Improving your code with Arm Forge

PDC-PRACE Workshop
"HPC Tools for the Modern Era"

Conrad Hillairet
conrad.hillairet@arm.com

An introduction to Arm

Arm is the world's leading semiconductor intellectual property supplier

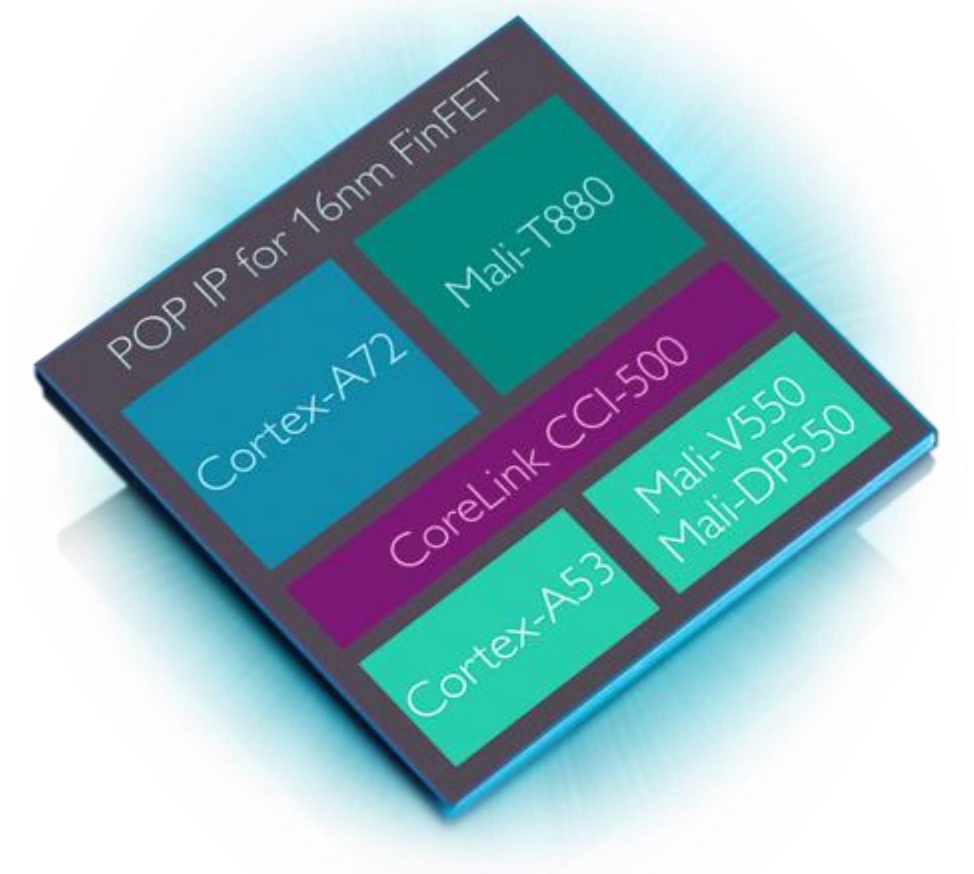
We license to over 350 partners: present in 95% of smart phones, 80% of digital cameras, 35% of all electronic devices, and a total of 60 billion Arm cores have been shipped since 1990

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



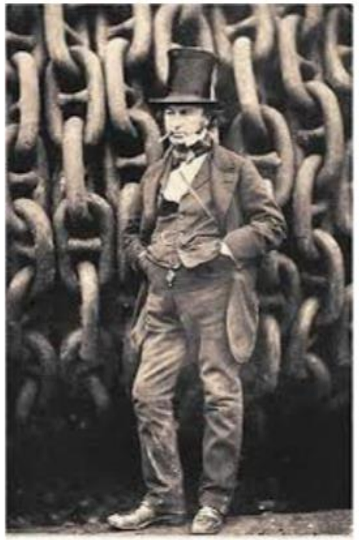
…and our IP extends beyond the CPU

Early HPC deployments

University of BRISTOL | Isambard | GW4

Isambard system specification (red = new info):

- Cray "Scout" system – XC50 series
  - Aries interconnect
- **10,000+** Armv8 cores
  - Cavium ThunderX2 processors
  - 2x 32core @ >2GHz per node
- Cray software tools
- Technology comparison:
  - x86, Xeon Phi, Pascal GPUs
- Phase 1 installed March 2017
- The Arm part arrives early 2018



I.K.Brunel 1804-1859

@simonmcs http://gw4.ac.uk/isambard/ 5 bristol.ac.uk

arm

Catalyst UK

Accelerating Arm adoption in the UK

Sites and Target HPC Applications:

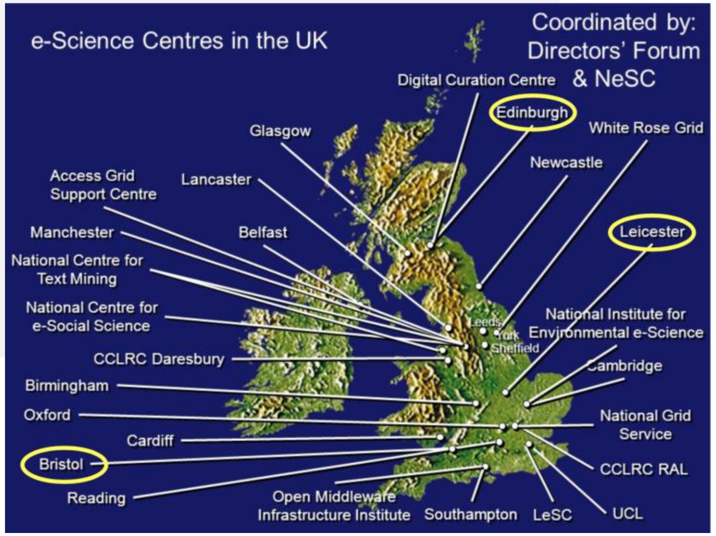
– **EPCC**: WRF, OpenFOAM, Rolls Royce Hydra opt, 2 PhD candidates

