arm

Debugging and Optimization Tools for Python HPC Codes

Beau Paisley<Beau.Paisley@arm.com>

RMACC 2021

Arm Forge == DDT + MAP + Performance Reports

An interoperable toolkit for debugging and performance engineering

Commercially supported by Arm

Fully Scalable

Very user-friendly

The de-facto standard for HPC development

- Most widely-used debugging and profiling suite in HPC
- Integration of debugger DDT, source code profiler MAP, and application profiler Performance Reports
- Fully supported by Arm on Intel, AMD, Arm, IBM Power, Nvidia GPUs, etc.
- Multithreaded and multi-process support for C/C++, Fortran and Python

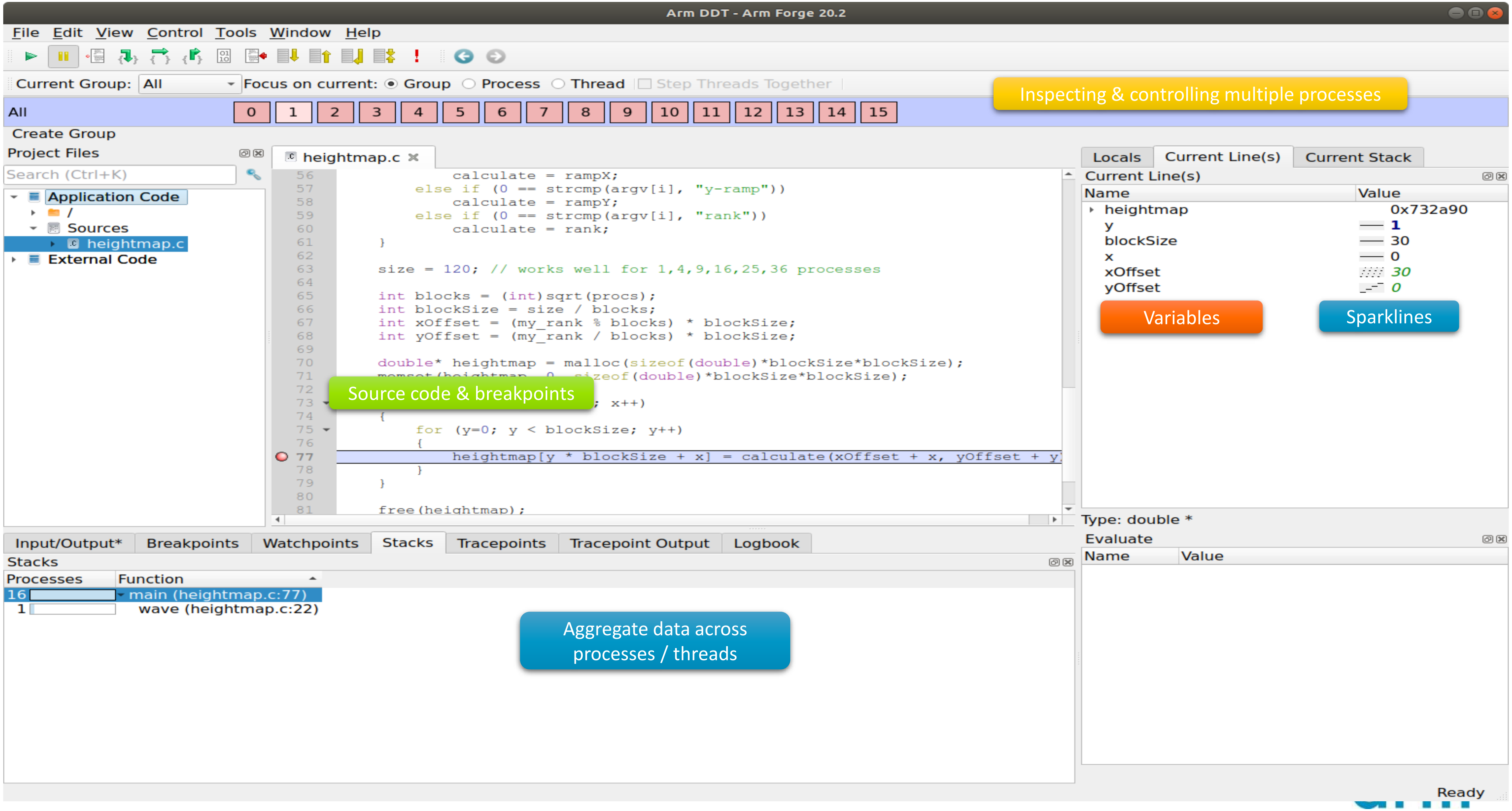
State-of-the art debugging and profiling capabilities

