Profiling without `printf`

John Linford <john.Linford@arm.com>

July 2021

© 2021 Arm

What's on the menu?

(most of these slides were written on a lunch break btw)

- Profiling methodology
  - The how, when, and why
  - Story time
- Profiling with ReFrame
  - Arm Forge: MAP and Performance Reports
  - perf
  - mpiP
  - perf-lib-tools
- More about Arm Forge -- MAP and Performance Reports
- Dessert: I/O profiling with MAP
  - Not presented today – go read the slides

© 2021 Arm
arm

Profiling Methodology

© 2021 Arm

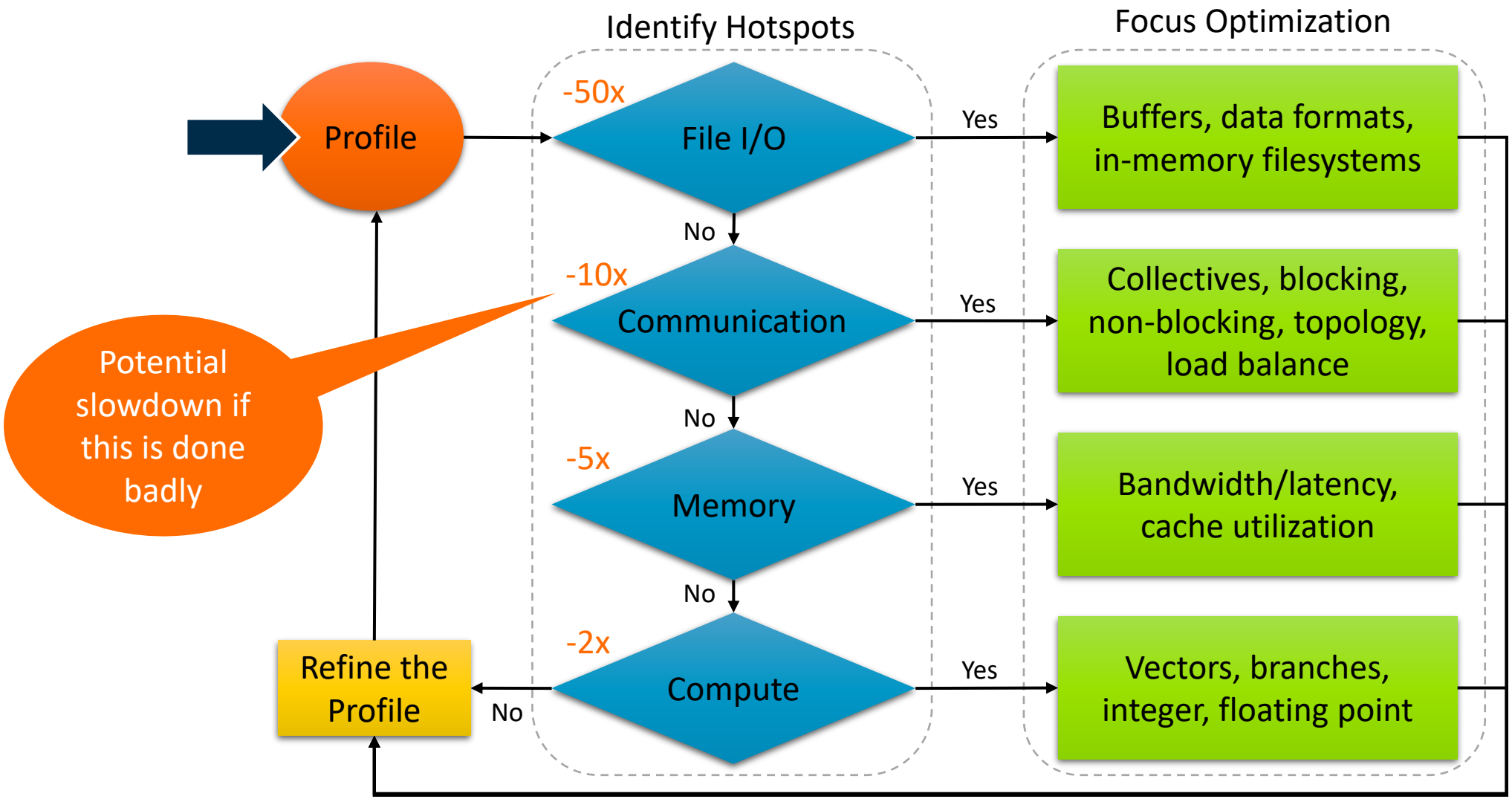
Step 0: Don’t Profile

The profiling preflight checklist

- The code compiles and runs at the desired scale
  - May need to run at a reduced scale, profile, then scale up
- Results are correct, and a copy of the unprofiled results are saved somewhere
  - Profilers occasionally expose surprising bugs. Keep a gold standard handy.
- I’ve recorded the wallclock runtime of the code *without a profiler*.
  - You’ll need this to keep track of profiler overhead.
  - Some profilers are “lightweight” with <10% overhead. Some may slow programs by 100x or more. Depends on what’s being measured.
- Don’t try to profile on the first run. It’s tempting, but you’ll probably waste time.

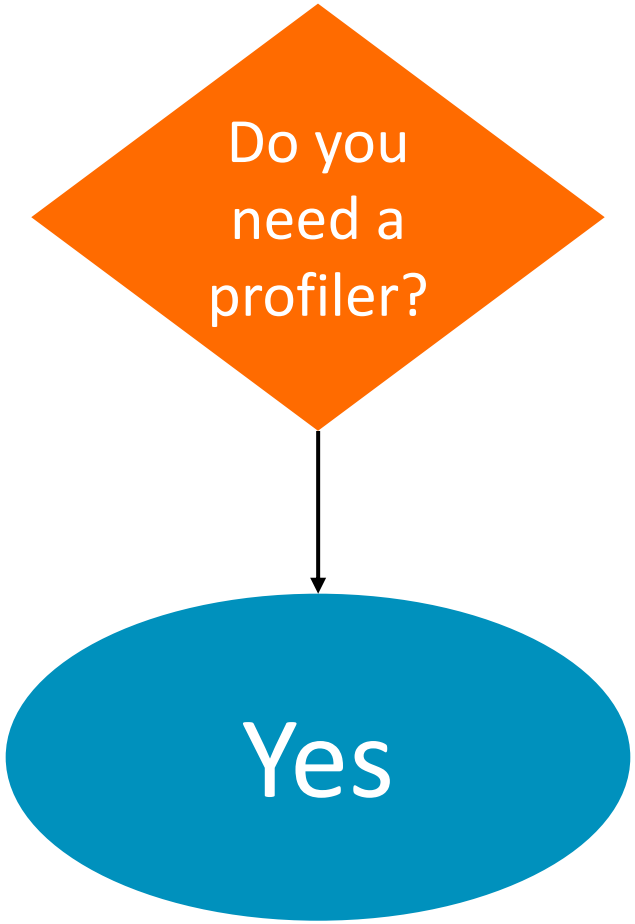
© 2021 Arm

Identifying and resolving performance issues



© 2021 Arm
arm

Do you ***really*** need to use a profiler?
To decide, just follow this handy flow chart



© 2021 Arm

Story time: CCC and the ORNL GPU Hackathon @ Pawsey

Quantum collisions in atomic and molecular physics

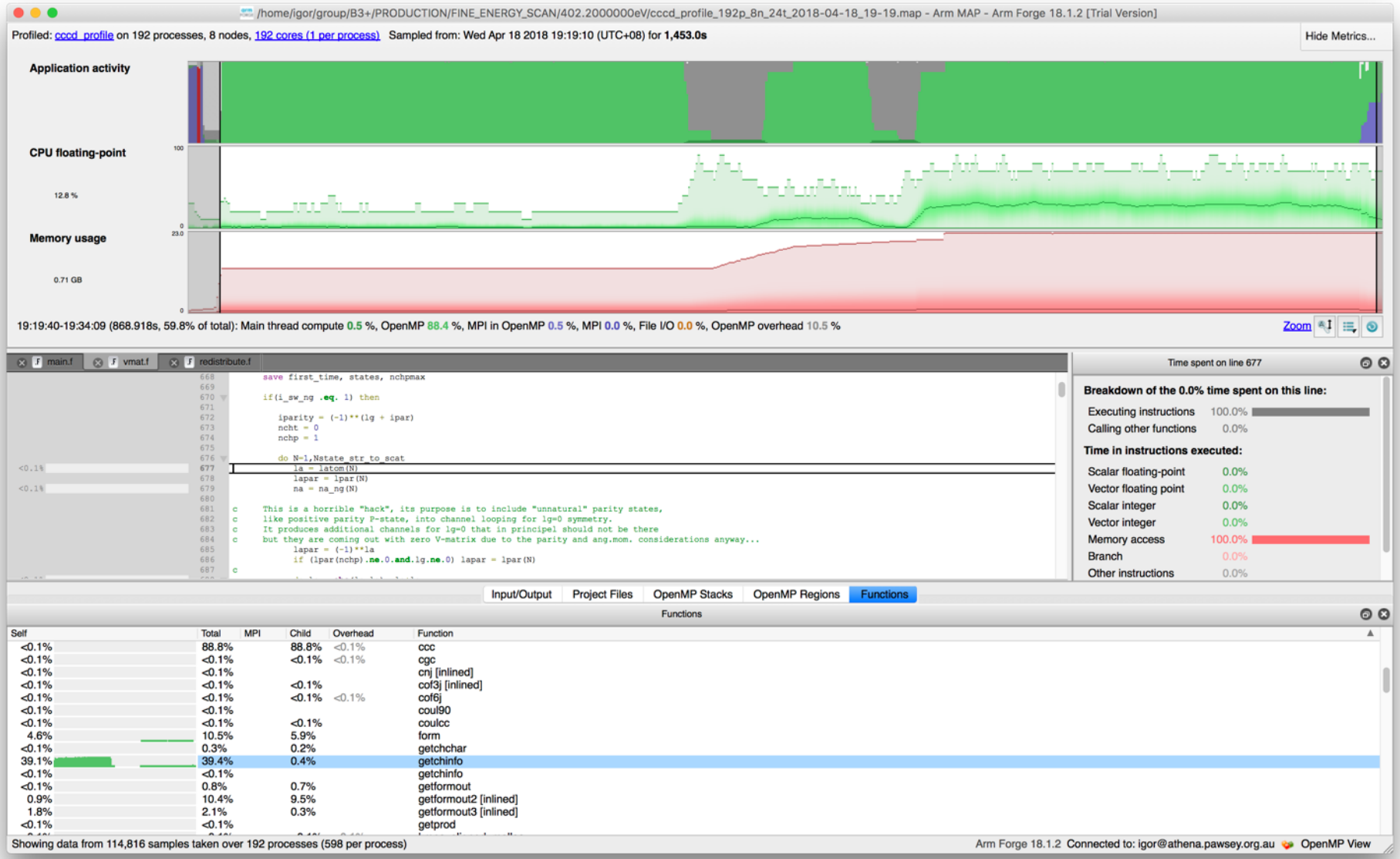
- CCC: Quantum mechanics
  - Fusion energy
  - Laser science
  - Lighting industry
  - Medical imaging / therapy
  - Astrophysics
- Igor Bray, Head of Physics and Astronomy, and the Theoretical Physics Group, in the Faculty of Science and Engineering, at Curtin University

Curtin University

© 2021 Arm

arm

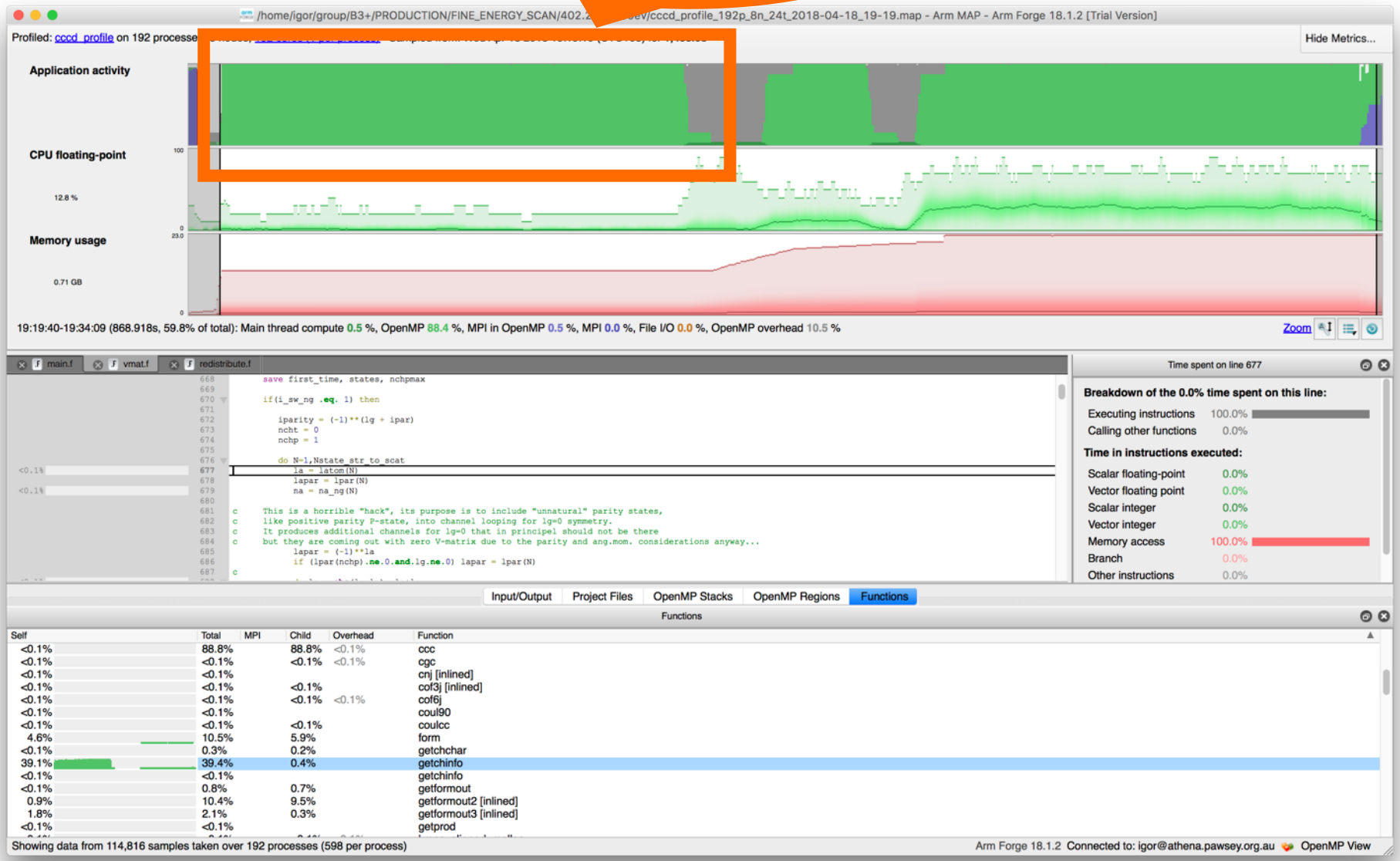
Initial Profile with Arm Forge (“MAP”)



