VI-HPS VIRTUAL INSTITUTE – HIGH PRODUCTIVITY SUPERCOMPUTING

Tools Guide

October 2016

allinea
JÜLICH FORSCHUNGSZENTRUM
RWTH AACHEN UNIVERSITY
TECHNISCHE UNIVERSITÄT DRESDEN
UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
Universität Stuttgart
Barcelona Supercomputing Center Centro Nacional de Supercomputación
Lawrence Livermore National Laboratory
TECHNISCHE UNIVERSITÄT DARMSTADT
Technische Universität München
UNIVERSITY OF OREGON
THE UNIVERSITY of TENNESSEE

Introduction

The mission of the Virtual Institute - High Productivity Supercomputing (VI-HPS[1]) is to improve the quality and accelerate the development process of complex simulation codes in science and engineering that are being designed to run on highly-parallel computer systems. For this purpose, the partners of the VI-HPS are developing integrated state-of-the-art programming tools for high-performance computing that assist programmers in diagnosing programming errors and optimizing the performance of their applications.

This Tools Guide offers a brief overview of the technologies and tools developed by the twelve partner institutions of the VI-HPS. It is intended to assist developers of simulation codes in deciding which of the tools of the VI-HPS portfolio is best suited to address their needs with respect to debugging, correctness checking, and performance analysis. To simplify navigation and to quickly locate the appropriate tool for a particular use case, an icon list on the left margin of each double page indicates the main characteristics of the corresponding tool. The following paragraphs provide brief definitions of what is meant by each of these icons in the context of this guide.

Focus
single
parallel

Single-node vs. Parallel: These icons indicate whether a tool focuses on either *single-node* or *parallel* characteristics, or both. Here, single-node refers to characteristics of serial, shared-memory or accelerated programs executing on a single system, while *parallel* relates to programs executing on multiple nodes of a cluster using some communication library such as MPI (i.e., using distributed memory parallelism).

Focus
perform
debug
correct

Performance vs. Debugging vs. Correctness: *Performance* tools provide information about the runtime behavior of an application and/or inefficient usage of the available hardware resources. This data can be obtained in various ways, e.g., through static code analysis, measurements, or simulation. *Debugging* tools, on the other hand, may be used to investigate a program – either live at execution time or post-mortem – for possible errors by examining the value of variables and the actual control flow. In contrast, a *correctness* checking tool detects errors in the usage of programming models such as MPI against certain error patterns and reports them to the user, usually performing the analysis right at runtime.

[1] http://www.vi-hps.org

Programming models: Over the years, many different programming models, libraries and language extensions have been developed to simplify parallel programming. Unfortunately, tools need to provide specific support for each programming model individually, due to their different characteristics. The corresponding icon list indicates which of the programming models and libraries most-commonly used in the area of high-performance computing are supported by a tool. In particular, these are the de-facto standard for distributed-memory parallelization *MPI*, the shared-memory programming extensions *OpenMP*, *Pthreads* (a.k.a. POSIX threads) and *OmpSs*, the programming models *CUDA*, *OpenCL* and *OpenACC* targeting accelerators, as well as the partitioned global address space (PGAS) languages/libraries *UPC* and *SHMEM*. However, it may be possible that a tool supports additional programming models, which will then be indicated in the tool description.

Prog. model

- MPI
- OpenMP
- Pthreads
- OmpSs
- CUDA
- OpenCL
- OpenACC
- UPC
- SHMEM

Languages: Finally, some tools may be restricted with respect to the programming languages they support, for example, if source-code processing is required. Here, we only consider the most commonly used programming languages in HPC, namely *C*, *C++* and *Fortran*. Again, it may be possible that tools support further languages or are even language-independent, which will then be mentioned in the description.

Language

- C
- C++
- Fortran

Imprint
Copyright © 2016 Partners of the Virtual Institute – High Productivity Supercomputing

Contact: info@vi-hps.org

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Allinea DDT

Allinea DDT is a modern and easy to use parallel debugger widely used by software developers and computational scientists in industry, academia and government research. It is designed to work at all scales and is the only tool proven in production usage at Petascale âĂŞ with production quality scalability beyond Petascale since 2010. Its interface simplifies concurrency and is highly responsive even at extreme scale.

The tool shares the same configuration and interface as its sister product Allinea MAP.

Typical questions Allinea DDT helps to answer

- Where is my application crashing?
- Why is my application crashing?
- Why is my application hanging?
- What is corrupting my calculation?

Workflow

Allinea DDT can be used on any supported platform to debug problems in application behaviour. The first step should be to compile the errant application with the “-g” compiler flag to ensure debugging information is provided to the debugger.

Allinea DDT can launch the application interactively, or via the batch scheduler, or from inside an existing batch scheduler allocation. Where an application has hung, the debugger can attach to existing processes. A native remote client allows users to debug graphically from remote locations.

Users interact with the debugged processes - being able to step or “play” processes, and examine where all processes are, and their variable values and array data across processes. Memory debugging can be enabled to detect common errors such as reading beyond array bounds automatically.

Platform support

IBM Blue Gene/Q, Linux x86_64, ARMv7, PowerPC (AIX and Linux) and all MPIs.

License

Commercial

Web page

`http://www.allinea.com/products`

Contact

support@allinea.com

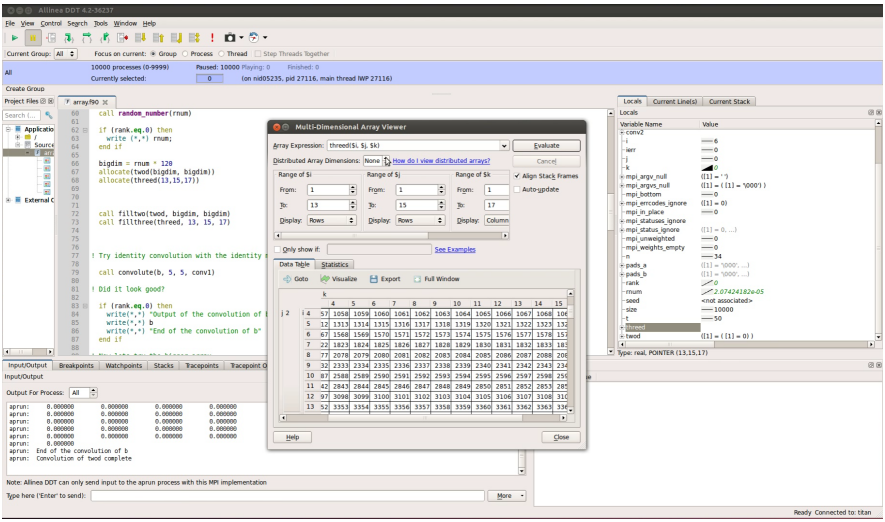


Figure 1: Allinea DDT parallel debugging session showing multi-dimensional array viewer.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Allinea MAP

Allinea MAP is a modern and easy to use profiling tool that is designed to help users visually identify performance issues in their application. It integrates profiling information alongside source code and can show metrics such as vectorization, communication and I/O. The tool shares the same configuration, interface and scalable architecture as its sister product Allinea DDT.

Typical questions Allinea MAP helps to answer

- Where is my code slow - what line and why?
- Am I achieving good vectorization?
- Is memory usage killing my performance?

Workflow

Applications can be launched via Allinea MAP and the performance data will be recorded automatically. There is no need to recompile, although a “-g” flag will ensure accuracy of source line information. The “.map” files are analysed inside the tool and are sufficiently compact to be easily shared.

Platform support

Linux x86_64 and all MPIs.

License

Commercial

Web page

`http://www.allinea.com/products/map`

Contact

support@allinea.com

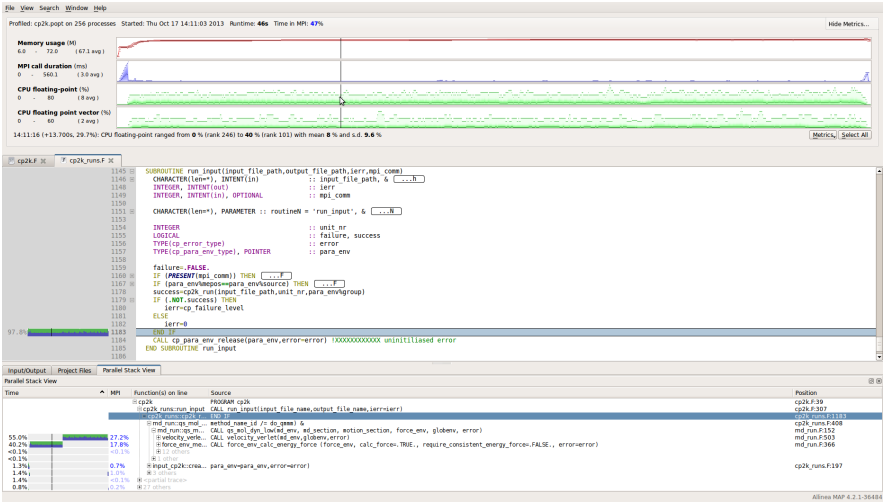


Figure 2: Allinea MAP parallel profiling session showing execution hotspots and evolution charts.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Allinea Performance Reports

Allinea Performance Reports is a performance tool that aims to provide information for anyone involved in HPC, not just software developers. It does not require configuration or any change to the profiled application.

