linaroforge

Linaro Forge

Performance Engineering with Linaro PR and Linaro MAP

Rudy Shand - Principal 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)

Linaro Forge

Debug	Profile	Analyse
Linaro DDT	**Linaro MAP**	**Linaro Performance Reports**

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

Compiler: Intel Compiler, ROCm, CCE, ACfL, GCC, NVHPC, IBM XL, Python

MPI: Intel MPI, HPE MPI, MPICH, Open MPI, …, IBM Spectrum MPI

Job Scheduler: Slurm, PALS

OS: RHEL 7+, SLES 15, Ubuntu 20.04+, macOS, Windows

GPU Accelerator: AMD ROCm, NVIDIA CUDA, Intel Xe-HPC

CPU Architecture: Intel (x86-64), AMD (x86-64), arm (aarch64)

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

Multi-threaded parallelism

SIMD parallelism

Load imbalance

OMP efficiency

System usage

CPU

A breakdown of the 91.2% CPU time:

Single-core code	30.6%
OpenMP regions	69.4%
Scalar numeric ops	9.5%
Vector numeric ops	0.0%
Memory accesses	

The per-core perform
identify time-consum
performance.

No time is spent in v
compiler's vectorizat
be vectorized.

MPI

Of the 41.3% total time spent in MPI calls:

Time in collective calls	100.0%
Time in point-to-point calls	0.0%
Estimated collective rate	4.07 bytes/s
Estimated point-to-point rate	0 bytes/s

All of the time is spent in collective calls with a very low transfer rate.
This suggests a signific
synchronization overhe
MPI profiler.

OpenMP

A breakdown of the 99.5% time in OpenMP regions:

Computation	58.9%
Synchronization	41.1%
Physical core utilization	100.0%
System load	99.7%

Significant time is spent synchronizing threads in parallel regions.
Check the affected regions with a profiler.

This may be a sign of overly fine-grained parallelism (OpenMP regions in tight loops) or workload imbalance.

I/O

A breakdown of how the 53.9% total I/O time was spent:

Time in reads
Time in writes
Estimated read rate
Estimated write rate

Most of the time is s
transfer rate. This m
inefficient access pa
write calls are affect

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	160 Mb
Peak process memo	
Peak node memory	

The peak node men
the total number of
processes and more

Lustre

Lustre file operations (per node)

Mean write r
Peak write ra
Mean file op
Mean metad

Energy

A breakdown of how the 32.3 Wh was used:

CPU	61.9%
System	38.1%
Mean node power	94.1 W
Peak node power	98.0 W

Significant time is spent waiting for memory accesses. Reducing the CPU clock frequency could reduce overall energy usage.

