Regional Profiling for Efficient Performance Optimization

Florent Lebeau, Patrick Wohlschlegel, and Olly Perks

Abstract Performance profiling and debugging are critical components in the HPC application development workflow, ensuring efficient utilization of hardware resources and correctness of solution. Having strong tools to underpin these requirements enables better software development and more efficient execution. The Arm Forge tool suite has long been recognized as industry leading within HPC, for delivering real world usability and actionable information. However, performance engineering is a moving target—due to changes in the HPC ecosystem, such as hardware, software and user workflow. As such the Arm Forge tools are in constant development—to adapt to, and exploit, the latest use cases. One such emerging use case is domain-specific contextual information, in the form of user annotations which can be embedded within performance profiles. Through a collaboration with Lawrence Livermore National Laboratory (LLNL), and their open-source Caliper tool, Arm was able to develop this concept into a fully integrated user workflow. This article will introduce Arm Forge's latest feature on regional profiling and how it complements the more traditional, and established, optimization methodology.

1 Introduction

Arm MAP is a lightweight and scalable profiling tool that provides a user-friendly and intuitive overview of the performance of Linux applications. Thanks to an adaptative sampling mechanism and data aggregation across processes, it is designed to have a low impact on the application's runtime performance and generate small result files. Along with Arm DDT and Arm Performance Reports, MAP is part of Arm Forge: they all share the same petascale-capable architecture [1]. Arm Forge allows

F. Lebeau (✉) · P. Wohlschlegel · O. Perks
Arm, Cambridge, UK
e-mail: Florent.Lebeau@arm.com

P. Wohlschlegel
e-mail: Patrick.Wohlschlegel@arm.com

O. Perks
e-mail: Olly.Perks@arm.com

H. Mix et al. (eds.), *Tools for High Performance Computing 2018 / 2019*,
https://doi.org/10.1007/978-3-030-66057-4_10

scientific developers to write better and more efficient code by providing them with a solution for their whole workflow. This 9-step guide to optimize HPC applications [2] illustrates when and how these tools can be used:

- Ensure application correctness and fix bugs at scale [DDT]
- Measure all performance aspects (computations, communications, IO) on real workload [Performance Reports]
- Inspect I/O patterns and their source code [MAP]
- Investigate workload imbalances and heavy synchronization between processes [Performance Reports, MAP]
- Analyze data transfer rates and slow communication patterns [Performance Reports, MAP]
- Investigate regions with high memory accesses [MAP]
- Evaluate core utilization and thread synchronization [Performance Reports, MAP]
- Inspect hot loops and vectorized instructions [Performance Reports, MAP]
- Validate corrections with automated tests [Performance Reports, MAP, DDT]

Whilst this methodology has proven to be particularly suitable for developers with a good understanding of computer science, the data captured to resolve these problems are, in the best case, only loosely correlated with an application's contextual information. The profiles relate performance, and time spent, to source code lines, functions and libraries: for those without an in-depth understanding of the source code layout this information may be confusing.

In 2019, Arm MAP was extended to support regional profiling using Caliper, a performance data collection and analysis tool developed by the LLNL [3]. The objective of this extension is to enable MAP to not only capture computer-centric data, but to add domain-specific contextual information. Using instrumentation, Caliper facilitates the identification of C/C++ and FORTRAN code regions for performance introspection. It can profile or trace these regions, provide auxiliary statistics (such as MPI or PAPI) and can be coupled to various third-party tools like TAU or Nvprof.

2 Motivation

Profiling with Arm MAP is easy: the user just needs to recompile their code with the debugging option and prefix the execution command with the map command to generate profiling results. The results can be open in the GUI for analysis. MAP straightforwardly represents the application activity in three main sections:

- the metrics graphs describe the activity of the different processes or threads of the application over time,
- the source code viewer displays the lines of code annotated with time and activity information,
- the stacks view aggregates time and activity information by call path.

MAP highlights activity patterns using different colors:

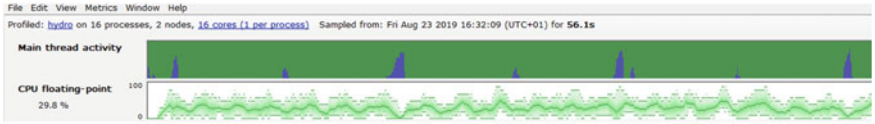

Fig. 1 Activity and CPU floating-point metric

- single-thread computations appear in dark green,
- multi-thread computations in light green,
- thread synchronization in grey,
- MPI calls in blue,
- IO calls in orange.

