Linaro Forge

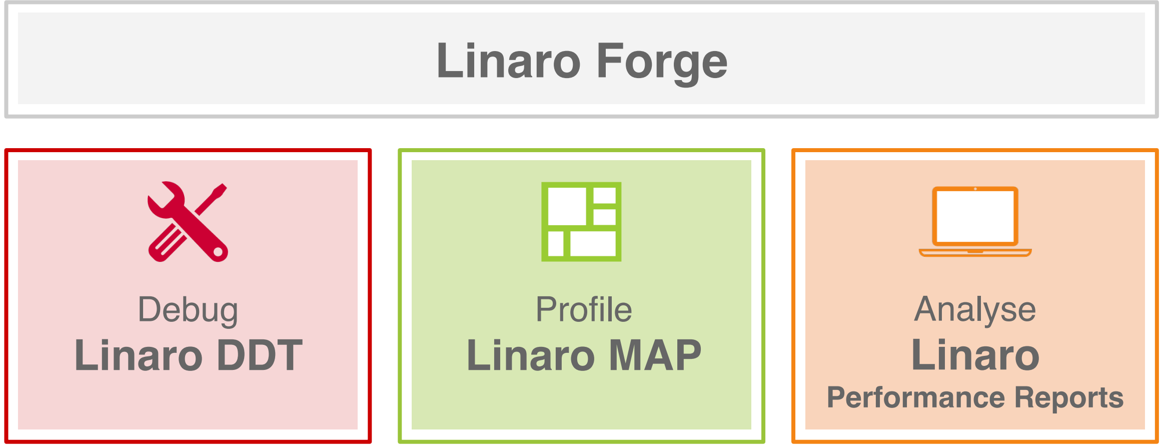
Performance Analysis with Linaro Forge

Rudy Shand - Field Application Engineer

Linaro Forge

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.

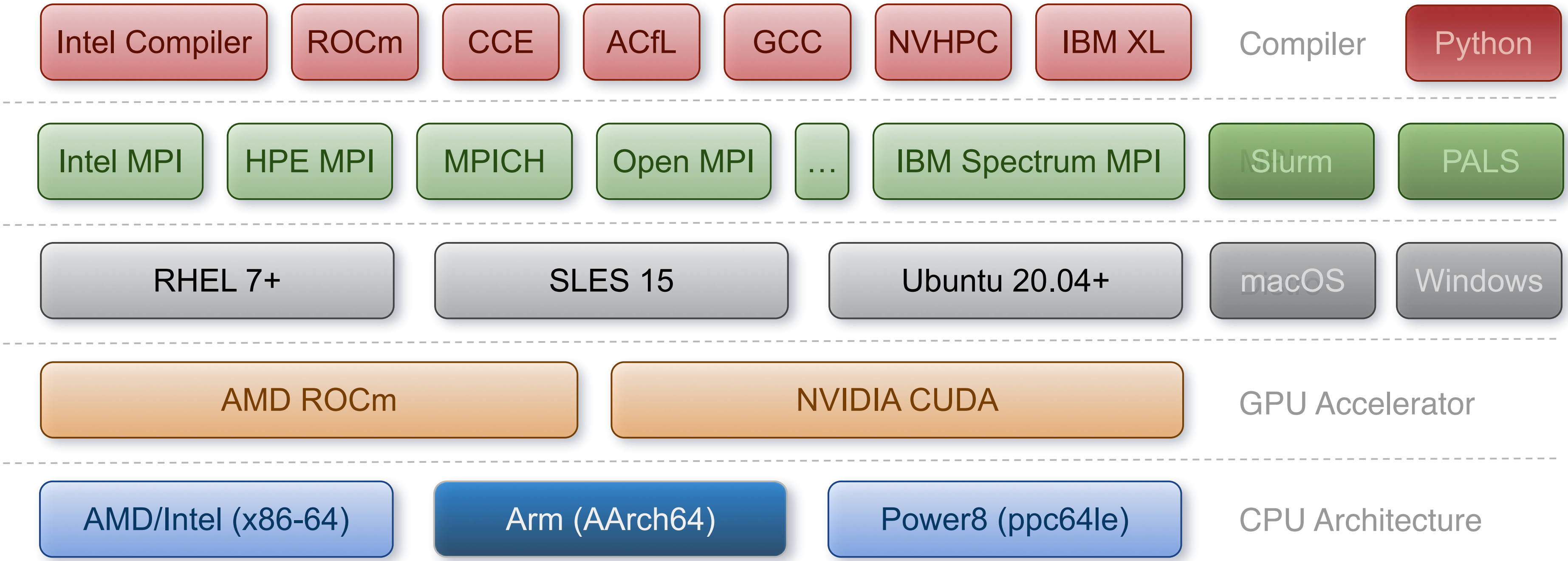
State-of-the art debugging and profiling capabilities

