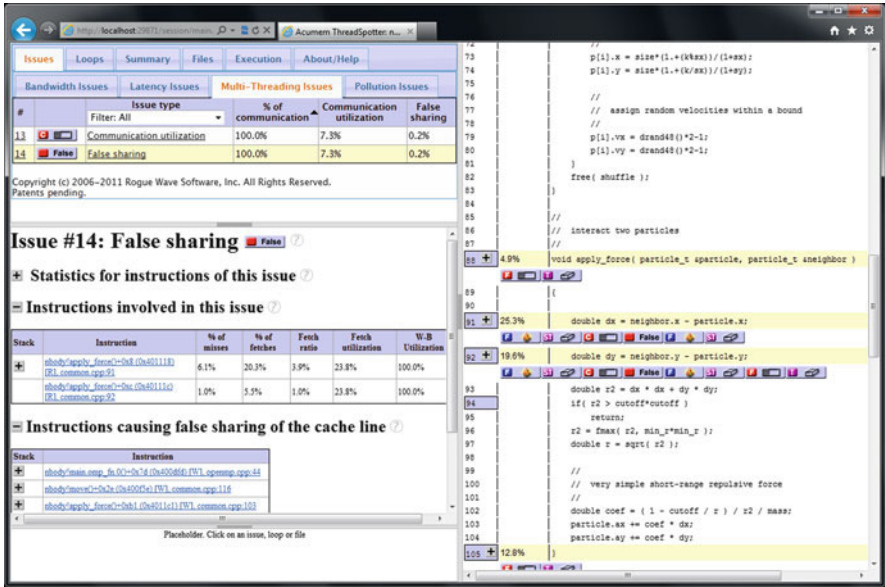


Fig. 7 Highlighting a “false sharing” situation. Top left part contains lists of problems. Lower left contains details, and annotated source code is to the right

Typical Questions Scalasca Helps to Answer

- Which call-paths in my program consume most of the time?
- Why is the time spent in communication or synchronisation higher than expected?
- Does my program suffer from load imbalance and why?

2.4.4 ThreadSpotter

ThreadSpotter [3] is a commercial tool that will help programmers optimise their programs with respect to architectural bottlenecks such as cache size and memory system bandwidth and point out inefficient communication modes between threads. Its scope is a single process, including both single-threaded as well as multi-threaded applications.

Some programming styles will exercise the memory system in suboptimal ways that can reduce performance drastically. Examples of these are failure to observe or exploit locality properties in code or data. Inappropriate communication through shared memory between threads may cause the coherence traffic to become a bottleneck.

ThreadSpotter explains the inefficiencies of observed memory access patterns on a high level in a graphical user interface (Fig. 7) and provides pointers to suggestions to optimise the code. It offers deep explanations on hardware level to back up the suggestions, educating the user as he uses the tool.

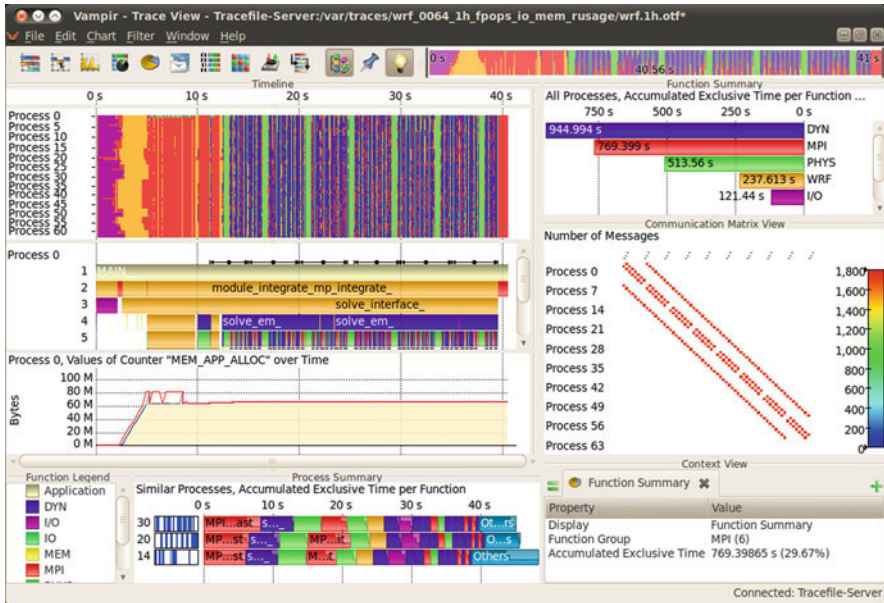


Fig. 8 Vampir GUI

Typical Questions ThreadSpotter Helps to Answer

- How does my program abuse the memory system and what can I do about it?
- Do the threads of my program exchange data with each other in an inefficient way?
- When adjusting my program, are the changes actually helping to minimise the footprint of the application?

2.4.5 Vampir

Vampir [4] is a graphical analysis framework that provides a large set of different chart representations of event-based performance data. These graphical displays, including timelines and statistics, can be used by developers to obtain a better understanding of their parallel program's inner working and to subsequently optimise it. See Fig. 8 for an impression of the Vampir GUI.

Vampir is designed to be an intuitive tool, with a GUI that enables developers to quickly display program behavior at any level of detail. Different timeline displays show application activities and communication along a time axis, which can be zoomed and scrolled. Statistical displays provide quantitative results for the currently selected time interval. Powerful zooming and scrolling along the timeline and process/thread axis allows pinpointing the causes of performance problems.

All displays have context-sensitive menus, which provide additional information and customisation options. Extensive filtering capabilities for processes, functions, messages or collective operations help to narrow down the information to the interesting spots. Vampir is based on Qt and is available for all major workstation operation systems as well as on most parallel production systems. The parallel version of Vampir, VampirServer, provides fast interactive analysis of ultra large data volumes.

Typical Questions Vampir Helps to Answer

- What happens in my application execution during a given time in a given process or thread?
- How do the communication patterns of my application execute on a real system?
- Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

2.5 Integration Among Performance Analysis Tools

Sharing the common measurement infrastructure Score-P and its data formats and providing conversion utilities if direct sharing is not possible, the performance tools in the HOPSA environment and workflow already make it easier to switch from higher-level analyses provided by tools like Scalasca to more in-depth analyses provided by tools like Paraver or Vampir. To simplify this transition even further, the HOPSA tools are integrated in various ways (see Fig. 9). With its automatic trace analysis, Scalasca locates call paths affected by wait states caused by load or communication imbalance. However, to find and fix these problems in a user application, it is in some cases necessary to understand the spatial and temporal context leading to the inefficiency, a step naturally supported by trace visualizers like Paraver or Vampir. To make this step easier, the Scalasca analysis remembers the worst instance for each of the performance problems it recognizes. Then, the Cube result browser can launch a trace browser and zoom the timeline into the interval of the trace that corresponds to the worst instance of the recognized performance problems.