– **Leicester**: Data-intensive apps, genomics, MOAB Torque, DiRAC collab

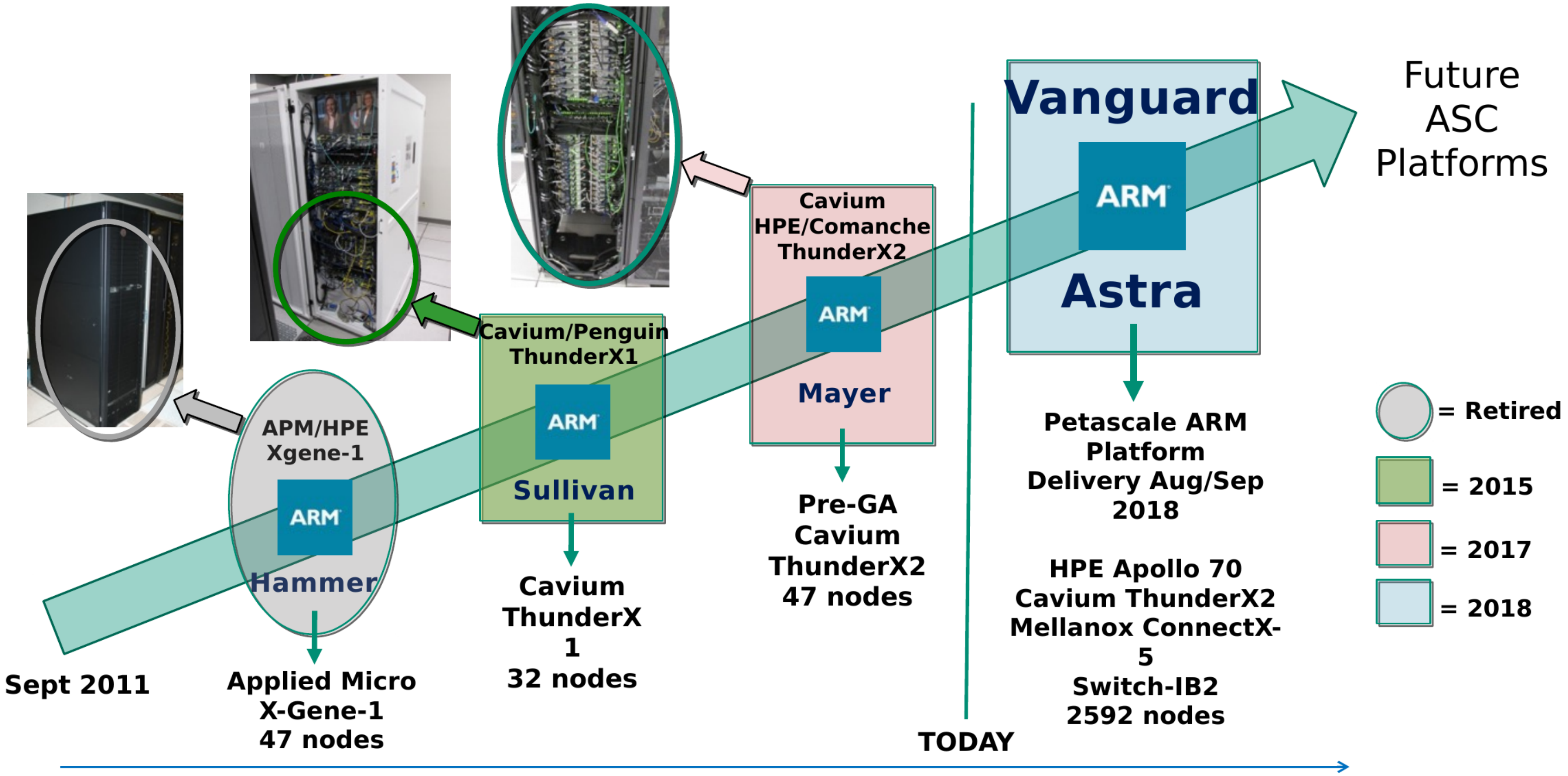
– **Bristol**: VASP, CASTEP, Gromacs, CP2K, Unified Model, Hydra, NAMD, Oasis, NEMO, OpenIFS, CASINO, LAMMPS

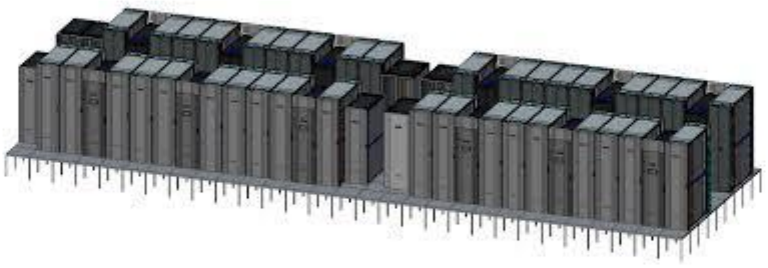
Typical Cluster for each site:

- 64 x Apollo 70 Compute Nodes (2 racks):
  - Dual socket Cavium 32c, 2.2 GHz
  - 256GB memory (16GB DIMMs)
  - Mellanox IB EDR CX5 Clos
  - 4096+ cores



Astra



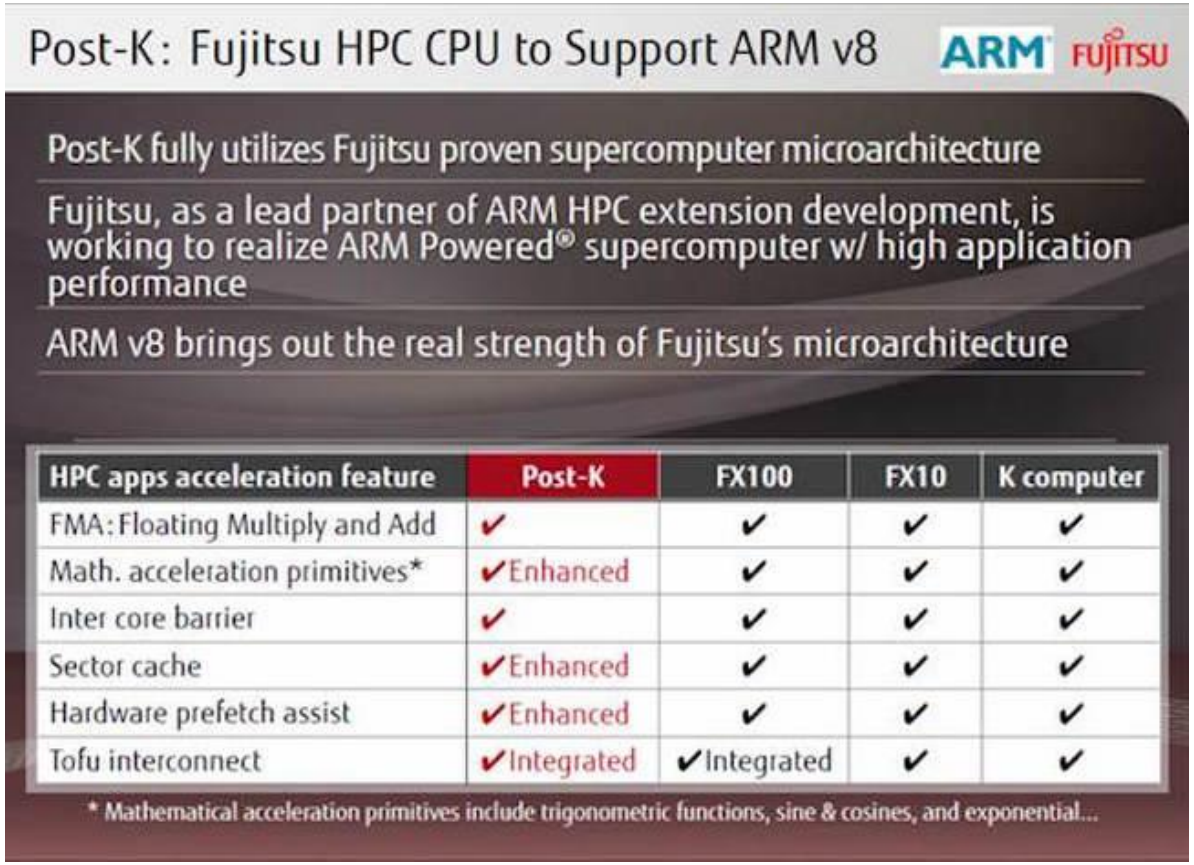


Beskow 2.43 petaflops (source)