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

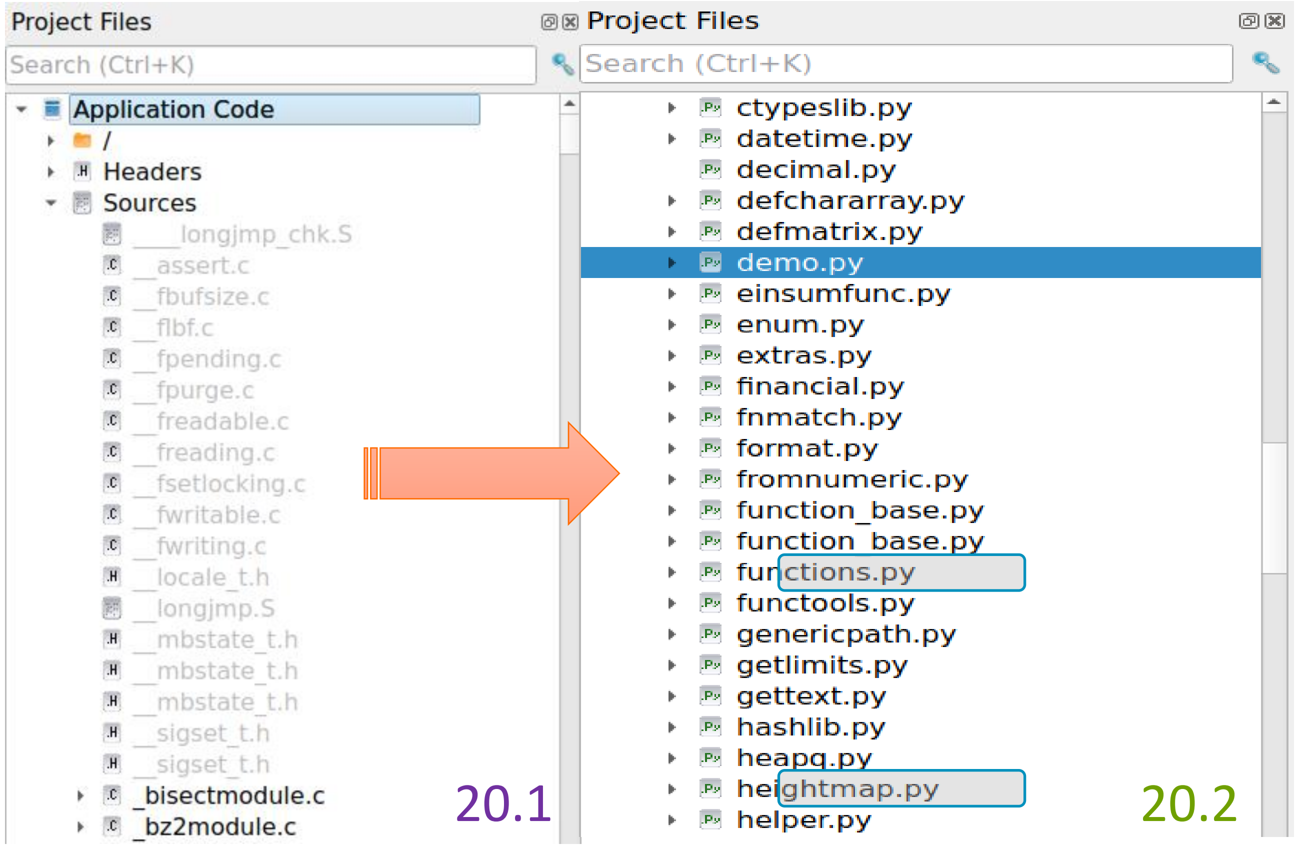
Easy to use by everyone

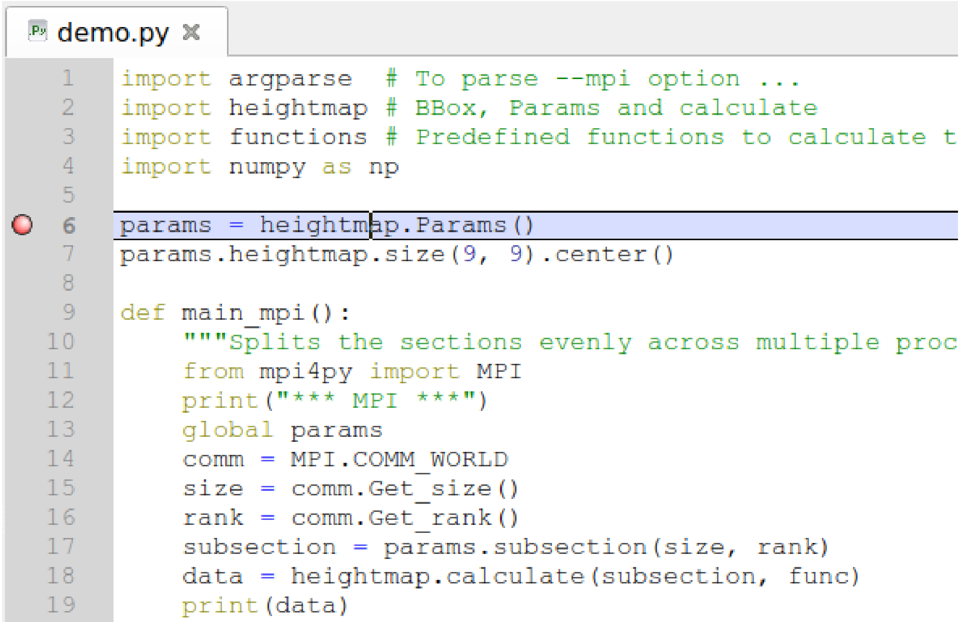
- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

The Scalable DDT GUI



Project Files





Inspecting Objects

New Python debugging features in DDT 20.2

Locals | Current Line(s) | Current Stack

Locals

Name	Value
MPI	<module at remote 0x7fffdc...
comm	<mpi4py.MPI.Intracomm at r...
size	3
rank	0
subsection	<Params(heightmap=<BBo...
data	<numpy.ndarray at remote ...

Evaluate

Name	Value
subsection._section.x	-4

20.1

Locals | Current Line(s) | Current Stack

Locals

Name	Value
MPI	module at remote 0x7fffdca...
comm	Intracomm
size	3
rank	0
subsection	heightmap.Params
_section	heightmap.BBox
h	9
w	3
x	-4
y	-4
heightmap	heightmap.BBox
data	numpy.ndarray at remote 0x...

