



arm

# Arm cross- platform tools

VI-HPS platform  
October 16, 2018

# An introduction to Arm

Arm is the world's leading semiconductor intellectual property supplier

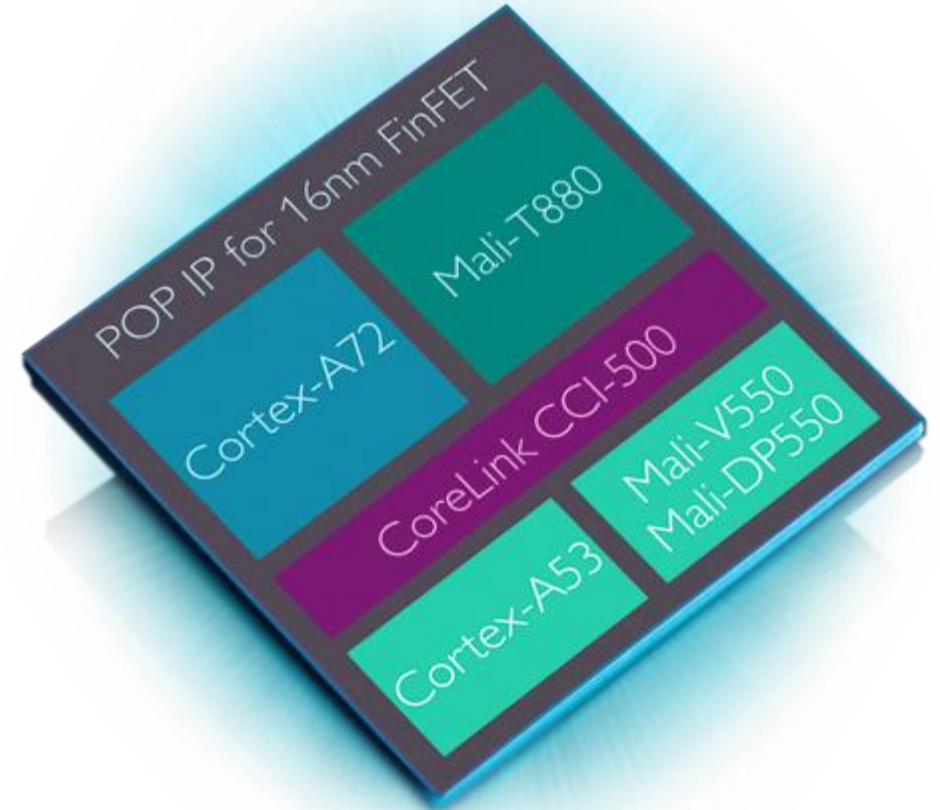
We license to over 350 partners: present in 95% of smart phones, 80% of digital cameras, 35% of all electronic devices, and a total of 60 billion Arm cores have been shipped since 1990

## 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





# 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

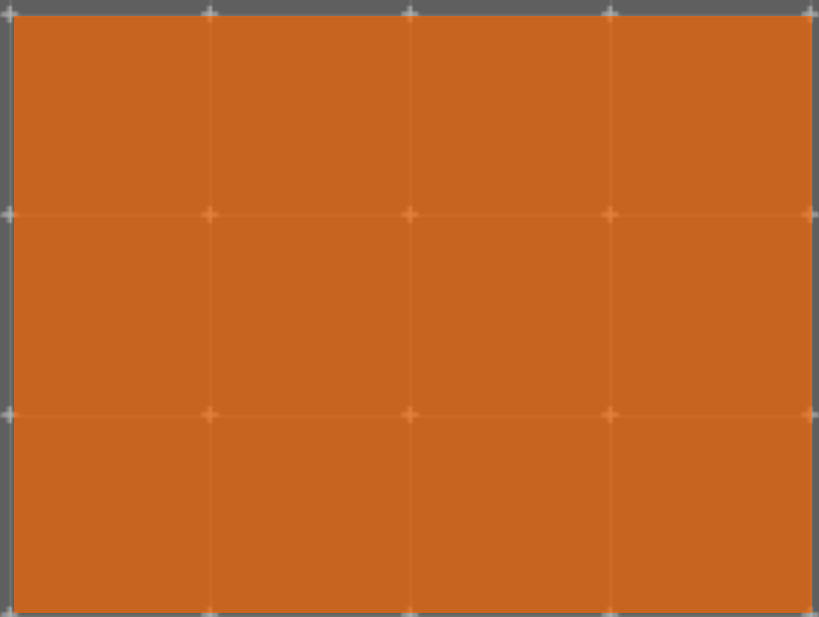
### Co-Design

- 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 Alinea 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  
ALLINEA STUDIO

- ❖ C/C++ Compiler
- ❖ Fortran Compiler
- ❖ Performance Libraries
- ❖ Forge (DDT and MAP)
- ❖ Performance Reports

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

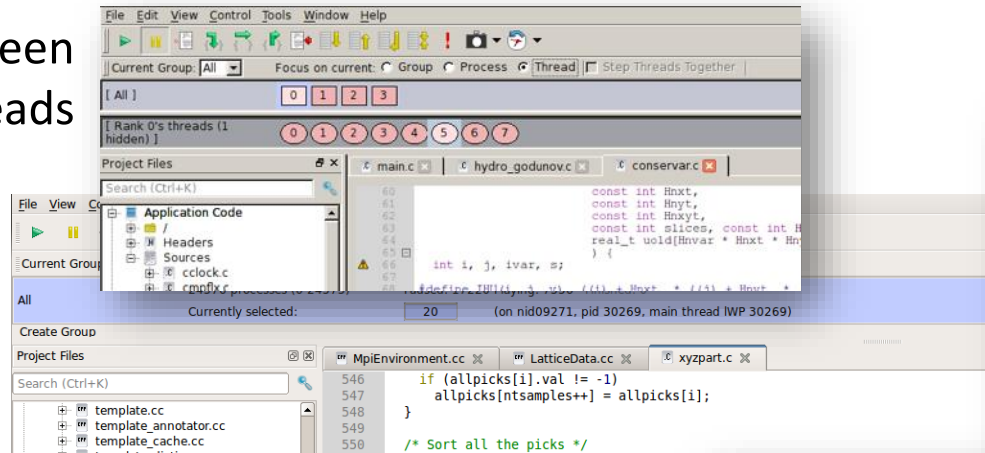
- Powerful and in-depth error detection mechanisms (including memory debugging)
- Sampling-based profiler to identify and understand bottlenecks
- Available at any scale (from serial to petaflop applications)