- 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

Supported Platforms



Intel Compiler | ROCm | CCE | ACfL | GCC | NVHPC | IBM XL | Compiler | Python

Intel MPI | HPE MPI | MPICH | Open MPI | … | IBM Spectrum MPI | Slurm | PALS

RHEL 7+ | SLES 15 | Ubuntu 20.04+ | macOS | Windows

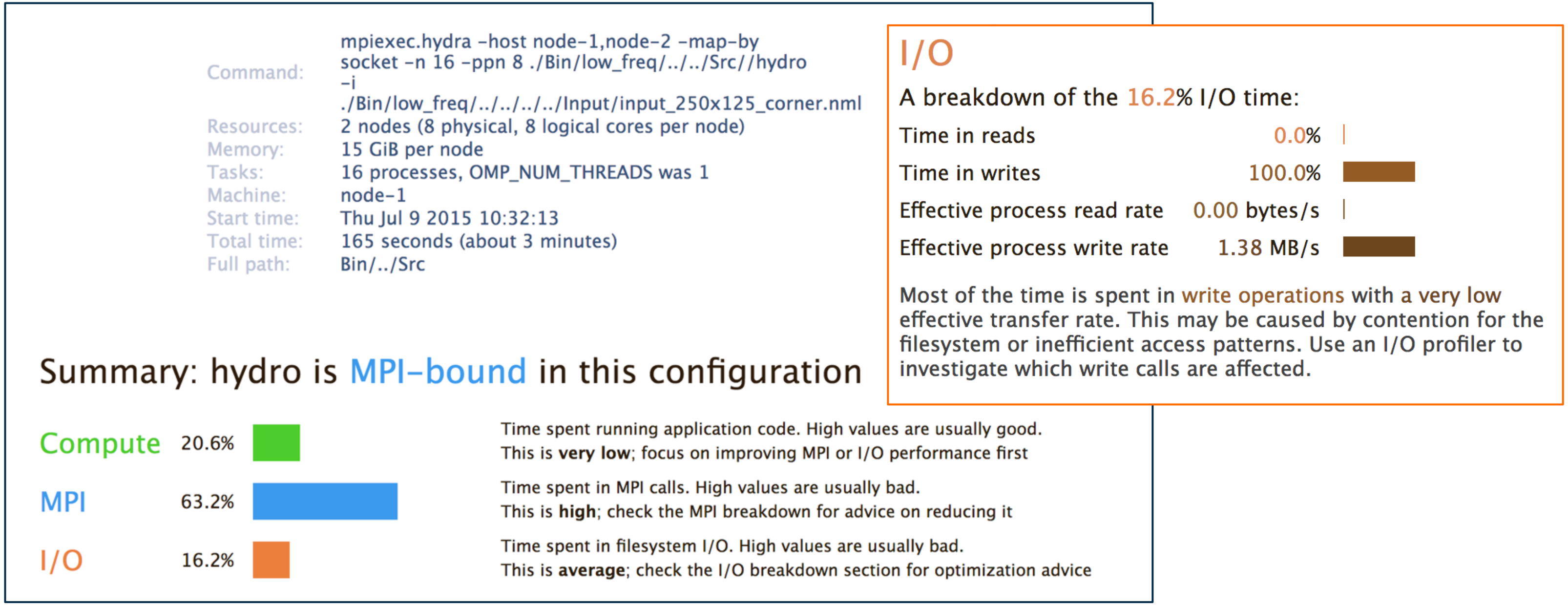
AMD ROCm | NVIDIA CUDA | GPU Accelerator

