Introduction

The mission of the Virtual Institute - High Productivity Supercomputing (VI-HPS) 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 fourteen 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, parallelization, 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.

**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).

**Performance vs. Debugging vs. Correctness vs. Workflow:** *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

---

1. https://www.vi-hps.org
right at runtime. Finally, *workflow* tools help to automate certain tasks and workflows and thus improve overall productivity.

**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*, *HIP*, *OpenCL* and *OpenACC* targeting accelerators, as well as the partitioned global address space (PGAS) languages/libraries *UPC*, *SHMEM*, and *GASPI*. However, it may be possible that a tool supports additional programming models, which will then be indicated in the tool description.

**Languages:** 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++*, *Fortran*, and *Python*. Again, it may be possible that tools support further languages or are even language-independent, which will then be mentioned in the description.

**Processor architectures:** Finally, tools may support only certain CPU architectures and/or offer support for GPUs. Others are essentially CPU architecture agnostic, however, may not have tested support for all architectures. Here the most common architectural families are distinguished, and details of variants (such as 32-bit vs 64-bit) may be found in the accompanying text. *x86* includes Intel Xeon Phi (MIC) and AMD x86-compatible processors, *Power* includes PowerPC, and *GPU* covers attached general-purpose graphical processing unit devices from Nvidia, AMD and others. Not all variants within these families may be supported, and additional processor architectures may be mentioned in the description.
**Archer**

Archer is a data race detector for OpenMP programs. It 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. Starting with LLVM/10, the core functionality of Archer is delivered with LLVM release builds.

**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 -fsanitize=thread -fopenmp example.c -o example
clang++ -fsanitize=thread -fopenmp example.cpp -o example
gfortan -fsanitize=thread -fopenmp -lomp example.f -o example
```

**Platform support**

Linux x86_64, IBM Power

Depends on LLVM/clang

**License**

Apache License 2.0 (LLVM)

**Web page**

https://github.com/pruners/archer
Figure 2 gives detailed information for a data race detected by Archer in the source code displayed in Figure 1.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int a = 0;

    #pragma omp parallel
    {
        // Unprotected read
        if (a < 100) {
            // Critical section
            #pragma omp critical
            {
                // Shared memory access
                a++;
            }
        }
    }
}
```

Figure 1: 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 2: Archer output for the data race.
Caliper

Caliper is a performance analysis toolbox in a library. Caliper can be used for lightweight always-on profiling, such as printing performance reports for application logs. In addition, Caliper supports advanced MPI, tracing, call-stack sampling, I/O, memory, CUDA, and hardware counter analyses. Caliper region annotations can also be forwarded to third-party tools, such as Allinea MAP, TAU, Intel VTune, and NVidia Visual Profiler.

Typical questions Caliper helps to answer

- How much time does each program region take? How much time is spent in MPI or CUDA calls?
- How much memory and I/O bandwidth is used in each region?
- How does performance differ with different program inputs?

Workflow

Mark code regions of interest with Caliper’s source-code annotation macros. Optionally, create a ConfigManager object at the start of the program, which provides access to Caliper’s built-in measurement configurations through a short configuration string. This string can be hard-coded or provided by the user, for example as an application command-line argument. Alternatively, custom measurement configurations can be provided through environment variables or configuration files.

Caliper can aggregate measurement results on-the-fly, both within processes and across MPI ranks, and write out results in human-readable text form using a hierarchical or flat table layout. Alternatively, data can be written to disk in Caliper’s native .cali format or various JSON formats for additional post-processing.

Caliper’s annotation macros are designed to be permanently integrated in the target codes to enable lightweight, always-on performance profiling. Annotations are extremely flexible - in addition to source code regions, developers can add custom key:value attributes to describe domain-specific concepts. Moreover, Caliper can record run metadata, such as the system environment or program configuration, to simplify performance comparisons across multiple program runs.
Platform support
Any POSIX compatible OS. C, C++, and Fortran codes.

License
Modified BSD license.