In the future, the same mechanisms will be available for a more detailed visual exploration of the results of Scalasca's root cause analysis as well as for further analyzing call paths involving user functions that take too much execution time. For the latter, ThreadSpotter will be available to investigate their memory, cache and multi-threading behaviour. If a ThreadSpotter report is available for the same executable and dataset, Cube will allow launching detailed ThreadSpotter views for each call path where data from both tools is available.

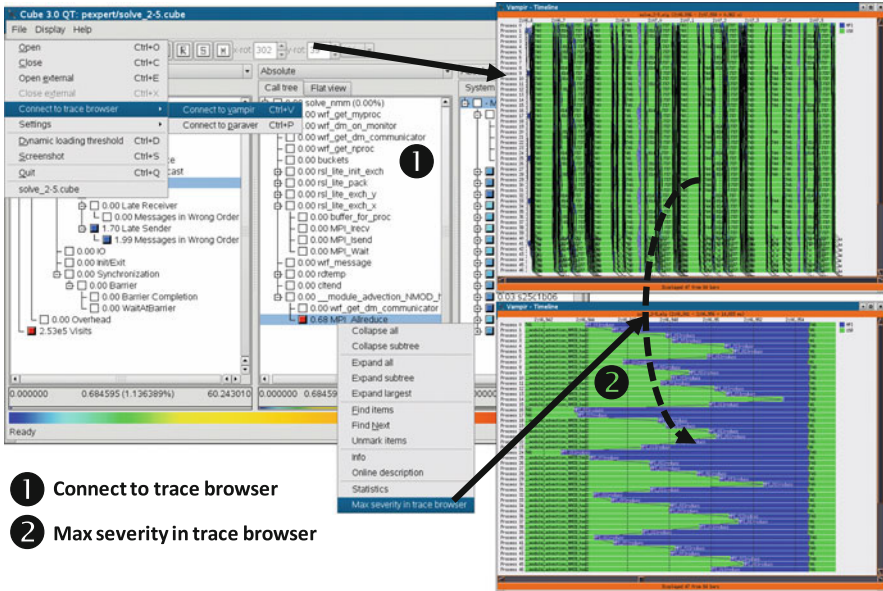


Fig. 9 Scalasca to Vampir or Paraver Trace browser integration. In a first step, when the user requests to connect to a trace browser, the selected visualizer is automatically started and the event trace, which was previously the basis of Scalasca’s trace analysis, is loaded. Now, in a second step, the user can request a timeline view of the worst instance of each performance bottleneck identified by Scalasca. The trace browser view automatically zooms to the right time interval. Now the user can use the full analysis power of these tools to investigate the context of the identified performance problem

2.6 Integration of System Data and Performance Analysis Tools

The Russian ClustrX.Watch management software provides node-level sensor information that can give additional insight for performance analysis of applications with respect to the specific system they are running on. This allows populating Paraver and Vampir traces with system information (the granularity will depend on the overhead to obtain the data) and to analyze them with respect to the system-wide performance.

The Russian LAPTA system data analysis and management software provides node-level sensor information that can give additional insight for performance analysis of applications with respect to the specific system they are running on. This allows populating Paraver and Vampir traces with system information collected by Clustrx, Ganglia, and other sources (the granularity will depend on the overhead to obtain the data) and to analyze them with respect to the system-wide performance. The system offers two different ways to access to the collected data:

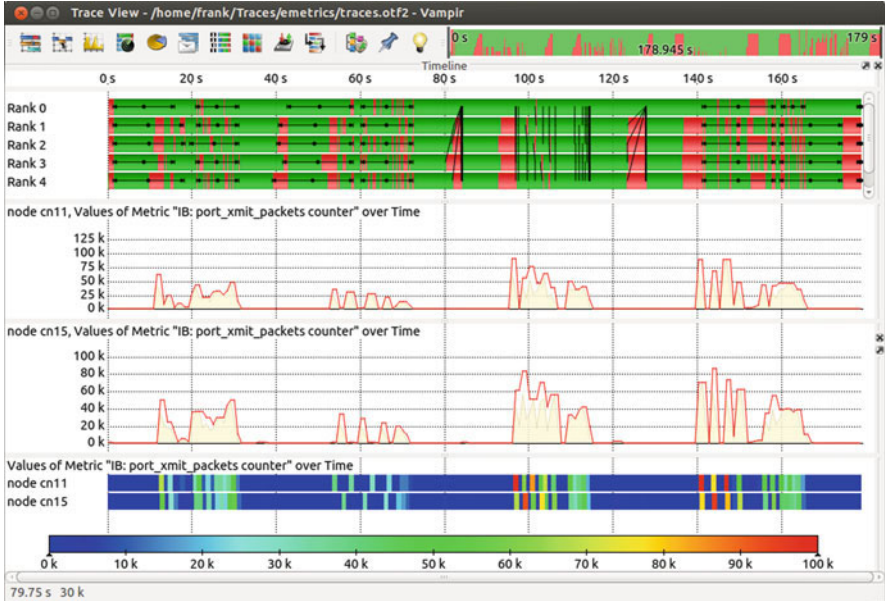


Fig. 10 Vampir screendump showing aligned system and application data

Historic information is stored with a given granularity for all the sensors and all the IP (nodes) on the system. The initial granularity was very coarse (1 min) and did not seem useful for the population of application trace files because there can be many different program phases in a 1 min interval. On the other hand, the circular buffer provides historical information with fine-grained detail (coarser or equal to 1 s depending on the sensor) for the last minutes (300 measurements). *Streamed information* can be requested for any range of sensors and IPs. The interface provides at least a value every 10 s unless there is a change greater than a 10 %. Currently, the finest available granularity is 1 s.

Both mechanisms use a connection through an HTTP protocol that in the case of the streamed data has to be refreshed periodically or dies after 5 min. We evaluated both alternatives to see their potential and identified possible drawbacks.

The Vampir team implemented a prototype Score-P adapter that enhances OTF2 traces at the end of the measurement. For evaluation, the benchmark code HPL was instrumented with Score-P. In addition to the application and MPI events, the trace was enhanced with HOPSA node-level metrics and per-process PAPI counters. Tested and working HOPSA sensors include node memory usage values and Infiniband packet counts. The evaluation shows that phases in the application clearly correlate to measured values of the node level sensors, e.g. heavy MPI communication to Infiniband packet counters (see Fig. 10). As a use case, this integration allows the user to analyze how the application utilizes network hardware of each node or how shared usage of network resources affects the application

