Acting on Insight

Tips for developing and optimizing scientific applications

Florent.Lebeau@arm.com
28/06/2019

Agenda

- Introduction
- Maximize application efficiency
- Analyze code performance
- Profile multi-threaded codes
- Optimize Python-based applications
- Visualize code regions with Caliper

Arm Technology Already Connects the World

Arm is ubiquitous

21 billion chips sold by partners in 2017

#1 in Infrastructure today with 28% market shares

Partnership is key

We design IP, we do not manufacture chips

Partners build products for their target markets

Choice is good

One size is not always the best fit for all

HPC is a great fit for co-design and collaboration

Arm’s solution for HPC application development and porting

Combines cross-platform tools with Arm only tools for a comprehensive solution

Cross-platform Tools

arm FORGE

DDT

MAP

arm PERFORMANCE REPORTS

+

Arm Architecture Tools

arm C/C++ & FORTRAN COMPILER

arm PERFORMANCE LIBRARIES

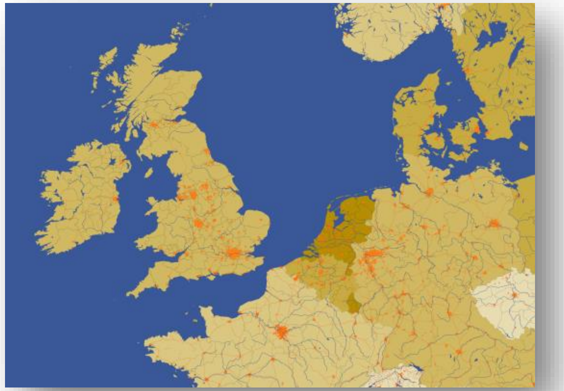
=

arm

The billion dollar question in "*weather and forecasting*"

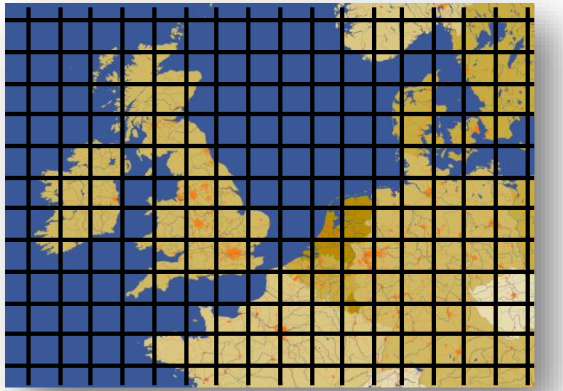
Is it going to rain tomorrow?

1. Choose domain

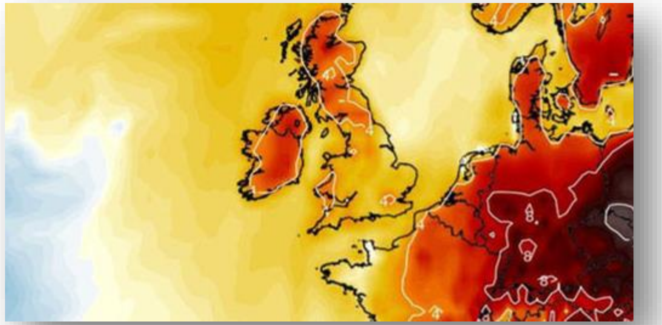


2. Gather Data

3. Create Mesh



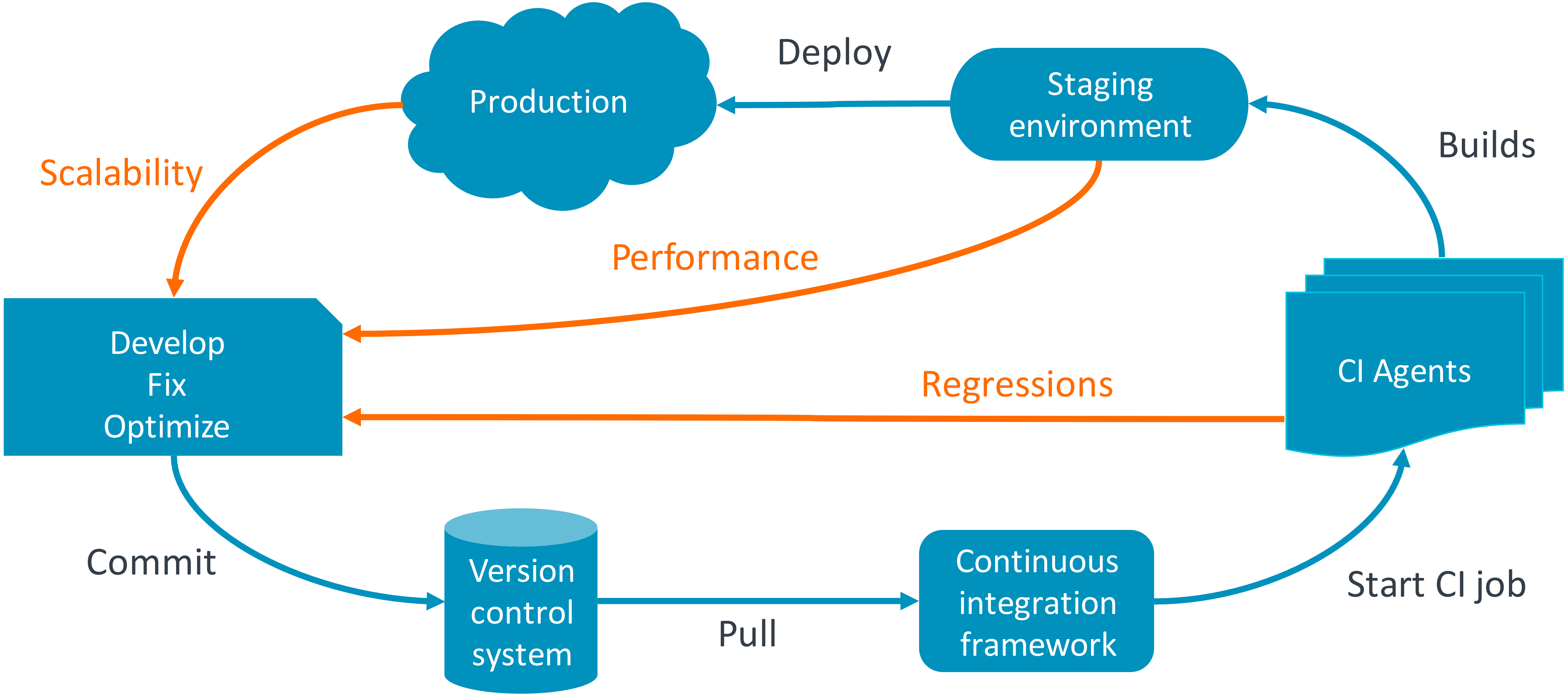
4. Match Data to Mesh



5. Simulate

6. Visualize

Weather forecasting workflow



- **24 hour timeframe**
- **2 to 3 test runs for 1 production run**

Application efficiency

Scientist

- Efficient use of allocation time
- Higher result throughput

Developer

- Characterize application behaviour
- Gets hints on next optimization steps

System admin

- Maximize resource usage
- Diagnose performance issues

Decision maker

- High-level view of system workload
- Reporting figures and analysis to help decision making

Arm Performance Reports

Characterize and understand the performance of HPC application runs

Commercially supported by Arm

Accurate and astute insight

Relevant advice to avoid pitfalls

Gathers a rich set of data

- Analyses metrics around CPU, memory, IO, hardware counters, etc.
- Possibility for users to add their own metrics

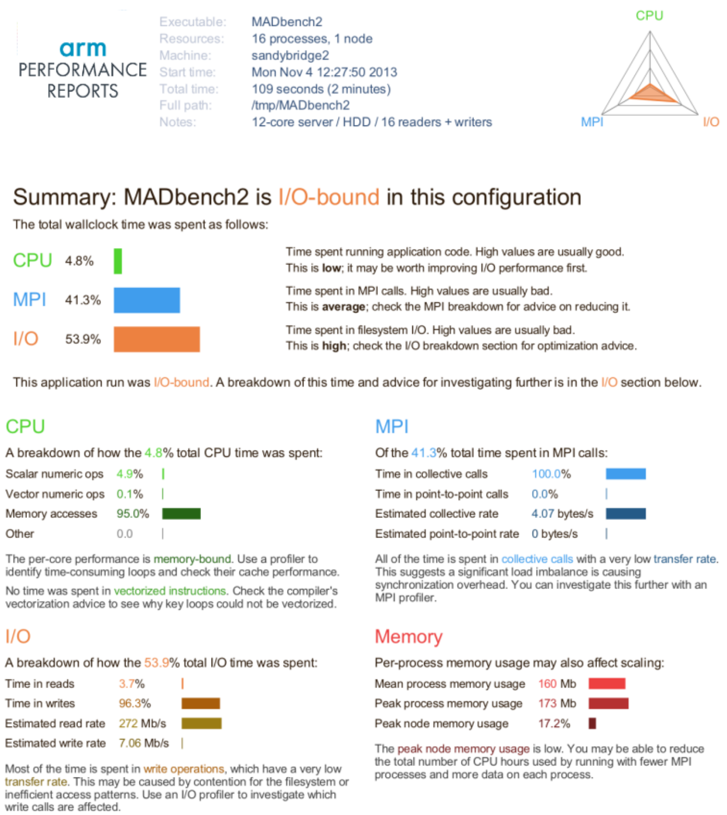
Build a culture of application performance & efficiency awareness

- Analyses data and reports the information that matters to users
- Provides simple guidance to help improve workloads' efficiency

Adds value to typical users' workflows

- Define application behaviour and performance expectations
- Integrate outputs to various systems for validation (e.g. continuous integration)
- Can be automated completely (no user intervention)

Analyze application behavior easily


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

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

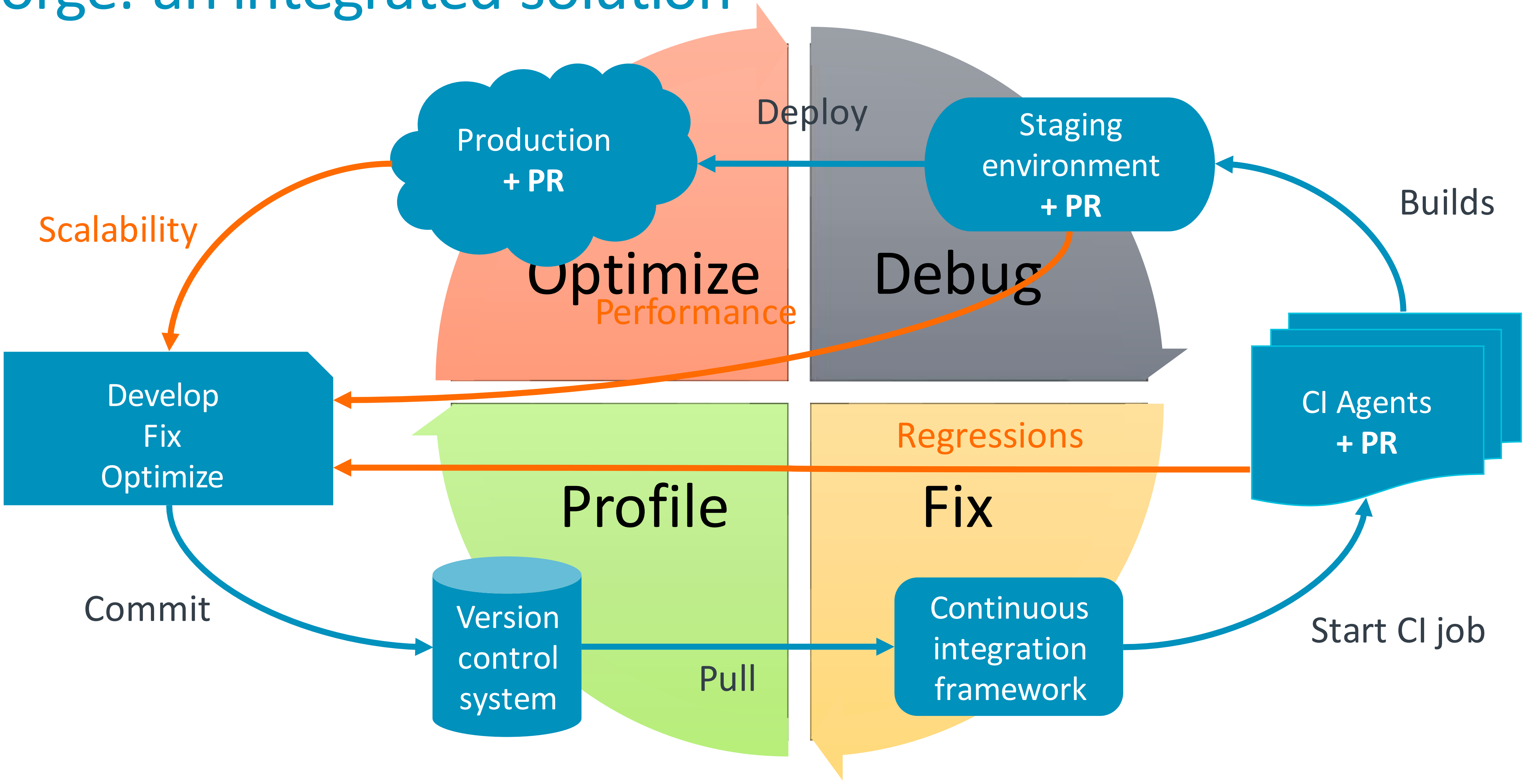
Per-process memory usage may also affect scaling:

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.