Web page
https://www.github.com/LLNL/Caliper

Contact
https://github.com/LLNL/Caliper/issues

Figure 3 shows Caliper printing a performance report for a serial execution of a Caliper-enabled program with the inclusive and exclusive time as well as the memory high-water mark for each annotated region. This example uses Caliper’s ConfigManager API and the configuration string given in the -P command-line option to control the performance measurement.

```
$ ./lulesh2.0 -q -i 10 -P runtime-report,mem.highwatermark,aggregate_across_ranks=false
Path Inclusive time Exclusive time Time % Max Alloc’d Mem
main 0.191317 0.009027 4.62802 8752390.00000
  lulesh_cycle 0.18290 0.000045 0.023071 8745749.00000
    LagrangeLeapFrog 0.182153 0.000046 0.023584 8745749.00000
      CalcTimeConstraintsForElems 0.002079 0.000046 0.005875 8747829.00000
      LagrangeElements 0.005915 0.0000214 0.109715 8747829.00000
        ApplyMaterialPropertiesForElems 0.064767 0.000046 0.254292 8963829.00000
        EvalEOSForElems 0.064271 0.018243 9.352938 9909669.00000
      CalcEnergyForElems 0.046028 0.046028 23.597931 9777229.00000
      CalcQOfElems 0.009755 0.005823 2.985373 10173429.00000
      CalcMonotonicQForElems 0.009392 0.003392 2.015883 10173429.00000
      CalcLagrangeElements 0.011179 0.000579 0.296845 9395829.00000
      CalcKinematicsForElems 0.018600 0.016000 5.434476 9395829.00000
     LagrangeNodal 0.094113 0.001768 0.986438 8747829.00000
       CalcForceForNodes 0.092345 0.000717 0.357596 8747829.00000
     CalcVolumeForceForElems 0.091628 0.000996 0.510636 9611829.00000
     CalcHourglassControlForElems 0.067312 0.037134 19.038098 19979829.00000
      CalcFBiHourglassForceForElems 0.030178 0.030178 15.471051 25163829.00000
     IntegrateStressForElems 0.023320 0.023320 11.955847 14795829.00000
     TimeIncrement 0.000092 0.000092 0.047157 8745757.00000
```

Figure 3: Printing a runtime report in a Caliper-enabled program.
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 3.14, this includes Linux on x86/x86_64, Power, ARM, MIPS).

License

GNU General Public License (GPL) v2
Figure 4: KCachegrind showing results from a Callgrind simulation run.

Figure 4 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).
**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), macOS (x86_64), Windows 10;

*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**

[https://www.scalasca.org](https://www.scalasca.org)
Figure 5 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.
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 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 (x86/x86_64, ARM, Power), SGI Altix, Fujitsu FX10/100, Cray XT, IBM Blue Gene, Intel Xeon Phi

License
GNU Lesser General Public License (LGPL) v2.1

Web page
http://tools.bsc.es/dimemas

Contact
tools@bsc.es

Figure 6 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 demands less bandwidth (256MB/s) than the ARW version.

Figure 6: Dimemas sensitivity analysis to network bandwidth.
DiscoPoP

DiscoPoP is a tool that helps software developers parallelize their programs with threads. It discovers potential parallelism in a sequential program and makes recommendations on how to exploit it using OpenMP. Because a compiler does not know the precise value of pointers and array indices computed at runtime, it may assume parallelism-preventing data dependences in places where they would never occur in practice. As a result, automatic parallelization becomes too conservative.

With our parallelism discovery tool DiscoPoP, we aim to circumvent this problem. We abandon the idea of fully automatic parallelization and instead, point the programmer to likely parallelization opportunities that we identify via a combination of static and dynamic dependence analysis. In this way, we consider only data dependences that actually occur. From these dynamic dependences, we derive possible parallel design patterns, which we propose to the programmers to parallelize their programs.

Typical questions DiscoPoP helps to answer

- Is there potential parallelism in my program?
- If yes, which parts of my program can I parallelize?
- How can I parallelize them?

Workflow

Figure 7 shows a high-level overview of DiscoPoP and how it finds parallelization opportunities.