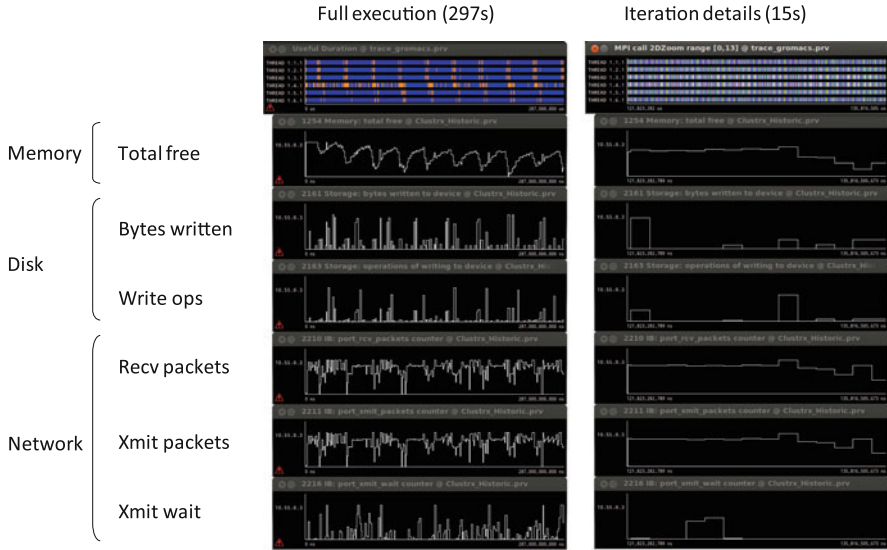


Fig. 11 Paraver screendumps showing aligned system and application data

execution. Currently the sensor values are available in 1 s granularity for the last 500 s and 1 min granularity before that.

The Paraver team experimented with Gromacs, a popular production code in life sciences, trying to correlate the sensors values to the activity of the application. As one can see in Fig. 11, on a higher level, it is possible to correlate system metrics with application program phases (left side of the picture). However, due to the limited resolution of the system metrics data, this is not possible on a more detailed level (see right side of the figure).

2.7 Opportunities for System Tuning

Several opportunities for system tuning arise from the availability of historic performance data collected by LWM². First, data on individual system nodes along an extended period of time in comparison to other nodes can be analysed to spot anomalies and detect deficient components. Second, data on the entire workload can be used to improve the understanding of the workload requirements and configure the system accordingly. The insights obtained may guide the evolution of the system and influence future procurement decision. Finally, knowledge of the resource requirements of individual jobs offers the chance to develop resource-aware scheduling algorithms that avoid oversubscription of shared resources such as the file system or the network.

3 Conclusion

The HOPSA project creates an integrated diagnostic infrastructure for combined application and system tuning. Starting from system-wide basic performance screening of individual jobs, an automated workflow routes findings on potential bottlenecks either to application developers or system administrators with recommendations on how to identify their root cause using more powerful diagnostics. This document specifies the performance analysis workflow that connects the different steps. At the same time, it provides an impression of the overall vision behind the project. The high-level description is intended to make it readable also for non-tool experts.

Although the specification is based on long experience with HPC application developers and how they tend to use performance tools, it is a blueprint that needs to be validated in practice. This validation is planned for the last quarter of the project at Moscow State University, once all the components are in place and, in particular, LWM² has been fully completed, tested, and integrated into the overall environment. During this validation process, some of the details presented in this document may change and ultimately result in a new revision. We expect though that all major elements will be retained.

Beyond the lifetime of the project, the HOPSA infrastructure is supposed to collect large amounts of valuable data on the performance of individual applications as well as the system workload as a whole. It will be of interest in three ways: to tune individual applications, to tune the system for a given workload, and finally to observe the evolution of this workload over time. The latter will allow the effectiveness of our strategy to be studied. An open research issue to be tackled on the way will be the reliable tracking of individual applications, which may change over time, across jobs based on the collected data. In this way, it will become possible to document the performance history of code projects and demonstrate the effects of our tool environment over time.

Acknowledgements HOPSA is a coordinated twin project funded under FP7-ICT-2011-EU-Russia grant number FP7-277463 and Russian Ministry of Education and Science contract number 07.514.12.4001. The authors also would like to thank our colleagues working with us on this project: Andrew Adinetz, Daniel Becker, Peter Bryzgalov, Jens Domke, Markus Geimer, Juan Gonzalez, André Grötzsch, Thomas Ilsche, Germán Llort, Christopher Schleiden, Konstantin Stefanov, Zoltán Szabenyi, Igor Zacharov, Pavel Saviankou, Igor Ustinov, Vadim Voevodin, and Sergey Zhumatiy as well as the Paraver, Scalasca, and Vampir teams in general.

References

1. J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris, DiP: A parallel program development environment. in: Proc. of the 2nd International Euro-Par Conference, Lyon, France, Springer, 1996.

2. M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, B. Mohr: The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.
3. E. Berg, E. Hagersten: StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In: *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, Austin, Texas, USA, March 2004.
4. W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.
5. D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A.D. Malony, W.E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S.S. Shende, M. Wagner, B. Wesarg, F. Wolf: Score-P: A Unified Performance Measurement System for Petascale Applications. In: *Competence in High Performance Computing 2010 (CiHPC)*, pp. 85–97. Gauß-Allianz, Springer (2012).
6. M. Gerndt and M. Ott. Automatic Performance Analysis with Periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.
7. S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006. SAGE Publications.
8. M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, B. J. N. Wylie: Further improving the scalability of the Scalasca toolset. In: *Proceedings of PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable tools for High Performance Computing*, Reykjavik, Iceland, June 6–9 2010, volume 7134 of *Lecture Notes in Computer Science*, pages 463–474, Springer, 2012.
9. D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, F. Wolf: Open Trace Format 2 – The Next Generation of Scalable Trace Formats and Support Libraries. In: *Proceedings of the International Conference on Parallel Computing (ParCo)*, Ghent, Belgium, 2011, volume 22 of *Advances in Parallel Computing*, pages 481–490, IOS Press, 2012.
10. A. Knüpfer, R. Brendel, H. Brunst, H. Mix, W. E. Nagel: Introducing the Open Trace Format (OTF), In: Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, Jack Dongarra (Eds): *Computational Science – ICCS 2006: 6th International Conference*, Reading, UK, May 28–31, 2006, *Proceedings, Part II*, Springer Verlag, ISBN: 3-540-34381-4, pages 526–533, Vol. 3992, 2006.
11. F. Wolf, B. Mohr: EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, 2004.
12. H. Servat Gelabert, G. Llort Sanchez, J. Gimenez, and J. Labarta. Detailed performance analysis using coarse grain sampling. In: *Euro-Par 2009 – Parallel Processing Workshops*, Delft, The Netherlands, August 2009, volume 6043 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2010.
13. T-Platforms, Moscow, Russia. Clustrx HPC Software. <http://www.t-platforms.com/products/software/clustrxproductfamily.html>, last accessed September 2012.
14. A.V. Adinets, P.A. Bryzgalov, Vad.V. Voevodin, S.A. Zhumatiy, D.A. Nikitenko. About one approach to monitoring, analysis and visualization of jobs on cluster system (In Russian). In: *Numerical Methods and Programming*, 2011, vol. 12, Pp. 90–93
15. Z. Szebenyi, F. Wolf, B. J.N. Wylie. Space-Efficient Time-Series Call-Path Profiling of Parallel Applications. In: *Proc. of the ACM/IEEE Conference on Supercomputing (SC09)*, Portland, OR, USA, ACM, 2009.