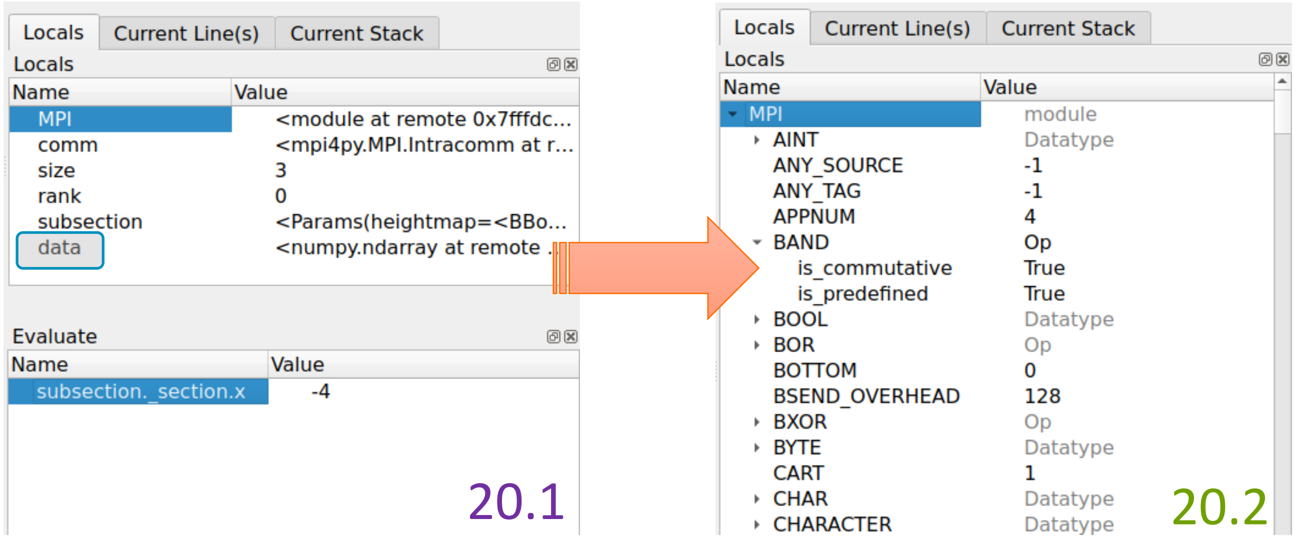
20.2

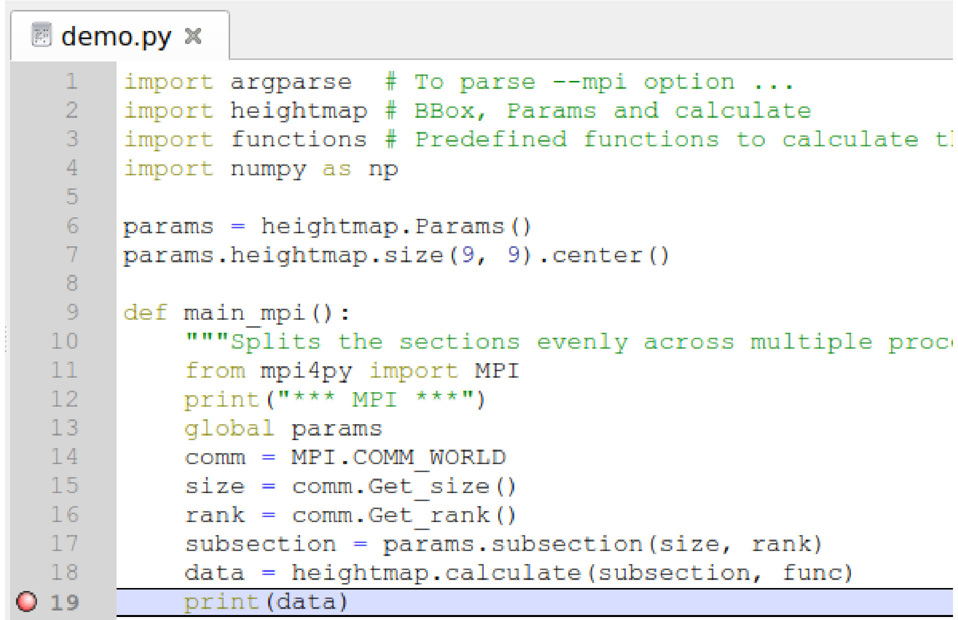
demo.py

```
import argparse  # To parse --mpi option ...
import heightmap # BBox, Params and calculate
import functions # Predefined functions to calculate t
import numpy as np

params = heightmap.Params()
params.heightmap.size(9, 9).center()

def main_mpi():
    """Splits the sections evenly across multiple proc
    from mpi4py import MPI
    print("*** MPI ***")
    global params
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()
    subsection = params.subsection(size, rank)
    data = heightmap.calculate(subsection, func)
    print(data)
```

Inspecting Modules

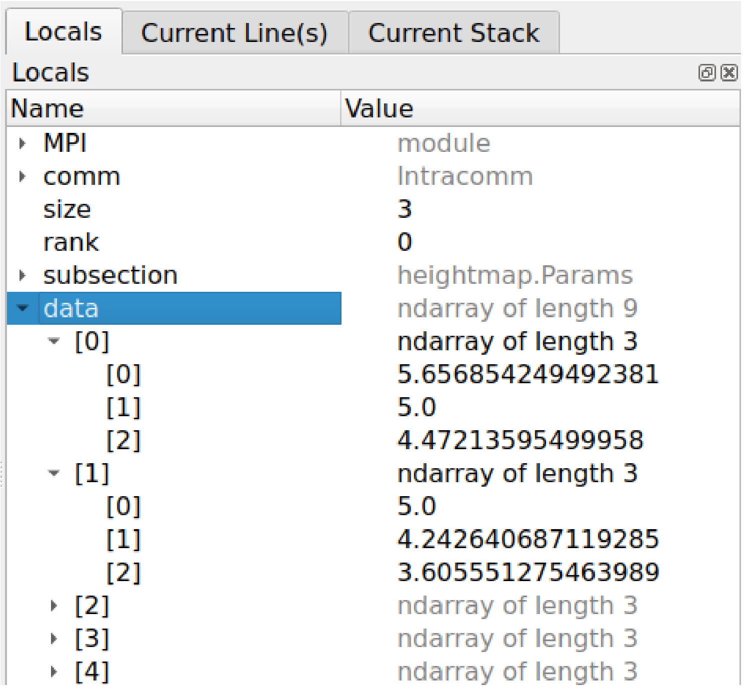
New Python debugging features in DDT 20.2

