

Linaro Forge

Performance Engineering with Linaro PR and Linaro MAP

Rudy Shand - Services and Support Director
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
Linaro DDT



Profile
Linaro MAP



Analyse
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

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

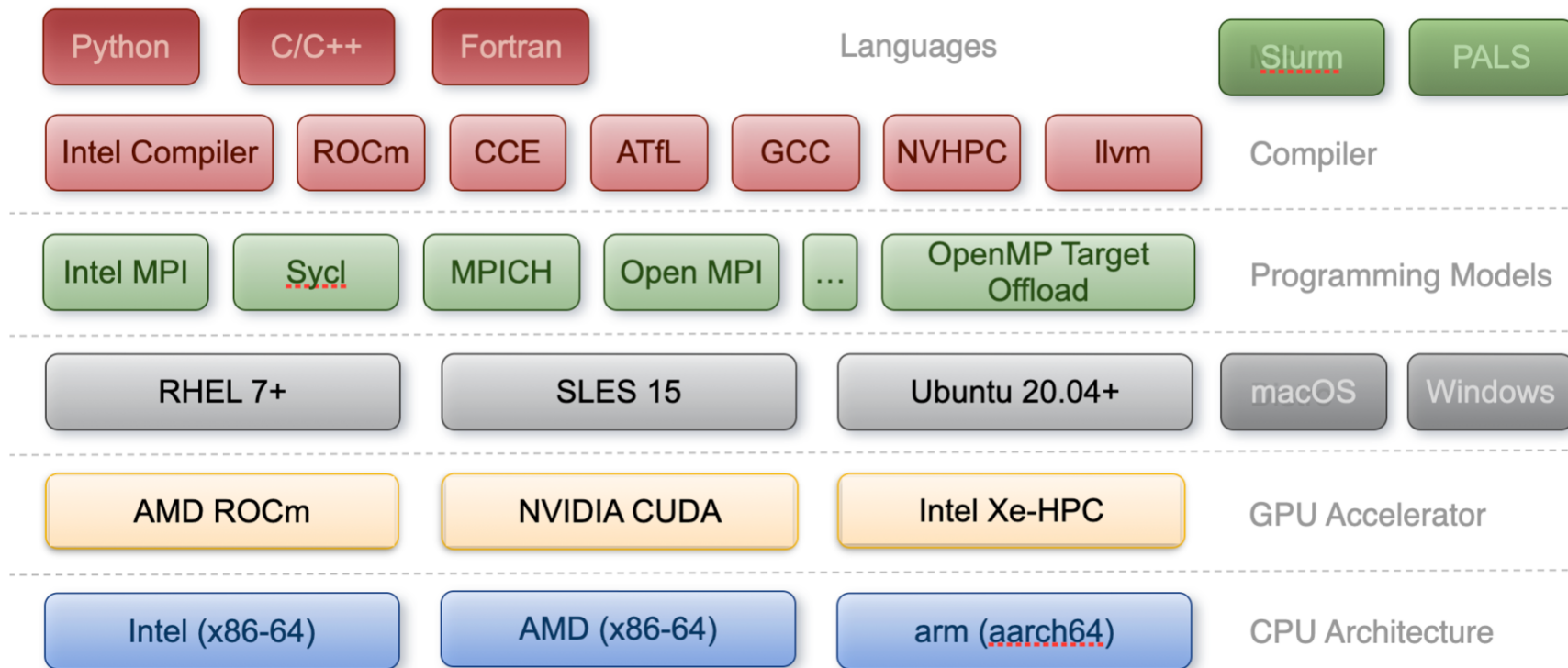


Relevant advice
to avoid pitfalls

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)

Supported Platforms



GDB under the hood

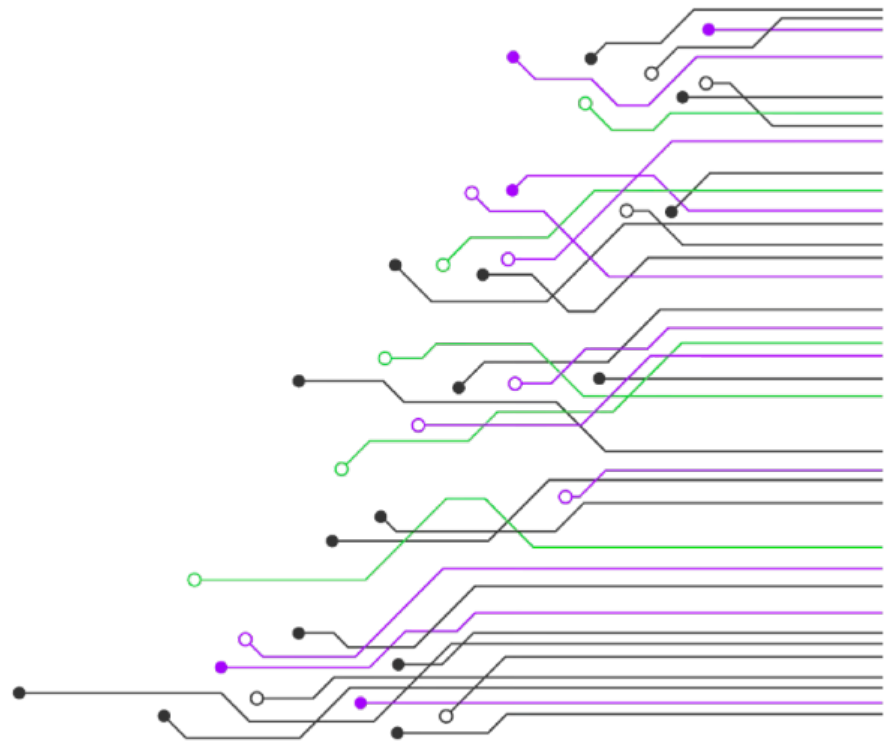
Leveraging the gdb community

GDB is the underlying technology

- Linaro upstreams patches to the gdb community
- Forge team raises and fixes gdb bugs

Nimble at supporting new technologies

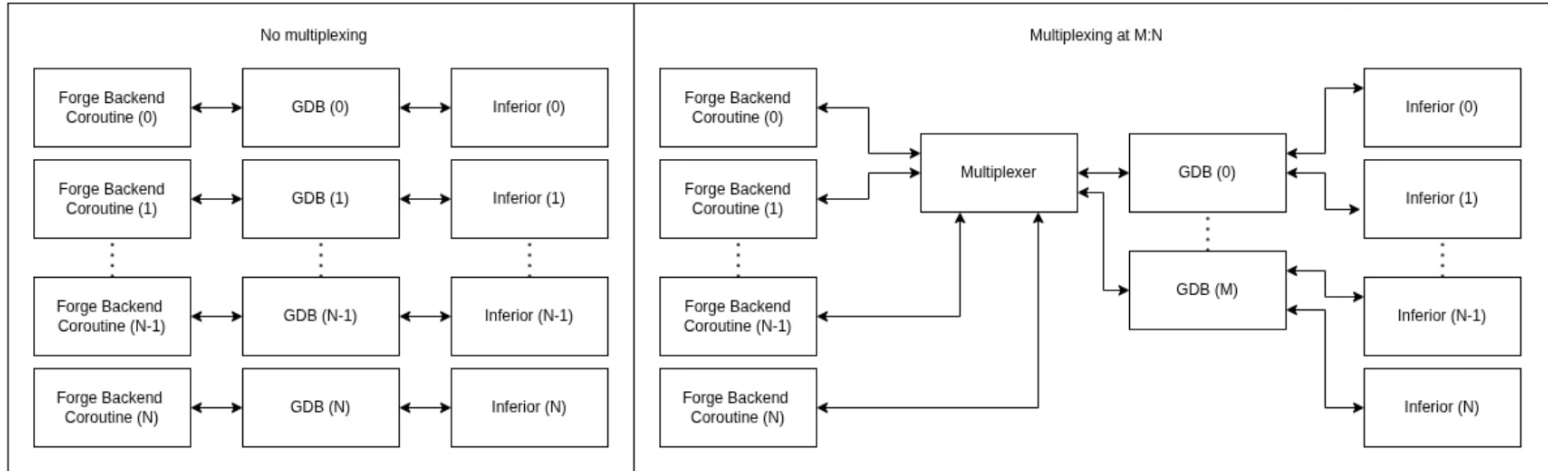
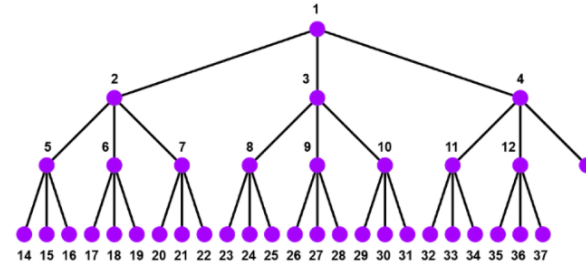
- Rely on hardware vendors to add gdb support
- Rely on software consortiums to add gdb support
- Helps us to stay current with technologies
- Provides a state of the art debugger



Scaling GDB

Tree network topology

- Tree server design is how Forge is able to scale
- Send bulk commands and merge responses
- Aggregate the data instead of broadcasting thousands of responses
- Configurable gdb inferior multiplexing



Linaro Performance Reports

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

Command: `mpirun.hydra -host node-1,node-2 -map-by socket -n 16 -ppn 8 ./Bin/low_freq/../../Src//hydro -i ./Bin/low_freq/../../Src//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% 

MPI 63.2% 

I/O 16.2% 

Time spent running application code. High values are usually good.
This is **very low**; focus on improving MPI or I/O performance first

Time spent in MPI calls. High values are usually bad.
This is **high**; check the MPI breakdown for advice on reducing it

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 |

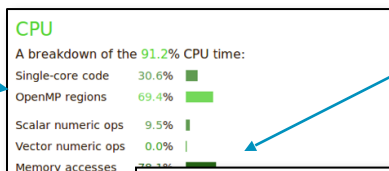
Effective process write rate 1.38 MB/s 

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

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 significant synchronization overhead in the MPI profiler.

OpenMP

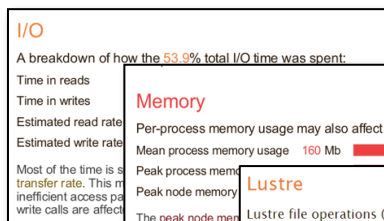
A breakdown of the 99.5% time in OpenMP regions:

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.

Load
imbalance



Memory

Per-process memory usage may also affect scaling:

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

The peak node memory is the total number of processes and more.

