linaroforge

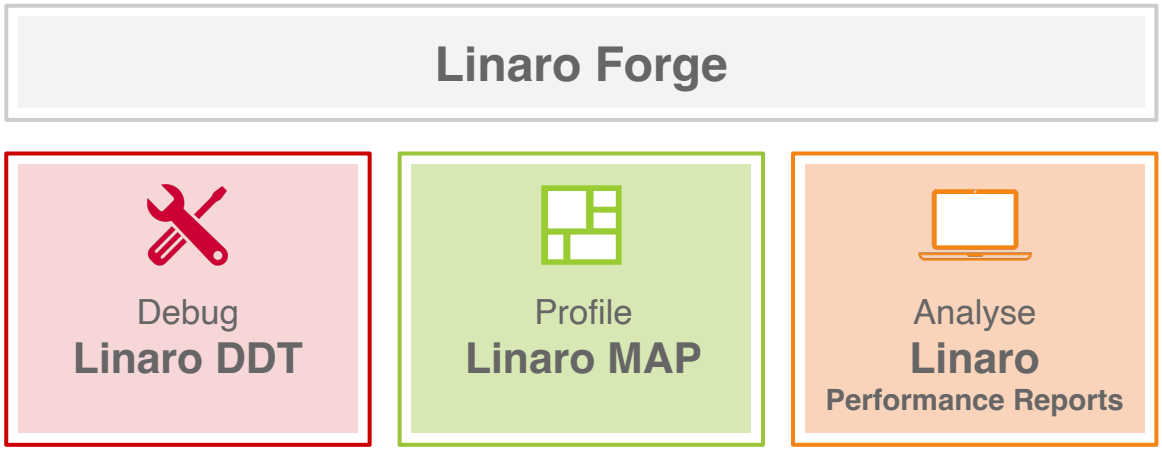
Linaro Forge

Performance Engineering with Linaro PR and Linaro MAP

Rudy Shand - Field Application Engineer
Linaro
rudy.shand@linaro.org

HPC Development Solutions from Linaro

Best in class commercially supported tools for Linux and high-performance computing (HPC)



Performance Engineering for any architecture, at any scale

Linaro Forge

An interoperable toolkit for debugging and profiling

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.

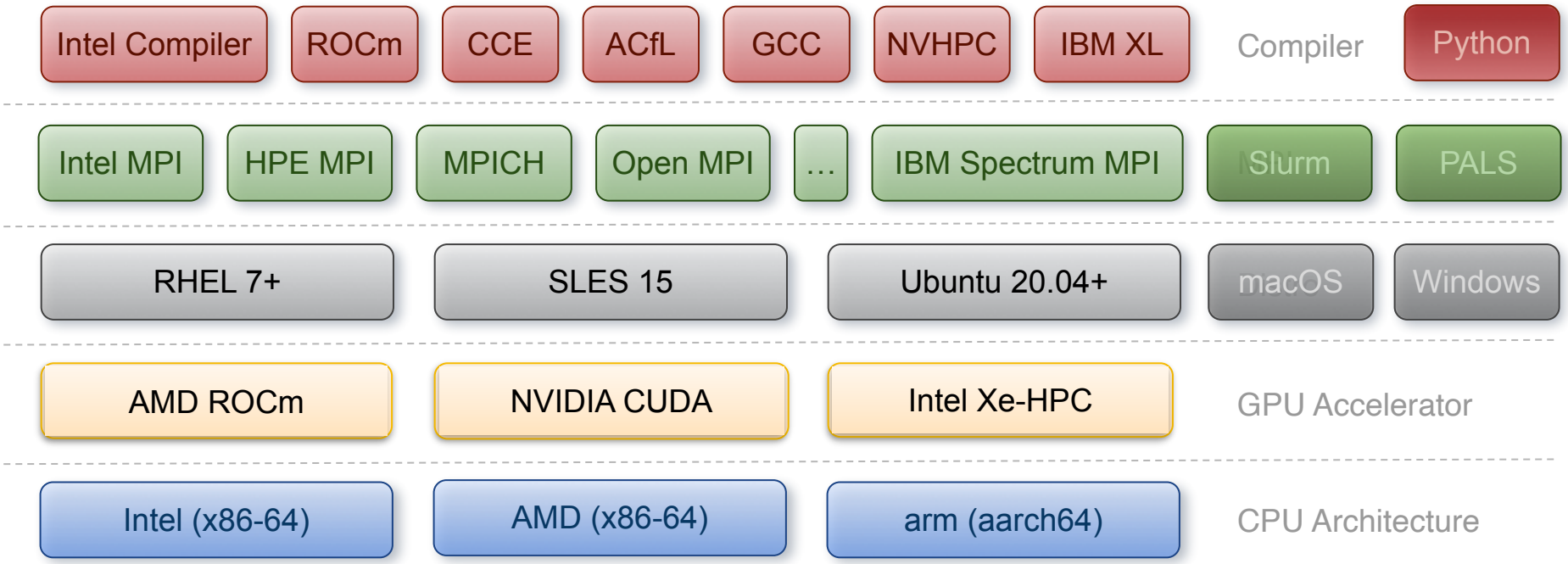
State-of-the art debugging and profiling capabilities