AMD/Intel (x86-64) | Arm (AArch64) | Power8 (ppc64le) | CPU Architecture

Linaro Forge

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:

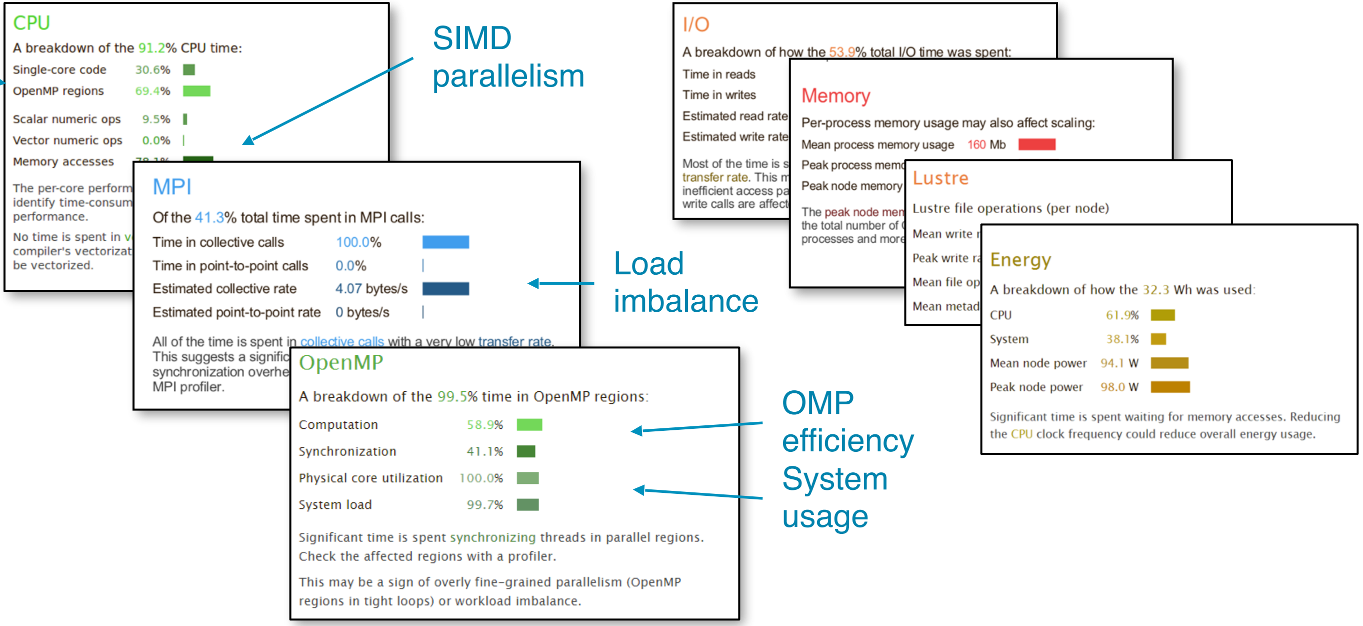
Time in reads	0.0%	|
Time in writes	100.0%	🟫
Effective process read rate	0.00 bytes/s	|
Effective process write rate	1.38 MB/s	🟫

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 Forge

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	78.1%

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 mem
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.