The output provided from a run is a single one-page report on application performance - containing information such as vectorization, communication, energy usage and I/O - with advice about what can be explored to improve the performance.

Typical questions Allinea Performance helps to answer

- Am I achieving good vectorization?
- Is memory usage killing my performance?
- What system or usage changes could I make to improve performance?

Workflow

Applications are launched with a simple prefix-command (“perf-report”) to the existing MPI launch line. There is no need to recompile or relink on most platforms. The “.html” report file created is then viewable in any standard browser.

Platform support

Linux x86_64 and all major MPIs.

License

Commercial

Web page

`http://www.allinea.com/products/performance`

Contact

support@allinea.com

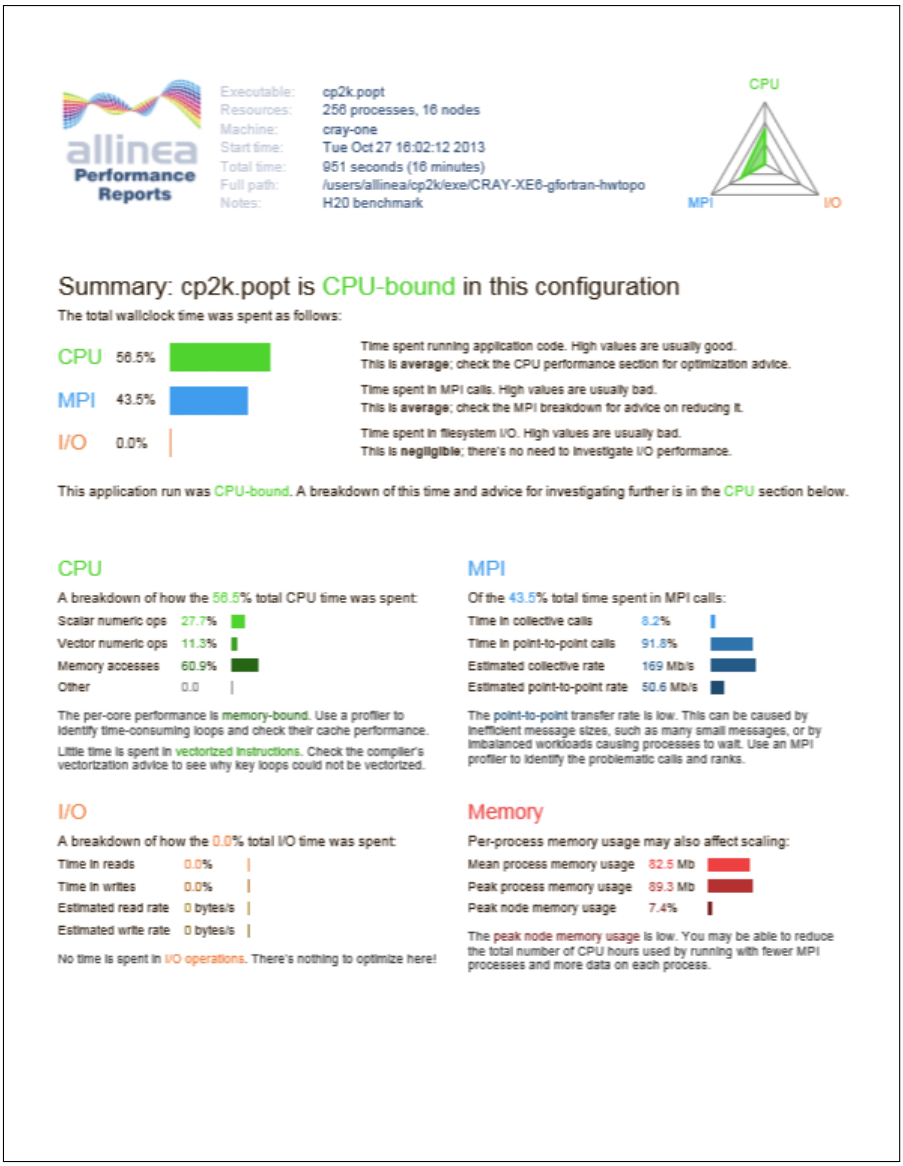


Figure 3: Allinea Performance Reports single page report of an application's CPU, MPI, I/O and Memory usage.

Focus

- single
- parallel
- perform
- debug
- correct

Prog. model

- MPI
- OpenMP
- Pthreads
- OmpSs
- CUDA
- OpenCL
- OpenACC
- UPC
- SHMEM

Language

- C
- C++
- Fortran

Archer

Archer is a data race detector for OpenMP programs.

Archer combines static and dynamic techniques to identify data races in large OpenMP applications, leading to low runtime and memory overheads, while still offering high accuracy and precision. It builds on open-source tools infrastructure such as LLVM and ThreadSanitizer to provide portability.

Typical questions Archer helps to answer

- My OpenMP program intermittently fails (e.g. hang, crash, incorrect results) or slows down, is this caused by a data race?
- At what point of execution (i.e., source line and stack trace), does this race occur exactly?
- What is the root cause (e.g., incorrect variable marking and unsynchronized global variable access)?

Workflow

Compile application with

```
clang-archer example.c -o example
```

Platform support

Linux x86_64, IBM Power
Depends on LLVM/clang

License

BSD 3-Clause License

Web page

`https://github.com/pruner/archer`

Contact

simone@cs.utah.edu

Figure 5 gives detailed information for a data race detected by Archer in the source code displayed in Figure 4.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 0;
6
7      #pragma omp parallel
8      {
9          // Unprotected read
10         if (a < 100) {
11             // Critical section
12             #pragma omp critical
13             {
14                 // Shared memory access
15                 a++;
16             }
17         }
18     }
19 }
```

Figure 4: OpenMP example with a data race.

```
WARNING: ThreadSanitizer: data race (pid=174295)
  Read of size 4 at 0x7fffffffdcdc by thread T2:
    #0 .omp_outlined. race.c:10:9 (race+0x0000004a6dce)
    #1 __kmp_invoke_microtask <null> (libomp_tsan.so)