© 2021 Arm

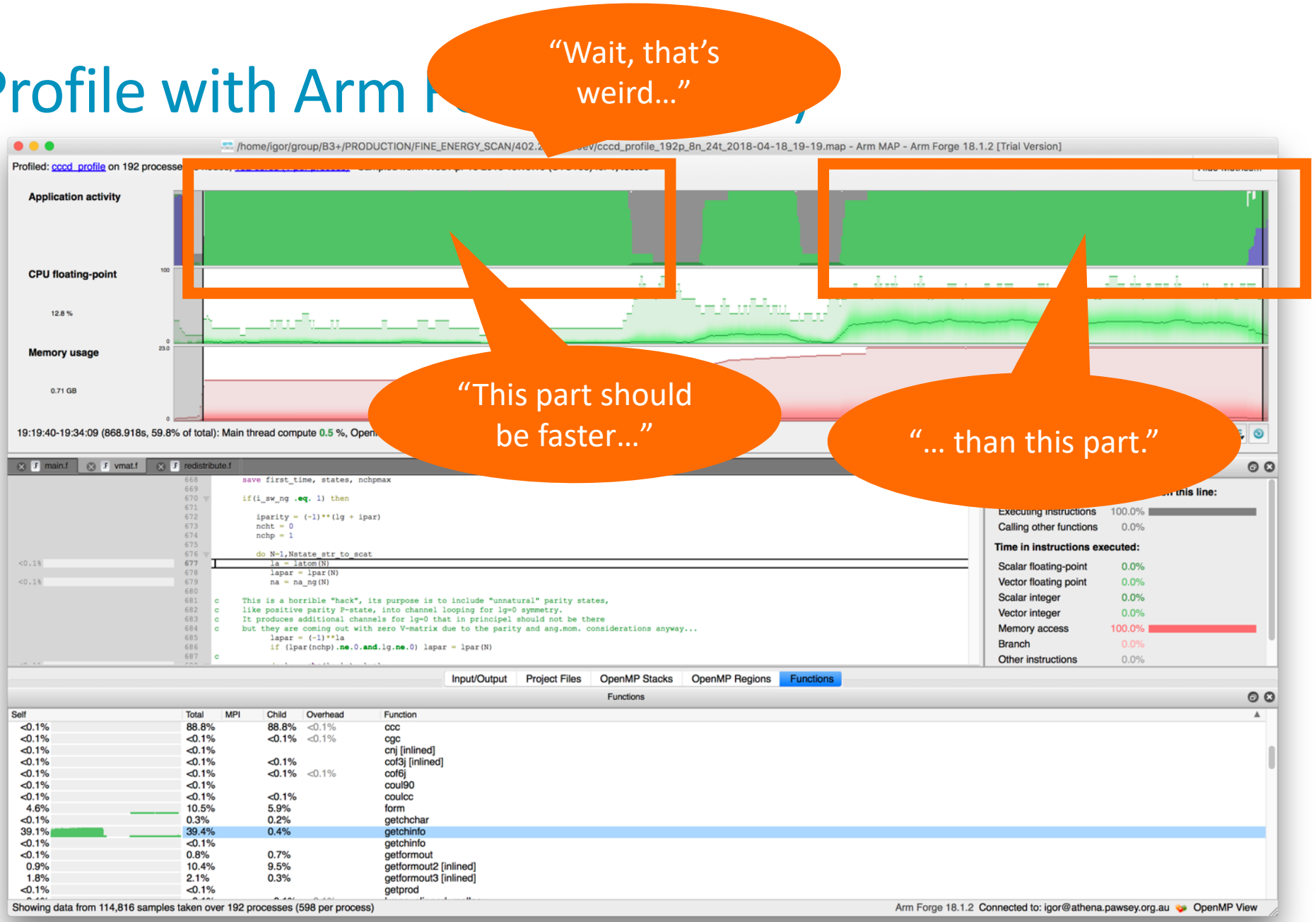
Initial Profile with Arm M

"Wait, that's weird…"



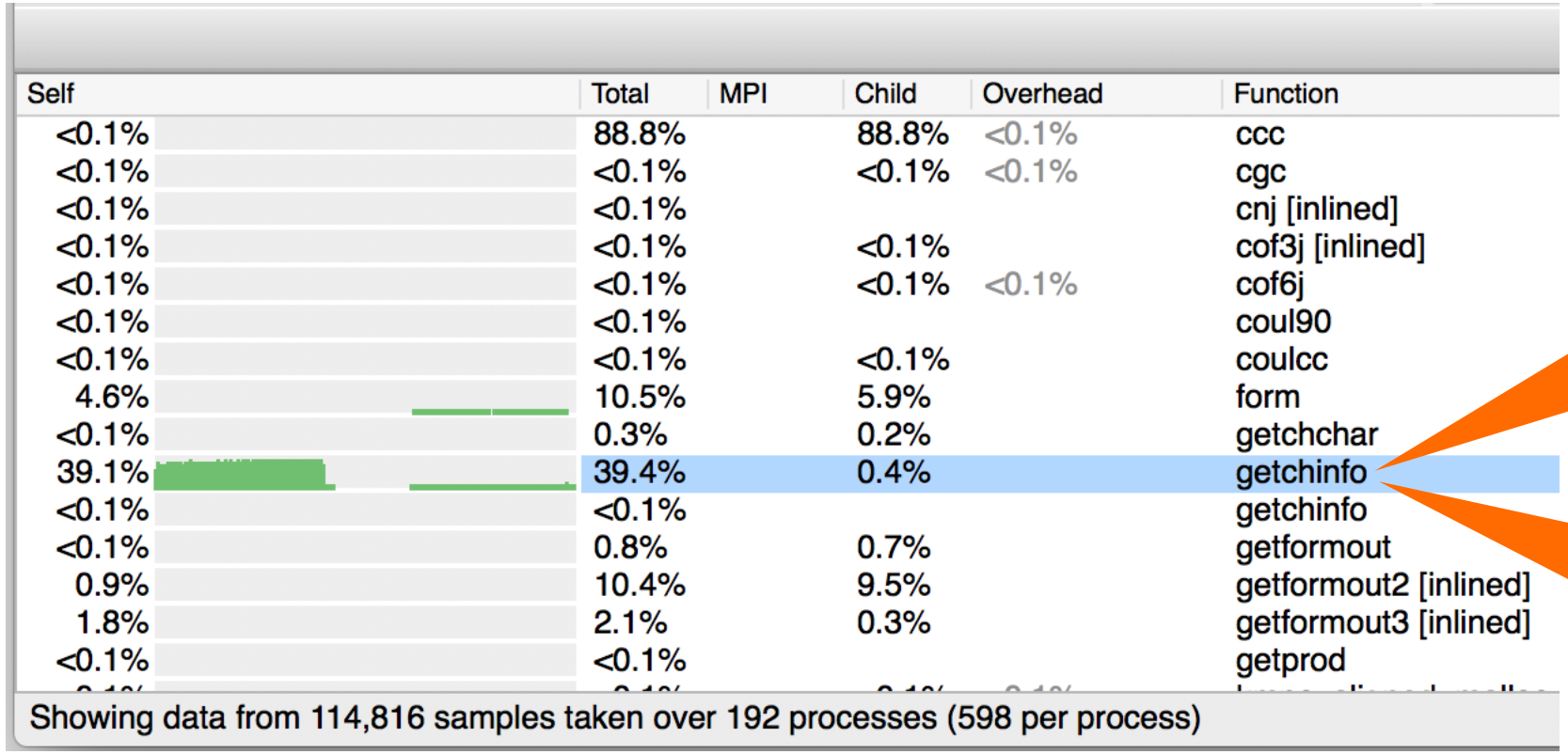
© 2021 Arm

Initial Profile with Arm M



© 2021 Arm

What's taking so much time in the first green block?



Self	Total	MPI	Child	Overhead	Function
<0.1%	88.8%		88.8%	<0.1%	ccc
<0.1%	<0.1%		<0.1%	<0.1%	cgc
<0.1%	<0.1%				cnj [inlined]
<0.1%	<0.1%		<0.1%		cof3j [inlined]
<0.1%	<0.1%		<0.1%	<0.1%	cof6j
<0.1%	<0.1%				coul90
<0.1%	<0.1%		<0.1%		coulcc
4.6%	10.5%		5.9%		form
<0.1%	0.3%		0.2%		getchchar
39.1%	39.4%		0.4%		getchinfo
<0.1%	<0.1%				getchinfo
<0.1%	0.8%		0.7%		getformout
0.9%	10.4%		9.5%		getformout2 [inlined]
1.8%	2.1%		0.3%		getformout3 [inlined]
<0.1%	<0.1%				getprod

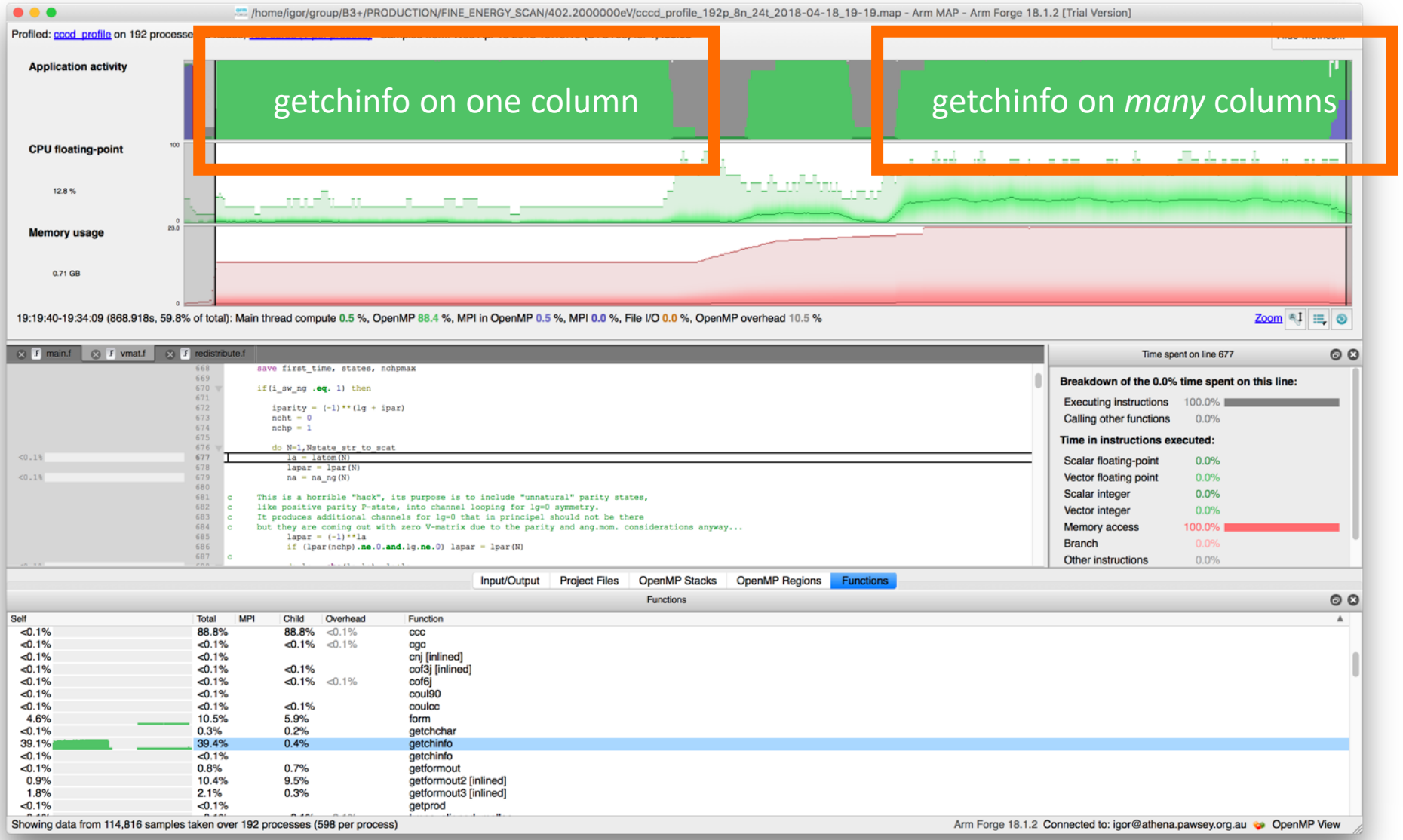
Showing data from 114,816 samples taken over 192 processes (598 per process)

`getchinfo(X, Y)` operates on matrix columns X…Y

`getchinfo(X, X)` takes as much time as `getchinfo(X, Y)` ????