This makes it easy to understand the various stages of an applications such as synchronizations, data loads or checkpoints. However, analyzing largely compute-bound, MPI-bound or I/O-bound profiles can be more difficult. In this section, we will illustrate some current design limitation by profiling a modified version of the Hydro benchmark [4].

2.1 Application Activity and Metrics

The main thread activity in Fig. 1 pictures the type of operation performed by the 16 processes running Hydro across time. The color code listed above allows to identify what looks like an iterative pattern: MPI calls are performed regularly as the application runs.

This application is compute-bound: 97% of the activity is spent in computations. The CPU floating-point activity graph underneath the main thread activity aggregates data across all processes to display the average. Shading is used to represent the difference between the average and the minimum and maximum values recorded for each sample. Floating-point activity is high for the whole run, especially when compute activity has been recorded, but the graph doesn't highlight any additional pattern.

The lack of more diversity in terms of activity doesn't provide more information at this stage and we need to complement our analysis by investigating the source code of the application.

2.2 Function and Stack View

Figure 2 shows the source code of the main function of Hydro. The iterative aspect is immediately confirmed thanks to the annotations: the same compute-bound function is called by all the processes in two code paths alternately.

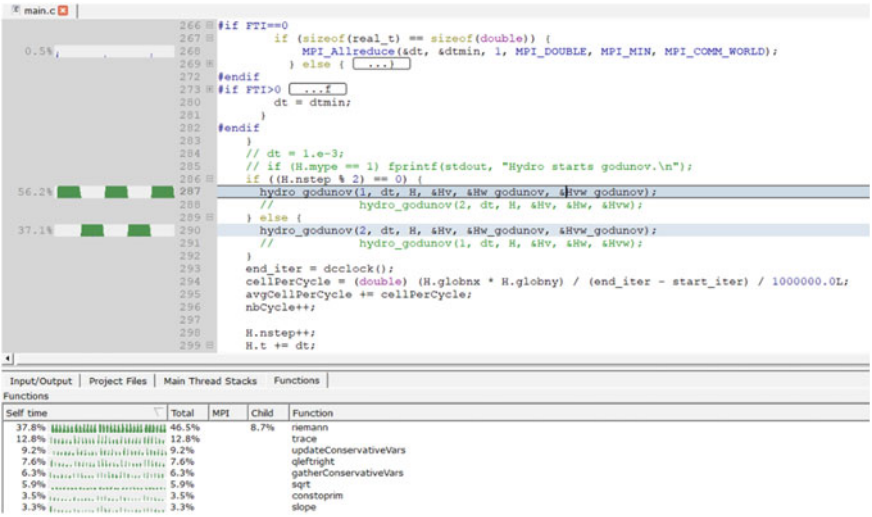

Fig. 2 Source and Function view

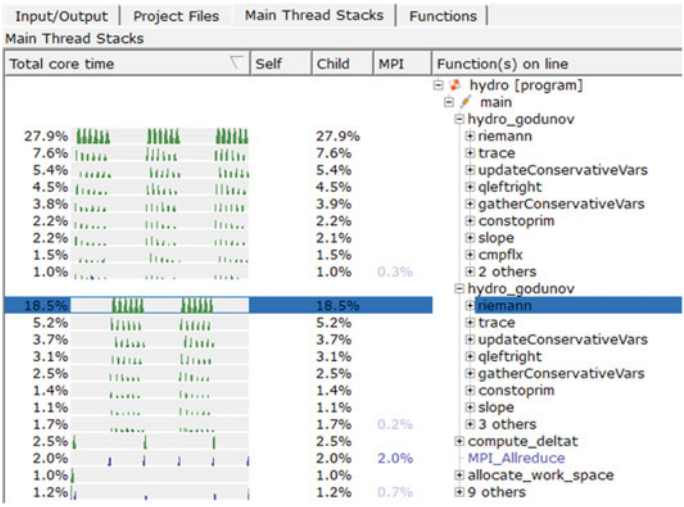

Fig. 3 Stack view

The functions view is displayed underneath the source code viewer and lists the functions sorted by execution time. Thus, it is easy to find the three first bottlenecks of the application: *riemann*, *trace* and *updateConservativeVars*. Time glyphs indicate that they are called all along the execution and that they are compute-bound.

The stack view, shown in Fig. 3, illustrates how these functions are called from *main*, in the two branches of code that were identified earlier.

