Arm cross-platform tools

VI-HPS platform

October 16, 2018

An introduction to Arm

Arm is the world's leading semiconductor intellectual property supplier

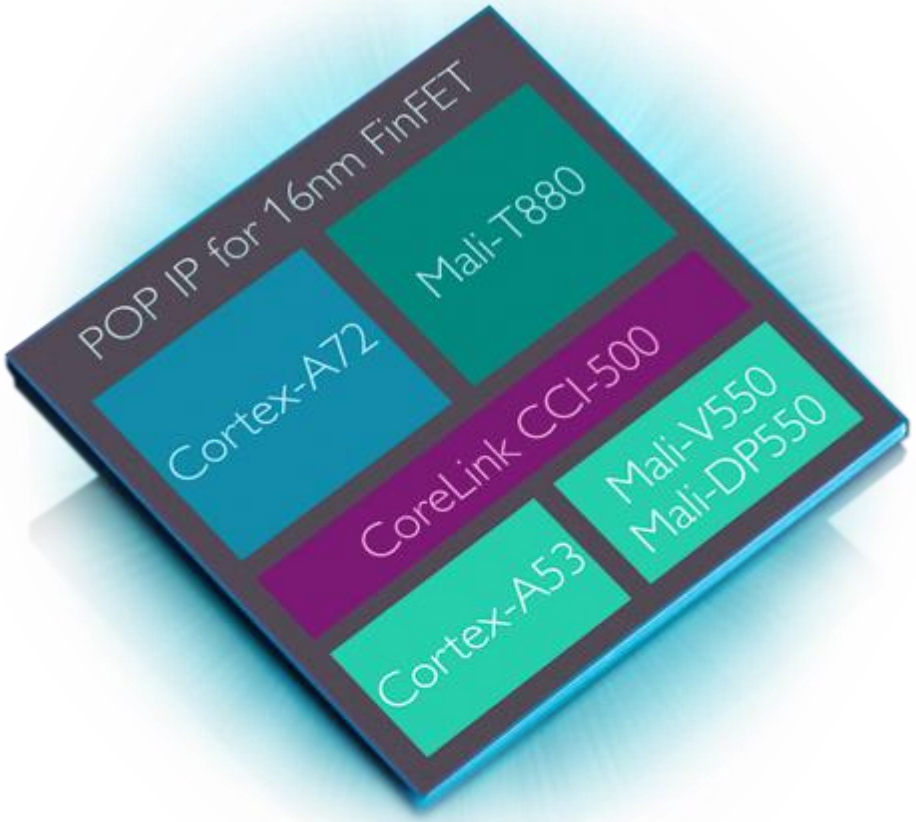
We license to over 350 partners: present in 95% of smart phones, 80% of digital cameras, 35% of all electronic devices, and a total of 60 billion Arm cores have been shipped since 1990

Our CPU business model:

License technology to partners, who use it to create their own system-on-chip (SoC) products

- We may license an instruction set architecture (ISA) such as "Armv8-A"
- or a specific implementation, such as "Cortex-A72"

Partners who license an ISA can create their own implementation, as long as it passes the compliance tests



...and our IP extends beyond the CPU

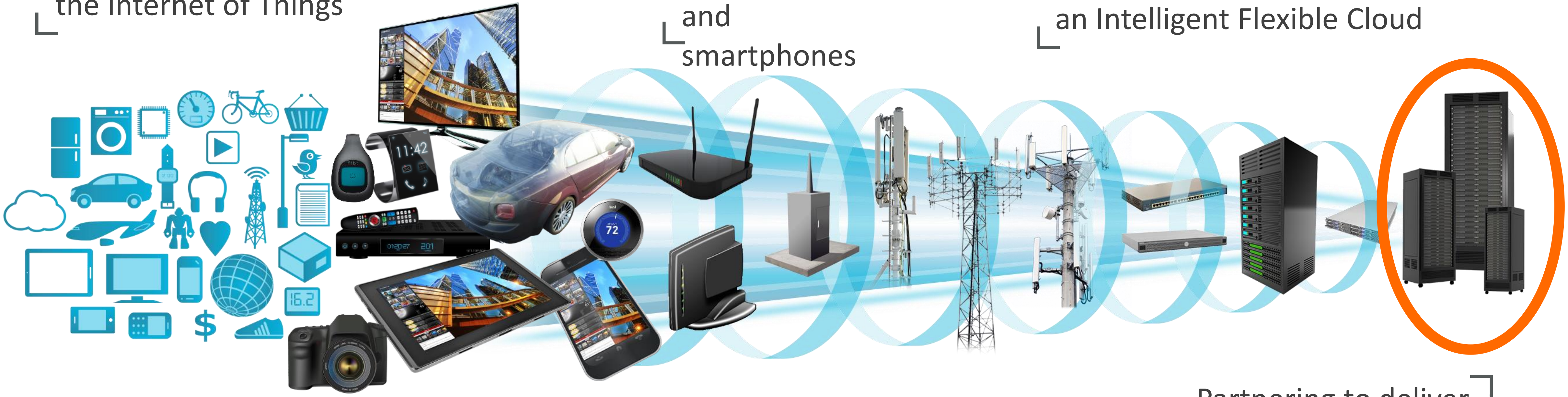
ARM’s mission

- Deploy energy-efficient ARM-based technology, wherever computing happens…

Leading in wearables and the Internet of Things

~85% share of laptops, tablets, and smartphones

Driving the transformation of the network and data center to an Intelligent Flexible Cloud



Enabling innovation and creativity with embedded intelligence

Taking mobile computing to the next four billion people

Partnering to deliver data center efficiency

arm

HPC strategy

Mission:
Enable the world's first Arm supercomputers

Strategy:
Enablement + Co-Design + Partnership

Building Blocks

Enablement

- Address gaps in computational capability and data movement within Architecture
- Seed the software ecosystem with open source support for Armv8 and SVE libraries, tools, and optimized workloads
- Provide world class tools for compilation, analysis, and debug at large scale

Co-Design

- Work with key end-customers in DoE, DoD, RIKEN, and EU to design balanced architecture, uArchitecture and SoCs based on real-world workloads, not benchmarks
- Develop simulation and modelling tools to support co-design development with end-customers, partners, and academia

Partnership

- Work with Architecture partners to bring optimized solutions to market quickly
- Work with ATG & uArchitecture design teams to steer future designs to be more relevant for HPC, HPDA, and ML
- Work with key ISVs to enable mid-market