Lustre

Lustre file operations (per node)

Mean write rate	
Peak write rate	
Mean file operations	
Mean metadata	

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.

OMP
efficiency
System
usage

MAP Capabilities

MAP is a sampling based scalable profiler

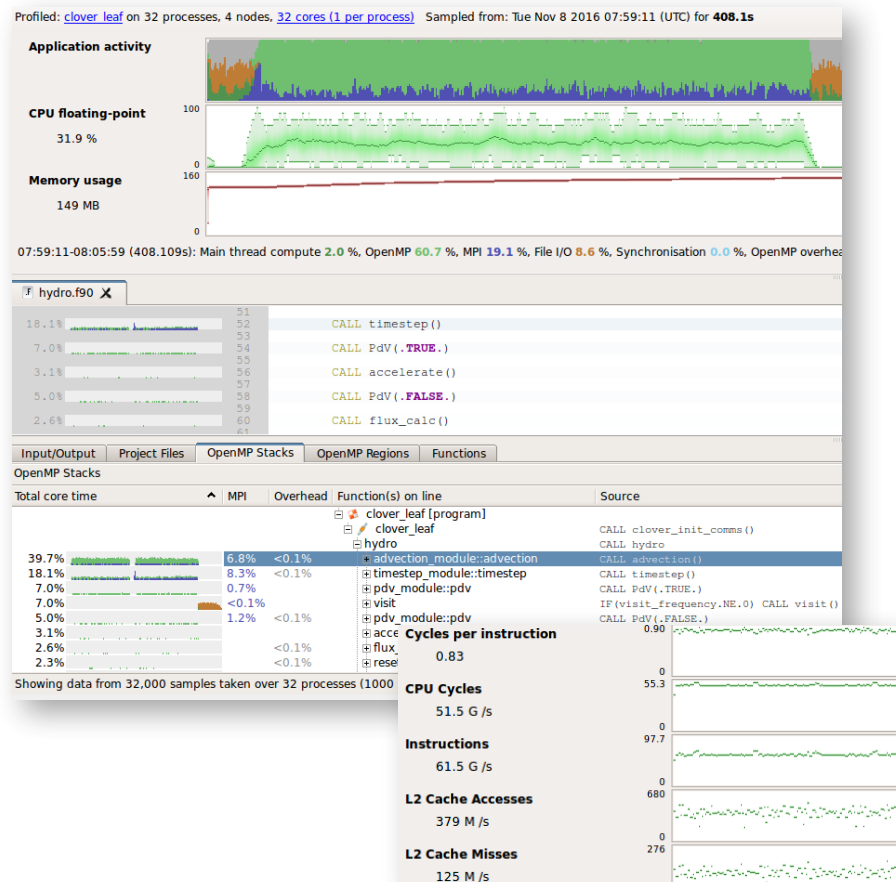
- Built on same framework as DDT
- Parallel support for MPI, OpenMP, CUDA
- Designed for C/C++/Fortran

Designed for 'hot-spot' analysis

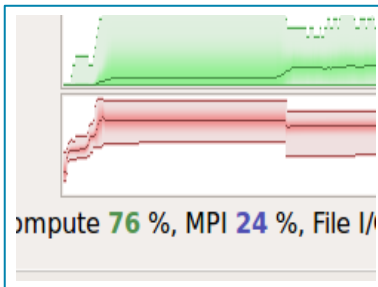
- Stack traces
- Augmented with performance metrics

Adaptive sampling rate

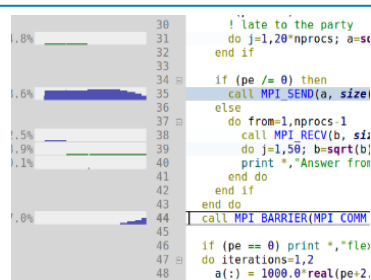
- Throws data away - 1,000 samples per process
- Low overhead, scalable and small file size



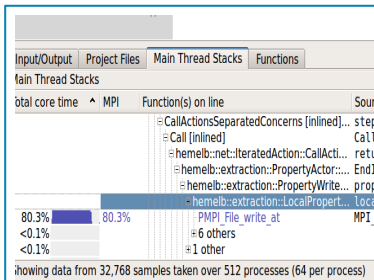
Linaro MAP Source Code Profiler Highlights



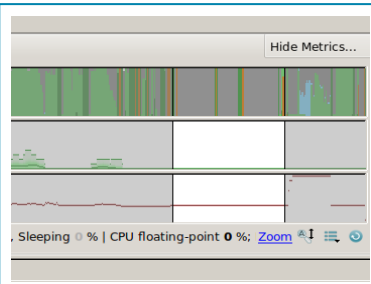
Find the peak memory use



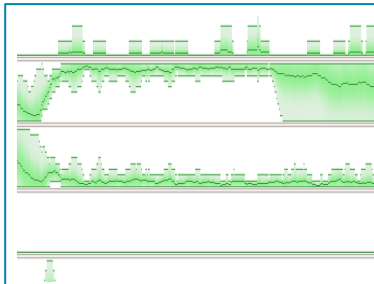
Fix an MPI imbalance



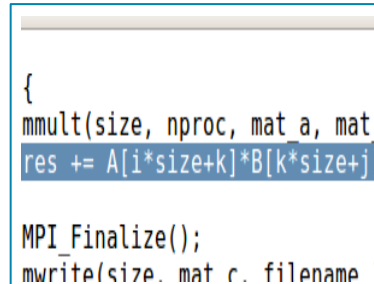
Remove I/O bottleneck



Make sure OpenMP regions make sense



Improve memory access



Restructure for vectorization

Case Study: Fire Dynamics Simulator (FDS)

Issue that we encountered with AWS when porting an app called FDS: <https://github.com/firemodels/fds>.

Software for fire & smoke modelling

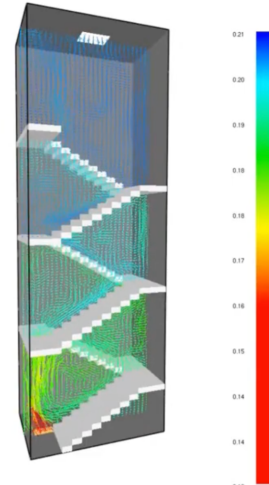
Collaboration between Arm, AWS and NIST

Porting the application from x86_64 to Arm Graviton.

Spotted an inefficient MPI call routine using MAP

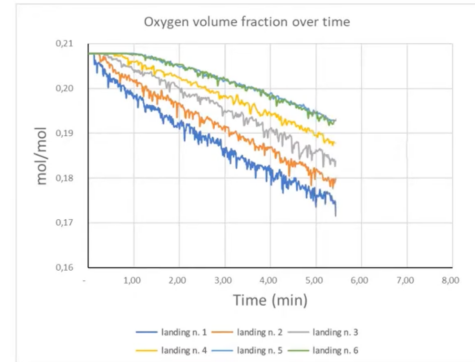
Fair amount of time that was spent on this “inquire” line.

Oxygen Volume Fraction (mol/mol)

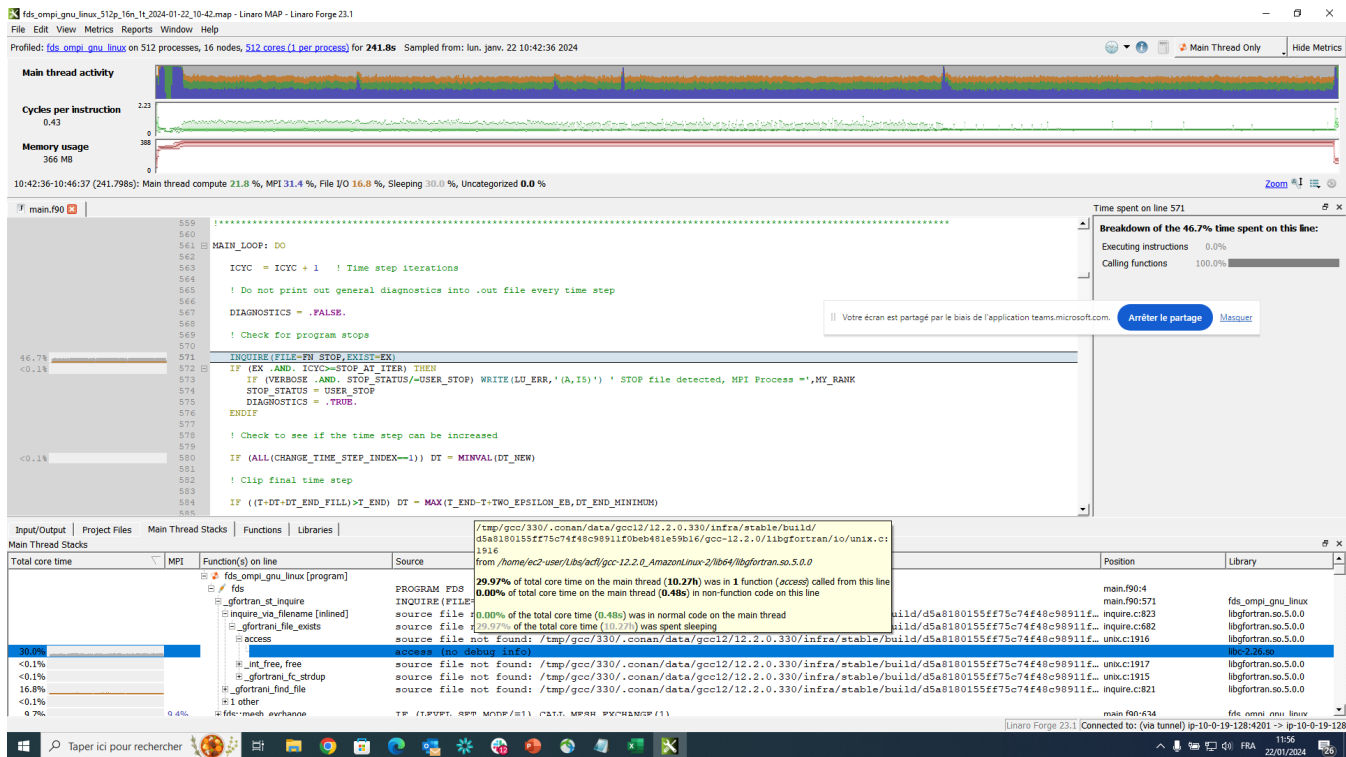


Simulation inputs:

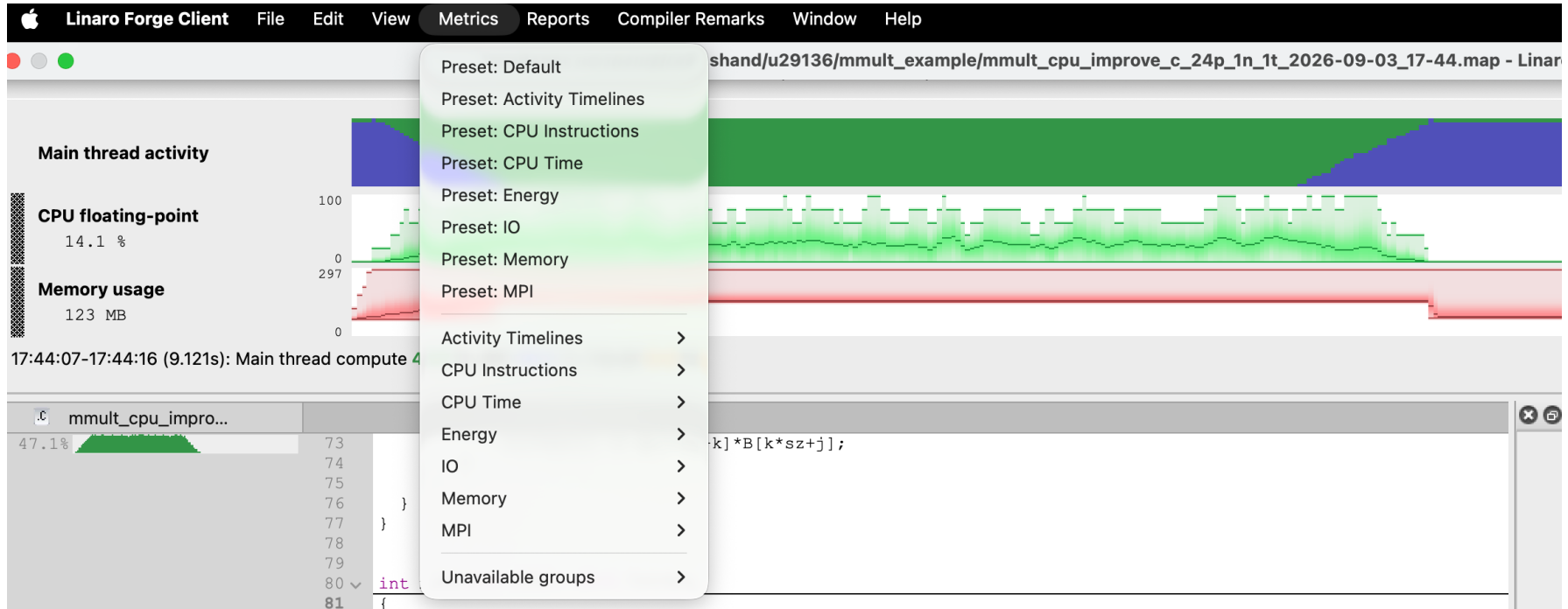
- Soot yield = 0,03 kg/kg
- Carbon monoxide yield = 0,07 kg/kg
- Heat Release Rate = 120 kW



Spotting the bottleneck with MAP

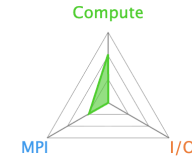


Adding metrics to MAP



linaro PERFORMANCE REPORTS

Command: `mpirun mmult_c 3072`
 Resources: 1 node (96 physical, 192 logical cores per node)
 Memory: 503 GiB per node
 Tasks: 24 processes
 Machine: c0210
 Architecture: x86_64
 CPU Family: <unknown>
 Start time: Wed Sep 2 12:27:53 2026
 Total time: 16 seconds
 Full path: `/mnt/vast-nhr/home/rudy.shand/u29136/mmult_example`



Summary: mmult_c is **Compute-bound** in this configuration

Compute 68.5% 10.8s

Time spent running compiled application code. High values are usually good.

This is **average**; check the CPU performance section for advice

MPI 31.2% 4.9s

Time spent in MPI calls. High values are usually bad.

This is **average**; 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 **Compute-bound** (based on main thread activity). A breakdown of this time and advice for investigating further is in the **CPU** section below.

CPU Metrics issue

CPU

A breakdown of the 68.5% (10.8s) CPU time:

Scalar numeric ops	26.6%	2.9s	
Vector numeric ops	0.0%	0.0s	
Memory accesses	73.4%	7.9s	

The percentage of time spent in memory access operations, such as mov, load, store. A portion of the time spent in instructions that use indirect addressing is also included here. A high figure here shows the application is memory-bound and is not able to take full advantage of the CPU resources. Often it is possible to reduce this figure by analyzing loops for poor cache performance and problematic memory access patterns, improving performance significantly. A high percentage of time spent in memory accesses in an OpenMP program is often a scalability problem. If each core spends most of its time waiting for memory, or the L3 cache, then adding further cores rarely improves matters. Equally, false sharing, in which cores block attempts to access the same cache lines, and the over-use of the atomic pragma, show up as increased time spent in memory accesses.

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.

MPI

A breakdown of the 31.2% (4.9s) MPI time:

Time in collective calls	76.9%	3.8s	
Time in point-to-point calls	23.1%	1.1s	
Effective process collective rate	0.00 bytes/s		
Effective process point-to-point rate	146 MB/s		

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s	
Synchronization	0.0%	0.0s	
Physical core utilization	24.9%		
System load	126.2%		

No measurable time is spent in multithreaded code.

The system load is high – multiple processes may be sharing one core.

MAP CPU Metrics Drill down

Profiled: [mmult_c](#) on 24 processes, 1 node, [24 cores \(1 per process\)](#) for 15.8s Sampled from: Wed Sep 2 12:27:53 2026

Main Thread Only Hide Metrics

Main thread activity

CPU floating-point

22.0 %

CPU integer

5.3 %

CPU memory access

75.3 %

12:27:53-12:28:03 (9.936s, 63.0% of total): Main thread compute 100.0 %, MPI 0.0 %, File I/O 0.0 %

Zoom

mmu...

```
50 FILE *f = fopen(fn, "w+");
51
52 for(int i=0; i<sz; i++)
53 {
54     for(int j=0; j<sz; j++)
55     {
56         fprintf(f, "%g\t", A[i*sz+j]);
57     }
58     fprintf(f, "\n");
59 }
60 fclose(f);
61 }
62
63
64 void mmult(int sz, int nslices, double *A, double *B, double *C)
65 {
66     for(int i=0; i<sz/nslices; i++)
67     {
68         for(int j=0; j<sz; j++)
69         {
70             double res = 0.0;
71
72             for(int k=0; k<sz; k++)
73             {
74                 res += A[i*sz+k]*B[k*sz+j];
75             }
76
77             C[i*sz+j] += res;
78         }
79     }
80 }
81 }
```

100.0%

<0.1%

Time spent on line 75

Breakdown of the 100.0% time spent on this line:

Executing instructions	100.0%
Calling functions	0.0%

Time in instructions executed:

Scalar floating-point	22.4%
Vector floating-point	0.0%
Scalar integer	5.4%
Vector integer	0.0%
Memory access*	75.7%
Branch	0.0%
Other instructions	0.0%

* 72.2% memory access instructions, 3.5% implicit memory accesses in other instructions, also counted in their categories

Fixing the cache bottleneck

Profiled: [mmult_cpu_improve_c](#) on 24 processes, 1 node, [24 cores \(1 per process\)](#) for 9.1s Sampled from: Thu Sep 3 17:44:07 2026

🕒 ⓘ 📅 🗑️ Main Thread Only Hide Metrics

Main thread activity

CPU floating-point

28.8 %

CPU integer

17.4 %

CPU memory access

60.5 %

17:44:07-17:44:11 (3.800s, 41.7% of total): Main thread compute 100.0 %, MPI 0.0 %, File I/O 0.0 %

Zoom 🔍 ⌵ ⌶ ⌵ ⌵ ⌵

mmult_cpu_impro...

```
56     fprintf(f, "%g\t", A[i*sz+j]);
57 }
58 fprintf(f, "\n");
59 }
60 fclose(f);
61 }
62 }
63
64
65 void mmult(int sz, int nslices, double *A, double *B, double *C)
66 {
67     for(int i=0; i<sz/nslices; i++)
68     {
69         for(int k=0; k<sz; k++)
70         {
71             for(int j=0; j<sz; j++)
72             {
73                 C[i*sz+j] += A[i*sz+k]*B[k*sz+j];
74             }
75         }
76     }
77 }
```

<0.1%
2.4%
97.5%

Time spent on line 73

Breakdown of the 97.5% time spent on this line:

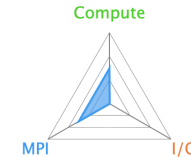
Executing instructions 100.0%
Calling functions 0.0%

Time in instructions executed:

Scalar floating-point 29.5%
Vector floating-point 0.0%
Scalar integer 15.9%
Vector integer 0.0%
Memory access* 59.6%
Branch 0.0%
Other instructions 2.2%

linaro PERFORMANCE REPORTS

Command: mpirun mmult_cpu_improve_c 3072
 Resources: 1 node (96 physical, 192 logical cores per node)
 Memory: 503 GiB per node
 Tasks: 24 processes
 Machine: c0301
 Architecture: x86_64
 CPU Family: <unknown>
 Start time: Thu Sep 3 17:44:07 2026
 Total time: 10 seconds
 Full path: /mnt/vast-nhr/home/rudy.shand/u29136/
 mmult_example



Summary: mmult_cpu_improve_c is **MPI-bound** in this configuration

Compute 49.7% 4.5s

Time spent running compiled application code. High values are usually good.

This is **low**; consider improving MPI or I/O performance first

MPI 49.9% 4.6s

Time spent in MPI calls. High values are usually bad.

This is **average**; check the MPI breakdown for advice on reducing it

I/O 0.4% 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** (based on main thread activity). A breakdown of this time and advice for investigating further is in the **MPI** section below.

MPI issue

MPI

A breakdown of the **49.9% (4.6s)** MPI time:

Time in collective calls	83.4%	3.8s	
--------------------------	-------	------	-------------

The percentage of time spent in collective MPI operations such as `MPI_Scatter`, `MPI_Reduce`, and `MPI_Barrier`.

Time in point-to-point calls	16.6%	0.8s	
------------------------------	-------	------	-------------

Effective process collective rate	0.00 bytes/s	
-----------------------------------	--------------	--

Effective process point-to-point rate	212 MB/s	
---------------------------------------	----------	-------------

CPU

A breakdown of the **49.7% (4.5s)** CPU time:

Scalar numeric ops	43.0%	1.9s	
--------------------	-------	------	-------------

Vector numeric ops	<0.1%	0.0s	
--------------------	-------	------	--

Memory accesses	56.9%	2.6s	
-----------------	-------	------	-------------

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.

