# Performance capture and analysis for applications and workflows

State-of-the-art report

Line Pouchard, Abid Malik, Kerstin Kleese Van Dam, Huub Van Dam, Wei Xu
Brookhaven National Laboratory

June 19, 2017

CODAR, the Center for Online Data Analysis and Reduction, is a DOE Exascale Computing Project (ECP) Co-design Center focused on providing the needed infrastructure for efficient and scalable online data reduction and analysis for present and future scientific applications of interest to the DOE and others, including NWChemEx, Lattice Quantum Chromodynamics (QCD), Fusion, and others. Limitations in the I/O infrastructure of the expected exascale architectures, drive a significant paradigm shift for many of these codes from post hoc result analysis to online data reduction and analysis. In designing these new complex simulation and analysis workflows, many questions arise in terms of the impact of online analysis on the overall simulation performance, the ideal design and placement of data reduction and analysis tasks on the systems, etc. In this context CODAR will not only provide the required online reduction and analysis infrastructure, but also the tool to assess the performance of proposed solutions and quantify potential trade-offs. One of the key capabilities needed here are tools that can capture and analyse performance metrics for the newly designed workflows not only for single runs, but also for empirical performance studies and performance evolution over time.

Today there are a plethora of performance measurement tools available, some of which supported by ECP, however the majority of these are focused on performance capture for single application, and are less prepared to deal with specific workflow and data driven aspects of performance. This document presents a set of CODAR developed evaluation criteria, that can be used to assess the suitability of specific performance measurement tools for baseline performance metrics capture and trade off studies for online data reduction and analysis workflows. Furthermore the document presents the results of the evaluation of two well known tools against these criteria: Score-P and TAU.

## Performance Tools Evaluation Criteria

CODAR has a number of potential choices, when it comes to performance tools that can be used to examine the tradeoffs applications are facing in the implementation of their online data reduction and analysis workflows. To aid the decision making process we have a compiled a list of evaluation criteria against which each possible tool can be assessed. The evaluation criteria are based on the requirements identified to date that such a tool would need to satisfy, as well as the potential for future extensions, which might be needed.

To make the evaluation easier, we have tried to group the criteria into different categories.

## Metrics

In this section we describe the performance metrics that need to be captured to enable us to assess the performance of workflows that orchestrate applications and their online data reduction and analysis tasks. While not all the metrics are required to be captured all the time, the possibility needs to exist to identify potential root causes for performance bottlenecks or performance variability.

A critical aspect that we added to the metrics is the concept of time series – frequent snap shots of particular performance parameters such as memory usage over time – that will enable the replay of resource usage over time to aid the performance analysis, in trade-off studies.

In addition to classic performance metrics a number of provenance metrics have been added to the list, they are essential to track performance influencing changes in the computer system architecture and system software, in the workflow applications and use cases through trade-off studies and empirical studies during a development project.

*Table 1: Performance and provenance metrics needed.*

| |
|---|
| Time |
| Execution time for overall serial or parallel scientific workflow |
| Execution time for each application |
| Execution time for each code region |
| Execution time per application per node |
| Idle times, per application, per node, per code region |
| Memory Usage |
| Each Application – min, max, average, time series |
| Each code region – min, max, average, time series |
| Support for programming paradigms |
| Support for parallel programming models and libraries (e.g. MPI, OpenMP, Shmem) |
| Accelerator-based |
| Communication/On system data movement |
| Message Performance – number, message size, execution time, wait time etc. |
| Communication times if mechanisms other than MPI are used e.g. disk, in-memory, message queues, file systemetc. – number, message sizes, time, wait time, lost messages – time series |
| Interconnect overall performance, load overall, application specific load and performance – time series |
| I/O |
| Number and volume of data read/write tasks, execution time, source-destination, max, min, average, time series |
| Load and performance of overall I/O system components, specific load created by application |
| System Architecture Description |
| Processors, memory incl. intermediate cache and solid state, interconnect, I/O subsystem (incl. burst buffer) |
| Application/Workflow Description |
| Names, version (github link), code regions, call trees |
| Names, components, input and output data files incl. version, size, location |