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

Cores

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

Vectorization

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

Verification

- Validate corrections and optimal performance

Key : Linaro Forge
Linaro Performance Reports

MAP Capabilities

MAP is a sampling based scalable profiler

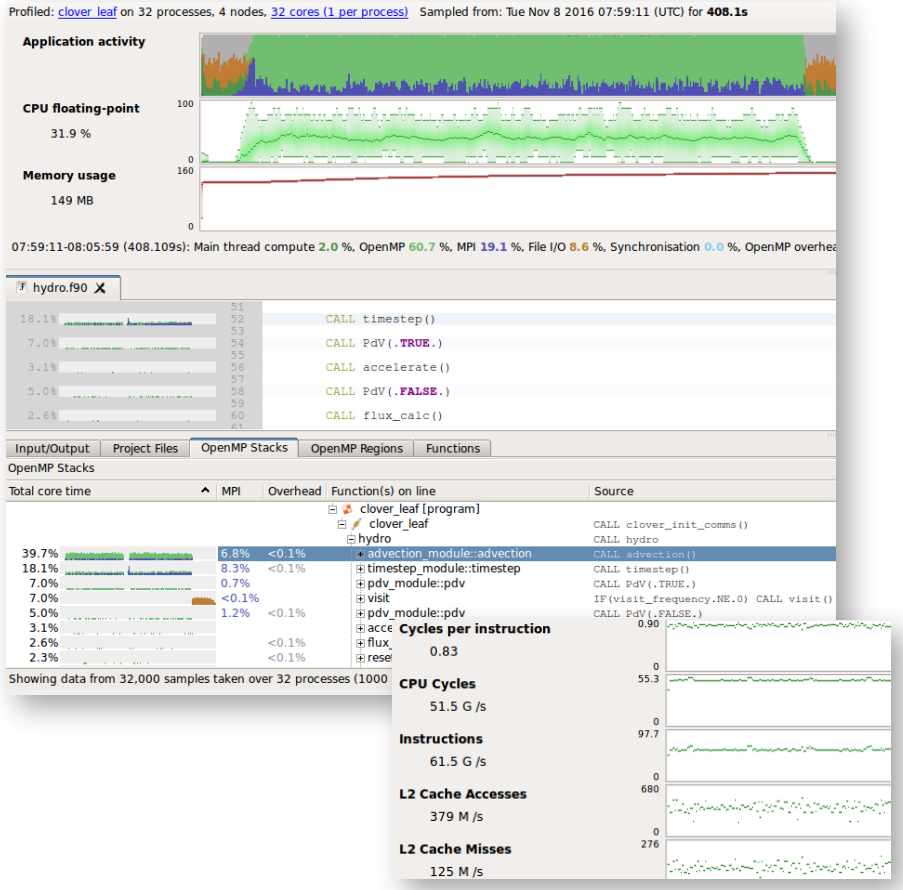
- Built on same framework as DDT
- Parallel support for MPI, OpenMP, CUDA
- Designed for C/C++/Fortran

Designed for 'hot-spot' analysis

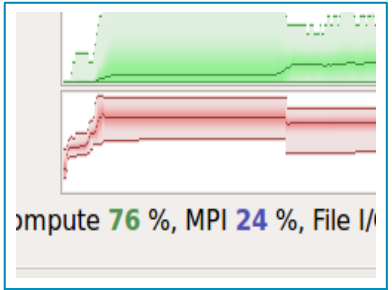
- Stack traces
- Augmented with performance metrics

Adaptive sampling rate

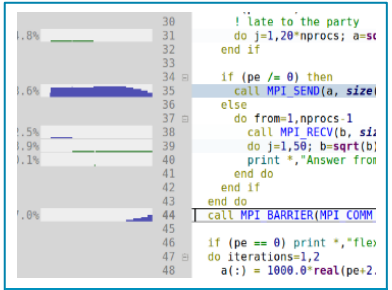
- Throws data away - 1,000 samples per process
- Low overhead, scalable and small file size



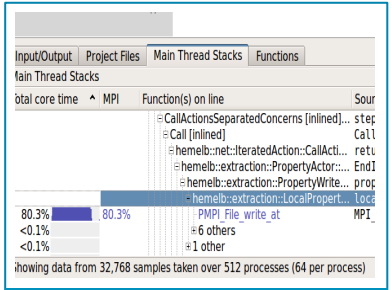
Linaro MAP Source Code Profiler Highlights