- Simple start-up
- No source code needed
- Scalable, low overhead
- Powerful metrics and analysis
- Human and machine-readable results

Forge: an integrated solution



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

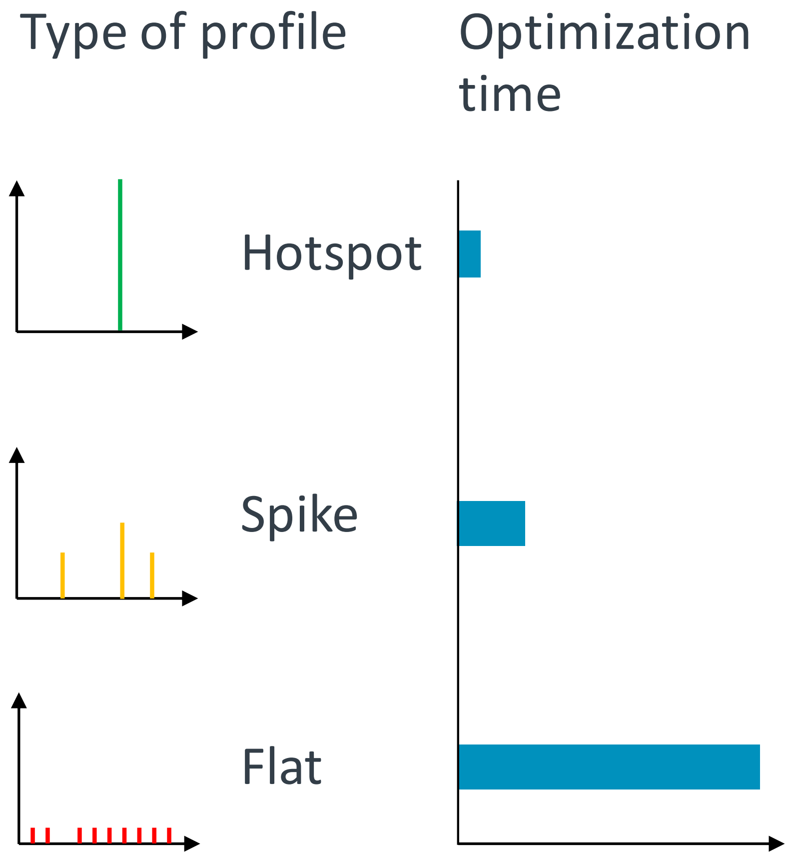
- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

Why profile?

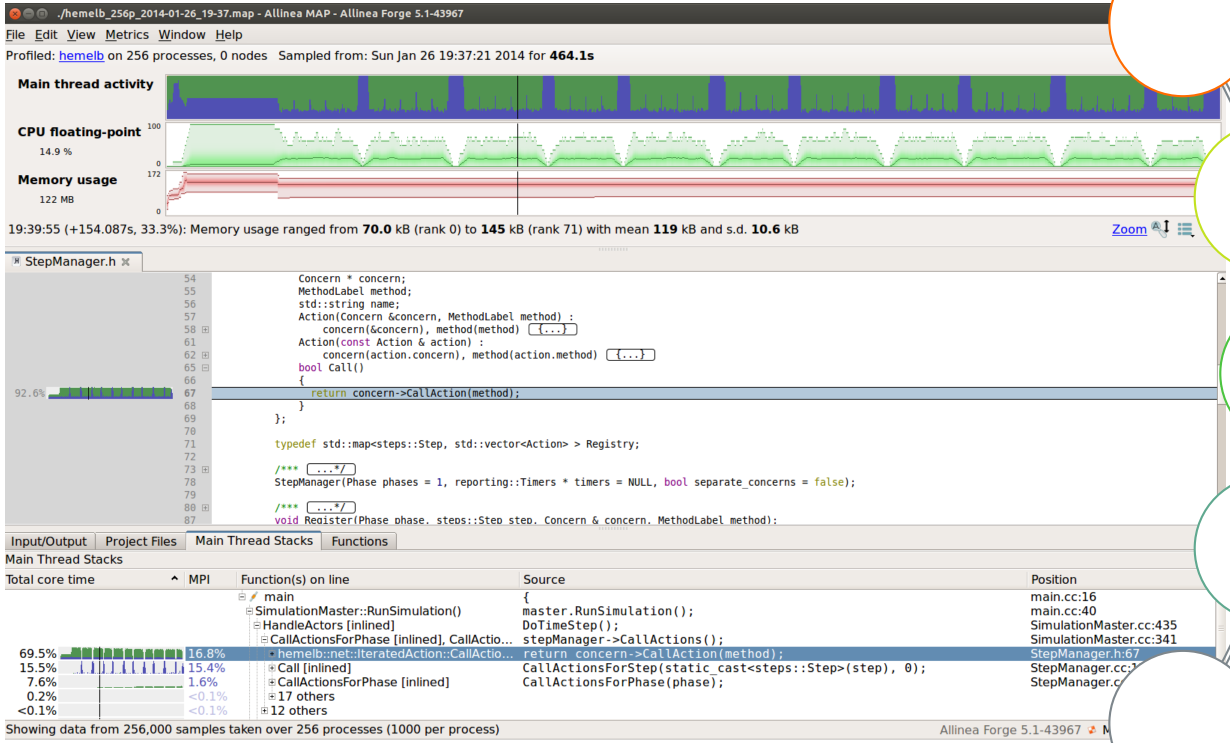
Profiling: a form of dynamic program analysis used to optimize an application.

How to optimize an application?

- Select representative test cases
- Profile
  - Tracing
  - Instrumenting
  - Sampling
- Optimize
- Profile and iterate until your speedup goal has been reached



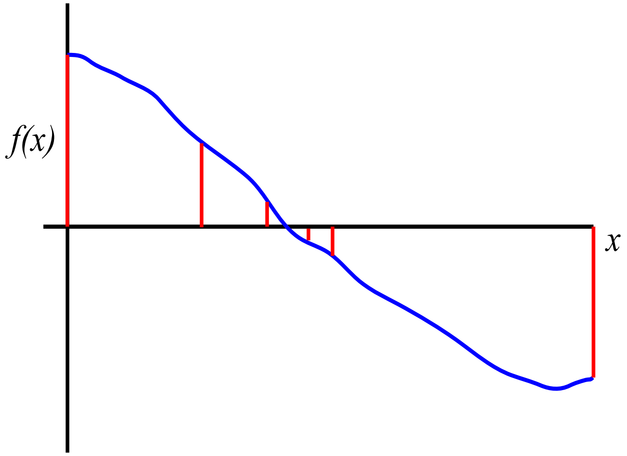
Multi-node low-overhead profiling with Arm MAP



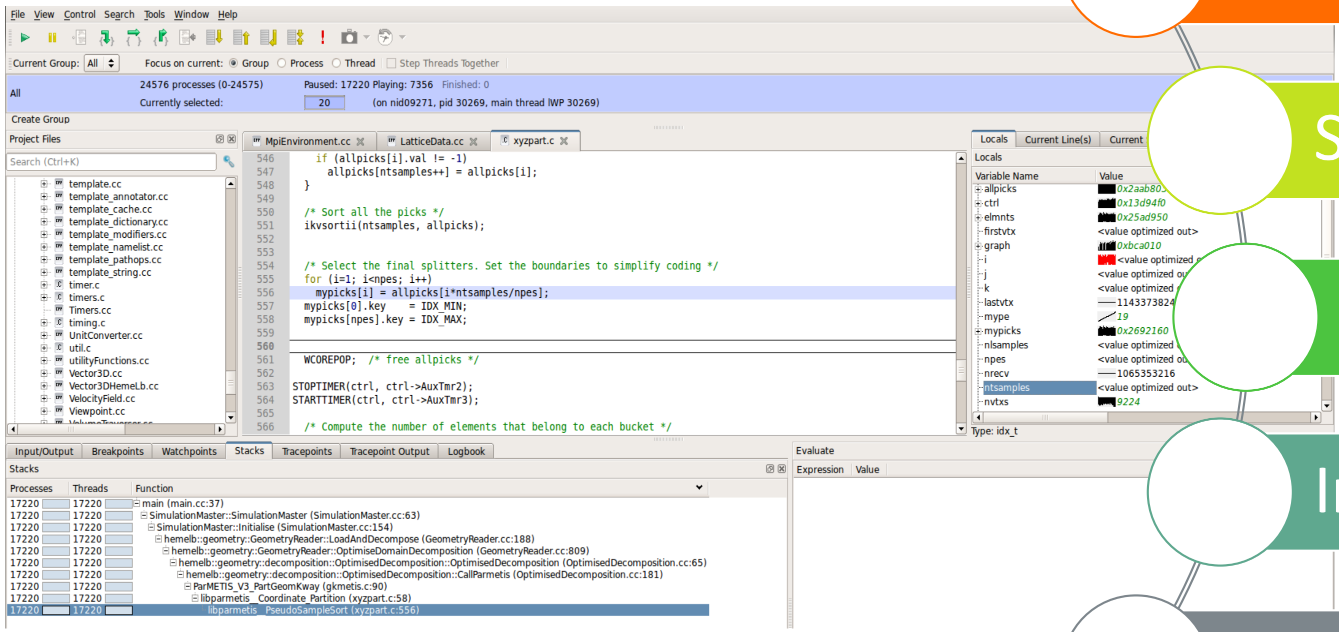
- No instrumentation
- Low overhead
- Scalable
- JSON export
- C/C++, F90, Python profiling

Basic debugging

- The first debugger: print statements
  - Each process prints a message or value at defined locations
  - Diagnose the problem from evidence and intuition
- A long slow process
  - Analogous to bisection root finding
- Broken at modest scale
  - Too much output – too many log files



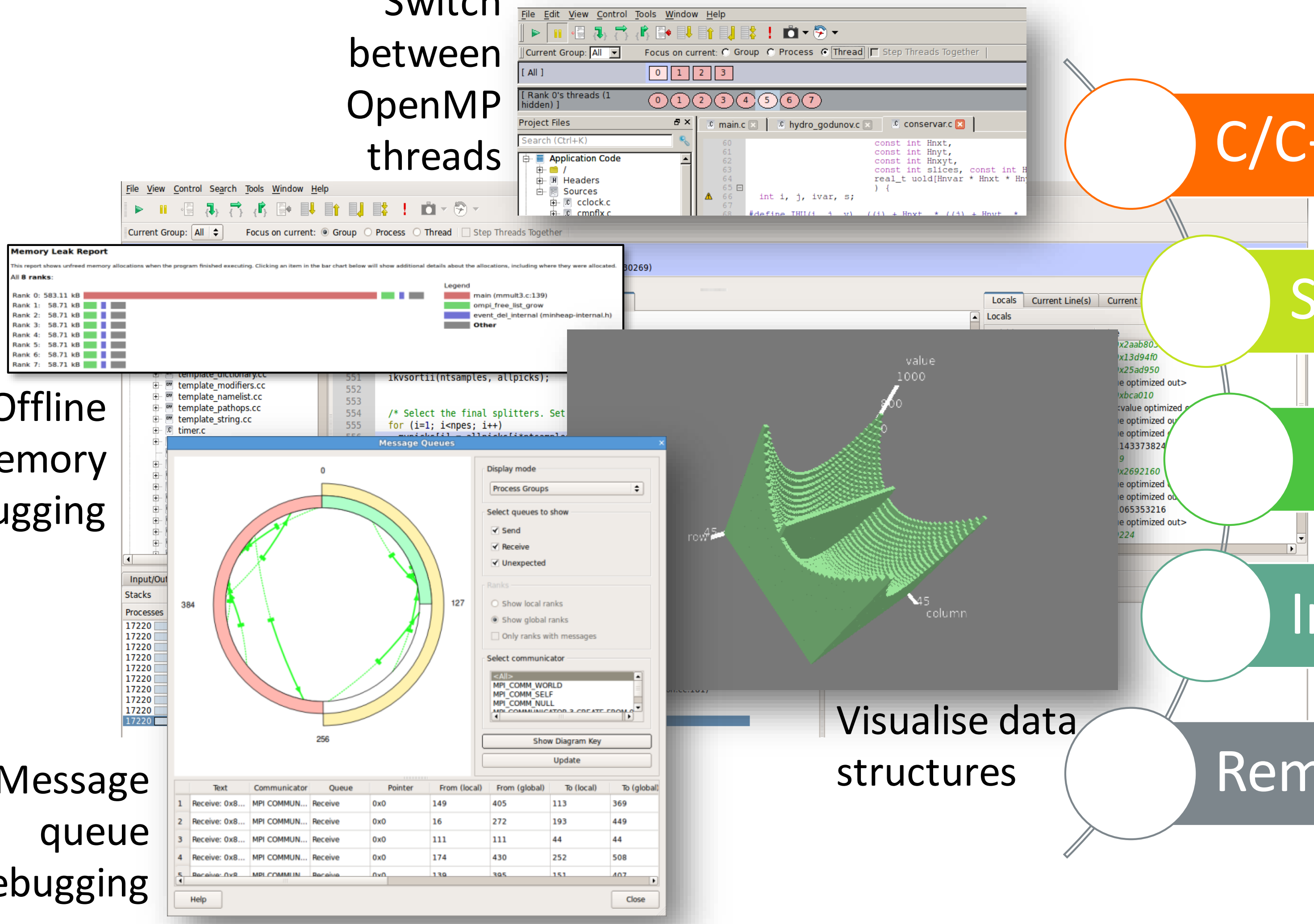
Professional debugging with Arm DDT



- C/C++, Fortran
- Scalable parallel debugger
- Interactive and non-interactive
- Intuitive
- Remote client available

Professional debugging with Arm DDT

Switch between OpenMP threads



Offline memory debugging

Message queue debugging

Visualise data structures

- C/C++, Fortran
- Scalable parallel debugger
- Interactive and non-interactive
- Intuitive
- Remote client available

Hands-on

Set up your environment

- Copy NPB in your workspace:

```
$ cd $SCRATCH/$USER
$ cp -r /p/scratch/share/VI-HPS/examples/NPB3.3-MZ-MPI.tar.gz .
$ tar xf NPB3.3-MZ-MPI.tar.gz
$ cd NPB3.3-MZ-MPI/
```

- Load the MPI, Forge and Performance Reports modules

```
$ module load Intel IntelMPI
$ module use /p/scratch/share/VI-HPS/JURECA/mf/
$ module load Arm-forge Arm-reports
```

Run Arm Performance Report

- Compile your application as usual – no requirements

```
$ make bt-mz CLASS=C NPROCS=8
```

- Edit the job script and submit

```
$ cd bin/
$ cp ../jobscript/jureca/reference.sbatch job.sub
```

Modify the following line to add:

```
perf-report srun -n $PROCS $EXE
$ sbatch -A <youraccount> job.sub
```

- View the results

```
$ firefox bt-mz_C_8p_2n_6t_YYYY-MM-DD_HH-MM.html
$ cat bt-mz_C_8p_2n_6t_YYYY-MM-DD_HH-MM.html
```