Astra 2.32 petaflops (source)

arm

Japan



Post-K: Fujitsu HPC CPU to Support ARM v8

ARM FUJITSU

Post-K fully utilizes Fujitsu proven supercomputer microarchitecture

Fujitsu, as a lead partner of ARM HPC extension development, is working to realize ARM Powered® supercomputer w/ high application performance

ARM v8 brings out the real strength of Fujitsu's microarchitecture

HPC apps acceleration feature	Post-K	FX100	FX10	K computer
FMA: Floating Multiply and Add	✔	✔	✔	✔
Math. acceleration primitives*	✔Enhanced	✔	✔	✔
Inter core barrier	✔	✔	✔	✔
Sector cache	✔Enhanced	✔	✔	✔
Hardware prefetch assist	✔Enhanced	✔	✔	✔
Tofu interconnect	✔Integrated	✔Integrated	✔	✔

* Mathematical acceleration primitives include trigonometric functions, sine & cosines, and exponential...

arm

Conrad : Support Engineer - Arm Allinea Studio and Arm Forge

A quick glance at what is in Arm Allinea Studio

C++

C/C++ Compiler

AArch64

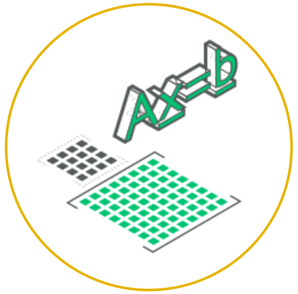
- C++ 14 support
- OpenMP 4.5 without offloading
- SVE ready

Fortran

Fortran Compiler

AArch64

- Fortran 2003 support
- Partial Fortran 2008 support
- OpenMP 3.1
- SVE ready



Performance Libraries

AArch64

- Optimized math libraries
- BLAS, LAPACK and FFT
- Threaded parallelism with OpenMP

Forge (DDT and MAP)

Cross Platform

- Profile, Tune and Debug
- Scalable debugging with DDT
- Parallel Profiling with MAP

Performance Reports
Cross Platform

- Analyze your application
- Memory, MPI, Threads, I/O, CPU metrics

arm

Conrad : Support Engineer - Arm Allinea Studio and Arm Forge

A quick glance at what is in Arm Allinea Studio

C++

C/C++ Compiler

AArch64

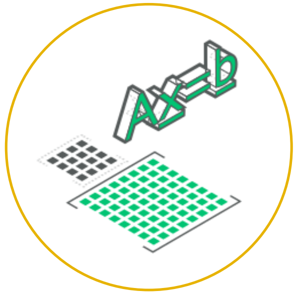
- C++ 14 support
- OpenMP 4.5 without offloading
- SVE ready

Fortran

Fortran Compiler

AArch64

- Fortran 2003 support
- Partial Fortran 2008 support
- OpenMP 3.1
- SVE ready



Performance Libraries

AArch64

- Optimized math libraries
- BLAS, LAPACK and FFT
- Threaded parallelism with OpenMP

Forge (DDT and MAP)

Cross Platform

- Profile, Tune and Debug
- Scalable debugging with DDT
- Parallel Profiling with MAP

Performance Reports
Cross Platform

- Analyze your application
- Memory, MPI, Threads, I/O, CPU metrics

arm

Summary

Summary :

Overview

Introduction

I

Arm Performance Reports
Arm MAP

Hands - On : Launch MAP
Hands - On : Launch Perf-reports

Hands - On : Vectorization

Hands - On : Workload Imbalance

III

II

Arm DDT

Hands - On : Launch DDT
Hands - On : SIGFPE
Hands - On : Memory Debugging

Overview

arm

Extra documentation

PDC Documentation : https://www.pdc.kth.se/software/software/allinea-forge/index.html

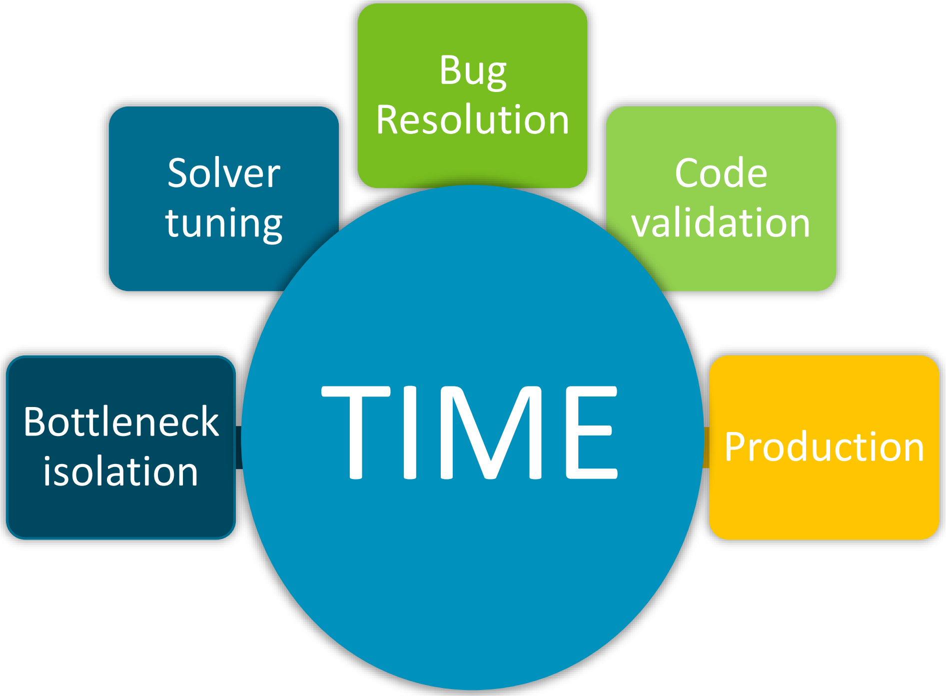
Arm DDT User Guide : https://developer.arm.com/docs/101136/latest/ddt

Arm MAP User Guide : https://developer.arm.com/docs/101136/latest/map

Arm Performance Reports User Guide : https://developer.arm.com/docs/101137/latest/introduction