Part IV
Performance Data Visualization

Visualizing More Performance Data Than What Fits on Your Screen

Lucas M. Schnorr and Arnaud Legrand

Abstract High performance applications are composed of many processes that are executed in large-scale systems with possibly millions of computing units. A possible way to conduct a performance analysis of such applications is to register in trace files the behavior of all processes belonging to the same application. The large number of processes and the very detailed behavior that we can record about them lead to a trace size explosion both in space and time dimensions. The performance visualization of such data is very challenging because of the quantities involved and the limited screen space available to draw them all. If the amount of data is not properly treated for visualization, the analysis may give the wrong idea about the behavior registered in the traces. This paper is twofold: first, it details data aggregation techniques that are fully configurable by the user to control the level of details in both space and time dimensions; second, it presents two visualization techniques that take advantage of the aggregated data to scale. These features are part of the **VIVA** open-source tool and framework, which is also briefly described in this paper.

1 Introduction

High performance computing systems today are composed of several thousands and sometimes millions of cores. The fastest parallel machine as defined by the Top500 [5] in June 2012 has more than 1.5 million cores. Parallel machines with billions of cores are expected in the way towards exascale computing. Parallel applications running on these large-scale platforms are therefore composed of many processes and threads that together explore the extreme concurrency available to solve problems in a variety of domains.

L.M. Schnorr (✉) · A. Legrand
INRIA MESCAL Research Team, CNRS LIG Laboratory, Grenoble, France
e-mail: lucas.schnorr@imag.fr

A possible way to conduct a performance analysis of such applications is by recording in trace files the behavior of all processes of the same application. The behavior of each process can be defined by a succession of timestamped events that register the important parts of the application, as defined by the analyst. With low-intrusion tracing techniques, the events can be as frequent as one event per nanosecond. Because of the quantity of processes and the amount of details we can collect about each of them, traces become very large, at least in the space and time dimensions.

Despite the technical challenge of managing possibly millions of very large traces that are scattered in the platform, another aspect is how to extract useful information from them. A possibility is to use performance visualization techniques to visually represent the behavior described in the traces. Several techniques exist, but the most prominent is the traditional space/time view – also known as timeline view or Gantt-like chart, shared by many trace visualization tools [3, 4, 11, 12, 18]. Other techniques such as classical histograms, communication matrices, and kivi diagrams also appear in the literature [3, 7]. Independently of which visualization technique is chosen to represent traces, the task is commonly very challenging and complex because of the quantities involved – number of processes, amount of details – and the limited screen space available to represent all the information.

Most of performance visualization tools today have implicit assumptions about how traces should be represented. Let us consider a very detailed trace of an application that executes for several days. When visually representing an overview of that trace on the screen, a pixel might represent many hours of execution. Some tools [3] take the more common application state on that time interval and choose its color to draw the pixel. Although this works well to scale the visualization, it might not work for all scenarios if, for example, the state chosen to draw the pixel is not the one that is the focus of the analysis. This also has the potential disadvantage of dropping important information when two different states have similar presence in the time interval of a given pixel. Some other tools [4] simply rely on graphical rendering – thereby ignoring completely the problem. This second case might lead to an extreme negative scenario where the understanding of the trace depends on the graphics card and library used to draw its representation. Generally, these implicit assumptions mixed with large-scale traces may give the wrong idea about the behavior registered in the traces. We argue that all assumptions done when visualizing trace data should be taken by the analyst, and not built-in in trace visualization tools.

This paper addresses the issue of visualizing more performance data than what could fit on the available screen space. Instead of directly drawing the trace events with a visualization technique, our approach transforms raw traces into aggregated traces – according to analyst needs – and then visually represent the transformed traces. Therefore, this paper is twofold: first, it details data aggregation techniques that transforms the raw traces and are fully configurable by the user to control the level of details in both space and time dimensions; second, it presents two visualization techniques that take advantage of the aggregated data to scale.

The paper is organized as follows. Section 2 presents an extended discussion about the problem of scalable performance visualization and a classification of tools considering implicit or explicit trace aggregation. Section 3 describes the multi-scale aggregation techniques applied to transform trace data before visualization. Section 4 presents two interactive visualization techniques designed to explore aggregated traces: the squarified treemap and the hierarchical graph views. Section 5 presents the **VIVA** visualization tool, which implements the data aggregation algorithms and the visualization techniques. Finally, Sect. 6 draws a conclusion and future work.

2 Motivation and Discussion

The performance analysis of parallel applications must deal with the problem of trace size. This increase in size is present in different forms. The more common situations are the following: spatial size increase, when the application is large with many processes; temporal, when many details for each process must be interpreted – even if there is only a few processes to be analyzed; or both. The use of visualization to analyze large-scale traces is especially influenced by this increase in size, since traditional visualization have scalability problems.

We take as example the Sweep3D benchmark [15] to illustrate the scalability issue in trace visualization. The experiment is configured as follows. The MPI application is executed on 16 nodes of the Griffon cluster, part of the Grid'5000 [2] platform. The TAU library [22] traces all the MPI operations, resulting in traces that are merged into a single file. This single file is converted to the Paje File Format [21] using the `tau2paje`¹ tool and finally exported to CSV (comma-separated values) using the `pj_dump`² tool. A combination of R [8] and the `ggplot2` package [23] is used to plot the resulting CSV file to a graphical representation based on the space/time view into a vector file.

Figure 1 shows two visualization of this vector file: one as seen by the Gnome Evince (left); the other by Acroread (right). As traditional space/time views, processes are listed in the vertical axis while the horizontal axis depicts the behavior of each processes along time. The different colors (or gray scales) represent different MPI operations. We can see that while both tools visualize the same file, the visualization is completely different depending on the viewer chosen, misguiding the analysis. If only Evince is used, the analysts might conclude that there are very few states denoted by the red color; if Acroread is used, it is hard to take the exact same conclusion, since we can see that red states are quite present for all processes.