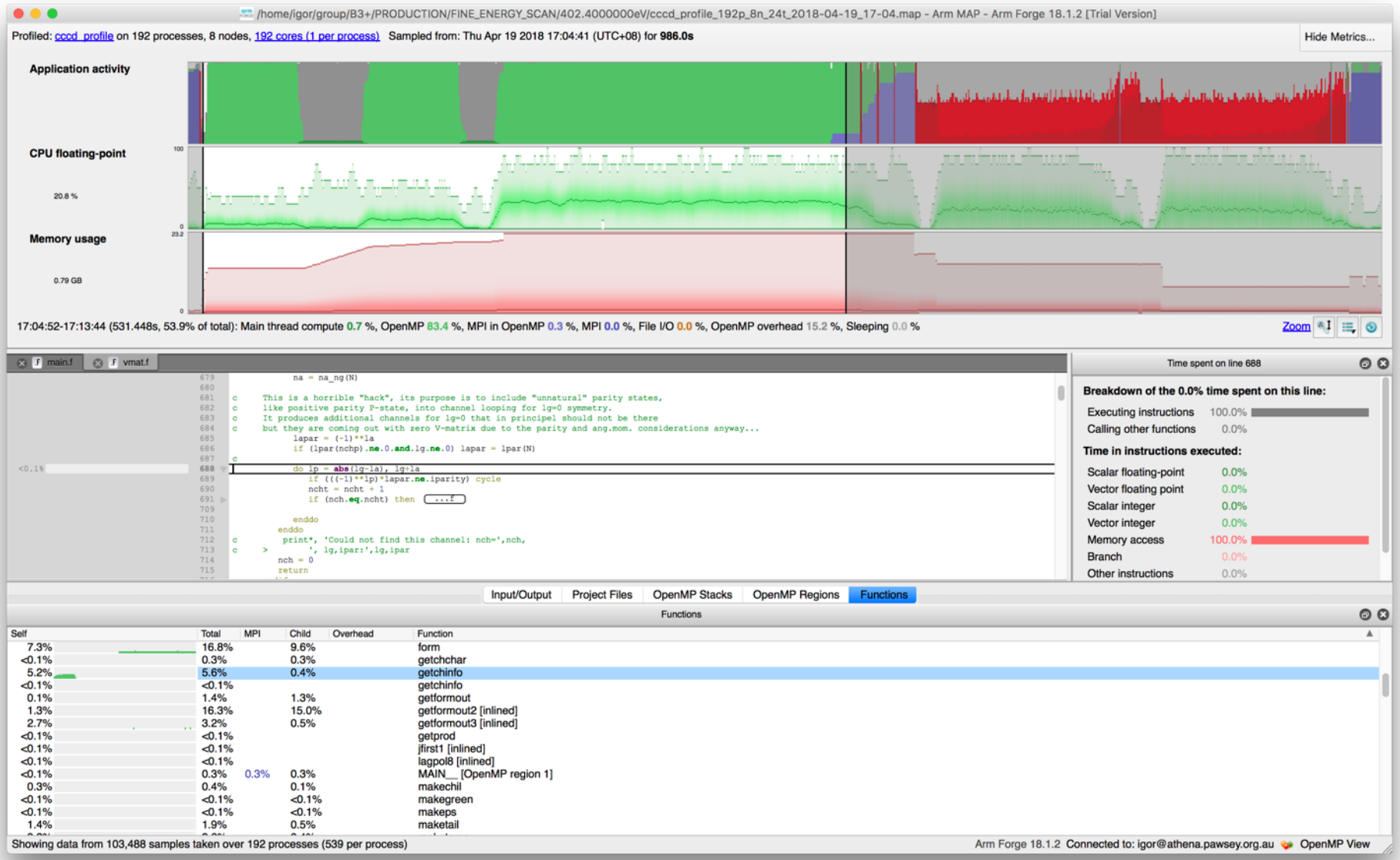
© 2021 Arm
arm

Initial Profile with Arm Forge (“MAP”)



© 2021 Arm

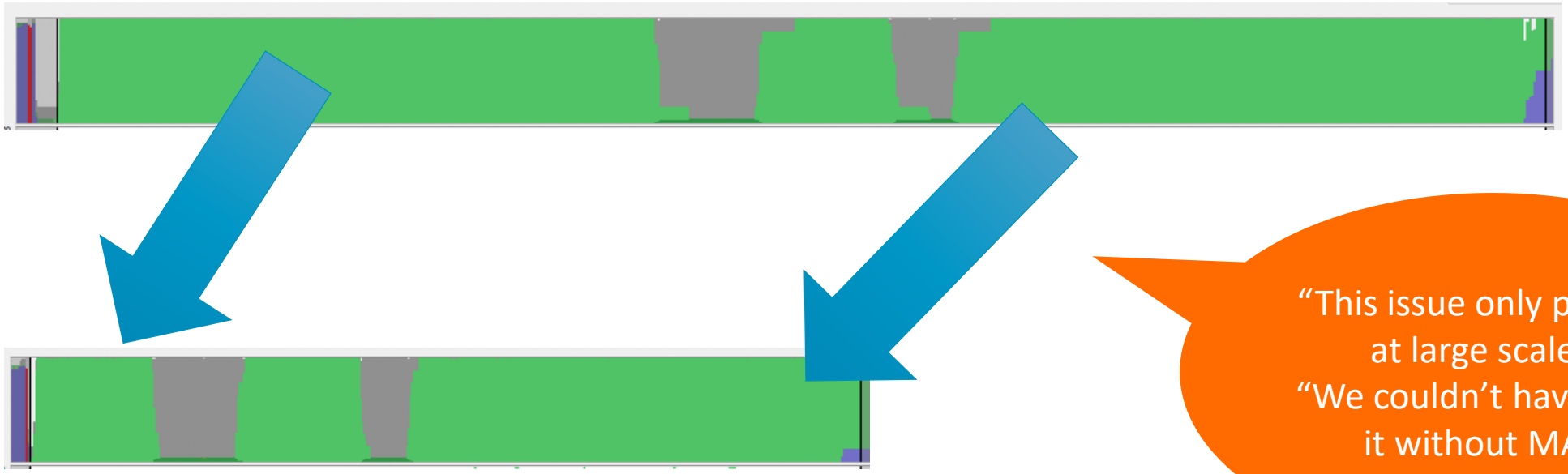
A bug! `a(:)` should be `a(1:N)`



© 2021 Arm

That’s better!

4.6%	10.5%	5.9%	form
<0.1%	0.3%	0.2%	getchchar
39.1%	39.4%	0.4%	getchinfo
<0.1%	<0.1%		getchinfo
<0.1%	0.8%	0.7%	getformout



Self	Total	MPI	Child	Overhead	Function
7.3%	16.8%		9.6%		form
<0.1%	0.3%		0.3%		getchchar
5.2%	5.6%		0.4%		getchinfo
<0.1%	<0.1%				getchinfo
0.1%	1.4%		1.3%		getformout

“This issue only presents at large scales.”
“We couldn’t have found it without MAP.”

© 2021 Arm
arm

Takeaway

- Listen to the experts
  - **#mentors-conclave** on slack
  - Just ask “does this look right?”
- If you’re an expert, do use a profiler
- If you’re not an expert, do use profiler
- Compare knowns to unknowns
  - Example: are the x86 and aarch64 profiles similar?
- Measure performance at scale, then “zoom in”

© 2021 Arm

Alright alright already! I'll use a profiler!

....

How, exactly?

© 2021 Arm

Profiling with ReFrame

© 2021 Arm

Quick Profiler Overview

Name	Typical Use	Typical Scale	Languages	Metrics
Arm MAP + Performance Reports	Initial high-level profile to identify hot spots	Single process to thousands of processes	C, C++, Fortran, Python	Wallclock time, hardware perf, MPI, OpenMP, CUDA, custom
perf	Quick detailed profile of a single process	Single process	Anything, usually compiled	Wallclock time, hardware perf
mpiP	MPI communication profile at any scale	From two to thousands	Anything with MPI	MPI
perf-lib-tools	Math library call profile	Single process	C, C++, Fortran	Count BLAS/LAPACK/FFT calls, call parameters

© 2021 Arm

Prerequisites – just for this hackathon

- Add the “extra” Arm repo
  - This connects the existing Arm Forge installation back to spack/ReFrame/etc.
  - https://cloud-hpc-hackathon.workshop.aws/spack/fork.html#add-arm-extra

```
if [[ $(uname -i) == "x86_64" ]];then echo "You are not running on an ARM instance, no need to do this"
else
  cd ${HOME}
  mkdir -p .spack
  git clone https://github.com/arm-hpc-user-group/arm_spack_extra.git
  echo -e "repos:\n- ${HOME}/arm_spack_extra" > ~/.spack/repos.yaml
fi
```

- spack install arm-forge
- spack install mpip
- spack install perf-libs-tools-git

© 2021 Arm

ReFrame + Arm MAP and Arm Performance Reports

Cloud-HPC-Hackathon-2021/Guides/**Profiling/**

- The “map” command comes before srun (or similar), so use a LaunchWrapper:

```
# Here we modify the launcher to use the MAP profiler, and generate a `profile.map` file
# We tell ReFrame to stage this file back too
@run_before('run')
def set_profiler(self):
  self.proffile = 'profile.map'
  self.keep_files.append(self.proffile)

  self.modules.append('armforge')

  self.job.launcher = LauncherWrapper(self.job.launcher, 'map',
                                      ['--profile', '--output='+self.proffile])
```

- The resulting profile is saved as profile.map
- Use “perf-report profile.map” to generate performance reports:
  - clover_leaf_64p_1n_1t_2021-07-14_02-43.html
  - clover_leaf_64p_1n_1t_2021-07-14_02-43.txt

© 2021 Arm

ReFrame + perf

Cloud-HPC-Hackathon-2021/Guides/**Profiling_Perf/**

- Unlike MAP, the `perf` command comes after srun (or similar)
- Not pretty, but the easiest way is to change the executable to "**`perf`**"

```
# Define Execution
executable = 'perf'
executable_opts = ['stat', 'clover_leaf']
```

- Performance data for each MPI rank is written in the job output file

© 2021 Arm

ReFrame + mpiP

Cloud-HPC-Hackathon-2021/Guides/**Profiling_mpiP/**

- mpiP relies on environment variables

```
# Profiling
@run_before('run')
def preload_mpiP(self):
    # Load mpiP module
    self.modules.append('mpip')

    # Store output in a folder
    mpip_output = 'mpip_output'
    self.prerun_cmds.append('mkdir "%s"' % mpip_output)
    # Configure mpiP
    self.variables['MPIP'] = '"-f %s"' % mpip_output
    # Tell reframe to keep the folder after cleanup
    self.keep_files.append(mpip_output)

    # Add the LD_PRELOAD to srun
    self.job.launcher.options = ['--export=ALL,LD_PRELOAD=libmpiP.so']
```


© 2021 Arm

ReFrame + perf-lib-tools

Cloud-HPC-Hackathon-2021/Guides/**PerfLibsTools/**

- perf-libs-tools relies on environment variables and a post-processing script

```
# Profiling
@run_before('run')
def perf_libs_tools(self):

    # Load perf-libs-tools-module
    self.modules.append('perf-libs-tools-git')

    # Make folder to save results to
    apl_dir = 'plt_out'
    # Create the folder in pre_run cmd
    self.prerun_cmds.append('mkdir {0}'.format(apl_dir))
    # Set the Fileroot to the new folder
    self.variables['ARMPL_SUMMARY_FILEROOT'] = '$PWD/{0}/'.format(apl_dir)
    # Tell reframe to keep the folder after cleanup
    self.keep_files.append(apl_dir)

    # Add the LD_PRELOAD to srun
    self.job.launcher.options = ['--export=ALL,LD_PRELOAD=libarmpl-summarylog.so']

    # Make summary file
    apl_file = '{0}_{1}_apl_summary.log'.format(self.log_app_name, self.log_test_name)
    # Run post process
    self.postrun_cmds.append('process_summary.py {0}/*.apl > {1}'.format(apl_dir, apl_file))
    # Keep the summary file
    self.keep_files.append(apl_file)
```