Fig. 4 Collapsed stack view

Figure 4 illustrates how the stack view can be expanded to show which lines of codes inside these functions are costly.

While the *updateConservativeVars* function spends 79% of its execution time in 3 lines of code, the *trace* function spends 69% in 15 lines and *riemann* 55% in 15 lines. MAP highlights that the internal profile of these last two bottlenecks is flat. Optimizing them might be time-consuming.

In addition to that, the *trace* and *riemann* functions are large pieces of code: approximately 200 and 340 lines respectively when the *updateConservativeVars* function is only 70 lines of code. Gathering application context information is important to make their optimization more efficient.

3 Instrumenting Code with Caliper

MAP has identified the main function bottlenecks and are listed in Table 1.

Table 1 Hydro flat profile

Function	Time spent in self (s)
Riemann	37.8
Trace	12.8
UpdateConservativeVars	9.2
Qleftright	7.6
GatherConservativeVars	6.3
Constoprim	3.5
Slope	3.3

Caliper allows to instrument functions very simply in C using high-level macros [5]:

- CALI_MARK_FUNCTION_BEGIN specifies where a function starts and CALI_MARK_FUNCTION_END specifies where it terminates. All exit points must be marked.
- CALI_LOOP_BEGIN specifies where a loop starts and CALI_LOOP_END specifies where it terminates. Inside the loop region, CALI_MARK_ITERATION_BEGIN identifies the start of an iteration and CALI_MARK_ITERATION_END identifies the end. All iteration exit points must be marked.
- CALI_MARK_BEGIN and CALI_MARK_END specify user-defined code regions.

In Hydro, comments left by the developers allow to break down the *riemann* function and label different sections as shown in Listing 1.

Listing 1 Pseudo-code with Caliper annotations

```
CALI_MARK_LOOP_BEGIN(riemann_slice_id,
                     "riemann_slices");
//compute pressure, density, velocity for each slice
for(s=0; s<slices; s++)
{
      CALI_MARK_ITERATION_BEGIN(riemann_slice_id, s);
      CALI_MARK_BEGIN("riemann_slice_precompute");
      for(i=0; i<narray; i++)
      { [...] }
      CALI_MARK_END("riemann_slice_precompute");
      CALI_MARK_BEGIN("riemann_slice_interfaces");
      for(iter=0; iter<Hniter_riemann; iter++)
      { [...] }
      CALI_MARK_END("riemann_slice_interfaces");
      CALI_MARK_BEGIN("riemann_slice_arrays");
      for(i=0; i<narray; i++)
      { [...] }
      CALI_MARK_END("riemann_slice_arrays");
      CALI_MARK_ITERATION_END(riemann_slice_id);
}
CALI_MARK_LOOP_END(riemann_slice_id);
```

Caliper can be used to generate profiling information on annotated regions. Through a configuration file, services can be selected to provide measurement data using sampling or tracing. The results can be displayed via standard output or stored in data files that can be queried afterwards.

To profile a Caliper-enabled application, MAP doesn't need a Caliper configuration file. MAP only relies on the high-level macros in the source code and automatically adjust the interface to present Caliper-specific information in a user-friendly way.

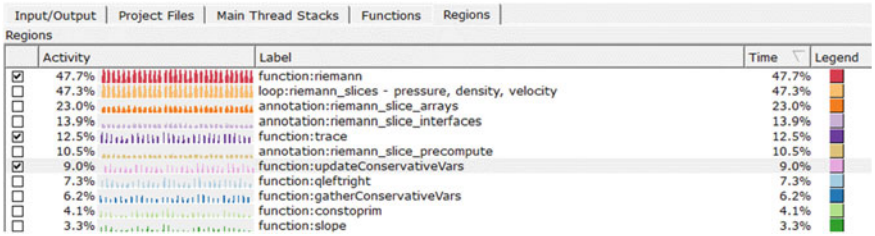

Fig. 5 Regions view displaying Caliper-annotated code sections

4 Visualizing and Analyzing Results

When opening the profile of a Caliper-enable application in MAP's GUI, additional sections automatically appear to display regional information: the regions view and the selected region activity graph.

4.1 Regions View

The regions view lists the code sections marked with Caliper high-level macros as shown in Fig. 5. For each region, the corresponding time spent as a percentage is given and a time glyph shows when and how the region is executed between processes.

Each region can be enabled or disabled to be displayed in the selected region activity graph. A color label allows to identify them in the graph.