Run Arm MAP

- Edit the makefile and compile

```
FFLAGS  = -O3 -g -fno-omit-frame-pointer -no-ip -no-ipo $(OPENMP)
```

  - The debugging option (**-g**) is a requirement for all applications profiled with MAP
  - With Intel compilers, aggressive optimizations can interfere with MAP. To prevent this use the following flags: **-fno-omit-frame-pointer -no-ip -no-ipo**

$ make bt-mz CLASS=C NPROCS=8

- Edit the job script job.sub and submit

```
map --profile srun -n $PROCS $EXE
```

  - **--profile** enables to run the profiler in non-interactive mode

```
$ sbatch -A <youraccount> job.sub
```

- View the results

```
$ map bt-mz_C_8p_2n_6t_YYYY-MM-DD_HH-MM.map
```

MPI_Init_thread limitations

- BT-MZ uses MPI_Init_thread() rather than MPI_Init()

- MAP provides limited support for MPI_THREAD_SERIALIZED or MPI_THREAD_MULTIPLE
- A warning message will be displayed if that’s the case
- MPI activity on non-main threads won’t contribute to the MPI metric graphs.
- Additional profiling overhead may appear
- Pthread view is recommended to view the profiling results

- MPI_THREAD_SINGLE or MPI_THREAD_FUNNELED are fully supported

Run Arm DDT

- Edit the makefile and compile

`FFLAGS  = -O0 -g $(OPENMP)`

  - The debugging option (**-g**) is a requirement for all applications debugged with DDT
  - Disabling compiler optimizations **-O0** is recommended

$ make bt-mz CLASS=C NPROCS=8

- Launch the debugger from the login node

$ ddt

- Edit the job script job.sub and submit

```
ddt --connect srun -n $PROCS $EXE
$ sbatch -A <youraccount> job.sub
```

- Accept the incoming connection, click on Run and debug interactively

Arm Remote Client

- To avoid using X forwarding when using Forge, a client is available for Linux, MacOS and Windows

- Install the Arm Remote Client

  https://developer.arm.com/products/software-development-tools/hpc/downloads/download-arm-forge

- Connect to the cluster with the remote client
  - Open your Remote Client
  - Create a new connection: Remote Launch ➔ Configure ➔ Add
    - Hostname: `<username>@jureca.fz-juelich.de`
    - Remote installation directory:

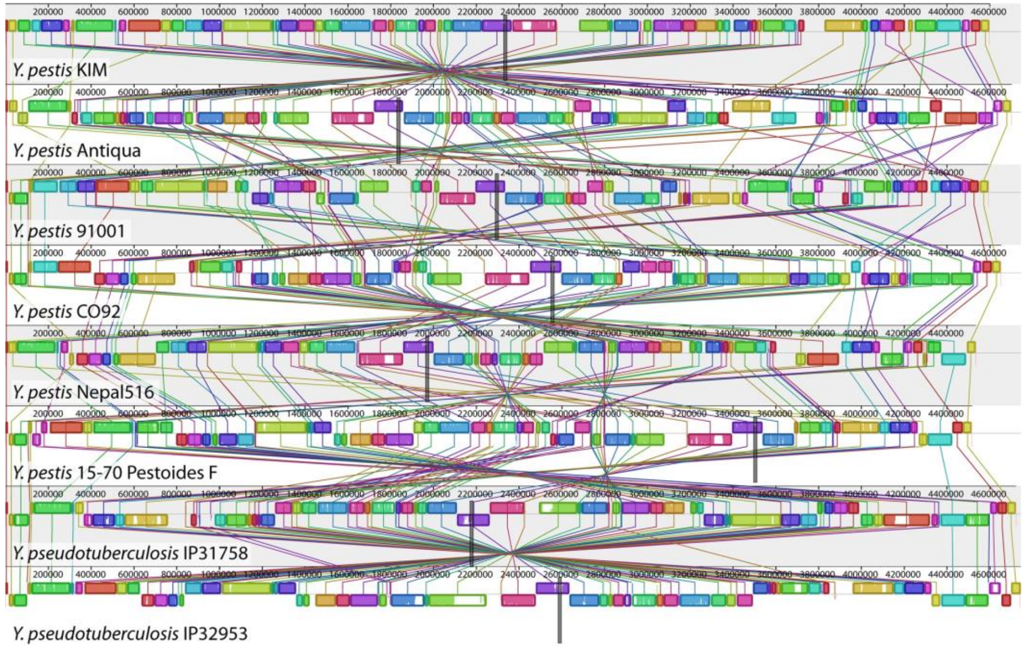
      `/p/scratch/share/VI-HPS/JURECA/packages/arm-forge-19.1/`
  - Connect!

Profile multithreaded codes

Genomics use case: DISCOVAR

Myth: genomic applications are I/O intensive

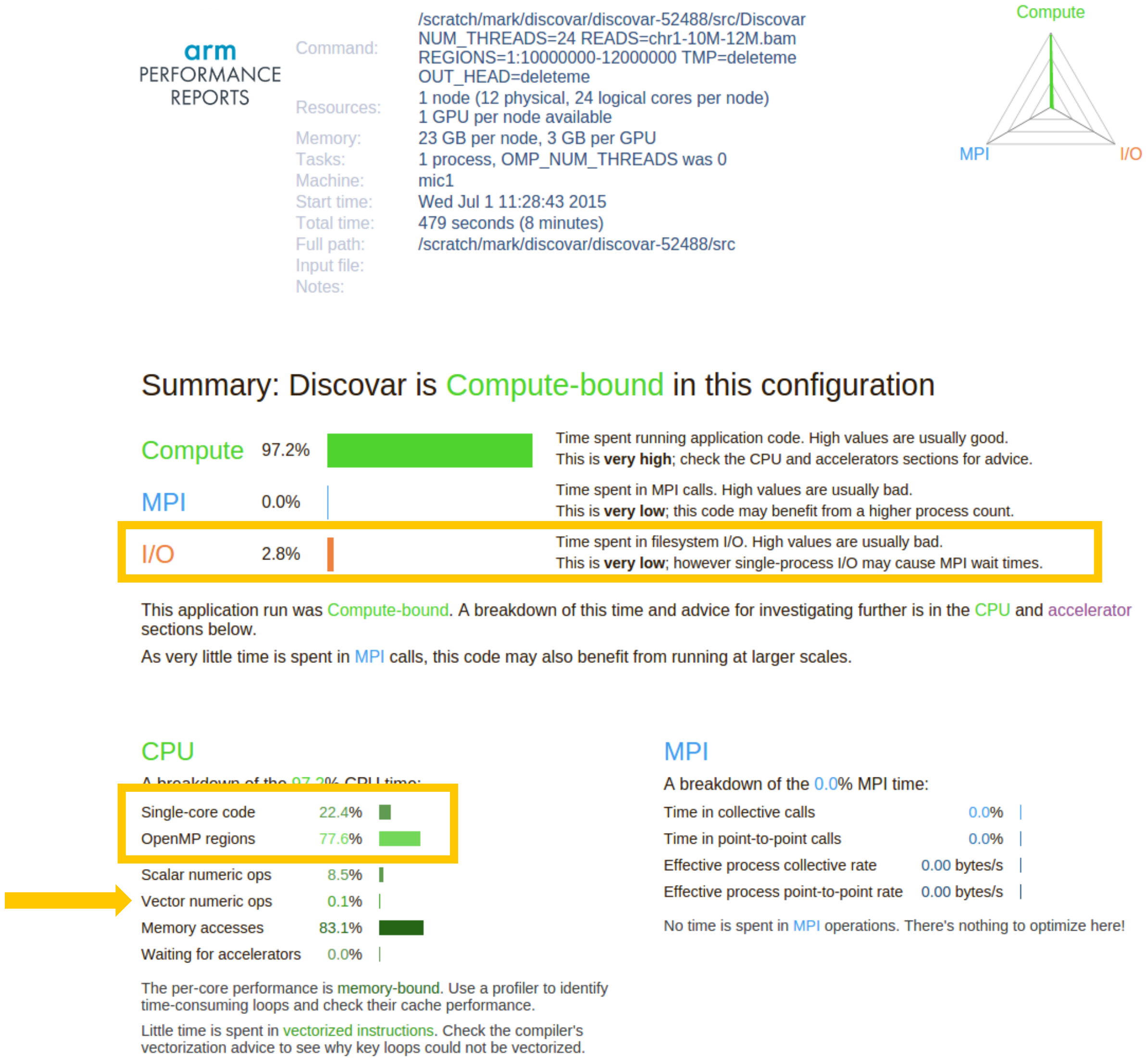
- Identifying DNA sequence variants helps understanding the genetic basis of many diseases (e.g. cancer) in order to develop:
  - New diagnosis
  - New therapies
- DISCOVAR
  - Variant caller and small genome assembler
  - Input: DNA sequencing files of sub-mammalian sized genomes
  - Newer DISCOVAR de novo for larger genomes
- C++ and OpenMP
- Developed by Broad Institute at MIT



A first look

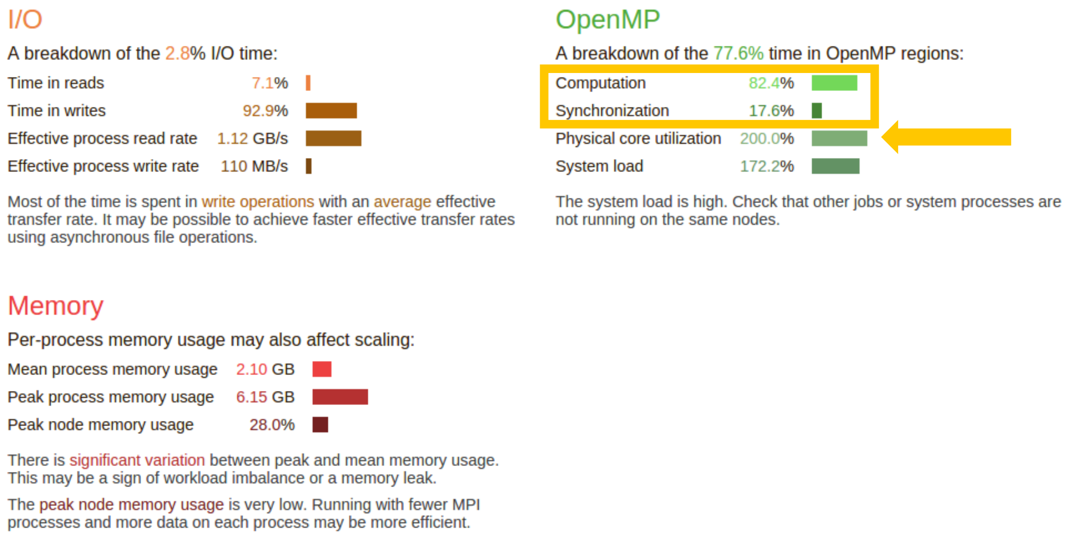
On real hardware

- It’s **not** I/O intensive
- Good quantity of OpenMP time
- No vectorization



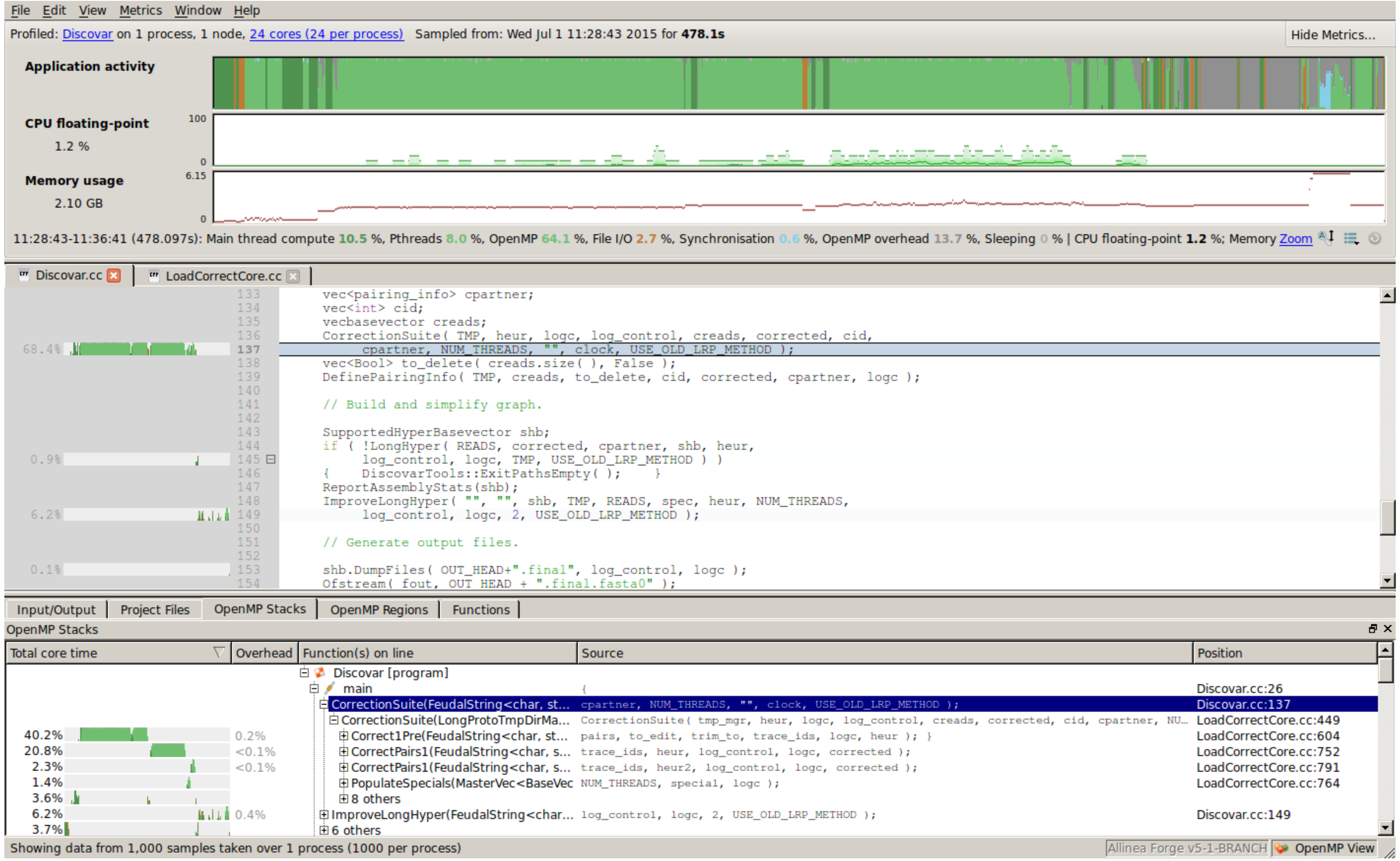
OpenMP in detail

- Physical cores are 200% loaded
  - Hyper-threading is on
- 17% of parallel region time is synchronization