- Powerful and in-depth error detection mechanisms (including memory debugging)
- Sampling-based profiler to identify and understand bottlenecks
- 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

Supported Platforms



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 Performance Reports

A high-level view of application performance with “plain English” insights

Command:	mpiexec.hydra -host node-1,node-2 -map-by socket -n 16 -ppn 8 ./Bin/low_freq/../../Src//hydro -i ./Bin/low_freq/../../../../Input/input_250x125_corner.nml
Resources:	2 nodes (8 physical, 8 logical cores per node)
Memory:	15 GiB per node
Tasks:	16 processes, OMP_NUM_THREADS was 1
Machine:	node-1
Start time:	Thu Jul 9 2015 10:32:13
Total time:	165 seconds (about 3 minutes)
Full path:	Bin/../Src

Summary: hydro is MPI-bound in this configuration

Compute	20.6%	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	63.2%	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	16.2%	Time spent in filesystem I/O. High values are usually bad. This is **average**; check the I/O breakdown section for optimization advice

I/O

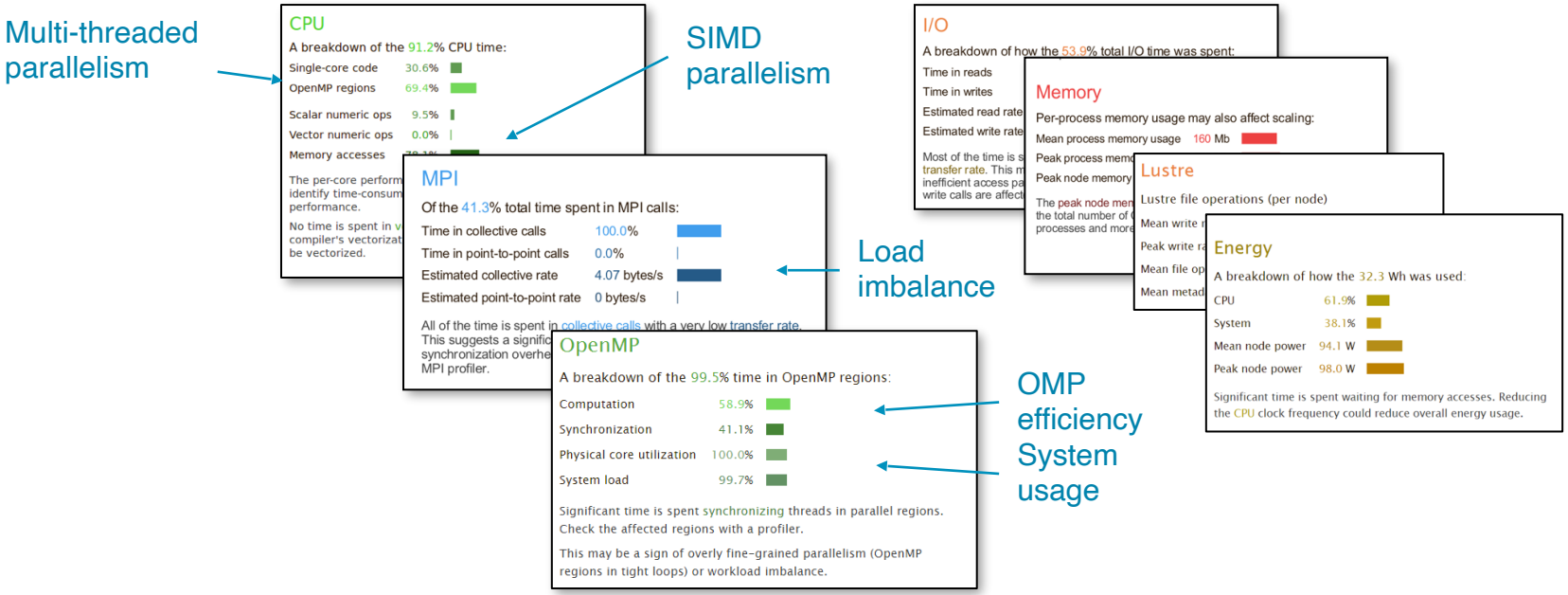
A breakdown of the 16.2% I/O time:

Time in reads	0.0%
Time in writes	100.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	1.38 MB/s

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Linaro Performance Reports Metrics

Lowers expertise requirements by explaining everything in detail right in the report

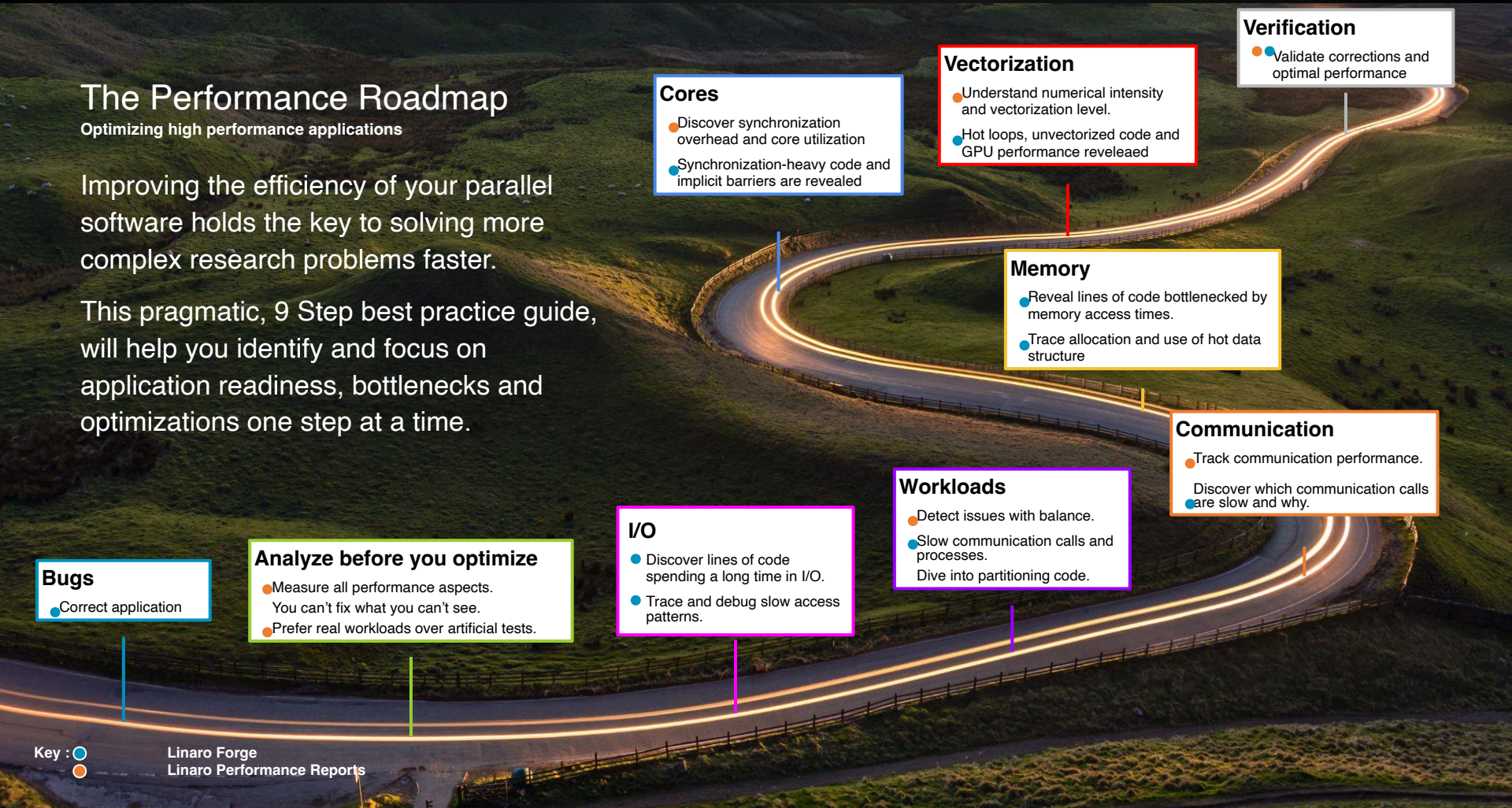


The Performance Roadmap

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.