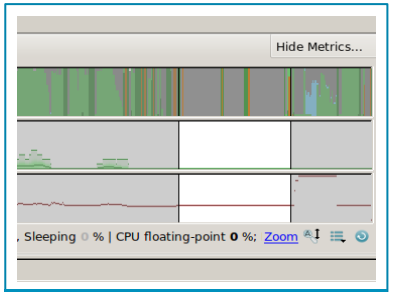
Find the peak memory use



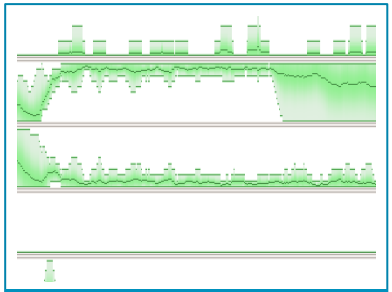
Fix an MPI imbalance



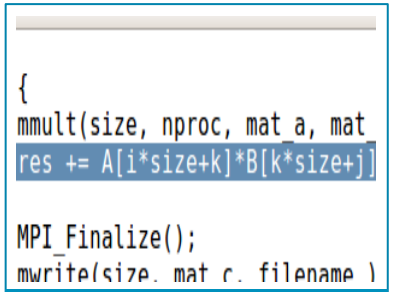
Remove I?O bottleneck



Make sure OpenMP regions make sense



Improve memory access



Restructure for vectorization

GPU Profiling

Profile Information

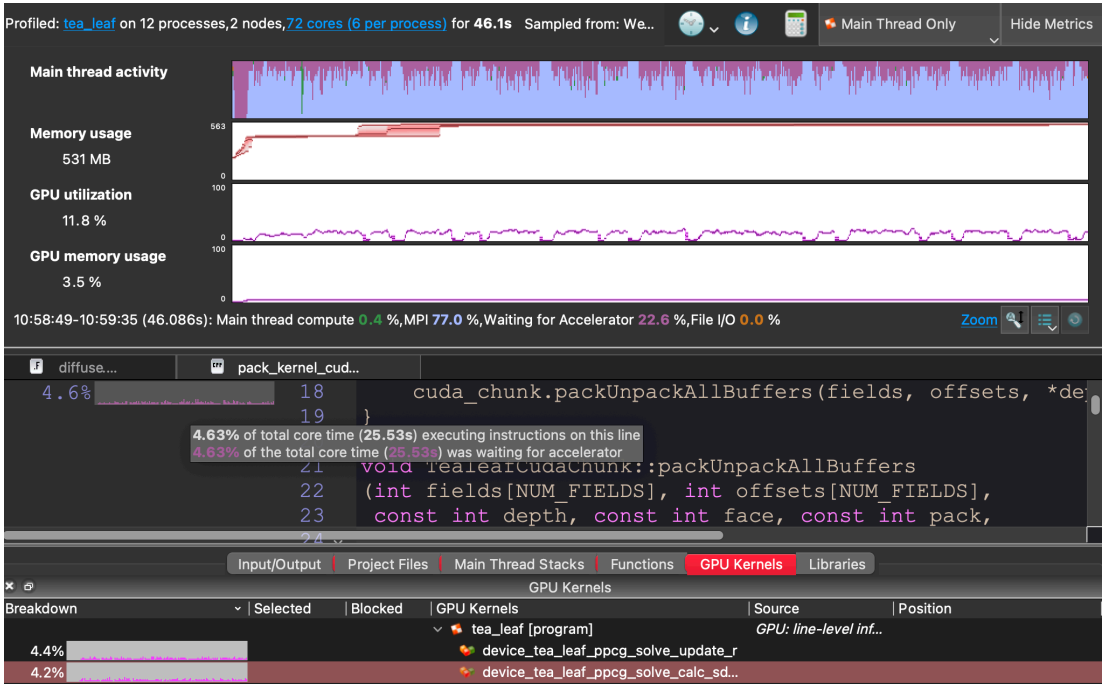
- Mixed CPU [green] / GPU [purple] application
- CPU time waiting for GPU Kernels [purple]
- GPU Kernels graph indicating Kernel activity

GUI information

- Can see a breakdown of time spent in each GPU Kernel
- Ranked by highest contributors to app time

GPU Metrics

- ***GPU Utilization***: Percent of time that the GPU card was in use
- ***GPU memory usage***: The memory allocated from the GPU Frame buffer memory