Arm Allinea Studio

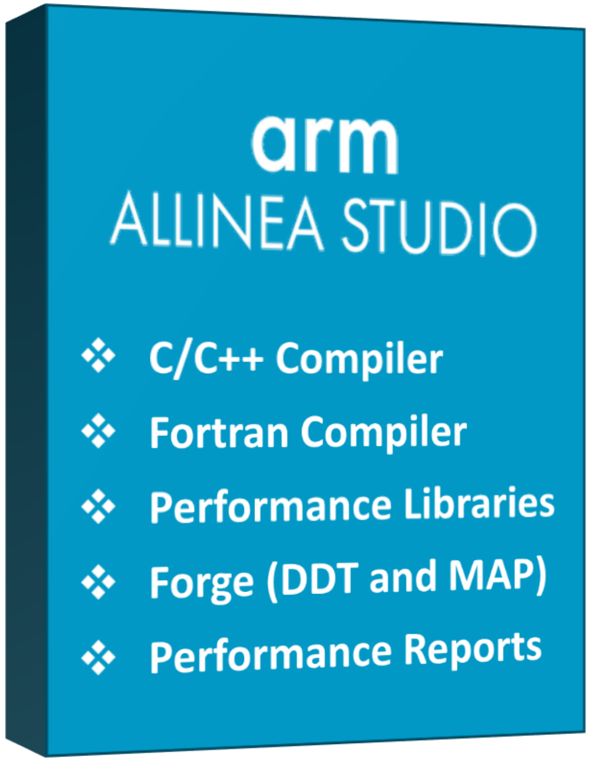
arm ALLINEA STUDIO | New commercial bundle

Meets the requirements of HPC developers on Arm

Cross-platform debug and profile tools

Arm-only Compiler and Libraries

Forge and Performance Reports with support for Arm



Arm Compilers interoperable with Forge and Performance Reports

Arm Forge

An interoperable toolkit for debugging and profiling

Commercially supported by Arm

Fully Scalable

Very user-friendly

The de-facto standard for HPC development

- Available on the vast majority of the Top500 machines in the world
- Fully supported by Arm on x86, IBM Power, Nvidia 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 petaflopic applications)

Easy to use by everyone

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

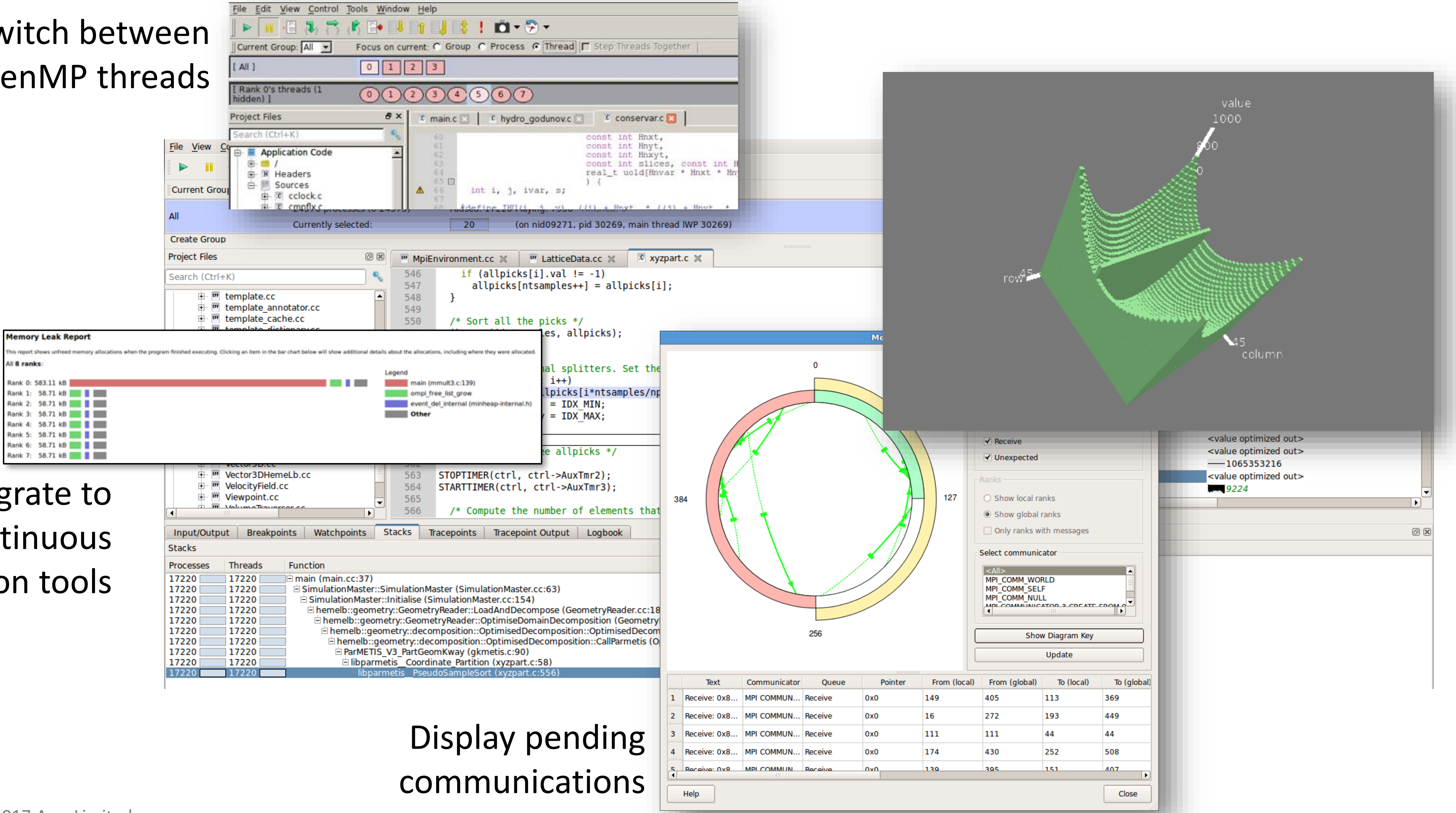
DDT

Switch between OpenMP threads

Visualise data structures

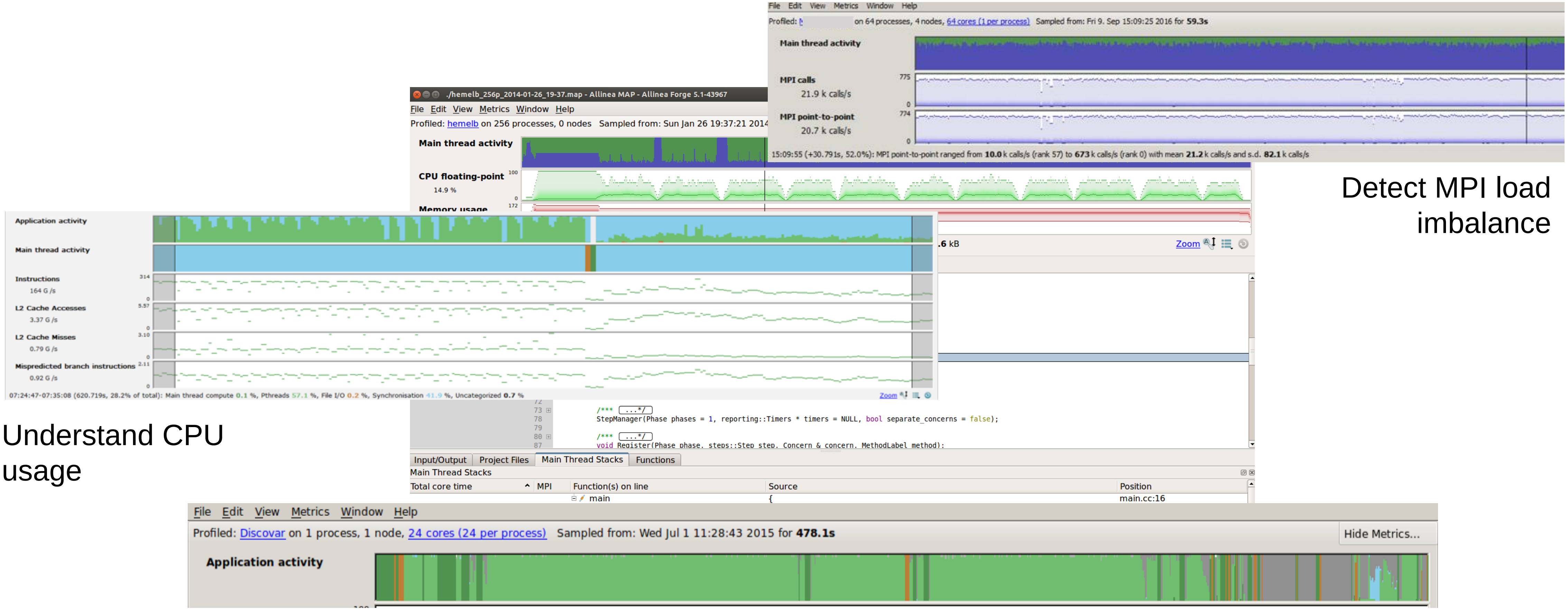
Integrate to continuous integration tools