The extreme negative scenario shown in Fig. 1 clearly illustrates the problem when visualizing too much data in the same screen without care. The trace obtained

¹Part of Akypuera toolset, available at <https://github.com/schnorr/akypuera>

²Part of PajeNG, available at <https://github.com/schnorr/pajeng>

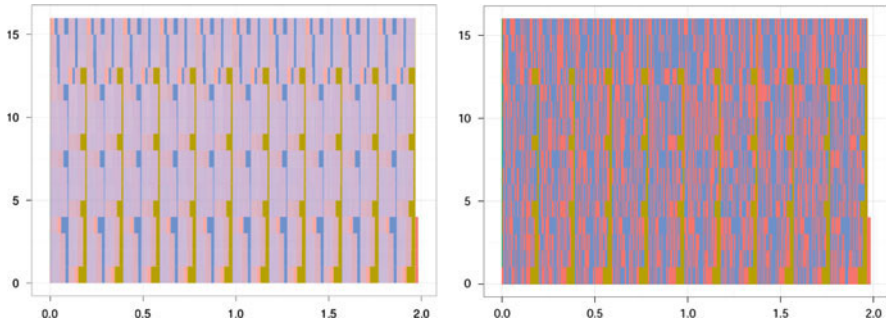


Fig. 1 Performance visualization of the Sweep3D MPI application with 16 processes – the vertical axis lists the processes, while the horizontal axis depicts time – as visualized with two PDF viewers: Gnome Evince (*left*) and Acroread (*right*): depending on the viewer, the analyst takes different conclusions about the performance behavior

from Sweep3D, even for this small run of 2 s, has many events in time – more events than the horizontal resolution is capable to contain. Since the vector file contains all events – most of them in the microseconds scale – many states have to be drawn in the same screen pixel. The color choice for a given pixel is taken by the renderer and the anti-aliasing algorithm, which is different depending on the visualization tool and does not make any sense for such Gantt-charts. That is why we get different views depending on the tool.

This issue about implicit data aggregation also appears on performance visualization tools which are specific to trace analysis. Considering space/time views, the scalability problem can appear in both dimensions: in the horizontal dimension when there is too much detail about each process (as in Fig. 1), in the vertical dimension when the application is composed by many thousands processes. Many data reduction techniques exist in several forms [1, 10, 13, 14, 16, 17, 19] to try to reduce the amount of data that is going to be visualized. They might be broadly classified in two groups: selection and aggregation. The first group – based on selection – encompasses all solutions that select a subset of the data according to some criteria, which on its turn can be either fully based on a direct choice made by the analyst or not. Clustering algorithms considering behavior summaries as similarity pattern are an example of automatic selection. The second group – based on aggregation – acts upon the data, transforming into another kind of data whose intent is to represent the aggregated entities.

There is always some kind of aggregation considering performance visualization of large traces, although it is often implicit and uncontrolled. We explain now the three possibilities that appear on performance visualization tools regarding trace aggregation:

Explicit Data Aggregation. The analyst keeps control of the aggregation operators and the data neighborhood that is going to be aggregated. Moreover, the visualization tool gives some feedback to let the analyst know that a data aggregation takes place and is being used in the visualization.

Implicit Data Aggregation. There is no way to distinguish in the visualization something that has been aggregated from raw traces. The Fig. 1 is an example of implicit data aggregation where the renderer takes the decision and the analyst is unaware about what is being visualized, if it is aggregated or not.

Forbidden Data Aggregation. The performance visualization tools forbids data aggregation at some level, commonly to avoid an implicit data aggregation that could mislead the analysis. The analyst is aware of that since the tool blocks, for example, an interactive operation, such as zoom out to get an overview.

Performance visualizations tools are sometimes present in more than one of these data aggregation categories. Paje's space/time view [12], for example, has explicit data aggregation in the temporal axis – the user realizes that an aggregation took place since aggregated states are slashed in the visualization, while forbidding data aggregation on the spatial axis – minimum size for each resource is 1 pixel. Another tool that falls in more than one category depending on the represented data is Vampir's master timeline [3]. For the temporal axis, it has implicit and explicit data aggregation. Implicit aggregation happens for function representations (colored horizontal bars): the tool choose the color of each screen pixel according to a histogram and selecting the more frequent function on that interval of time [6]. The user has no visual feedback if an aggregation takes place or not. Explicit aggregation appears in Vampir for the communication representation (arrows): a special message burst symbol [6] is used to tell the user that a zoom is necessary to get further details on the messages. Vite's timeline [4] has implicit data aggregation on spatial and temporal dimensions, since it draws everything no matter the size of the screen space dedicated to the visualization. Triva [20] has explicit data aggregation for spatial and temporal dimensions, but it uses different visualization techniques, such as the aggregated treemap, to visualize performance data.

Thus, the best scenario for a performance visualization tool is to provide pure explicit data aggregation. This choice enables users to realize what is happening with eventual trace transformations through a visual feedback in the visualization technique. This is commonly present on traditional statistical plots, where the amount of data is reduced with statistical mechanisms which are fully controlled by the analyst. Therefore, the goal on performance visualization is to propose explicit data aggregation which enables visualizations that are richer than classical statistical plots and, at the same time, safer than traditional trace visualization. Next section details our approach to explicit data aggregation for visualization.

3 Multi-scale Trace Aggregation for Visualization

We briefly detail how data aggregation is formally defined in our approach. Let us denote by \mathcal{R} the set of observed entities – which could be the set of threads, processes, machines, processors, or cores – and by \mathcal{T} the observation period. Assume we have measured a given quantity ρ – which is a tracing metric – on each resource:

$$\rho : \begin{cases} \mathcal{R} \times \mathcal{T} & \rightarrow \mathbb{R} \\ (r, t) & \mapsto \rho(r, t) \end{cases}$$

In our context, $\rho(r, t)$ could for example represent the CPU availability of resource r at time t . It could also represent the execution of a function by a thread r at time t . In most performance analysis situations, we have to depict several of such functions at once to investigate their correlation.

As we have have illustrated in Fig. 1, ρ is generally complex and difficult to represent. Studying it through multiple evaluations of $\rho(r, t)$ for many values of r and t is very tedious and one often miss important features of ρ doing so. This is also one of the reasons explaining the previous visualization artifact seen in the previous section.

Assume we have a way to define a neighborhood $N_{\Gamma, \Delta}(r, t)$ of (r, t) , where Γ represents the size of the spatial neighborhood and Δ represents the size of the temporal neighborhood. In practice, we could for example choose $N_{\Gamma, \Delta}(r, t) = [r - \Gamma/2, r + \Gamma/2] \times [t - \Delta/2, t + \Delta/2]$, assuming our resources have been ordered. Then, we can define an approximation $F_{\Gamma, \Delta}$ of ρ at the scale Γ and Δ as:

$$F_{\Gamma, \Delta} : \begin{cases} \mathcal{R} \times \mathcal{T} & \rightarrow \mathbb{R} \\ (r, t) & \mapsto \iint_{N_{\Gamma, \Delta}(r, t)} \rho(r', t') . dr' . dt' \end{cases} \quad (1)$$

Intuitively, this function averages the behavior of ρ over a given neighborhood of size Γ and Δ . For example a crude view of the system is given by considering the whole system as the spatial neighborhood and the whole timeline as the temporal neighborhood. Since Δ can be continuously adjusted, we can temporally zoom in and consider the behavior of the system at any time scale. Once this new time scale has been decided, we can observe the whole timeline by shifting time and considering different time intervals.

The analyst have to be careful about the conclusions that are taken during an analysis based on aggregated data. The nature of the data aggregation technique as presented here leads to the attenuation of behaviors registered in scales smaller than the one used to aggregate the data. For example, if a temporal aggregation is configured to integrate data using a 2 s interval, all the details smaller than 2 s are attenuated by the integration.