Next to the ability to capture the required performance metrics a number of additional features need to be provided by any tool that support performance analysis for workflows:

- Ability to correlate the metrics from different workflow tasks (that are running in parallel)
- Ability to integrate the metrics across the workflow tasks
- Ability to analyze, visualize and explore the captured workflow performance metrics for single workflows and for pratical or complete studies of workflow performance
- Ability to persist the results of continued performance metrics captures in a searchable archive to enable trade-off and empirical studies
- Performance metrics, in particular when captured at a fine grained level can produce significant amounts of data, too much for a data restricted exascale environment. Therefore it would be necessary for the tool to provide capabilities to selectively capture performance data, as well as online data reduction and analysis capabilities that reduce the performance data to critical events and trends.

## Access and Usability

While it is good to know that a tool has the necessary capabilities, it is equally important to have access to the code and to understand how it works. Therefore we also include criteria related to the SW development cycle of each tool including:

- Available as Open Source
- Available in an Open Repository
- User documentation (usage and examples)
- User support and responsiveness
- Ease of use (i.e. easy to introduce into the code, easy access to output)
- Available at LCFs
- Regularity of updates

## Support for Joint Developments

CODAR itself has some very specific additional requirements, to enable the project to implement its vision. This will require the ability to easily do joint development work with support of the Performance Tool developers. Questions related to these requirements arise:

- Is there an Open Developer Community or a Developer Forum?
- Do Developer Documentation and test cases exist?
- Do Open and documented APIs?
- Open Data Formats? Documentation?
- Access to raw performance data?
- Examples and clear route to add new capabilities?
- Existing online analysis capabilities?
- How will new code from CODAR be integrated into main distribution version?
- How long will it take to be available to users (if all is ok)?

- How often are new releases published?
- How responsive are the developers to questions? How good are the answers? Is it easy to find an expert for a specific feature?
- How are the developers organized?
- Is there a future roadmap that aligns with our requirements?

### Risk Assessment

- What features are available today / how many are future plans or under development?
- How long has the tool been available?
- Do we have reliability figures for the tools?
- How large is the developer team, how committed are they to collaborate with us, how solid is there funding base?TH
- Is there an alternative if things slip?
- How much are we tied in to one product? Is there a single point of failure?

## Evaluation

TAU and Score-P present similar features. Both Score-P and TAU produce similar output files in the form of Event Trace files and Profile files that can be read by performance and analysis tools such as Jumpshot, Paraprof, Vampir and others. The Profile file contains a concise summary of events. The Even Trace contains most details and can be very large.

The differences between the tools are in the instrumentation methods, the ease of use, and the ability to persist data into a data store. In addition, Score-P supports various programming models including the ability to instrument source code run on GPUs, as tested at BNL during the June 5-9, 2017 Hackathon. TAU's latest release enables the instrumentation of scientific workflows as described below.

### Output

Figure 1 present a trace file in text format containing available metrics. Figure 2 presents a profile file. These metrics were obtained by executing the dynamics and analysis concurrently for an NWChem MD calculation in an attempt to simulate a concurrent workflow situation. The MD calculation loops until a trajectory file appears. At that point the analysis workflow is started. Figure 3 presents a Trace file obtained with Score-P instrumenting a small H2O calculation with NWChem.