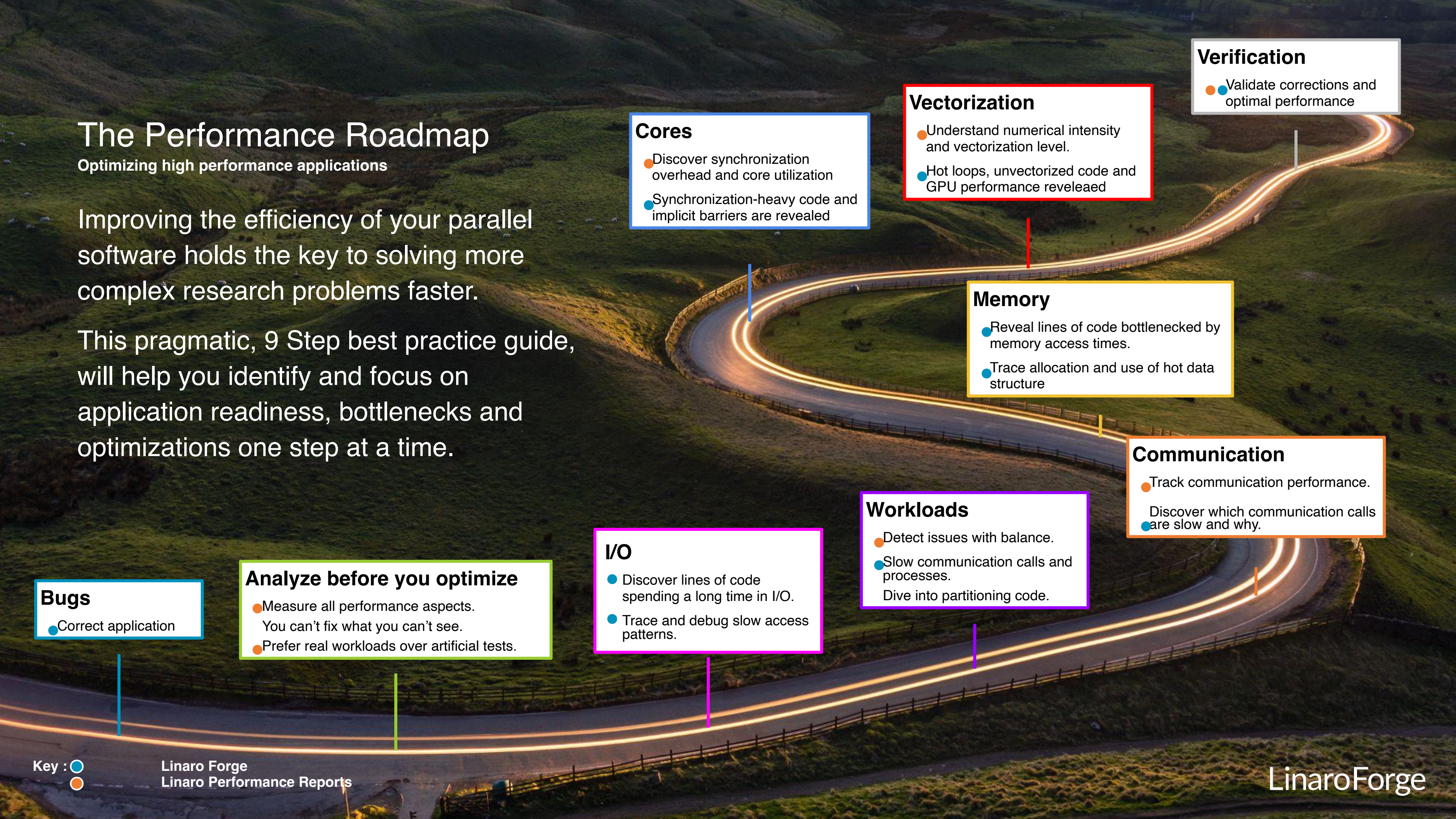
Linaro Forge

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

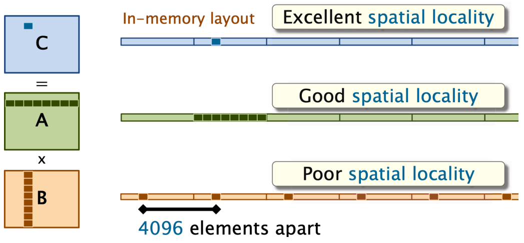
Key : 🔵 Linaro Forge

🟠 Linaro Performance Reports

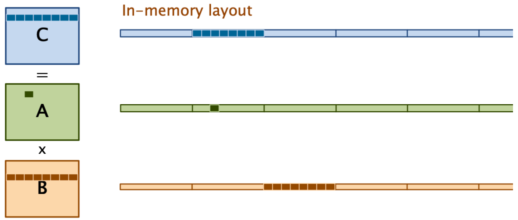
LinaroForge

Performance Improvement

i, j, k



i, k, j

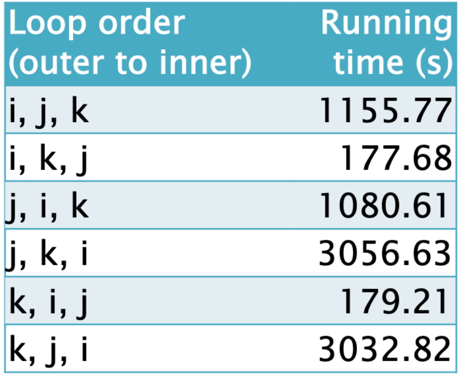


Think,

code,

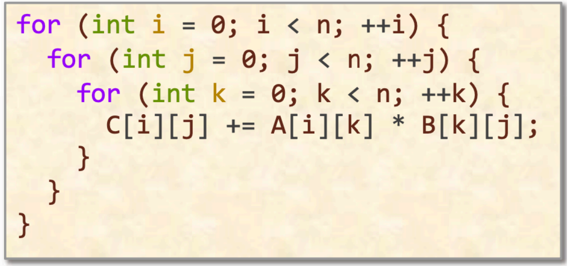
run, run, run…

…to test and measure many different implementations



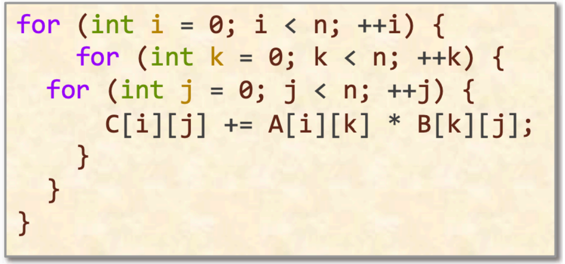
Loop order (outer to inner)	Running time (s)
i, j, k	1155.77