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s	
-------------	------	------	--

Synchronization	0.0%	0.0s	
-----------------	------	------	--

Physical core utilization	24.9%	
---------------------------	-------	-------------

System load	73.5%	
-------------	-------	-------------

No measurable time is spent in multithreaded code.

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

I/O

A breakdown of the **0.4% (0.0s)** I/O time:

Time in reads	0.0%	0.0s	
---------------	------	------	--

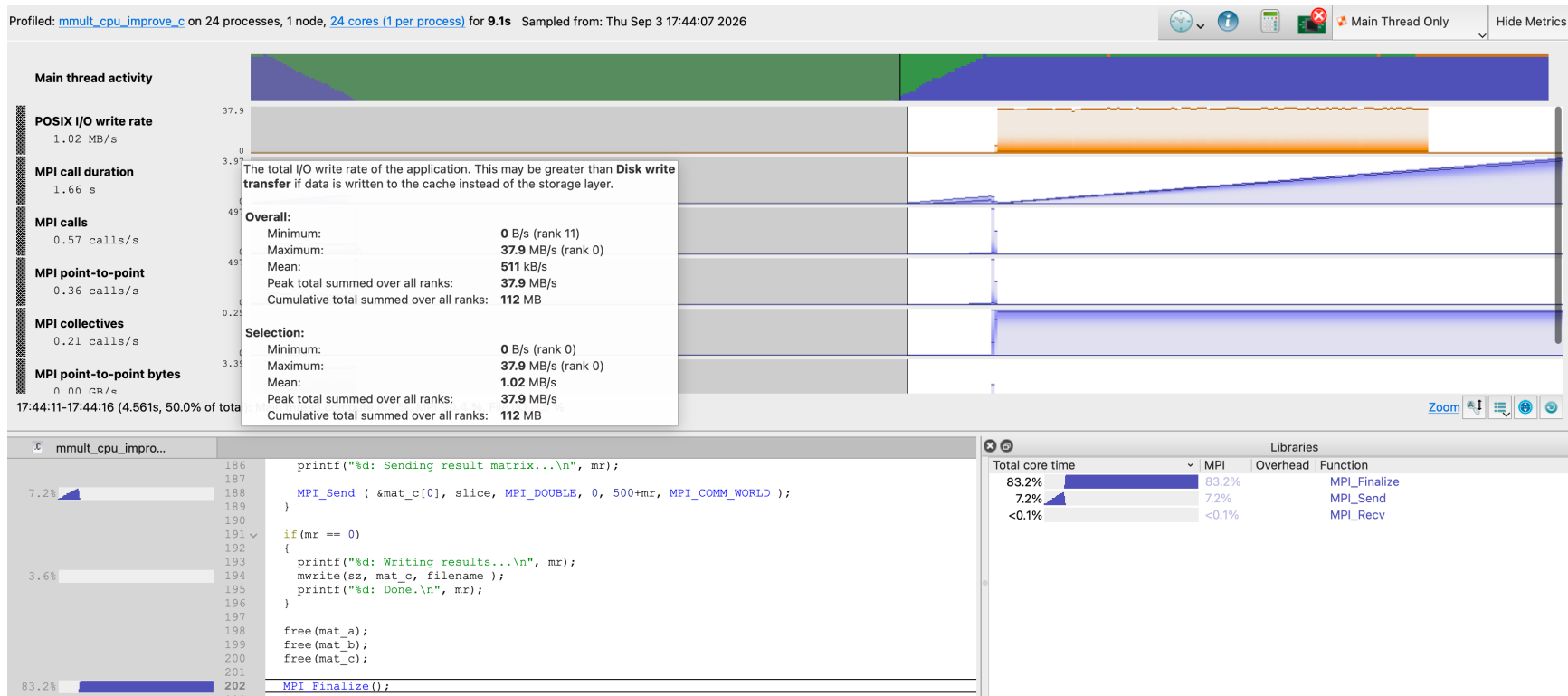
Time in writes	100.0%	0.0s	
----------------	--------	------	-------------

Effective process read rate	0.00 bytes/s	
-----------------------------	--------------	--

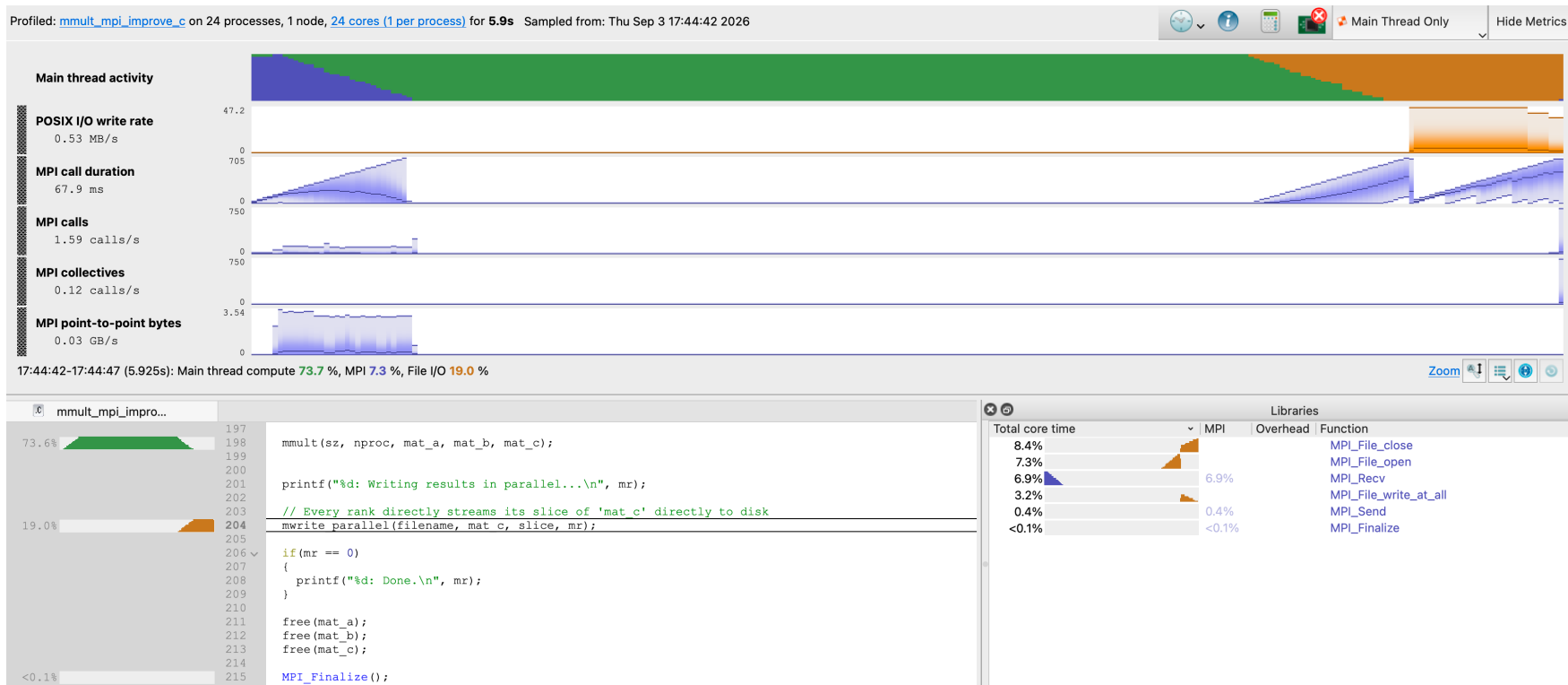
Effective process write rate	114 MB/s	
------------------------------	----------	-------------

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.

MAP MPI Metrics Drill down



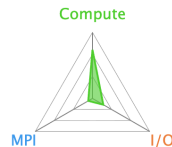
Fixing the MPI bottleneck



linaro

PERFORMANCE REPORTS

Command: `mpirun mmult_mpi_improve_c 3072`
 Resources: 1 node (96 physical, 192 logical cores per node)
 Memory: 503 GiB per node
 Tasks: 24 processes
 Machine: c0301
 Architecture: x86_64
 CPU Family: <unknown>
 Start time: Thu Sep 3 17:44:42 2026
 Total time: 6 seconds
 Full path: /mnt/vast-nhr/home/rudy.shand/u29136/mmult_example



Summary: `mmult_mpi_improve_c` is **Compute-bound** in this configuration

Compute 73.7% 4.4s

MPI 7.3% 0.4s

I/O 19.0% 1.1s

Time spent running compiled application code. High values are usually good.

This is **high**; check the CPU performance section for advice

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 **average**; check the I/O breakdown section for optimization advice

This application run was **Compute-bound** (based on main thread activity). 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.

Threads issue

Threads

A breakdown of how multiple threads were used:

Computation 0.0% 0.0s |

Synchronization 0.0% 0.0s |

Physical core utilization 25.0% ■

Modern CPUs often have multiple logical cores for each physical core. This is often referred to as hyper-threading. These logical cores can share logic and arithmetic units. Some programs perform better when using additional logical cores, but most HPC codes do not. The value here shows the percentage utilization of physical cores. A value over 100% indicates that more threads are executing than there are physical cores, indicating that hyper-threading is in use. Only threads actively and simultaneously consuming CPU time are included in this metric. A program can have many helper threads that do little except sleep, and are not shown.

System load 74.5% ■

No measurable time is spent in multithreaded code.

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

Thread Affinity

A breakdown of how software threads have been pinned to logical cores (2 per physical core).