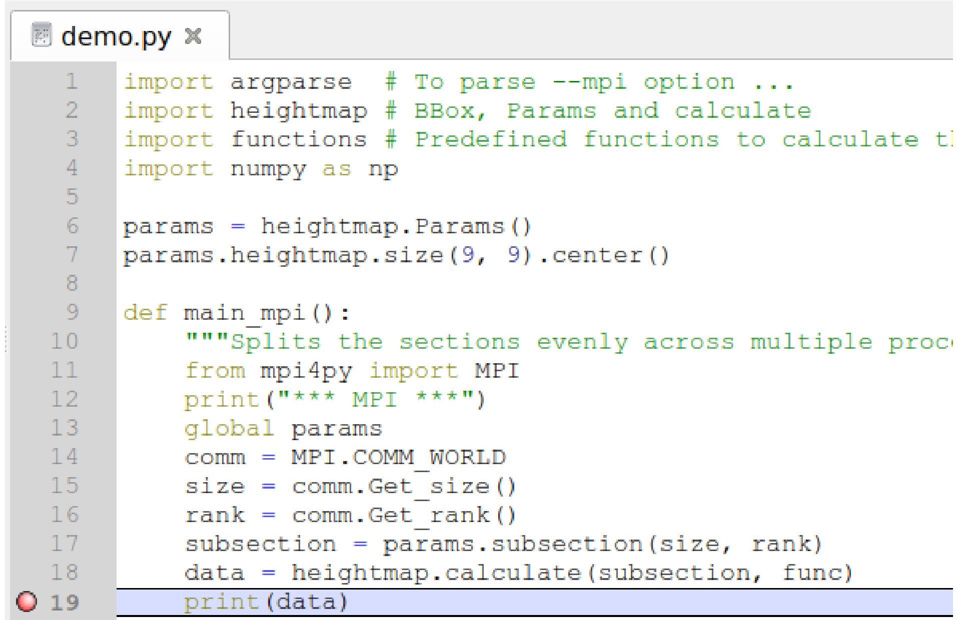




Inspecting NumPy Arrays

New Python debugging features in DDT 20.2

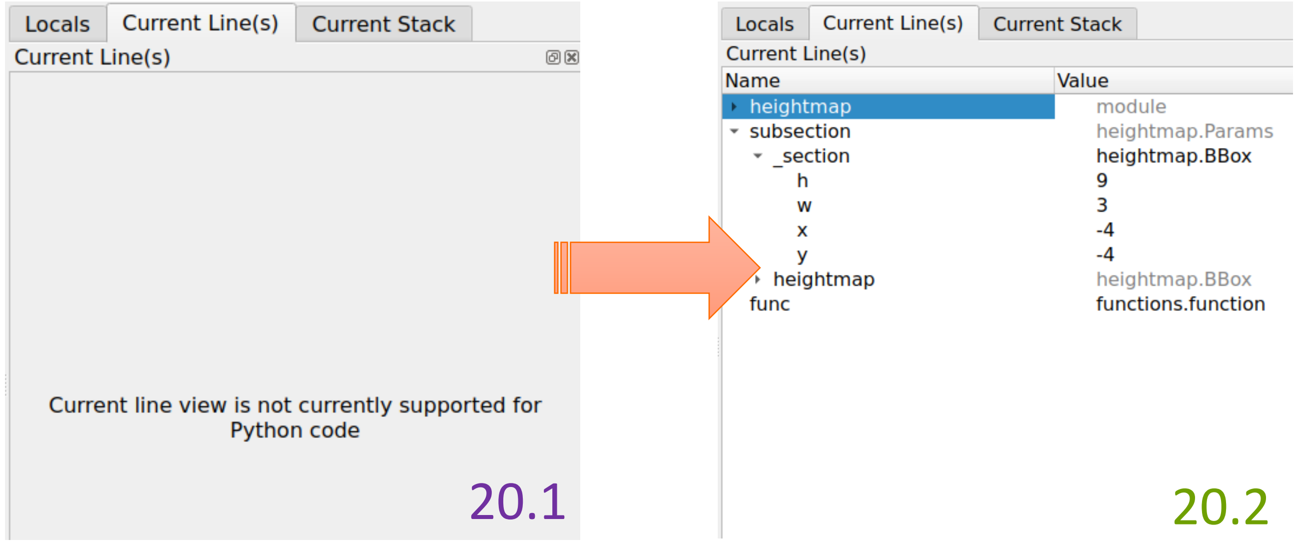


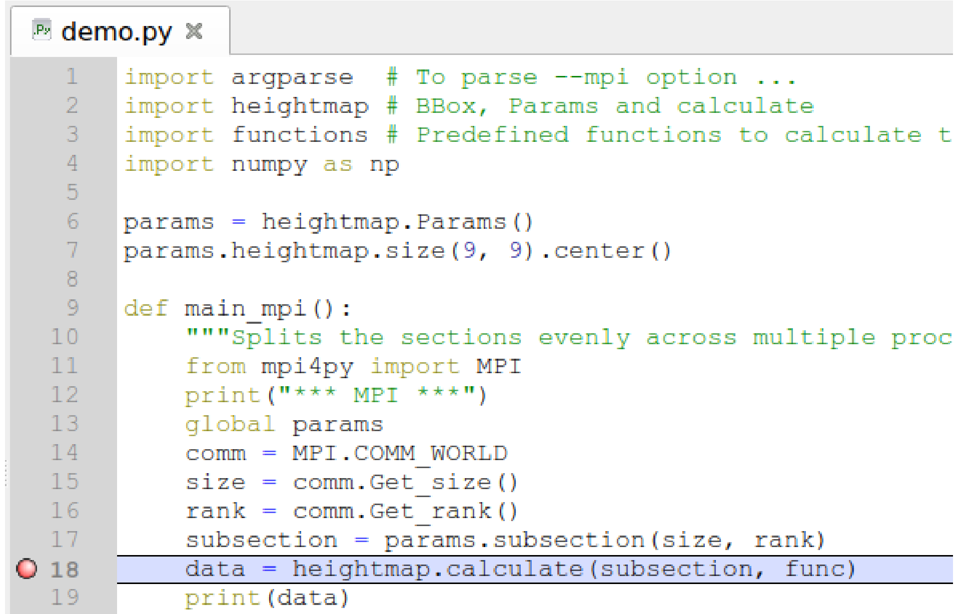


Not limited to NumPy arrays ➜ Python sequence protocol!

Variables on Current Line(s)

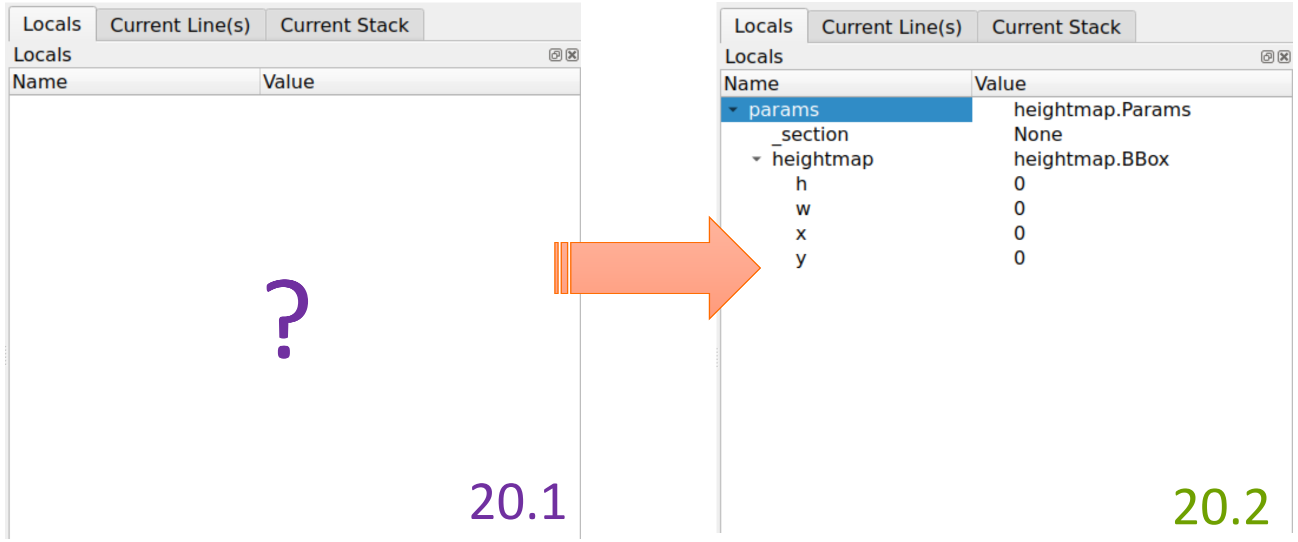
New Python debugging features in DDT 20.2