Verification

- Validate corrections and optimal performance

Vectorization

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

Cores

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

Memory

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

Communication

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

Workloads

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

I/O

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

Analyze before you optimize

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

Bugs

- Correct application

Key : Linaro Forge
Linaro Performance Reports

MAP Capabilities

MAP is a sampling based scalable profiler

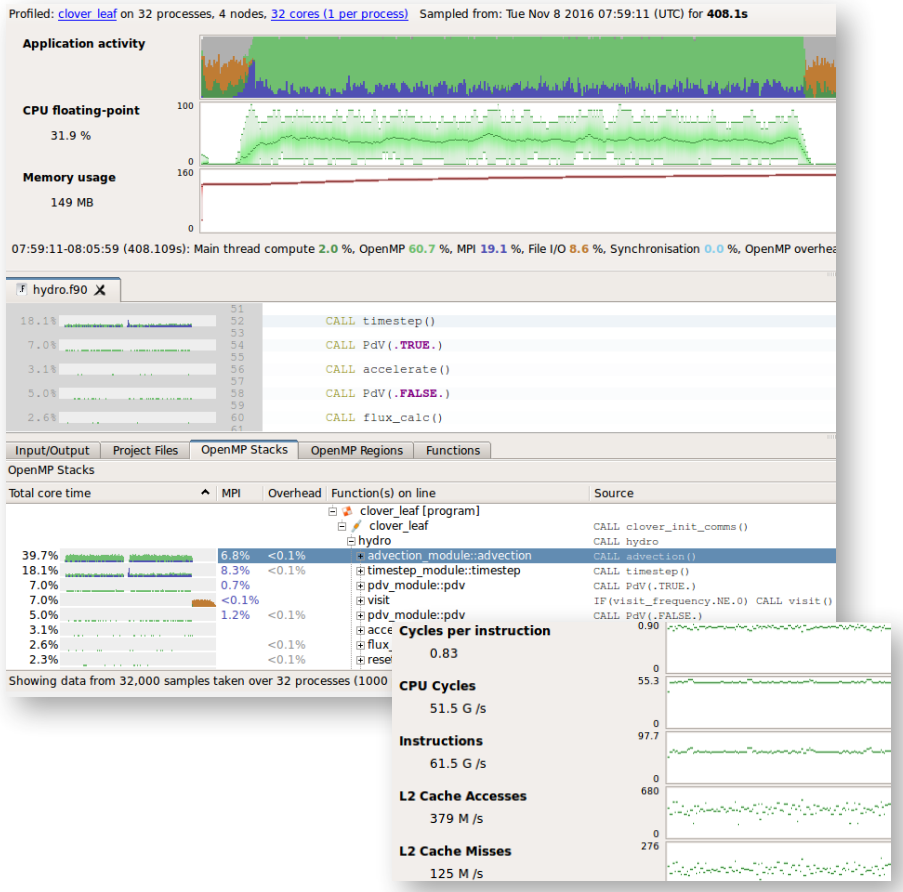
- Built on same framework as DDT
- Parallel support for MPI, OpenMP, CUDA
- Designed for C/C++/Fortran

Designed for ‘hot-spot’ analysis

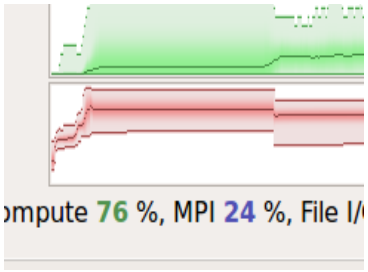
- Stack traces
- Augmented with performance metrics

Adaptive sampling rate

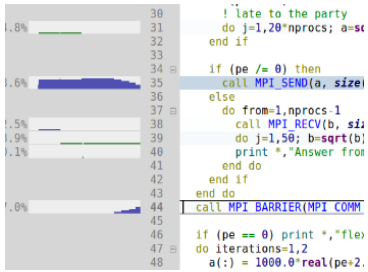
- Throws data away - 1,000 samples per process
- Low overhead, scalable and small file size