Display pending communications



arm

Arm MAP

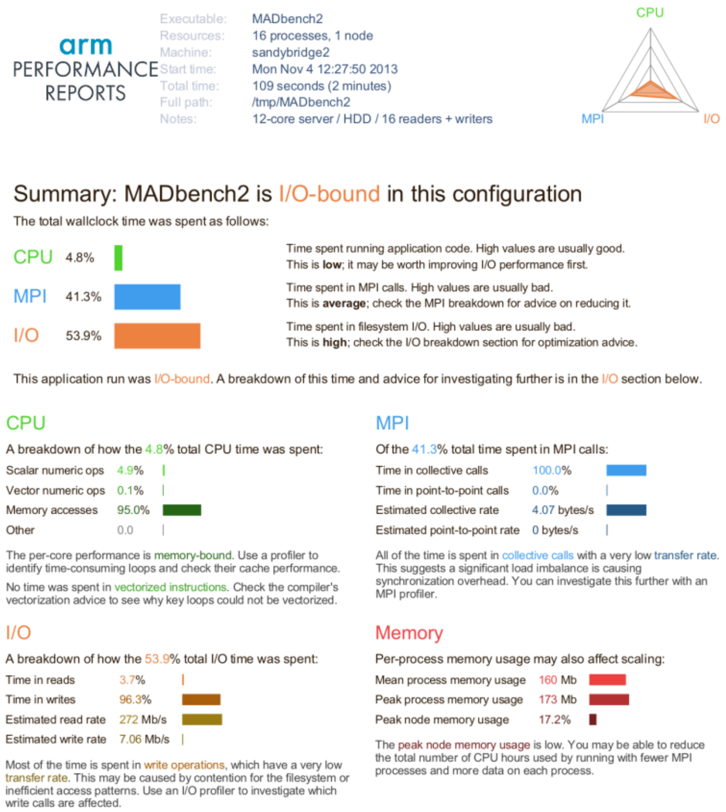


Detect MPI load imbalance

Understand CPU usage

Identify regions of high OpenMP synchronisation

Arm Performance Reports



arm PERFORMANCE REPORTS

Executable:	MADbench2
Resources:	16 processes, 1 node
Machine:	sandybridge2
Start time:	Mon Nov 4 12:27:50 2013
Total time:	109 seconds (2 minutes)
Full path:	/tmp/MADbench2
Notes:	12-core server / HDD / 16 readers + writers

CPU
MPI
I/O

Summary: MADbench2 is I/O-bound in this configuration

The total wallclock time was spent as follows:

CPU	4.8%	Time spent running application code. High values are usually good. This is **low**; it may be worth improving I/O performance first.
MPI	41.3%	Time spent in MPI calls. High values are usually bad. This is **average**; check the MPI breakdown for advice on reducing it.
I/O	53.9%	Time spent in filesystem I/O. High values are usually bad. This is **high**; check the I/O breakdown section for optimization advice.

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

CPU

A breakdown of how the 4.8% total CPU time was spent:

Scalar numeric ops	4.9%
Vector numeric ops	0.1%
Memory accesses	95.0%
Other	0.0

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

No time was spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not 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 significant load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

I/O

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

Time in reads	3.7%
Time in writes	96.3%
Estimated read rate	272 Mb/s
Estimated write rate	7.06 Mb/s

Most of the time is spent in write operations, which have a very low 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.

Memory

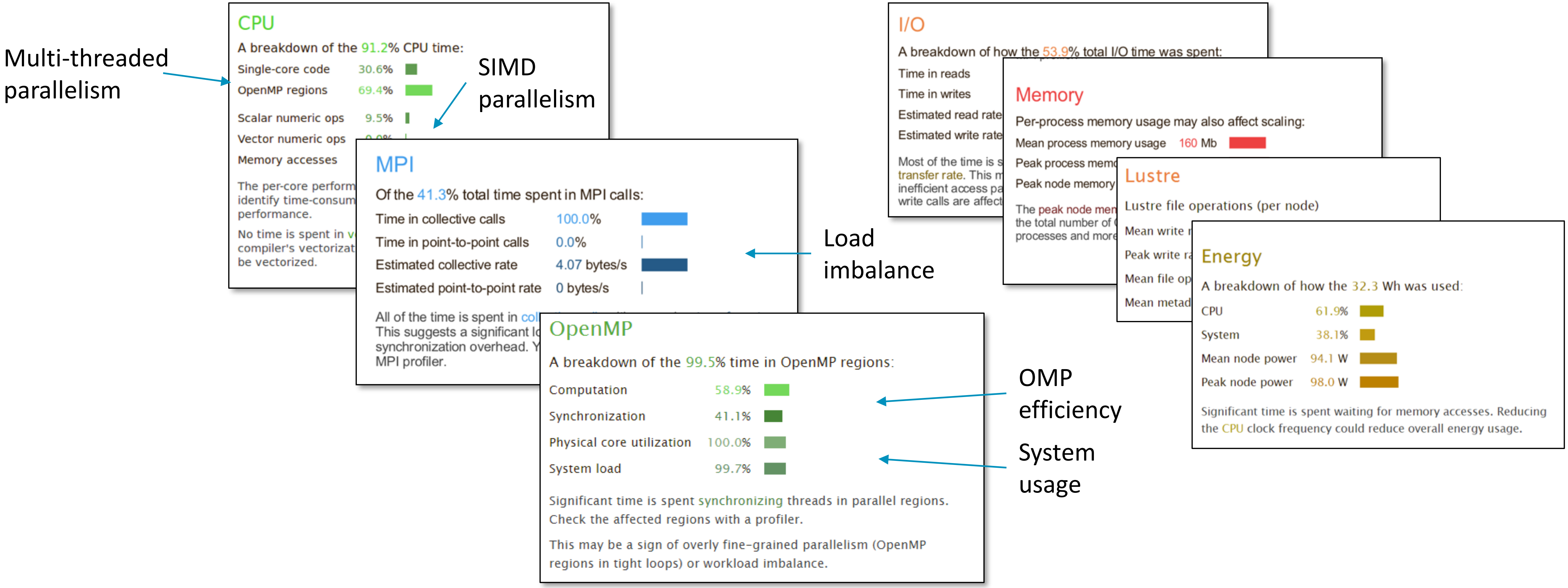
Per-process memory usage may also affect scaling:

Mean process memory usage	160 Mb
Peak process memory usage	173 Mb
Peak node memory usage	17.2%

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.