## Easy to use by everyone

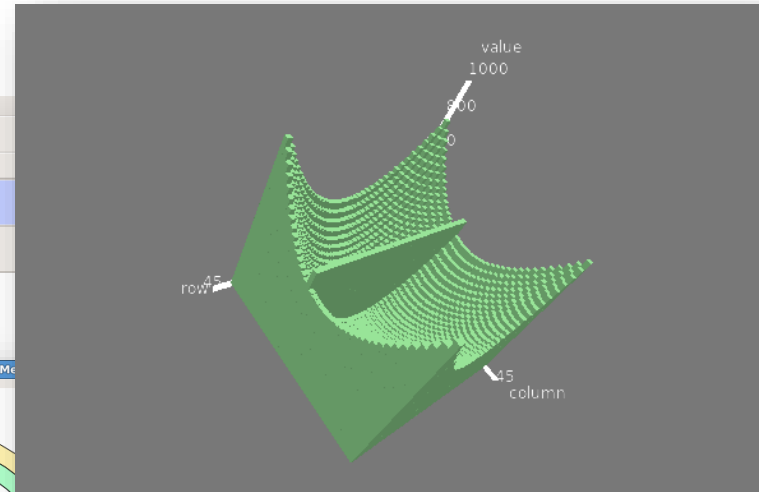
- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