Mean utilization 100.0% (48 of 48 cores utilized) ■

Max load 1.00 ■

Migration opportunity 2.00 ■

[ERROR] process imbalance detected across Packages

[ERROR] process imbalance detected across NUMA nodes

[ERROR] process imbalance detected across L3 Caches

MPI

A breakdown of the 7.3% (0.4s) MPI time:

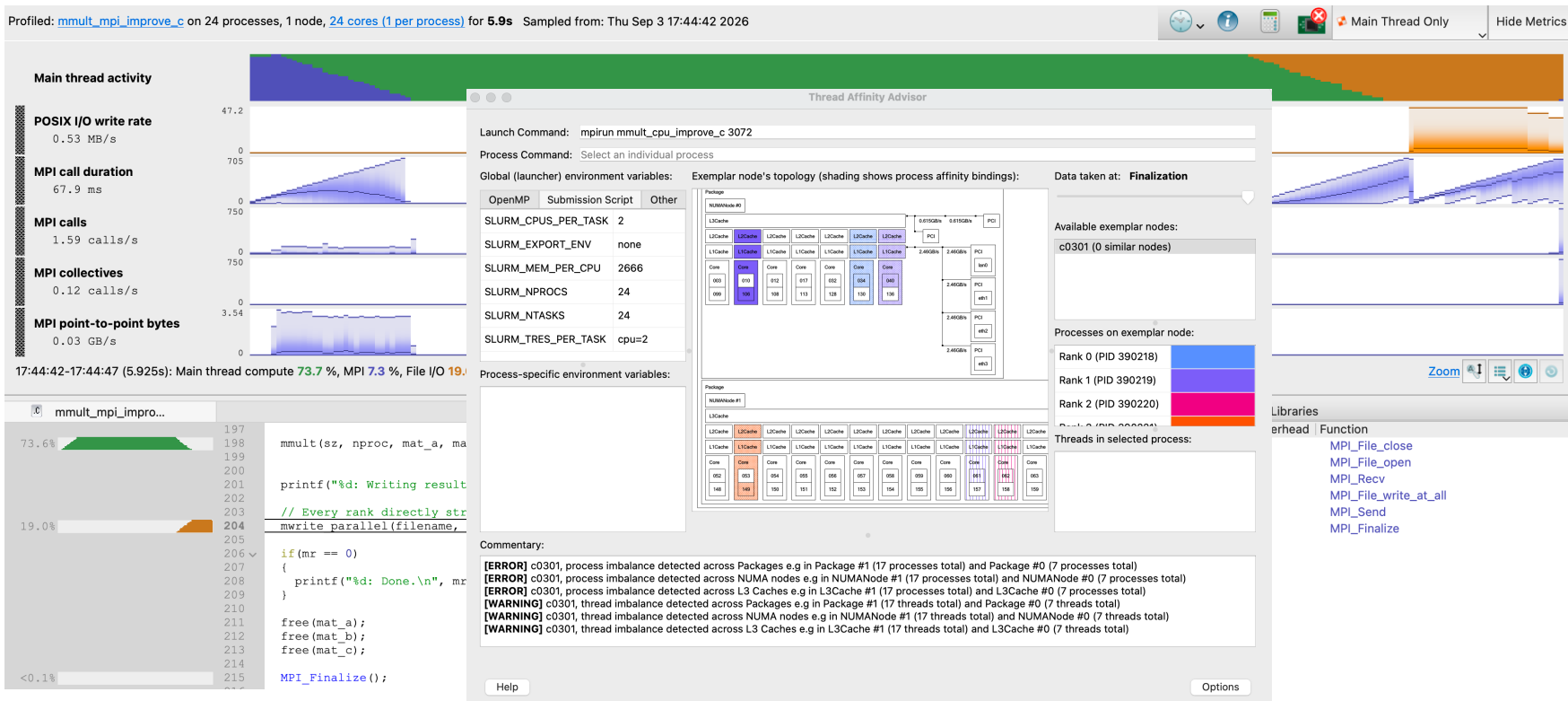
Time in collective calls 0.2% 0.0s |

Time in point-to-point calls 99.8% 0.4s ■

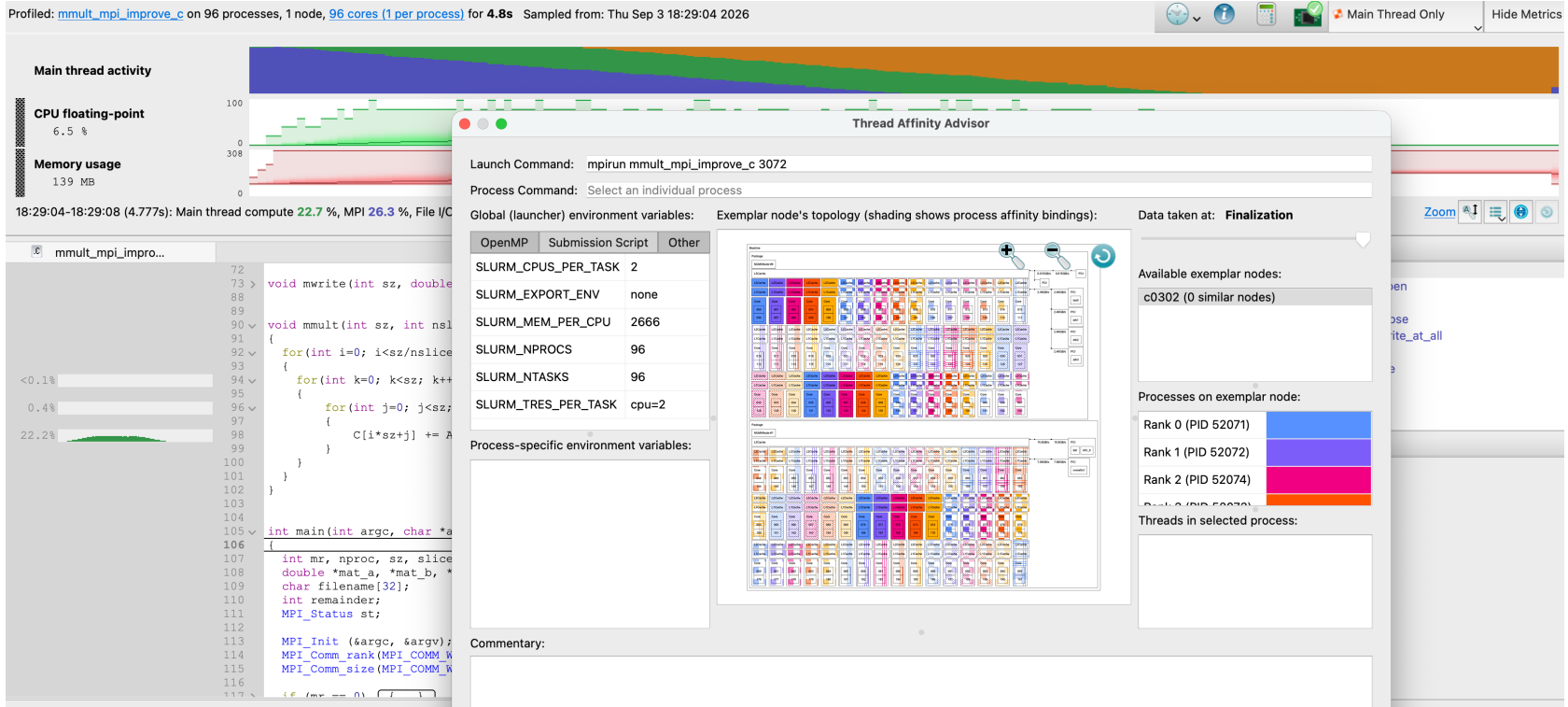
Effective process collective rate 0.00 bytes/s |

Effective process point-to-point rate 343 MB/s ■

Fixing the MPI bottleneck

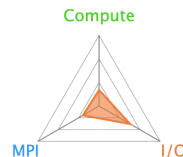


Fixing the thread issue



linaro PERFORMANCE REPORTS

Command: `mpirun mmult_mpi_improve_c 3072`
 Resources: 1 node (96 physical, 192 logical cores per node)
 Memory: 503 GiB per node
 Tasks: 96 processes
 Machine: c0302
 Architecture: x86_64
 CPU Family: <unknown>
 Start time: Thu Sep 3 18:29:04 2026
 Total time: 5 seconds
 Full path: `/mnt/vast-nhr/home/rudy.shand/u29136/mmult_example`



Summary: mmult_mpi_improve_c is **I/O-bound** in this configuration

Compute 22.7% 1.1s

MPI 26.3% 1.3s

I/O 51.0% 2.4s

Time spent running compiled application code. High values are usually good.

This is **very low**; focus on improving MPI or I/O performance first

Time spent in MPI calls. High values are usually bad.

This is **low**; this code may benefit from a higher process count

Time spent in filesystem I/O. High values are usually bad.

This is **very high**; check the I/O breakdown section for optimization advice

This application run was **I/O-bound** (based on main thread activity). A breakdown of this time and advice for investigating further is in the **I/O** section below.

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

I/O issue

I/O

A breakdown of the 51.0% (2.4s) I/O time:

Time in reads	0.0%	0.0s	
Time in writes	100.0%	2.4s	
Effective process read rate	0.00 bytes/s		
Effective process write rate	323 kB/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.

Thread Affinity

A breakdown of how software threads have been pinned to logical cores (2 per physical core).

Mean utilization	100.0%	(192 of 192 cores utilized)	
Max load	1.00		
Migration opportunity	2.00		

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s	
Synchronization	0.0%	0.0s	
Physical core utilization	100.0%		
System load	99.8%		

No measurable time is spent in multithreaded code.