i, k, j	177.68
j, i, k	1080.61
j, k, i	3056.63
k, i, j	179.21
k, j, i	3032.82

i, j, k



```
for (int i = 0; i < n; ++i) {
 for (int j = 0; j < n; ++j) {
  for (int k = 0; k < n; ++k) {
   C[i][j] += A[i][k] * B[k][j];
  }
 }
}
```

i, k, j



```
for (int i = 0; i < n; ++i) {
  for (int k = 0; k < n; ++k) {
 for (int j = 0; j < n; ++j) {
   C[i][j] += A[i][k] * B[k][j];
  }
 }
}
```

MAP Capabilities

MAP is a sampling based scalable profiler

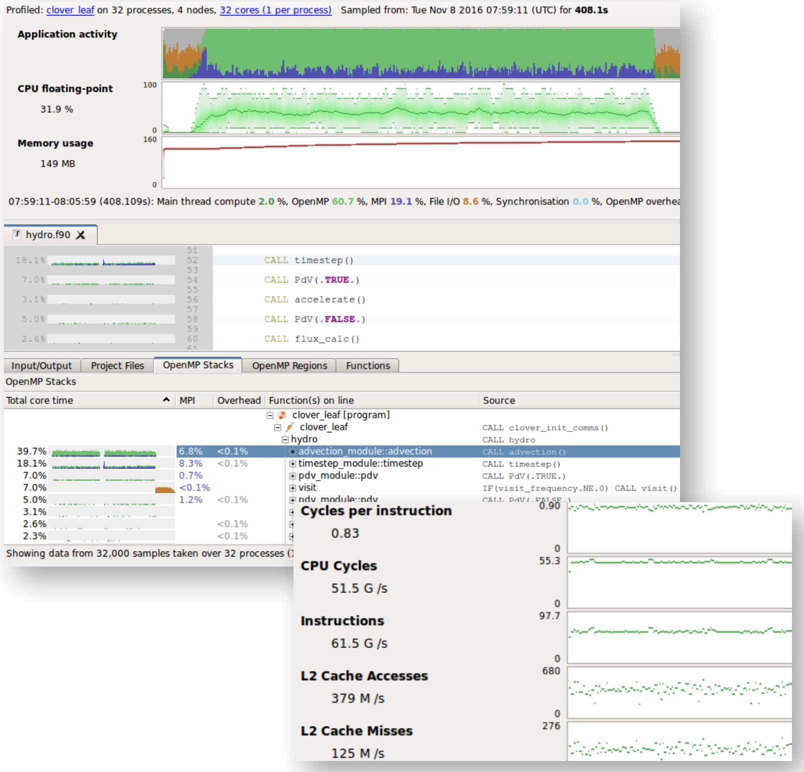
- Parallel support for MPI, OpenMP, CUDA
- Designed for C/C++/Fortran