# DDT

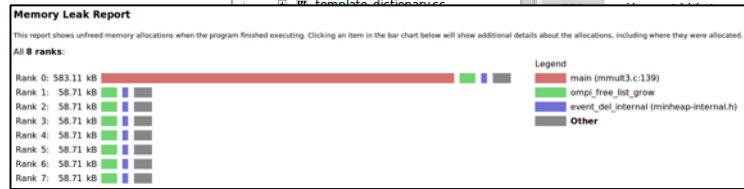
Switch between OpenMP threads



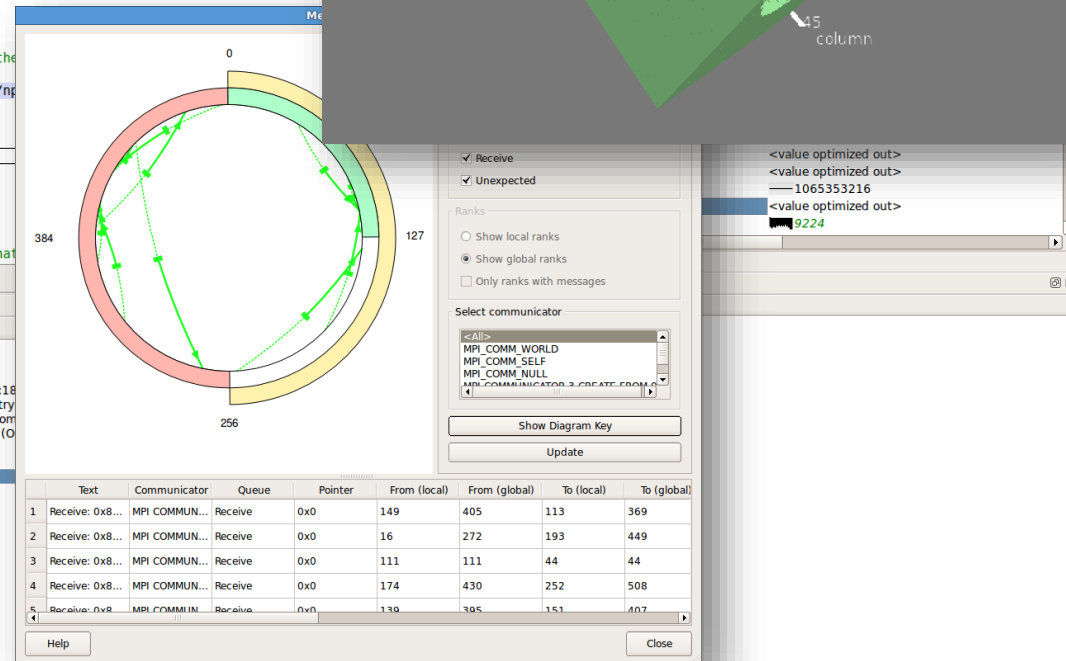
Visualise data structures



Integrate to continuous integration tools

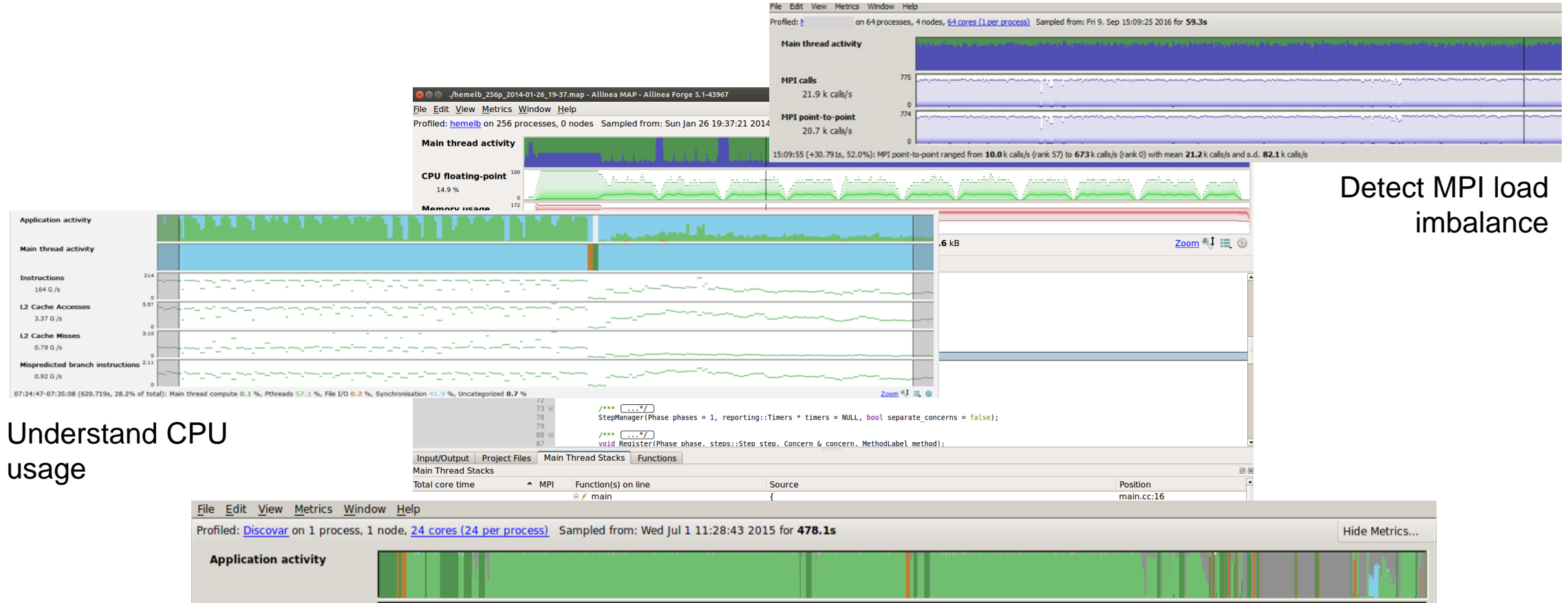


Display pending communications





# Arm MAP

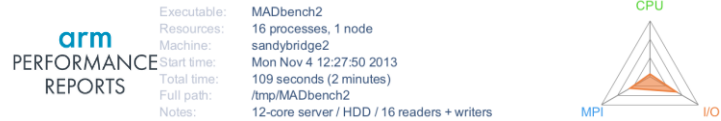


Detect MPI load imbalance

Understand CPU usage

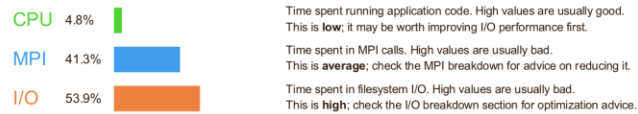
Identify regions of high OpenMP synchronisation

# Arm Performance Reports



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

The total wallclock time was spent as follows:



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

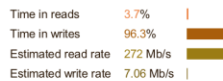
A breakdown of how the **4.8%** total CPU time was spent:



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.