  Previous write of size 4 at 0x7fffffffdcdc by main thread:
    #0 .omp_outlined. race.c:15:10 (race+0x0000004a6e2c)
    #1 __kmp_invoke_microtask <null> (libomp_tsan.so)
```

Figure 5: Archer output for the data race.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Callgrind

Callgrind is a profiling tool for multithreaded, compiled binary code using execution-driven cache simulation. It is able to build the dynamic call graph from execution on the fly. The results are best browsed with the KCachegrind GUI, which provides call graph and treemap visualizations as well as annotated source and assembler instruction views.

Simulating an easy-to-understand machine model, Callgrind allows for reproducible measurements which may not be available through hardware, such as sub-cacheline utilization.

Typical questions Callgrind helps to answer

- What is the dynamic call graph of a program?
- Is bad cache exploitation the reason for slow program execution?
- What are the call-paths suffering from bad cache behavior?
- Does a given cache optimization actually reduce misses?

Workflow

Callgrind does its observation of code execution by automatic runtime instrumentation using the open-source tool Valgrind. As such, the only preparation needed for detailed analysis is to add debug information to the optimized binary, typically via compiler options “-g -O2”. As simulation can induce a slowdown of up to factor 100, the program may be modified to execute only relevant parts. Further, for sections of code, cache simulation and/or call graph generation may be skipped for faster execution (with slowdown down to factor 3). The reproducibility of simulation allows for very detailed comparison of the effect of code modifications (especially cache optimization).

Platform support

Callgrind is part of Valgrind releases, and supports the same platforms (for Valgrind 3.9.0, this includes Linux on x86/x86_64, Power, ARM, MIPS).

License

GNU General Public License (GPL) v2

Web page

`http://www.valgrind.org`, `http://kcachegrind.sourceforge.net`

Contact

kcachegrind-callgrind@lists.sourceforge.net

Figure 6 shows the call graph of the inner workings of the Intel OpenMP runtime, calling tasks from a Jacobi solver which uses recursive blocking for cache optimization. Callgrind allows recursion levels of the same function to be shown as separate items.

While the GUI is comfortable, Callgrind also comes with standard terminal tools to show the results, such as annotated butterfly call relation lists. Further, it is possible to control running simulations (show current execution context, dump results, switch simulation on/off).

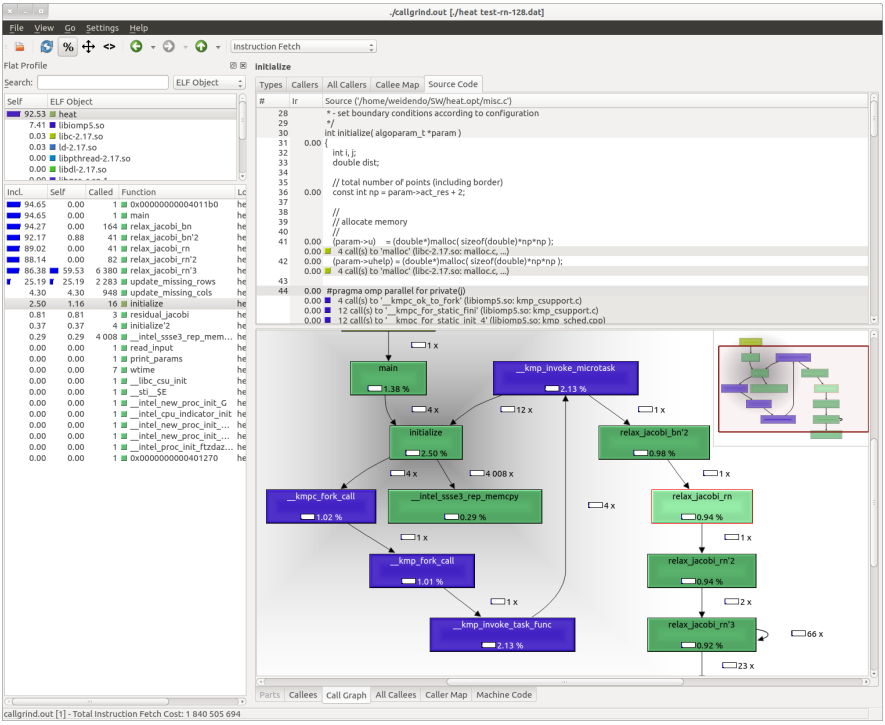


Figure 6: KCachegrind showing results from a Callgrind simulation run.

Focus

- single
- parallel
- perform
- debug
- correct

Prog. model

- MPI
- OpenMP
- Pthreads
- OmpSs
- CUDA
- OpenCL
- OpenACC
- UPC
- SHMEM

Language

- C
- C++
- Fortran

Cube

Cube is a generic tool for manipulating and displaying a multi-dimensional performance space consisting of the dimensions (i) performance metric, (ii) call path, and (iii) system resource. Each dimension can be represented as a tree, where non-leaf nodes of the tree can be collapsed or expanded to achieve the desired level of granularity and present inclusive or exclusive metric values. In addition, Cube can display multi-dimensional Cartesian process topologies, highlight a region from a source file, and present descriptions of metrics.

Typical questions Cube helps to answer

- Which metrics have values indicating performance problems?
- Which call-paths in the program have these values?
- Which processes and threads are most affected?
- How are metric values distributed across processes/threads?
- How do two analysis reports differ?

Workflow

Scalasca, Score-P and other tools use the provided libraries to write analysis reports in Cube format for subsequent interactive exploration in the Cube GUI. Additional utilities are provided for processing analysis reports.

Platform support

GUI: Linux (x86/x86_64/IA64/PPC/Power), Mac OS X (x86_64), Windows 7;

Libraries & utilities: IBM Blue Gene/P/Q, Cray XT/XE/XK/XC, SGI Altix (incl. ICE + UV), Fujitsu FX-10/100 & K Computer, Tianhe-1A, IBM SP & Blade clusters (incl. AIX), Intel Xeon Phi, Linux clusters (x86/x86_64)

License

BSD 3-Clause License

Web page

`http://www.scalasca.org`

Contact

scalasca@fz-juelich.de

Figure 7 shows a screenshot of a Scalasca trace analysis of the Zeus/MP2 application in the Cube analysis report explorer. The left panel shows that about 10% of the execution time is spent in the “Late Sender” wait state, where a blocking receive operation is waiting for data to arrive. The middle panel identifies how this wait state is distributed across the call tree of the application. For the selected `MPI_Waitall` call, which accumulates 12.8% of the Late Sender time, the distribution across the system is presented in the right panel, here in the form of a 3D process topology which reflects the domain decomposition used by Zeus/MP2.

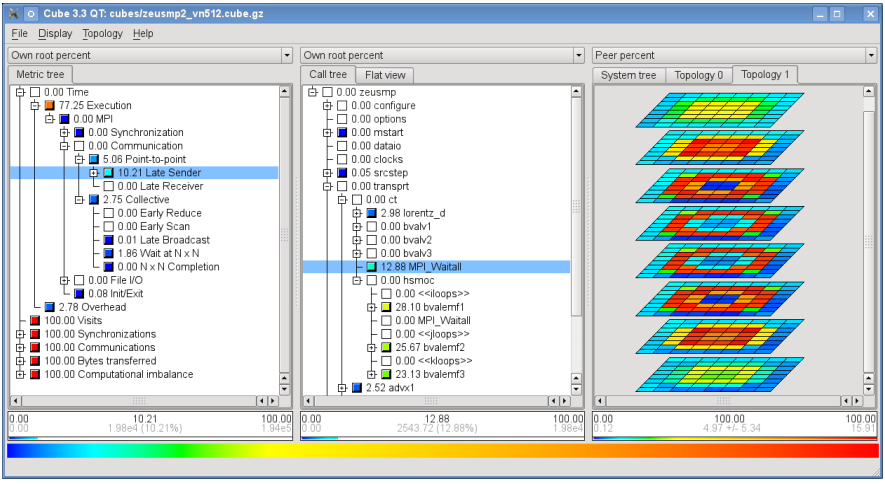


Figure 7: Scalasca trace analysis result displayed by Cube for exploration.

Focus
single
parallel
perform
debug
correct

Prog. model
MPI
OpenMP
Pthreads
OmpSs
CUDA
OpenCL
OpenACC
UPC
SHMEM

Language
C
C++
Fortran

Dimemas

Dimemas is a performance analysis tool for message-passing programs. The Dimemas simulator reconstructs the temporal behavior of a parallel application using a recorded event trace and allows simulating the parallel behavior of that application on a different system. The Dimemas architecture model is a network of parallel clusters. Dimemas supports two main types of analyses: what-if studies to simulate how an application would perform in a given scenario (e.g. reducing to half the network latency, moving to a CPU three times faster...), and parametric studies to analyze the sensitivity of the code to system parameters (e.g. the execution time for varying network bandwidths).. The target system is modeled by a set of key performance factors including linear components like the MPI point to point transfer time, as well as non-linear factors like resources contention. By using a simple model Dimemas allows executing parametric studies in a very short time frame. Dimemas can generate among others a Paraver trace file, enabling the user to conveniently examine and compare the simulated run and understand the application behavior.

Typical questions Dimemas helps to answer

- How would my application perform in a future system?
- Increasing the network bandwidth would improve the performance?
- Would my application benefit from asynchronous communications?
- Is my application limited by the network or the serializations and dependency chains within my code?
- What would be the impact of accelerating specific regions of my code?

Workflow

The first step is to translate a Paraver trace file to Dimemas format. Thereby, it is recommended to focus on a representative region with a reduced number of iterations. Second, the user specifies via a configuration file the architectural parameters of the target machine and the mapping of the tasks on to the different nodes. Third, the output Paraver trace file allows then to analyze and compare the simulated scenario with the original run using the Paraver tool.

Platform support

Linux ix86/x86_64, IBM Blue Gene/P/Q, ARM, Fujitsu FX10, SGI Altix, Power, Cray XT, Intel Xeon Phi, GPUs (CUDA, OpenCL)...

License

GNU Lesser General Public License (LGPL) v2.1

Web page

`http://www.bsc.es/dimemas`

Contact

tools@bsc.es

Figure 8 shows the results of an analysis of sensitivity to network bandwidth reductions for two versions of WRF code, NMM and ARW, and with different number of MPI ranks. We can see that the NMM version demand less bandwidth (256MB/s) than the ARW version.

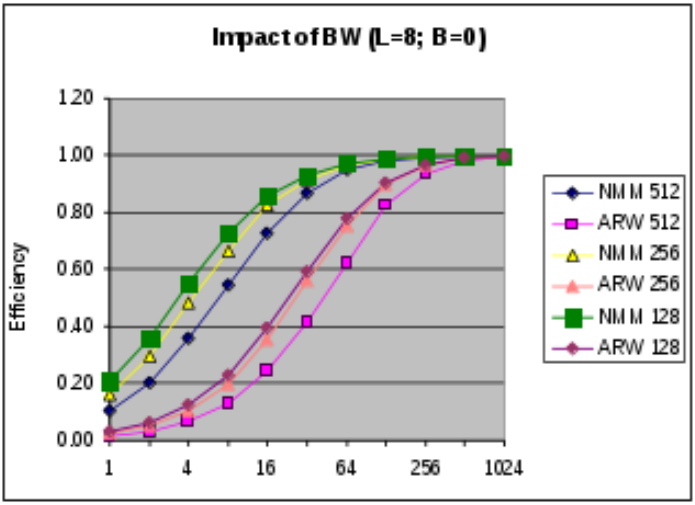


Figure 8: Dimemas sensitivity analysis to network bandwidth.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Extra-P

Extra-P is an automatic performance-modeling tool that supports the user in the identification of *scalability bugs*. A scalability bug is a part of the program whose scaling behavior is unintentionally poor, that is, much worse than expected.