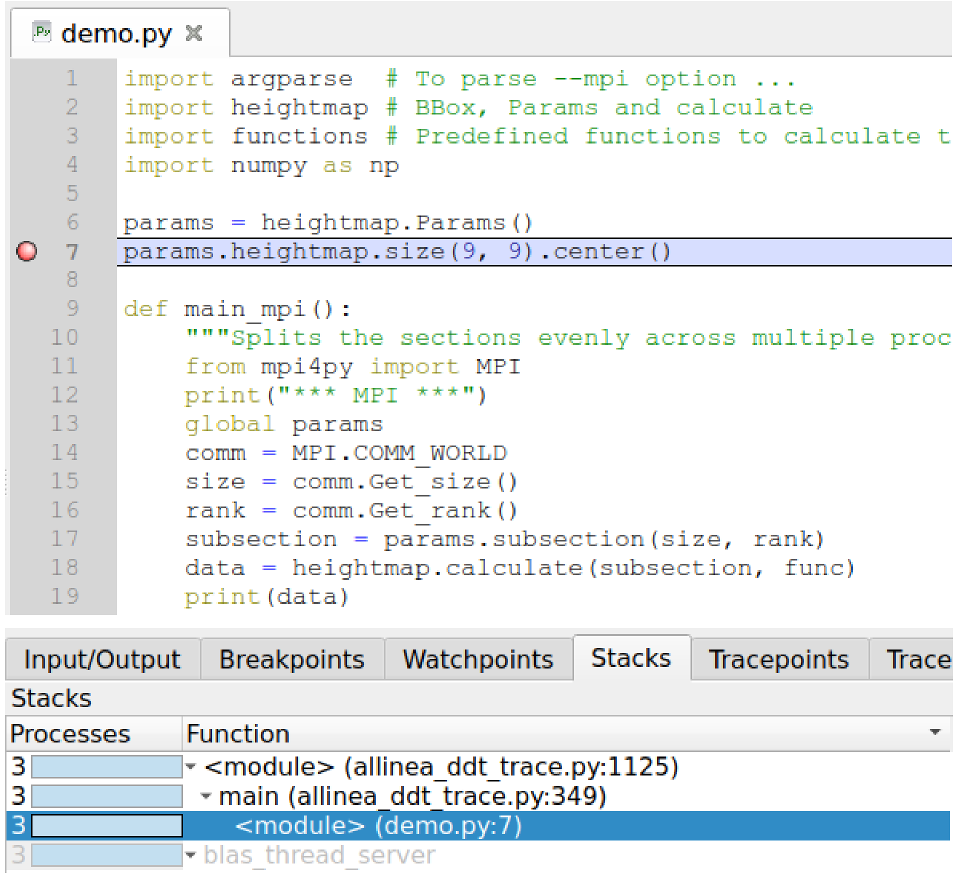




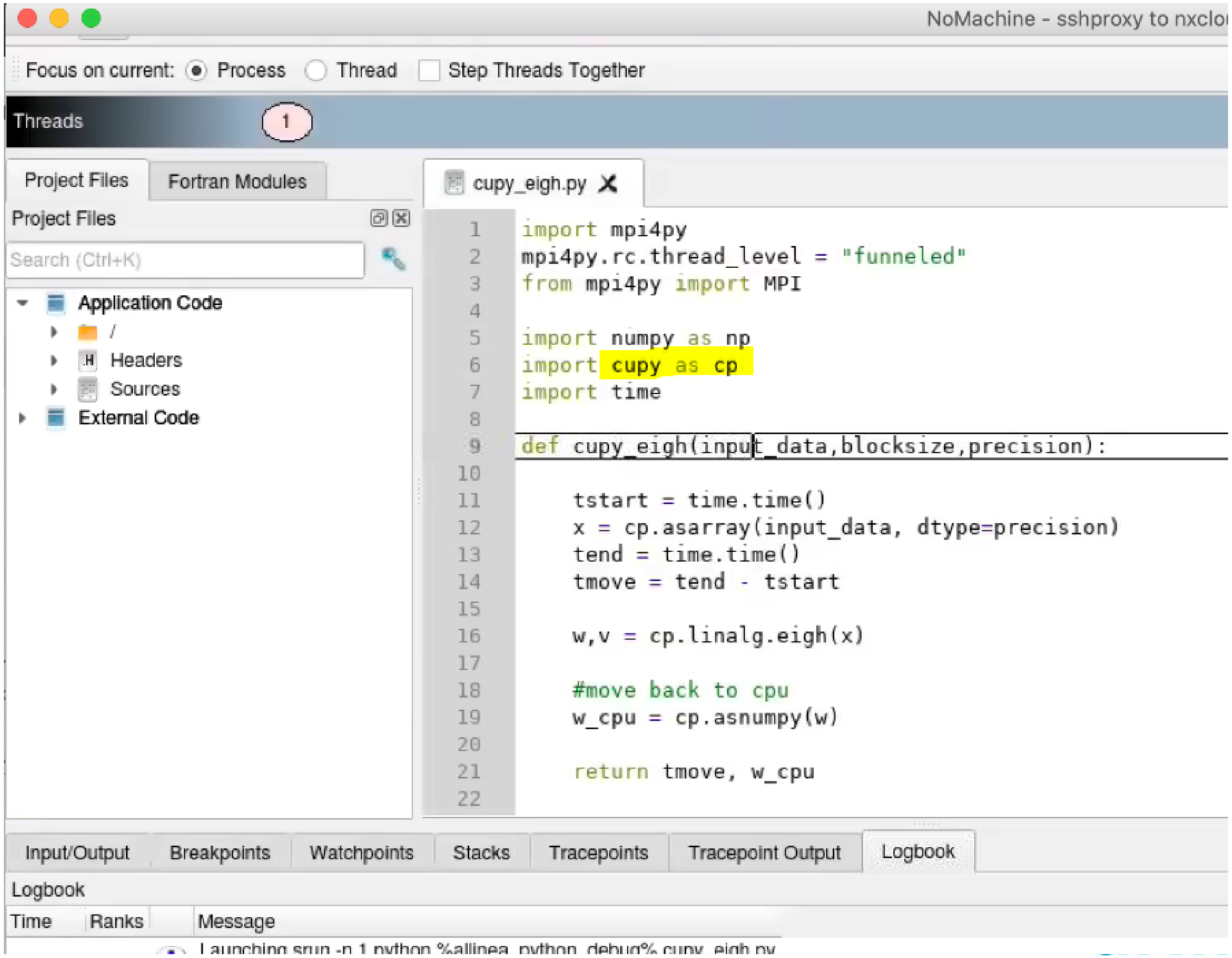
Quick Script: Globals?

New Python debugging features in DDT 20.2





Debugging Python that Uses cuda-based Toolkits



9 Step guide: optimizing high performance applications

arm

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.

1 Bugs

- ✓ Correct application.

2 Analyze before you optimize

- ✓ Measure all performance aspects. You can't fix what you can't see.
- ✓ Prefer real workloads over artificial tests.

3 I/O

- ✓ Discover lines of code spending a long time in I/O.
- ✓ Trace and debug slow access patterns.

4 Workload

- ✓ Detect issues with balance.
- ✓ Slow communication calls and processes. Dive into partitioning code.

5 Communication

- ✓ Track communication performance.
- ✓ Discover which communication calls are slow and why.

6 Memory

- ✓ Reveal lines of code bottlenecked by memory access times.
- ✓ Trace allocation and use of hot data structures.

7 Cores

- ✓ Discover synchronization overhead and core utilization.
- ✓ Synchronization-heavy code and implicit barriers are revealed.

8 Vectorization

- ✓ Understand numerical intensity and vectorization level.
- ✓ Hot loops, unvectorized code and GPU performance revealed.

9 Verification

- ✓✓ Validate corrections and optimal Performance.

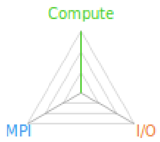
Key:

- ✓ arm PERFORMANCE REPORTS
- ✓ arm FORGE

Performance Snapshots with Arm Performance Reports

arm PERFORMANCE REPORTS

Command:	/ace/home/HCEEC002/nnm08/oxp09-nnm08/CloverLeaf_OpenMP/clover_leaf
Resources:	1 node (96 physical, 96 logical cores per node)
Memory:	126 GiB per node
Tasks:	1 process, OMP_NUM_THREADS was 8
Machine:	arm2
Start time:	Tue Aug 1 2017 14:55:32 (UTC+01)
Total time:	8 seconds
Full path:	/ace/home/HCEEC002/nnm08/oxp09-nnm08/CloverLeaf_OpenMP



Summary: clover_leaf is Compute-bound in this configuration

Compute	100.0%	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	0.0%	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 Metrics section below.

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

MPI

A breakdown of the 0.0% MPI time:

Time in collective calls	0.0%
Time in point-to-point calls	0.0%
Effective process collective rate	0.00 bytes/s
Effective process point-to-point rate	0.00 bytes/s

No time is spent in MPI operations. There's nothing to optimize here!

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

OpenMP

A breakdown of the 99.7% time in OpenMP regions:

Computation	85.6%
Synchronization	14.4%
Physical core utilization	8.3%
System load	7.8%

Physical core utilization is low and some cores may be unused. Try increasing OMP_NUM_THREADS to improve performance.