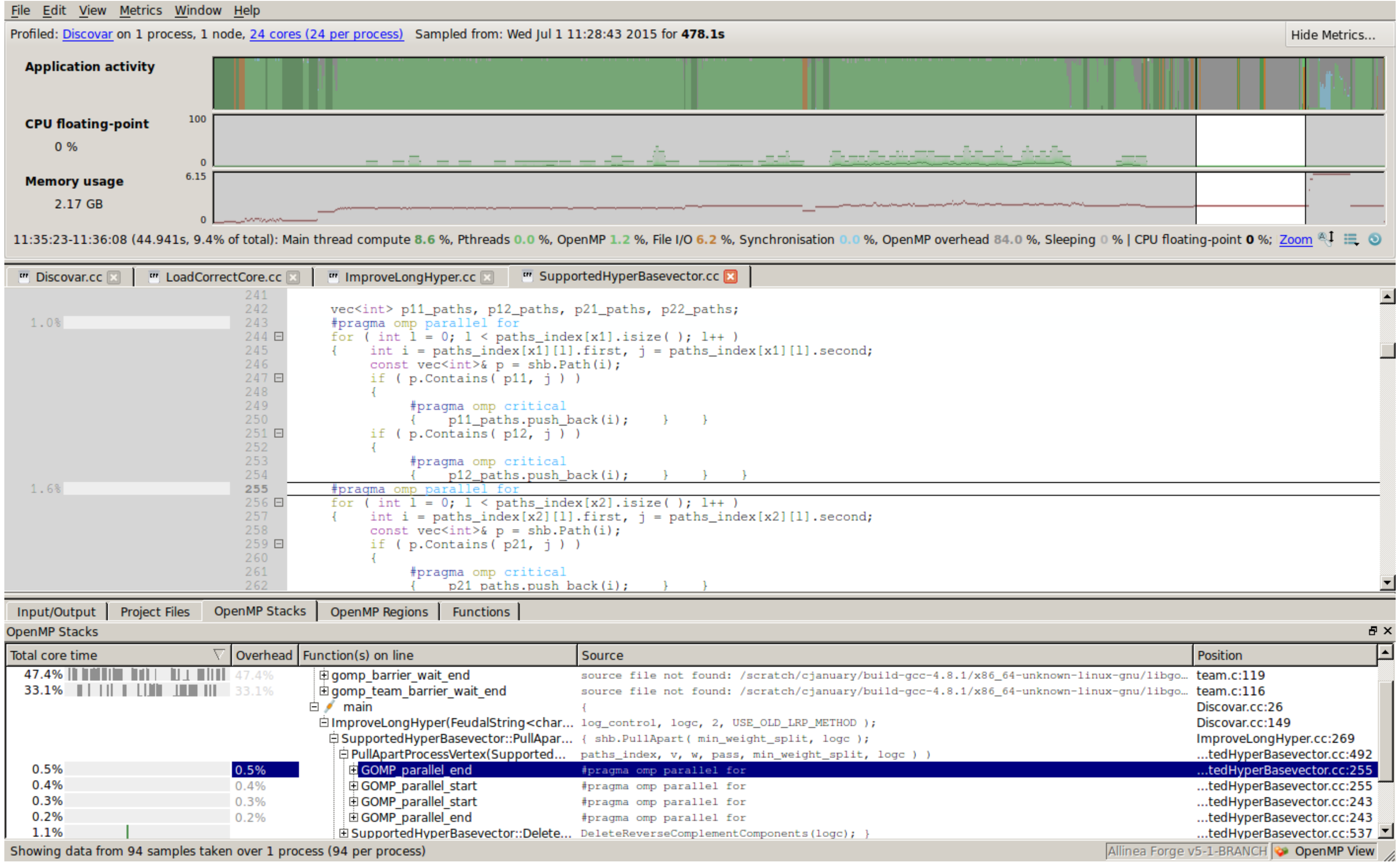
DISCOVAR – Investigating the OpenMP synchronization

- Horizontal time axis: colour coded
  - Dark green – single core
  - Light green – OpenMP work
  - Light blue – Pthread sync
  - Grey – idle
- Vertical axis
  - #cores doing something
- Something's very wrong towards the end…



Zoom in on the region

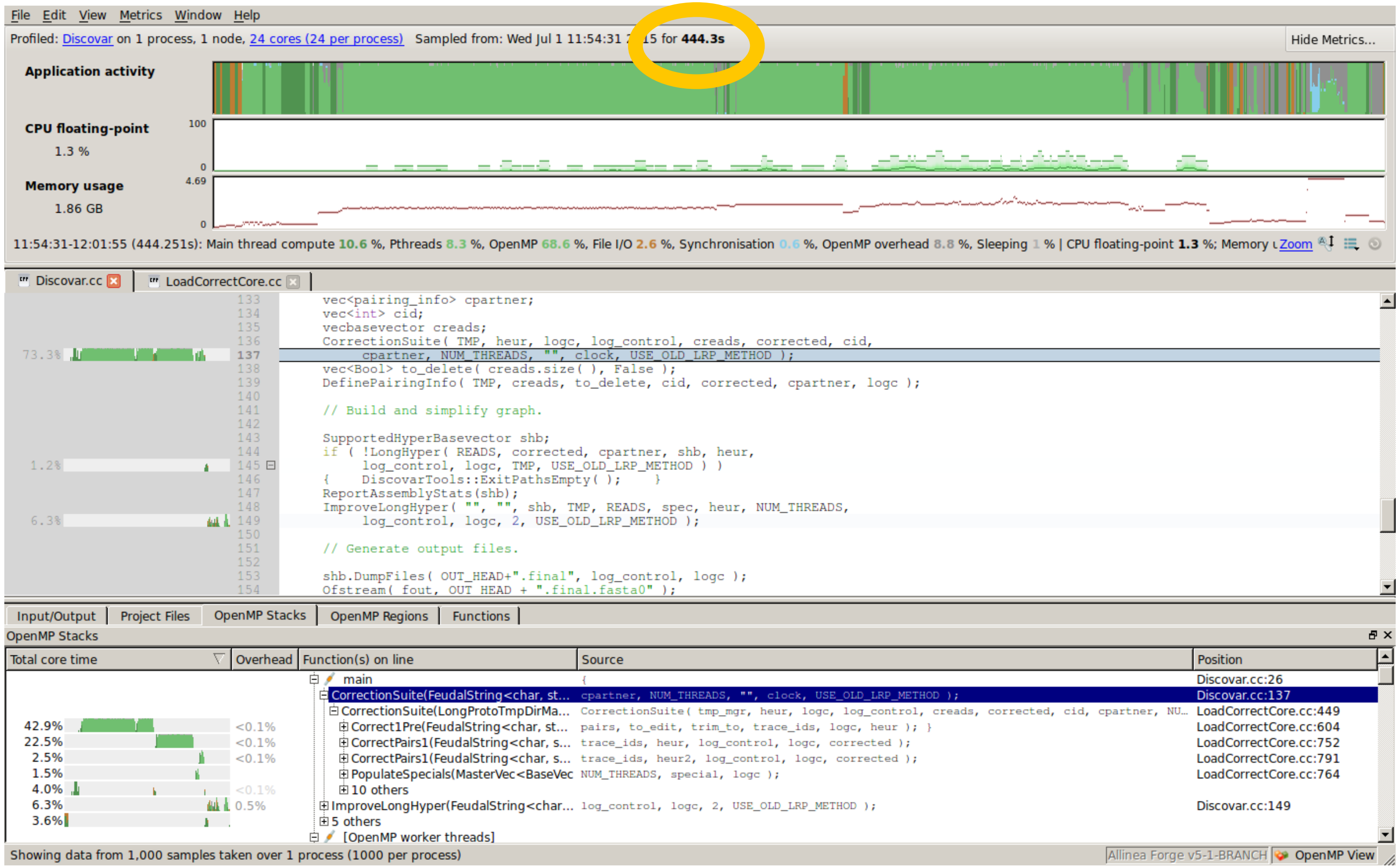
- Arm MAP lets you zoom
  - Stacks, code, regions, time all focused on zoom area
- Key observation:
  - OpenMP “critical” region



Fixing

- `#pragma omp critical`
  - Execute exactly one thread at a time to ensure safety
- is costing too much
  - Passing “token” from thread to thread to do small pieces of work.
- Run whole section on one thread instead
  - Has same semantics

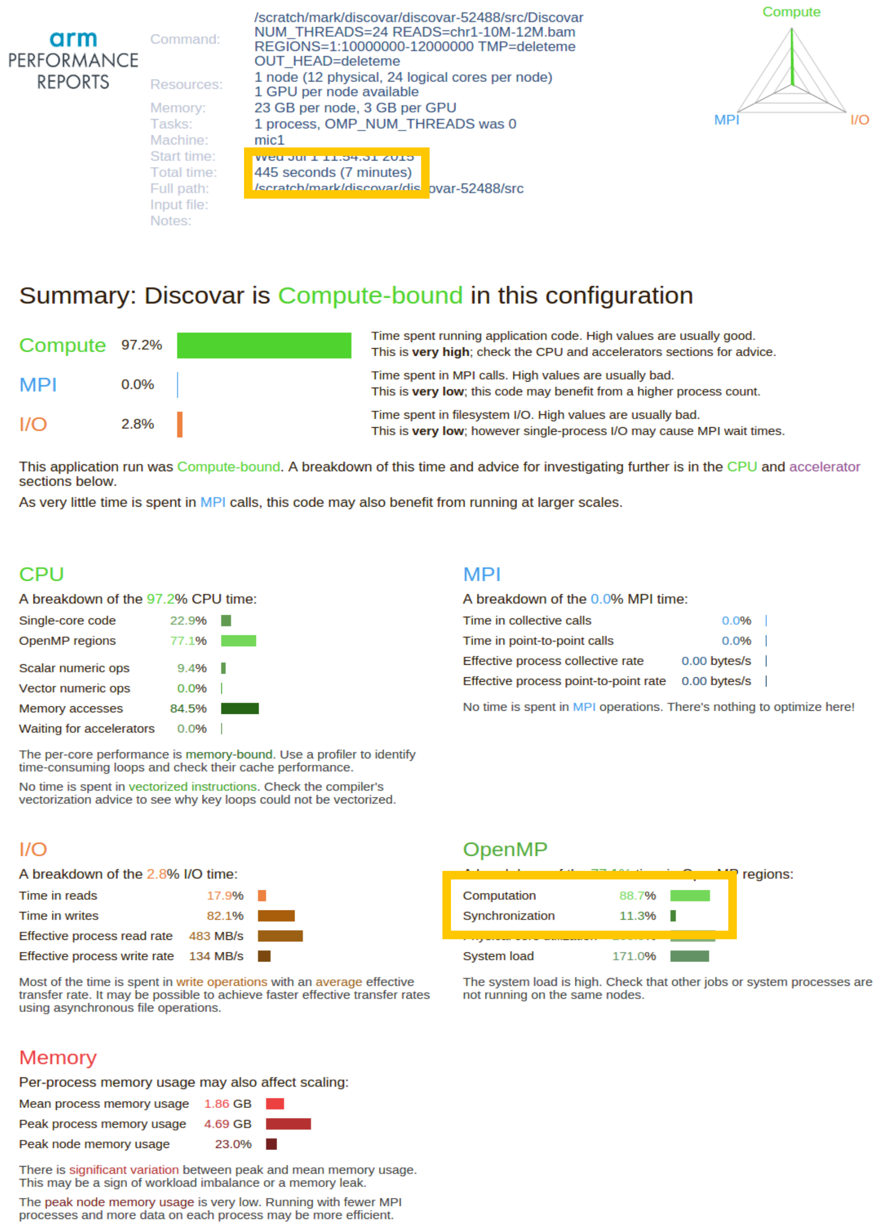
Impact of change



- Runtime down by 7%

As a performance report

- Improvements in
  - Runtime
  - Synchronization overhead



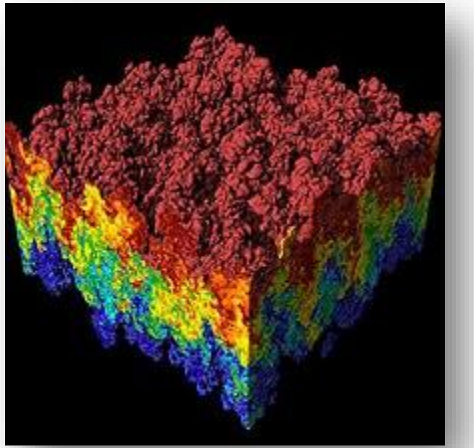
Optimize Python-based applications

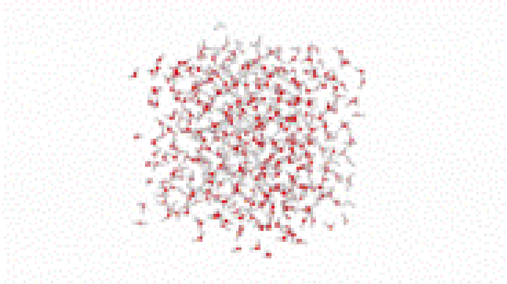
Python in HPC

- Essential modules:
  - **NumPy**: support of large multi-dimensional arrays and matrices
  - **SciPy**: support for linear algebra, integration, interpolation, FFT, …
  - **MPI4Py**: provides bindings of the MPI standard

- Rely on highly-optimized libraries
  - Written in lower-level languages:
    - C, FORTRAN, …
  - BLAS, LAPACK, FFTW, …