Extra-P uses measurements of various performance metrics at different processor configurations as input to represent the performance of code regions (including their calling context) as a function of the number of processes. All it takes to search for scalability issues even in full-blown codes is to run a manageable number of small-scale performance experiments, launch Extra-P, and compare the asymptotic or extrapolated performance of the worst instances to the expectations. Besides the number of processes, it is also possible to consider other parameters such as the input problem size.

Extra-P generates not only a list of potential scalability bugs but also human-readable models for all performance metrics available such as floating-point operations or bytes sent by MPI calls that can be further analyzed and compared to identify the root causes of scalability issues.

Typical questions Extra-P helps to answer

- Which regions of the code scale poorly?
- Which metrics cause the run-time to scale poorly?
- What are the best candidates for optimization?
- How will my application behave on a larger machine?

Workflow

Extra-P accepts input files in the Cube format and processes them into a condensed Cube format containing functions for each metric and call path rather than individual measured values. Tools such as Scalasca, Score-P, and others are provided with libraries that produce analysis reports in the Cube format. GUI plugins are provided to visualize, browse, and manipulate the resulting models in the Cube browser. Detailed textual results are also generated by Extra-P for the in-depth analysis of sensitive code regions.

Platform support

Linux (x86/x86_64/IA64/PPC/Power), Mac OS X (x86_64);

License

BSD 3-Clause License

Web page

`http://www.scalasca.org`

Contact

calotoiu@cs.tu-darmstadt.de

Figure 9 shows performance models generated for different call paths in UG4, a multigrid framework. The call tree on the left allows the selection of models to be plotted on the right. The color of the squares in front of each call path highlights the complexity class. The execution time of the call path ending in kernel `norm` has a measured complexity of $\sqrt{p}$.

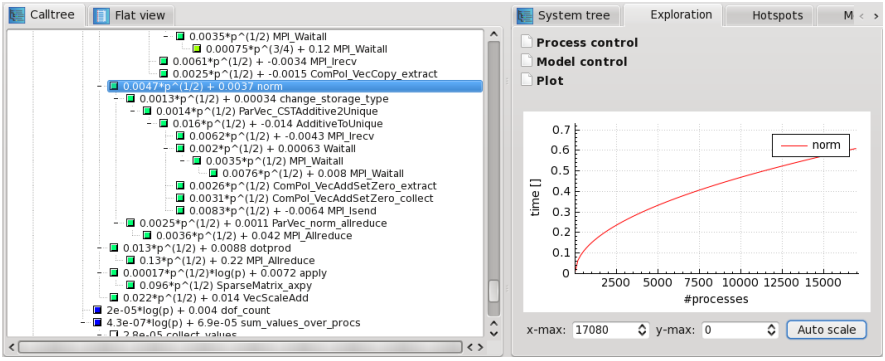


Figure 9: Interactive exploration of performance models in Extra-P.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

MAQAO

MAQAO (**M**odular **A**ssembly **Q**uality **A**nalyzer and **O**ptimizer) is a performance analysis and optimization tool suite operating at binary level (no recompilation necessary). The main goal of MAQAO is to provide application developers with synthetic reports in order to help them optimizing their code. The tool mixes both dynamic and static analyses based on its ability to reconstruct high level structures such as functions and loops from an application binary.

Another key feature of MAQAO is its extensibility. Users can easily write their own modules thanks to an API using the Lua scripting language, allowing fast prototyping of new MAQAO modules.

Typical questions MAQAO helps to answer

- What is the time breakdown between I/O, MPI, OpenMP, PThreads in my application ?
- Which functions and loops are the most time consuming ?
- Are all my hotspots consuming the same amount of time across all the processes/threads (load balancing)?
- How can I optimize a loop ? Which performance factor may I gain ?

Workflow

The first step consists in pinpointing the most time consuming hotspots in order to quickly identify where optimization efforts should be directed to. This is done through the LProf module, a sampling-based lightweight profiler that offers results at the function and loop levels. LProf is also able to categorize its results depending on their source: parallelization (runtime), I/O, memory, main code, etc.

The Code Quality Analyzer (CQA) module can then help users optimizing hot loops. CQA performs static analysis to assess the quality of the code generated by the compiler and produces a set of reports describing potential issues, an estimation of the potential gain if fixed, and hints on how to achieve this through compiler flags or source code transformations.

Platform support

Linux clusters (Intel 64 and Xeon Phi)

License

GNU Lesser General Public License (LGPL) v3

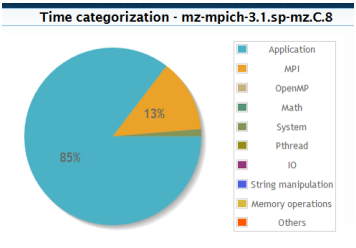
Web page

`http://www.maqao.org`

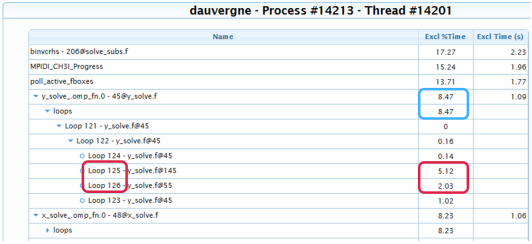
Contact

support@maqao.org

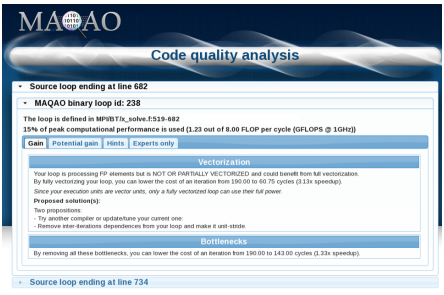
As presented in Figure 10, MAQAO features high level HTML outputs providing synthetic metrics. Figure 10(a) displays profiling results categorized by their source. Figure 10(b) displays function and loop hotspots as identified by profiling. Figures 10(c) and 10(d) present related code quality reports showing potential gain and hints.



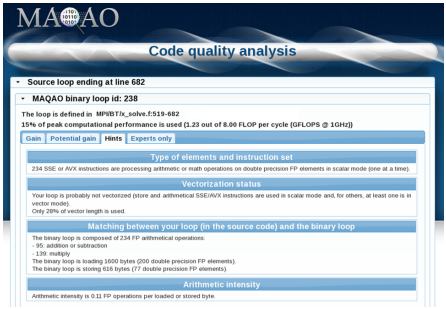
(a) Profiling categorization



(b) Profiling report by loops



(c) Code quality report



(d) Code quality advanced report

Figure 10: HTML outputs for various MAQAO modules.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

MUST

MUST detects whether an application conforms to the MPI standard and is intended to scale with the application (O(10,000) processes). At runtime it transparently intercepts all MPI calls and applies a wide range of correctness checks (including type matching checks and deadlock detection) to their arguments. This allows developers to identify manifest errors (ones you already noticed), portability errors (manifest on other platforms), and even unnoticed errors (e.g., silently corrupted results). When an application run with the tool finishes it provides its results in a correctness report for investigation.

Typical questions MUST helps to answer

- Has my application potential deadlocks?
- Am I doing type matching right?
- Does my application leak MPI resources?
- Other hidden errors?

Workflow

Replace mpiexec/mpirun/runjob/.. by mustrun:
`mpiexec -np 1024 executable` → `mustrun -np 1024 executable`
After the run inspect the outputfile `MUST_Output.html` with a browser (w3m, firefox, . . .).

For Batchjobs: Note that the run uses extra MPI processes to execute checks, use "--must:info" to retrieve resource allocation information.

Platform support

Linux x86_64, IBM Blue Gene/Q, Cray XE (early support), SGI Altix4700
Tested with various MPI implementations:
Open MPI, Intel MPI, MPICH, MVAPICH2, SGI MPT, . . .

License

BSD 3-Clause License

Web page

`https://www.itc.rwth-aachen.de/must`

Contact

must-feedback@lists.rwth-aachen.de

Figure 11 gives detailed information for a deadlock situation detected by MUST (caused by mismatching *tags*):

Rank 0 reached `MPI_Finalize`.

Rank 1 is at `MPI_Recv(`*`src`*`=MPI_ANY_SOURCE, `*`tag`*`=42)`.

Rank 2 did `MPI_Ssend(`*`dest`*`=1, `*`tag`*`=43)`.

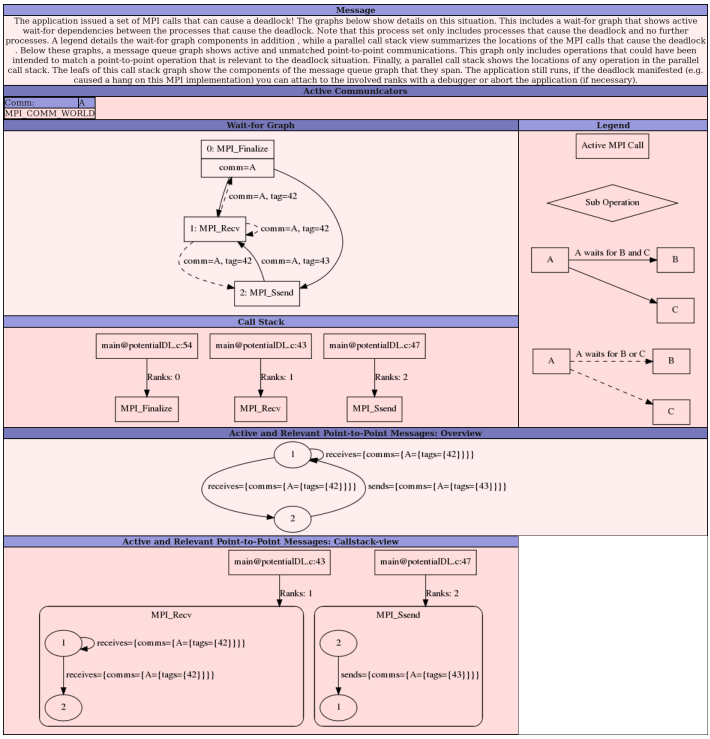


Figure 11: Visualization of a deadlock situation.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Open|SpeedShop

Open|SpeedShop is an open source multi platform performance tool that is targeted to support performance analysis of applications running on both single nodes and large scale platforms. Open|SpeedShop is explicitly designed with usability in mind and provides both a comprehensive GUI as well as a command line interface (CLI). The base functionality includes sampling experiments, support for callstack analysis, access to hardware performance counters, tracing and profiling functionality for both MPI and I/O operations, as well floating point exception analysis. Each of these functionalities is available as an *Experiment* that a user can select and execute on a given target application. Several other experiments, such as memory analysis and CUDA support, are available in experimental versions.

Typical questions this tool helps to answer

- In which module, function, loop or statement is my code spending most of its time (Experiment name: pcsamp)?
- On which call paths were my hotspots reached (Experiment name: usertime)?
- Which hardware resources cause bottlenecks for my execution (Experiment name: hwcsamp)?
- How do hardware performance counter results, like TLB misses, map to my source (Experiment name: hwc/hwctime)?
- How much time am I spending in I/O or MPI operations (Experiment name: io/iot and mpi/mpit)?

Workflow

Open|SpeedShop can be applied to any sequential or parallel target application in binary form. To get finer grained attribution of performance to individual statements, it is recommended to apply the tool to codes compiled with -g, although this is not a requirement. The user picks an experiment (starting with the simple sampling experiment pcsamp is typically a good idea) and prepends the execution of the code (incl. MPI job launcher) with an Open|SpeedShop launch script for that experiment.

For example, if the target application is typically launched with:

```
mpirun -np 128 a.out
```

launching it with the pcsamp experiment would be:

```
osspcsamp "mpirun -np 128 a.out"
```

At the end of the execution, the tool provides a first overview of the observed performance and then creates a database with all performance data included, which can viewed in the Open|SpeedShop GUI:

```
openss -f <database-filename.openss>
```

Platform support

Linux x86_64 workstations and clusters, IBM Blue Gene, and Cray.

License

Open|SpeedShop is available under LGPL (main tool routine: GPL).

Web page

`http://www.openspeedshop.org/`

Contact

oss-questions@krellinst.org

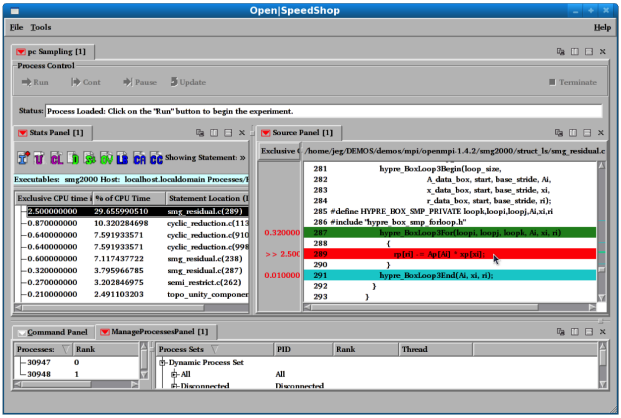


Figure 12: GUI showing the results of a sampling experiment (left: time per statement, right: information mapped to source)

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

PAPI

Parallel application performance analysis tools on large scale computing systems typically rely on hardware counters to gather performance data. The PAPI performance monitoring library provides tool designers and application engineers with a common and coherent interface to the hardware performance counters (available on all modern CPUs) and other hardware components of interest (e.g., GPUs, network, and I/O systems).

Typical questions PAPI helps to answer

- What is the relation between software performance and hardware events?
- What are the number of cache misses, floating-point operations, executed cycles, etc. of the routines, loops in my application?
- How much data is sent over the network? How much data originates from a node and how much is passed through a node?
- What is the system's power usage and energy consumption when my application is executed?

Workflow

While PAPI can be used as a stand-alone tool, it is more commonly applied as a middleware by third-party profiling, tracing as well as sampling tools (e.g., CrayPat, HPCToolkit, Scalasca, Score-P, TAU, Vampir), making it a de facto standard for hardware counter analysis.

The events that can be monitored involve a wide range of performance-relevant architectural features: cache misses, floating point operations, retired instructions, executed cycles, and many others. By tightly coupling PAPI with the tool infrastructure, pervasive performance measurement capability - including accessing hardware counters, power and energy measurements, and data transfers, at either the hardware or software library level - can be made available.

Platform support

AMD, ARM Cortex A8, A9, A15 (coming soon ARM64), Cray, Fujitsu K Computer, IBM Blue Gene Series (including Blue Gene/Q: 5D-Torus, I/O system, CNK, EMON power), IBM Power Series; Intel (including RAPL power/energy, MIC power/energy), Linux clusters (x86/x86_64, MIPS), NVidia (Tesla, Kepler, NVML).

License

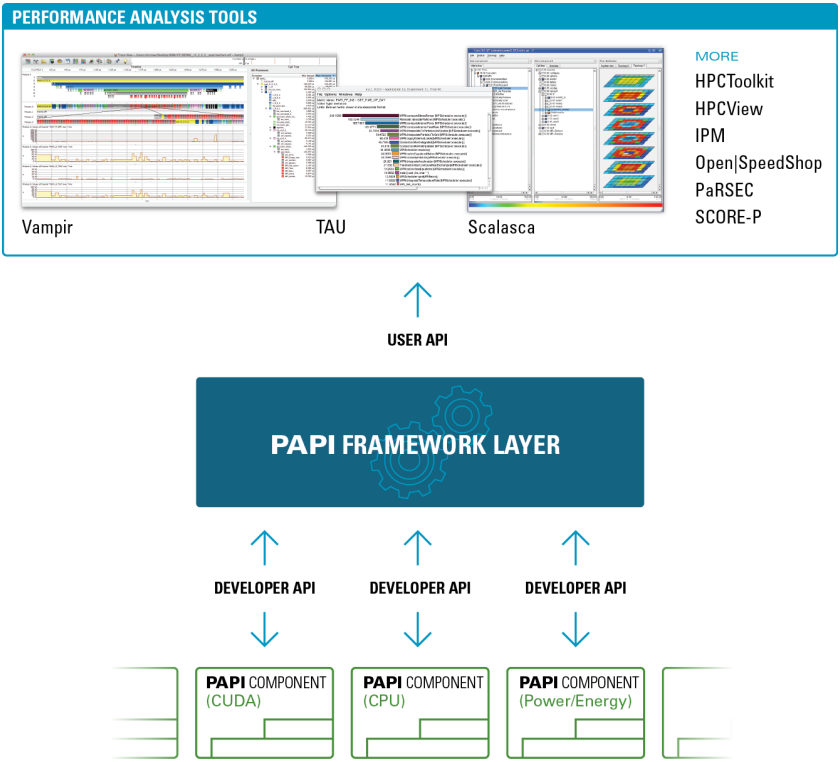
BSD 3-Clause License

Web page

`http://icl.cs.utk.edu/papi`

Contact

ptools-perfapi@eecs.utk.edu



Over time, other system components, beyond the processor, have gained performance interfaces (e.g., GPUs, network, I/O, Power/Energy interfaces). To address this change, PAPI was redesigned to have a component architecture that allows for modular access to these new sources of performance data.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Paraver

Paraver is a performance analyzer based on event traces with a great flexibility to explore the collected data, supporting a detailed analysis of the variability and distribution of multiple metrics with the objective of understanding the application's behavior. Paraver has two main views with high flexibility to define and correlate them. The timeline view displays the application behavior with time, while the statistics view (histograms, profiles) complements the analysis with distribution of metrics. To facilitate extracting insight from detailed performance data, during the last years new modules introduce additional performance analytics techniques: clustering, tracking and folding allow the performance analyst to identify the program structure, study its evolution and look at the internal structure of the computation phases. The tool has been demonstrated to be very useful for performance analysis studies, with unique features that reveal profound details about an application's behavior and performance.

Typical questions Paraver helps to answer

- How well does the parallel program peform and how does the behavior change over time?
- What is the parallelization efficiency and the effect of communication?
- What differences can be observed between two executions?
- Are performance or workload variations the cause of load imbalances in computation?
- Which performance issues are reflected by hardware counters?

Workflow

The basis of an analysis with Paraver is a measurement of the application execution with its performance monitor Extrae. After opening the resulting trace file in Paraver the user can select from a subset of introductory analysis views that are hinted by the tool based on the recorded metrics. These basic views allow an easy overview of the application behavior. Next to that, Paraver includes a multitude of predefined views enabling a deeper analysis. Furthermore, Paraver offers a very flexible way to combine multiple views, so as to generate new representations of the data and more complex derived metrics. Once a desired view is obtained, it can be stored in a configuration file to apply it again to the same trace or to a different one.

Platform support

Linux ix86/x86_64, IBM Blue Gene/P/Q, ARM, Fujitsu FX10, SGI Altix, Power, Cray XT, Intel Xeon Phi, GPUs (CUDA, OpenCL), ...

License

GNU Lesser General Public License (LGPL) v2.1

Web page

`http://www.bsc.es/paraver`

Contact

tools@bsc.es

Figure 13 shows a histogram of the computation phases colored by the clustering tool. Small durations are located in the left of the picture, and large durations on the right. The variability between the cells that have the same color indicate variance on the duration that would be paid as waiting time within MPI. We can see that the larger computing region (light green on the right) is the one with larger imbalance.

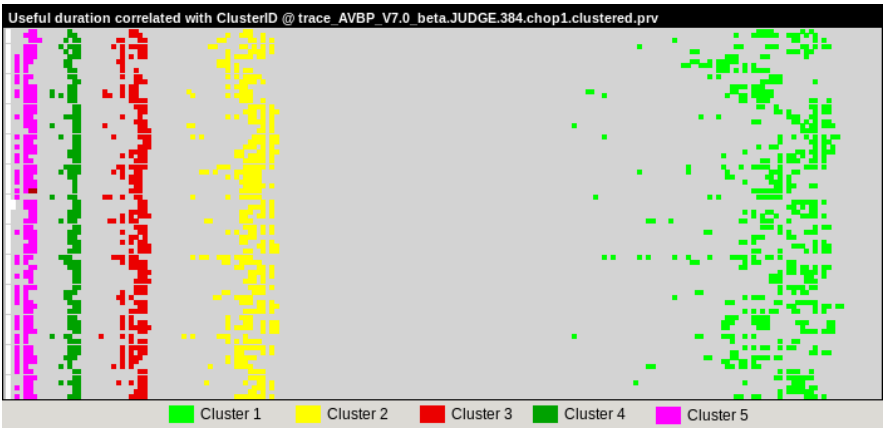


Figure 13: Paraver histogram of the computation phases colored with the cluster ID.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Periscope Tuning Framework