Memory

Per-process memory usage may also affect scaling:

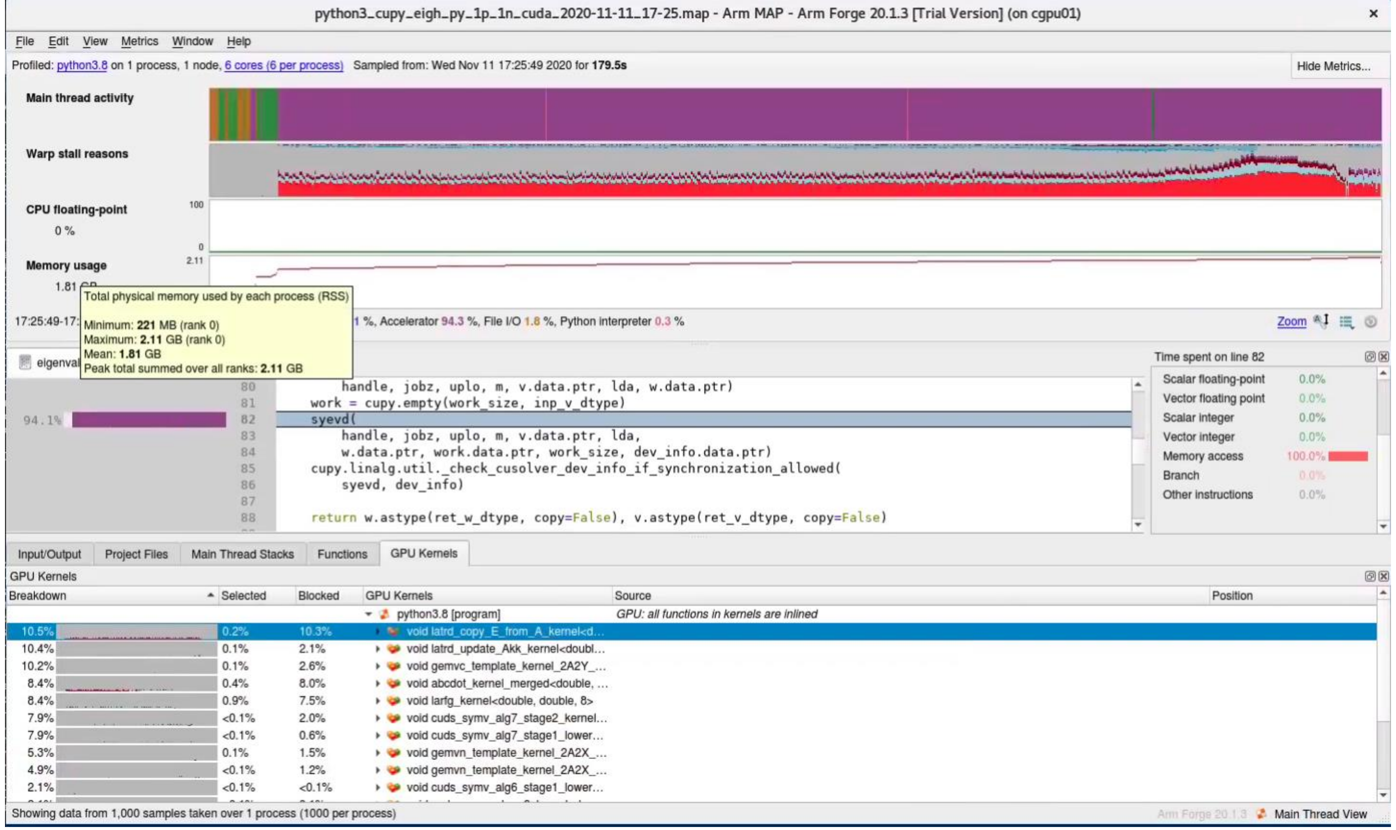
Mean process memory usage	312 MiB
Peak process memory usage	314 MiB
Peak node memory usage	2.0%

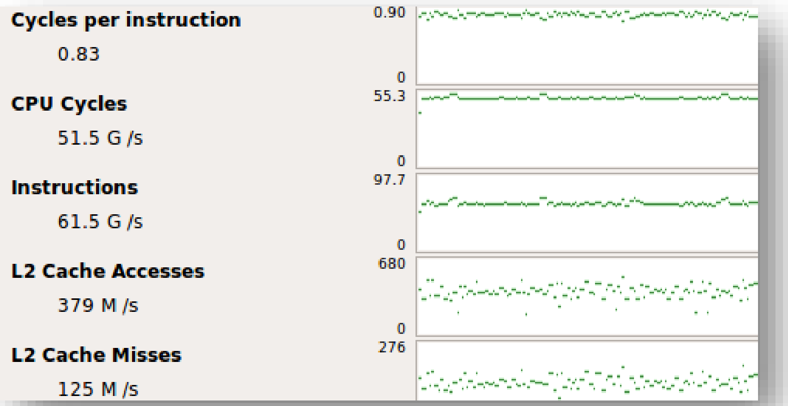
The peak node memory usage is very low. Larger problem sets can be run before scaling to multiple nodes.

- No source code needed
- Less than 5% runtime overhead
- Fully scalable
- Run regularly – or in regression tests
- Explicit and usable output

MAP Highlights

- MAP is a sampling based scalable profiler
  - Built on same framework as DDT
  - Parallel support for MPI, OpenMP, CUDA
  - Designed for C/C++/Fortran, and Python
- Designed for ‘hot-spot’ analysis
  - Stack traces
  - Augmented with performance metrics
- Adaptive sampling rate
  - Periodically resamples and modifies sampling frequency
  - Low overhead, scalable and small file size