In Figure 1, EV_INIT represents an init record. One may ignore the 3 parameter. For the other events, 1 represents entry and -1 exit. The size of this Trace file is 812MB, and represents a classical MD run of 320 timesteps on 4 cores as well as the corresponding analysis of 100 timesteps on 1 core. In this case the trace file is represented in a text format, in binary format it is 224MB in size. From start to finish the MD run took 38.0 seconds wall clock time. In addition the program was compiled to suppress the instrumentation of all subroutines apart from the main MD and analysis routines.

```
#  creation program: tau_convert -dump
#     creation date: May-26-2017
#    number records: 9770199
# number processors: 5
# max processor num: 4
#   first timestamp: 1495821922987098
#    last timestamp: 1495821961150021

#=NO= ======================EVENT== ==TIME [us]= =NODE= =THRD= ==PARAMETER=
    1                        "EV_INIT" 1495821922987098      0      0            3
    2             ".TAU application " 1495821922987098      0      0            1
    3 "NWCHEM [{nwchem.F} {1,7}-{397 1495821922987352      0      0            1
    4                   "MPI_Init()  " 1495821922988244      0      0            1
    5                        "EV_INIT" 1495821922992870      1      0            3
    6             ".TAU application " 1495821922992870      1      0            1
    7 "NWCHEM [{nwchem.F} {1,7}-{397 1495821922993156      1      0            1
    8                   "MPI_Init()  " 1495821922994054      1      0            1
    9                        "EV_INIT" 1495821922995017      2      0            3
   10             ".TAU application " 1495821922995017      2      0            1
   11 "NWCHEM [{nwchem.F} {1,7}-{397 1495821922995238      2      0            1
   12                   "MPI_Init()  " 1495821922996048      2      0            1
   13                        "EV_INIT" 1495821923000757      3      0            3
   14             ".TAU application " 1495821923000757      3      0            1
   15 "NWCHEM [{nwchem.F} {1,7}-{397 1495821923000979      3      0            1
   16                   "MPI_Init()  " 1495821923001792      3      0            1
   17                   "MPI_Init()  " 1495821923138067      0      0           -1
   18                     "WALL_CLOCK" 1495821923138119      0      0   1495821923
   19                   "MPI_Init()  " 1495821923138124      2      0           -1
   20                     "WALL_CLOCK" 1495821923138154      2      0   1495821923
   21                   "MPI_Init()  " 1495821923138404      1      0           -1
   22                     "WALL_CLOCK" 1495821923138460      1      0   1495821923
   23                   "MPI_Init()  " 1495821923138508      3      0           -1
   24                     "WALL_CLOCK" 1495821923138550      3      0   1495821923
   25      "TauTraceClockOffsetStart" 1495821923141786      0      0            0
   26      "TauTraceClockOffsetStart" 1495821923141786      2      0            0
   27      "TauTraceClockOffsetStart" 1495821923141855      3      0            0
   28      "TauTraceClockOffsetStart" 1495821923141865      1      0            0
   29        "MPI_Errhandler_set()  " 1495821923141973      0      0            1
   30        "MPI_Errhandler_set()  " 1495821923141981      2      0            1
   31        "MPI_Errhandler_set()  " 1495821923142054      3      0            1
   32        "MPI_Errhandler_set()  " 1495821923142067      1      0            1
   33        "MPI_Errhandler_set()  " 1495821923142095      0      0           -1
   34        "MPI_Errhandler_set()  " 1495821923142095      2      0           -1
   35             "MPI_Comm_size()  " 1495821923142175      2      0            1
   36             "MPI_Comm_size()  " 1495821923142176      0      0            1
   37             "MPI_Comm_size()  " 1495821923142239      2      0           -1
   38             "MPI_Comm_size()  " 1495821923142242      0      0           -1
```

Figure 1: NWChem trace file compiled with TAU shows the header and Enter and Leave timestamps for the MPI calls recorded in the run.