The Periscope Tuning Framework (PTF) is a scalable online automatic tuner and analyzer of HPC applications' performance and energy. PTF performs automatic online search for performance bottlenecks and, when requested, automatically searches for an optimum of a given tuning objective by evaluating a space of tuning alternatives. At the end of the experiment, detailed recommendations are given to the code developer on how to apply the identified optimal configurations in production runs. Also detected performance bottlenecks, called properties, can be displayed in the PTF GUI.

Typical questions PTF helps to answer

- What are the best MPI parameters for eager limit, buffer space, collective algorithms, IO data sieving and number of aggregators?
- What is the most energy-efficient Dynamic Voltage Frequency Scaling setting?
- How many OpenMP threads can be employed productively?
- What compiler flags provide the best performance?

Workflow

Before using PTF, an application has to be instrumented with PTF instrumenter (later also with Score-P) by marking a phase region with the corresponding user instrumentation pragmas. The body of the progress loop of the application is typically selected as the phase region in order to utilize iterative execution of the application for multiple tuning experiments.

After instrumentation, PTF is initiated by starting the front-end agent and specifying the instrumented application, a type of the tuning/analysis to be performed and the execution configuration (number of MPI processes and OpenMP threads). The optimal tuning configuration is printed into the standard output.

Detected performance bottlenecks are stored in PTF report file (.psc) and can be loaded into the PTF GUI in Eclipse.

Platform support

Linux clusters (x86/x86_64)

License

BSD 3-Clause License

Web page

`http://periscope.in.tum.de/`

Contact

periscope@in.tum.de

Figure 14 shows the PTF GUI. Performance bottlenecks found by PTF are shown in the multi-functional table in the lower part of the screen. Double-clicking on any of the bottlenecks results in opening the corresponding file and highlighting the source lines responsible for the inefficiency in the left-upper part of the screen.The right-upper part is dedicated for project exploration.

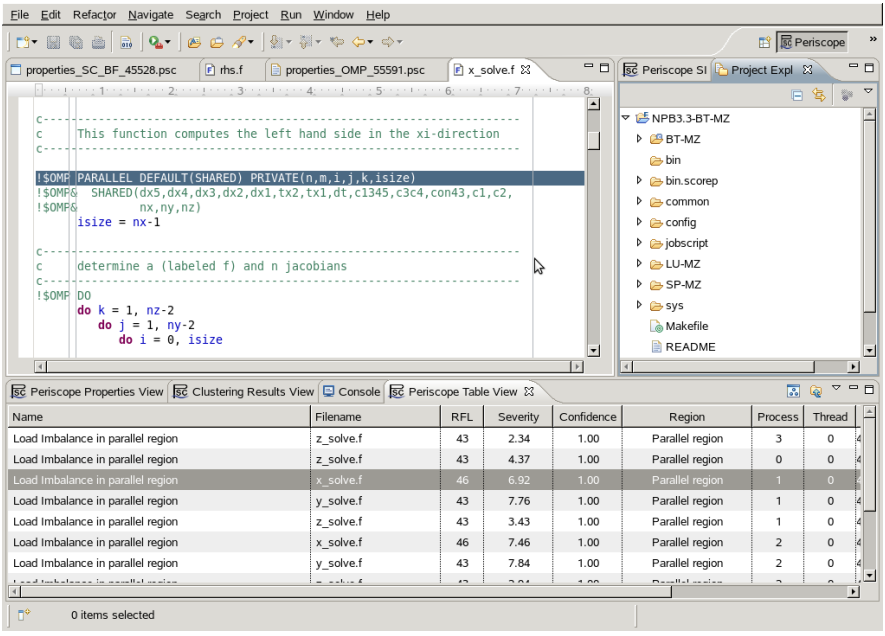


Figure 14: PTF perspective in Eclipse

Focus

- single
- parallel
- perform
- debug
- correct

Prog. model

- MPI
- OpenMP
- Pthreads
- OmpSs
- CUDA
- OpenCL
- OpenACC
- UPC
- SHMEM

Language

- C
- C++
- Fortran

Scalasca Trace Tools

The Scalasca Trace Tools support performance optimization of parallel programs with a collection of highly scalable trace-based tools for in-depth analyses of concurrent behavior. The Scalasca tools have been specifically designed for use on large-scale systems such as the IBM Blue Gene series and Cray XT and successors, but is also well suited for small- and medium-scale HPC platforms. The automatic analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes.

Typical questions the Scalasca Trace Tools help to answer

- Which call-paths in my program consume most of the time?
- Why is the time spent in communication or synchronization higher than expected?
- For which program activities will optimization prove worthwhile?
- Does my program suffer from load imbalance and why?

Workflow

Before any Scalasca analysis can be carried out, an execution trace of the target application needs to be collected. For this task, Scalasca leverages the community-driven instrumentation and measurement infrastructure Score-P. After an optimized measurement configuration has been prepared based on initial profiles, a targeted event trace in OTF2 format can be generated, and subsequently analyzed by Scalasca's automatic event trace analyzer after measurement is complete. This scalable analysis searches for inefficiency patterns and wait states, identifies their root causes (i.e., delays) also along far-reaching cause-effect chains, collects statistics about the detected wait-state instances, and determines a profile of the application's critical path. The result can then be examined using the interactive analysis report explorer Cube.

Platform support

IBM Blue Gene, Cray XT/XE/XK/XC, SGI Altix (incl. ICE/UV), Fujitsu FX-10/100 & K Computer, Tianhe-1A & 2, IBM SP & Blade clusters, Intel Xeon Phi, Linux clusters (x86/x86_64, Power, ARM)

License

BSD 3-Clause License

Web page

`http://www.scalasca.org`

Contact

scalasca@fz-juelich.de

Figure 15 shows part of a time-line of events from three processes, exemplifying results of the Scalasca trace analyzer. First, wait states in communications and synchronizations are detected, such as the "Late Sender" wait states in both message transfers (C→A and A→B) due to receive operations blocked waiting for messages to arrive. Second, the analysis identifies that the wait state on process A is caused directly by the excess computation in `foo` on process C. Besides the extra receive operation on process A, this imbalance is also identified as a cause for the wait state on process B through propagation: by inducing the wait state on process A it is also delaying the following send operation further. Finally, the analysis determines the critical path of execution (outlined), whose profile highlights call paths that are good candidates for optimization.

The analyzer quantifies metric severities for each process/thread and call path, and stores them in an analysis report for examination with Cube. Additional wait-state instance statistics can be used to direct Paraver or Vampir trace visualization tools to show and examine the severest instances.

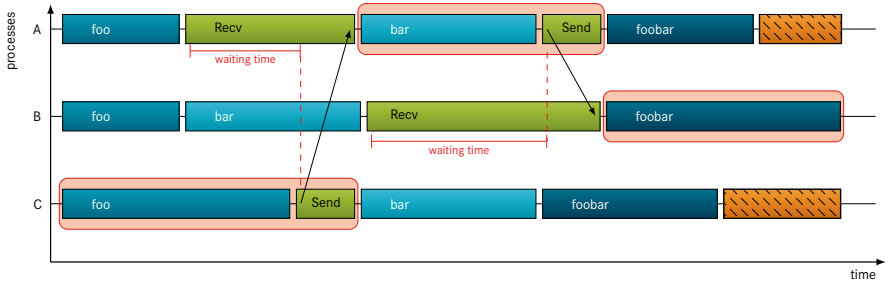


Figure 15: Scalasca automatic trace analysis identification of time in message-passing wait states and on the critical path.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Score-P

The Score-P measurement infrastructure is a highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis. It supports a wide range of HPC platforms and programming models. Score-P provides core measurement services for a range of specialized analysis tools, such as Vampir, Scalasca, TAU, or Periscope.

Typical questions Score-P helps to answer

- Which call-paths in my program consume most of the time?
- How much time is spent in communication or synchronization?

Further analysis tools can also be employed on Score-P measurements.

Workflow

1. Preparation. To create measurements, the target program must be instrumented. Score-P offers various instrumentation options, including automatic compiler instrumentation or manual source-code instrumentation. As an alternative to automatic compiler instrumentation, events can be generated using a sampling approach.