CPU

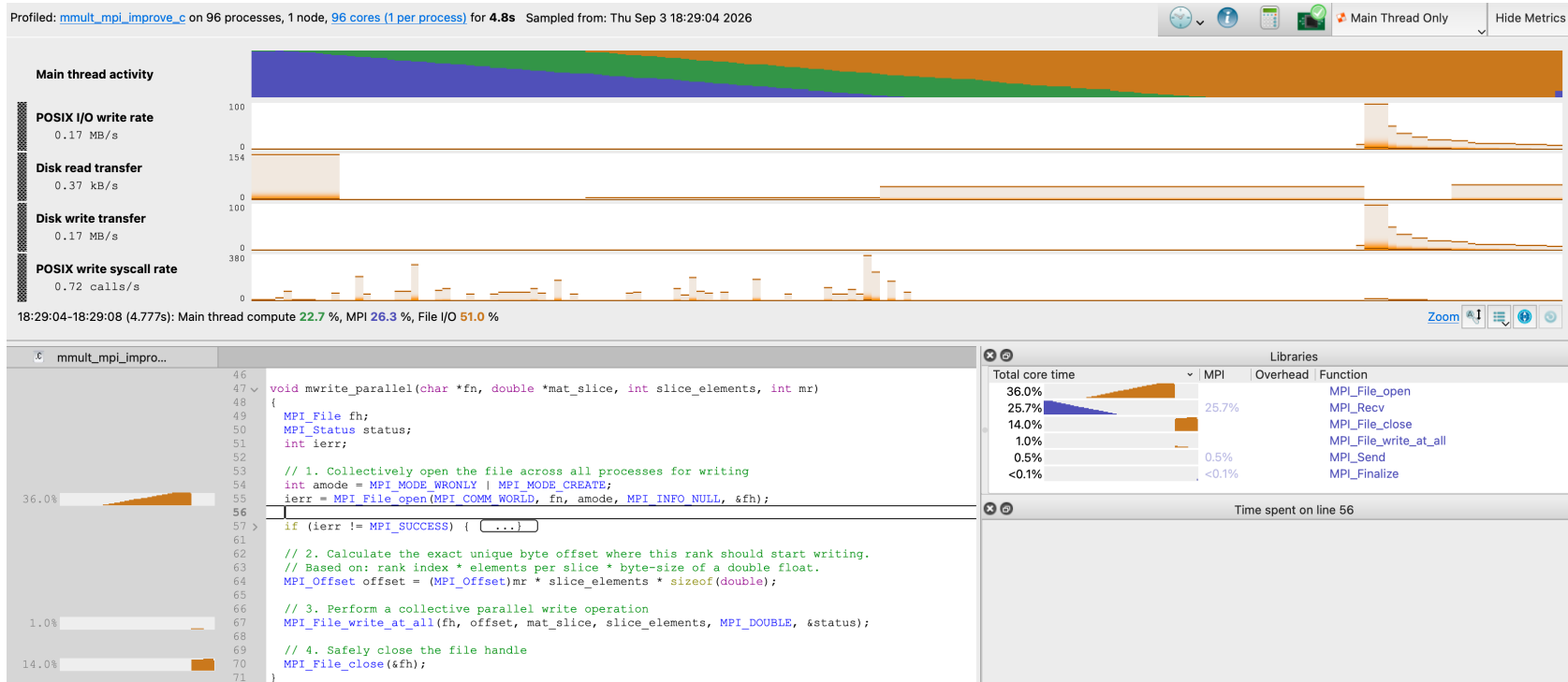
A breakdown of the 22.7% (1.1s) CPU time:

Scalar numeric ops	42.3%	0.5s	
Vector numeric ops	0.0%	0.0s	
Memory accesses	57.7%	0.6s	

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.

MAP I/O Metrics Drill down

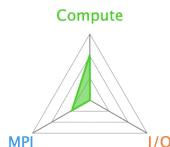


Summary

Original

linaro PERFORMANCE REPORTS

Command: mpirun mmult_c 3072
 Resources: 1 node (96 physical, 192 logical cores per node)
 Memory: 503 GiB per node
 Tasks: 24 processes
 Machine: c0210
 Architecture: x86_64
 CPU Family: <unknown>
 Start time: Wed Sep 2 12:27:53 2026
 Total time: 16 seconds
 Full path: /mnt/vast-nhr/home/rudy.shand/u29136/mmult_example



Summary: mmult_c is **Compute-bound** in this configuration

Compute 68.5% 10.8s

Time spent running compiled application code. High values are usually good.

This is **average**; check the CPU performance section for advice

MPI 31.2% 4.9s

Time spent in MPI calls. High values are usually bad.

This is **average**; 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.

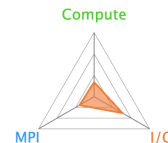
This is **very low**; however single-process I/O may cause MPI wait times

This application run was **Compute-bound** (based on main thread activity). A breakdown of this time and advice for investigating further is in the **CPU** section below.

Optimised

linaro PERFORMANCE REPORTS

Command: mpirun mmult_mpi_improve_c 3072
 Resources: 1 node (96 physical, 192 logical cores per node)
 Memory: 503 GiB per node
 Tasks: 96 processes
 Machine: c0302
 Architecture: x86_64
 CPU Family: <unknown>
 Start time: Thu Sep 3 18:29:04 2026
 Total time: 5 seconds
 Full path: /mnt/vast-nhr/home/rudy.shand/u29136/mmult_example



Summary: mmult_mpi_improve_c is **I/O-bound** in this configuration

Compute 22.7% 1.1s

Time spent running compiled application code. High values are usually good.

This is **very low**; focus on improving MPI or I/O performance first

MPI 26.3% 1.3s

Time spent in MPI calls. High values are usually bad.

This is **low**; this code may benefit from a higher process count

I/O 51.0% 2.4s

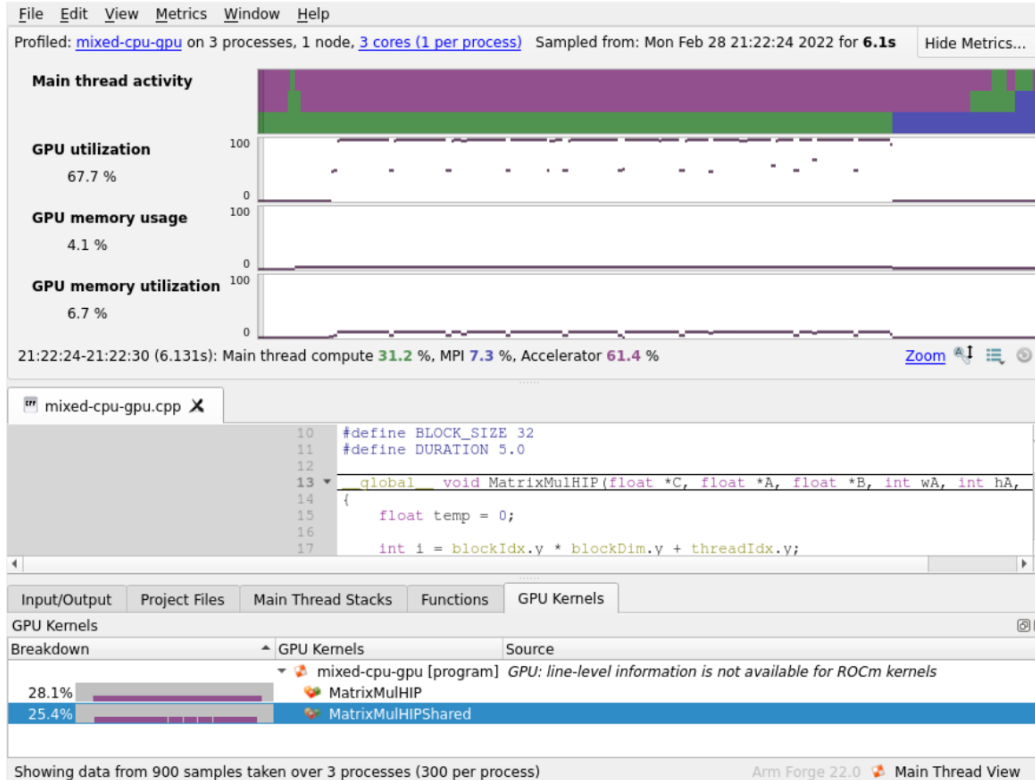
Time spent in filesystem I/O. High values are usually bad.

This is **very high**; check the I/O breakdown section for optimization advice

This application run was **I/O-bound** (based on main thread activity). A breakdown of this time and advice for investigating further is in the **I/O** section below.