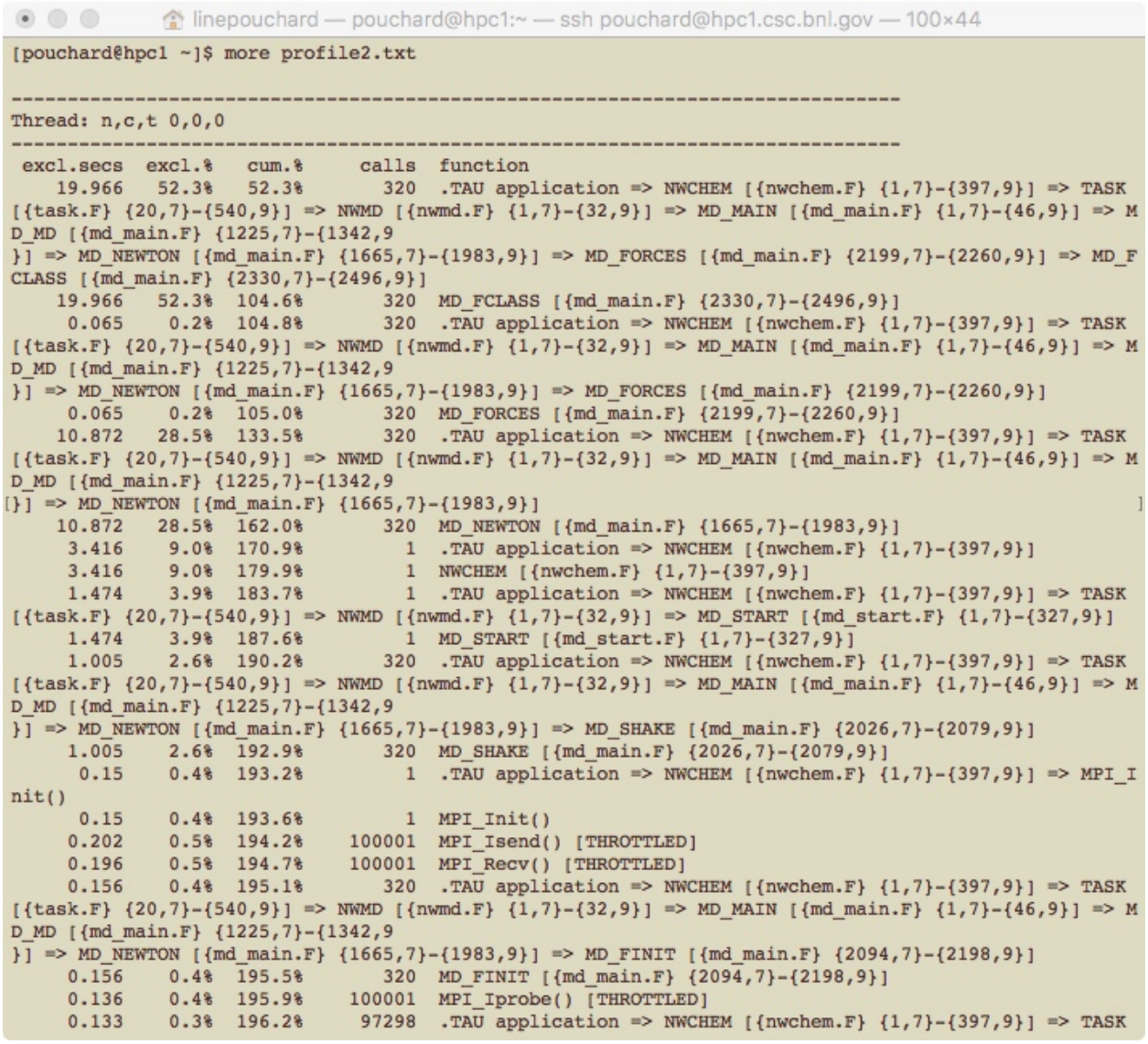


Figure 2: Profile of NWChem workflow produced by TAU. The call tree is visible under the "function" column.

The numbers in bracket {1,7} in Figure 2 represent the lines in the code where this function appears. More specifically it says that the NWCHEM program starts on line 1 at column 7 in nwchem.F (i.e. the beginning of the "program nwchem" statement) and ends on line 397 in column 9 of the same file (i.e. the end of the "end" statement).

~/Documents/CODAR/results/h2o 2/run01/traces.txt

```
=== OTF2-PRINT ===
=== Events ================================================================
Event                                Location            Timestamp  Attributes
------------------------------------------------------------------------------
ENTER                                       3    47695807063997768  Region: "task_gradient" <20>
ENTER                                       1    47695807063998240  Region: "task_gradient" <20>
ENTER                                       2    47695807064033160  Region: "task_gradient" <20>
ENTER                                       0    47695807064057112  Region: "task_gradient" <20>
ENTER                                       2    47695825121291816  Region: "ga_dgemm_" <21>
ENTER                                       0    47695825121341368  Region: "ga_dgemm_" <21>
ENTER                                       1    47695825121506160  Region: "ga_dgemm_" <21>
ENTER                                       3    47695825121549616  Region: "ga_dgemm_" <21>
LEAVE                                       3    47695825236595864  Region: "ga_dgemm_" <21>
LEAVE                                       0    47695825236613096  Region: "ga_dgemm_" <21>
LEAVE                                       2    47695825236723400  Region: "ga_dgemm_" <21>
LEAVE                                       1    47695825236732496  Region: "ga_dgemm_" <21>
ENTER                                       3    47695825264532192  Region: "dft_scf" <22>
ENTER                                       1    47695825264567632  Region: "dft_scf" <22>
ENTER                                       2    47695825264577648  Region: "dft_scf" <22>
ENTER                                       0    47695825264689408  Region: "dft_scf" <22>
ENTER                                       2    47695825468883224  Region: "dft_guessin" <23>
LEAVE                                       2    47695825468914608  Region: "dft_guessin" <23>
```