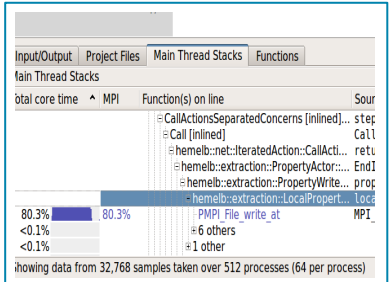
Linaro MAP Source Code Profiler Highlights



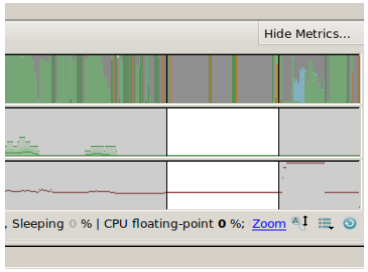
Find the peak memory use



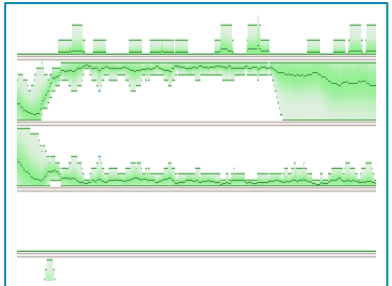
Fix an MPI imbalance



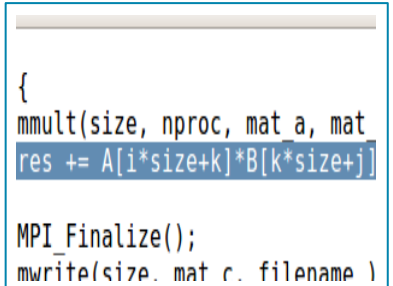
Remove I?O bottleneck



Make sure OpenMP regions make sense

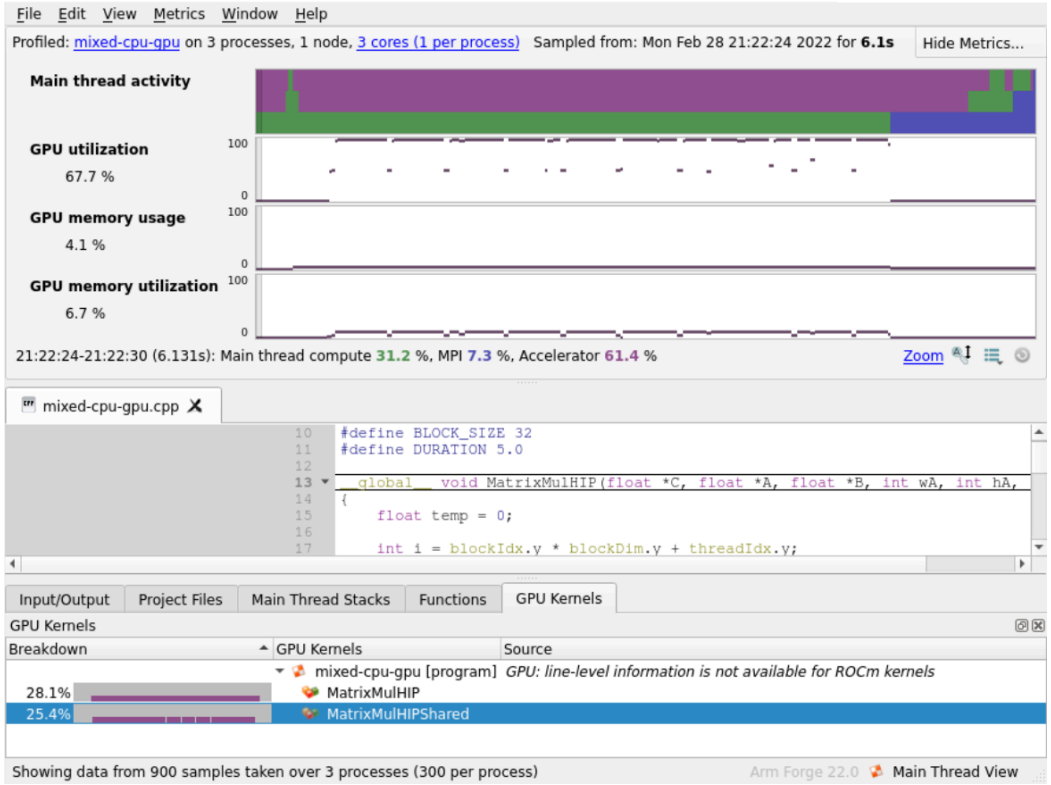


Improve memory access



Restructure for vectorization

GPU Profiling



Profile

- Supports both AMD and Nvidia GPUs
- Able to bring up metadata of the profile
- Mixed CPU [green] / GPU [purple] application
- CPU time waiting for GPU Kernels [purple]
- GPU Kernels graph indicating Kernel activity