DiscoPoP is built on top of LLVM and achieves its goals in four steps: the decomposition of the program into parts with negligible internal parallelism, called computational units, the identification of data dependences among those units, the selection of parallel design patterns, and finally the suggestion of suitable OpenMP parallelization constructs and data-sharing clauses to the programmer. To find data dependences, the tool instruments all memory accesses and control regions. The instrumented application is then
executed on actual hardware, and profiling data generated by the instrumented code is analyzed on-the-fly to find data dependences among the computational units. Based on the resulting dependence graph, DiscoPoP discovers parallelism in terms of parallel design patterns, including pipeline, doall, geometric decomposition, reduction, and task parallelism. Finally, it issues recommendations on how to parallelize the program using OpenMP. Figure 8 shows the parallelization of an example program using DiscoPoP. Because DiscoPoP relies on dynamic information, users still need to validate the final result - an effort that is, however, not greater than with manual parallelization.

Platform support

Linux x86_64, depends on LLVM/Clang

License

BSD 3-Clause License

Web page

[https://www.discopop.tu-darmstadt.de/](https://www.discopop.tu-darmstadt.de/)
[https://github.com/discopop-project/discopop](https://github.com/discopop-project/discopop)

Contact

discopop-support@lists.parallel.informatik.tu-darmstadt.de

```c
1 void kernel_jacobi_2d(i1
2   for (t = 0; t < TSTEPS; t++)
3       for (i = 1; i < N - 1; i++)
4           for (j = 1; j < N - 1; j++)
5               B[i][j] = 0.2 * (A[i+1][j] + A[i-1][j] +
6                      A[i][j+1] + A[i][j-1]) +
7                      A[i+1][j] + A[i-1][j]);
8   
9   for (i = 1; i < N - 1; i++)
10      for (j = 1; j < N - 1; j++)
11         A[i][j] = 0.2 * (B[i+1][j] + B[i-1][j] +
12                        B[i][j+1] + B[i][j-1]) +
13                        B[i+1][j] + B[i-1][j]);
14 }
```

Line 3 (doall pattern):
`#pragma omp parallel for\private(i,j) shared(B) firstprivate(N,A)`

Line 10 (doall pattern):
`#pragma omp parallel for\private(i,j) shared(A) firstprivate(N,B)`

Parallelization suggestions

Sequential program

Figure 8: The automation of the parallelization process with DiscoPoP.
Extrae

The Extrae measurement infrastructure is an easy-to-use tool for event tracing and online analysis. It uses different interposition mechanisms to inject probes into the target application gathering information regarding the application performance. Most of these mechanisms work directly with the production binary, not requiring any special compilation or linking.

Extrae is the instrumentation tool for Paraver and Dimemas and supports a wide range of HPC platforms and programming models and languages.

Typical questions Extrae helps to answer

- How much time is spent in the parallel runtimes?
- What is the average IPC achieved?
- What is the location in the source code of a given MPI call?

Workflow

Instrumenting with Extrae the production binary only requires to modify few lines of the execution script. The execution command of the program to analyze has to be preceded by a launcher script (namely trace.sh). This script contains just a few definitions to load and configure the Extrae tool. Users only need to specify:

1. where is Extrae installed (EXTRAE_HOME);
2. which information will be captured (EXTRAE_CONFIG_FILE); and
3. the Extrae tracing library (LD_PRELOAD). Please select the proper library depending on the type of parallel application (MPI, OpenMP, OmpSs, Pthreads, CUDA, OpenACC, OpenCL, GASPI, or hybrid combinations).

Once the trace is collected, it is ready to be analysed with Paraver.

Platform support

Linux (x86/x86_64, ARM, RISC-V, Power), SGI Altix, Fujitsu FX10/100, Cray XT, IBM Blue Gene, Intel Xeon Phi, GPU (CUDA, OpenCL)

License

GNU Lesser General Public License (LGPL) v2.1
Figure 9 illustrates two basic examples of how to use the Extrae instrumentation package to generate a Paraver trace for MPI (9(a)) and OpenMP (9(b)) applications. For further reference, please refer to the Extrae's user guide:


```
#!/usr/bin/env sh

export EXTRAЕ_HOME=/extrae/installation/root
export EXTRAЕ_CONFIG_FILE=extrae-config.xml
source ${EXTRAЕ_HOME}/etc/extrae.sh
export LD_PRELOAD=${EXTRAЕ_HOME}/lib/libmpitrace.so

> mpirun ./trace.sh <mpi_binary> <args>
```

(a) MPI applications

```
#!/usr/bin/env sh

export EXTRAЕ_HOME=/extrae/installation/root
export EXTRAЕ_CONFIG_FILE=extrae-config.xml
source ${EXTRAЕ_HOME}/etc/extrae.sh
export LD_PRELOAD=${EXTRAЕ_HOME}/lib/libomptrace.so

> ./trace.sh <omp_binary> <args>
```

(b) OpenMP applications

Figure 9: Basic examples to activate Extrae
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, as well as combinations of multiple parameters.

