

The background features a dark green, semi-transparent hexagonal shape on the left side, containing a circular emblem with crossed wrench and screwdriver tools. The rest of the background is a vibrant green circuit board pattern with glowing nodes and lines. The text is overlaid on this background.

Linaro Forge

Debugging and Optimization Tools for HPC

Linaro Forge

Agenda

- A Brief history
- DDT Overview (Debugger)
- MAP Overview (Profiler)
- Performance Reports Overview

A Brief History



2014: Release of Allinea tools 5.0, with addition of the new Allinea Performance Reports.

December 2016: Arm extends HPC offering with acquisition of software tools provider Allinea Software.



12 major releases



Linaro Forge

30th January 2023: Linaro to Acquire Arm Forge Software Tools Business.

HPC Development Solutions from Linaro

Best in class commercially supported tools for HPC

Linaro Forge



Debug
Linaro DDT



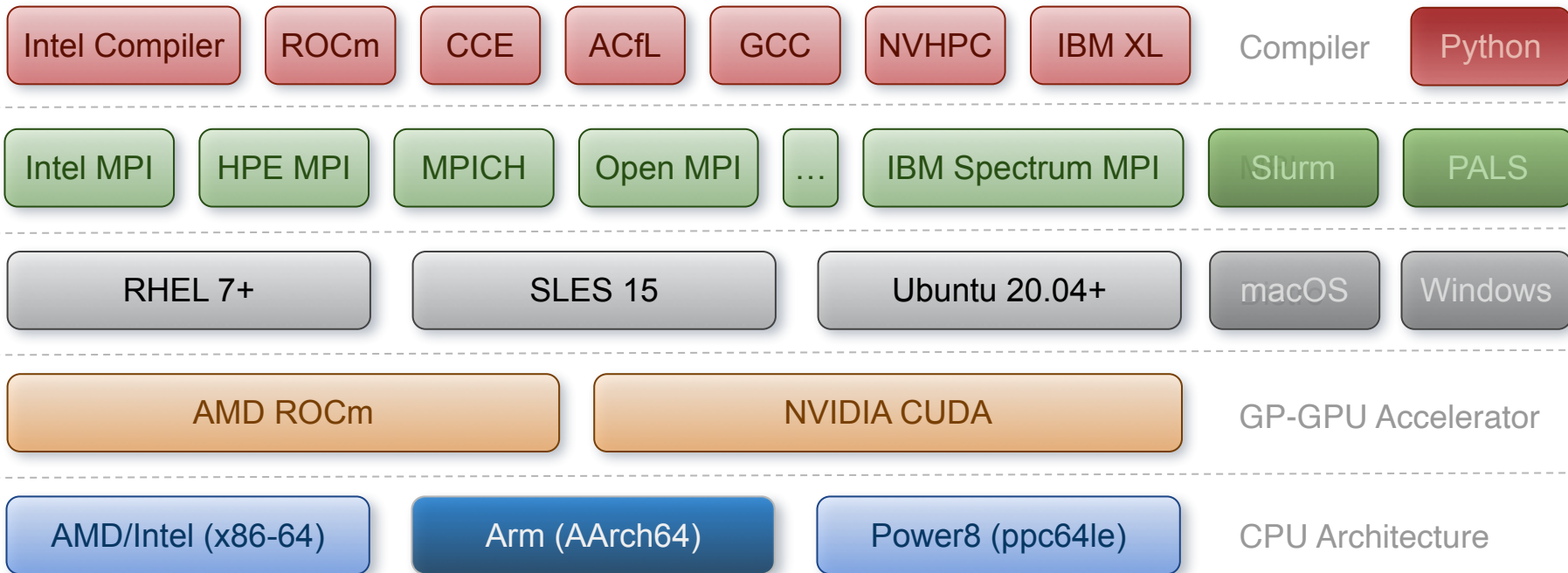
Profile
Linaro MAP



Analyse
Linaro
Performance Reports

Performance Engineering for any architecture, at any scale

Supported Platforms



Linaro Forge

An interoperable toolkit for debugging



The de-facto standard for HPC development

- Most widely-used debugging and profiling suite in HPC
- Fully supported by Linaro on Intel, AMD, Arm, Nvidia, AMD GPUs, etc.



State-of-the art debugging capabilities

- Powerful and in-depth error detection mechanisms (including memory debugging)
- Available at any scale (from serial to exascale applications)



Easy to use by everyone

- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

Multi-dimensional Array Viewer

What does your data look like at runtime?