4.2 Selected Region

The selected regions graph displays which Caliper region is executed as Hydro runs: each sample or horizontal point indicate how many processes are executing the code regions labelled with different colors. Figure 6 illustrates how the sub iterations of Hydro can be identified easily when enabling the *riemann* (in red), *trace* (purple) and *updateConservativeVars* (pink) Caliper code regions.

In addition, thanks to Caliper the CPU floating-point metric graph highlights that the *riemann* function is particularly responsible for high values. As suggested in the 9-step guide to optimize HPC applications, additional CPU performance aspects can be analyzed more closely: MAP can display many additional metrics. Here, the amount of CPU memory accesses is average, but the amount of CPU vector floating-point operations is low. Figure 7 shows how MAP allows to zoom in a time frame and pinpoint that the *riemann* function is not performing any vector instruction at all.

Table 2 Summary of bottlenecks classified by Caliper region

Caliper region	Number of lines of code	Time spent (%)
Riemann_slice_precompute	4	14
Riemann_slice_interfaces	5	17
Riemann_slice_arrays	6	24

5 Optimizing

Instead of selecting Caliper function regions, arbitrary code regions can be selected to provide insight about how the *riemann* function is executed. The selected regions in Fig. 8 shows how the *riemann_slice_precompute*, *riemann_slice_interfaces*, and *riemann_slice_arrays* regions are executed over time and between processes. MAP is also able to display this information in the source code viewer and in the stack viewer.

MAP highlights that these regions of code in the *riemann* function are not vectorized. Expanding the stack gives more information as shown in Fig. 9 and summarized in Table 2.

These code regions correspond to three different loop nests that the compiler doesn't seem to be able to vectorize. Inserting OpenMP SIMD directives can help and we can generate new profiles with MAP to check if the optimization has been successful.

Figure 10 shows the result of the optimization: the three loop nests are now efficiently vectorized, leading to a speed-up of 1.57 on the whole application.

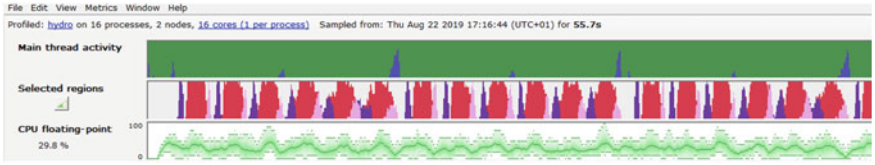

Fig. 6 Selected regions showing the execution of Caliper-annotated code sections

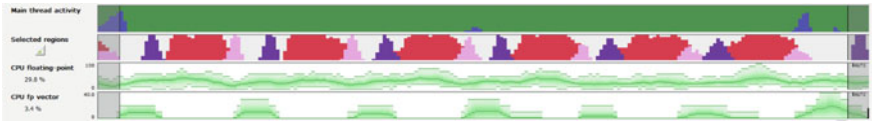

Fig. 7 CPU vector metric

6 Current Limitations

The instrumentation of fine-grain loops can be problematic. It may increase the memory footprint of the application and may result in a significant overhead. However, as shown in Table 3 using MAP on Caliper-enabled application doesn't add overhead compared to using Caliper only.

For now, neither MAP nor Caliper propagates Caliper attributes set on the main thread to OpenMP worker threads when entering an OpenMP parallel region. As a result, the Caliper regions executed by worker threads may not be available. This might be addressed in the future either by MAP or Caliper itself.