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 generates performance models 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. The Extra-P GUI provides the means to visualize, browse, and manipulate the resulting models. Detailed textual results are also generated by Extra-P for the in-depth analysis of sensitive code regions.
Platform support

Extra-P is platform independent. It requires only a working Python installation (≥3.7) as it is installed via pip.

License

BSD 3-Clause License

Web page

https://github.com/extra-p/extrap

Contact

extra-p-support@lists.parallel.informatik.tu-darmstadt.de

Figure 10 shows performance models as generated for different call paths in Kripke, an open-source 3D Sn deterministic particle transport code. The performance models are functions of number of processes $p$, the number of direction-sets $d$, and the number of energy groups $g$. 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.

Figure 10: Interactive exploration of performance models in Extra-P.
**JUBE**

The JUBE environment provides a script-based application and platform independent framework, which allows the creation and parametrisation of an automatic workflow execution to be used in benchmark, test or production scenarios.

**Typical questions JUBE helps to answer**

- How to run my application in a reproducible way?
- How to easily create a parameter study for my application?
- How to parametrise the different parts of my application from a single point?

**Workflow**

JUBE is a Python-based tool which is configured using XML files. Within these input files an application workflow is based on different steps, where dependencies and related files can be configured. For program execution JUBE uses normal Linux shell commands, which allows developers to keep their existing mechanism to start or manage applications.

In addition JUBE allows a flexible way to specify parameters, which can be used to control the compilation of the application, its runtime arguments and behaviour, or the job execution environment.

After program execution, JUBE can also run post-processing tools or scan any ASCII-based program output to extract useful information like timing information or performance data. This information is gathered and displayed in a combined output form together with the selected parametrisation.

**Platform support**

Linux x86_64, (Python2.6, Python2.7, Python3.2 or any newer version)

**License**

GPLv3
Figure 11 shows an example of the command-line interface used to control the JUBE execution. Each individual run is stored separately, with a unique identifier, in the filesystem to allow reproducibility and easier data exchange.

![Command-line view of a JUBE-based benchmark execution.](image)

Figure 11: Command-line view of a JUBE-based benchmark execution.
LIKWID

LIKWID is a tool suite for performance-oriented programmers offering command line tools for system topology, CPU/task affinity, hardware performance monitoring, micro-benchmarking and more. Besides the command line tools, almost all functionality is provided as a C library to be integrable in other tools.

Typical questions LIKWID helps to answer

- How does my system look like? How many threads are available?
- How well does my code exploit the provided hardware features?
- How to measure typical performance metrics (like floating-point operations, memory bandwidth or performance per watt) for my application
- How to reduce performance variation and control the placement of my software threads?
- How can I benchmark my code with different CPU frequencies?
- How does my code behave when hardware prefetcher X is disabled?
- How to run my application on X nodes and measure the metric Y for all processes?

Workflow

When freshly accessing a new machine, you want to get the system topology (likwid-topology) to determine the number of threads and the hardware threads (CPU cores) the application should run on (likwid-pin). If you have an MPI application, determine the affinity strategy once and run your MPI+X application (likwid-mpirun). What is the metric X for my whole application run or how does it evolve over time (likwid-perfctr)? If you want to measure loop(s) or routine(s) running on CPUs or GPUs, add MarkerAPI instrumentation to your code once and control the measurement from the command line.

You want to get an impression how well a feature improves your performance? Write a small benchmark in assembly (to avoid compiler "optimizations") and run it in a controlled environment.
### Platform support

x86/x86_64 (Intel & AMD), ARM8 (Marvell Thunder X2, Fujitsu A64FX, AWS Graviton2), POWER (POWER8 and POWER9) and Nvidia/AMD GPUs under the Linux OS.