View arrays

- On a single process
- Or distributed on many ranks

Use metavariables to browse the array

- Example: $\$i$ and $\$j$
- Metavariables are unrelated to the variables in your program
- The bounds to view can be specified
- Visualise draws a 3D representation of the array

Data can also be filtered

- “Only show if”: $\$value > 0$ for example $\$value$ being a specific element of the array

The screenshot shows the Multi-Dimensional Array Viewer interface. At the top, the Array Expression is set to `tables[$i][$j]`. Below this, there are controls for Distributed Array Dimensions (set to None), Staggered Array (unchecked), and Range of $\$i$ and $\$j$ (both set to From: 0, To: 11). The Display options are set to Rows and Columns. There are buttons for Evaluate, Cancel, and a checkbox for Align Stack Frames. A section for filtering with "Only show if:" and a "See Examples" link is also present.

The Data Table tab is active, displaying a 12x12 grid of values:

	0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	6	8	10	12	14	16	18	20	22	24
2	3	6	9	12	15	18						
3	4	8	12	16	20	24						
4	5	10	15	20	25	30						
5	6	12	18	24	30	36						
6	7	14	21	28	35	42						
7	8	16	24	32	40	48						
8	9	18	27	36	45	54						
9	10	20	30	40	50	60						
10	11	22	33	44	55	66						

The 3D visualization shows a green surface plot of the data, with axes labeled 'row', 'column', and 'value'. The value axis ranges from 0 to 1000.

The Performance Roadmap

Optimizing high performance applications

Improving the efficiency of your parallel software holds the key to solving more complex research problems faster.

This pragmatic, 9 Step best practice guide, will help you identify and focus on application readiness, bottlenecks and optimizations one step at a time.

Bugs

- Correct application

Analyze before you optimize

- Measure all performance aspects. You can't fix what you can't see.
- Prefer real workloads over artificial tests.

I/O

- Discover lines of code spending a long time in I/O.
- Trace and debug slow access patterns.

Workloads

- Detect issues with balance.
- Slow communication calls and processes. Dive into partitioning code.

Communication

- Track communication performance. Discover which communication calls are slow and why.

Memory

- Reveal lines of code bottlenecked by memory access times.
- Trace allocation and use of hot data structure

Vectorization

- Understand numerical intensity and vectorization level.
- Hot loops, unvectorized code and GPU performance revealed

Cores

- Discover synchronization overhead and core utilization
- Synchronization-heavy code and implicit barriers are revealed

Verification

- Validate corrections and optimal performance

Linaro Performance tools

Characterize and understand the performance of HPC application runs



Commercially supported
by Linaro

Gather a rich set of data

- Analyses metric around CPU, memory, IO, hardware counters, etc.
- Possibility for users to add their own metrics



Accurate and
Astute insight

Build a culture of application performance & efficiency awareness

- Analyses data and reports the information that matters to users
- Provides simple guidance to help improve workloads' efficiency

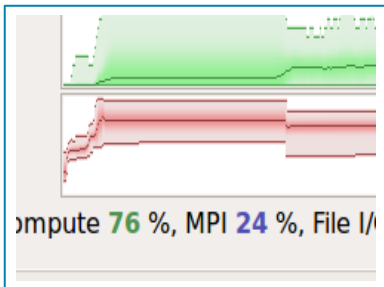


Relevant advice
to avoid pitfalls

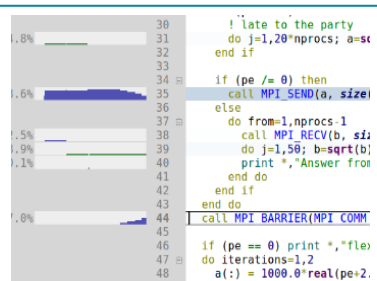
Adds value to typical users' workflows

- Define application behaviour and performance expectations
- Integrate outputs to various systems for validation (eg. continuous integration)
- Can be automated completely (no user intervention)

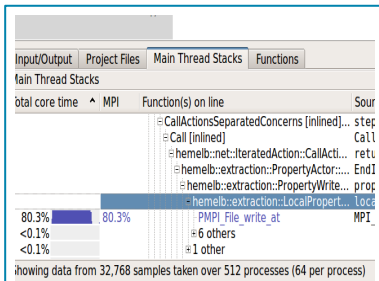
Linaro MAP Source Code Profiler Highlights