2. Measurement. The instrumented program can be configured to record an event trace or produce a call-path profile. Optionally, PAPI, rusage, and perf hardware metrics can be recorded. Filtering techniques allow precise control over the amount of data to be collected.

3. Analysis. Call-path profiles can be examined in TAU or the Cube profile browser. Event traces can be examined in Vampir or used for automatic bottleneck analysis with Scalasca. Alternatively, on-line analysis with Periscope is possible.

Platform support

IBM Blue Gene/P/Q, Cray XT/XE/XK/XC, SGI Altix/ICE, Fujitsu K Computer, FX10 and FX100, ARM32- and 64-bit, IBM Power 8, Intel Xeon Phi (native mode), Linux clusters (x86/x86_64)

License

BSD 3-Clause License

Web page

`http://www.score-p.org`

Contact

support@score-p.org

Figure 16 is an overview of the Score-P instrumentation and measurement infrastructure and the analysing tools from the VI-HPS ecosystem. Supported programming models and other event sources are modularized at the lowest level. Score-P instruments the application at build time with the necessary code to perform the measurement. Measurement mode and any external sources such as PAPI are specified at runtime. The performance data is stored for postmortem analysis in the open data formats CUBE4 for call-path profiles and OTF2 for event traces. Multiple analysis tools can then work on the same data from a single measurement run.

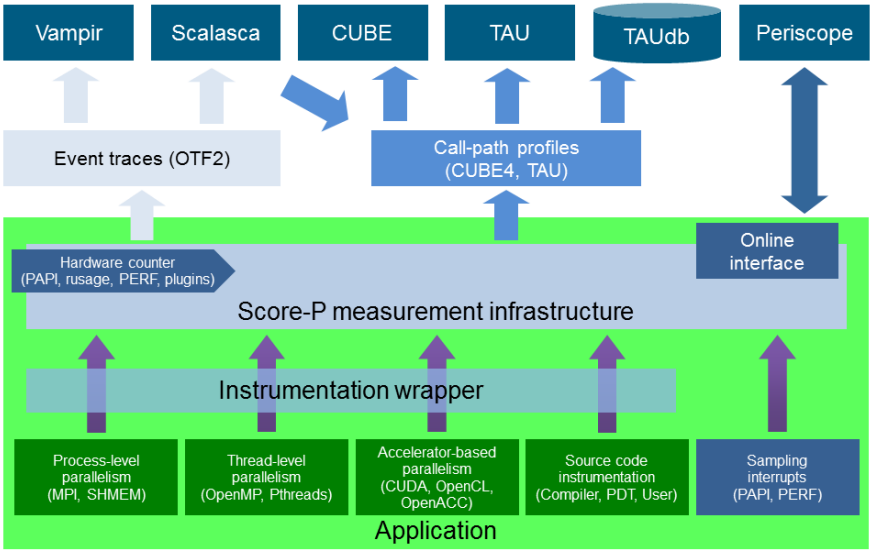


Figure 16: Overview of Score-P instrumentation and measurement infrastructure including produced dataformats and analysing tools.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

STAT — The Stack Trace Analysis Tool

The Stack Trace Analysis Tool gathers and merges stack traces from all processes of a parallel application. The tool produces call graphs: 2D spatial and 3D spatial-temporal; the graphs encode calling behavior of the application processes in the form of a prefix tree. The 2D spatial call prefix tree represents a single snapshot of the entire application. The 3D spatial-temporal call prefix tree represents a series of snapshots from the application taken over time (see Figure 17). In these graphs, the nodes are labeled by function names. The directed edges, showing the calling sequence from caller to callee, are labeled by the set of tasks that follow that call path. Nodes that are visited by the same set of tasks are assigned the same color, giving a visual reference to the various equivalence classes.

Typical questions STAT helps to answer

- Where is my code stuck?
- Which processes have similar behavior?
- Where do I need to start debugging?

Workflow

STAT comes with its own GUI, invoked with the `stat-gui` command. Once launched, this GUI can be used to select the application to debug (in the context of MPI applications typically the job launch process, i.e., mpirun or equivalent). STAT will then attach to the target application processes, gather the stack traces and display them within the GUI for analysis.

Platform support

Linux x86_64 workstations and clusters, IBM Blue Gene, and Cray XT/XE/XK.

License

BSD

Web page

`http://www.paradyn.org/STAT/STAT.html`

Contact

Greg Lee, LLNL (lee218@llnl.gov)

Figure 17 shows a call prefix tree generated by STAT from a sample MPI application which is stalled. At a high-level (before MPI internals), the code has three groups of processes: rank 1 in `do_SendOrStall`, rank 2 in `MPI_Waitall`, and the other 4094 processes in `MPI_Barrier`. Using this information it is sufficient to apply a debugger only to one representative process from each group in order to be able to investigate this problem.

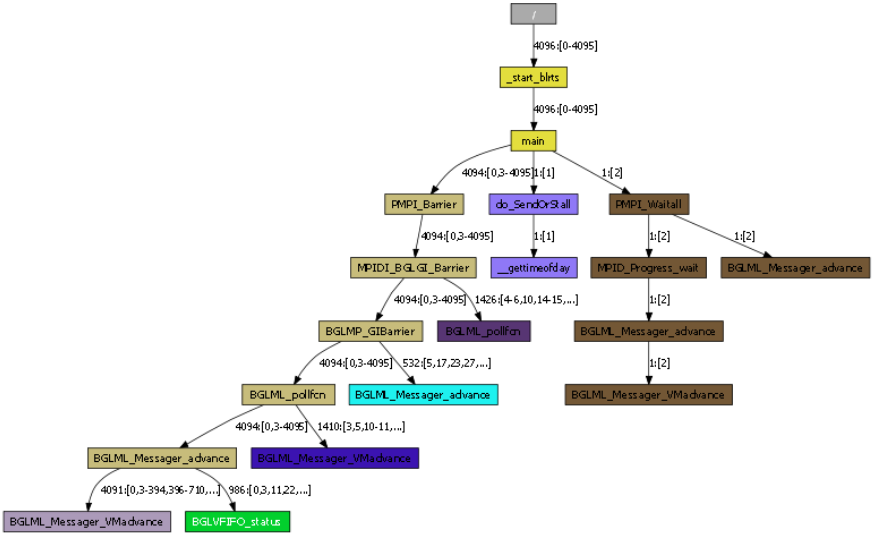


Figure 17: STAT 3D spatial-temporal call prefix tree of stalled execution.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

TAU

TAU is a comprehensive profiling and tracing toolkit that supports performance evaluation of programs written in C++, C, UPC, Fortran, Python, and Java. It is a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, debugging, analysis, and visualization of large-scale parallel computer systems and applications. TAU supports both direct measurement as well as sampling modes of instrumentation and interfaces with external packages such as Score-P, PAPI, Scalasca, and Vampir.

Typical questions TAU helps to answer

- Which routines, loops, and statements in my program consume most of the time?
- Where are the memory leaks in my code and where does my program violate array bounds at runtime?
- What is the extent of I/O and what is the bandwidth of I/O operations?
- What is the performance of kernels that execute on accelerators such as GPUs and Intel Xeon co-processors (MIC).
- What is the extent of variation of the power and heap memory usage in my code? When and where does it show extremes?

Workflow

TAU allows the user to instrument the program in a variety of ways including rewriting the binary using *tau_rewrite* or runtime pre-loading of shared objects using *tau_exec*. Source level instrumentation typically involves substituting a compiler in the build process with a TAU compiler wrapper. This wrapper uses a given TAU configuration to link in the TAU library. At runtime, a user may specify different TAU environment variables to control the measurement options chosen for the performance experiment. This allows the user to generate callpath profiles, specify hardware performance counters, turn on event based sampling, generate traces, or specify memory instrumentation options.

Performance-analysis results may be stored in TAUdb, a database for cross-experiment analysis and advanced performance data mining operations using TAU's PerfExplorer tool. It may be visualized using ParaProf, TAU's 3D profile browser that can show the extent of performance variation and compare executions.

Supported platforms

IBM Blue Gene P/Q, NVIDIA and AMD GPUs and Intel MIC systems, Cray XE/XK/XC30, SGI Altix, Fujitsu K Computer (FX10), NEC SX-9, Solaris & Linux clusters (x86/x86_64,MIPS,ARM), Windows, Apple Mac OS X.

Supported Runtime Layers

MPI, OpenMP (using GOMP, OMPT, and Opari instrumentation), Pthread, MPC Threads, Java Threads, Windows Threads, CUDA, OpenCL, OpenACC.

License

BSD style license

Web page

`http://tau.uoregon.edu`

Contact

tau-bugs@cs.uoregon.edu

Figure 18 below shows a 3D profile of the IRMHD application that shows the extent of variation of the execution time over 2048 ranks. Notice the shape of the *MPI_Barrier* profile.