Python Profiling

19.0 adds support for Python

- Call stacks
- Time in interpreter

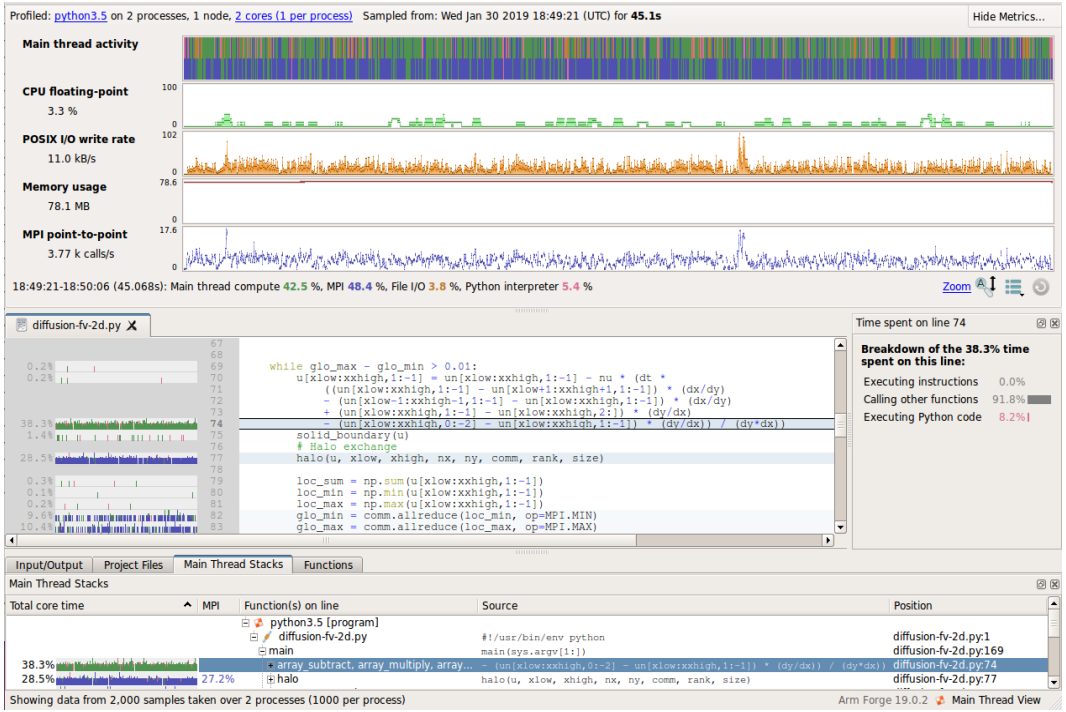
Works with MPI4PY

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

Compiler Remarks

Annotates source code with compiler remarks

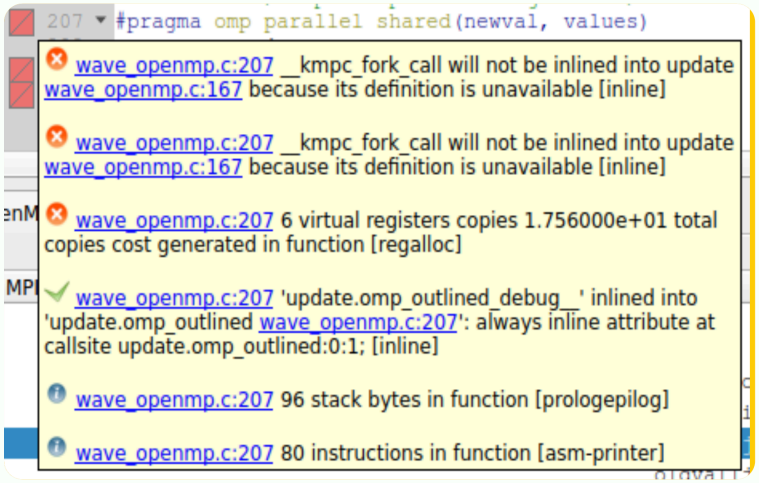
- Remarks are extracted from the compiler optimisation report
- Compiler remarks are displayed as annotations next to your source code

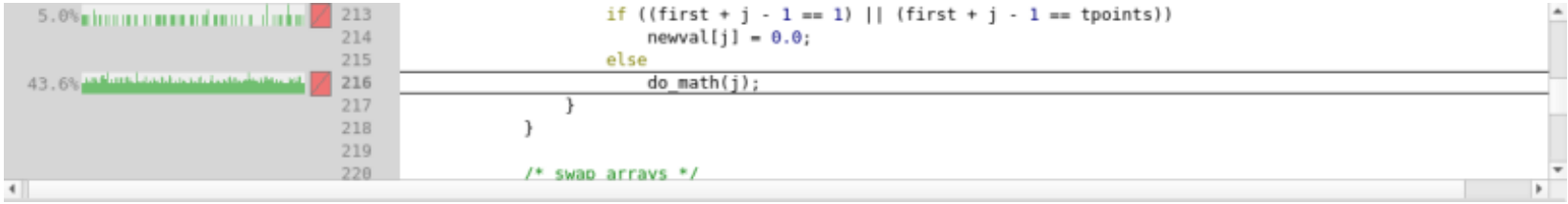
Colour coded

- 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





MAP Thread Affinity Advisor

Snapshot Selector
Change at which point of a run the Affinity data is shown (*Library Load, Initialisation, Finalization*).