- Can easily be interfaced with other languages





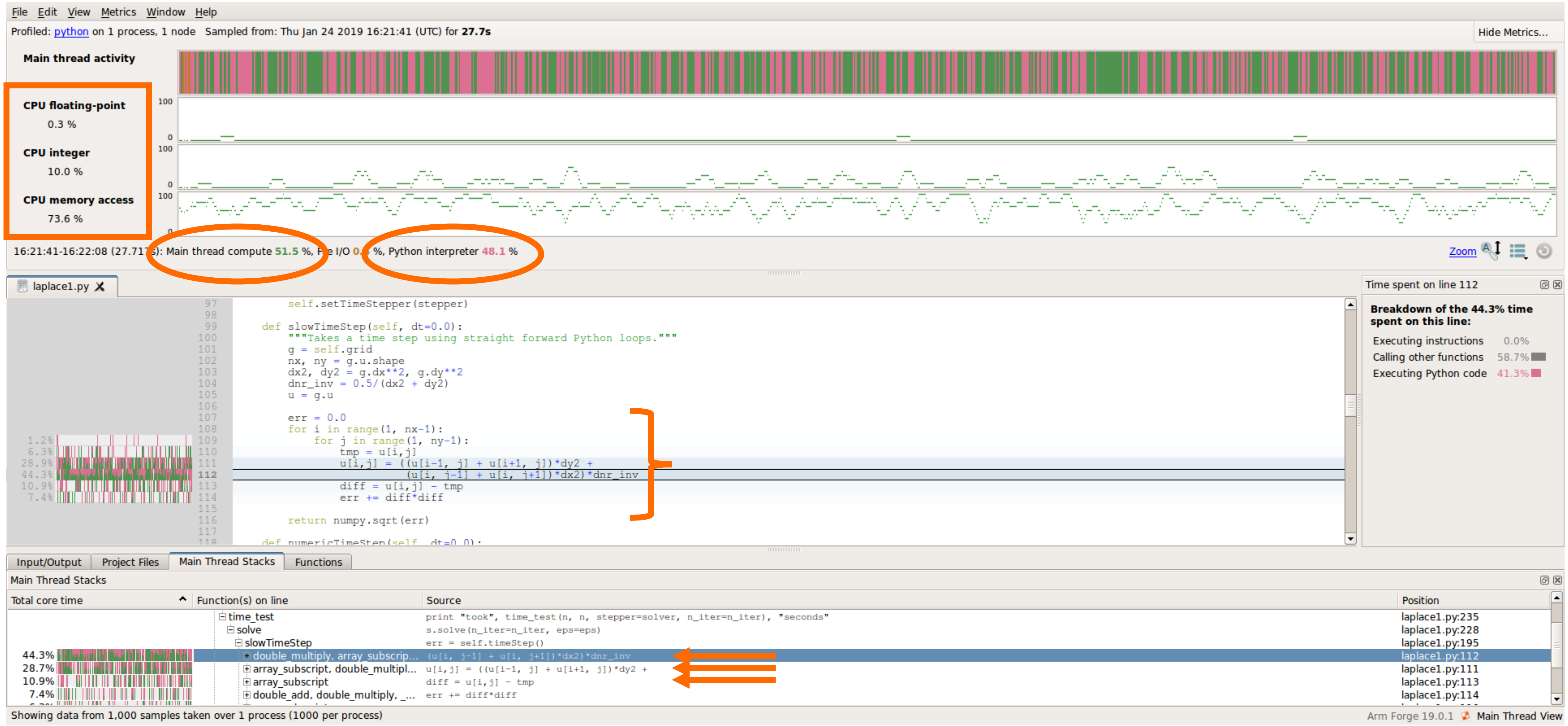
Use Arm MAP on Python applications

- Launch command
  - $ **python** ./laplace1.py slow 100 100
- Profiling command
  - $ **map --profile python** ./laplace1.py slow 100 100
  - --profile: non-interactive mode
  - --output: name of output file
- Display profiling results
  - $ **map** laplace1.map

Laplace1.py

```
[…]
err = 0.0
for i in range(1, nx-1):
  for j in range(1, ny-1):
    tmp = u[i,j]
    u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
      (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
    diff = u[i,j] - tmp
    err += diff*diff
return numpy.sqrt(err)
[…]
```

Naïve Python loop



Optimizing computation on NumPy arrays

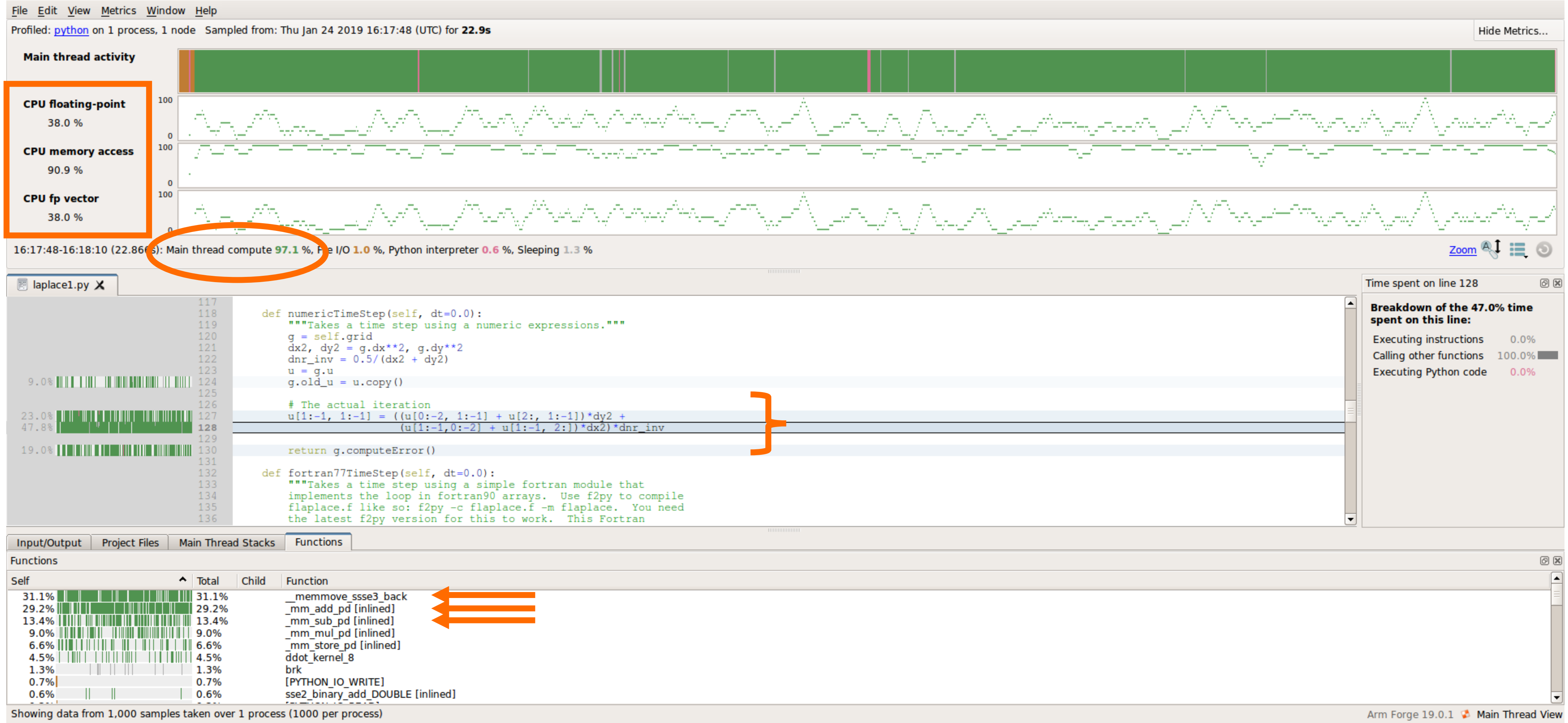
Naïve Python loop

```
err = 0.0
for i in range(1, nx-1):
  for j in range(1, ny-1):
    tmp = u[i,j]
    u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
    (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
    diff = u[i,j] - tmp
    err += diff*diff
return numpy.sqrt(err)
```

NumPy loop

```
u[1:-1, 1:-1] =
    ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

return g.computeError()
```

NumPy array notation



Use FORTRAN in Python applications

- F2PY: FORTRAN to Python interface generator

- Part of NumPy

- Compile with debugging flag for profiling
  - $ f2py **--debug** -c flaplace90_arrays.f90 -m flaplace90_arrays
  - Relies on underlying compiler: GCC, IFORT, PGI
  - Generates a *.so library imported in the Python script:

    ```
    import flaplace90_arrays
    ```
  - --debug: enables debug information

Use FORTRAN in Python applications

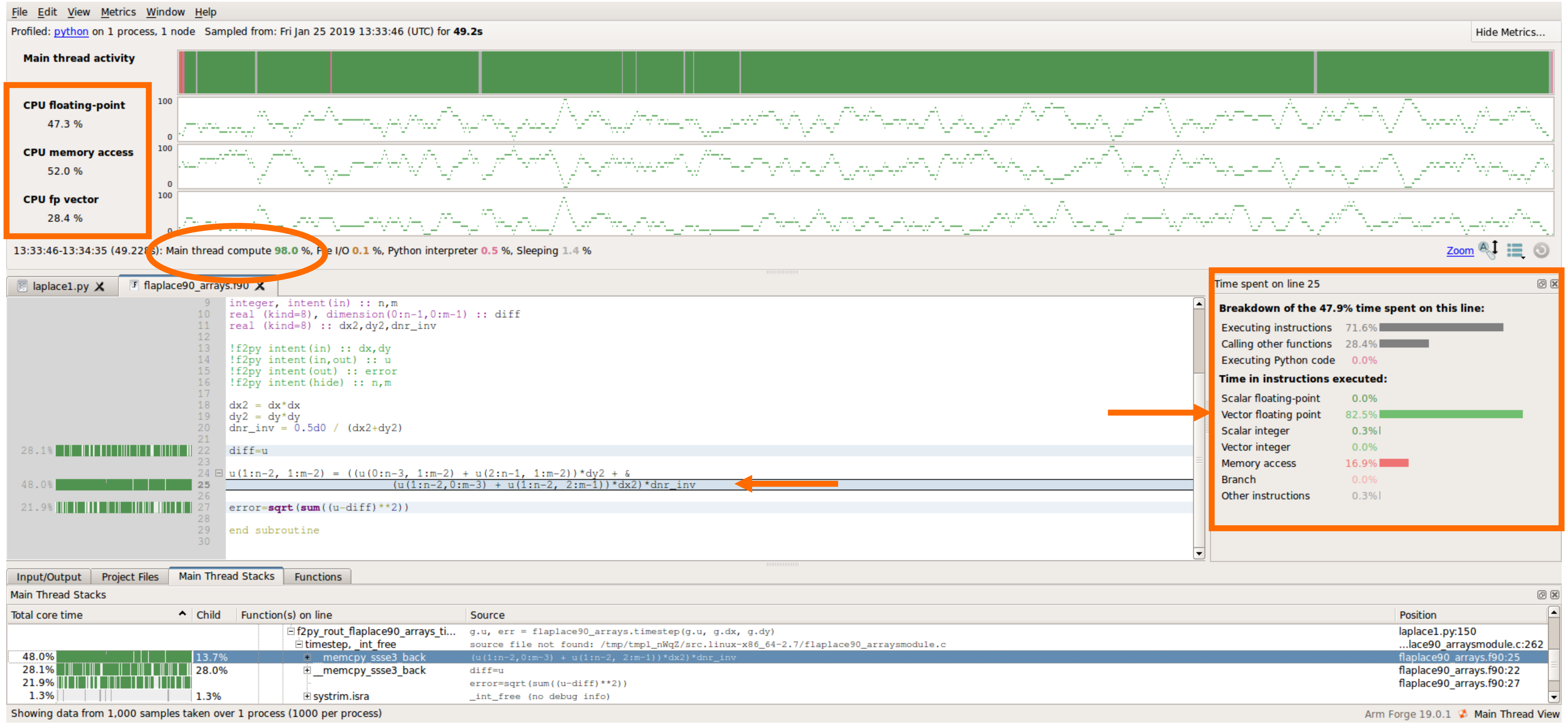
FORTRAN loop

```
subroutine timestep(u,n,m,dx,dy,error)
[…]
!f2py intent(in) :: dx,dy
!f2py intent(in,out) :: u
!f2py intent(out) :: error
!f2py intent(hide) :: n,m
[…]

u(1:n-2, 1:m-2)=((u(0:n-3, 1:m-2) + u(2:n-1, 1:m-2))*dy2 + &
       (u(1:n-2,0:m-3) + u(1:n-2, 2:m-1))*dx2)*dnr_inv

error=sqrt(sum((u-diff)**2))
end subroutine
```