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



Profiled: clover_leaf on 32 processes, 4 nodes, 32 cores (1 per process) Sampled from: Tue Nov 8 2016 07:59:11 (UTC) for **408.1s**

07:59:11-08:05:59 (408.109s): Main thread compute 2.0 %, OpenMP 60.7 %, MPI 19.1 %, File I/O 8.6 %, Synchronisation 0.0 %, OpenMP overhea

```
      CALL timestep()

      CALL PdV(.TRUE.)

      CALL accelerate()

      CALL PdV(.FALSE.)

      CALL flux_calc()
```

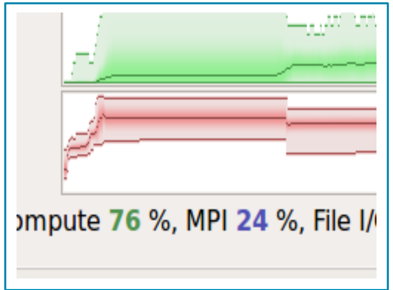
OpenMP Stacks

Total core time	MPI	Overhead	Function(s) on line	Source
			clover_leaf [program]	
			clover_leaf	CALL clover_init_comms()
			hydro	CALL hydro
39.7%	6.8%	<0.1%	advection_module::advection	CALL advection()
18.1%	8.3%	<0.1%	timestep_module::timestep	CALL timestep()
7.0%	0.7%		pdv_module::pdv	CALL PdV(.TRUE.)
7.0%	<0.1%		visit	IF(visit_frequency.NE.0) CALL visit()
5.0%	1.2%	<0.1%	pdv_module::pdv	CALL PdV(.FALSE.)
3.1%				
2.6%		<0.1%		
2.3%		<0.1%		

Showing data from 32,000 samples taken over 32 processes (1

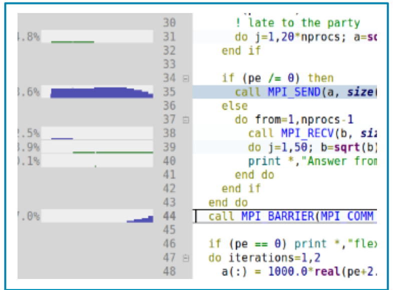
Linaro Forge

Linaro MAP Source Code Profiler Highlights



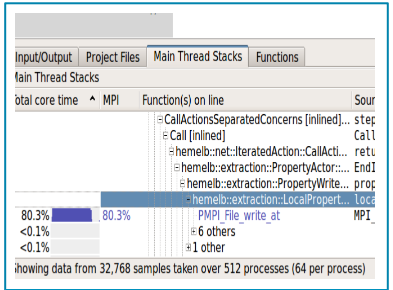
ompute 76 %, MPI 24 %, File I/

Find the peak memory use



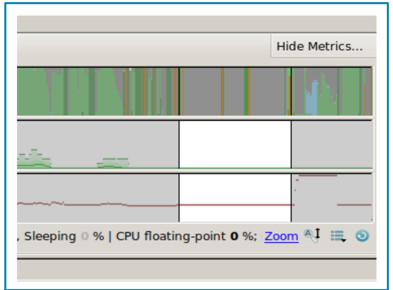
```
        ! late to the party
        do j=1,20*nprocs; a=sc
      end if

      if (pe /= 0) then
        call MPI_SEND(a, size
      else
        do from=1,nprocs-1
          call MPI_RECV(b, si
          do j=1,50; b=sqrt(b)
          print *,"Answer from
        end do
      end if
    end do
    call MPI_BARRIER(MPI_COMM

    if (pe == 0) print *,"flex
    do iterations=1,2
      a(:) = 1000.0*real(pe+2.
```