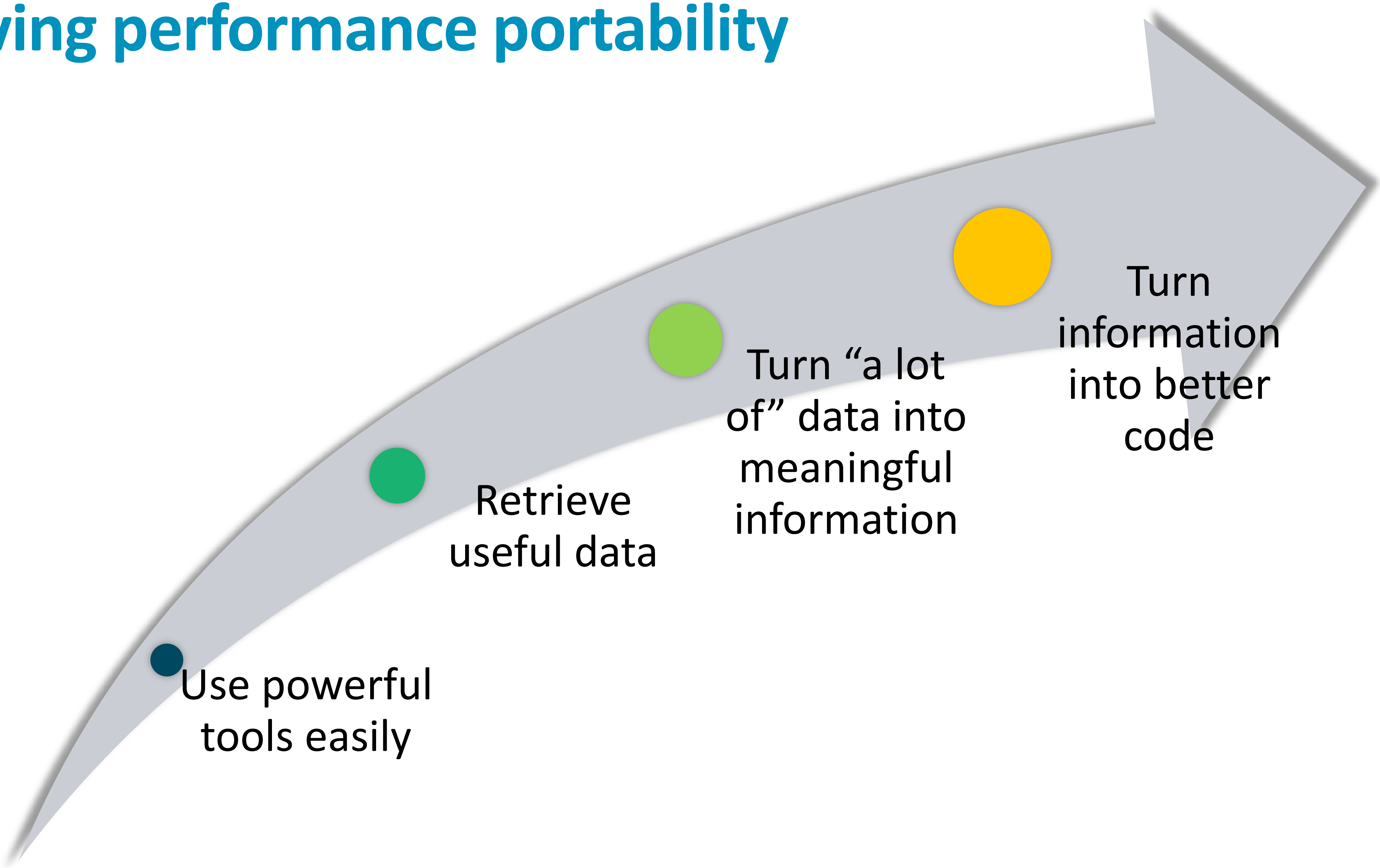
Arm Forge Webinars : https://developer.arm.com/products/software-development-tools/hpc/training/arm-hpc-tools-webinars

We do tools for a single reason: help people save their time.



arm

Achieving performance portability



Using powerful tools more easily

Remote Client

- Fast and easy alternative to X-Forwarding and VNC

Reverse Connect

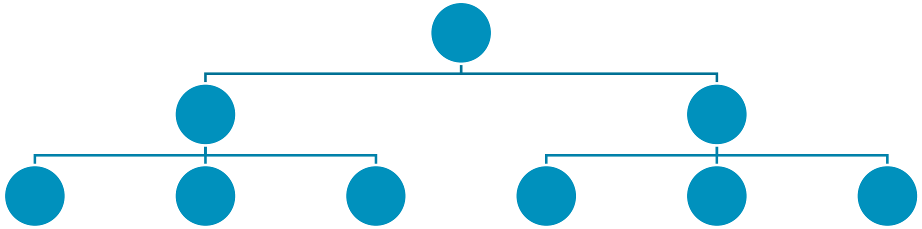
- Simplifies integration with job submission scripts

Continuous Integration

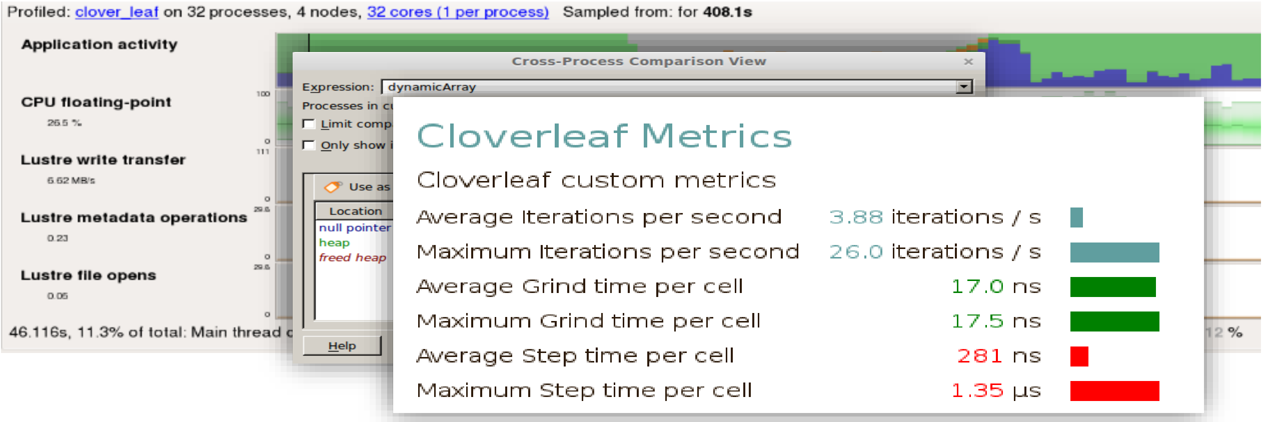
- Automation of debugging & profiling for professional workflows

Generating useful and meaningful information

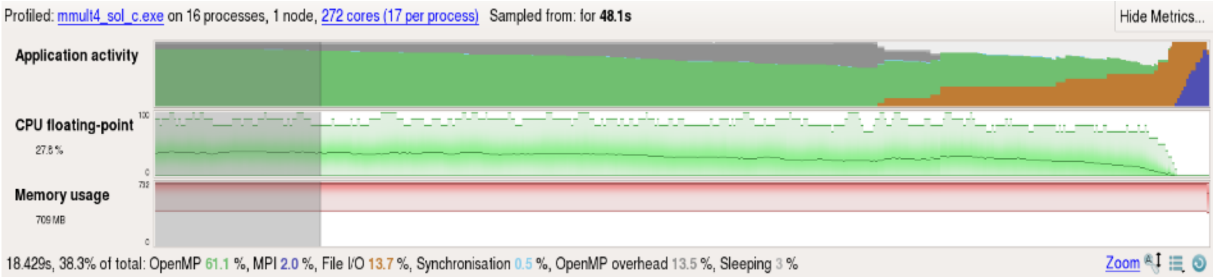
Scalable & Portable



Data collection



Data processing



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

- Most widely-used debugging and profiling suite in HPC
- Fully supported by Arm on Intel, AMD, Arm, 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