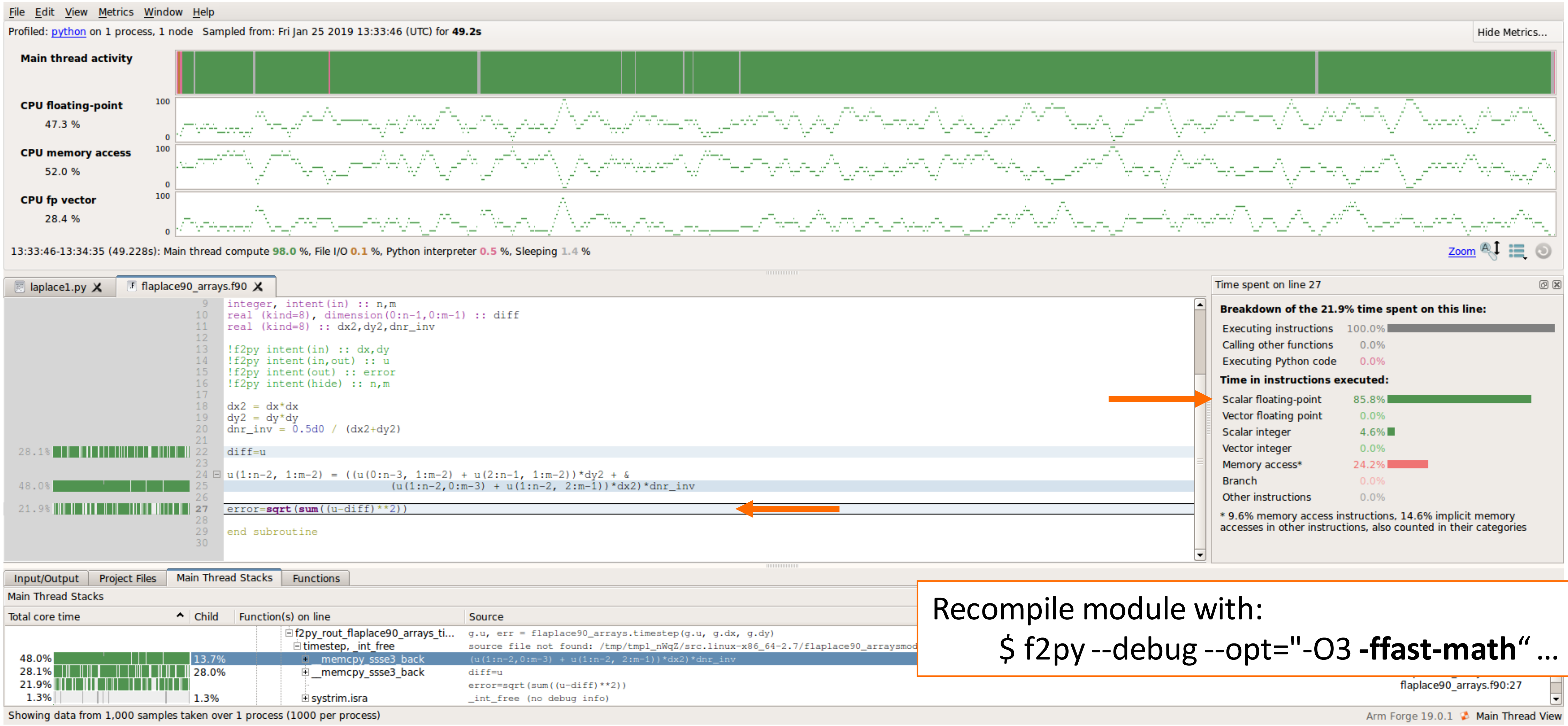
Python script

```
import flaplace90_arrays

[…]

def fortran90TimeStep(self, dt=0.0):
  g = self.grid
  g.u,err =
    flaplace90_arrays.timestep(g.u, g.dx, g.dy)
  return err

[…]
```

FORTRAN code



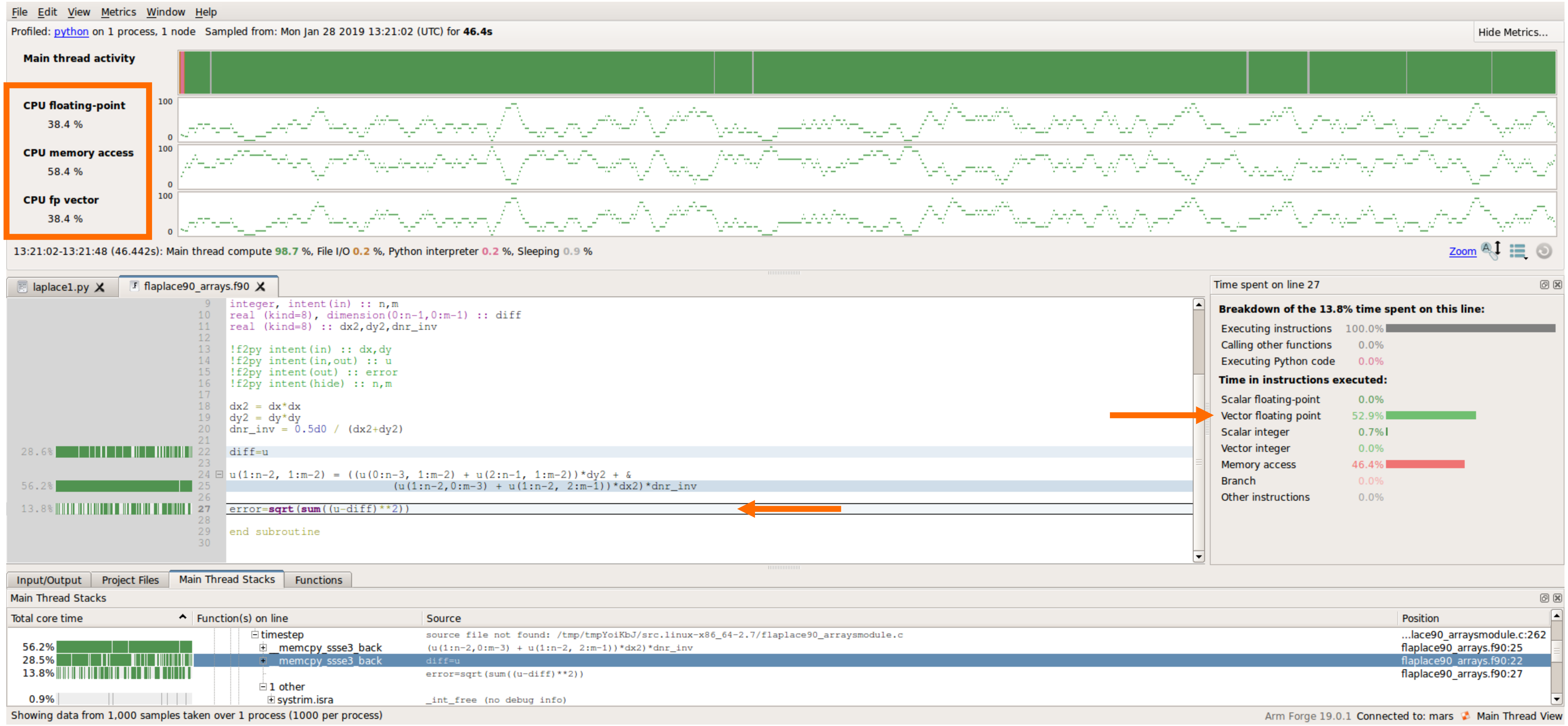
FORTRAN code



Recompile module with:

$ f2py --debug --opt="-O3 **-ffast-math**" ...

FORTRAN code fast-math



Multi-processing in Python

- MPI4Py
  - Provides Python bindings of the MPI standard
  - MPI: Message Passing Interface

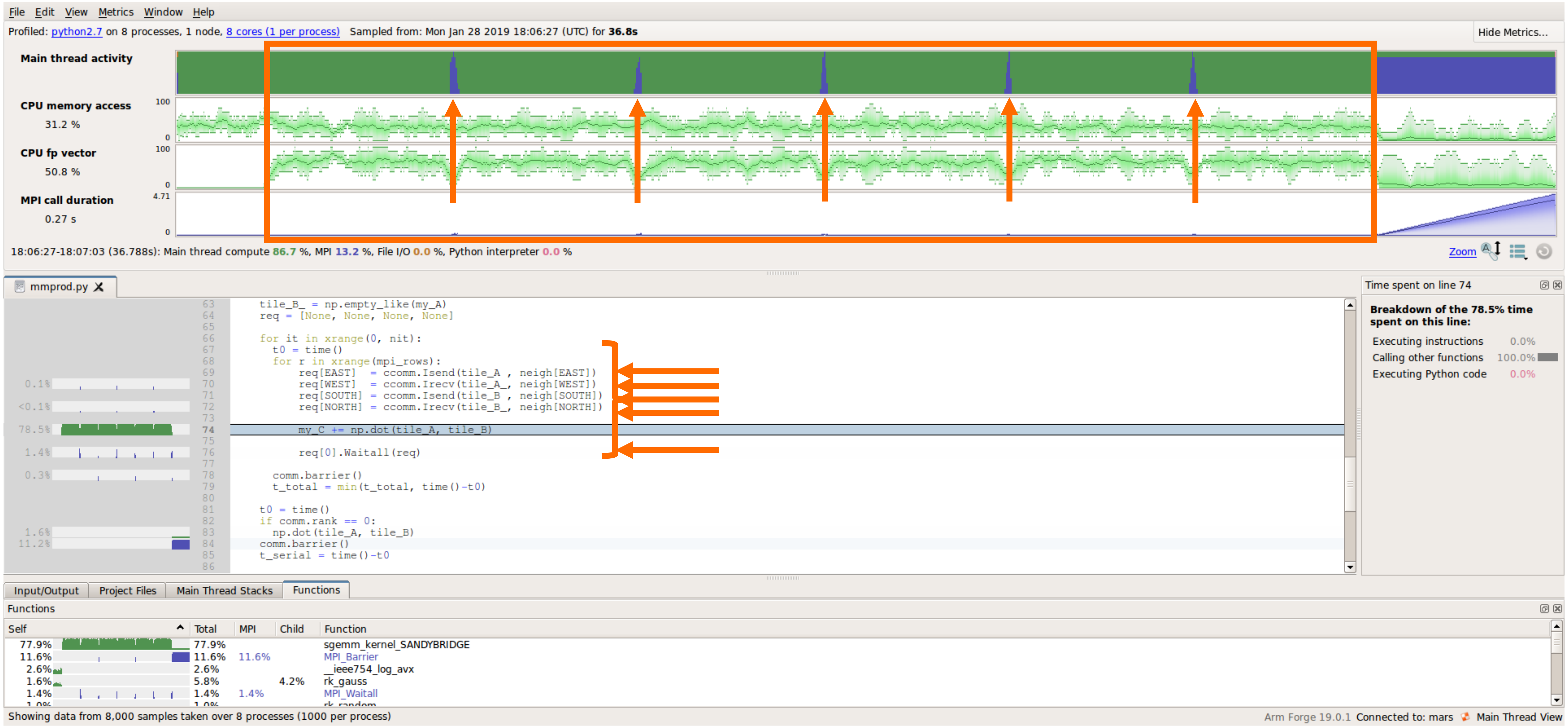
- Rely on existing MPI infrastructure
  - MPICC must be in path when installing module
  - MPIRUN enables to launch the application