Figure 3: Trace file obtained with ScoreP for a small NWChem trajectory calculation

Profile and Event Trace can be converted to text format using the command:

```
>paraprof --oss profile.cubex
```

```
>otf2-print traces.otf2
```

Both Score-P and TAU record execution time spent per code regions and where in a call tree time is actually spent. This is an advantage over traditional performance profiling tools such as gprof that records the total time spent in a given call, the number of such calls and outputs an average, thus potentially missing important performance variation per code region.

Both Score-P and TAU record execution time spent in code regions recorded in ENTER/LEAVE counters. The call tree is recorded in Profile.

## Instrumentation

Score-P and TAU differ in their methods for instrumenting scientific code and collecting data.

1) Score-P 3.1

Available from: http://score-p.org and http://www.vi-hps.org/projects/score-p/

Score-P presents flexible measurements without re-compilation which makes it more easily usable when profiling complex scientific codes. A scientific code compiled with ScoreP will collect all available metrics. As data needs to be reduced, a ScoreP filter is added at run time that allows a user to reduce or

filter out metrics to be collected in trace and profile files. If a different set of metrics is needed, only the ScoreP filter needs to be changed.

In our experience, Score-P must be compiled with the `unwind` library to access the call stack information. `scorep-score` does not include call tree information in the profiles it produces.

Timers in Score-P - The timestamps reported by `scorep-info –A trace.otf2` are read from the CPU Time Stamp Counter (TSC) rather than calling `clock_gettime`. The timer that Score-P uses can be changed with the environment variable `SCOREP_TIMER`. A small C code for interpreting score-P timestamps is available here.

To get more detailed descriptions of the configuration variables, one can use the command

```
scorep-info config-vars –full
```

In addition the —full option will in many case also provide the available alternatives in addition to the current settings.

Score-P outputs the name of the metrics it produces along with the data. Score-P groups MPI calls into groups to filter out unwanted groups and reduce the trace using the ScoreP filter.

Details on compiling a ScoreP example are included in Appendix A.

2) TAU

The current version is TAU 2.26.2 available from:

https://www.cs.uoregon.edu/research/tau/downloads.php and http://tau.uoregon.edu/tau

TAU offers several methods for collecting performance data: dynamic instrumentation (sampling), compiler-based instrumentation, and source code instrumentation. Documentation updated March 2017 is available here and shows details about the options ordered by complexity and richness of output. https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch01.html

Dynamic instrumentation is achieved through library pre-loading with `tau_exec`. MPI calls are profiled by default while other options (tracking MPI, io, memory, cuda, opencl library calls) need to be specified on the `tau_exec` command line.

The time-based sampling of processes achieved through dynamic instrumentation is useful as a preliminary step to determine the amount of time spent in a code area. It does not contain instrumentation and the output data may not be useful as sampling may survey calls uninteresting for performance and will not show outliers.

The compiler-based and source code instrumentation options are described here:
https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch01s02.html

Any change in desired metrics such as profiling output metrics, code regions, or other options requires re-compiling the entire application with TAU. For this reason we used source instrumentation when compiling with NWChem. In addition, our experience with LQCD application shows that TAU itself needs to be compiled with the same compiler version and options as the application it profiles in order to extract the desired metrics. This precludes installing a shared version on systems and adds a burden on the scientific user to recompile the appropriate version in their home area.