© 2021 Arm

arm

More on MAP

© 2021 Arm

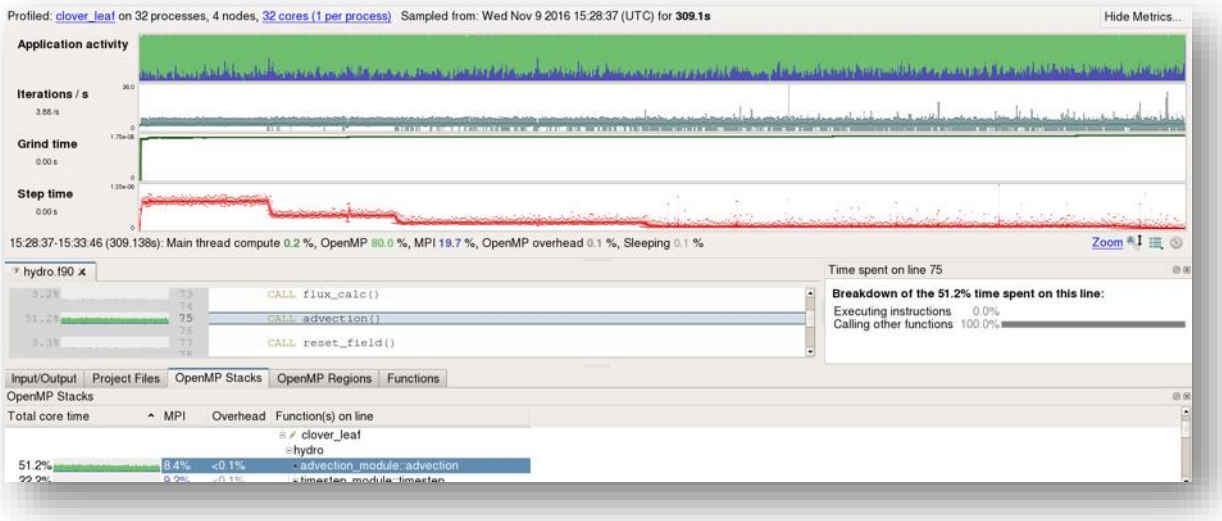
Arm MAP: Production-scale application profiling

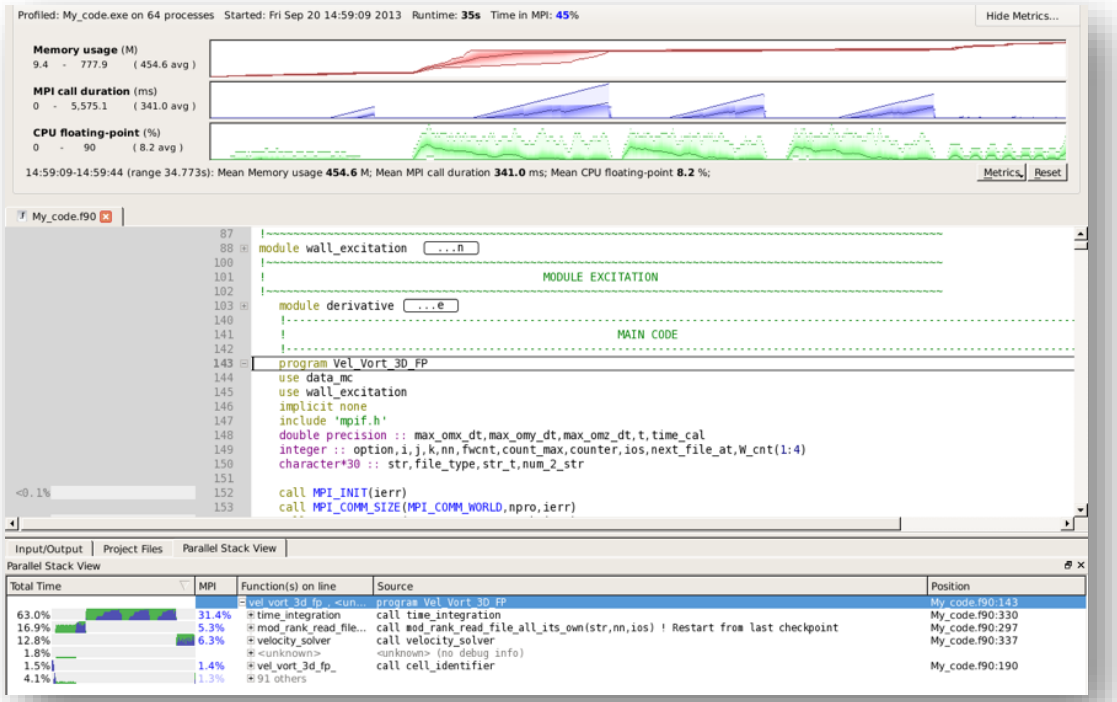
Identify bottlenecks and characterize performance at scale

- Runs from single thread up to many thousands of MPI ranks
- High level, statistical profiler with very low overhead at all scales
- Finds bottlenecks – doesn't say what to do about them

Example:

```
$> map -profile mpirun –n 48 ./example
```





© 2021 Arm

Arm MAP Overview

A lightweight sampling-based profiler for large scale jobs

Core Features

- MAP is a sampling based scalable profiler
  - Built on same framework as DDT
  - Parallel support for MPI, OpenMP
  - Designed for C/C++/Fortran/Python
- Designed for simple ‘hot-spot’ analysis
  - Stack traces
  - Augmented with performance metrics
- Lossy sampler
  - Throws data away – 1,000 samples / process
  - Low overhead, scalable and small file size

Performance Metrics

- Time classification
  - Based on call stacks
  - MPI, OpenMP, I/O, Synchronization
- Feature-specific metrics
  - MPI call and message rates
    - (P2P and collective bandwidth)
  - I/O data rates (POSIX or Lustre)
  - Energy data (IPMI or RAPL for Intel)
- Instruction information (hardware counters)
  - x86 – instruction breakdown + PAPI
  - aarch64 – perf metric for hardware counters

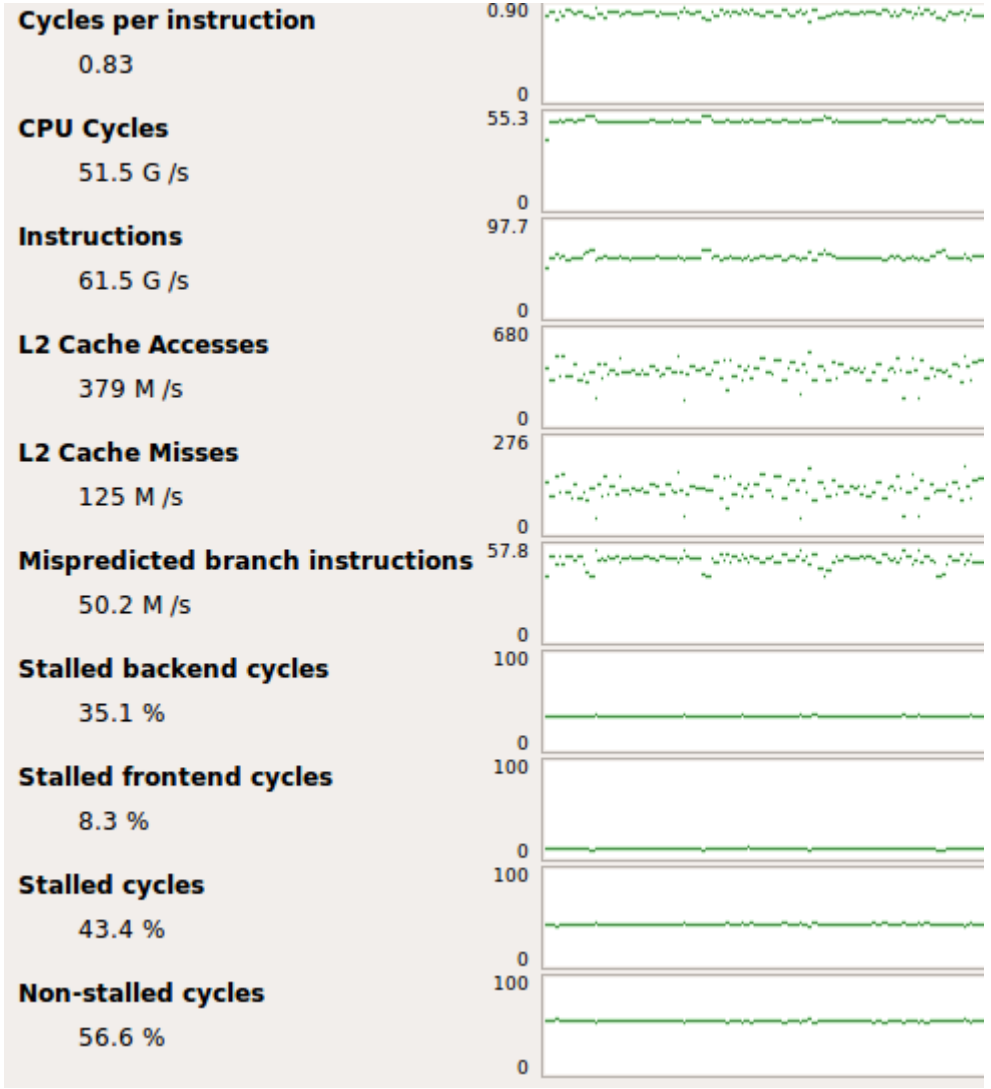
© 2021 Arm

Hardware Performance Metrics on Arm

MAP uses perf or PAPI to gather data.

- Predefined sets of hardware metrics
- Different CPUs have different capabilities, so you'll see different counters

- Generally, MAP tries to report on instruction mix
  - CPU, vectorization, memory, etc



© 2021 Arm

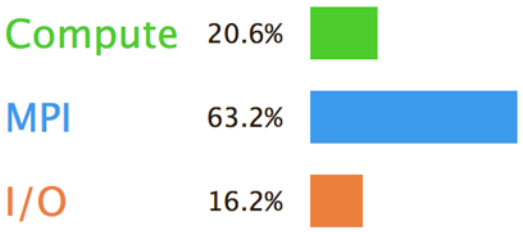
Arm Performance Reports

A high-level view of application performance with “plain English” insights

arm PERFORMANCE REPORTS

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



Time spent running application code. High values are usually good.
This is **very low**; focus on improving MPI or I/O performance first

Time spent in MPI calls. High values are usually bad.
This is **high**; check the MPI breakdown for advice on reducing it

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:

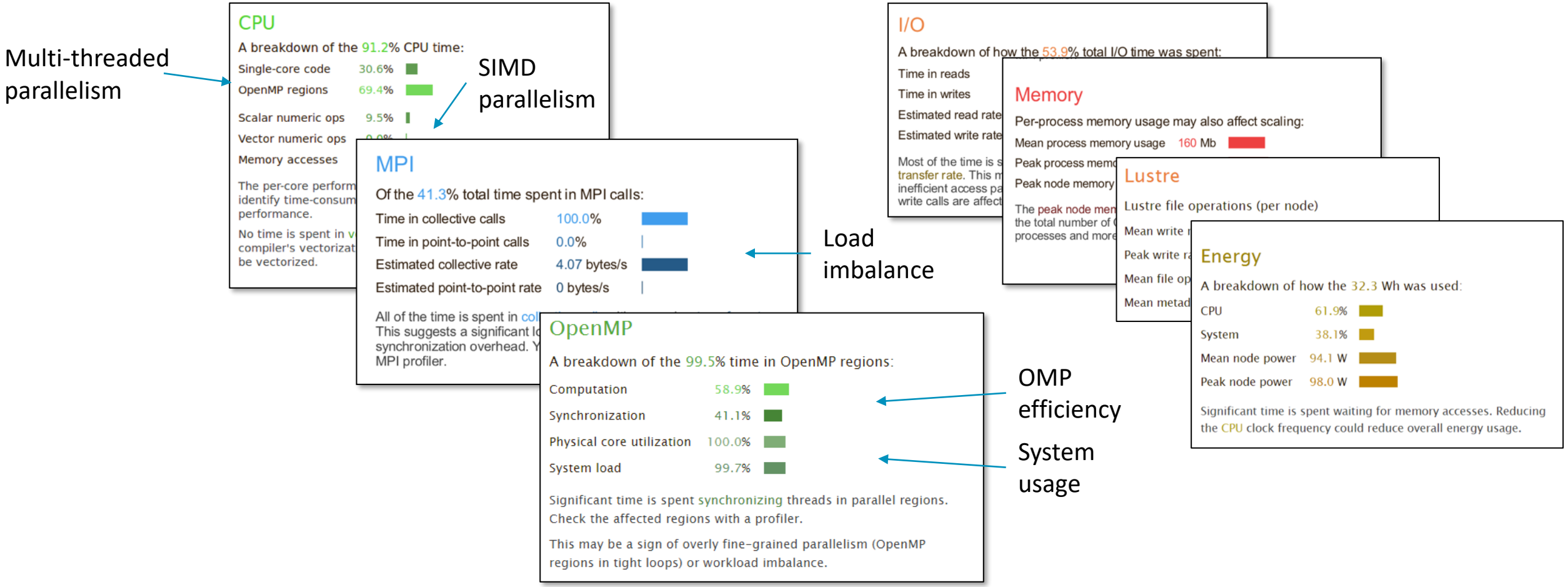
Time in reads	0.0%
Time in writes	100.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	1.38 MB/s

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.

© 2021 Arm
arm

Arm Performance Reports Metrics

Lowers expertise requirements by explaining everything in detail right in the report.