## I/O

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



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.

## MPI

Of the **41.3%** total time spent in MPI calls:



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.

## Memory

Per-process memory usage may also affect scaling:



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

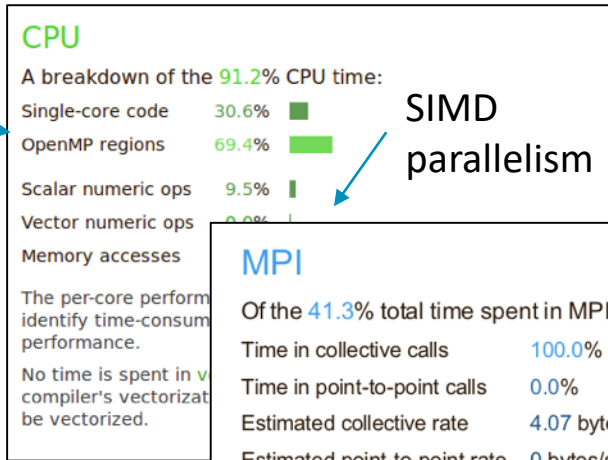
Fully scalable, very low overhead

Rich set of metrics

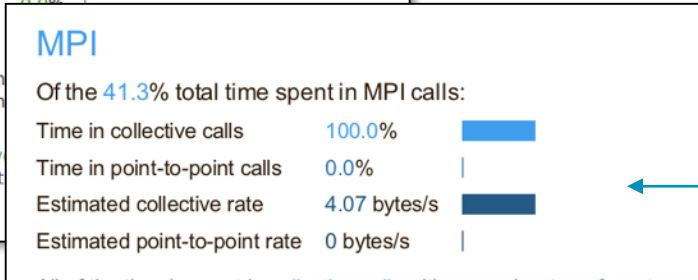
Powerful data analysis

# Arm Performance Reports

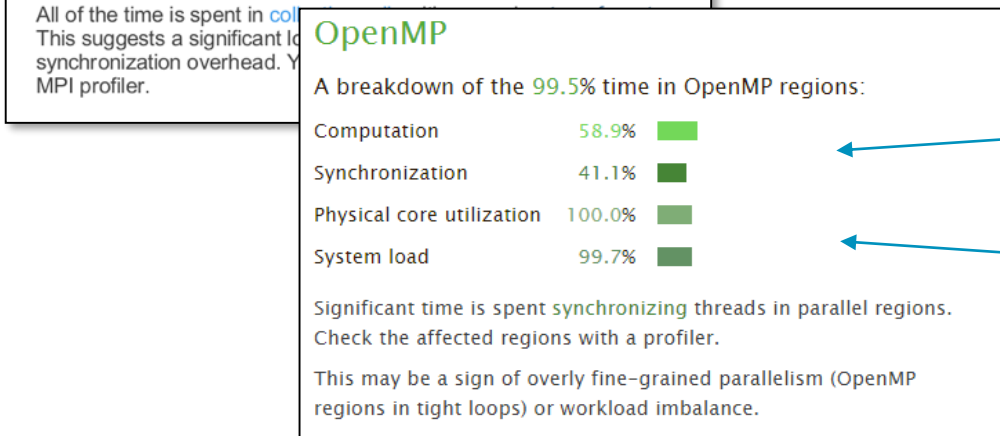
Multi-threaded parallelism



SIMD parallelism

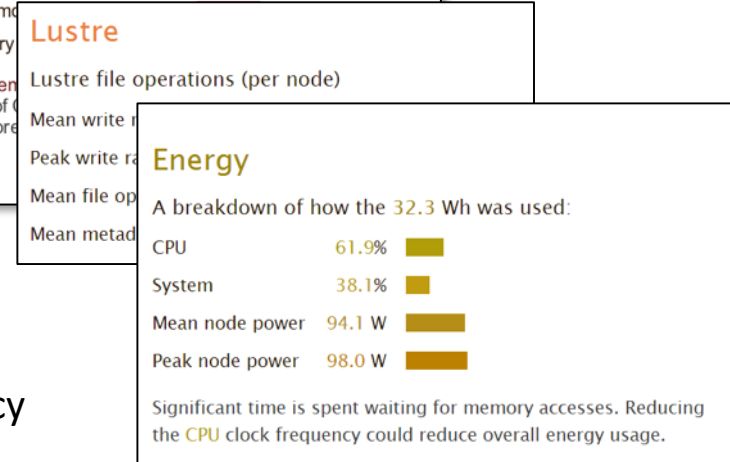
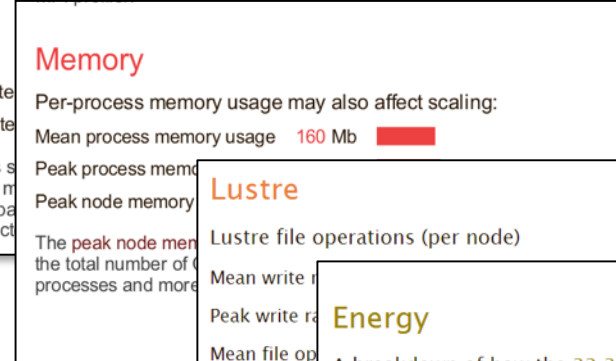
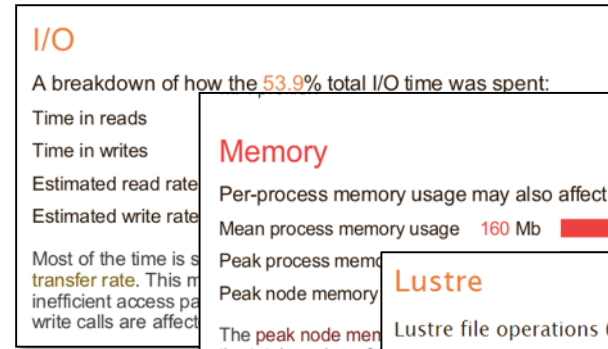