### License

GPLv3

### Web page

- [https://hpc.fau.de/research/tools/likwid/](https://hpc.fau.de/research/tools/likwid/)
- [https://github.com/RRZE-HPC/likwid/wiki](https://github.com/RRZE-HPC/likwid/wiki)

Python interface: [https://github.com/RRZE-HPC/pylikwid](https://github.com/RRZE-HPC/pylikwid)

Julia interface: [https://github.com/JuliaPerf/LIKWID.jl](https://github.com/JuliaPerf/LIKWID.jl)

### Contact

rrze-likwid@fau.de or matrix.org chat

---

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>2.4796</td>
<td>2.4796</td>
<td>2.4796</td>
<td>2.4796</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>2.2949</td>
<td>2.2932</td>
<td>2.2868</td>
<td>2.2935</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>3392.1738</td>
<td>3392.1734</td>
<td>3392.1724</td>
<td>3392.1754</td>
</tr>
<tr>
<td>CPI</td>
<td>5.1572</td>
<td>5.0843</td>
<td>5.1161</td>
<td>5.1234</td>
</tr>
<tr>
<td>Load to store ratio</td>
<td>1.8224</td>
<td>1.8419</td>
<td>1.8332</td>
<td>1.8309</td>
</tr>
<tr>
<td>Load ratio</td>
<td>0.2352</td>
<td>0.2313</td>
<td>0.2329</td>
<td>0.2335</td>
</tr>
<tr>
<td>Store ratio</td>
<td>0.1291</td>
<td>0.1256</td>
<td>0.1270</td>
<td>0.1275</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s] STAT</td>
<td>9.9184</td>
<td>2.4796</td>
<td>2.4796</td>
<td>2.4796</td>
</tr>
<tr>
<td>Runtime unhalted [s] STAT</td>
<td>9.1684</td>
<td>2.2868</td>
<td>2.2949</td>
<td>2.2921</td>
</tr>
<tr>
<td>Clock [MHz] STAT</td>
<td>33568.695</td>
<td>3392.1724</td>
<td>3392.1754</td>
<td>3392.1737</td>
</tr>
<tr>
<td>CPI STAT</td>
<td>20.4810</td>
<td>5.0843</td>
<td>5.1572</td>
<td>5.1262</td>
</tr>
<tr>
<td>Load to store ratio STAT</td>
<td>7.3284</td>
<td>1.8224</td>
<td>1.8419</td>
<td>1.8321</td>
</tr>
<tr>
<td>Load ratio STAT</td>
<td>0.9329</td>
<td>0.2313</td>
<td>0.2352</td>
<td>0.2332</td>
</tr>
<tr>
<td>Store ratio STAT</td>
<td>0.5692</td>
<td>0.1256</td>
<td>0.1291</td>
<td>0.1273</td>
</tr>
</tbody>
</table>

Figure 12: Measurement of the load-store-ratio (DATA performance group) of an application running on four CPU cores (0−3). The first table lists metrics for each CPU core while the second table contains statistics of the per-core measurements.
Linaro DDT

Linaro DDT is a modern and easy to use parallel debugger widely used by software developers and computational scientists in industry, academia and government research. Its interface simplifies concurrency and is highly responsive even at extreme scale.

The tool is part of Linaro Forge, a development solution that includes both debugging and profiling capabilities.

**Typical questions Linaro 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**

Linaro 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.

Thanks to its "reverse connect" capability, Linaro DDT can be very easily attached to an application submitted via the batch scheduler. 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**

Any Linux running on aarch64, x86_64, and Nvidia GPUs.

**License**

Commercial
Figure 13: Linaro DDT parallel debugging session showing multi-dimensional array viewer.
**Linaro MAP**

Linaro 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 is part of Linaro Forge, a development solution that includes both debugging and profiling capabilities.

**Typical questions Linaro 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 Linaro 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**

Any Linux running on aarch64, x86_64 and Nvidia GPUs.