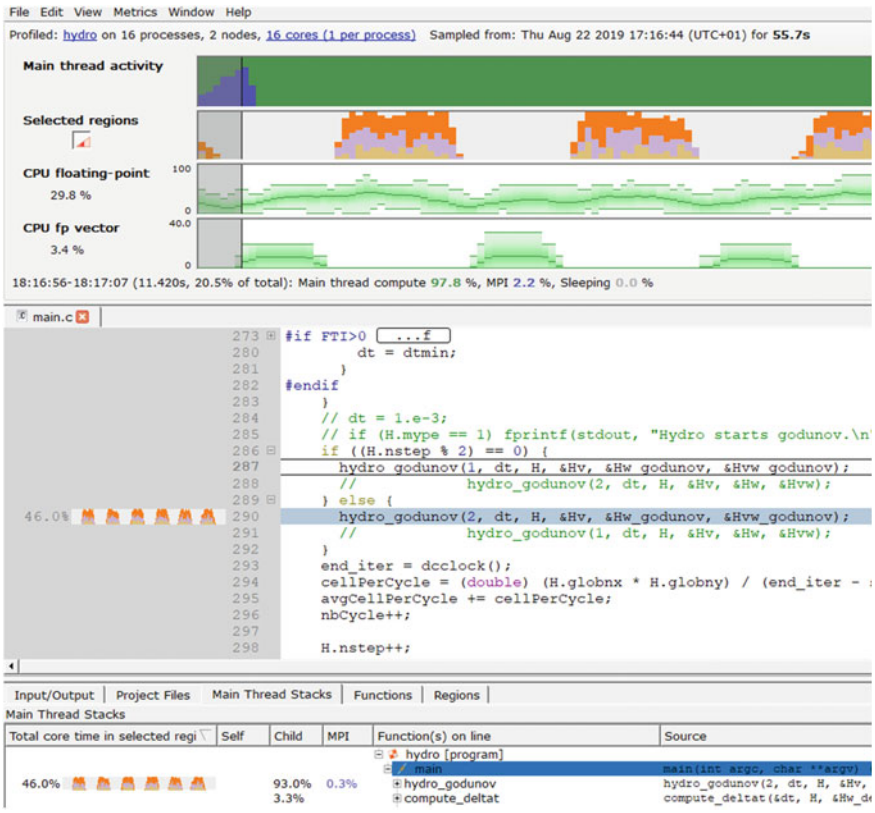

Fig. 8 MAP profiling results with activate regions focused view enabled

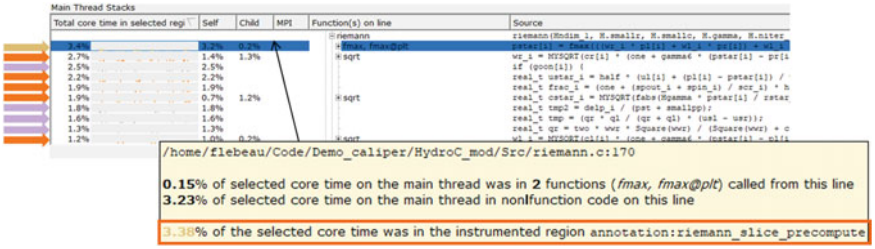

Fig. 9 Stack view with Caliper information

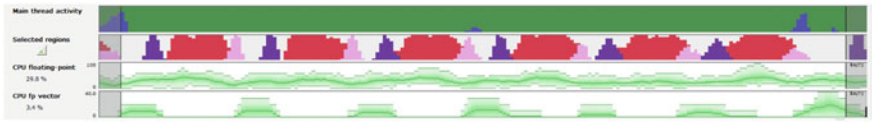

Fig. 10 Caliper annotated with vectorized loops

Table 3 Overhead figures of MAP and Caliper

Application	Run time (s)	Run time with MAP (s)	Run time with MAP and Caliper (%)
Hydro	36.09	36.47	1
Hydro with Caliper	36.71	37.42	2
Hydro with Caliper and fine-grain loop instrumentation	112.44	124.5	11

7 Conclusion

We have presented MAP, a lightweight profiling tool for HPC and Caliper, a performance introspection framework. We have illustrated how MAP can benefit from Caliper: it brings meaningful information to application profile and helps analyzing and optimizing applications faster. We have also demonstrated how MAP contributes to the Caliper ecosystem and how it can complement the work done with other third-party tools.

Thanks to the support for Caliper annotations, the 9-step guide to optimize HPC applications can now be used by domain scientists in addition to computer scientists. They can work hand in hand and optimize the application further, verify if the changes produce correct results or if there are any bug left in the application. The Arm DDT parallel debugger can help with this, by allowing users to inspect data structures and check for memory leaks for instance.

Other possible use cases could be to analyze the behavior of the application when scaling up to more nodes or with different test cases. Caliper annotations could

help finding misbehavior in functions or sections of code that only appear on some configurations.

References

1. January, C., Lecomber, D., O'Connor, M.: Debugging at petascale and beyond. In: Cray User Group 2011 Proceeding, Anchorage, USA (2011)
2. Arm.: Performance Roadmap (2018). https://youtu.be/Lxrpl5HqBKs
3. Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., Schulz, M.: Caliper: performance introspections for HPC software stacks. In: Proceeding SC'16, Salt Lake City, USA (2016)
4. de Verdiere, G.C.: Hydro Benchmark. https://github.com/HydroBench/Hydro.git (2013)
5. Boehme, D.: Caliper: a performance analysis toolbox in a library. https://www.vi-hps.org/cms/upload/material/tw31/Caliper.pdf (2019)