Exemplar Nodes
Selectable list of exemplars, allowing ability to switch data between nodes of a run. Nodes with similar affinity/structures are merged.

Global (launcher) environment variables
List of Environment Variables which were set at launch which might be relevant to how threads are distributed.

Processes List
List of processes (by MPI rank) of the selected exemplar. Shows the key for the node topology diagram and selecting one shows all threads for the process.

Process-specific env vars
List of Environment Variables which might affect the affinity of a given rank.

Threads List
List of all threads for the selected process. Selecting threads highlights which cores they are bound to in the topology view.

Commentary
A list of commentary, providing information and advice on Memory Imbalance, Core Utilization etc.

Initial

Summary: mmult_c is Compute-bound in this configuration

Compute	86.5%	38.5s	Time spent running compiled application code. High values are usually good. This is **high**; check the CPU performance section for advice
MPI	13.4%	6.0s	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	<0.1%	0.0s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 86.5% (38.5s) CPU time:

Scalar numeric ops	51.9%	20.0s
Vector numeric ops	0.0%	0.0s
Memory accesses	48.1%	18.5s

The per-core performance is arithmetic-bound. Try to increase the amount of time spent in vectorized instructions by analyzing the compiler's vectorization reports.

Significant time is spent on memory accesses. Use a profiler to identify time-consuming loops and check their cache performance.

CPU Metrics

Configured Linux perf event metrics:

cache-misses 68.3 M/s

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	28.6%	
System load	28.3%	

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	245 MiB
Peak process memory usage	444 MiB
Peak node memory usage	9.0%

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

MPI

A breakdown of the 13.4% (6.0s) MPI time:

Time in collective calls	95.2%	5.7s
Time in point-to-point calls	4.8%	0.3s
Effective process collective rate	0.00 bytes/s	
Effective process point-to-point rate	673 MB/s	

I/O

A breakdown of the <0.1% (0.0s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	0.0s
Effective process read rate	0.00 bytes/s	
Effective process write rate	420 MB/s	

Most of the time is spent in write operations with an average effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

Energy

A breakdown of how the 1.07 Wh was used:

CPU	100.0%
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

The whole system energy has been calculated using the CPU energy usage.

System power metrics: Cray power not supported

Reordered for loops

Summary: mmult_c is MPI-bound in this configuration

Compute	47.3%	5.0s	Time spent running compiled application code. High values are usually good. This is **low**; consider improving MPI or I/O performance first
MPI	52.4%	5.5s	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	0.3%	0.0s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

This application run was MPI-bound. A breakdown of this time and advice for investigating further is in the MPI section below.

CPU

A breakdown of the 47.3% (5.0s) CPU time:

Scalar numeric ops	40.7%	2.0s
Vector numeric ops	0.0%	0.0s
Memory accesses	59.3%	3.0s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

CPU Metrics

Configured Linux perf event metrics:

cache-misses 4.03 M/s

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	28.5%	
System load	28.6%	

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	213 MiB
Peak process memory usage	428 MiB
Peak node memory usage	9.0%

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

MPI

A breakdown of the 52.4% (5.5s) MPI time:

Time in collective calls	95.0%	5.2s
Time in point-to-point calls	5.0%	0.3s
Effective process collective rate	0.00 bytes/s	
Effective process point-to-point rate	636 MB/s	

I/O

A breakdown of the 0.3% (0.0s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	0.0s
Effective process read rate	0.00 bytes/s	
Effective process write rate	445 MB/s	

Most of the time is spent in write operations with an average effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

Energy

A breakdown of how the 0.267 Wh was used:

CPU	100.0%
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

The whole system energy has been calculated using the CPU energy usage.