- Very simple start-up
- Fully scalable, very low overhead
- Rich set of metrics
- Powerful data analysis

Arm Performance Reports



arm

Imbalance

Challenges

- Workload imbalance results in productivity loss

End-users

- Longer application runtime
- Wasted CPU hours
- Less “science” done per project

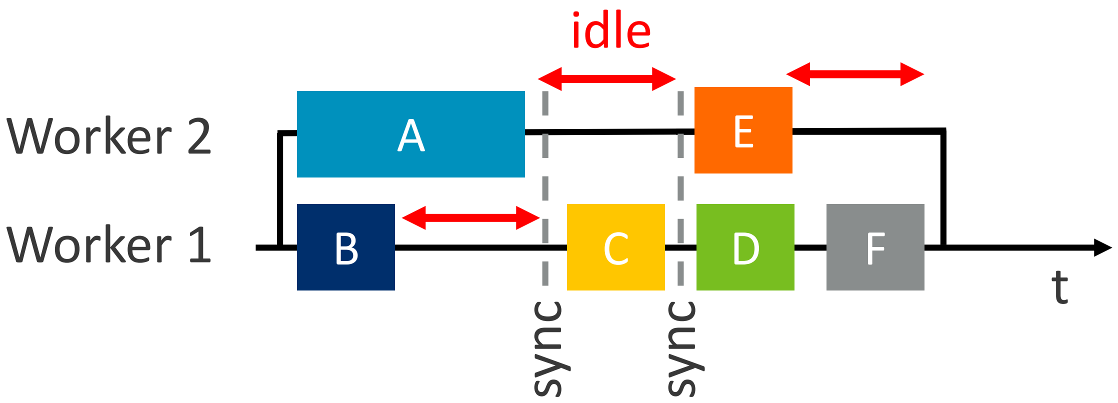
System Administrators

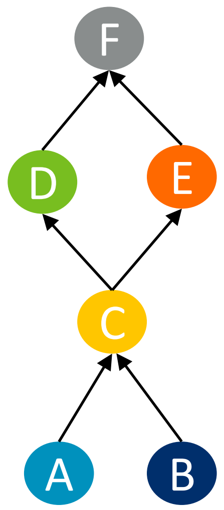
- Pressure to identify & improve inefficient workloads
- Translates the high system usage into a need for more hardware

System Owners

- Smaller return on investment
- Increased hardware costs to absorb the demand
- Bigger energy bill

In practice





MPI and OpenMP imbalance

- Clues: excessive synchronization
  - MPI collective calls with no actual data transfer
  - Idle cores where threads are stuck in locks/mutexes

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 load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

OpenMP

A breakdown of the 74.5% time in OpenMP regions:

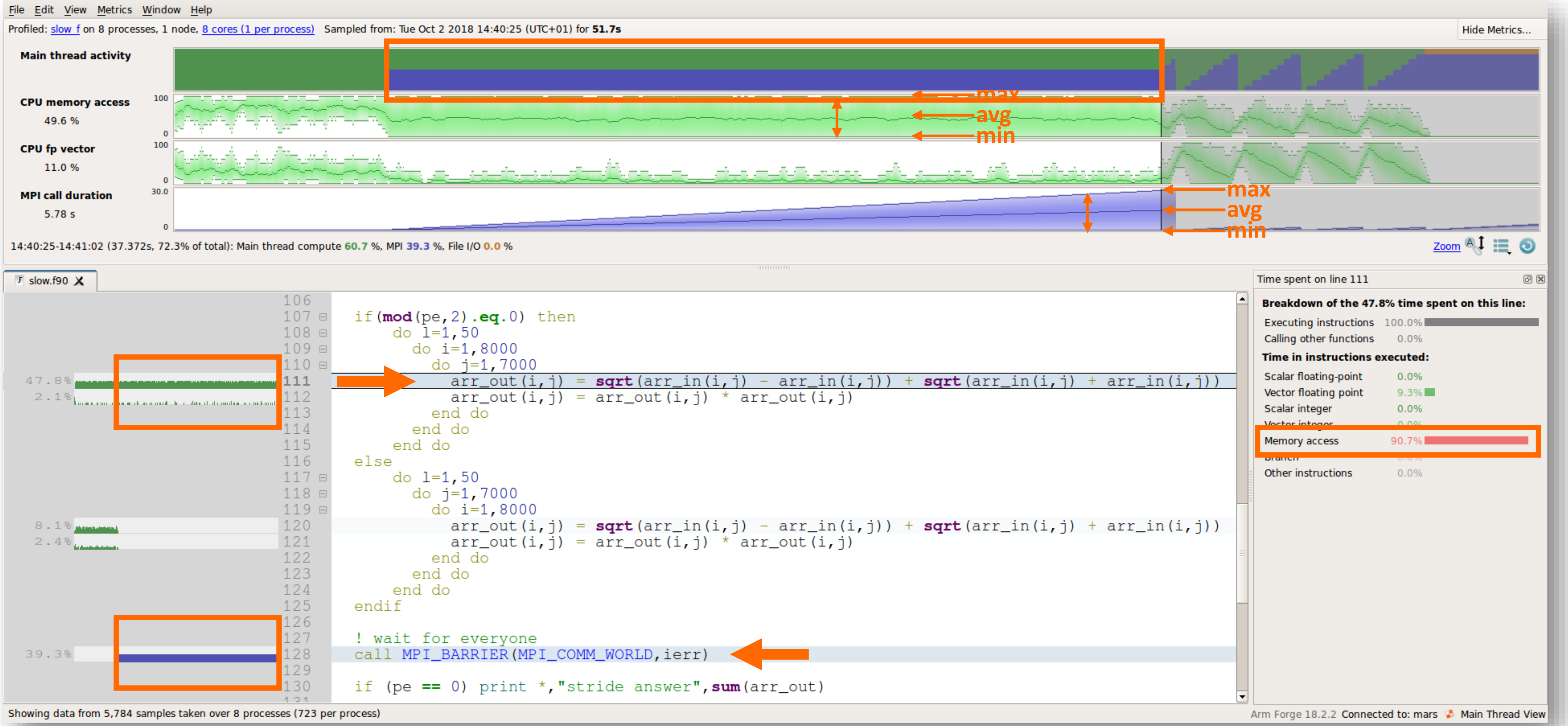
Computation	53.6%
Synchronization	46.4%
Physical core utilization	100.0%
System load	78.0%

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.