© 2021 Arm

Arm MAP + Performance Reports Cheat Sheet

Generate profiles and view offline

- Load the environment module
  - $ module load **armforge**
  - $ spack load arm**-**forge

- Compile with “-g”
  - $ mpicc **-O -g** myapp.c -o myapp.exe
  - $ mpfort **-O -g** myapp.f -o myapp.exe
- Run Arm MAP in “profile” mode
  - $ **map --profile** mpirun ./myapp.exe arg1 arg2
- Generate a performance report:
  - $ **perf-report** <profile_file>.map
- View performance report or view profile in MAP:
  - Load the corresponding file using the remote client connected to the remote system or locally

© 2021 Arm

MAP command line options

```
[johlin02@ip-10-0-0-135 Profiling]$ map --help
Arm Forge 21.0.2 - Arm MAP

Version Information: 21.0.2
The build date is Apr 27 2021 13:34:00
MAP provides an elegant graphical user interface for starting and profiling MPI
programs.

Usage: map [OPTION...] [PROGRAM [PROGRAM_ARGS]]
       map [OPTION...] (mpirun|mpiexec|aprun|...) [MPI_ARGS] PROGRAM
[PROGRAM_ARGS]
Start MAP and use it to select and start an MPI program for profiling.
```


© 2021 Arm

Performance Reports command line options

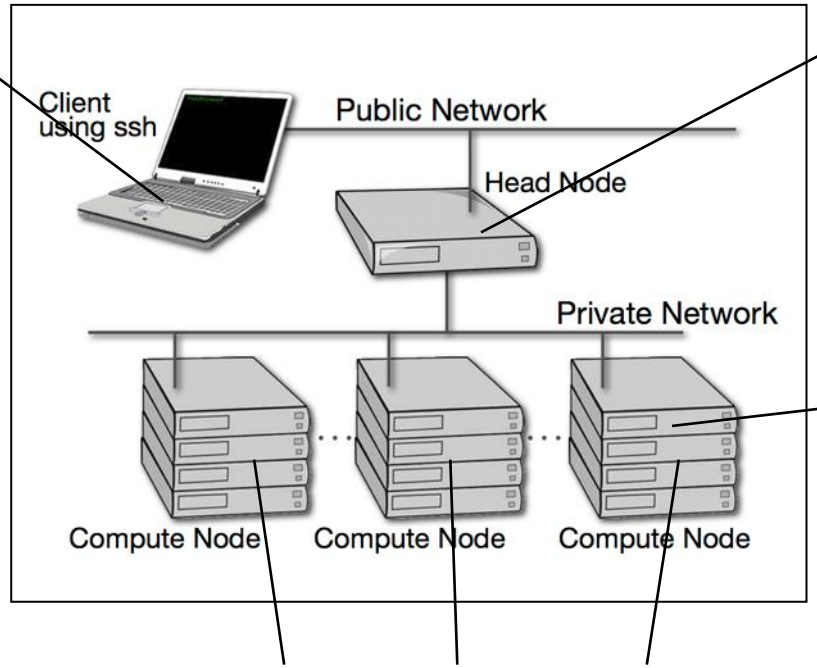
```
[johlin02@ip-10-0-0-135 Profiling]$ perf-report --help
Arm Forge 21.0.2 - Arm Performance Reports

Version Information: 21.0.2
The build date is Apr 27 2021 13:35:30

Run and monitor an MPI program; a performance report file measuring time spent in
MPI, I/O and CPU with breakdowns and tuning advice is written to the
current directory in both .html and .txt formats.

Usage: perf-report [OPTION...] PROGRAM [PROGRAM_ARGS]
       perf-report [OPTION...] (mpirun|mpiexec|aprun|...) [MPI_ARGS] PROGRAM
[PROGRAM_ARGS]
       perf-report [OPTION...] MAP_FILE
```


© 2021 Arm

The Forge GUI and where to run it

DDT and MAP provide powerful GUIs that can be run in a variety of configurations.

Remote client
(remote launch + reverse connect)



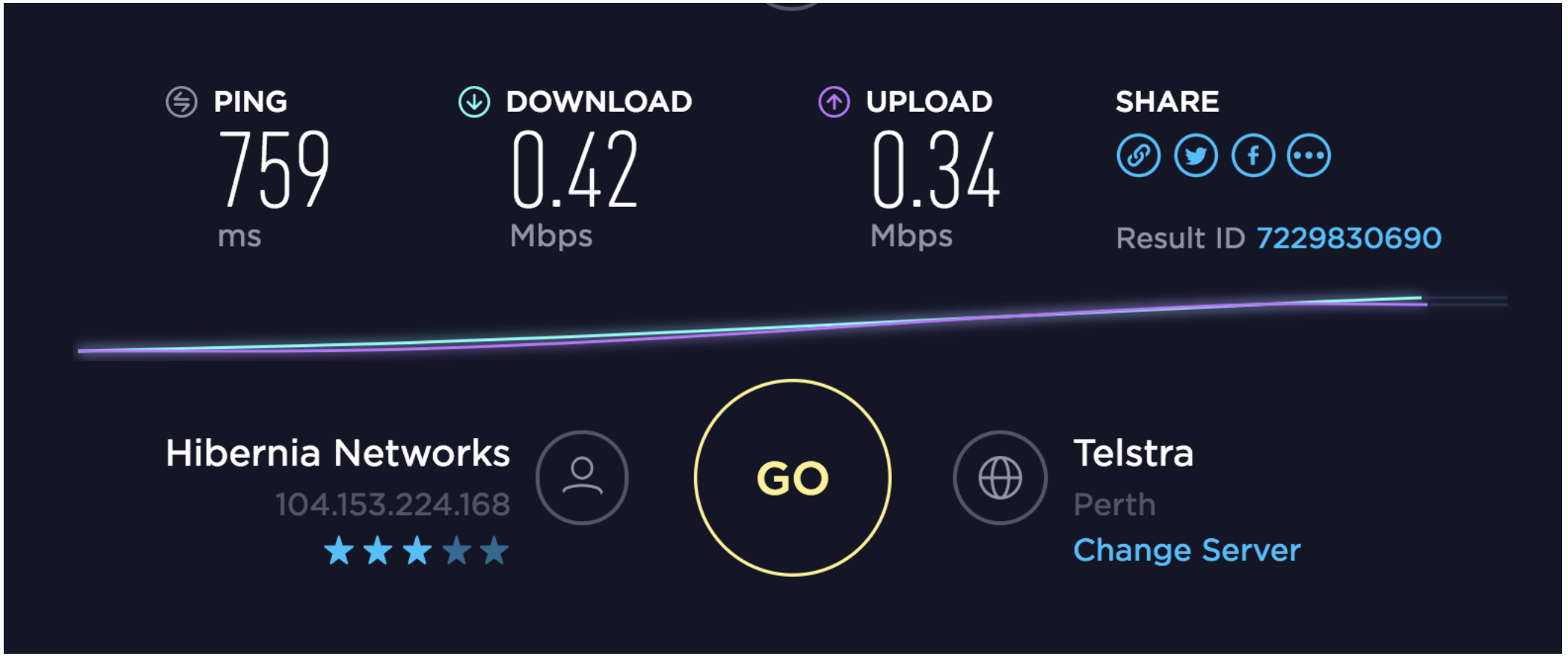
On the head node
(interactive mode + reverse connect)

On the compute node
(offline OR interactive mode)

Ultimately, that's where the tools will run.
But what about the GUI?

© 2021 Arm

DDT somewhere over the Pacific at 41,000ft and 550MPH



© 2021 Arm

Launching the Forge Remote Client

The remote client is a stand-alone application that runs on your local system

Install the Arm Remote Client (Linux, macOS, Windows)

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

Connect to the cluster with the remote client

- Open Forge Remote Client
- Create a new connection: Remote Launch ➔ Configure ➔ Add
  - Hostname: <username>@<hostname>
  - Remote installation directory: </path/to/arm-forge/X.Y/>
- Connect!

© 2021 Arm

Remote connect

arm FORGE

arm DDT

arm MAP

RUN
Run and debug a program.

ATTACH
Attach to an already running program.

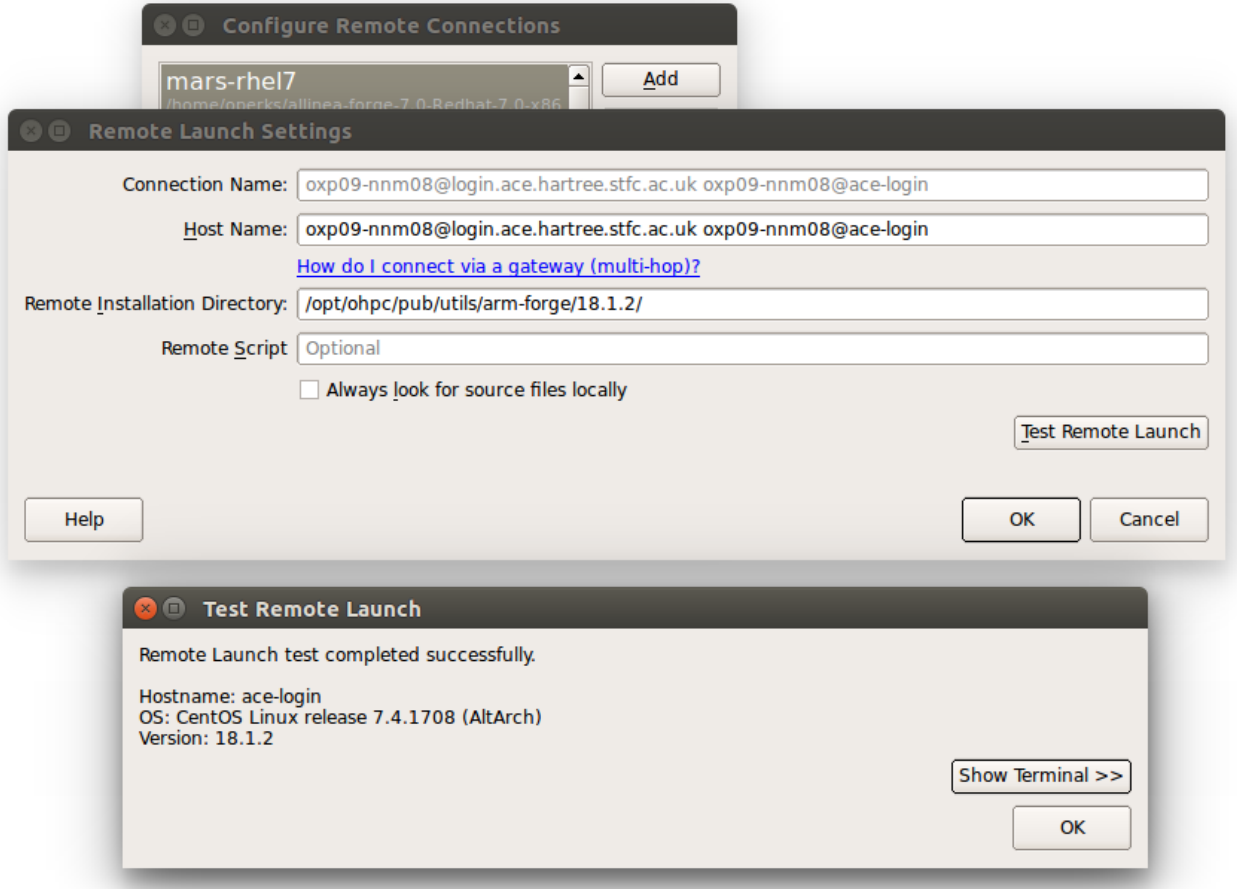
OPEN CORE
Open a core file from a previous run.

MANUAL LAUNCH (ADVANCED)
Manually launch the backend yourself.

OPTIONS

Remote Launch:
Configure...

QUIT



© 2021 Arm

arm

Wrap up

- Use a profiler
- Talk to the experts on #mentors-conclave
- Review the profiling with ReFrame guides
  - Cloud-HPC-Hackathon-2021/Guides at main · arm-hpc-user-group/Cloud-HPC-Hackathon-2021
- Start with MAP to get a high-level profile.
  - You might not need more detail than this
- If you need more detail, focus on one area of interest / aspect of performance
  - Use a detailed profiler, e.g. “spack install scorep”