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

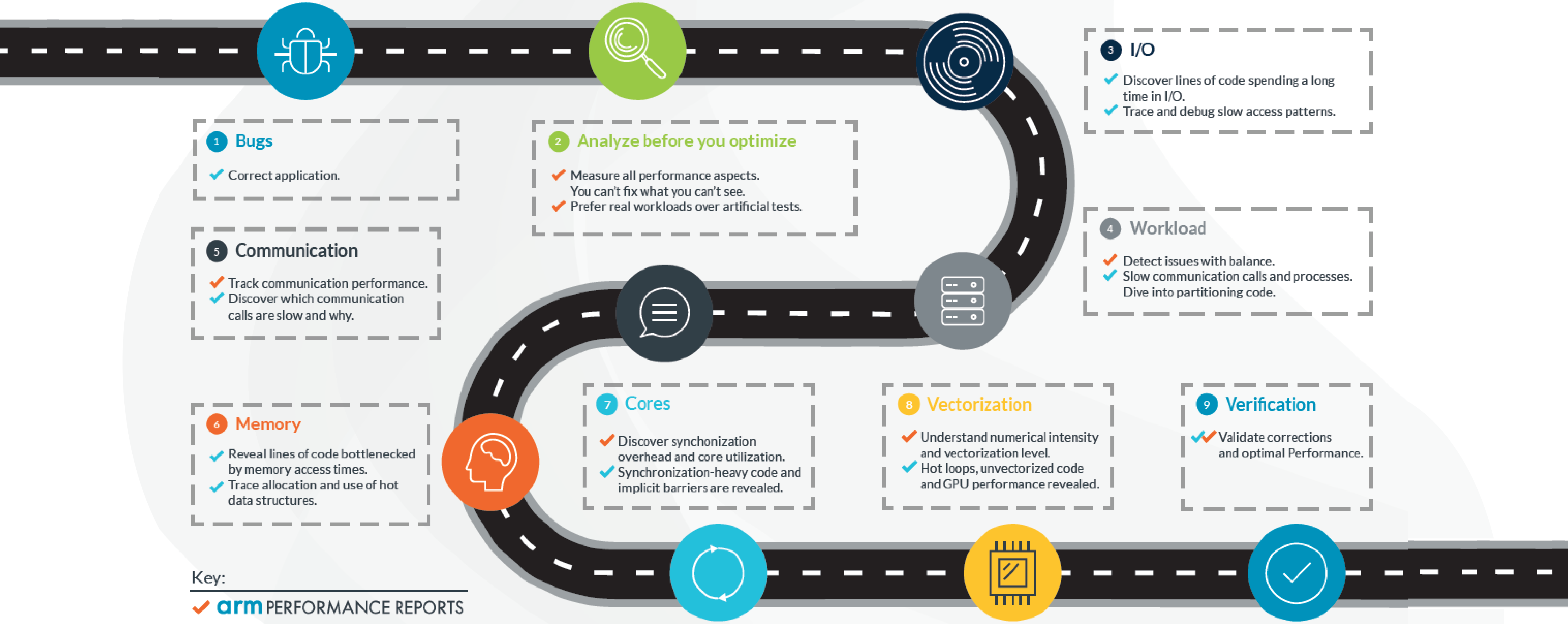
Adds value to typical users' workflows

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

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

Arm DDT

arm

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

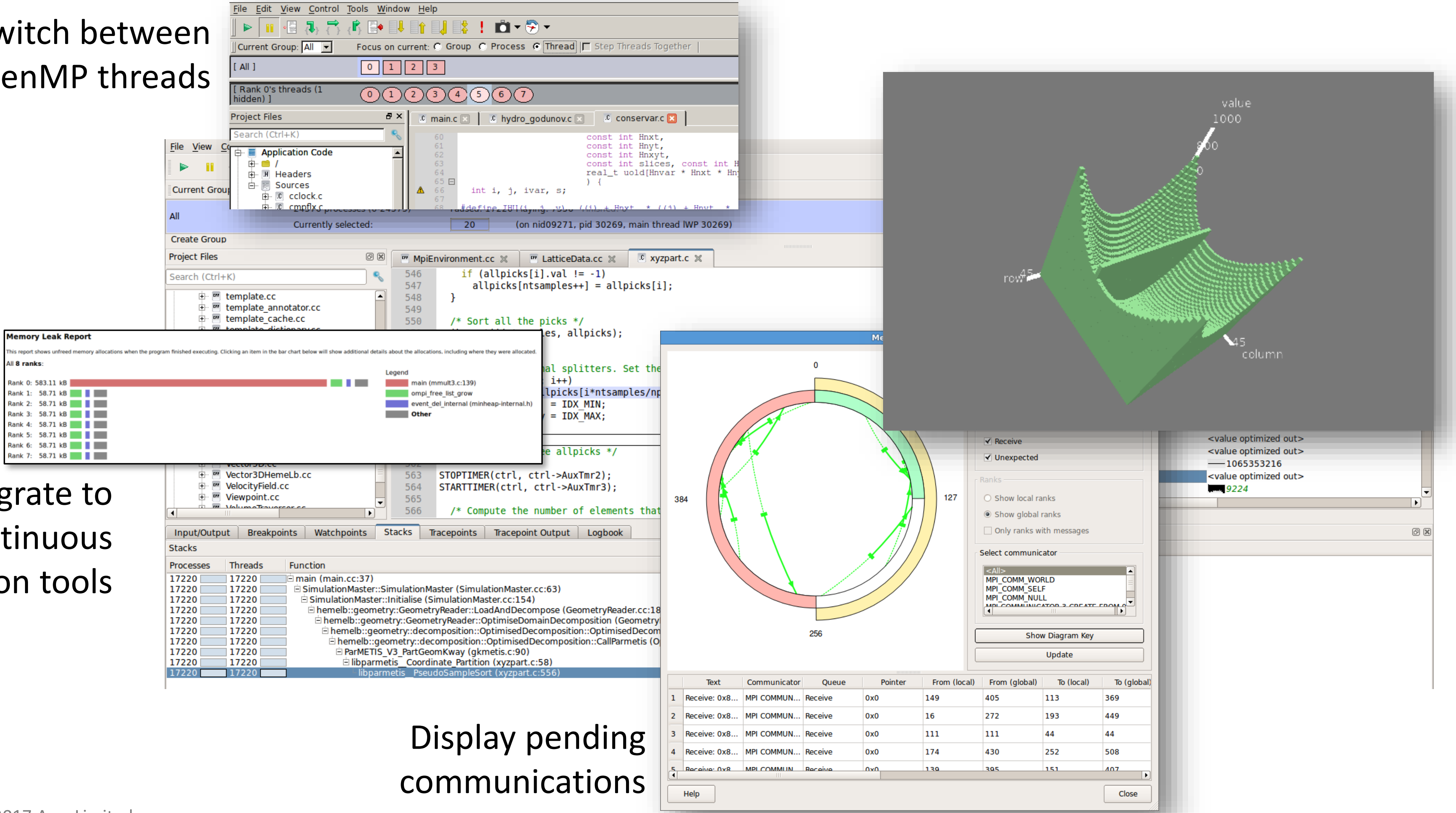
- ✔✔ Validate corrections and optimal Performance.