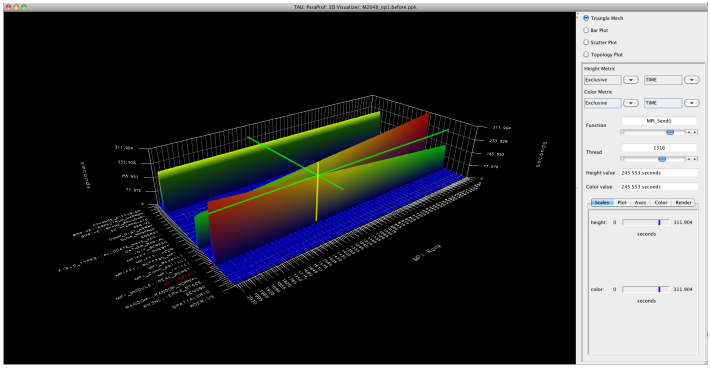


Figure 18: TAU's ParaProf 3D profile browser shows the exclusive time spent (height, color) over ranks for all routines in a code.

Focus

single

parallel

perform

debug

correct

Prog. model

MPI

OpenMP

Pthreads

OmpSs

CUDA

OpenCL

OpenACC

UPC

SHMEM

Language

C

C++

Fortran

Vampir

The Vampir performance visualizer allows to quickly study a program's runtime behavior at a fine level of detail. This includes the display of detailed performance event recordings over time in timelines and aggregated profiles. Interactive navigation and zooming are the key features of the tool, which help to quickly identify inefficient or faulty parts of a program.

Typical questions Vampir helps to answer

- How well does my program make progress over time?
- When/where does my program suffer from load imbalances and why?
- Why is the time spent in communication or synchronization higher than expected?
- Are I/O operations delaying my program?
- Does my hybrid program interplay well with the given accelerator?

Workflow

Before using Vampir, an application program needs to be instrumented and executed with Score-P. Running the instrumented program produces a bundle of trace files in OTF2-format with an anchor file called traces.otf2. When opening the anchor file with Vampir, a timeline thumbnail of the data is presented. This thumbnail allows to select a subset or the total data volume for a detailed inspection. The program behavoir over time is presented to the user in an interactive chart called Master Timeline. Further charts with different analysis focus can be added.

Platform support

IBM Blue Gene/Q, IBM AIX (x86_64,POWER6), Cray XE/XK/XC, SGI UV/ICE/Altix, Linux clusters (x86/x86_64), Windows, Apple Mac OS X.

License

Commercial

Web page

`http://www.vampir.eu`

Contact

service@vampir.eu

After a trace file has been loaded by Vampir, the Trace View window opens with a default set of charts as depicted in Figure 19. The charts can be divided into timeline charts and statistical charts. Timeline charts (left) show detailed event based information for arbitrary time intervals while statistical charts (right) reveal accumulated measures which were computed from the corresponding event data. An overview of the phases of the entire program run is given in the Zoom Toolbar (top right), which can also be used to zoom and shift to the program phases of interest.

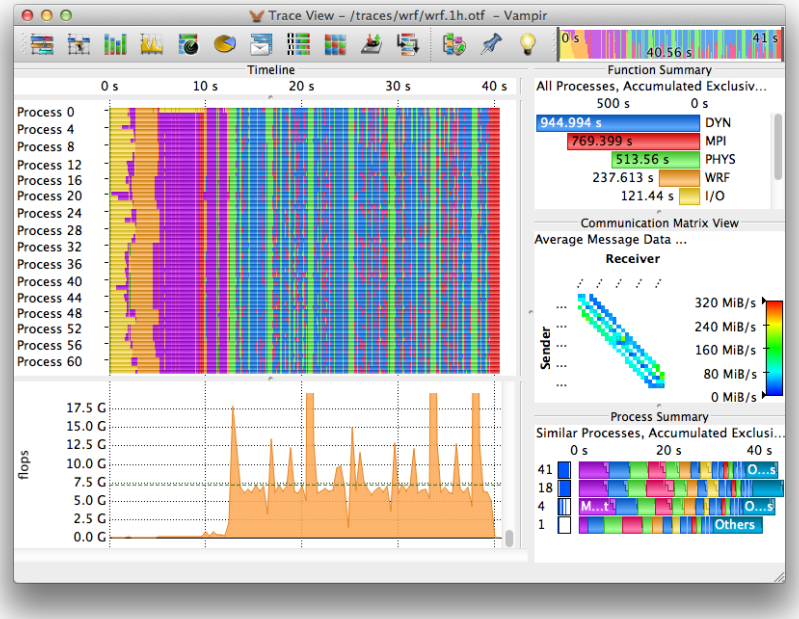


Figure 19: A trace file in the Vampir performance browser.

VI-HPS training

Next to the development of state-of-the-art productivity tools for high-performance computing, the VI-HPS also provides training in the effective application of these tools. Workshops and tutorials are orchestrated in close collaboration of the host organization to fit the particular need of the audience.

Training events can be a tuning workshop, a custom workshop or course, or a tutorial conducted in collaboration with an HPC-related conference. Sign up to the VI-HPS news mailing list via our website to receive announcements of upcoming training events.

Tuning workshop series VI-HPS Tuning Workshops are the major training vehicle where up to 30 participants receive instruction and guidance applying VI-HPS tools to their own parallel application codes, along with advice for potential corrections and optimizations. Feedback to tools developers also helps direct tools development to user needs, as well as improve tool documentation and ease of use. These workshops of three to five days at HPC centres occur several times per year, and feature a variety of VI-HPS tools.

Other training events VI-HPS Tuning Workshops are complemented by additional courses and tutorials at conferences, seasonal schools and other invited training events which have taken place on four continents. Training events of individual VI-HPS partners can also be found on their own websites.

Course material Coordinated tools training material is available with emphasis on hands-on exercises using VI-HPS tools individually and interoperably. Exercises with example MPI+OpenMP parallel applications can be configured to run on dedicated HPC compute resources or within the virtual environment provided by a free Linux Live ISO that can be booted and run on an x86_64 notebook or desktop computer.

Linux Live-ISO The downloadable VI-HPS Linux Live-ISO image provides a typical HPC development environment for MPI and OpenMP containing the VI-HPS tools. Once booted, the running system provides the GNU Compiler Collection (including support for OpenMP multithreading) and OpenMPI message-passing library, along with a variety of parallel debugging, correctness checking and performance analysis tools.

The latest ISO/OVA files are currently only available as 64-bit versions, requiring a 64-bit x86-based processor and a 64-bit OS if running a virtual machine. Depending on available memory, it should be possible to apply the provided tools and run small-scale parallel programs (e.g., 16 MPI processes or OpenMP threads). When the available processors are over-subscribed, however, measured execution performance will not be representative of dedicated HPC compute resources. Sample measurements and analyses of example and real applications from a variety of HPC systems (many at large scale) are therefore provided for examination and investigation of actual execution performance issues.

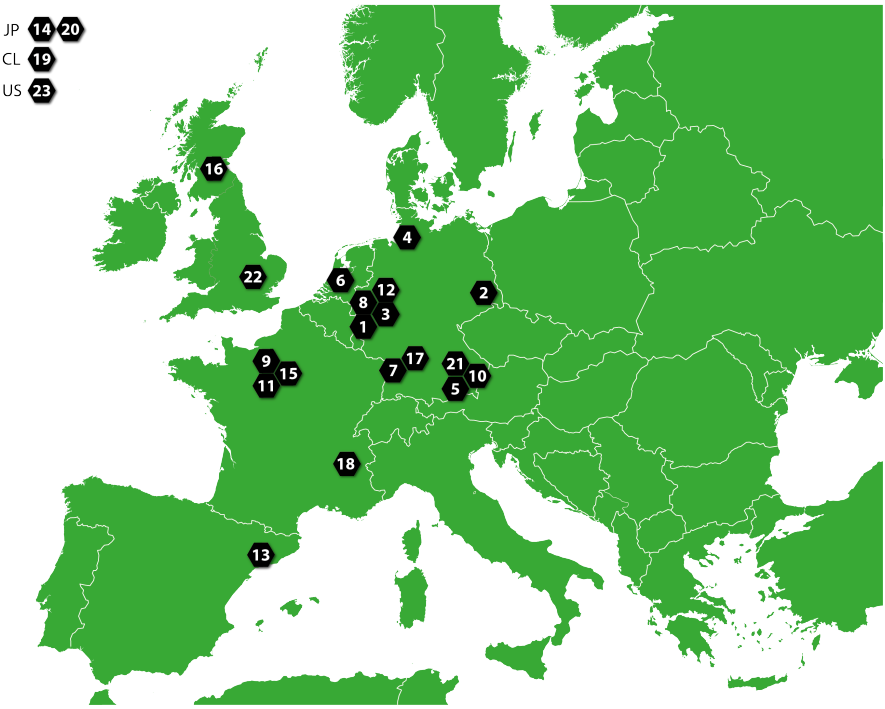
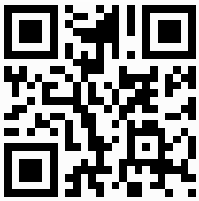


Figure 20: VI-HPS Tuning Workshop locations (2008–2016).

VI-HPS Tools Guide

The Virtual Institute – High Productivity Supercomputing (VI-HPS) aims at improving the quality and accelerating the development process of complex simulation codes in science and engineering that are being designed to run on highly-parallel HPC computer systems. For this purpose, the partners of VI-HPS are developing integrated state-of-the-art programming tools for high-performance computing that assist programmers in diagnosing programming errors and optimizing the performance of their applications.

This VI-HPS Tools Guide provides a brief overview of the technologies and tools developed by the twelve partner institutions of the VI-HPS. It is intended to assist developers of simulation codes in deciding which of the tools of the VI-HPS portfolio is best suited to address their needs with respect to debugging, correctness checking, and performance analysis.



www.vi-hps.org | info@vi-hps.org