Load imbalance

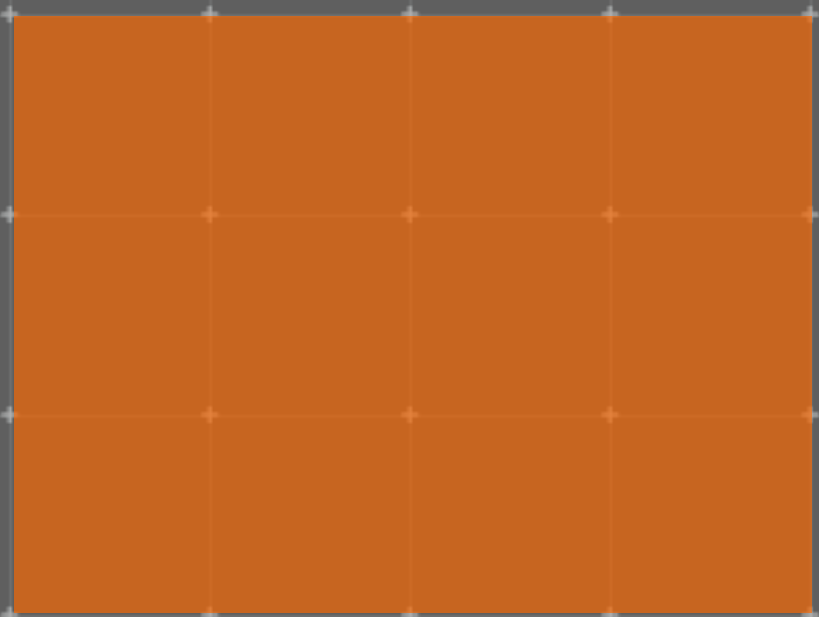


OMP efficiency

System usage

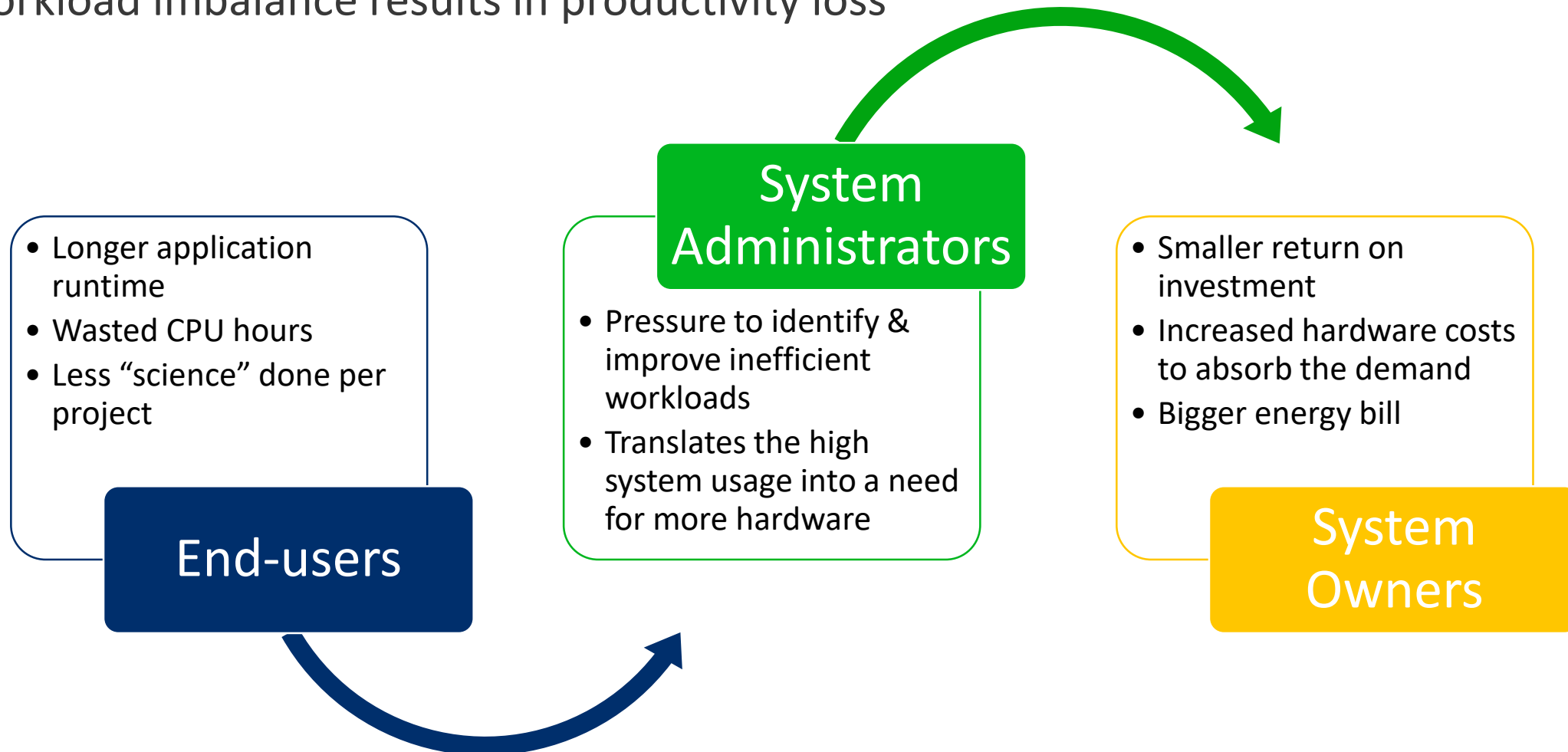


# Imbalance



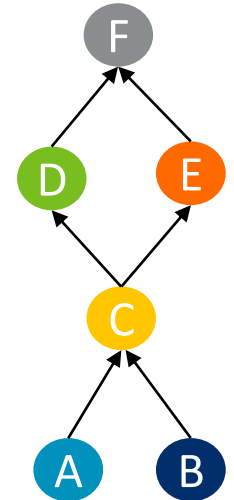
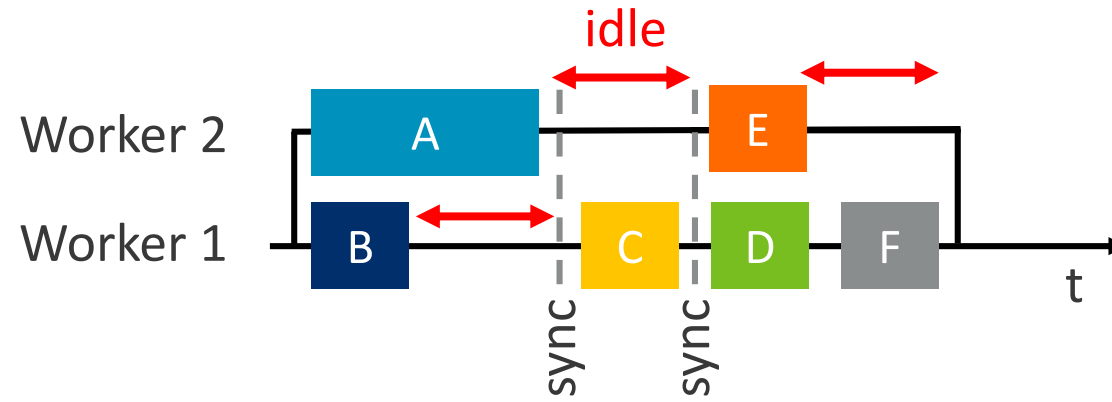
# Challenges

- Workload imbalance results in productivity loss





# 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

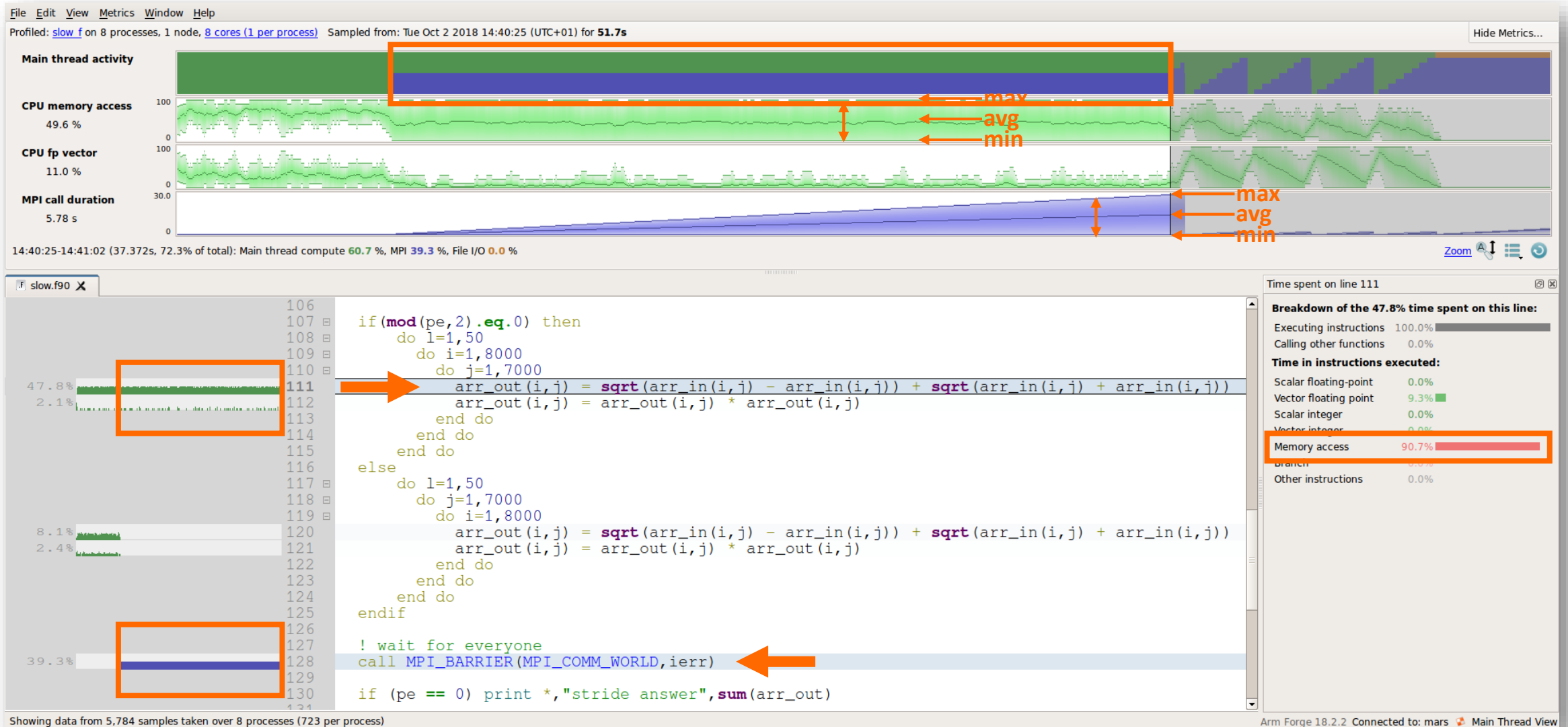
A breakdown of the 74.5% time in OpenMP regions:

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

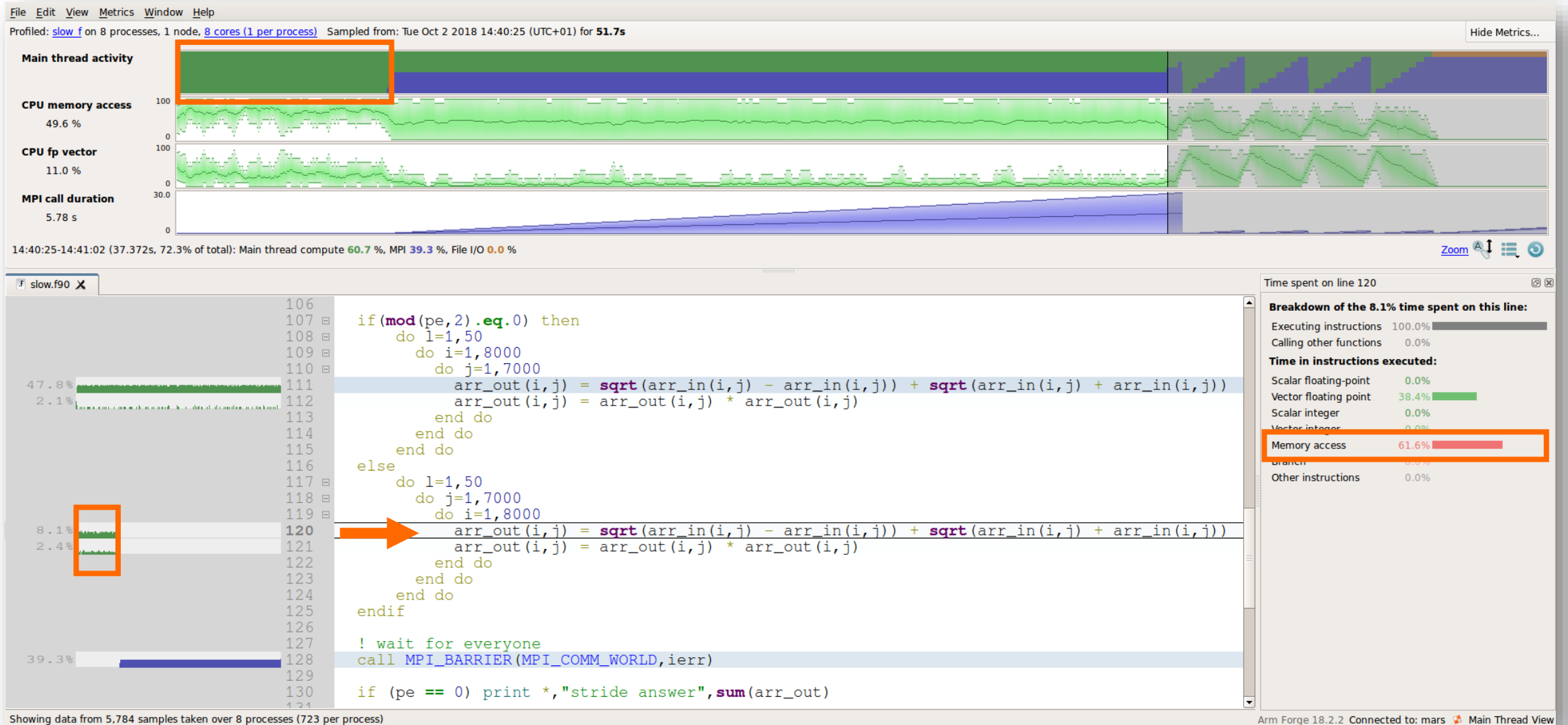
Significant time is spent **synchronizing** threads in parallel regions. Check the affected regions with a profiler.

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

# MPI imbalance: barrier



# MPI imbalance: barrier

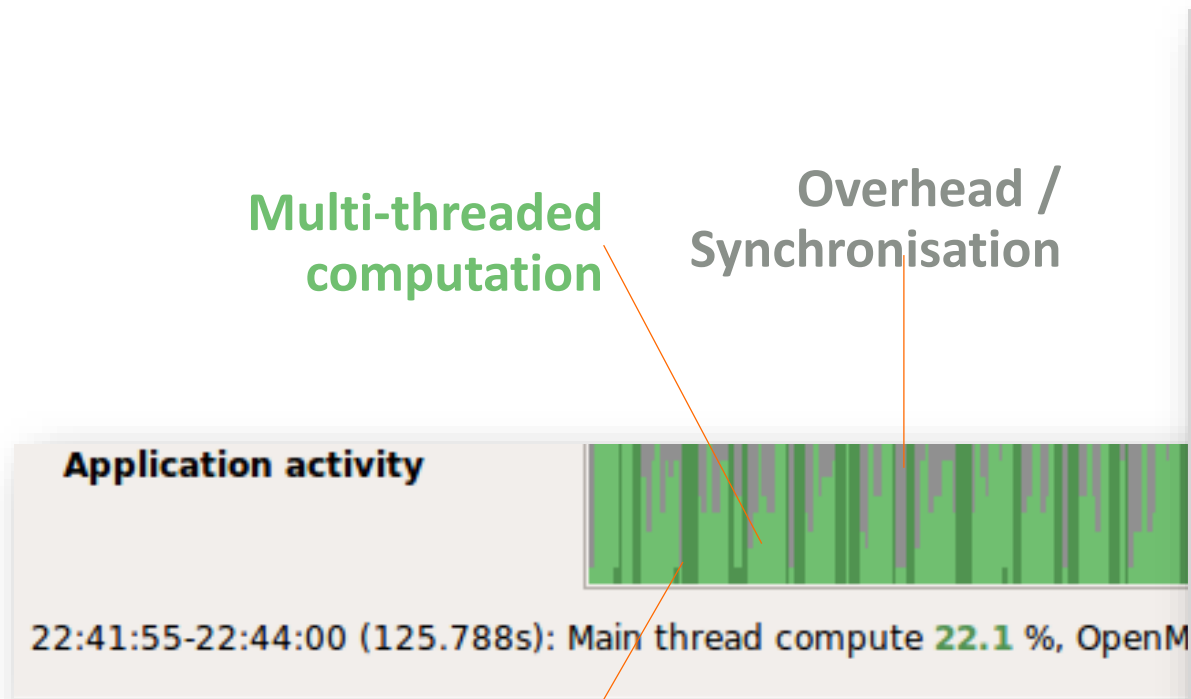


# IO imbalance





# OpenMP imbalance: implicit synchronization



Multi-threaded  
computation

Overhead /  
Synchronisation

Single-threaded  
computation

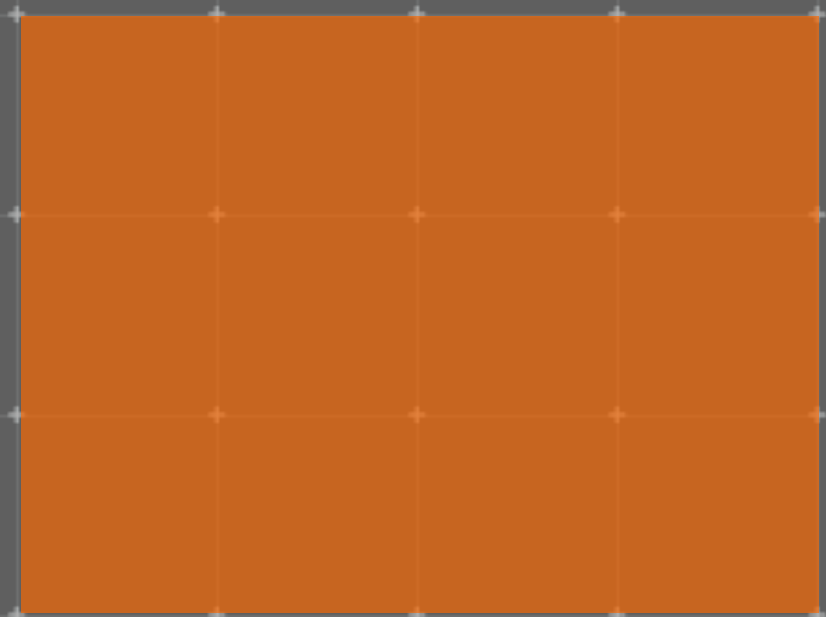


# Understanding resource usage

- Memory accesses



# 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

<b>Compute</b>	93.4%		Time spent running application code. High values are usually good. This is <b>very high</b> ; check the CPU performance section for advice
<b>MPI</b>	6.6%		Time spent in MPI calls. High values are usually bad. This is <b>very low</b> ; this code may benefit from a higher process count
<b>I/O</b>	0.0%		Time spent in filesystem I/O. High values are usually bad. This is <b>negligible</b> ; 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:

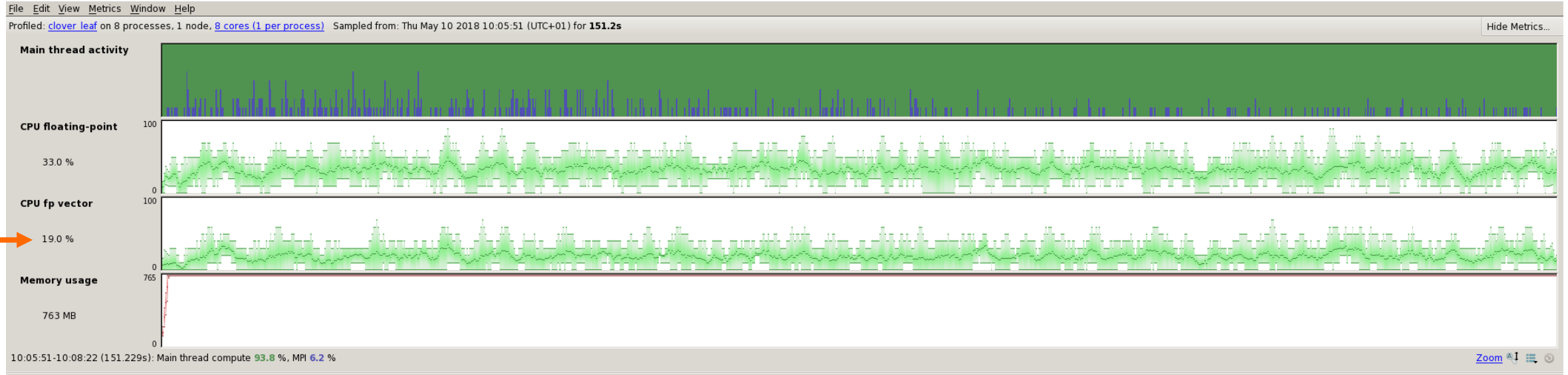
Time in collective calls	20.9%	
Time in point-to-point calls	79.1%	
Effective process collective rate	1.55 kB/s	
Effective process point-to-point rate	33.1 MB/s	

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?

Profiled: [clover leaf](#) on 8 processes, 1 node, [8 cores \(1 per process\)](#) Sampled from: Thu May 10 2018 10:05:51 (UTC+01) for **151.2s**



Total core time	MPI	Function(s) on line	Source
53.3%	2.0%	clover_leaf [program]	PROGRAM clover_leaf
16.5%	2.8%	clover_leaf	PROGRAM clover_leaf
10.2%	0.5%	advection_module::advection	CALL advection()
6.4%	0.7%	timestep_module::timestep	CALL timestep()
4.9%		pdv_module::pdv	CALL PdV(.TRUE.)
4.2%		pdv_module::pdv	CALL PdV(.FALSE.)
3.4%		accelerate_module::accelerate	CALL accelerate()
1.2%	0.2%	reset_field_module::reset_field	CALL reset_field()
		flux_calc_module::flux_calc	CALL flux_calc()
		4 others	

Total core time	MPI	Function(s) on line	Source
11.1%		clover_leaf [program]	PROGRAM clover_leaf
10.2%		clover_leaf	PROGRAM clover_leaf
9.8%		hydro	CALL hydro
9.1%		advection_module::advection	CALL advection()
7.1%		advec_mom_driver_module::advec...	CALL advec_mom_driver(tile,xvel,direction,sweep_number)
6.6%		advec_mom_driver_module::advec...	CALL advec_mom_driver(tile,yvel,direction,sweep_number)
1.7%	1.7%	update_halo_module::update_halo	CALL update_halo(fields,2)
0.9%	0.7%	2 others	
12.2%	3.2%	timestep_module::timestep	CALL timestep()
10.2%	0.4%	pdv_module::pdv	CALL PdV(.TRUE.)
6.5%	0.7%	pdv_module::pdv	CALL PdV(.FALSE.)
4.8%		accelerate_module::accelerate	CALL accelerate()
4.2%		reset_field_module::reset_field	CALL reset_field()
4.2%		flux_calc_module::flux_calc	CALL flux_calc()
0.6%	<0.1%	3 others	
0.6%	0.1%	1 other	

Showing data from 8,000 samples taken over 8 processes (1000 per process)





# Where is the code vectorized?

The screenshot shows a code editor with Fortran code and a performance analysis window. The code editor has three tabs: 'advec\_mom\_driver.f90', 'advec\_mom\_kernel.f90', and 'advec\_mom\_driver.f90'. The 'advec\_mom\_kernel.f90' tab is active and highlighted with an orange box. An orange arrow points from this tab to the performance analysis window. The code in the editor is as follows:

```
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)
161     limiter=0.0
162     IF (vdiffuw*vdiffdw.GT.0.0) THEN
163     auw=ABS (vdiffuw)
164     adw=ABS (vdiffdw)
165     wind=1.0_8
166     IF (vdiffdw.LE.0.0) wind=-1.0_8
167     limiter=wind*MIN (width* (2.0_8-sigma) * adw/wid
168     ENDIF
169     advec_vel_s=vel1(donor,k) + (1.0-sigma) * limiter
170     mom_flux(j,k)=advec_vel_s*node_flux(j,k)
171     ENDDO
172     ENDDO
173     !$OMP END DO
174     !$OMP DO
```

The performance analysis window is titled 'Time spent on line 159'. It shows a breakdown of the 0.1% time spent on this line:

Breakdown of the 0.1% time spent on this line:	
Executing instructions	100.0%
Calling other functions	0.0%
Time in instructions executed:	
Scalar floating-point	63.6%
Vector floating point	0.0%
Scalar integer	18.2%
Vector integer	0.0%
Memory access*	81.8%
Branch	0.0%
Other instructions	0.0%

\* 18.2% memory access instructions, 63.6% implicit memory accesses in other instructions, also counted in their categories

Orange arrows in the image point from the 'advec\_mom\_kernel.f90' tab to line 159 in the code editor, and from the 'Vector floating point' and 'Vector integer' categories in the performance analysis window to the code editor.