- Oh! And keep reading for a case study of I/O optimization with Arm MAP

© 2021 Arm

arm

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

© 2021 Arm

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks

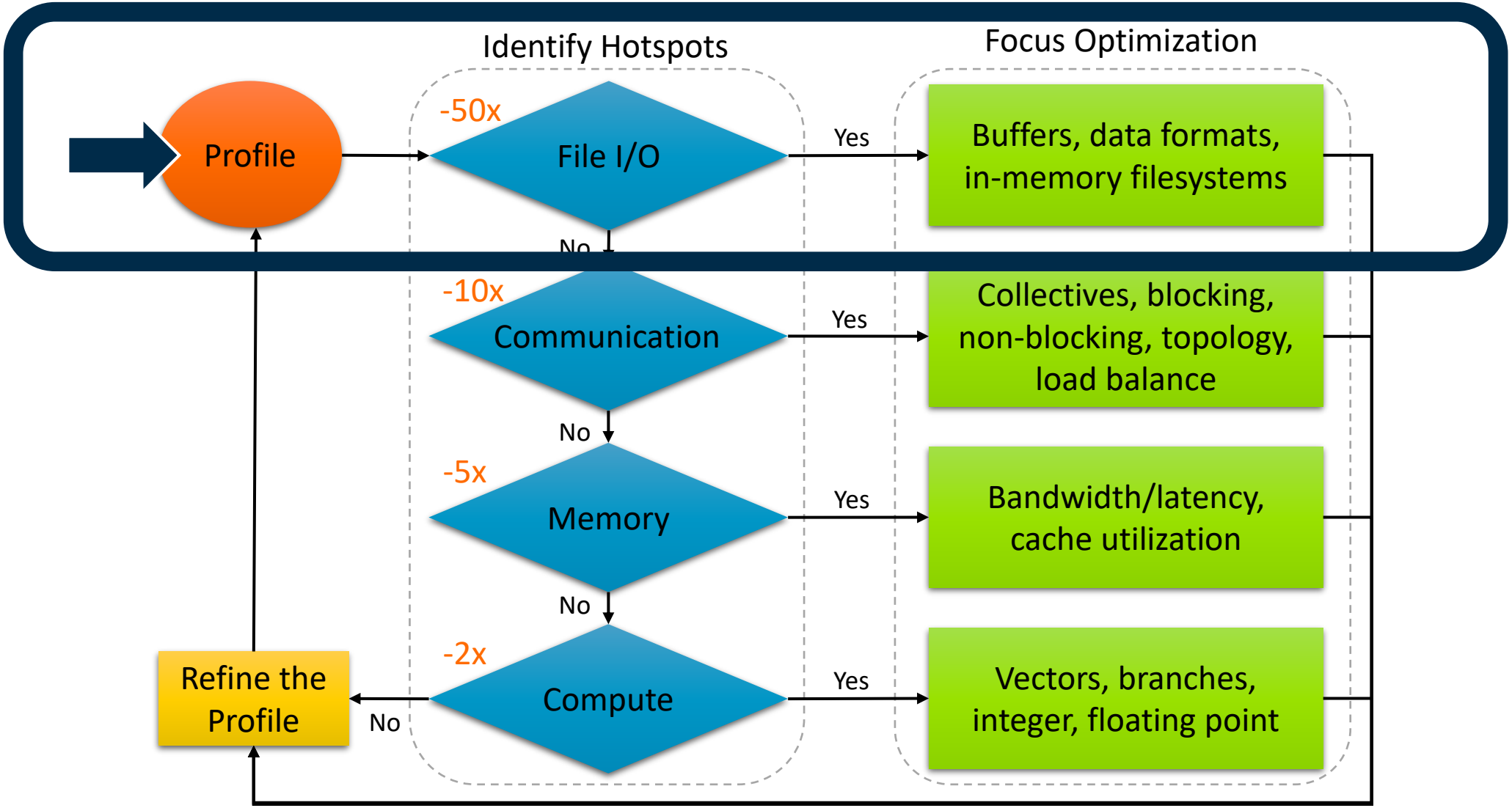
© 2021 Arm

arm

I/O Optimization

© 2021 Arm

Identifying and resolving performance issues

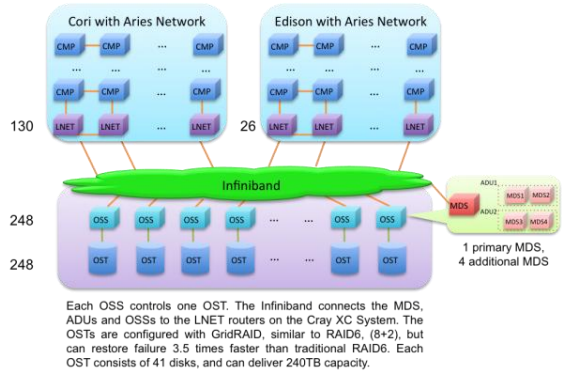


© 2021 Arm

Why does I/O have such a huge impact on performance?

I/O has the potential to make or break the performance of the whole system.

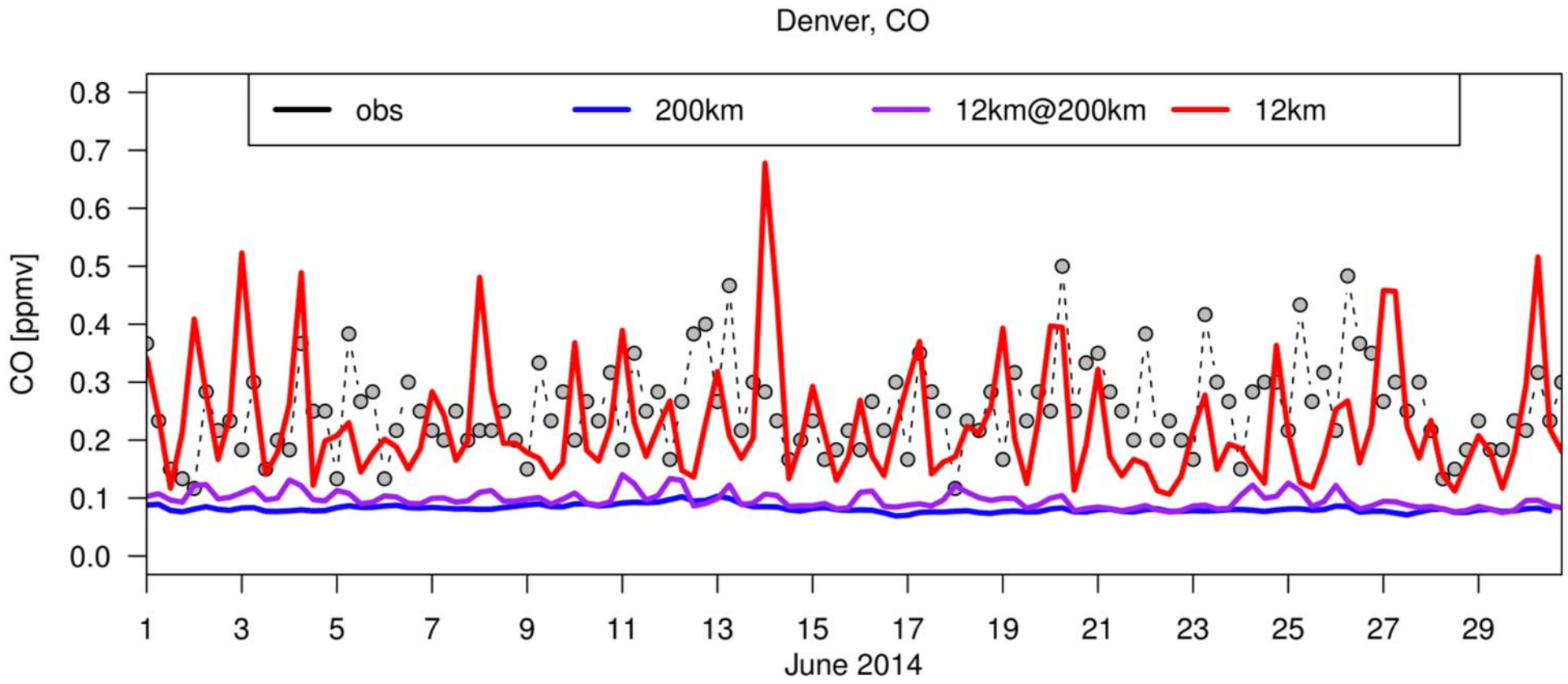
- A shared resource on practically all HPC systems.
  - Bandwidth to disk is shared between processes.
  - Bandwidth to network is shared between nodes.
- Has the potential to affect the performance of other users' jobs.
  - Data are physically located outside the compute node.
  - Using shared I/O outside the compute node has an impact on the performance of other users' jobs.
  - Even if other users are not using the shared filesystem, communicating with the filesystem over the network can affect other user’s inter-node communications (e.g. MPI).
- The slowest tier of the memory hierarchy.
  - Small mistakes in I/O can cost more than huge mistakes on-chip
  - Simple, low effort optimizations in I/O will pay out more than high effort optimizations on-chip.



© 2021 Arm

Reduction isn’t an option: have to optimize I/O

Models require high resolutions to accurately describe physical conditions.



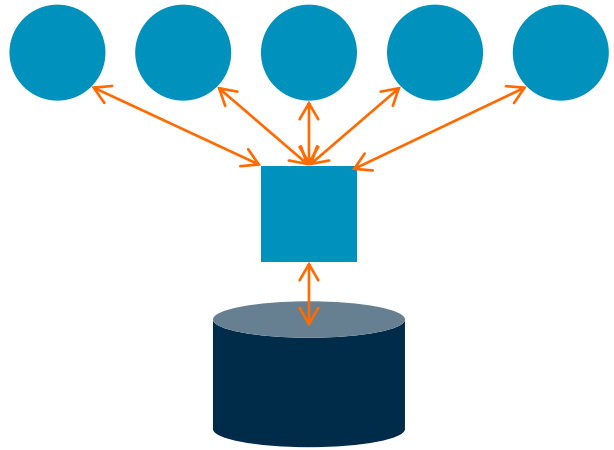
Credit: NASA GMAO, Christoph Keller.

© 2021 Arm

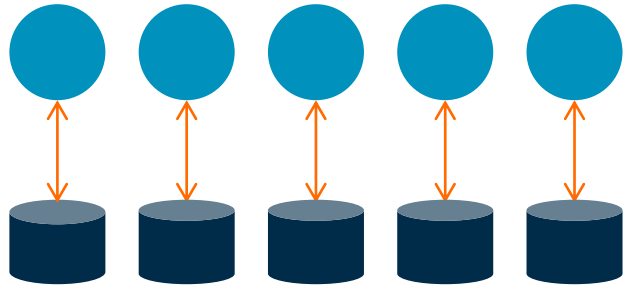
Simple approaches to parallel I/O

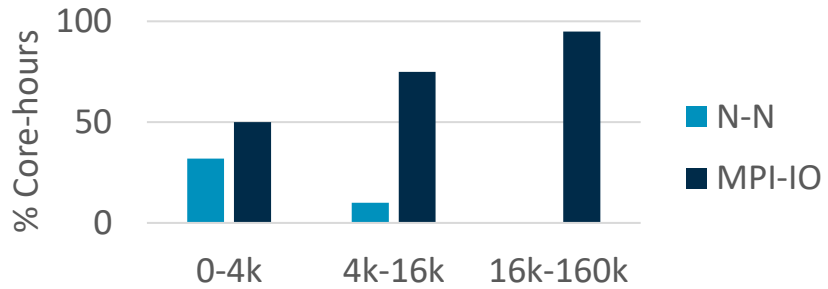
Simple approaches work for small applications, but typically don't scale.

- 1 – 1: Master and workers
  - A master process performs I/O on behalf of many workers.
  - Collective operations (e.g. MPI_Gather, MPI_Scatter) move data to/from workers.
  - Performance bottleneck at the master.



- N – N: Every process for itself
  - Each process reads/writes it's own data in a uniquely named file.
  - Large number of open files can quickly degrade performance.





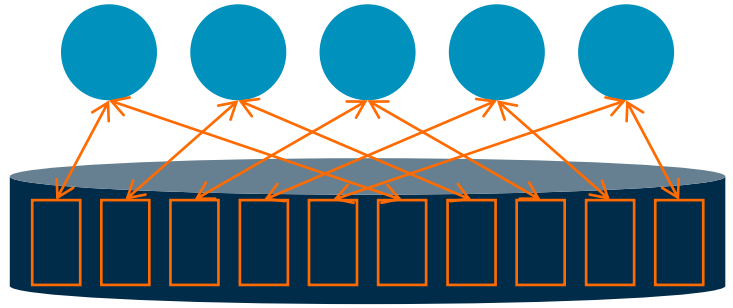
Credit: Argonne National Lab

© 2021 Arm

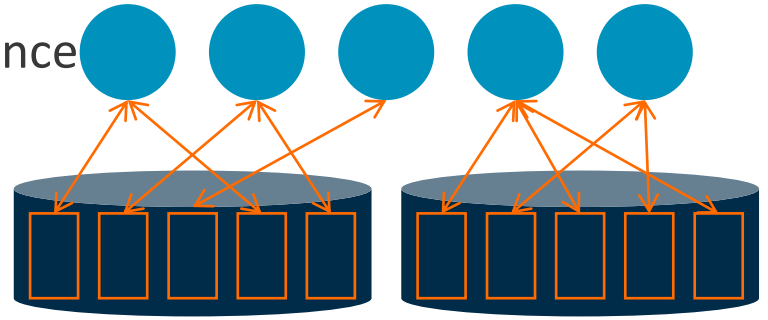
Treating parallel I/O like shared memory

Use a library like MPI-IO or HDF5 for optimal portability and performance.

- N – 1: Multiple writers to same resource
  - Many processes read/write to the same resource, e.g. a file.
  - Files broken up in to lock units; boundaries determined by system.
  - Clients must obtain locks before performing I/O.
  - Enables caching: as long as client holds the lock the cache is valid.



- N – M: Cooperating gangs
  - Groups of processes combine to operate on shared resources.
  - Mirroring physical hardware infrastructure can improve performance
  - Implementation best left to the libraries.
  - Balance gang size against available bandwidth.



© 2021 Arm

Understand your I/O system

Use portable, cross-platform tools and libraries.

- Storage systems host filesystems
  - Lustre, GPFS, BeeGFS: POSIX-compliant block storage designed for scalability.
  - Ceph: Object storage, block storage, and POSIX-compliant filesystem.
- Infrastructure hosts storage systems
  - The network fabric connects all compute nodes in a predefined (physically hard wired) topology.
  - I/O nodes serve multiple compute nodes (potential bottleneck)
- Infrastructure can be optimized for HPC
  - Small local (i.e. non-shared) filesystems, possibly in memory (e.g. /dev/shm)
  - Burst buffers
  - NVDIMMS.