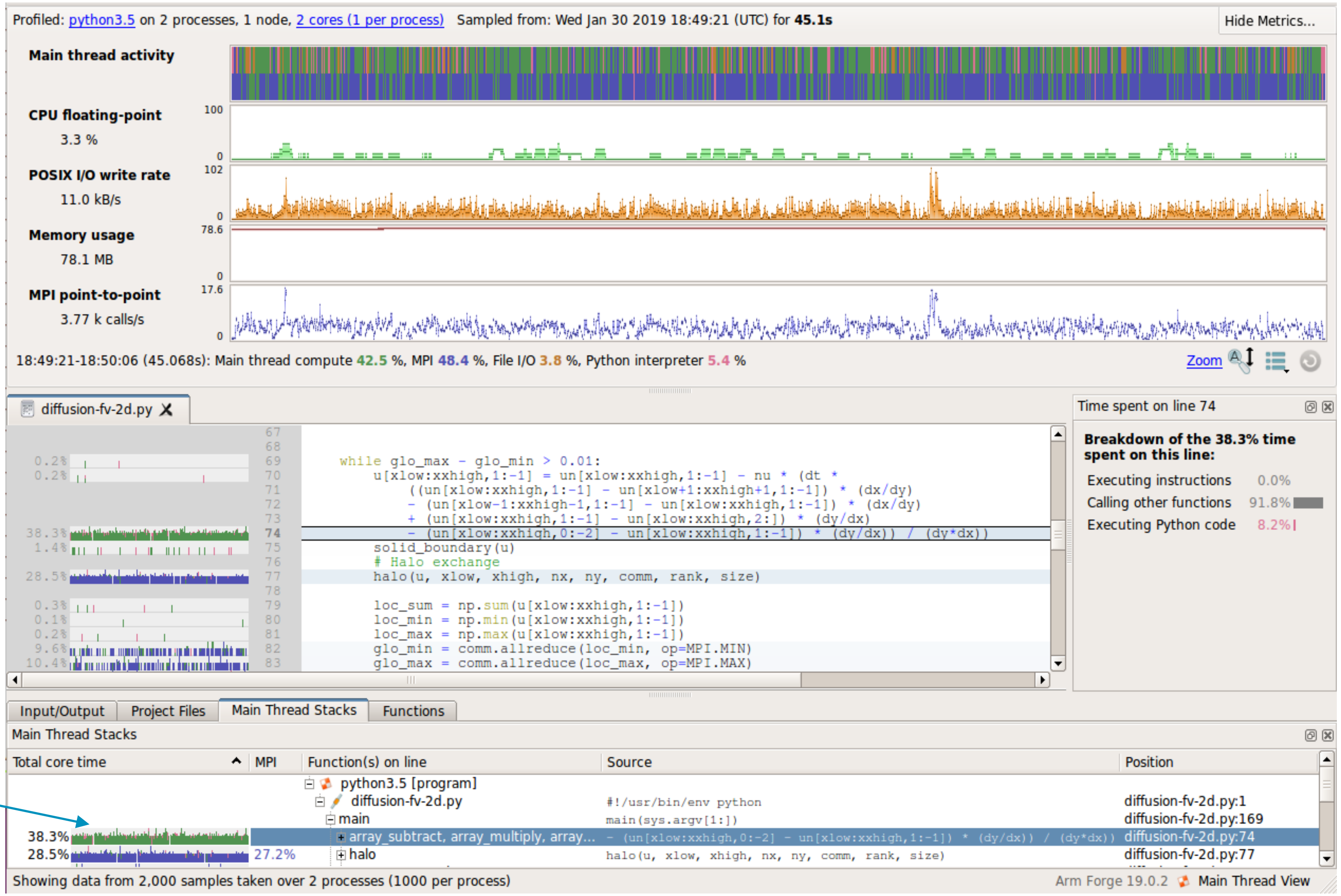


arm

Python Source Code Profiling with MAP

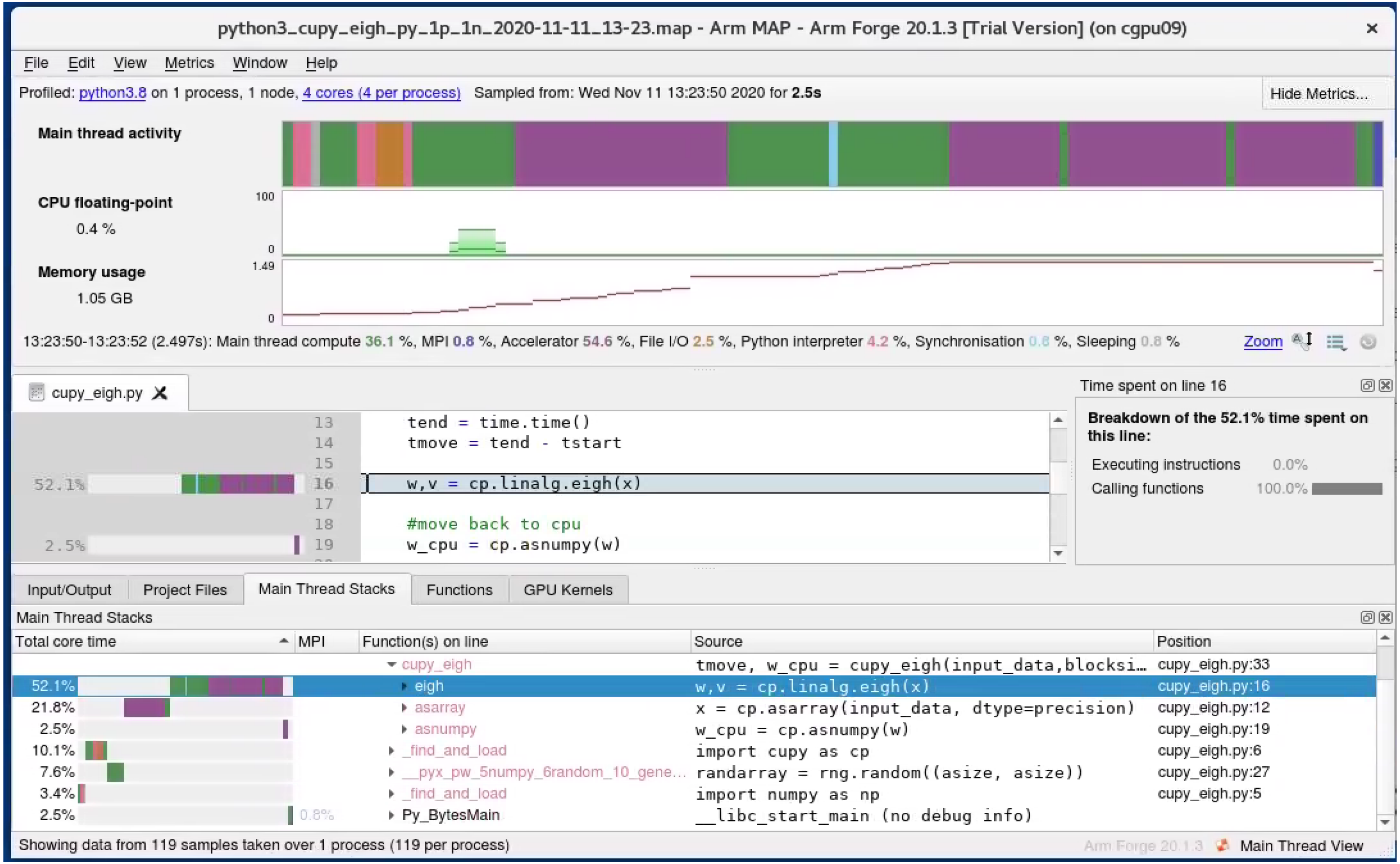
- Profile Pure and Mixed code
  - Call stacks
  - Time in interpreter
- Works with MPI4PY
  - Usual MAP metrics
- Source code view
  - Mixed language support

Note: Green as operation is on numpy array, so backed by C routine, not Python (which would be pink)

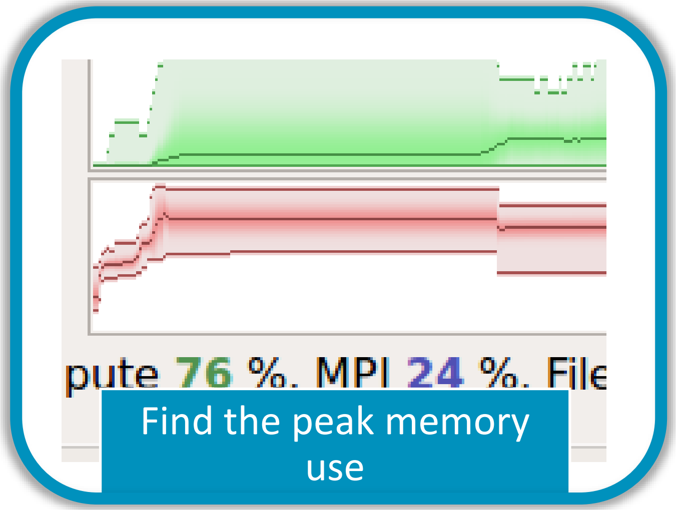



```
map --profile jsrun -n 2 python3 ./diffusion-fv-2d.py
```