GUI information

- GUI is consistent across platforms
- Zoom into main thread activity
- Ranked by highest contributors to app time

Python Profiling

19.0 adds support for Python

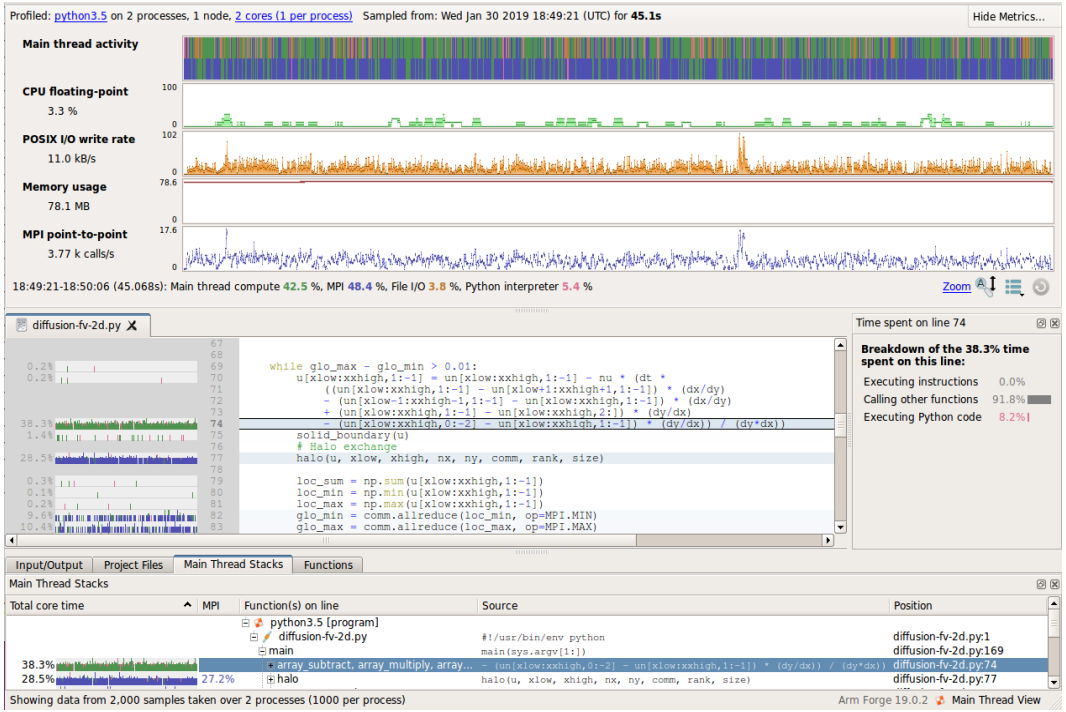
- Call stacks
- Time in interpreter

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 srun -n 2 python3 ./diffusion-fv-2d.py
```

Compiler Remarks

Annotates source code with compiler remarks