Key:

- ✔ arm PERFORMANCE REPORTS
- ✔ arm FORGE

Migrate and debug application

Switch between OpenMP threads

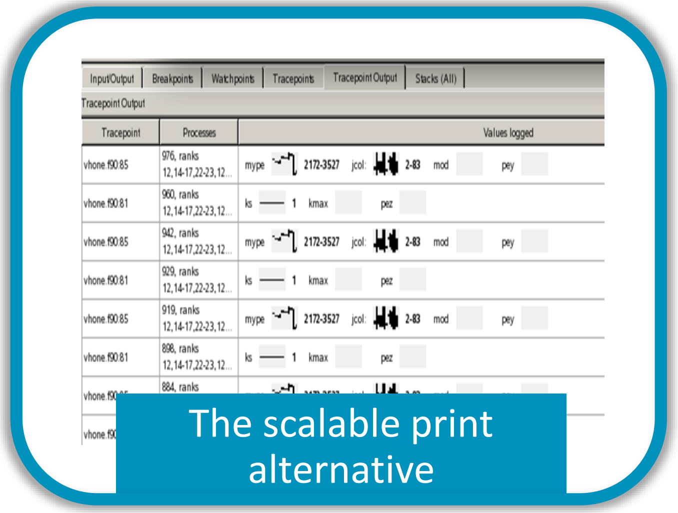


Visualise data structures

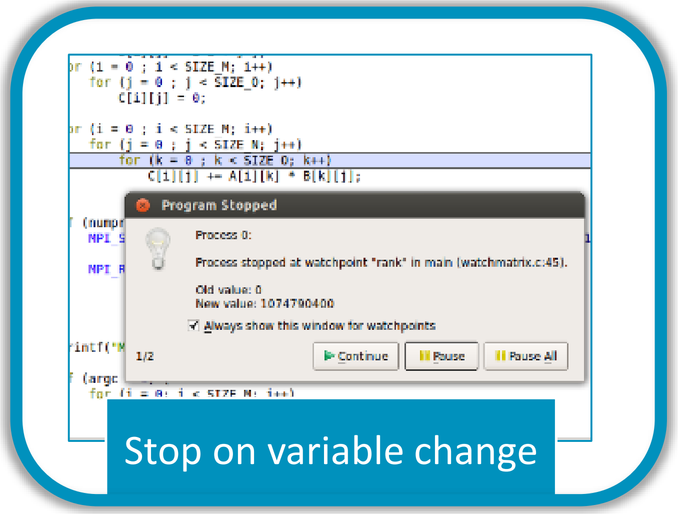
Integrate to continuous integration tools

Display pending communications

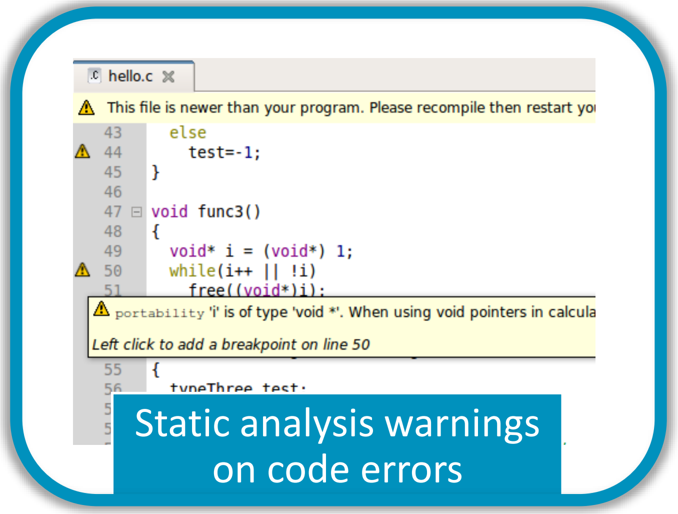
Five great things to try with Arm DDT



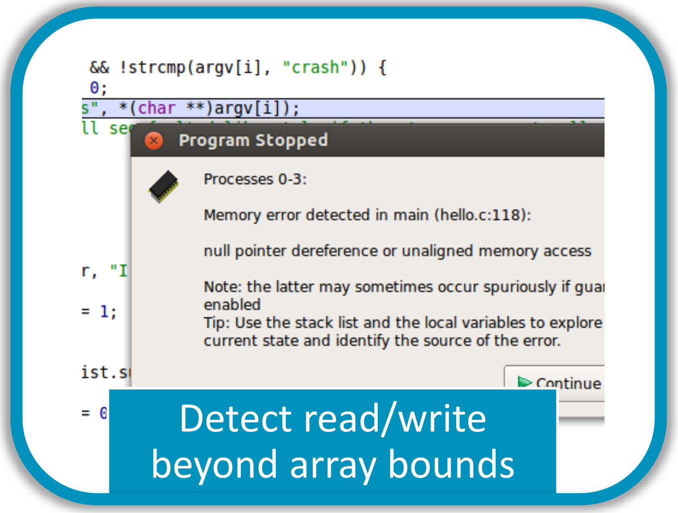
The scalable print alternative



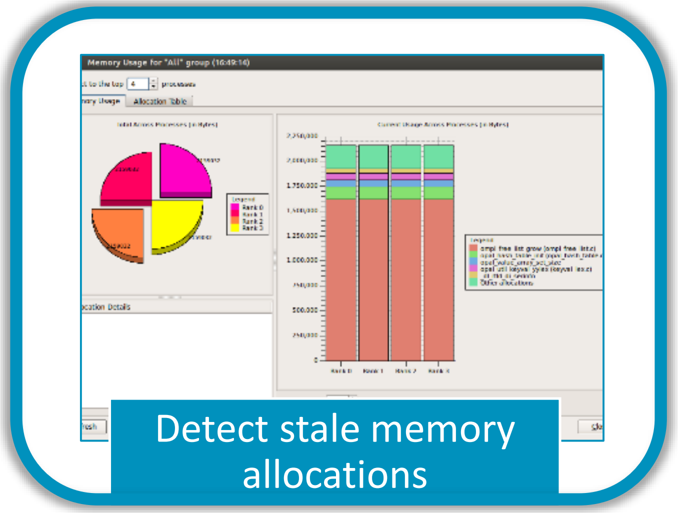
Stop on variable change



Static analysis warnings on code errors



Detect read/write beyond array bounds



Detect stale memory allocations

Arm DDT – The Debugger

Who had a rogue behavior ?

- Merges stacks from processes and threads

Where did it happen?