© 2021 Arm

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 world's top supercomputers
- 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)

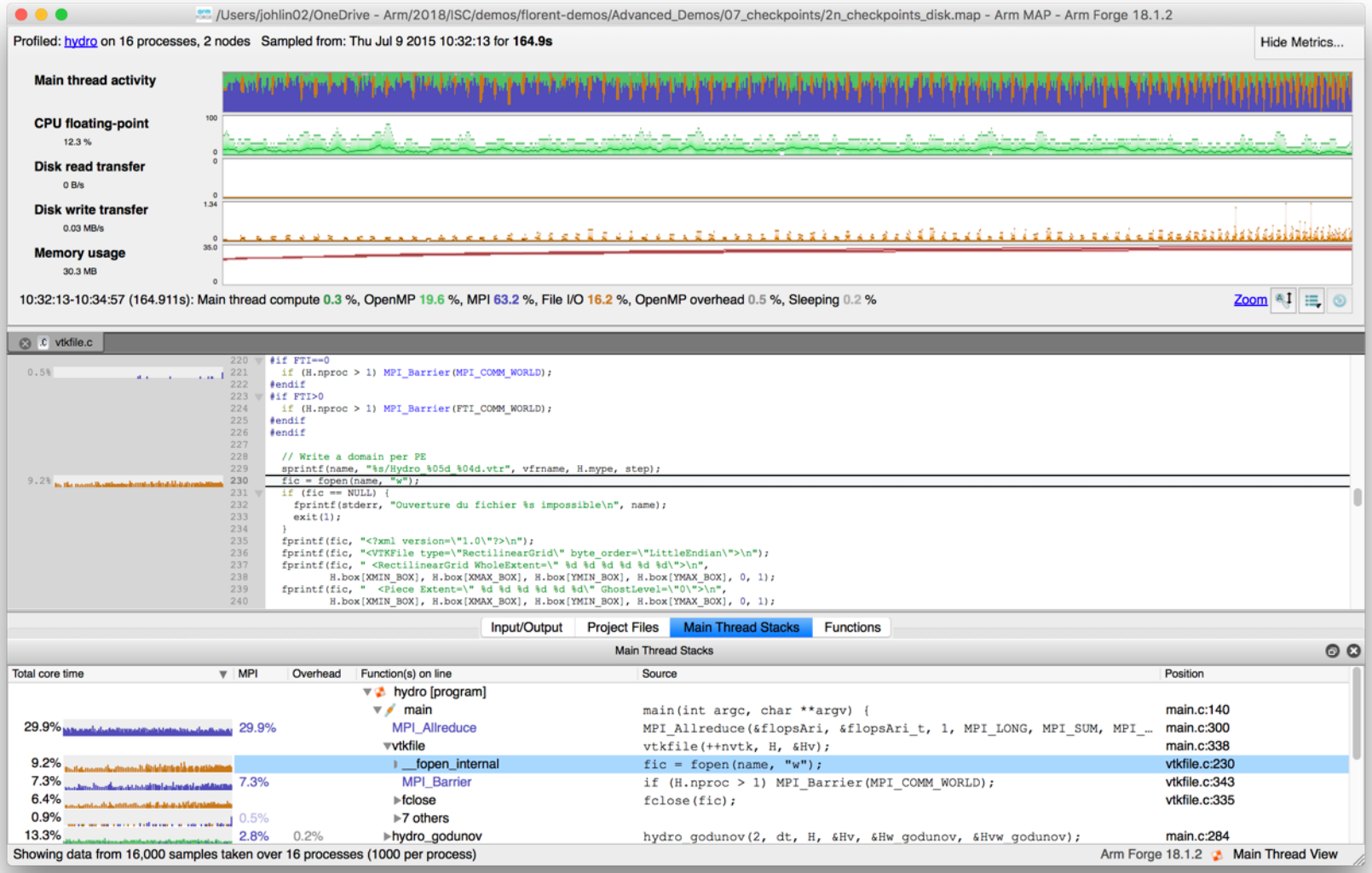
Easy to use by everyone

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

© 2021 Arm

Initial profile shows 9.2% of runtime spent just opening files

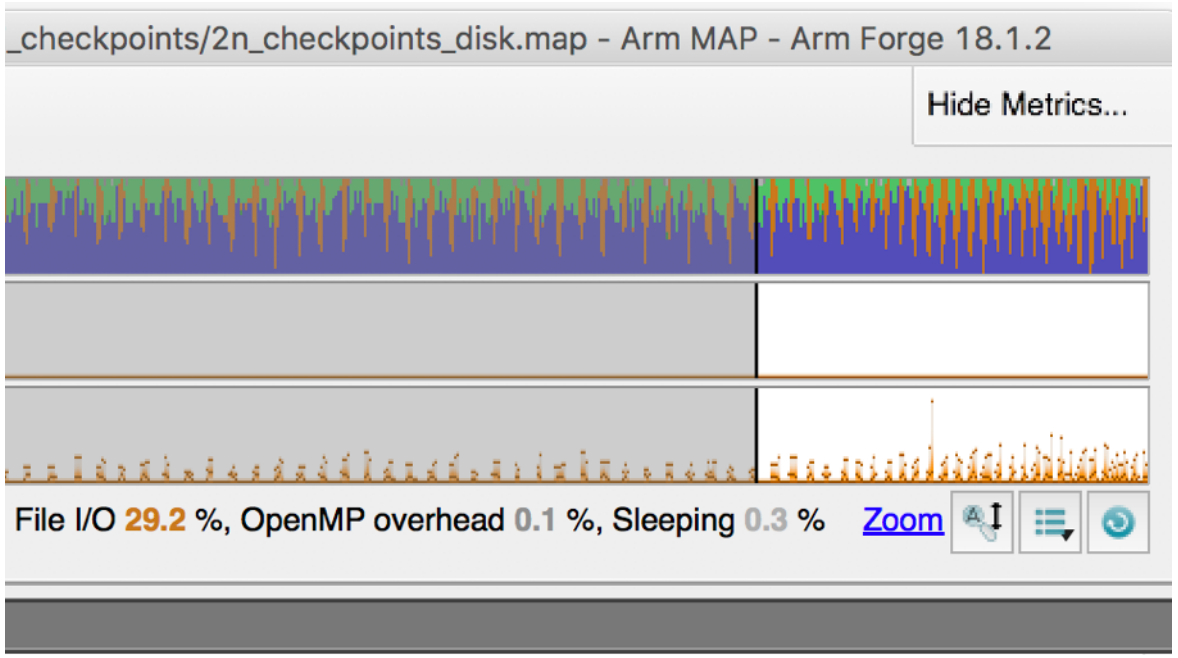
16.2% of runtime is I/O, but only 5% is spent in read/write operations.

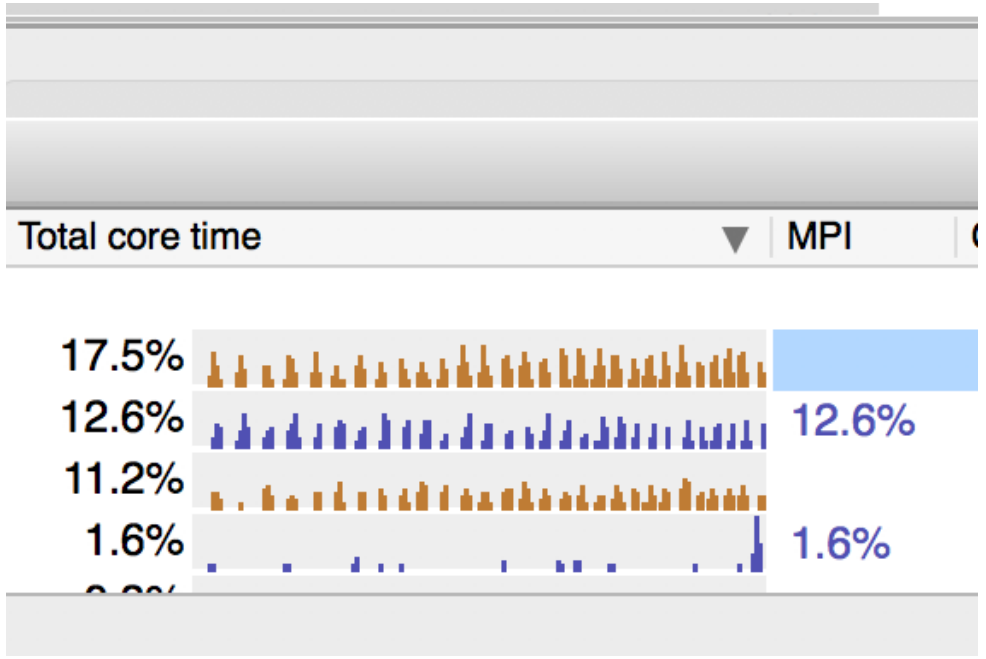


© 2021 Arm

Almost 30% of hotspot runtime is I/O

File open and close operations are very expensive on this filesystem.



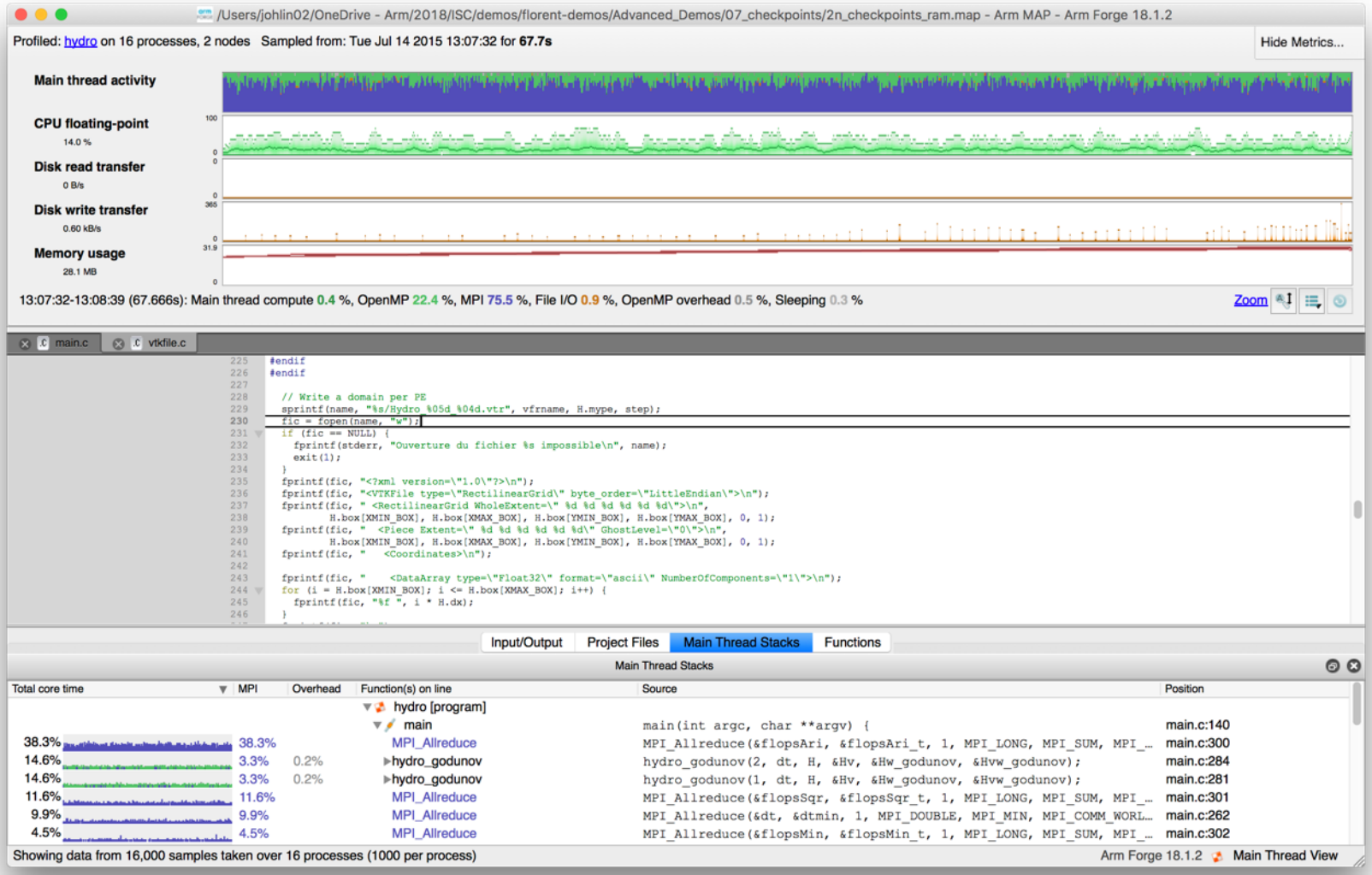


- Intermediate files for visualization are being written to disk.
- Fix: write intermediate files to an in-memory filesystem, e.g. /dev/shm.

© 2021 Arm
arm

Easy fix: write intermediate files to /dev/shm

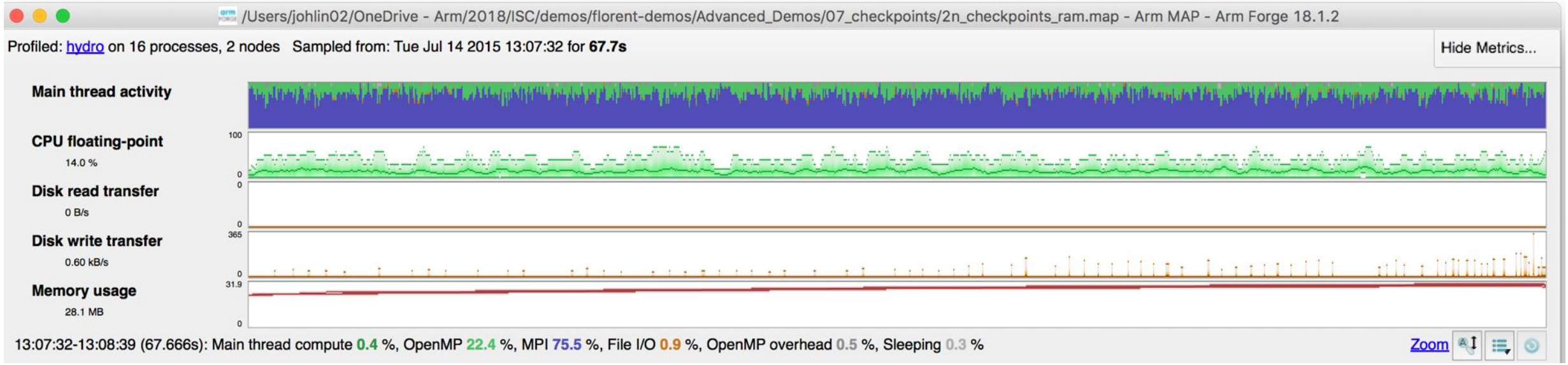
Writing temporary files to in-memory filesystem can dramatically improve performance.