- Profiling command
  - $ **map --profile mpirun -n 8 python** ./mmprod.py

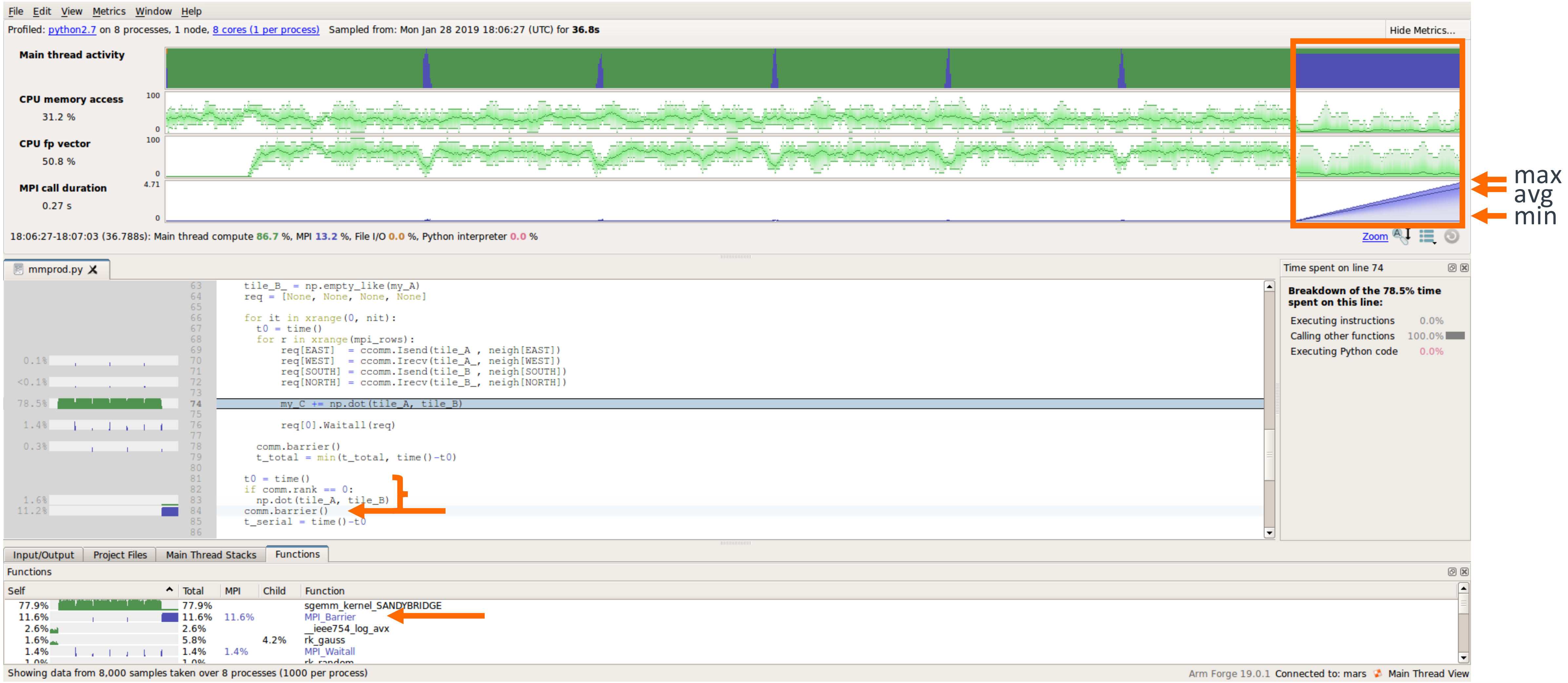
MPI4Py example

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source=0)
    print('On process 1, data is ',data)
```

MPI-parallel matrix multiplication

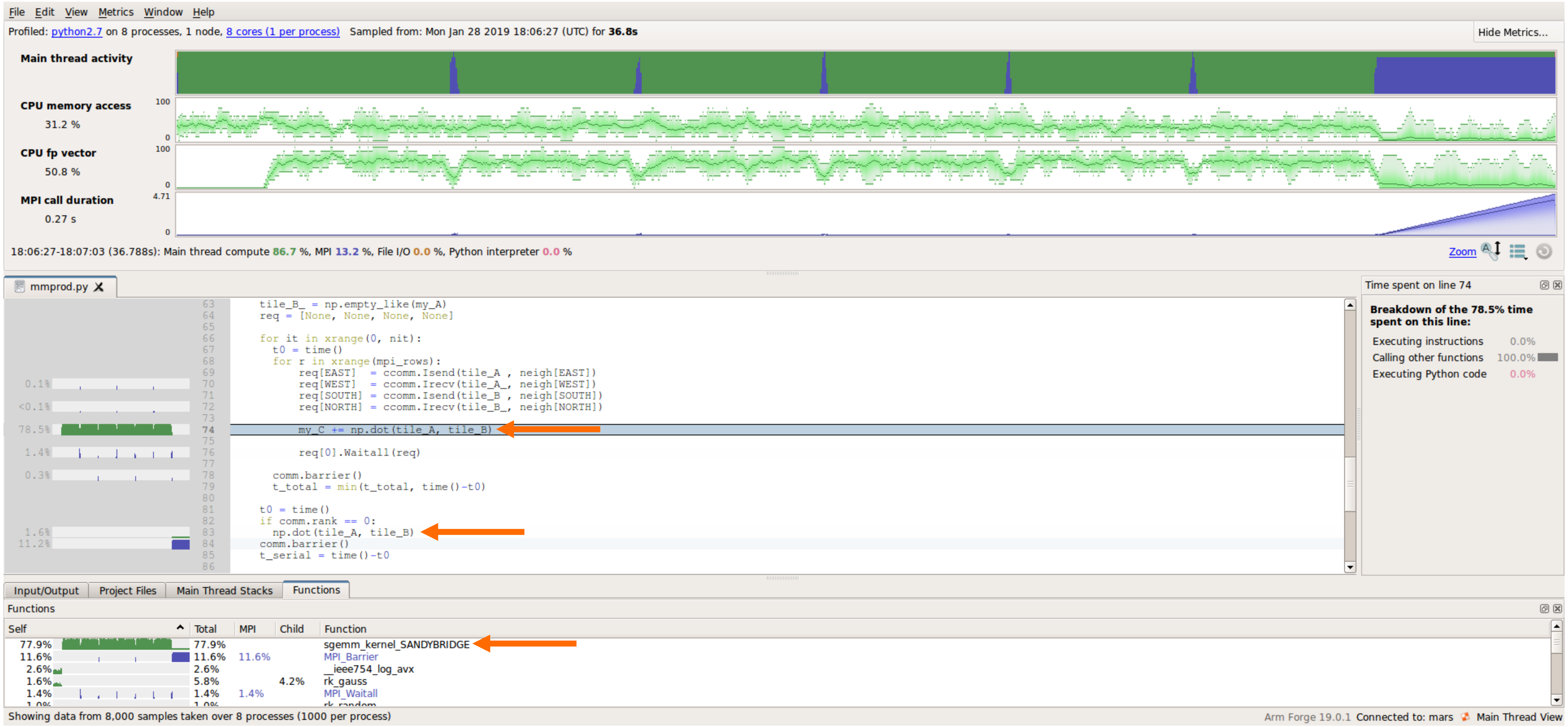


MPI-parallel matrix multiplication

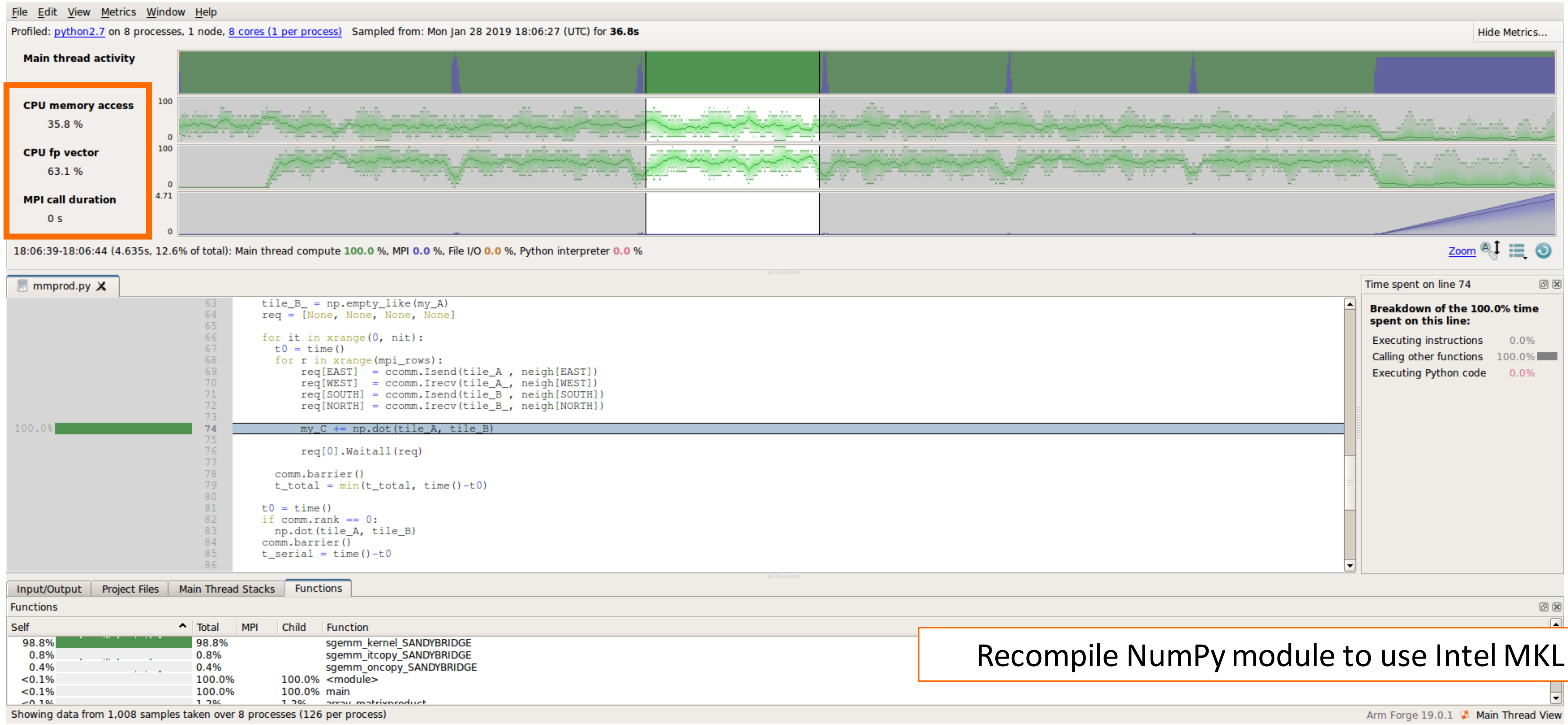


arm

MPI-parallel matrix multiplication

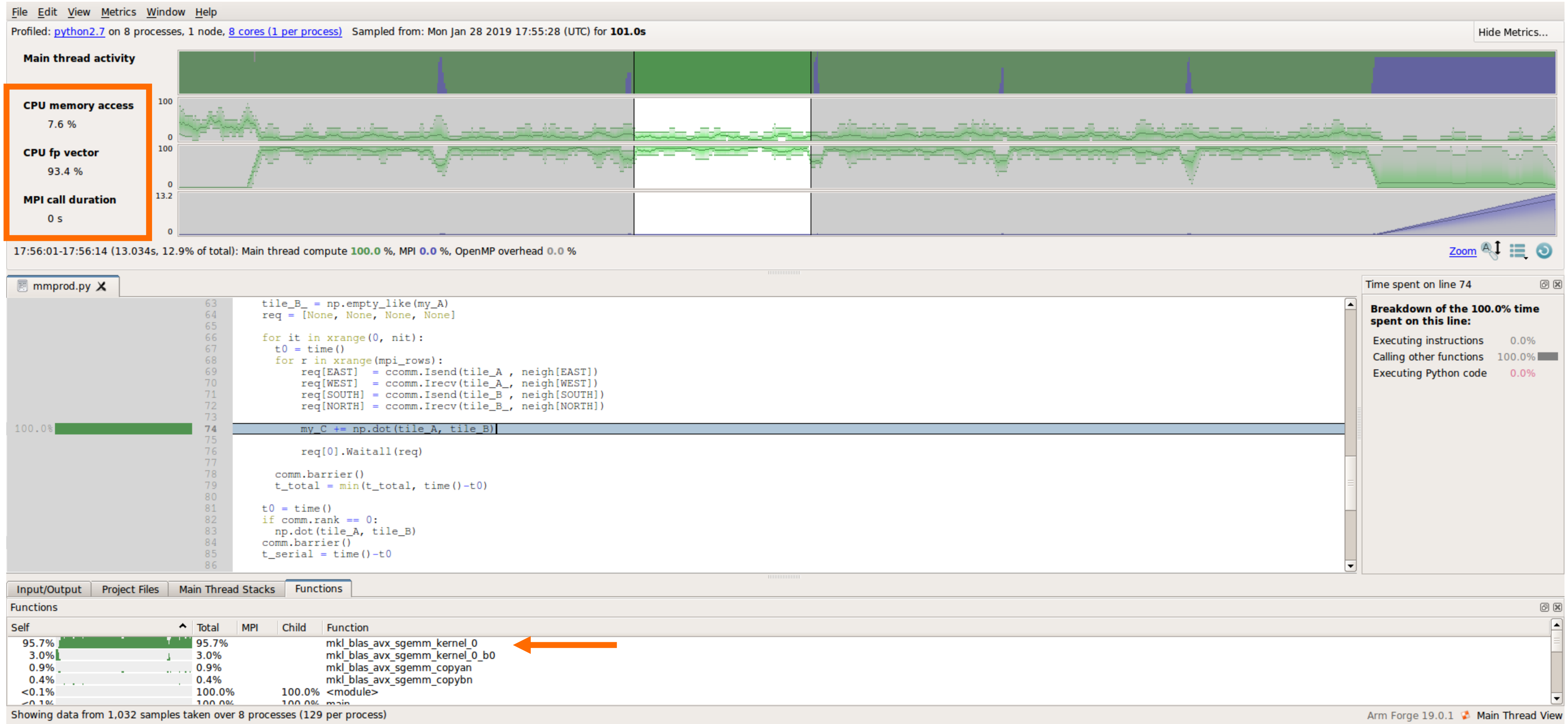


MPI-parallel matrix multiplication: OpenBLAS



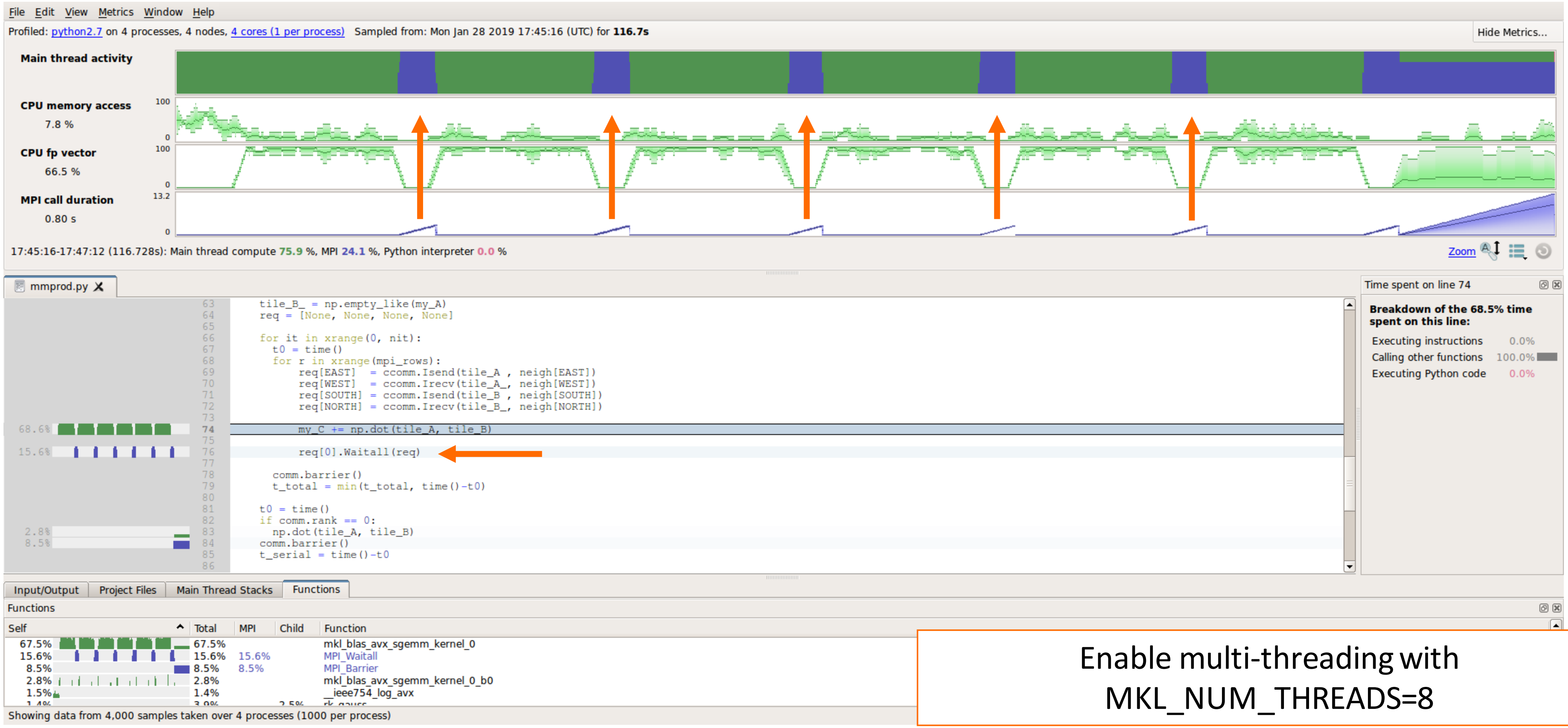
Recompile NumPy module to use Intel MKL

MPI-parallel matrix multiplication: MKL



arm

Multi-node matrix multiplication: MKL



Enable multi-threading with MKL_NUM_THREADS=8

Hybrid OMP/MPI matrix multiplication

Multithreaded/OpenMP MKL

Visualize code regions with Caliper

Region profiling with Caliper by LLNL

Retrieve contextual information as part of your performance profiles

Caliper in brief

Provider: LLNL

What: General purpose Application Introspection System

For who: Developers working with complex workflows & applications (e.g. combination of packages, solvers, libraries)

Why: Get contextual information of an application performance

Instrument with Caliper API

```
CALI_MARK_LOOP_BEGIN(riemann_slice_id);