# 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
```

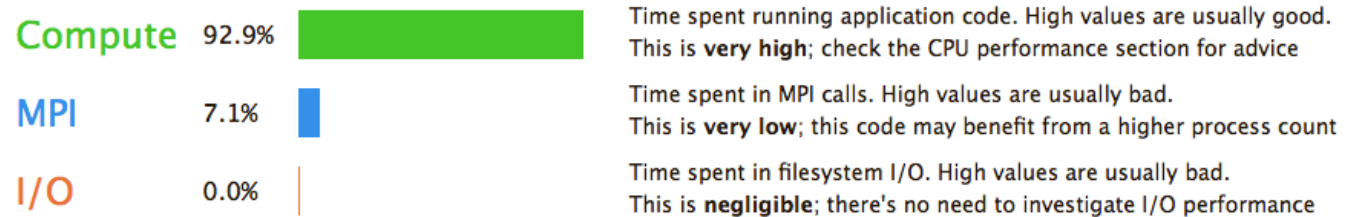
```
LOOP BEGIN at advec_mom_kernel.f90(145,9)
<Remainder loop for vectorization>
LOOP END
```

# 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%

Summary: clover\_leaf is **Compute-bound** in this configuration

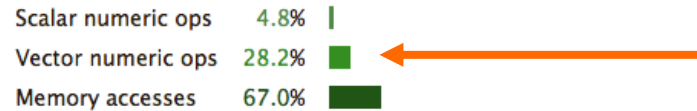


This application run was **Compute-bound**. A breakdown of this time and advice for investigating further is in the **CPU** section below.

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

## CPU

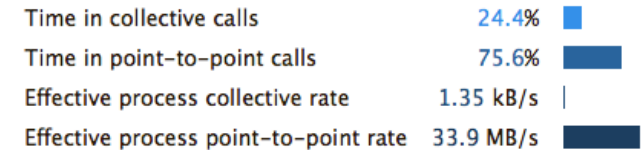
A breakdown of the **92.9%** CPU time:



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

## MPI

A breakdown of the **7.1%** MPI time:



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.



# Where is the code vectorized?

Profiled: [clover leaf](#) on 8 processes, 1 node, [8 cores \(1 per process\)](#) Sampled from: Thu May 10 2018 09:41:23 (UTC+01) for **143.6s**

```
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)
161     limiter=0.0
162     IF(vdiffuw*vdiffdw.GT.0.0) THEN
163         auw=ABS(vdiffuw)
164         adw=ABS(vdiffdw)
165         wind=1.0_8
166         IF(vdiffdw.LE.0.0) wind=-1.0_8
167         limiter=wind*MIN(width*((2.0_8-sigma)*adw/width+(1.0_8+sigma)*auw/width)/6.0_8,auw,adw)
168     ENDIF
169     advect_vel_s=vel1(donor,k)+(1.0-sigma)*limiter
170     mom_flux(j,k)=advect_vel_s*node_flux(j,k)
171     ENDDO
172     ENDDO
173     !$OMP END DO
174     !$OMP DO
175     DO k=y_min,y_max+1
176     DO j=x_min,x_max+1
177         vel1(j,k)=(vel1(j,k)*node_mass_pre(j,k)+mom_flux(j-1,k)-mom_flux(j,k))/node_mass_post(j,k)
178     ENDDO
179     ENDDO
180     !$OMP END DO
181     ELSEIF(direction.EQ.2) THEN
182     IF(which_vel.EQ.1) THEN
183     !$OMP DO
184     DO k=y_min-2,y_max+2
185     DO j=x_min,x_max+1
```

### Time spent on line 159

**Breakdown of the 0.4% time spent on this line:**

Executing instructions	100.0%
Calling other functions	0.0%

**Time in instructions executed:**

Scalar floating-point	0.0%
Vector floating point	28.6%
Scalar integer	0.0%
Vector integer	0.0%
Memory access	68.6%
Branch	0.0%
Other instructions	2.9%

Total core time	MPI	Function(s) on line	Source
56.6%	2.4%	clover_leaf [program]	PROGRAM clover_leaf
12.2%	3.2%	hydro	CALL hydro
10.2%	0.4%	advection_module::advection	CALL advection()
6.5%	0.7%	timestep_module::timestep	CALL timestep()
4.8%		pdv_module::pdv	CALL PdV(.TRUE.)
4.2%		pdv_module::pdv	CALL PdV(.FALSE.)
4.2%		accelerate_module::accelerate	CALL accelerate()
4.2%		reset_field_module::reset_field	CALL reset_field()
0.6%	<0.1%	flux_calc_module::flux_calc	CALL flux_calc()
		3 others	

Main Thread Stacks	Total core time	MPI
advec_mom_driver_module::advec...	10.9%	
advec_mom_driver_module::advec...	9.7%	
advec_cell_driver_module::advec...	8.9%	
advec_cell_driver_module::advec...	8.5%	
advec_mom_driver_module::advec...	6.8%	
advec_mom_driver_module::advec...	6.2%	
update_halo_module::update_halo...	1.5%	1.4%
2 others	0.8%	0.6%
timestep_module::timestep	16.5%	2.8%
pdv_module::pdv	10.2%	0.5%
pdv_module::pdv	6.4%	0.7%
accelerate_module::accelerate	4.9%	
reset_field_module::reset_field	4.2%	
flux_calc_module::flux_calc	3.4%	
4 others	1.2%	0.2%

advec_mom_driver_module::advec...	CALL advec_mom_driver(tile,xvel,direction,sweep_number)
advec_mom_driver_module::advec...	CALL advec_mom_driver(tile,xvel,direction,sweep_number)
advec_cell_driver_module::advec...	CALL advec_cell_driver(tile,sweep_number,direction)
advec_cell_driver_module::advec...	CALL advec_cell_driver(tile,sweep_number,direction)
advec_mom_driver_module::advec...	CALL advec_mom_driver(tile,yvel,direction,sweep_number)
advec_mom_driver_module::advec...	CALL advec_mom_driver(tile,yvel,direction,sweep_number)
update_halo_module::update_halo...	CALL update_halo(fields,2)
2 others	
timestep_module::timestep	CALL timestep()
pdv_module::pdv	CALL PdV(.TRUE.)
pdv_module::pdv	CALL PdV(.FALSE.)
accelerate_module::accelerate	CALL accelerate()
reset_field_module::reset_field	CALL reset_field()
flux_calc_module::flux_calc	CALL flux_calc()
4 others	





Thank You!

Danke!

Merci!

谢谢!

ありがとう!

Gracias!

Kiitos!

**arm**