System power metrics: Cray power not supported

Vectorised loops

Summary: mmult_c is MPI-bound in this configuration

Compute	42.2%	3.9s	Time spent running compiled application code. High values are usually good. This is **low**; consider improving MPI or I/O performance first
MPI	57.5%	5.4s	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	0.2%	0.0s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

This application run was MPI-bound. A breakdown of this time and advice for investigating further is in the MPI section below.

CPU

A breakdown of the 42.2% (3.9s) CPU time:

Scalar numeric ops	16.0%	0.6s
Vector numeric ops	21.1%	0.8s
Memory accesses	63.0%	2.5s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

CPU Metrics

Configured Linux perf event metrics:

cache-misses 8.89 M/s

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	28.5%	
System load	27.7%	

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	209 MiB
Peak process memory usage	441 MiB
Peak node memory usage	9.0%

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

MPI

A breakdown of the 57.5% (5.4s) MPI time:

Time in collective calls	96.2%	5.2s
Time in point-to-point calls	3.8%	0.2s
Effective process collective rate	0.00 bytes/s	
Effective process point-to-point rate	841 MB/s	

I/O

A breakdown of the 0.2% (0.0s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	0.0s
Effective process read rate	0.00 bytes/s	
Effective process write rate	619 MB/s	

Most of the time is spent in write operations with an average effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

Energy

A breakdown of how the 0.227 Wh was used:

CPU	100.0%
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

The whole system energy has been calculated using the CPU energy usage.

System power metrics: Cray power not supported

Cheat sheet

Training material

*cp /cluster/vi-hps_tuning_workshop/examples/linaro/*linaro-forge-training.tar.gz .
tar -xf linaro-forge-training.tar.gz

Training slides

/cluster/vi-hps_tuning_workshop/slides/FORGE.pdf

Forge Client (On local machine)

Install Forge client *https://www.linaroforge.com/downloadForge*

Running with a batch script

./<FORGE_TRAINING>/scripts/submit-slurm.sh

Linux Perf metrics

map —list-target-hosts
<forge-installdirectory>/bin/forge-probe --install=user
map —perf-metrics=list

Forge commands

map --profile # offline profile
perf-report # performance report

Guides

Forge userguide

Running

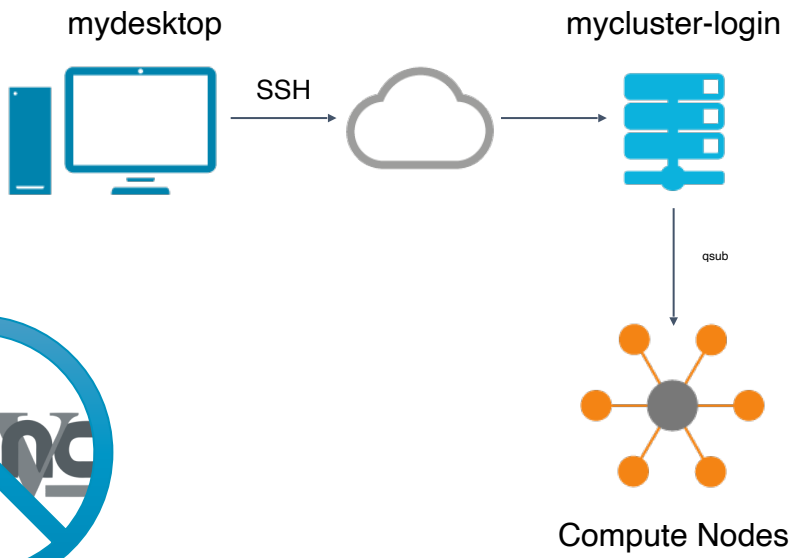
salloc --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen --time=00:20:00 --ntasks-per-node=72 --nodes=2