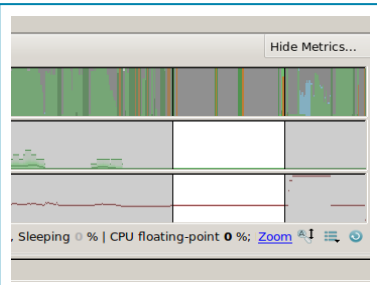
Find the peak memory use



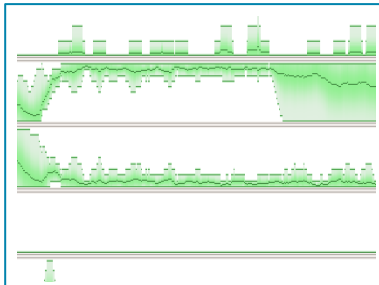
Fix an MPI imbalance



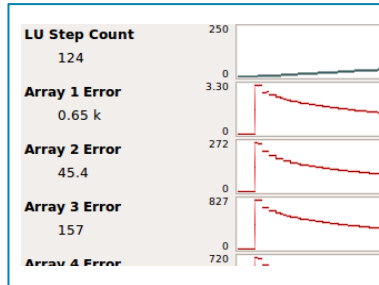
Remove I/O bottleneck



Make sure OpenMP regions make sense



Improve memory access



Custom Metrics

MAP Capabilities

MAP is a sampling based scalable profiler

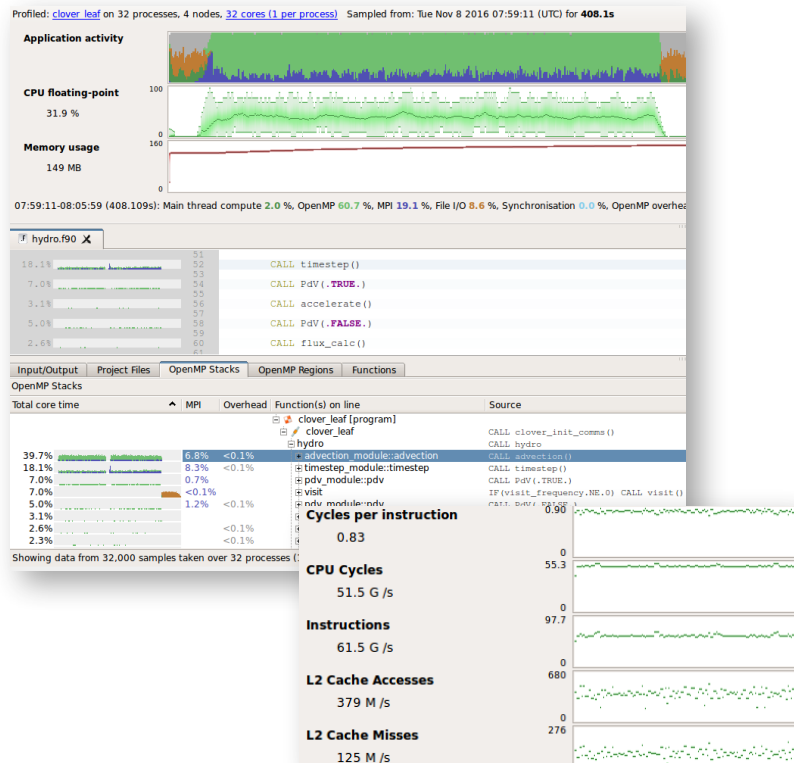
- Built on same framework as DDT
- Parallel support for MPI, OpenMP, CUDA
- Designed for C/C++/Fortran

Designed for 'hot-spot' analysis

- Stack traces
- Augmented with performance metrics

Adaptive sampling rate

- Throws data away - 1,000 samples per process
- Low overhead, scalable and small file size





Thank you

Go to www.linaroforge.com
rudyshand@linaro.org

Linaro Forge

Hands on examples

Install Forge <https://www.linaroforge.com/downloadForge>

Forge user guide <https://docs.linaroforge.com/23.0.1/html/forge/forge/index.html>

/grand/ATPESC2023/Linaro-Forge/examples

Installed as part of Forge tools as well

<forge location>/examples

Use the temporary license shown below

```
export ALLINEA_FORCE_LICENCE_FILE=/grand/ATPESC2023/Linaro-Forge/Licence.trial
```

Remote client cheat sheet

Install the Remote Client

<https://www.linaroforge.com/downloadForge>

Setup the client

1. Open your Remote Client
2. Create a new connection: RemoteLaunch → Configure → Add
3. Hostname: `<username>@theta.alcf.anl.gov`
4. Remote installation directory: `/soft/debuggers/forge-22.0.4-2022-08-02`

Setup the remote side

1. `qsub -l -n 8 -A ATPESC2023 -q debug-cache-quad -t 30 --attrs filesystems=home,grand,eagle`
2. `module load forge`
3. `module unload xalt`
4. `module unload darshan/3.3.0`
5. `ddt --connect --mpi="Cray XT/XE/XK (MPI/shmem)" aprun -n 8 ./hello_c`

Debugging on Thetagpu

The latest Forge modules are not available on thetagpu, but you can use the installed software directly

Debug your GPU code using:
`ddt --connect gpu_code.exe`

Profiling on Theta

Although static binaries are created by default on Theta, it is recommended to build dynamic executables for profiling purposes with the compiler flag **-dynamic**