The compiler-based and source instrumentation options provide more details about the application being profiled but they are also more complex and more verbose. In addition, source code instrumentation requires loading the TAU PDT (Program Database Toolkit), a tool infrastructure for providing access to the high-level interface of source code. PDT v.3.24 was released March 2017. Appendix B gives details on how TAU was compiled with NWChem.

A `-g` compiler flag is needed to output the names of the metrics along of the data (instead of addresses). If TAU does not recognize a compiler variable, it will wait for user input unless the configuration specifies that it will not be given.

Many of TAU's boasted features are not documented and personal emails to the TAU developers are required to understand what is actually measured.

Support for workflows in TAU: TAU started merging trace and profile data produced with running applications in parallel or in serial with the `tau–coalesce` feature in version 2.26.2 (Appendix C). Support for workflows has been tested at BNL for using the NWChem parallel workflow described for Figure 1. This workflow only includes 2 applications running in parallel in **different directories**. At the time of this writing a major issue is that for large workflows with workflow applications required to run in the same directory the tau_coalesce feature overwrites its own output data as it is dumped by multiple applications. See discussion below in the Gaps section.

## Feature comparison

*Table 2: Feature comparison based on Table 1 criteria*

| Criteria | Score-P | TAU |
|---|---|---|
| Time | | |
| Execution time for overall serial or parallel scientific workflow | | Partially supported with tau_coalesce if each workflow application runs into its own directory |
| Execution time for each application | ENTER/LEAVE timestamps<br>Aggregation is provided | ENTER/LEAVE timestamps<br>User must aggregate totals |
| Execution time for each code region | ENTER/LEAVE timestamps<br>User must calculate deltas | ENTER/LEAVE timestamps<br>User must calculate deltas |
| Execution time per application per node | ENTER/LEAVE timestamps<br>User must calculate deltas | ENTER/LEAVE timestamps<br>User must calculate deltas |
| Execution time per application | ENTER/LEAVE timestamps | ENTER/LEAVE timestamps |

| | | |
|---|---|---|
| per node per code region | User must calculate deltas | User must calculate deltas |
| Memory Usage | | |
| Each Application – min, max, average, time series | Y | Y |
| Each code region – min, max, average, time series | Y | Y |
| Programming paradigms | | |
| Support for parallel programming models and libraries (e.g. MPI, OpenMP, Shmem) | MPI, OpenMP, Pthreads | MPI, MPI threads, OpenMP |
| Accelerator-based | CUDA (tested at BNL), OpenCL, OpenACC prototype | CUDA and OpenCLL claimed but untested. |
| Communication/On system data movement | | |
| Message Performance – number, message size, execution time, wait time etc. | User must aggregate total number of messages<br>User must aggregate total time spent in communication<br>User must substract Enter and Leave timestamps for same node | User must aggregate total number of messages<br>User must aggregate total number of time spent in communication<br>User must substract Enter and Leave timestamps for same thread and node |
| Communication times if other mechanisms are used than MPI e.g. disk, in-memory, message queues, file system, etc. – number, message sizes, time, wait time, lost messages – time series | N | N |
| Interconnect overall performance, load overall, application specific load and performance – time series | N | N |
| I/O | | |
| Number and volume of data read/write tasks, execution time, source-destination, max, min, average, time series | N | N |
| Load and performance of overall I/O system components, specific load created by application | N | N |
| System Architecture Description | | |
| Processors, memory incl. intermediate cache and solid state, interconnect, I/O subsystem (incl. burst buffer) | Score-P records some system level information if accessible using the SCOREP_METRIC_PLUGINS and SCOREP_METRIC_PERFCOMPONENT_PLUGIN | |

| | | |
|---|---|---|
| | as described here (untested in this report)<br>https://github.com/score-p/scorep_plugin_perf<br>Score-P uses PAPI library for this task | |
| Application/Workflow Description | | |
| Names | N | N |
| Versions | N | N |
| Code region names | Y | Y – compiler flag must be on to obtain names |
| Call trees | Y | Y |
| Components names | N | N |
| Input data (version, size, location) | N | N |
| Output data (version, size, locations) | N | N |
| Misc | | |
| Merge data from nodes | Score-P aggregates files produced for nodes | Trace files produced for each node must be merged<br>Profile files for workflow components each running in a single directory are merged |
| Persistence of data | Y | SQLLite at each node<br>SoSFLow – under development |
| Licensing | New BSD Open Source | BSD |

# Gaps and Future development

## Support for Workflows

TAU has been tested with a prototype concurrent workflow for the NWChem scientific application running 2 processes in parallel **in different directories**.