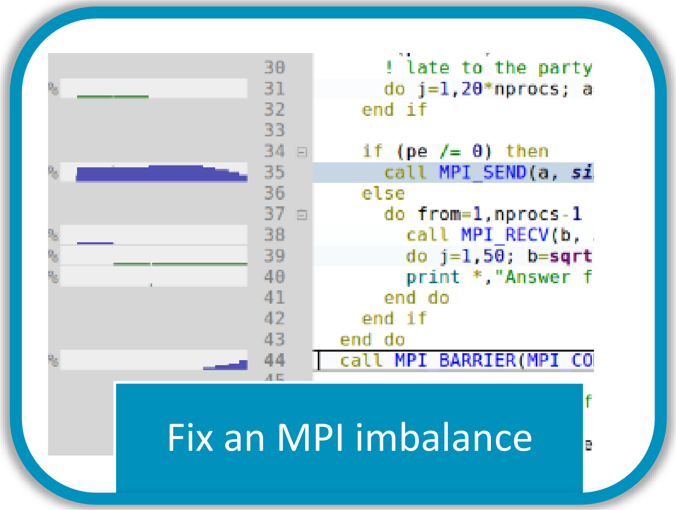
Profiling Python that Uses Cuda-Based Toolkits



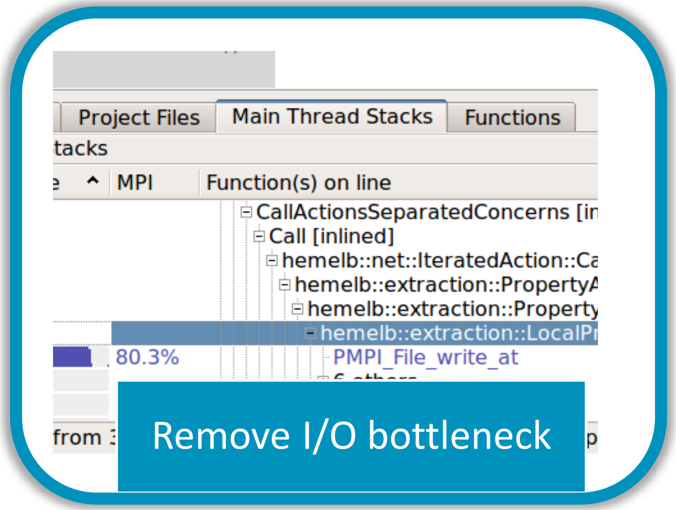
MAP Source Code Profiler Highlights



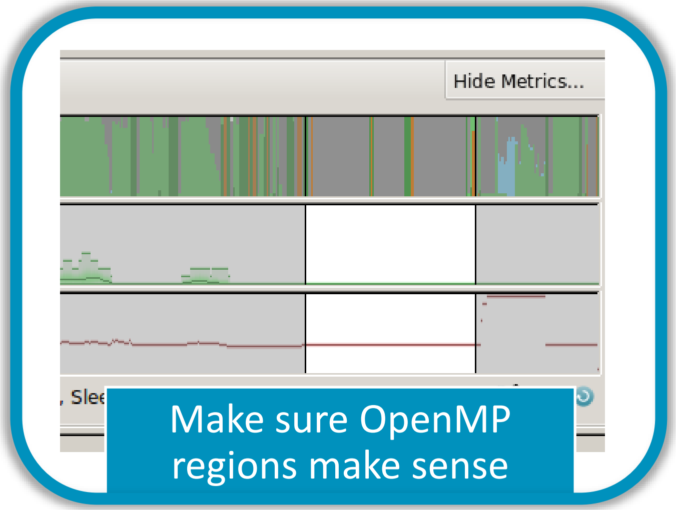
Find the peak memory use



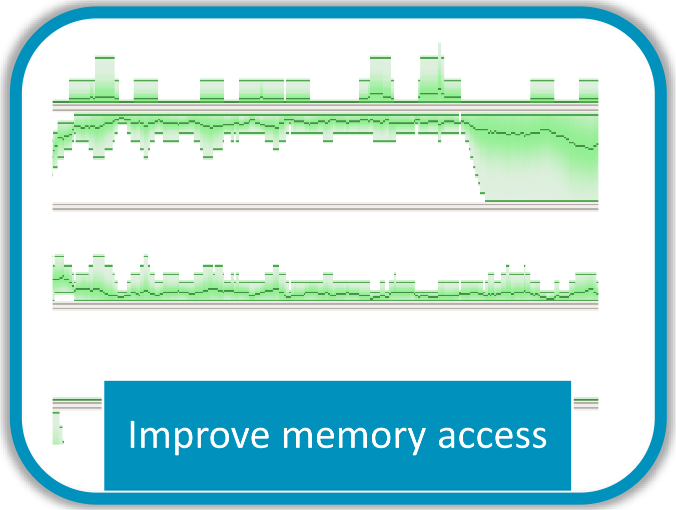
Fix an MPI imbalance



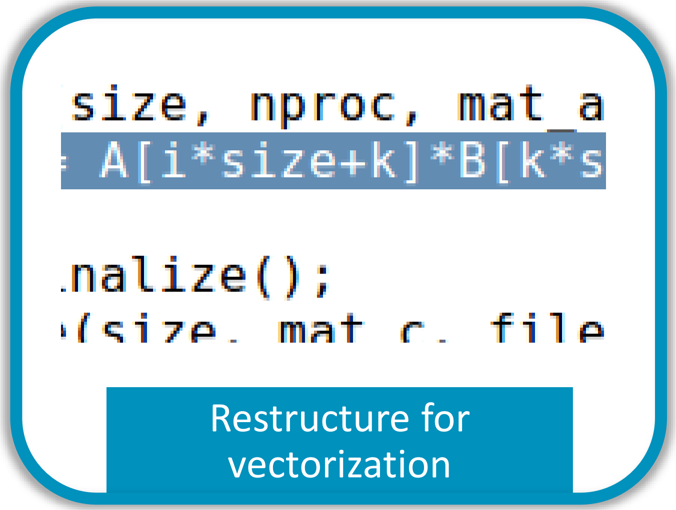
Remove I/O bottleneck



Make sure OpenMP regions make sense



Improve memory access



Restructure for vectorization

Next Steps

- Contact the Arm HPC Tools team to learn more
- Schedule a live demo to see the tools in action
- Obtain a trial to have a look yourself (Most RMACC members have licenses for Forge)

Beau Paisley, Beau.Paisley@arm.com, Solutions Architect, North America – HPC Tools

Andrew Westergren, Andrew.Westergren@arm.com Senior Manager, North America – HPC Tools