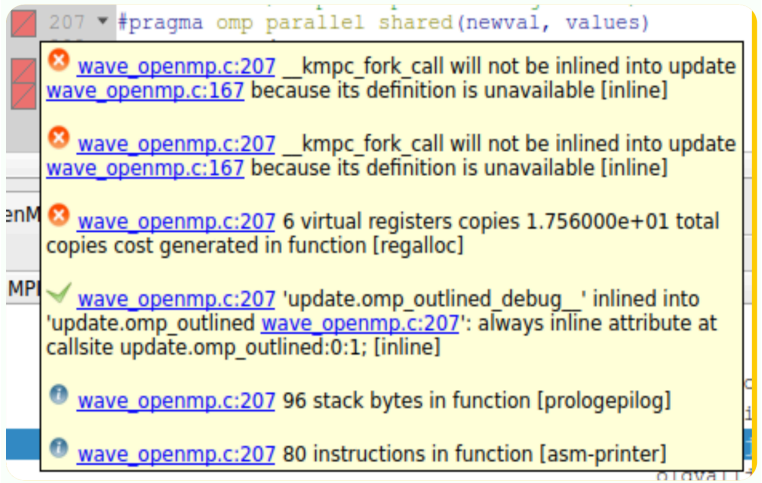
- Remarks are extracted from the compiler optimisation report
- Compiler remarks are displayed as annotations next to your source code

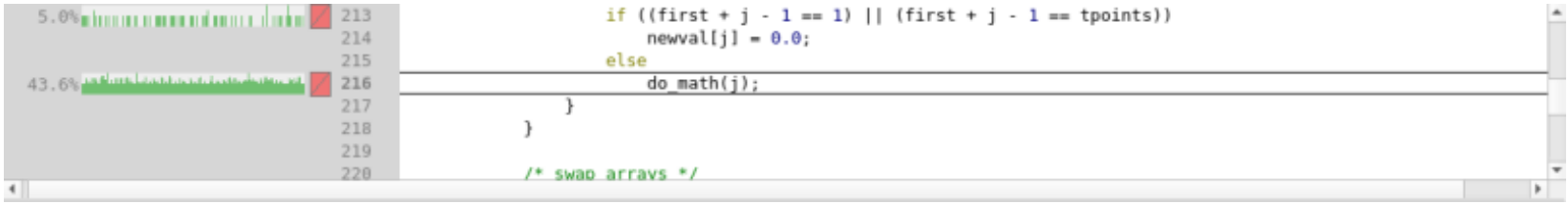
Colour coded

- Their colour indicates the type of remark present in the following priority order:
  1. Red: failed or missed optimisations
  2. Green: successful or passed optimisations
  3. White: information or analysis notes