© 2021 Arm

After fix, only 0.9% of runtime spent in I/O

Writing temporary files to in-memory filesystem can dramatically improve performance.



Total core time	MPI	Overhead	Function(s) on line
			hydro [program]
			main
38.3%	38.3%		MPI_Allreduce
14.6%	3.3%	0.2%	hydro_godunov
14.6%	3.3%	0.2%	hydro_godunov
11.6%	11.6%		MPI_Allreduce
9.9%	9.9%		MPI_Allreduce
4.5%	4.5%		MPI_Allreduce

Showing data from 16,000 samples taken over 16 processes (1000 per process)

© 2021 Arm

Arm Performance Reports

High-level view of application performance shows low write rate.

arm PERFORMANCE REPORTS

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.

© 2021 Arm

After the fix, write rate has improved 41.6x

Eliminating file open/close bottleneck has dramatically improved I/O performance.

arm PERFORMANCE REPORTS

Command:	mpiexec.hydra -host node-1,node-2 -map-by socket -n 16 -ppn 8 ./Bin/../Src//hydro -i ./Bin/../../../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:	Tue Jul 14 2015 13:07:32
Total time:	68 seconds (about 1 minutes)
Full path:	Src

Summary: hydro is MPI-bound in this configuration

Compute	23.5%	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	75.5%	Time spent in MPI calls. High values are usually bad. This is **very high**; check the MPI breakdown for advice on reducing it
I/O	0.9%	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

I/O

A breakdown of the 0.9% I/O time:

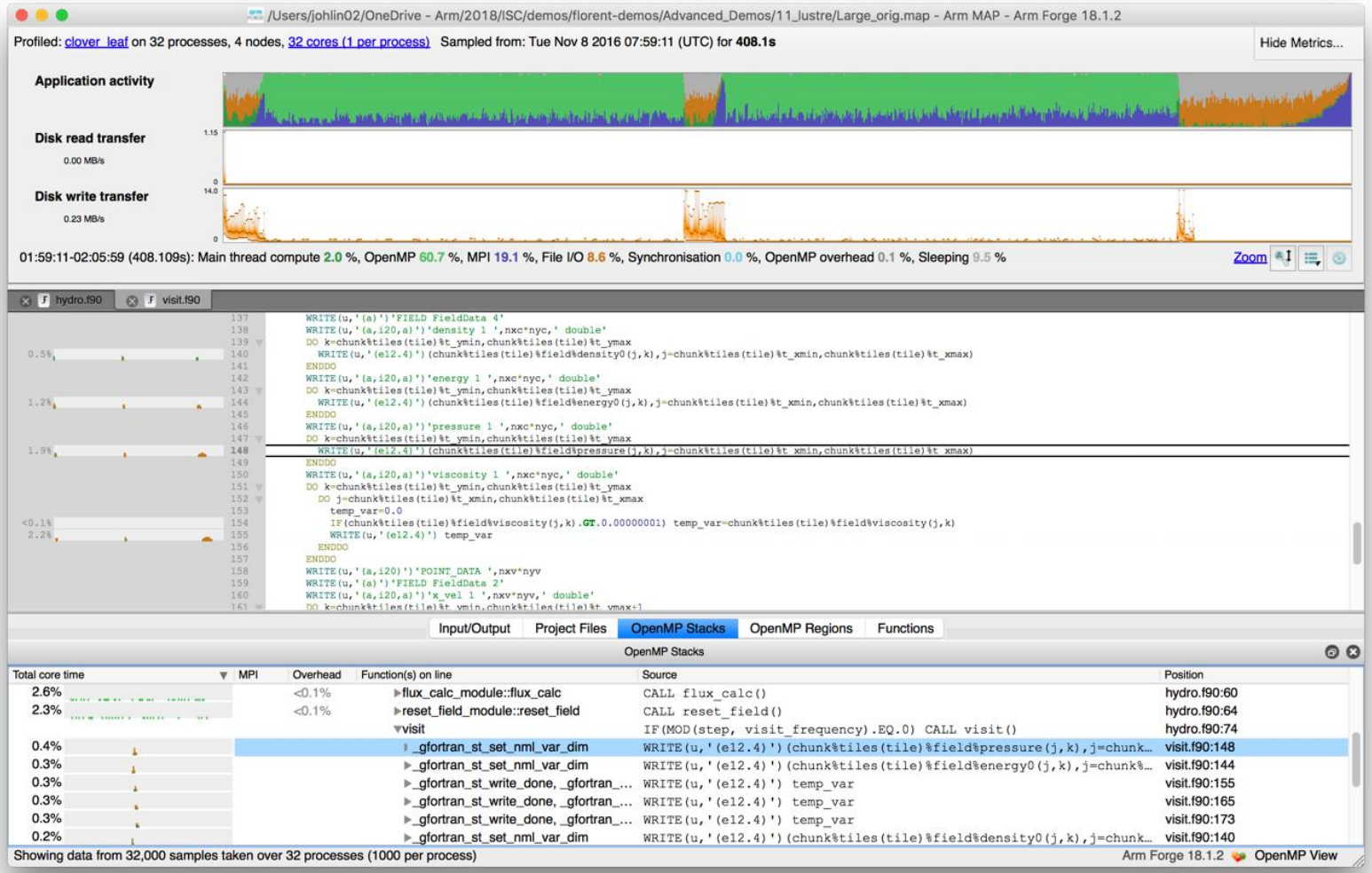
Time in reads	0.0%
Time in writes	100.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	57.5 MB/s

Most of the time is spent in write operations with a 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.

© 2021 Arm

Initial profile of CloverLeaf shows surprisingly unequal I/O

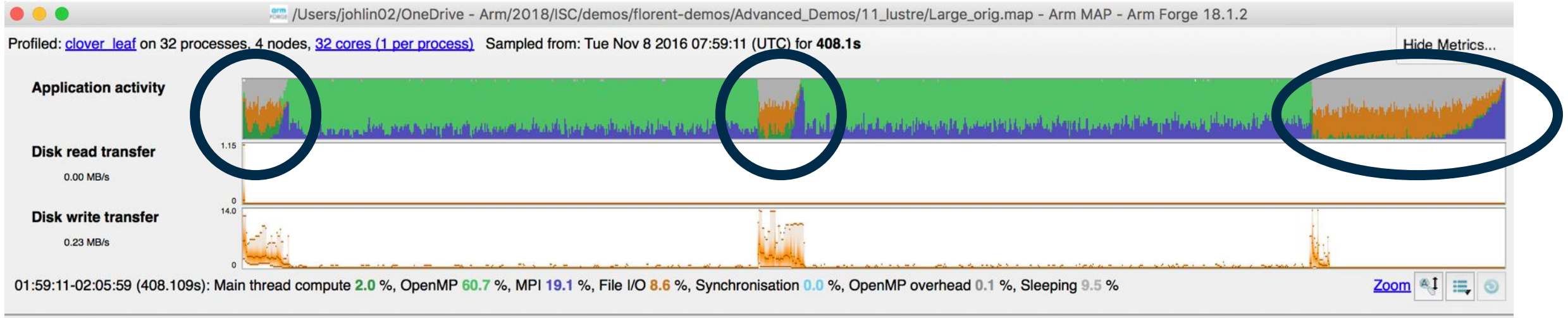
Each I/O operation should take about the same time, but it’s not the case.



© 2021 Arm

Symptoms and causes of the I/O issues

Sub-optimal file format and surprise buffering.

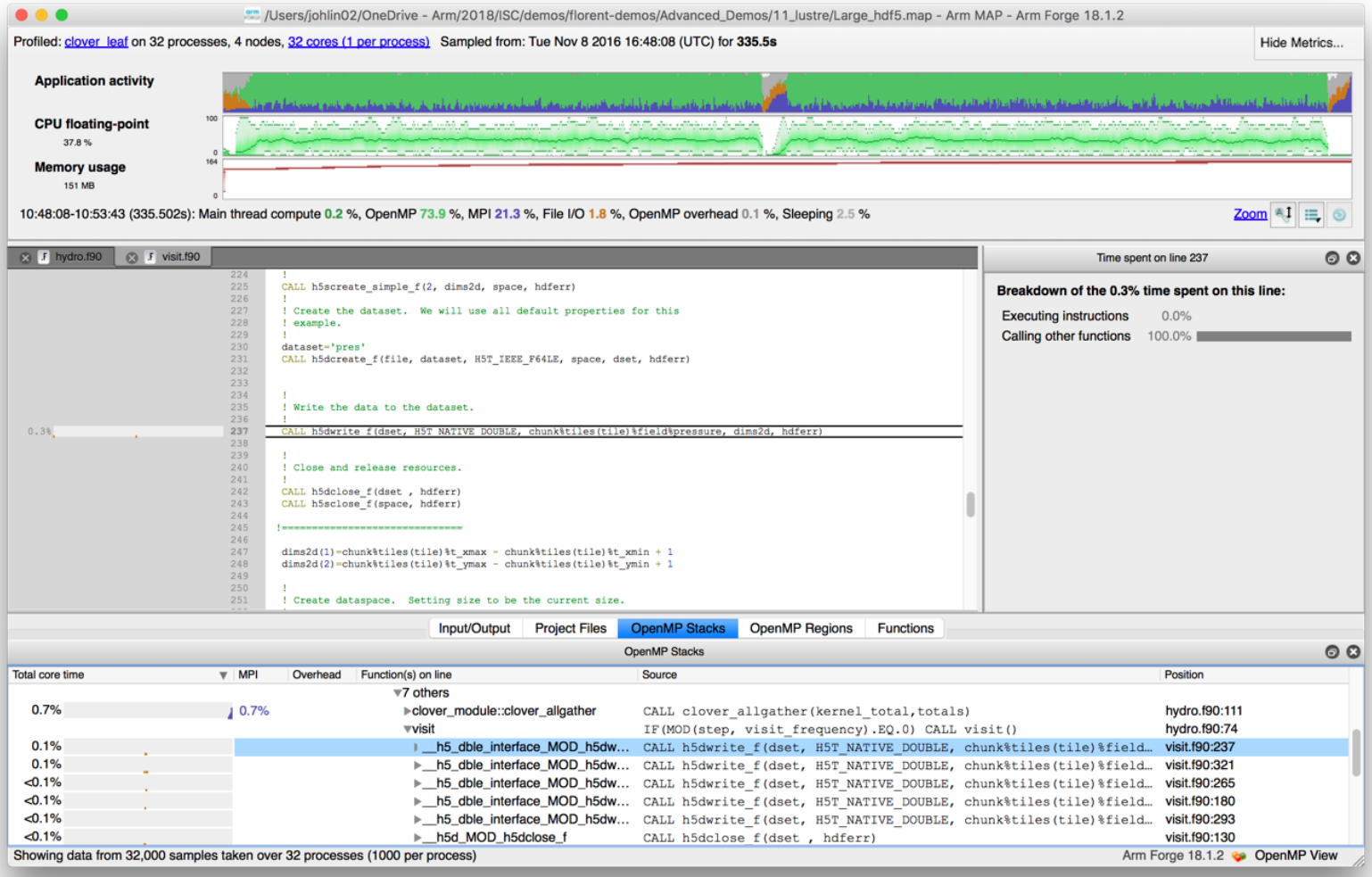


- Write rate is less than 14MB/s.
- Writing an ASCII output file.
- Writes not being flushed until buffer is full.
  - Some ranks have much less buffered data than others.
  - Ranks with small buffers wait in barrier for other ranks to finish flushing their buffers.

© 2021 Arm
arm

Solution: use HDF5 to write binary files

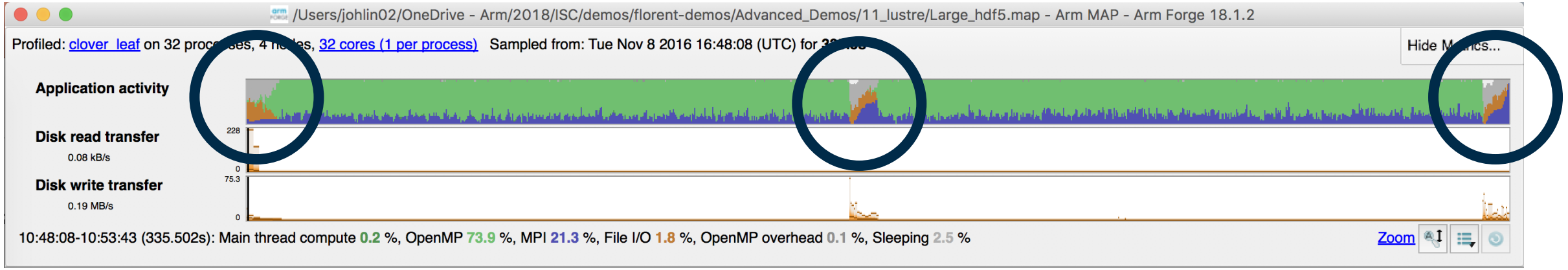
Using a library optimized for HPC I/O improves performance and portability.



© 2021 Arm

Solution: use HDF5 to write binary files

Using a library optimized for HPC I/O improves performance and portability.



- Replace Fortran write statements with HDF5 library calls.
  - Binary format reduces write volume and can improve data precision.
  - Maximum transfer rate now 75.3 MB/s, over 5x faster.
- Note MPI costs (blue) in the I/O region, so room for improvement.

© 2021 Arm