**License**

Commercial

**Web page**

https://www.linaroforge.com

**Contact**

https://www.linaro.org/support#for-linaro-forge4
Figure 14: Linaro MAP parallel profiling session showing execution hotspots and evolution charts.
VI-HPS Tools Overview

**Optimization**
- Visual trace analysis
- Automatic profile & trace analysis

**Debugging, error & anomaly detection**
- Hardware monitoring

**Execution**
- CALIPER / LIKWID / MAP/PR / MPIP / O|SS / MAQAO
- PAPI
- KCACHEGRIND
- MUST / ARCHER
- DDT
- STAT
- LIKWID / MEMCHECKER / SPINDLE / SIONLIB
- JUBE
Linaro Performance Reports

Linaro 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.

The tool is part of Linaro Forge, a development solution that includes both debugging and profiling capabilities.

Typical questions Linaro Performance helps to answer

• What can I do to improve the efficiency of my application?
• What system or usage changes could I make to improve performance?
• How does the underlying hardware impact the performance of my applications?

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

Any Linux running on aarch64, x86_64, and Nvidia GPUs.

License

Commercial

Web page

https://www.linaroforge.com
Figure 15: Linaro Performance Reports single page report of an application’s CPU, MPI, I/O and Memory usage.
MAQAO

MAQAO (Modular Assembly Quality Analyzer and Optimizer) is a core performance centric binary analysis and optimization framework operating on executable binary applications (no recompilation necessary). The tool applies dynamic and static analyses on extracted/reconstructed high level structures - loops and functions - to guide application developers through optimization by providing synthetic and human-friendly reports and hints.

MAQAO was designed as an extensible framework allowing users to easily develop analysis and instrumentation modules using a Lua API, and it comes with three ready-to-use modules: LProf (Lightweight Profiler), a sampling-based profiler that provides a list of hot spots (loops and functions) collected during program execution; CQA (Code Quality Analyzer), a static analyzer that uses a machine model to evaluate the assembly code quality of target loops and functions by using key metrics such as arithmetic units usage and vector length usage; ONE View, a module that invokes LProf and CQA and aggregates their results in order to build a human-friendly report that spans a large set of key performance elements.

Typical questions MAQAO helps to answer

- Which functions and loops are the most profitable to optimize?
- What optimizations will be beneficial to a loop and how much gain can be expected?
- Is a specific loop or function compute-bound or memory-bound?
- How much gain improved parallelization can provide?

Workflow

The analysis process is driven by ONE View, using a configuration file containing the parameters necessary to launch the target application and run the analysis modules. Once the analysis is complete, ONE View generates a human-friendly synthetic HTML report.

This report includes a list of hot spots categorized by their origin: parallel runtime, I/O, memory, main code, etc. For each hot spot, the report provides a description of potential issues, an estimation of the impact on overall application performance, and hints on how to improve performance through compiler directives or implementation tweaks. MAQAO can also generate comparison reports between multiple runs of an application with different datasets or parallel execution parameters.
Platform support
Linux clusters (Intel 64 and Xeon Phi)

License
GNU Lesser General Public License (LGPL) v3

Web page
http://www.maqao.org

Contact
contact@maqao.org

Figure 16 presents views from the HTML output generated by ONE View: (16(a)) an overview of the application performance and estimated achievable speedups, (16(b)) a comparison between multiple runs with different parallel execution parameters, (16(c)) the list of functions and loops hot spots, (16(d)) code quality hints and associated relative potential gains.

(a) Global view

(b) Multi-runs comparison (scalability mode)

(c) Functions profile

(d) Code quality report

Figure 16: Some of MAQAO ONE View HTML outputs.
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
Figure 17 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 17: Visualization of a deadlock situation.
OSACA

The Open-Source Architecture Code Analyzer (OSACA) is a Python-based static analysis tool for in-core performance prediction of inner-most assembly loops. It is available both as command line application and as tool within the Compiler Explorer and provides the user with a throughput prediction of their code in steady state, a critical path analysis and an across-loops-dependency analysis. Besides its provided machine models for several x86 and ARM micro-architectures, it is designed in a way so that the user can easily add its own machine models for architectural exploration and integrate it in their own workflow via a Python API.

Typical questions OSACA helps to answer

- How fast can my code possibly run (when all data is in L1 cache)?
- What optimizations will be beneficial for my code?
- What is a more realistic roofline for my target code region?
- What are the bottlenecks of my code?
- Why does the code of compiler A perform better than the code of compiler B?