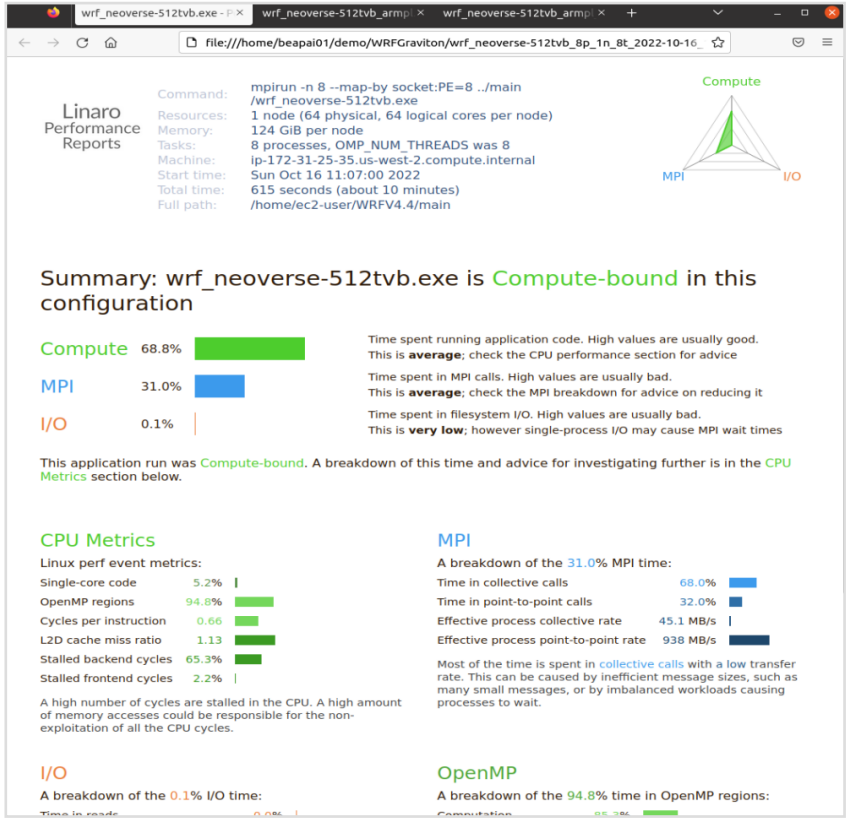
Compiler Remarks menu.

- Specify build directories for non-trivial build systems
- Filter out remarks

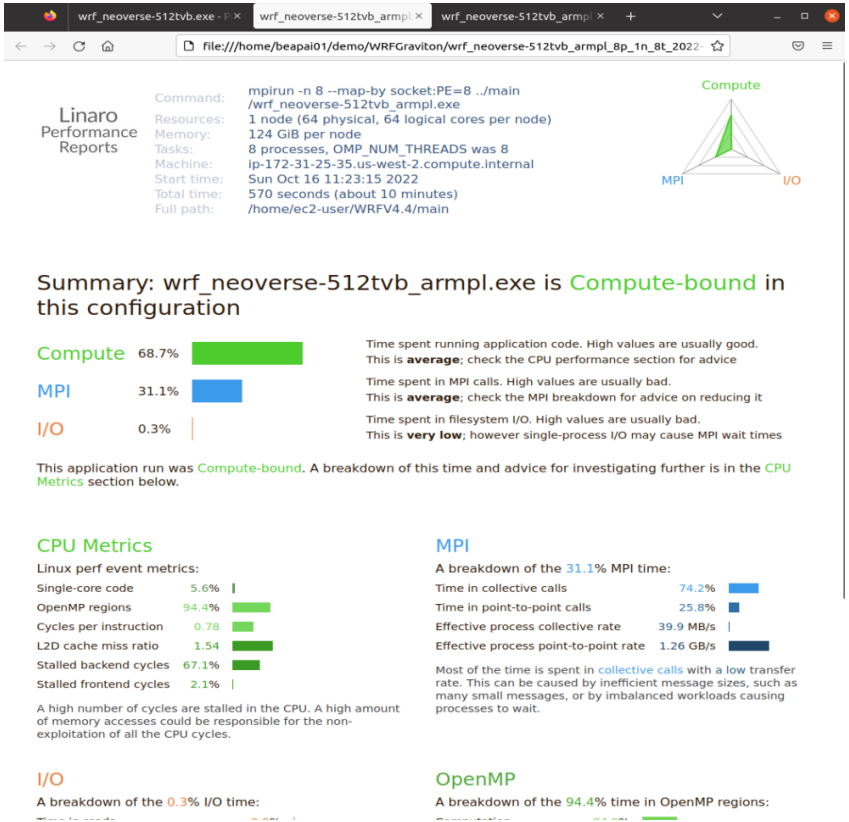




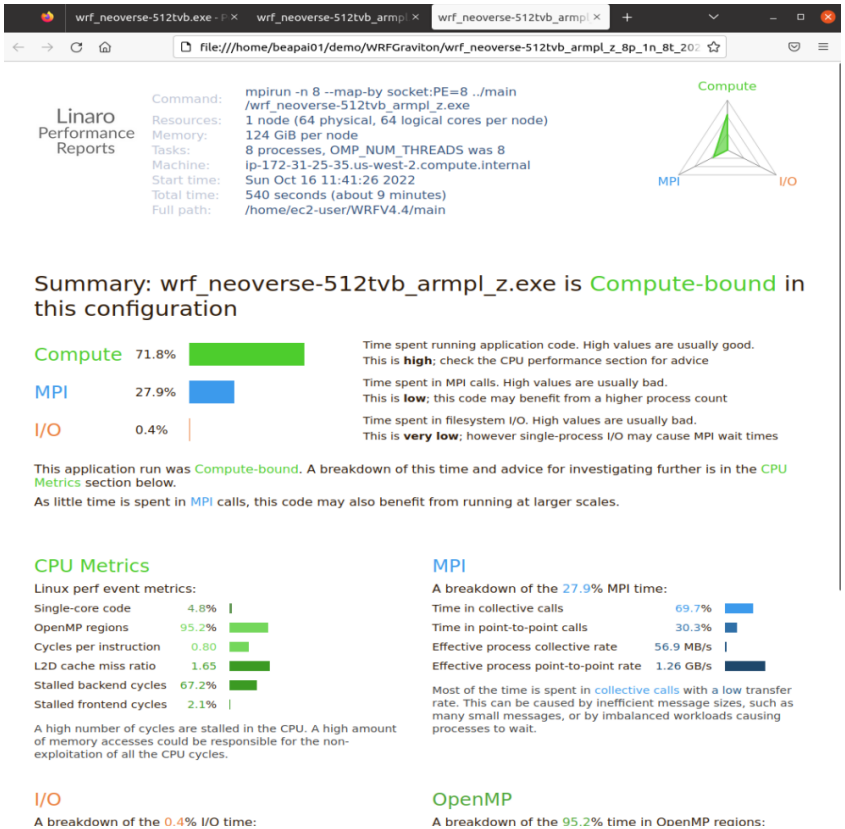
WRF build without enhancements



WRF with Arm Performance Libraries



WRF with Arm Performance Libraries and IO compression libraries



Thank You

rudy.shand@linaro.org