In the case of workflows, it is critical that identifiers for each workflow component, and component instance be provided to the performance tool and preserved throughout. TAU in particular outputs one Trace and one Profile per application each running in its own directory. If two profiling applications run in the same directory the output files get continuously overwritten, and data is lost.

In the case of very large workflows running up to a 1000 applications concurrently, it is not feasible to ask scientific users to re-organise their workflows to run each process in a separate directory.

We recommend the use of a naming scheme for the output files that would allows distinguishing between various applications that are part of the same workflow run in a single directory. Ideally, the name of the executable, the process ID and the node ID should be used in the naming scheme to help keep track of what we are measuring.

We also recommend the use of a workflow system to provide identifiers to each component that will be especially useful when running large workflows.

## Metrics

The execution time of an application per node, per thread and per code region can be deduced from available metrics, providing that the user has compiled the application with the correct flags to export the names rather than the addresses.

With communication strictly defined as MPI calls, total communication times per application per node can be obtained by aggregating message performance.

## API

Specific for CODAR: an API to the raw data to get the data from ProvEn server, and for data analysis and viz online, with well-documented for API and data format would also be useful to enable provenance inquiries.

## Data persisting

Persisting all metrics produced in trace files requires data to be reduced due to the large volumes produced by performance tools. If the performance tools are run with data reduction, it is imperative that the data reduction process supports the transfer of all metadata and metric names produced by the performance tools in order to enable future aggregation of time spent per node, code region, thread, etc.

In order to enable historical analysis of performance data, these data must be persisted in a queriable system. TAU currently uses SQLite process at each node for such task. SoSFlow, another system

designed to perform such task is described in this paper [1]. Continous development of SoSFlow is encouraged.

### Visualization

The reader is referred to Wei Xu's May 31 report (BNL) "Survey of performance visualization," for a detailed analysis of existing visualization tools for performance metrics.

Fromthis report, we gather that existing tools can visualize single applications but not workflows composing serial or parallel executions as is required by our scientific use cases. These tools lack the ability to illustrate workflow structure, I/O and metadata between workflow components.

## References:

1 Wood, C., Sane, S., Ellsworth, D., Gimenez, A., Huck, K., Gamblin, T., and Malony, A.: 'A scalable observation system for introspection and in situ analytics', in Editor (Ed.)^(Eds.): 'Book A scalable observation system for introspection and in situ analytics' (IEEE Press, 2016, edn.), pp. 42-49.

## APPENDIX A: instrumentation with Score-P

This example is provided courtesy of Jean-Francois Paquet, Stony Brook and Duke University.

The code profiled with Score-P v.3.1 is a hydrodynamics code, located here
https://github.com/miawmiaw/music_gpu

We installed the version 3.1 of ScoreP:
---
wget http://www.vi-hps.org/upload/packages/scorep/scorep-3.1.tar.gz
---

We compiled with these options:
---
```
../configure              '--prefix=/sdcc/u/jfpaquet/scorep-3.1' \
                          '--with-nocross-compiler-suite=gcc' \
                          '--with-machine-name=IC' \
                          '--enable-shared' \
                          '--enable-cuda' \
                          '--with-pdt=/hpcgpfs01/software/pdtoolkit/3.23-gcc/x86_64/bin' \
                          '--with-libcudart-include=/hpcgpfs01/software/cuda/8.0/include' \
                          '--with-libcudart-lib=/hpcgpfs01/software/cuda/8.0/lib64' \
                          '--with-libcupti-include=/hpcgpfs01/software/cuda/8.0/extras/CUPTI/include' \
                          '--with-libcupti-lib=/hpcgpfs01/software/cuda/8.0/extras/CUPTI/lib64' \
                          '--without-mpi'
```
---
We used the gcc-6.1 and we compiled on icsubmit01.sdcc.bnl.gov