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

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

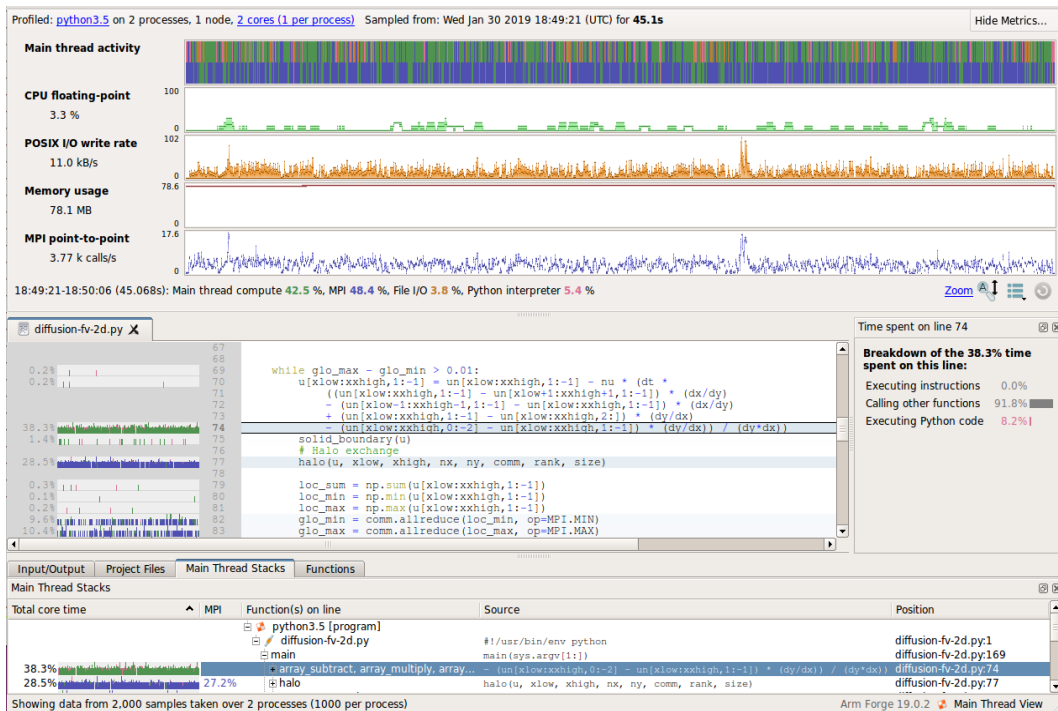
Works with MPI4PY

- 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
```

Hardware Performance Metrics

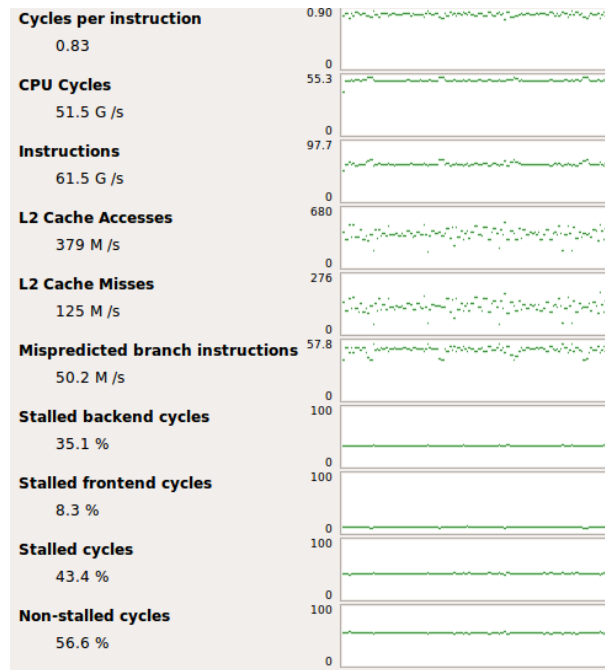
MAP uses Linux perf to gather hardware data

On x86 MAP reports on instruction mix

- CPU, vectorization, memory, etc
- Linaro are researching ways to provide the same

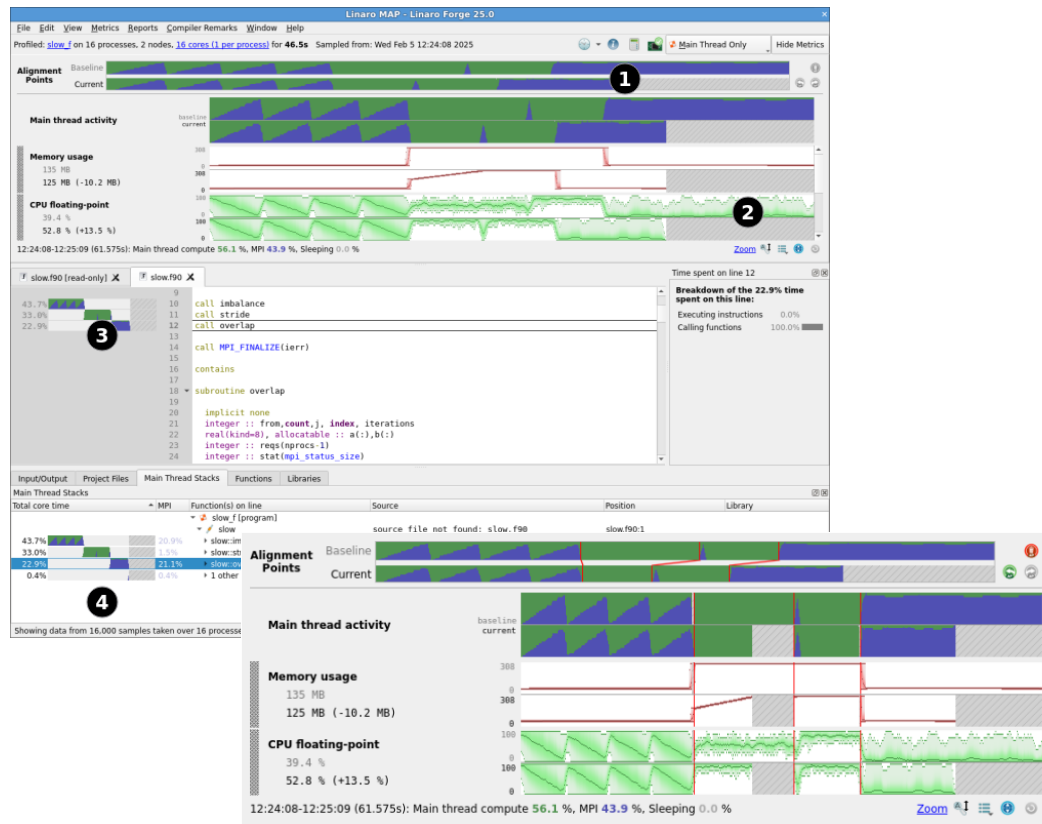
Instruction activity via perf

- Raw rates presented - not interpolated
- Need to set the perf_event_paranoid 2 or lower

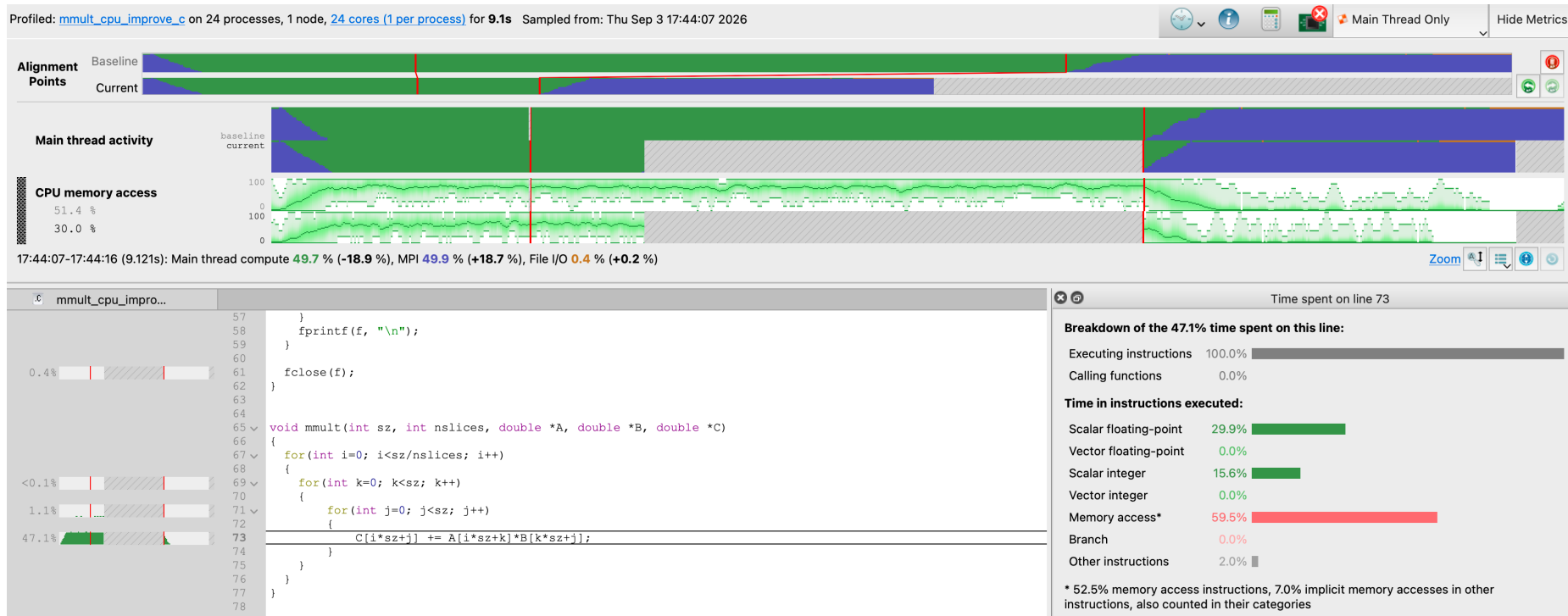


MAP Diff (--baseline support)

- MAP Diff allows comparisons of two MAP profiles, useful for identifying performance changes between different parameters, compilers, libraries and systems.
- Use the alignment points view (1) to line up phases of execution.
- Compare metric graphs of the two profiles (2), including metric summaries for each.
- See gaps in activity in the source code viewer (3) and stacks views (4), including OpenMP Regions View, Functions View, Library view and GPU Kernels/Memory Transfer View.



Diff the cache improvement



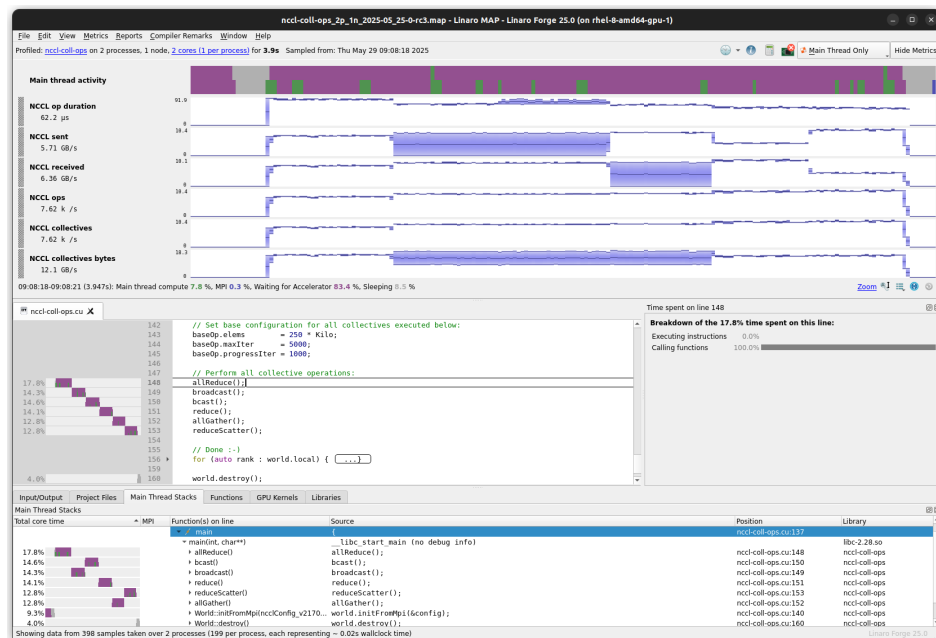
NCCL metrics for ML workloads

Prerequisites

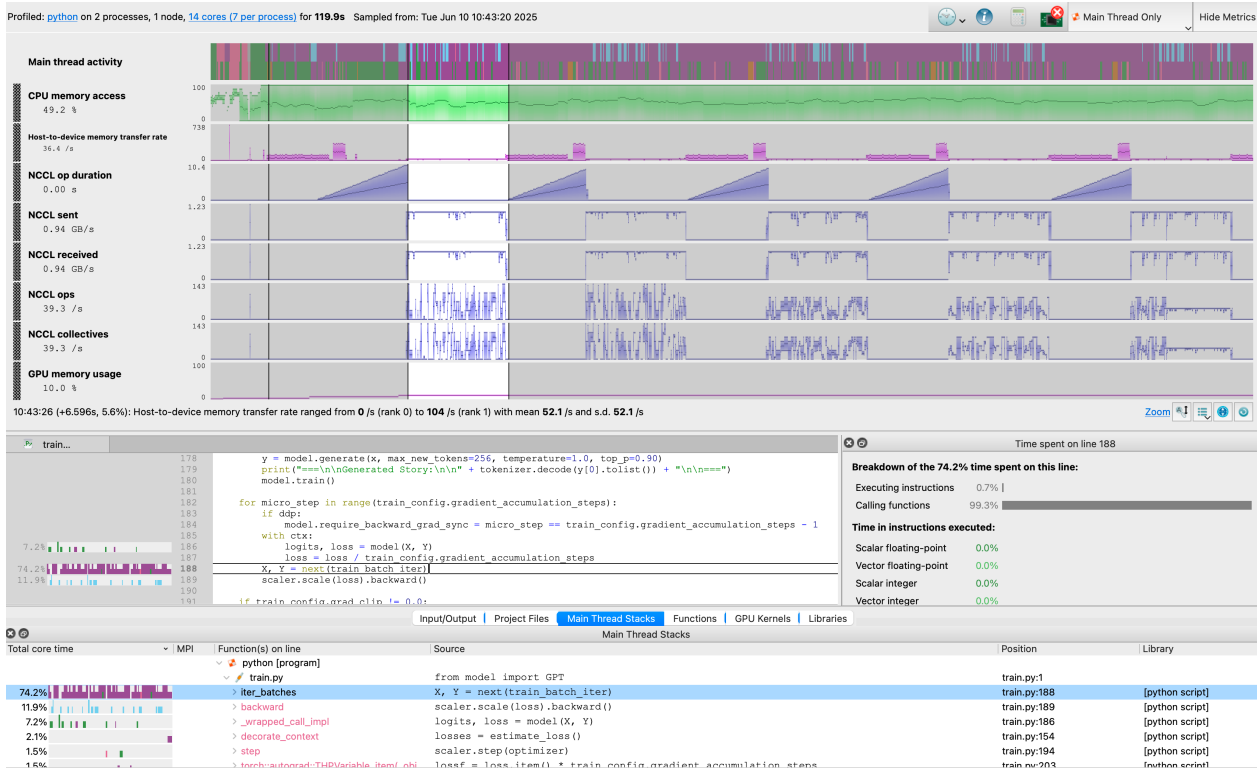
- NCCL $\geq$ 2.27.3 (earlier versions do not provide required profiling capabilities).

MAP will collect the following NCCL metrics

- The *NCCL op duration* metric tracks the time spent in a NCCL op so far, which is measured based on the time the corresponding GPU kernel is active.
- The *NCCL sent/received* pair of metrics tracks the number of bytes that NCCL ops operate on per second. NB: This is not the same as the speed with which data is transmitted over interconnects, as that information is not available.
- The *NCCL ops* metric tracks the number of NCCL point-to-point and collective ops per second, ditto for NCCL collectives metric.



Profile of an LLM running on A100s



MAP Thread Affinity Advisor

Snapshot Selector

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

Launch Command: `srun -n 16 python3 /global/homes/f/rshand/finaro-forge-training/performance/mmult.py -s 3072`

Process Command: Select an individual process

Global (launcher) environment variables: Exemplar node's topology (shading shows process affinity bindings):

OpenMP | Submission Script | Other

SLURM_CPUS_PER_TASK 16

SLURM_NPROCS 16

SLURM_NTASKS 16

SLURM_NTASKS_PER_NODE 16

cpu:16

Machine

Package

NUMANode #0

L3Cache

L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache

L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache

Core Core Core Core Core Core Core Core Core Core Core Core Core Core Core Core

NUMANode #1

L3Cache

L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache L2Cache

L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache L1Cache

Core Core Core Core Core Core Core Core Core Core Core Core Core Core Core Core

NUMANode #2

(multiple items selected)

Processes on exemplar node:

Rank 0 (PID 1166384)

Rank 1 (PID 1166385)

Rank 2 (PID 1166387)

Rank 3 (PID 1166389)

Rank 4 (PID 1166391)

Rank 5 (PID 1166393)

Threads in selected processes:

✓ pthread (LWP 1167177) 000-0

✓ pthread (LWP 1166919) 000-0

✓ Main thread (LWP 1166384) 000-0

✓ pthread (LWP 1167181) 032-0

✓ pthread (LWP 1166929) 032-0

✓ Main thread (LWP 1166391) 032-0

Commentary:

[ERROR] nid004343, ranks 0-15 (processes 1166384-1166385,1166387,1166389,1166391,1166393-1166394,1166397,1166399,1166401,1166403,1166405,1166407,1166409,1166411,1166413) contain at least one compute thread which has an overlapping thread affinity mask with another compute thread, e.g. threads 1166391 and 1166929.

[INFORMATION] nid004343, number of threads allocated to node may be less than ideal. 48 are currently allocated, but consider using 128 (1 per core) for improved utilization.

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

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

Exemplar Nodes

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

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.

Threads List

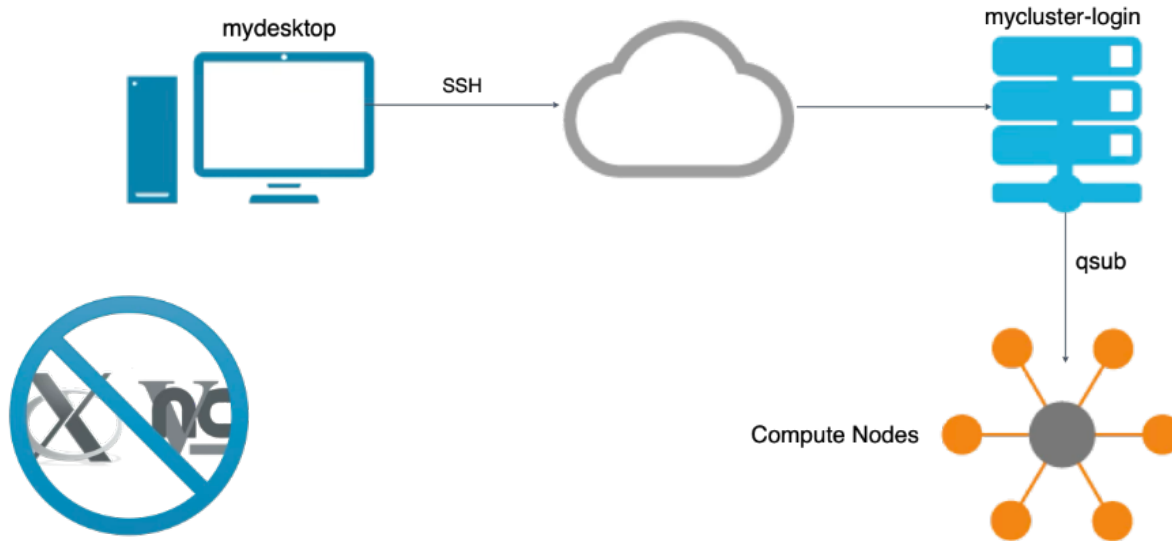
List of all threads for the selected process. Selecting threads highlights which cores they are bound to in the topology view.

Commentary

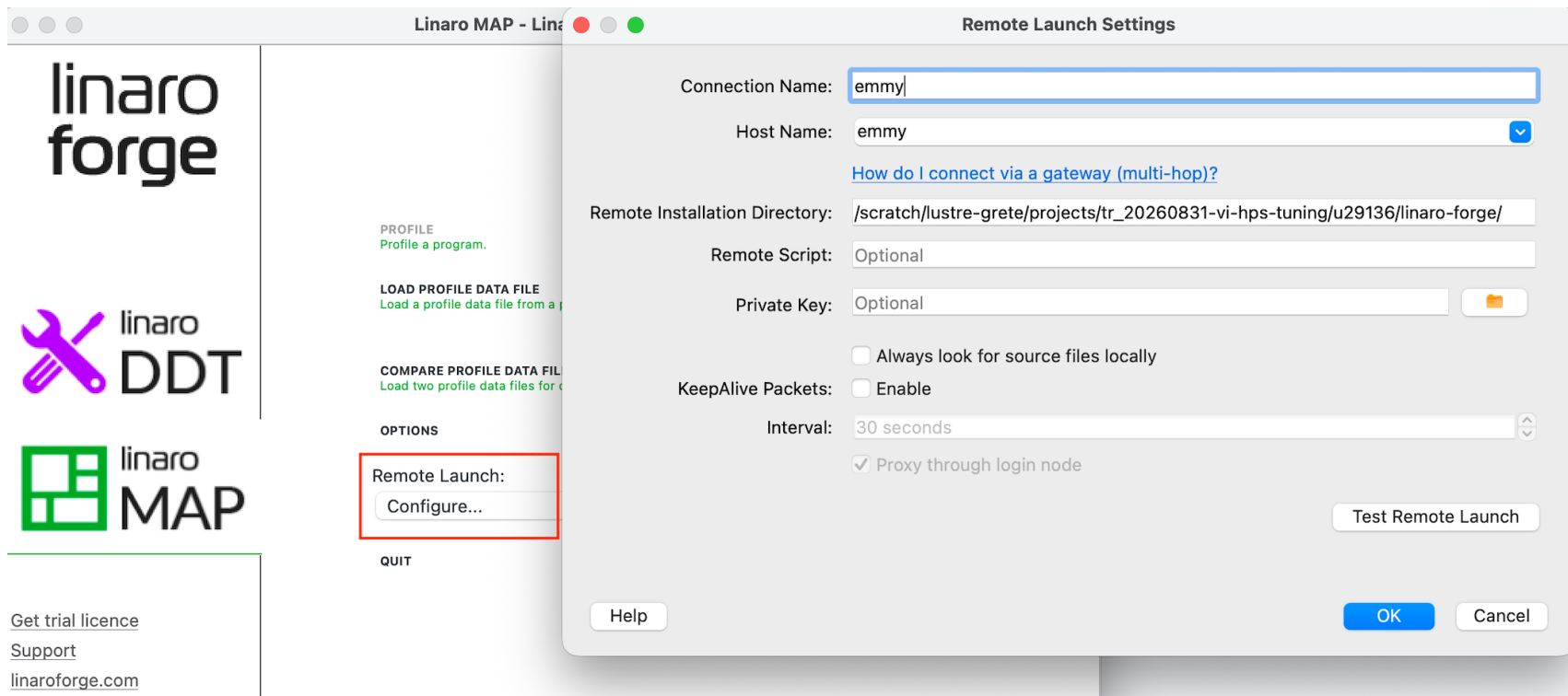
A list of commentary, providing information and advice on Memory Imbalance, Core Utilization etc.

The Forge GUI and where to run it

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



Remote connection



Hands-On

Training material

`cp /scratch/lustre-grete/projects/tr_20260831-vi-hps-tuning/u29136/
linaro-forge/examples/mmult_example.tar.gz .`

Training content

mmult examples & makefile
job.sh (Job script for emmy)

Forge Client (On local machine)

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

Running with a batch script

`sbatch job.sh`

Forge commands

`ddt --connect` *# Reverse connect*
`ddt --offline` *# Run DDT without GUI*
`map --profile` *# Profile without GUI*
`perf-report` *# Generate Performance Report*

slurm queue commands

`sbatch <jobscript>` *# Submit a job to queue*
`squeue` *# Check job queue*
`scancel <job-id>` *# Remove job from queue*

Guides

[Forge userguide](#)
[GWDG HPC Documentation](#)

Thank You

rudy.shand@linaro.org