#compute pressure, density, velocity for each slice

for(s=0; s<slices; s++) {
  CALI_MARK_ITERATION_BEGIN(riemann_slice_id, s);
  CALI_MARK_BEGIN("riemann_slice_precompute");
  for(i=0; i<narray; i++) { […] }
  CALI_MARK_END("riemann_slice_precompute");
  […]
  CALI_MARK_BEGIN("riemann_slice_arrays");
  for(i=0; i<narray; i++) { […] }
  CALI_MARK_END("riemann_slice_arrays");
  CALI_MARK_ITERATION_END(riemann_slice_id, s);
}

CALI_MARK_LOOP_END(riemann_slice_id);
```

Get contextual information

Path	Inclusive time (usec)	Exclusive time (usec)	Time (%)
updateConservativeVars	1637063.000000	1637063.000000	7.955088
riemann	8586175.000000	1317.000000	0.006400
riemann_slices – pressure	8584307.000000	1190957.000000	5.787295
riemann_slice_arrays	2907784.000000	2907784.000000	14.129986
riemann_slice_interfaces	2873562.000000	2873562.000000	13.963688
riemann_slice_precompute	1612004.000000	1612004.000000	7.833317
qleftright	1787885.000000	1787885.000000	8.687987
trace	2218274.000000	2218274.000000	10.779404
slope	1037166.000000	1037166.000000	5.039969
constoprim	1195203.000000	1195203.000000	5.807928
gatherConservativeVars	1559916.000000	1559916.000000	7.580202

For more information about Caliper…

- Project landing page https://computation.llnl.gov/projects/caliper
- Download https://github.com/LLNL/Caliper
- Documentation https://llnl.github.io/Caliper/
- Tutorial https://computation.llnl.gov/sites/default/files/public/Caliper%20vi-hps%20tutorial.pdf

Performance Analysis with MAP, part of Forge

By default, provides the "computer science" view of an application performance

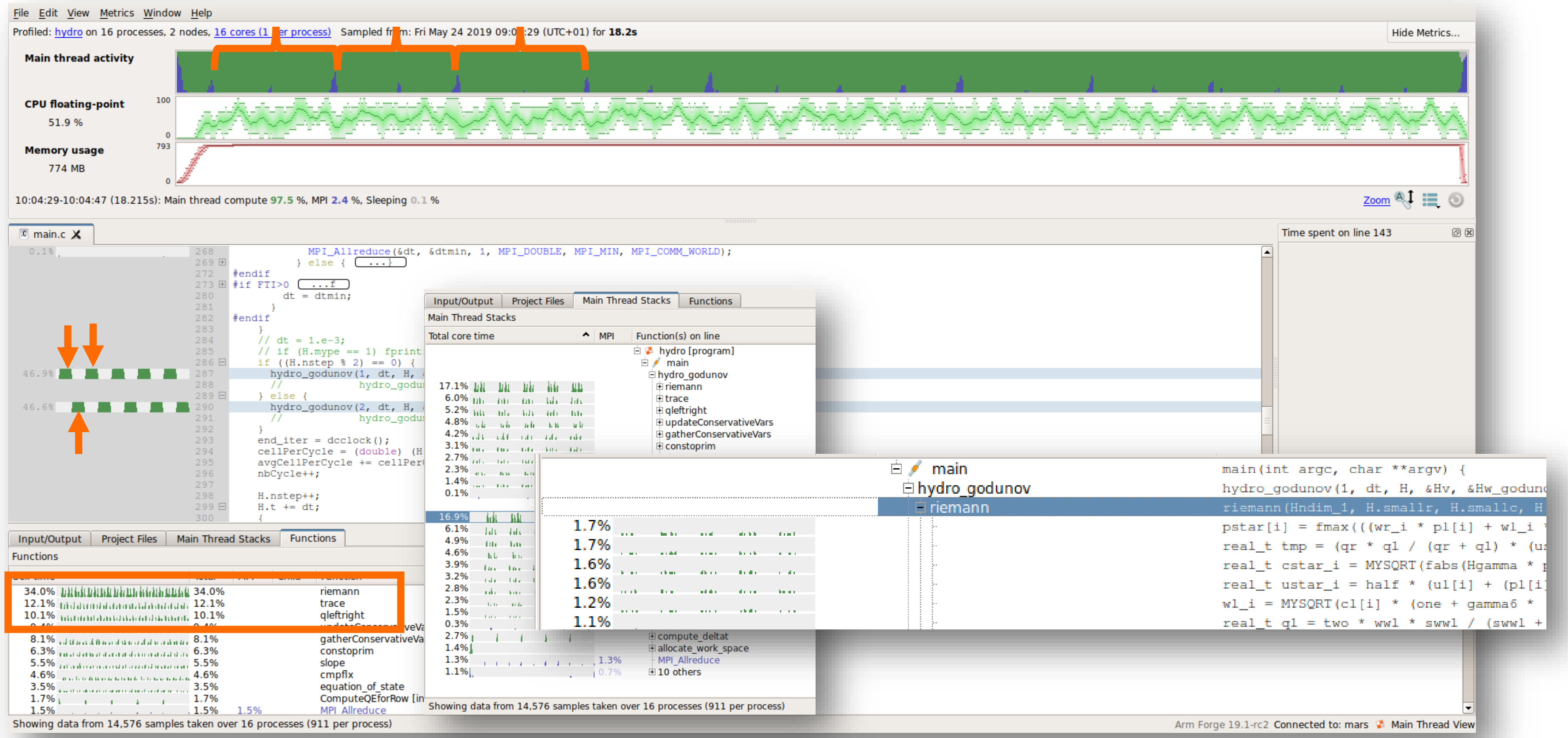
MAP in brief

Provider: Arm

What: Lightweight, highly scalable profiler for HPC applications on any hardware

For who: Developers of all level looking to improve the performance of their C/C++ and Fortran codes

Why: Extract the last drop of performance by identifying & diagnosing a wide range of bottlenecks (e.g. network, CPU, IO, etc.)



Caliper regions support in MAP

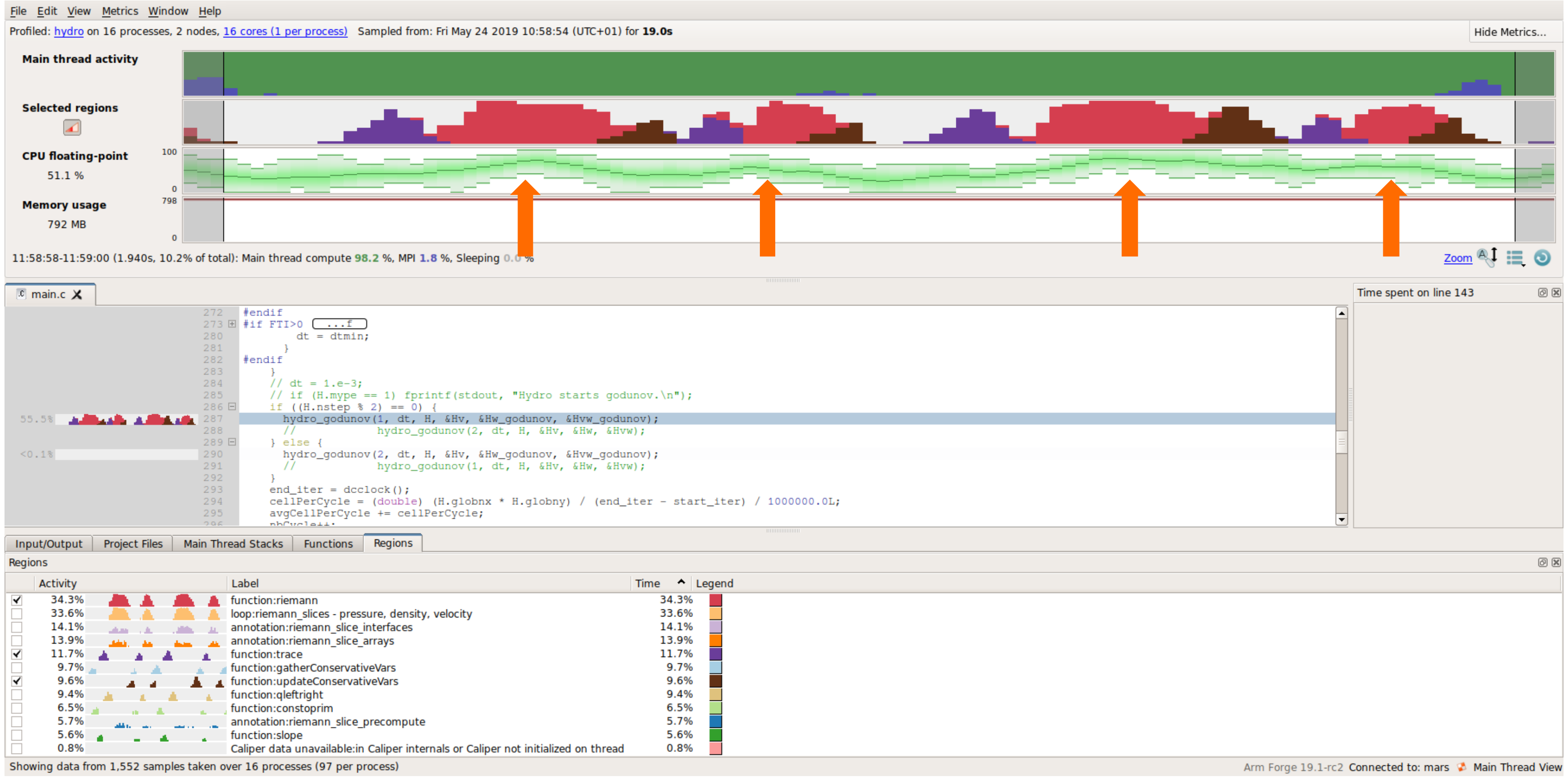
Combining contextual information with data collection in a slick GUI

MAP and Caliper

What: Collecting & presenting Caliper's data into MAP's GUI

- Correlates regions with performance metrics & data
- Associates regions with the timeline
- Ability to sort and filter by regions

Benefit: Makes it easy for users to understand what scientific phenomenon or stage in a workflow is slow and why



Caliper regions support in MAP

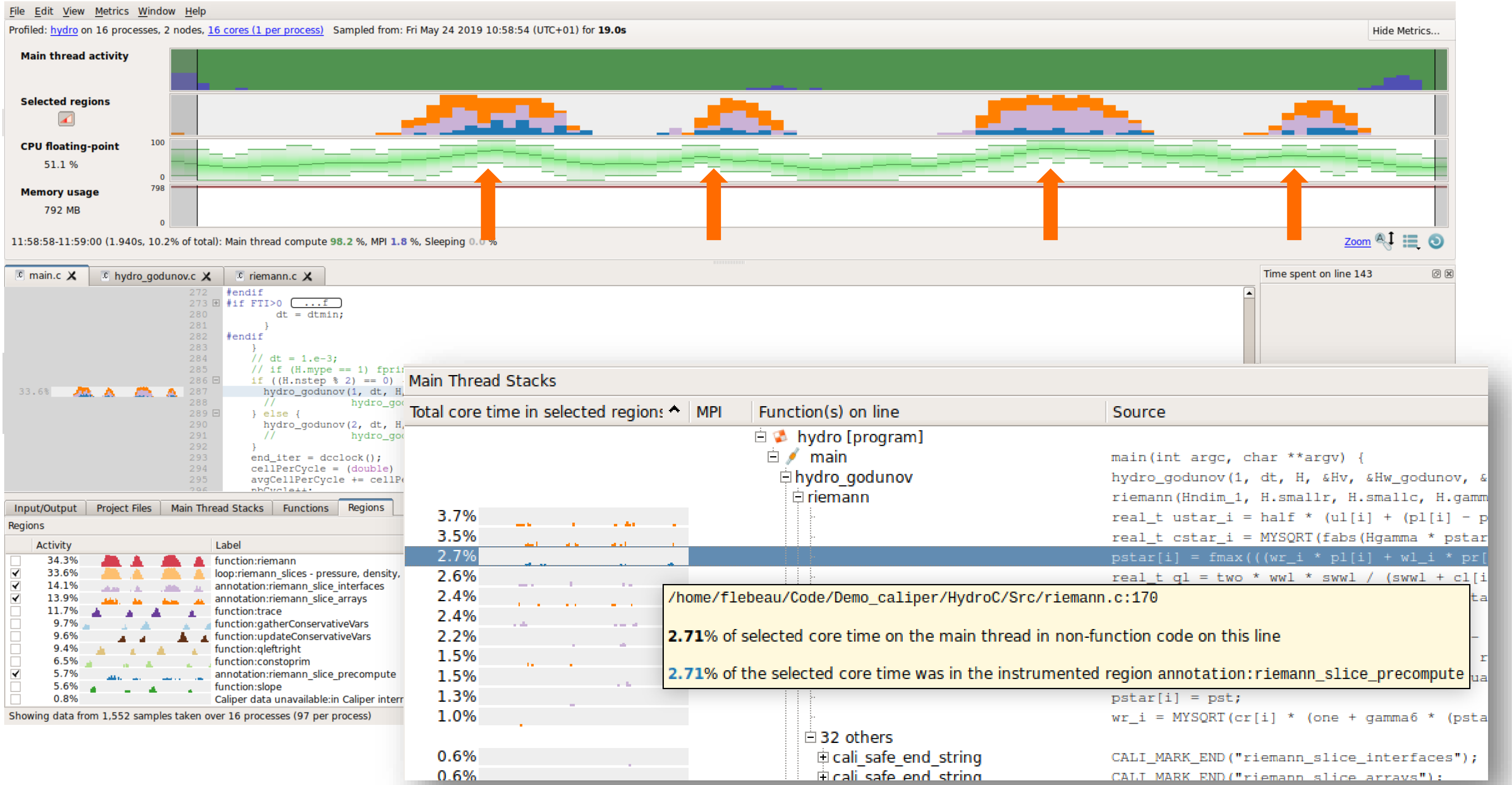
Combining contextual information with data collection in a slick GUI

MAP and Caliper

What: Collecting & presenting Caliper's data into MAP's GUI

- Correlates regions with performance metrics & data
- Associates regions with the timeline
- Ability to sort and filter by regions

Benefit: Makes it easy for users to understand what scientific phenomenon or stage in a workflow is slow and why



Summary: Arm provides…

Leadership in HPC software tools

- The only scalable cross-platform tool suite for HPC

Active development and investment

- Innovative, extremely scalable tools with unique visual insight

Rapid innovation and user focus

- Providing what science needs, first.

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

arm