At the same time, the analyst needs to be aware that, although some information is lost, such aggregation generally lead to better visualization which can allow for the detection of anomalies that could pass undetected without data aggregation. Our approach deals with such questions by letting the analyst choose freely which scale is used to aggregate trace data. That's why this approach can be considered a pure explicit data aggregation method, where traces are transformed before being visualized. Next section details two visualization techniques for data that is aggregated following this method.

4 Visualization Techniques

Previous sections have shown the importance of explicit data aggregation algorithms for performance analysis through visualization. We present now two visualization techniques that have interactive mechanisms to deal with different aggregation levels, at the same time giving the user a feedback when the information in traces is aggregated: the squarified treemap and the hierarchical graph view. Their common characteristic is the lack of a timeline, as the one used on Gantt-charts, because the trace information is temporally aggregated. This lack of timeline also enables the use of both screen dimensions to draw observed entities and thus display more information.

4.1 Squarified Treemap View

The treemap technique [9] represents an annotated hierarchical structure on the screen using a space-filling approach. The recursive technique starts on the root of the tree, dividing the screen space among its children depending on their values. The screen surface each node occupies is proportional to its value. This mechanism allows an easy comparison of the characteristic of the different nodes of the structure, even in large-scale scenarios.

The squarified treemap view [19], in our approach, is capable to represent performance data that is temporally and spatially aggregated: one screenshot represents therefore the application behavior in a time slice and spatial cut. These levels of detail are configurable by the user and can be changed interactively during the analysis. However, spatial aggregation is only calculated when the available performance data is hierarchically organized (threads grouped by process, process by machine, and so on). The hierarchy is therefore used as spatial neighborhood criteria (as presented in Sect. 3). Whenever the time slice or the spatial cut is changed, a new squarified treemap is calculated and drawn. Three interactive transitions of this kind are depicted in Fig. 2.

Figure 2 shows four treemaps of the same trace at a given time interval configured by the analyst. The top-right treemap of the figure shows, for instance, the Executing and Blocked state for the six clusters of this synthetic example (as indicated by the rounded dashed rectangles). We can see the three clusters per site and the two sites. The values for the states for a cluster are calculated by the aggregation algorithm considering the Blocked and Executing states for the machines belonging to that cluster.

The main advantage of the original treemap representation over traditional space/time views is its visual scalability. The technique is, as any other visualization technique, also limited by the screen size. The treemap A of Fig. 3 shows the representation of synthetic trace with 100 thousands processors. We can see that with that large number of processors, the visualization suffers from implicit data aggregation

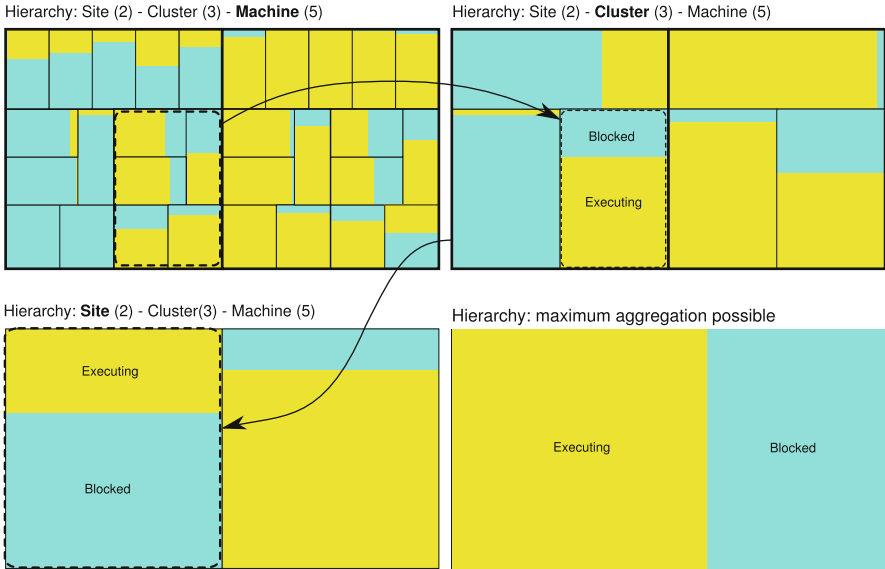


Fig. 2 Four treemaps to show the per-level aggregation of *Blocked* and *Executing* states

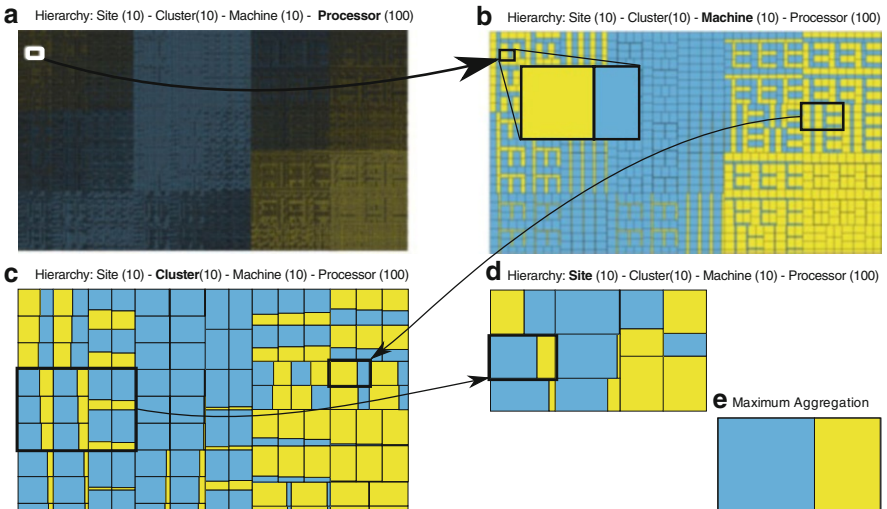


Fig. 3 Normal (a) and 4 aggregated treemap visualizations (b-e) of 2 states for 100 thousand processors (based on synthetic trace)

during graphical rendering – there is a lack of pixels to draw all processors in a way that we could extract useful information from them. With the aggregated treemap view, the scalability limits of the representation are pushed forward since it is no longer necessary to view the behavior of every single monitored entity

in the analysis. Treemaps B–E of Fig. 3 depicts aggregated behavior of groups of processes. The scalability limit lies on how deep the trace hierarchy is for a given trace. If there is not enough levels on this hierarchy, they can be created by grouping nodes according to some analysis criteria.

The aggregated treemap view, as presented here, is a complementary technique in performance visualization. Although it offers a very scalable way to represent aggregated trace data, it has some disadvantages. It lacks, for example, support for a causal order analysis, which is obvious when using space/time views. By using treemaps as a complementary view, it is possible to get the best of both techniques. Another problem of treemaps (and space/time views) is that it lacks topological information, which might be crucial when analyzing performance behavior considering the network bandwidth limitations.

Next section details the second visualization technique based on hierarchical graphs to offer a topological analysis of parallel applications behavior.