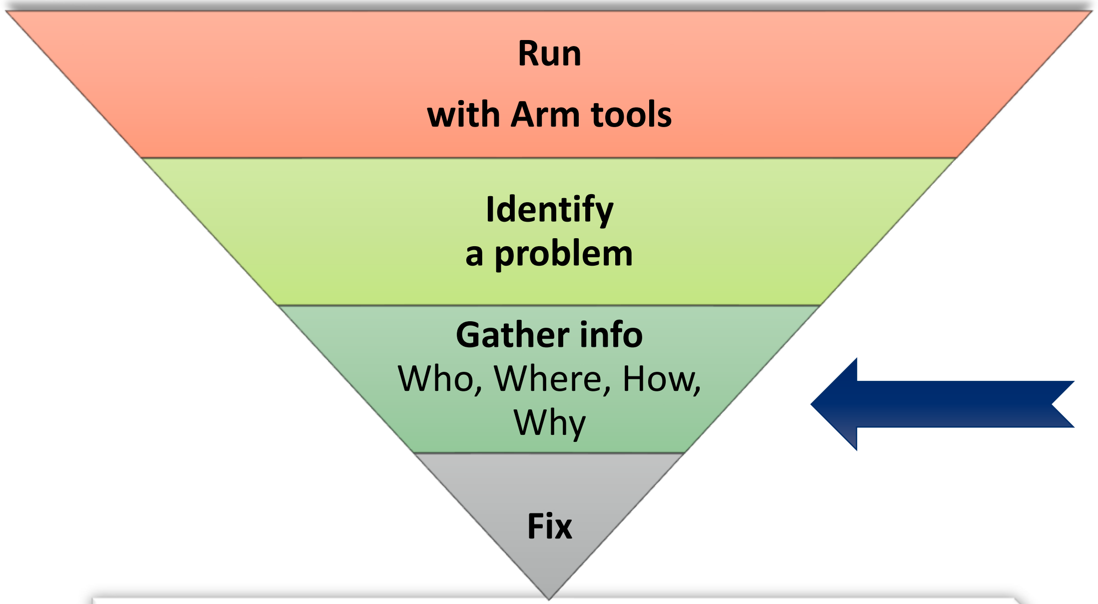
- leaps to source

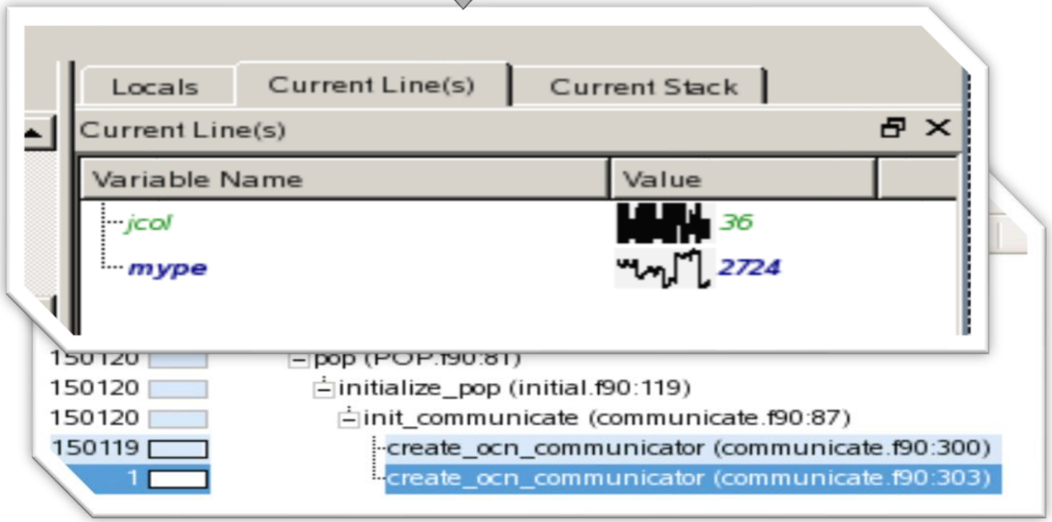
How did it happen?

- Diagnostic messages
- Some faults evident instantly from source

Why did it happen?