Fix an MPI imbalance



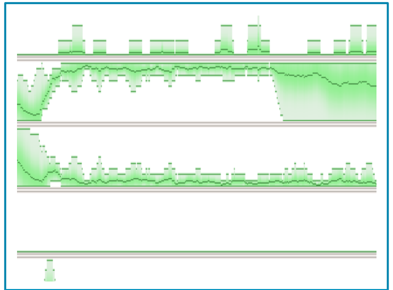
```
Main Thread Stacks
Total core time ^  MPI     Function(s) on line                          Sour
                           ⊟CallActionsSeparatedConcerns [inlined]...   step
                            ⊟Call [inlined]                             Call
                             ⊟hemelb::net::IteratedAction::CallActi...  retu
                              ⊟hemelb::extraction::PropertyActor::...   EndI
                               ⊟hemelb::extraction::PropertyWrite...    prop
                                ⊟hemelb::extraction::LocalPropert...    loca
80.3%              80.3%          PMPI_File_write_at                    MPI_
<0.1%                            ⊞6 others
<0.1%                           ⊞1 other
Showing data from 32,768 samples taken over 512 processes (64 per process)
```

Remove I?O bottleneck

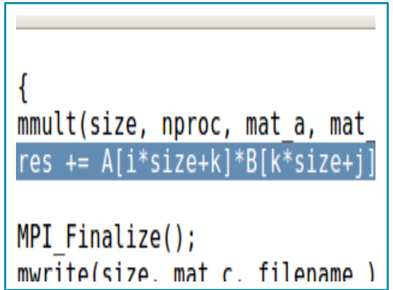


, Sleeping 0 % | CPU floating-point 0 %; Zoom

Make sure OpenMP regions make sense



Improve memory access

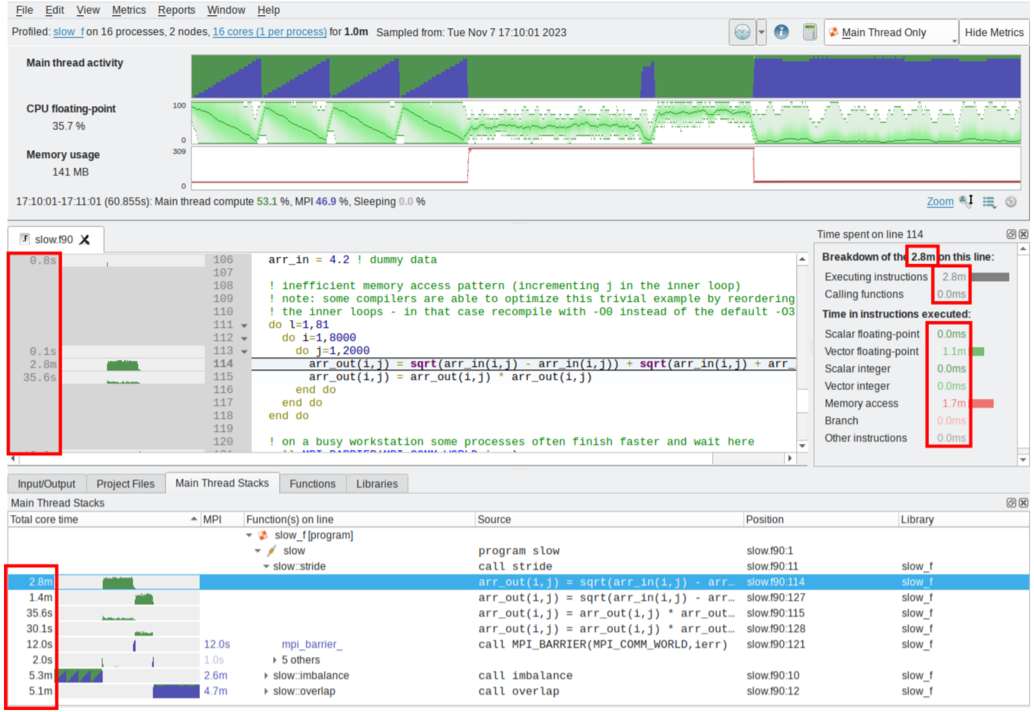


```
{
mmult(size, nproc, mat_a, mat
res += A[i*size+k]*B[k*size+j]

MPI_Finalize();
mwrite(size, mat_c, filename )
```