4.2 Hierarchical Graph View

The traditional timeline view is expected by most of users of high performance computing, as can be observed by the number of visualization tools that implement it. Although useful to show the causal order in program behavior, timeline views lack topological information. Contrasting behavior with topological data is sometimes crucial for the comprehension of application behavior, especially because it allows to find the origin of contentions and better adapt the application to these constraints. The hierarchical graph view, presented in this section, enables an analysis that correlates all program behavior with a graph. The hierarchical aspect is used to tackle the scalability issues of graphs. We detail here two basic aspects of the hierarchical graph view: how temporal-aggregated metrics are mapped to the graph; and how the spatial-aggregated data is used to achieve visualization scalability.

The mapping from the trace to the graph works as follows. All monitored entities are mapped to nodes, while edges indicate a connection between two monitored entities. Monitored entities can be physical components of the system (machines, network links) but also the applications components (threads, processes). They are differentiated in the representation through different geometric shapes, while their attributes (size, color, filling) are mapped from the trace metrics associated to each monitored entity. Generally, all geometric shapes and properties can be somehow configured depending on trace metrics.

Figure 4 shows an example of six variables (two per resource) showing resource availability (normal line) and consumption (dotted line) and three graph mappings depending on the selected time intervals (A, B and C). Hosts are mapped to squares, while links are mapped to diamonds. The size of the geometric forms are equivalent to the time-integrated resource availability, while the filling comes from resource consumption.

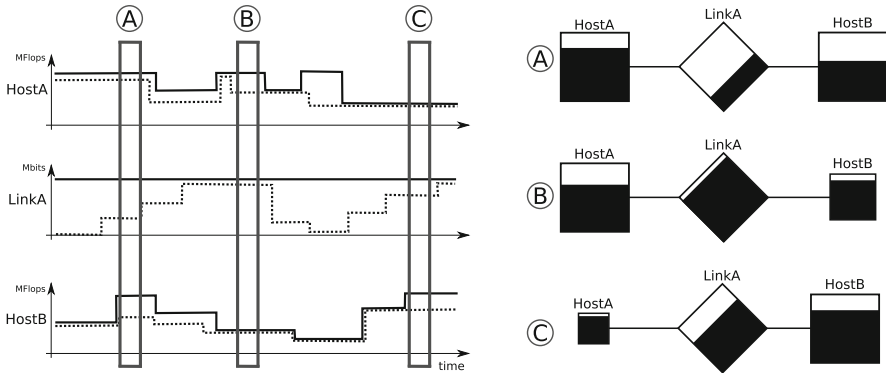


Fig. 4 Mapping temporally-integrated trace metrics (*left*) to three graph representation (*right*) depending on the selected time intervals, considering that hosts are mapped to *squares*, links to *diamonds*

The spatial aggregation algorithm presented in Sect. 3 also influences how the graph representation in our approach. Figure 5 shows an example that illustrates how the spatial aggregation affects the representation. As before, hosts are represented by squares filled by their utilization, links by diamonds, also filled according to their utilization. We consider for this example that the time-slice is already fixed. In the left of the figure, *GroupA* indicates the first neighborhood taken into account during the first spatial aggregation. All data within this group is space aggregated following the Eq. 1. The resulting representation is depicted in the center of the figure, surrounded by the dashed gray line: it combines a square, representing all hosts, and a diamond, representing all links (in this case there is only one). The properties of these two geometric shapes are calculated according to the space-aggregated values of the traces, considering all the entities within the group used to do the aggregation. The example ends with a second spatial aggregation, considering the whole *GroupB*, with all monitored entities. As of result, in the right of the figure, there are only one square and one diamond representing all the hosts and all the links of the initial representation.

Spatial aggregation as defined in Sect. 3 plays a major role in the scalability of the hierarchical graph view. Graphs are by nature non-scalable, as we increase the number of nodes, the harder it gets to analyze and understand patterns. The possibility to interactively aggregate a portion of the graph, while keeping its general behavior through the use of aggregated values, enables an analysis of large-scale scenarios. Figure 6 shows an example of this change of spatial detail with the Grid5000 platform and its network topology. Each graph node represents a machine, and its size is equivalent to the computing power of the machine. The leftmost graph depicts all the 2,170 computing nodes. Subsequent plots represent higher-level cuts on the hierarchy defining aggregated graphs – in the cluster and site levels. The rightmost graph represents the full aggregation considering the computing power of all hosts (on its left) and the bandwidth of all network links (on its right).

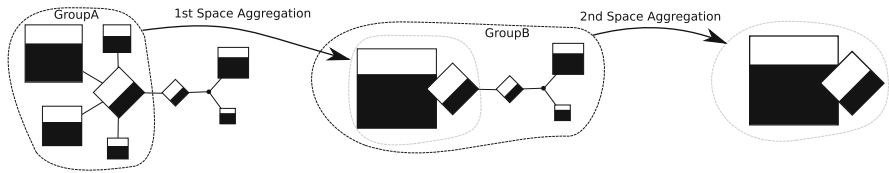


Fig. 5 Two spatial-aggregation operations and how they affect the topology-based representation

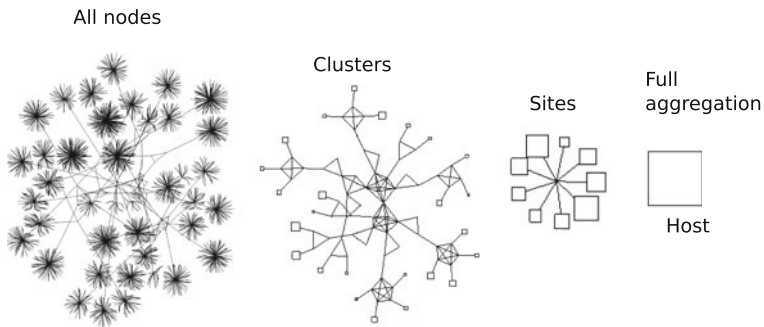


Fig. 6 Grid5000 network topology with 2,170 computing nodes, depicted in 4 different aggregation levels of the hierarchical graph view: resource capacity is used to draw the size of geometric shapes

5 The Viva Visualization Tool

The **VIVA**³ visualization tool implements the multi-scale aggregation algorithm (Sect. 3) and the two visualization techniques of the previous section. It is implemented in C++ using the Qt libraries as user interface. The tool also uses the **PAJENG**⁴ framework to deal with traces. This framework encloses all the basic building blocks such as reading trace files, simulating their behavior and offering access to trace data through the Paje protocol.

6 Conclusion

The performance visualization of traces might be a complex task because of the size of parallel applications and the amount of detail collected for each process. Besides dealing with the technical issues of large-scale traces, there is the problem of how to keep a given visualization technique useful, capable of detecting performance

³ Available at <http://github.com/schnorr/viva/>

⁴ Available at <http://github.com/schnorr/pajeng/>

problems, on scale. We have shown that if traces are represented without care, the visualization might misguide the analysis.