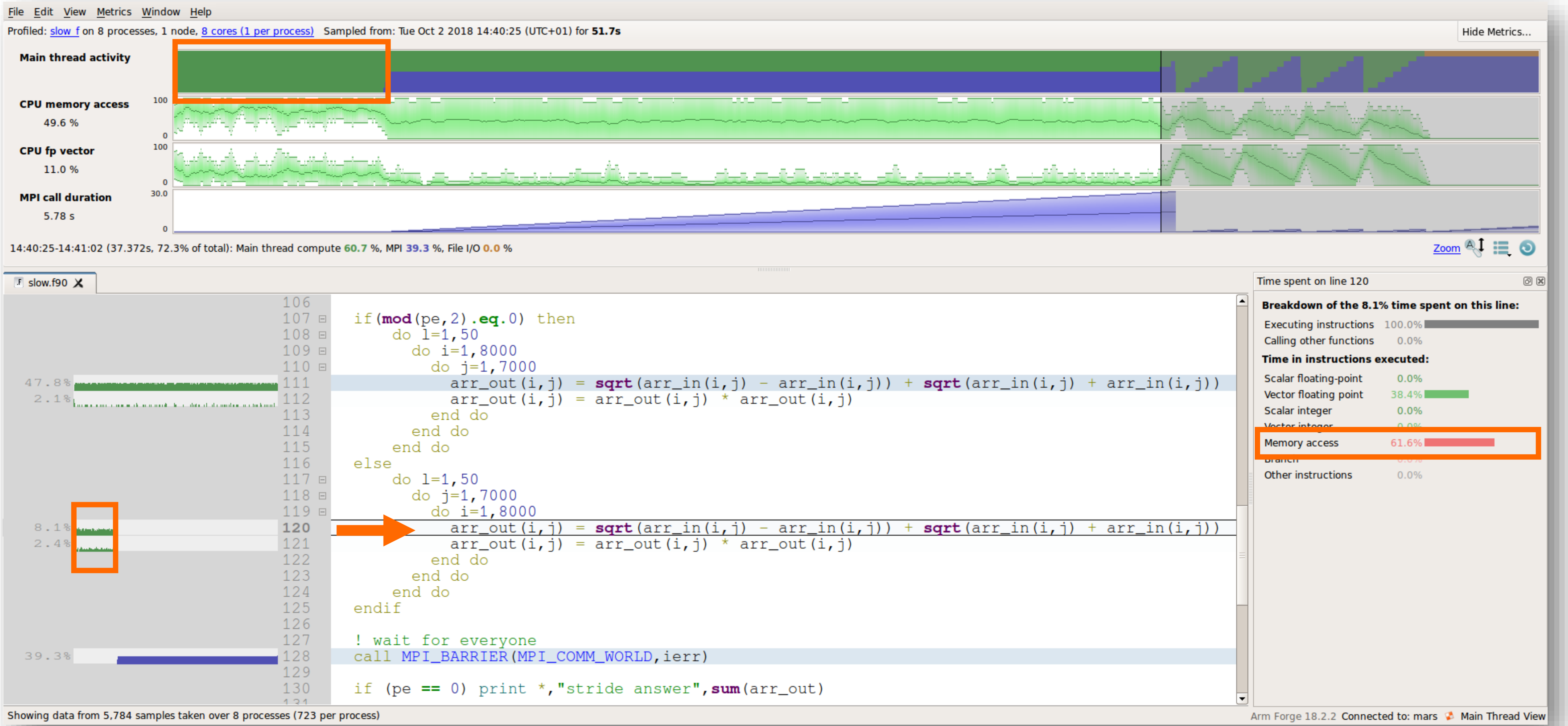
arm

MPI imbalance: barrier

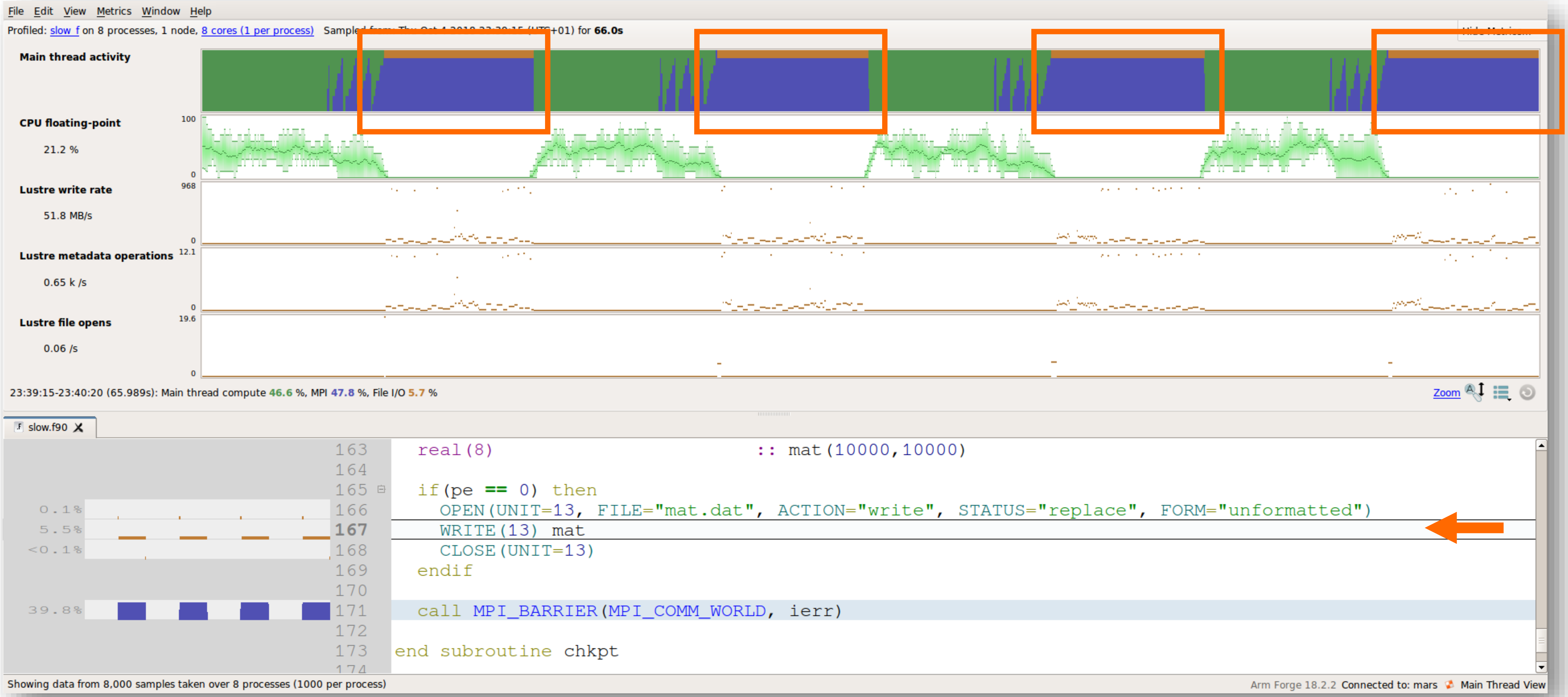


arm

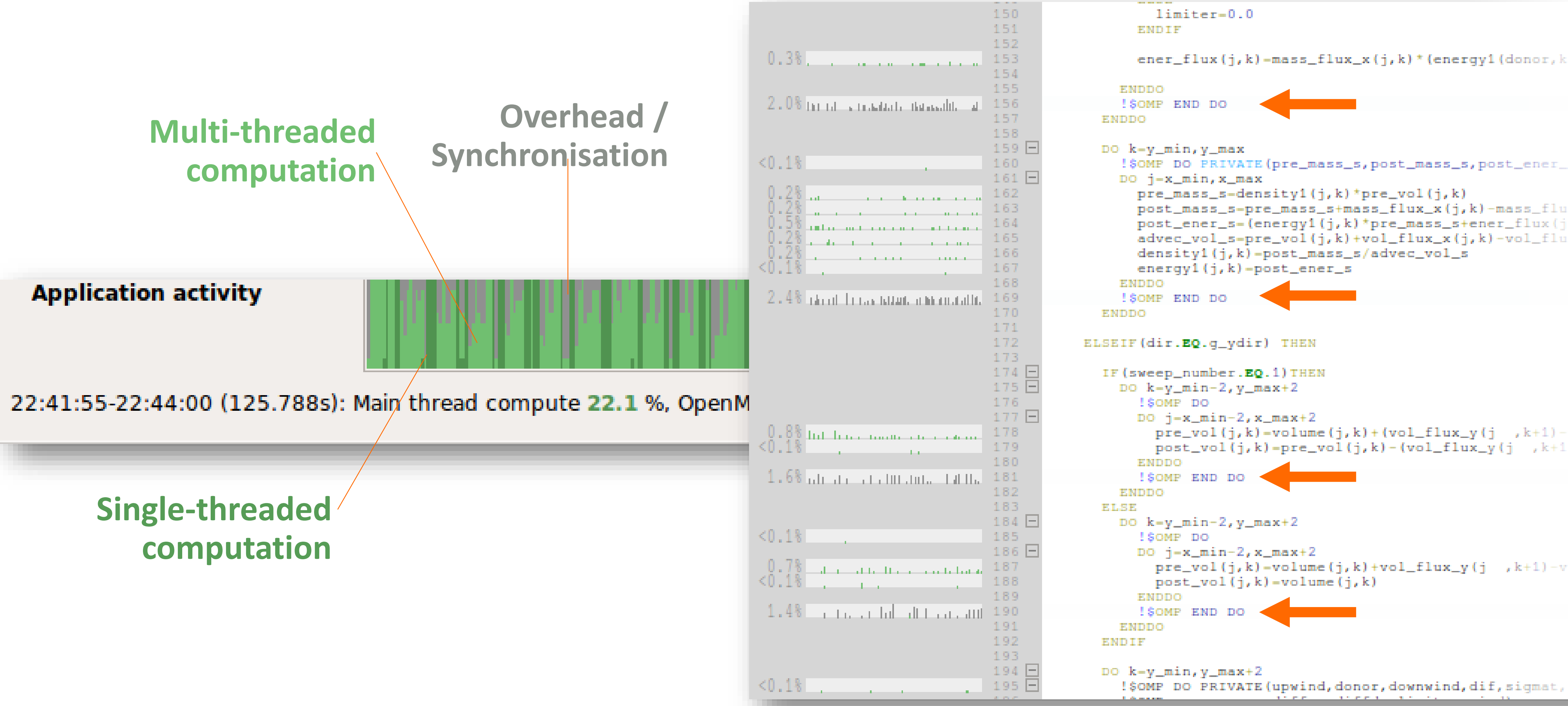
MPI imbalance: barrier



IO imbalance



OpenMP imbalance: implicit synchronization



arm

Understanding resource usage

- Memory accesses

arm

Vectorization

Analyze the results

Running Performance Reports with CloverLeaf using 8 MPI tasks indicates that:

- Time spent in scalar ops is 14.7%
- Time spent in vector ops 18.9%