Workflow

Having the assembly code of your target high-level code, identify the region of interest and mark it with specific OSACA comment markers (either manually or with the built-in -insert-marker feature) or reduce the code to the specific section with the -lines option. Start the analysis by running OSACA and specify your target hardware with -arch. You will receive a light-speed throughput prediction showing a perfect scheduling based on the machine model, the critical path (CP) of your code, and all loop-carried dependencies (LCD). As a rule of thumb for steady-state loop kernels, the runtime prediction is the maximum of the most-occupied port and the LCD. Furthermore, you can generate a .dot graph file to visualize your dependencies in your assembly loop.

Platform support

The CLI runs under any platform with a Python 3 installation. OSACA currently comes with support for various micro-architectures, including...
Intel x86_64 (Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake X, Cascade Lake X, Ice Lake Server), AMD x86_x86 (Zen 1, Zen 2, Zen 3), and ARM AArch64 (Marvell ThunderX2, ARM Neoverse N1, ARM Cortex A72, Fujitsu A64FX, HiSilicon TaiShan v110).

License

GNU Affero General Public License v3.0 (AGPL-3.0)

Web page

https://github.com/RRZE-HPC/OSACA
https://github.com/RRZE-HPC/OSACA/wiki
Python ReadTheDocs: https://osaca.readthedocs.io/en/latest/
https://godbolt.org/

Contact

nhr-osaca@fau.de or OSACA Github Issues

Figure 18: OSACA analysis of the Gauss-Seidel method, compiled on and analyzed for an Intel Ice Lake Server system. The predicted runtime would be 56 cy per assembly loop.
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**


**Contact**

oss-questions@krellinst.org

![Figure 19: GUI showing the results of a sampling experiment (left: time per statement, right: information mapped to source)](image-url)
**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).
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.
**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: The timeline view displays the application behavior over 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 perform 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 (x86/x86_64, ARM, Power), SGI Altix, Fujitsu FX10/100, Cray XT, IBM Blue Gene, Intel Xeon Phi, Windows, macOS

License

GNU Lesser General Public License (LGPL) v2.1

Web page

http://tools.bsc.es/paraver

Contact

tools@bsc.es

Figure 20 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 20: Paraver histogram of the computation phases colored with the cluster ID.
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

Continuously tested on: HPE Cray XC and EX, various Linux (Intel, AMD, IBM, ARM) clusters with GNU, Intel, NVIDIA, IBM, and AMD compilers.
Previously tested on: Intel Xeon Phi (KNL), IBM Blue Gene/Q, Cray XT/XE/XK, Fujitsu FX systems
Figure 21 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 21: Scalasca automatic trace analysis identification of time in message-passing wait states and on the critical path.
**Score-P**

The Score-P measurement infrastructure is a highly scalable and easy-to-use tool suite for profiling and event tracing. 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 Extra-P.

**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 and serve as Extra-P input. Event traces can be examined in Vampir or used for automatic bottleneck analysis with Scalasca.

**Platform support**

Continuously tested on: HPE Cray XC and EX, various Linux (Intel, AMD, IBM, ARM) clusters with GNU, Intel, NVIDIA, IBM, and AMD compilers.

Previously tested on: Intel Xeon Phi, IBM Blue Gene/Q, Cray XT/XE/XK, Fujitsu FX systems

**License**

BSD 3-Clause License
Figure 22 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.
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 23). 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
Figure 23 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 23: STAT 3D spatial-temporal call prefix tree of stalled execution.
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, macOS.

**Supported Runtime Layers**

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

**License**

BSD style license

**Web page**

http://tau.uoregon.edu

**Contact**

tau-bugs@cs.uoregon.edu

Figure 24 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 24: TAU’s ParaProf 3D profile browser shows the exclusive time spent (height, color) over ranks for all routines in a code.
**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 behavior 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**

Linux (x86/x86_64, ARM, Power), Windows, Apple macOS.

**License**

Commercial
After a trace file has been loaded by Vampir, the Trace View window opens with a default set of charts as depicted in Figure 25. 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 25: 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 26: VI-HPS Tuning Workshop locations (2008–2022).
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 fourteen 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.