This paper addresses the issue of visualizing more performance data than what could fit on the available screen space. Instead of directly drawing the trace events, our approach is to use a multi-scale data aggregation algorithm to transform raw events into aggregated traces. These transformed traces are then visualized by two visualization techniques especially tailored to handle aggregated data: the squarified treemap and the hierarchical graph view. The squarified treemap view enables the comparison of processes behavior by mapping per-process trace metrics to screen space. The hierarchical graph view enables the correlation of application behavior with the network topology by mapping trace metrics to geometrical attributes of a graph representation. Both techniques can scale since they expect spatial-aggregated data as input.

Relying on data aggregation is fundamental to scale the visualization techniques. However, it also has some disadvantages. As defined in Sect. 3, our multi-scale aggregation algorithm averages behavior in space and time dimensions. Depending on the situation, this average might smooth or even completely hide a certain behavior from the analysis. In addition, it is likely that space and time scales should be linked – a zoom in/out in one dimension implicates a zoom in/out in the another one – since it is meaningless to visualize the behavior of several thousands processes in a 1- μ s time interval. Currently, we transfer to the analyst the responsibility to choose a space/time neighborhood in order to mitigate these issues.

Beyond this space/time rescaling issue, we can identify at least three other interesting research topics. The first one is to revisit space/time representations (also known as timeline views) to draw aggregated data instead of raw events, diminishing the problems detailed on Sect. 2 and especially in Fig. 1. The second topic consists in studying new aggregation techniques and operators that take into account the uncertainty of events in the temporal dimension, in particular to deal with large-scale scenarios and in the presence of time synchronization issues. And finally, since aggregations smooth and may lead to potential loss of information, being able to quantify such loss could be used to provide feedback to the analyst. This feedback would indicate where particular attention is necessary due to aggressive aggregation.

Acknowledgements This work is partially funded by the french SONGS project (ANR-11-INFRA-13) of the *Agence Nationale de la Recherche* (ANR). We thank Augustin Degomme for providing the sweep3D MPI traces. We also thank the organizers of the *6th International Parallel Tools Workshop* for the invitation.

References

1. Aguilera, G., Teller, P., Taufer, M., Wolf, F.: A systematic multi-step methodology for performance analysis of communication traces of distributed applications based on hierarchical clustering. In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, p. 8 pp. (2006). DOI 10.1109/IPDPS.2006.1639645

2. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lantéri, S., Leduc, J., Melab, N., R. Namyst, G.M., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications* **20**(4), 481–494 (2006)
3. Brunst, H., Hackenberg, D., Juckeland, G., Rohling, H.: Comprehensive performance tracking with vampir 7. In: M.S. Müller, M.M. Resch, A. Schulz, W.E. Nagel (eds.) *Tools for High Performance Computing 2009*, pp. 17–29. Springer Berlin Heidelberg (2010). DOI http://dx.doi.org/10.1007/978-3-642-11261-4_2
4. Coulomb, K., Faverge, M., Jazeix, J., Lagrasse, O., Marcoueille, J., Noisette, P., Redondy, A., Vuchener, C.: Visual trace explorer (vite) (2009)
5. Dongarra, J., Meuer, H., Strohmaier, E.: Top500 supercomputer sites. *Supercomputer* **13**, 89–111 (1997)
6. Gmbh, G.T.: Vampir 7 User Manual. Technische Universität Dresden, Blasewitzer Str. 43, 01307 Dresden, Germany, 2011-11-11 / vampir 7.5 edn. (2011)
7. Heath, M., Etheridge, J.: Visualizing the performance of parallel programs. *IEEE software* **8**(5), 29–39 (1991)
8. Ihaka, R., Gentleman, R.: R: A language for data analysis and graphics. *Journal of computational and graphical statistics* pp. 299–314 (1996)
9. Johnson, B., Shneiderman, B.: Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: *Proceedings of the IEEE Conference on Visualization*, pp. 284–291. IEEE Computer Society Press Los Alamitos, CA, USA (1991). DOI 10.1109/VI-SUAL.1991.175815
10. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.K.: Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers* **55**, 769–782 (2006). DOI <http://doi.ieeecomputersociety.org/10.1109/TC.2006.85>
11. Kalé, L.V., Zheng, G., Lee, C.W., Kumar, S.: Scaling applications to massively parallel machines using projections performance analysis tool. *Future Generation Comp. Syst.* **22**(3), 347–358 (2006)
12. de Kergommeaux, J.C., de Oliveira Stein, B., Bernard, P.E.: Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing* **26**(10), 1253–1274 (2000)
13. Knupfer, A., Nagel, W.: Construction and compression of complete call graphs for post-mortem program trace analysis. In: *Parallel Processing, 2005. ICPP 2005. International Conference on*, pp. 165–172 (2005). DOI 10.1109/ICPP.2005.28
14. Lee, C., Mendes, C., Kalé, L.: Towards scalable performance analysis and visualization through data reduction. In: *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1–8. IEEE (2008). DOI <http://dx.doi.org/10.1109/IPDPS.2008.4536187>
15. Lubeck, O., Lang, M., Srinivasan, R., Johnson, G.: Implementation and performance modeling of deterministic particle transport (sweep3d) on the ibm cell/b.e. *Scientific Programming* **17** (2009)
16. Mohror, K., Karavanic, K.L.: Evaluating similarity-based trace reduction techniques for scalable performance analysis. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, pp. 55:1–55:12. ACM, New York, NY, USA (2009). DOI <http://doi.acm.org/10.1145/1654059.1654115>. URL <http://doi.acm.org/10.1145/1654059.1654115>
17. Nickolayev, O., Roth, P., Reed, D.: Real-time statistical clustering for event trace reduction. *International Journal of High Performance Computing Applications* **11**(2), 144 (1997)
18. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualise and analyze parallel code. In: *Proceedings of Transputer and occam Developments, WOTUG-18., Transputer and Occam Engineering*, vol. 44, pp. 17–31. [S.I.]: IOS Press, Amsterdam (1995)
19. Schnorr, L.M., Huard, G., Navaux, P.O.A.: A hierarchical aggregation model to achieve visualization scalability in the analysis of parallel applications. *Parallel Computing* **38**(3), 91–110 (2012). DOI 10.1016/j.parco.2011.12.001

20. Schnorr, L.M., Legrand, A., Vincent, J.M.: Detection and analysis of resource usage anomalies in large distributed systems through multi-scale visualization. *Concurrency and Computation: Practice and Experience* **24**(15), 1792–1816 (2012). DOI 10.1002/cpe.1885
21. Schnorr, L.M., de Oliveira Stein, B., de Kergommeaux, J., Mounié, G.: Pajé trace file format. Tech. rep., ID-IMAG, Grenoble, France (2012). <http://paje.sf.net>
22. Shende, S., Malony, A.: The tau parallel performance system. *International Journal of High Performance Computing Applications* **20**(2), 287 (2006)
23. Wickham, H.: *ggplot2: elegant graphics for data analysis*. Springer-Verlag New York Inc (2009)