Summary: clover_leaf is Compute-bound in this configuration

Compute	93.4%	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	6.6%	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	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

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 93.4% CPU time:

Scalar numeric ops	14.7%
Vector numeric ops	18.9%
Memory accesses	66.3%

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.

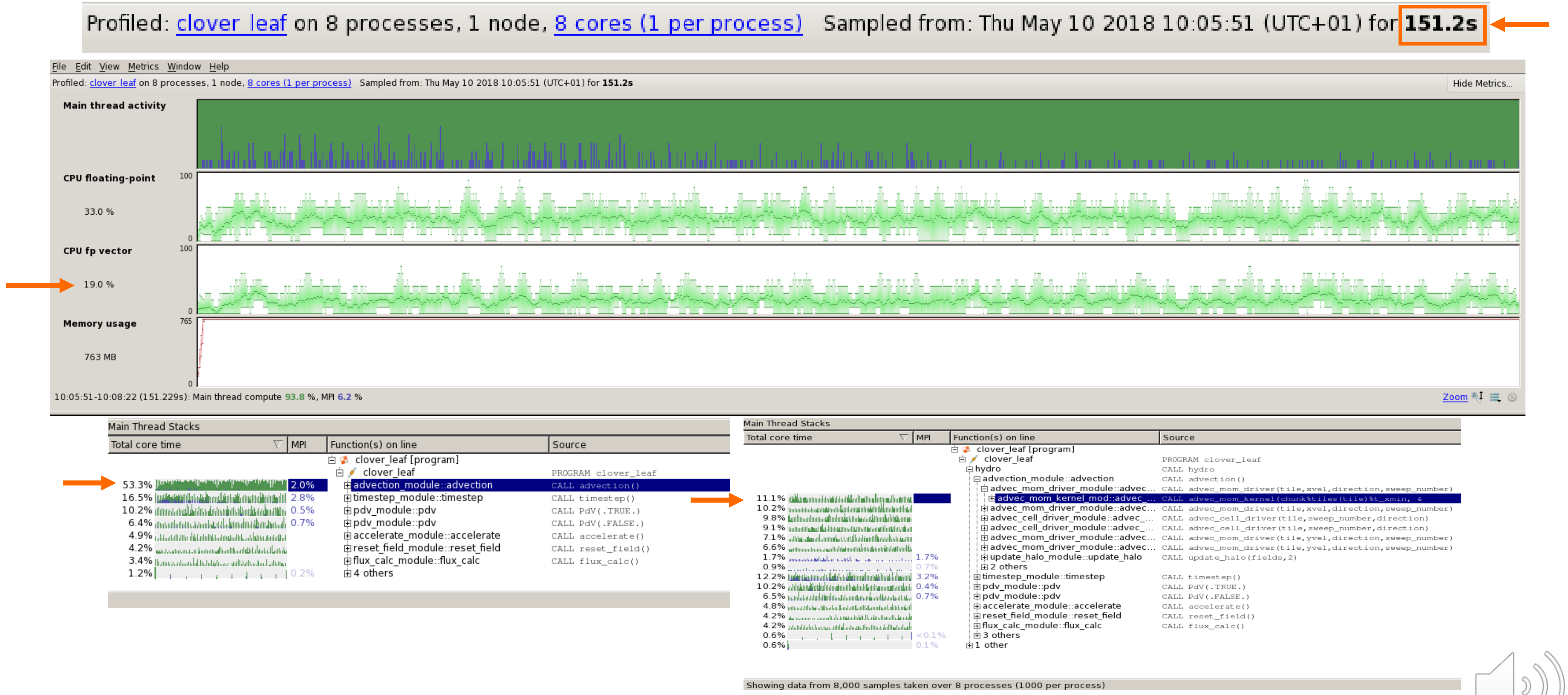
MPI

A breakdown of the 6.6% MPI time:

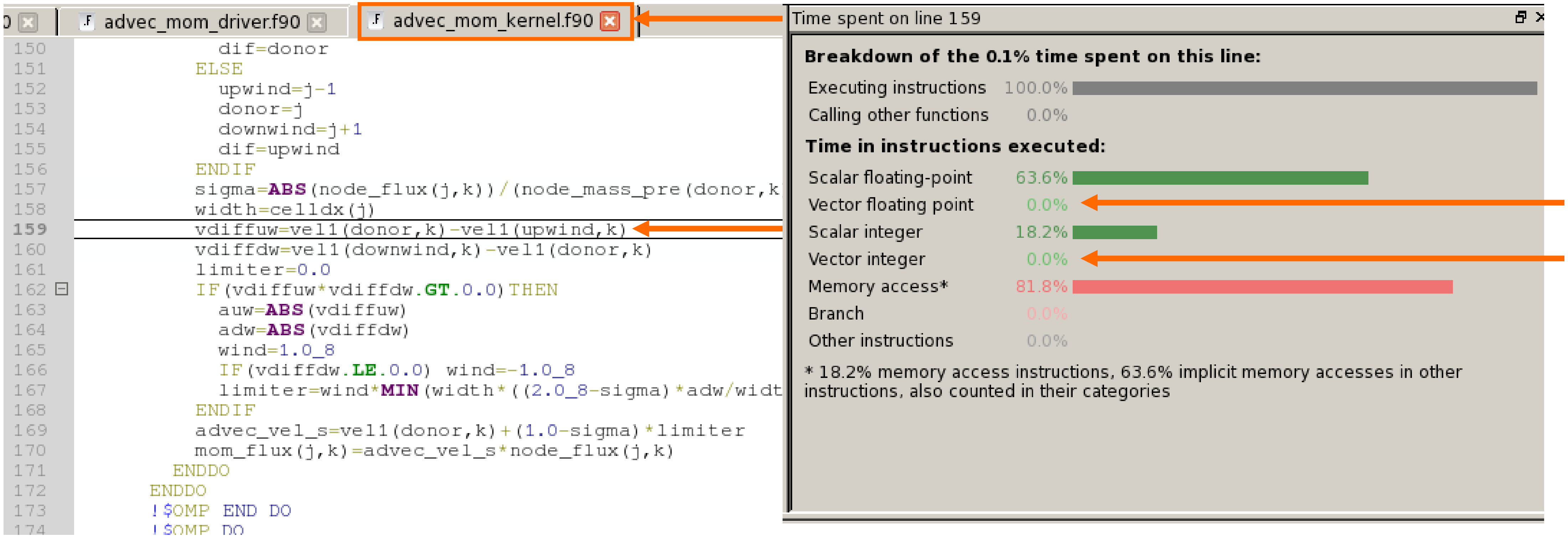
Time in collective calls	20.9%
Time in point-to-point calls	79.1%
Effective process collective rate	1.55 kB/s
Effective process point-to-point rate	33.1 MB/s

Most of the time is spent in point-to-point calls with a low transfer rate. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait.

How much of the code is vectorized?



Where is the code vectorized?



Follow Performance Reports advice

```
advec_mom_kernel.f90
...
144    DO k=y_min,y_max+1
145      DO j=x_min-1,x_max+1
146        IF(node_flux(j,k).LT.0.0)THEN
147          upwind=j+2
148          donor=j+1
149          downwind=j
150          dif=donor
151        ELSE
152          upwind=j-1
153          donor=j
154          downwind=j+1
155          dif=upwind
156        ENDIF
157        sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
158        width=celldx(j)
159        vdiffuw=vel1(donor,k)-vel1(upwind,k)
160        vdiffdw=vel1(downwind,k)-vel1(donor,k)
...
```

```
-fopt-info-vec-missed

advec_mom_kernel.f90:145: note: not vectorized: control flow in loop.
advec_mom_kernel.f90:145: note: bad inner-loop form.
advec_mom_kernel.f90:145: note: not vectorized: Bad inner loop.
advec_mom_kernel.f90:145: note: bad loop form.
Analyzing loop at advec_mom_kernel.f90:145

advec_mom_kernel.f90:145: note: not vectorized: control flow in loop.
advec_mom_kernel.f90:145: note: bad loop form.
```

How well is the compiler vectorizing?