To compile, we had to modify our makefile as follow:
---
```
CC := g++
```
---
was changed to
---
```
CC := /gpfshome01/u/jfpaquet/scorep-3.1/bin/scorep --thread=omp:pomp_tpd g++ -Wl,-rpath -
Wl,/sdcc/u/jfpaquet/scorep-3.1/lib
```
---
and then, after filtering,
---
```
CC := /gpfshome01/u/jfpaquet/scorep-3.1/bin/scorep --instrument-
filter=/gpfshome01/u/jfpaquet/music_gpu/filter.dat  --thread=omp:pomp_tpd g++ -Wl,-rpath -
Wl,/sdcc/u/jfpaquet/scorep-3.1/lib
```
---

The filtering file looks like:

---
```
SCOREP_REGION_NAMES_BEGIN
      EXCLUDE
            *Minmod::minmod_dx*
            *EOS::get_pressure*
            *Advance::get_TJb_new*
SCOREP_REGION_NAMES_END
```
---

The workflow was:
1) Compile with ScoreP
2) Run the code as usual
3) Take a first look at the profile:
---
```
 ../scorep-3.1/bin/scorep-score -r scorep-20170605_1128_12113780001392826/profile.cubex
```
---
4) Identify the functions that are called very often using the output of the above command, and add these functions to the filter.
5) Re-compile ScoreP with a reference to the newly created filter file (see above)
6) Re-run ScoreP
7) Look at the ScoreP profile using the GUI interface
---
```
../scorep-3.1/bin/cube scorep-20170605_1128_12113780001392826/profile.cubex
```
---

## APPENDIX B: instrumentation with TAU

There are three different ways to use Tau in combination with a code of interest:

1. Sampling (or dynamic instrumentation): This only requires compiling the code with the "-g" flag as preparation but this does not output data on the communication
2. Source code instrumentation: Compile the code with the tau_cxx.sh, tau_cc.sh or tau_f90.sh scripts and the "-g" flag. These scripts put calls to Tau instrumentation routines into the source code and subsequently invoke the compiler on the instrumented code.
3. Compiler instrumentation: Compile the code with taucxx, taucc, or tauf90 and the "-g" flag to instruct the compiler to insert special calls. With NWChem this approach had problems detecting the symbol names.

Because of the issues with other approaches I prefer the source code instrumentation approach and the instructions below are specific to the source code instrumentation approach.

Before you compile your code with the Tau scripts you need to select a Tau Makefile that gives Tau more information about what your Tau installation supports. For example for an MPI based parallel program you would select a Makefile with MPI capabilities:

export TAU_MAKEFILE=/software/tau/2.26.1-gnu/x86_64/lib/Makefile.tau-mpi-pthread-pdt

Subsequently compiling the code replacing the usual compiler names with tau_cxx.sh, tau_cc.sh or tau_f90.sh as appropriate will generate an instrumented code.

Note that by default Tau will instrument everything. If needed you can turn instrumentation off with a selection file. For example:

```
BEGIN_FILE_EXCLUDE_LIST
input_*.F
dcopy.f
END_FILE_EXCLUDE_LIST
BEGIN_EXCLUDE_LIST
tddft
END_EXCLUDE_LIST
```

To use this file "select.tau" add it to the Tau options as:

export TAU_OPTIONS=-optTauSelectFile=select.tau

**APPENDIX C: TAU support for workflows**

TAU produces Trace or Profile file per directory where an application run. For this reason, Trace and Profile files for a workflow composed of several applications must be merged to produce a Trace or Profile for a workflow composed of several applications.

The workflow data merge tool is called tau_coalesce:

```
tau_coalesce [dir1] [dir2]
```

The tool creates a merged `tau.trc` `tau.edf` file in the current directory and remaps the node ids for each entry/exit record in the trace for the traces in the three folders. Also, if there are `profile.<rank>.0.<thread>` files in the above directories, it will generate the remapped profile.* files in the current directory so tools such as paraprof/pprof may be used on the merged profile dataset.