Restructure for vectorization

Linaro Forge

Toggle percentage-time and core-time in MAP



Profiled: slow_f on 16 processes, 2 nodes, 16 cores (1 per process) for **1.0m** Sampled from: Tue Nov 7 17:10:01 2023

CPU floating-point 35.7 %

Memory usage 141 MB

17:10:01-17:11:01 (60.855s): Main thread compute 53.1 %, MPI 46.9 %, Sleeping 0.0 %

```
0.8s   106  arr_in = 4.2 ! dummy data
       107
       108  ! inefficient memory access pattern (incrementing j in the inner loop)
       109  ! note: some compilers are able to optimize this trivial example by reordering
       110  ! the inner loops - in that case recompile with -O0 instead of the default -O3
       111  do l=1,81
       112    do i=1,8000
0.1s   113      do j=1,2000
2.8m   114        arr_out(i,j) = sqrt(arr_in(i,j) - arr_in(i,j)) + sqrt(arr_in(i,j) + arr_
35.6s  115        arr_out(i,j) = arr_out(i,j) * arr_out(i,j)
       116      end do
       117    end do
       118  end do
       119
       120  ! on a busy workstation some processes often finish faster and wait here
```

Time spent on line 114

Breakdown of the 2.8m on this line:

Executing instructions	2.8m
Calling functions	0.0ms

Time in instructions executed:

Scalar floating-point	0.0ms
Vector floating-point	1.1m
Scalar integer	0.0ms
Vector integer	0.0ms
Memory access	1.7m
Branch	0.0ms
Other instructions	0.0ms

Main Thread Stacks

Total core time	MPI	Function(s) on line	Source	Position	Library
		slow_f [program]			
		slow	program slow	slow.f90:1	
		slow::stride	call stride	slow.f90:11	slow_f
2.8m			arr_out(i,j) = sqrt(arr_in(i,j) - arr…	slow.f90:114	slow_f
1.4m			arr_out(i,j) = sqrt(arr_in(i,j) - arr…	slow.f90:127	slow_f
35.6s			arr_out(i,j) = arr_out(i,j) * arr_out…	slow.f90:115	slow_f
30.1s			arr_out(i,j) = arr_out(i,j) * arr_out…	slow.f90:128	slow_f
12.0s	12.0s	mpi_barrier_	call MPI_BARRIER(MPI_COMM_WORLD,ierr)	slow.f90:121	slow_f
2.0s	1.0s	5 others			
5.3m	2.6m	slow::imbalance	call imbalance	slow.f90:10	slow_f
5.1m	4.7m	slow::overlap	call overlap	slow.f90:12	slow_f

Use for direct comparisons between runs at the same scale (process/core counts).

- Easily determine if a change has made a portion of code faster, slower, or largely unchanged.
- Performance report automatically includes both percentage-time and core time
- Core-time is an estimation, but should be very close to the application run time

Hands on Setup

Remote System

```
Host dine
    Hostname login8.cosma.dur.ac.uk
    user <username>
```

Examples are in */cosma/home/do009/linaro/performance*

```
module load allinea/ddt/23.1.0
```

Local Machine

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

[Forge userguide]()

Matrix Multiplication example

Build and run matrix multiplication example

https://docs.linaroforge.com/23.1.1/html/forge/worked_examples_appendix/mmult/analyze.html

Using the Intel modules available on dine

```
# Build C Examples
 make -f mmult.makefile DEBUG=1

# Run with MAP or Performance reports
  map --profile --mpi=generic -n 16 <performance folder>/mmult_c 3072
  perf-report --mpi=generic -n 16 <performance folder>/mmult_c 3072

# Offline profile
 /cosma/home/do009/linaro/submit-job.sh
```

Thank you

~

rudy.shand@linaro.org

Linaro Forge