If you get library missing errors, reload the intel module

```
moduleunloadintel
```

```
moduleloadintel
```

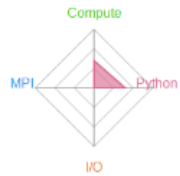
If you get GdbmiParser errors set the following environment variable

```
exportALLINEA_FORCE_DEBUGGER=gdb-82
```

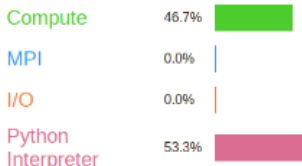
Debugging and Performance Engineering for Nvidia and AMD GPUs

Linaro Performance Reports

Command: `/opt/python/3.8.5-shared/bin/python3 python-profiling.py`
 Resources: 1 node (12 physical, 16 logical cores per node)
 Memory: 15 GiB per node
 Tasks: 1 process
 Machine: `thoth`
 Architecture: `x86_64`
 CPU Family: `alderlake`
 Start time: `Tue May 30 14:44:01 2023`
 Total time: `29 seconds`
 Full path: `/opt/python/3.8.5-shared/bin`



Summary: `python-profiling.py` is **Python Interpreter-bound** in this configuration



Time spent running application code. High values are usually good. This is **low**, consider improving MPI or I/O performance first

Time spent in MPI calls. High values are usually bad. This is **very low**, this code may benefit from a higher process count

Time spent in filesystem I/O. High values are usually bad. This is **negligible**, there's no need to investigate I/O performance

Time spent in the Python interpreter. Calling out to precompiled libraries may be more efficient. This is **average**, consider moving more logic to compiled libraries

Profiled: `clover_leaf` on 6 processes, 1 node, **12 cores (2 per process)**. Sampled from: `Mon Nov 5 2018 11:20:25 (UTC-05)` for **300.2s**

Main thread activity

Warp stall reasons

Memory usage

521 MB

11:20:25-11:25:25 (300.205s): MPI **9.0 %**, Accelerator **91.0 %**

Python Profiling

21.0 - improved python support

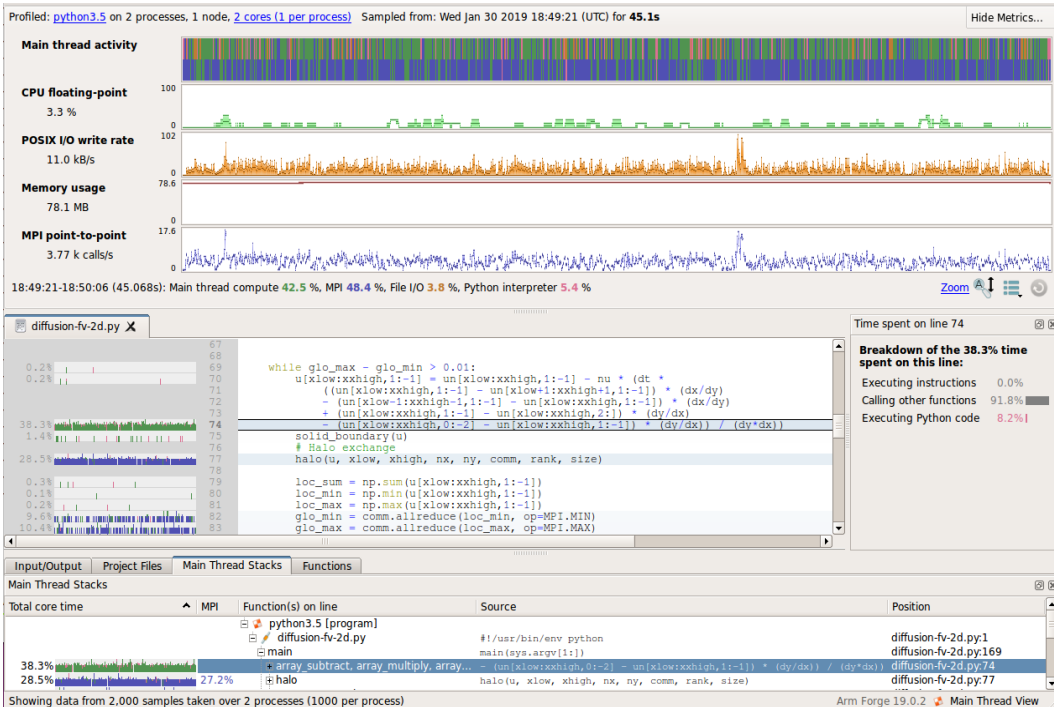
- Call stacks
- Time in interpreter

Works with MPI4PY

- Usual MAP metrics

Source code view

- Mixed language support



Note: Green as operation is on numpy array, so backed by C routine, not Python (which would be pink)

```
map --profile jsrun -n 2 python3 ./diffusion-fv-2d.py
```