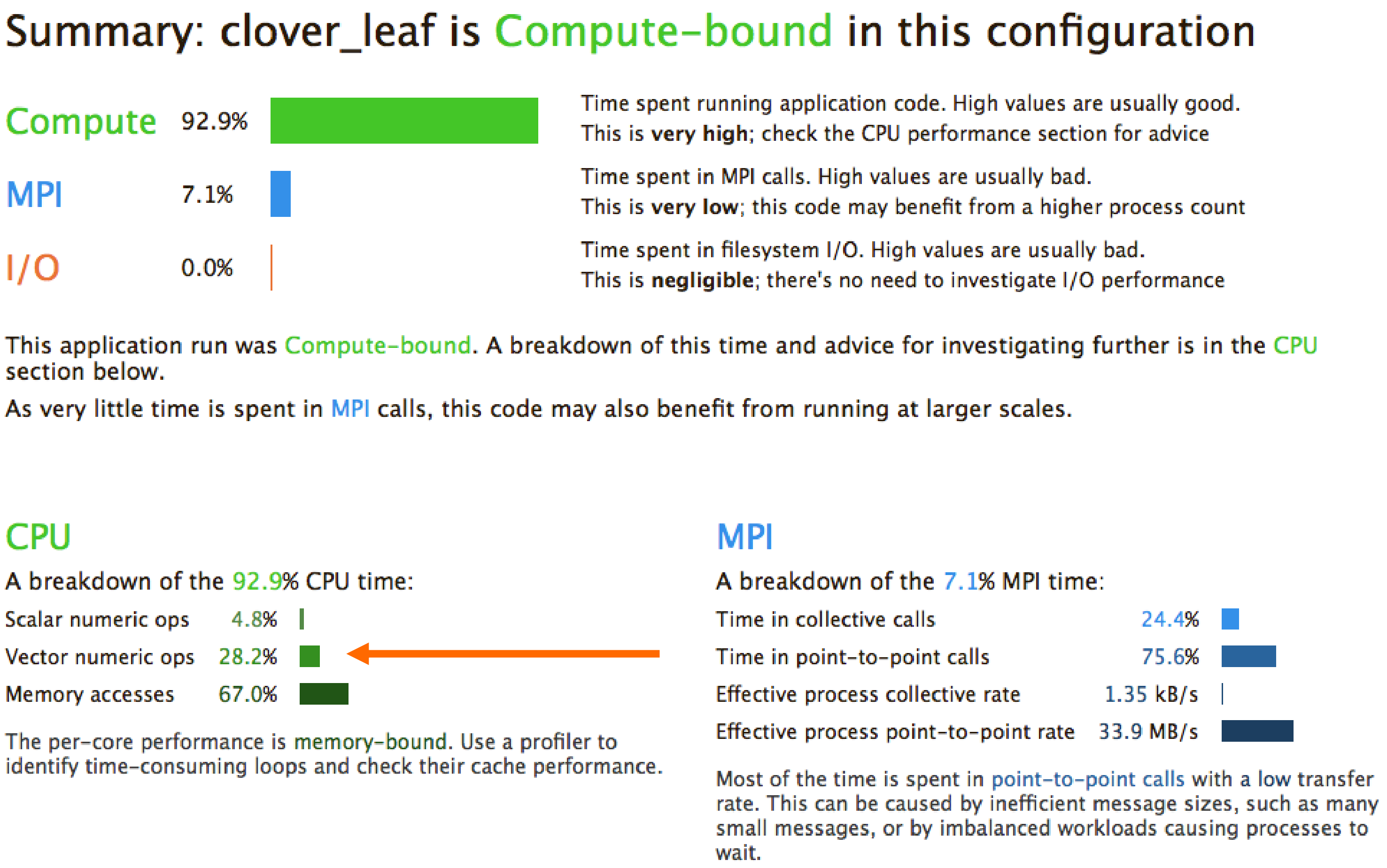
```
advec_mom_kernel.f90
...
144    DO k=y_min,y_max+1
145      DO j=x_min-1,x_max+1
146        IF(node_flux(j,k).LT.0.0)THEN
147          upwind=j+2
148          donor=j+1
149          downwind=j
150          dif=donor
151        ELSE
152          upwind=j-1
153          donor=j
154          downwind=j+1
155          dif=upwind
156        ENDIF
157        sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
158        width=celldx(j)
159        vdiffuw=vel1(donor,k)-vel1(upwind,k)
160        vdiffdw=vel1(downwind,k)-vel1(donor,k)
...
```

```
-qopt-report=2

LOOP BEGIN at advec_mom_kernel.f90(145,9)
  <Peeled loop for vectorization>
    remark #25456: Number of Array Refs Scalar Replaced In Loop: 2
  LOOP END

  LOOP BEGIN at advec_mom_kernel.f90(145,9)
    remark #15300: LOOP WAS VECTORIZED
  LOOP END

  LOOP BEGIN at advec_mom_kernel.f90(145,9)
  <Remainder loop for vectorization>
  LOOP END
```

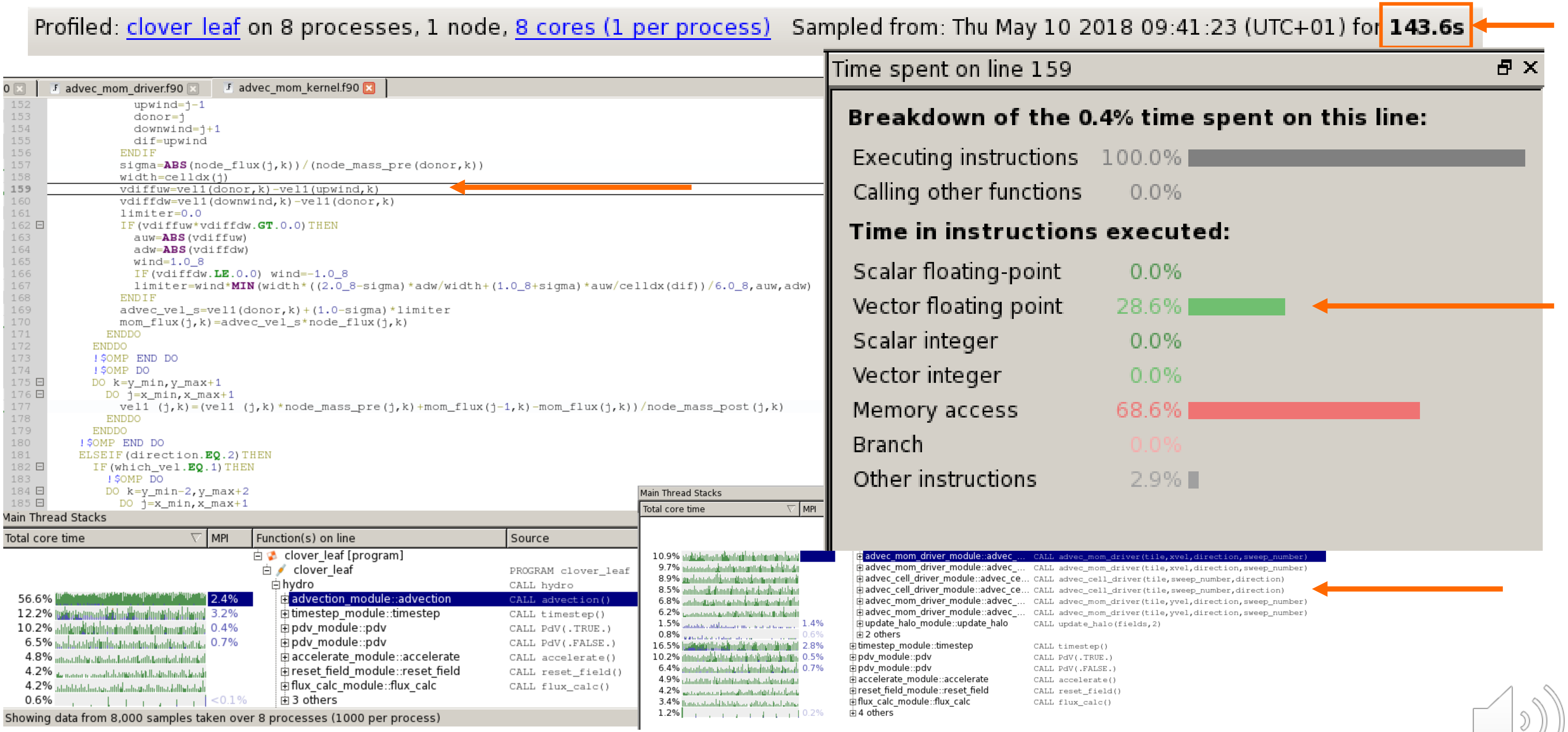
Analyze the results

Running Performance Reports with CloverLeaf using 8 MPI tasks indicates that:

- Time spent in scalar ops is 4.8%
- Time spent in vector ops 28.2%



Where is the code vectorized?



Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!