export PATH=$PATH:/cluster/vi-hps_tuning_workshop/examples/linaro/forge/25.0/bin

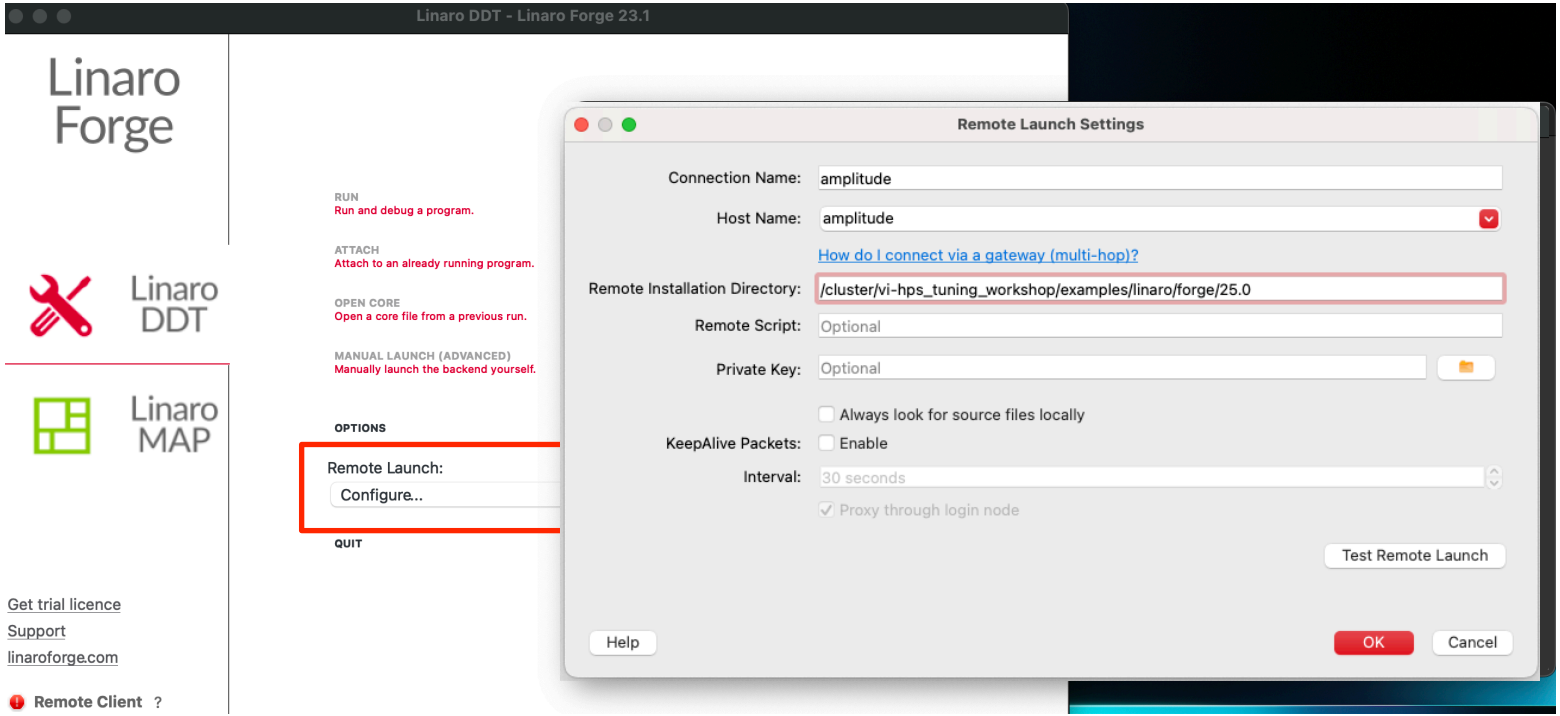
~/linaro/forge/25.0/bin/map --profile --np=4 --mpi=generic ./mmult_c 3072

The Forge GUI and where to run it

Forge provides a powerful GUIs that can be run in a variety of configurations.



Remote connection to amplitude



Matrix Multiplication example

Build and run matrix multiplication example

https://docs.linaroforge.com/latest/html/forge/worked_examples_appendix/mmult/analyze.html

```
module load openmpi/5.0.3-intel24
export FORGE_TRAINING=<linaro-forge-training>

# Build C and Fortran Examples
 cd $FORGE_TRAINING/performance
 make -f mmult.makefile

# Offline profile
 sbatch submit-slurm.sh

map --profile --np=4 --mpi=generic ./mmult_c 3072
```

Thank You

rudy.shand@linaro.org