- Unique "Smart Highlighting"
- Sparklines comparing data across processes



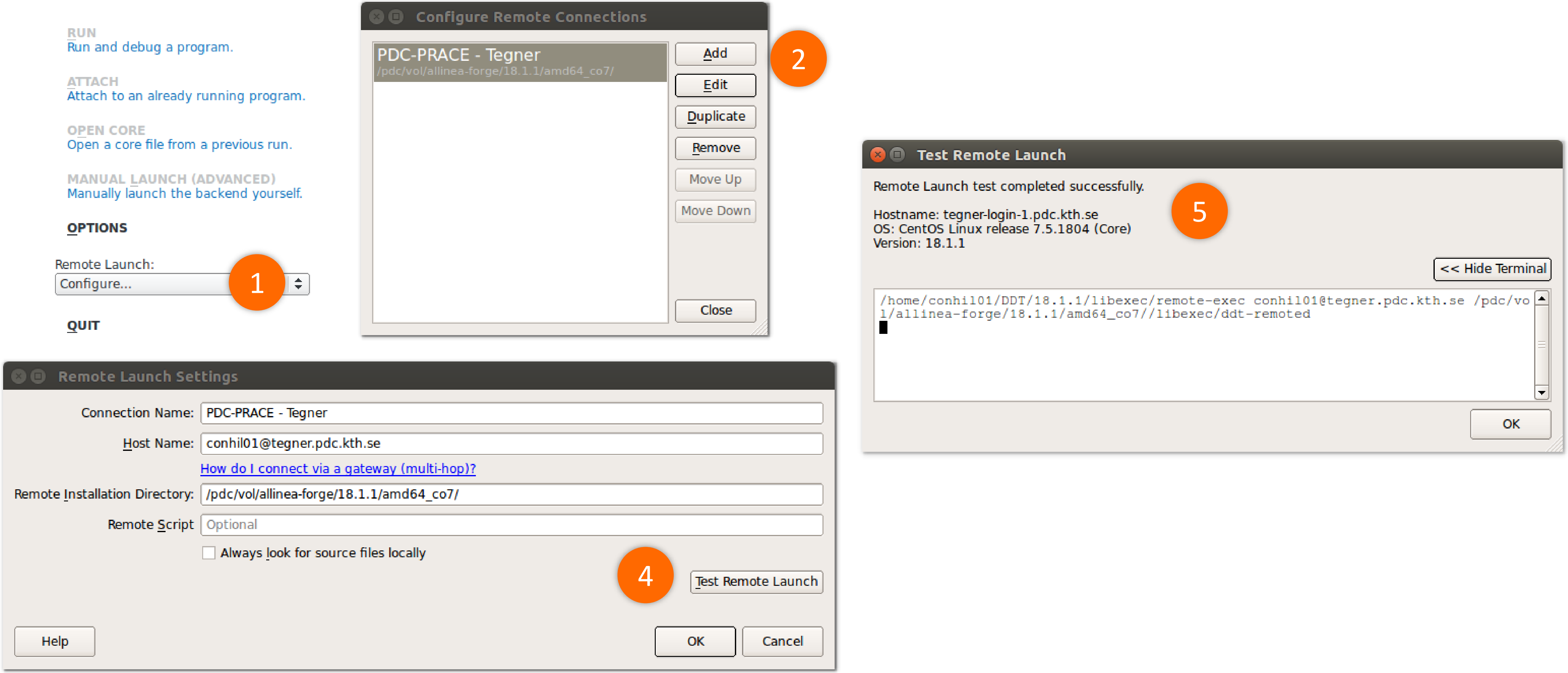


Hands – On :
Set up the Tools

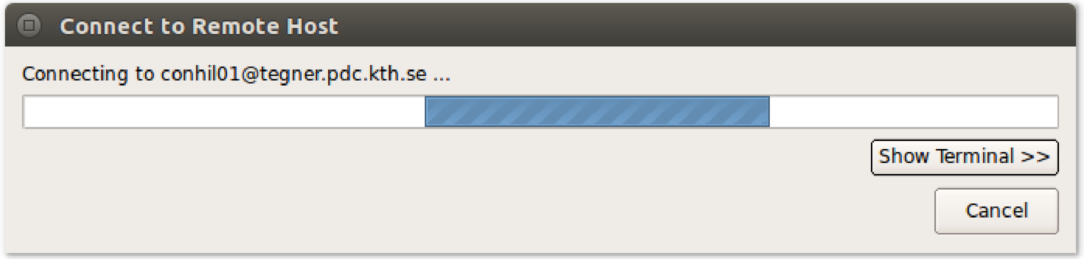
arm

Reverse-Connect – Client / Laptop side

```
kinit -f <userName>@NADA.KTH.SE
klist -f
export PATH=$PATH:<pathToForgeInstall>/bin
export PATH=$PATH:/home/prace/arm/forge/bin
ddt --version
ddt
```

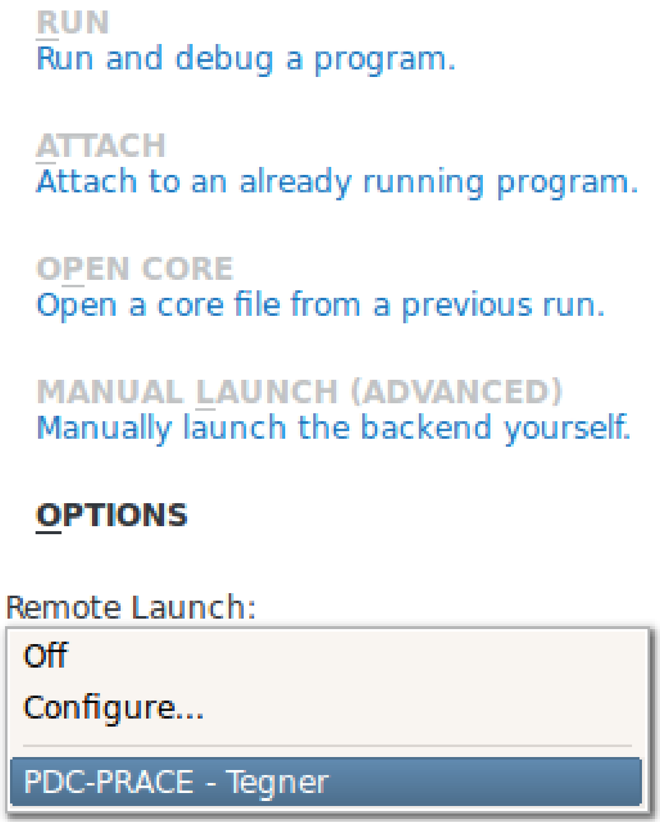

RUN
Run and debug a program.
ATTACH
Attach to an already running program.
OPEN CORE
Open a core file from a previous run.
MANUAL LAUNCH (ADVANCED)
Manually launch the backend yourself.
OPTIONS
Remote Launch:
Configure...
QUIT
1
Configure Remote Connections
PDC-PRACE - Tegner
/pdc/vol/allinea-forge/18.1.1/amd64_co7/
Add
Edit
Duplicate
Remove
Move Up
Move Down
Close
2
Remote Launch Settings
Connection Name: PDC-PRACE - Tegner
Host Name: conhil01@tegner.pdc.kth.se
How do I connect via a gateway (multi-hop)?
Remote Installation Directory: /pdc/vol/allinea-forge/18.1.1/amd64_co7/
Remote Script Optional
Always look for source files locally
4
Test Remote Launch
Help
OK
Cancel
Test Remote Launch
Remote Launch test completed successfully.
Hostname: tegner-login-1.pdc.kth.se
OS: CentOS Linux release 7.5.1804 (Core)
Version: 18.1.1
5
<< Hide Terminal
/home/conhil01/DDT/18.1.1/libexec/remote-exec conhil01@tegner.pdc.kth.se /pdc/vol/allinea-forge/18.1.1/amd64_co7//libexec/ddt-remoted
OK

arm

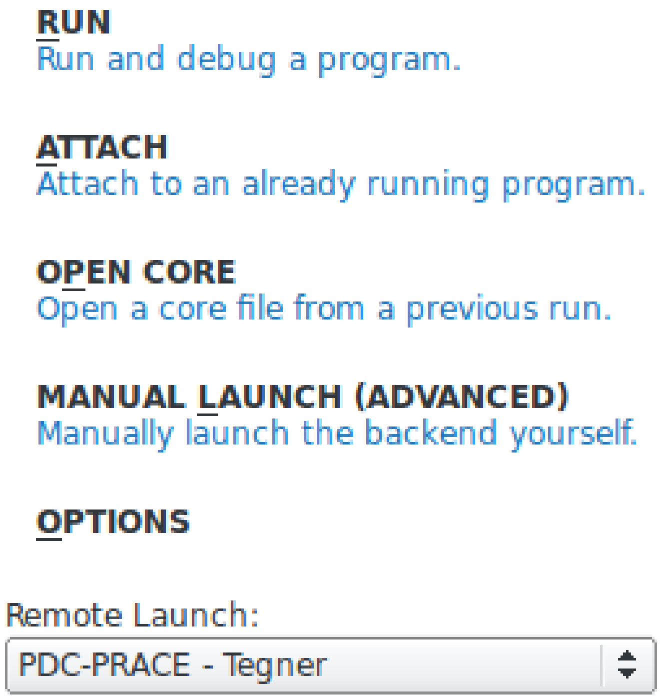


2

Pop – Up
Wait



1



3

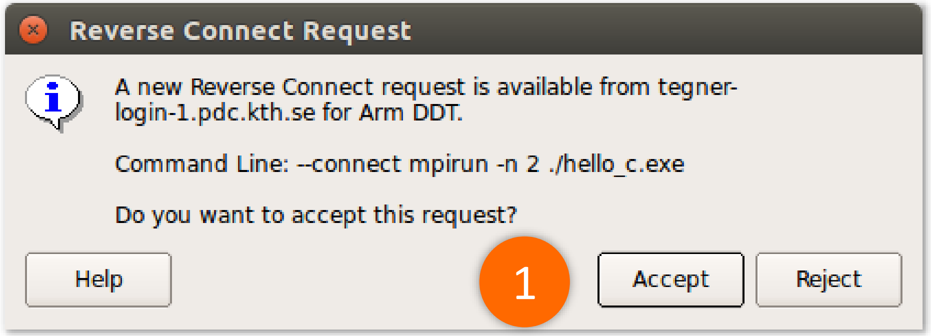
Reverse-Connect
Client ready

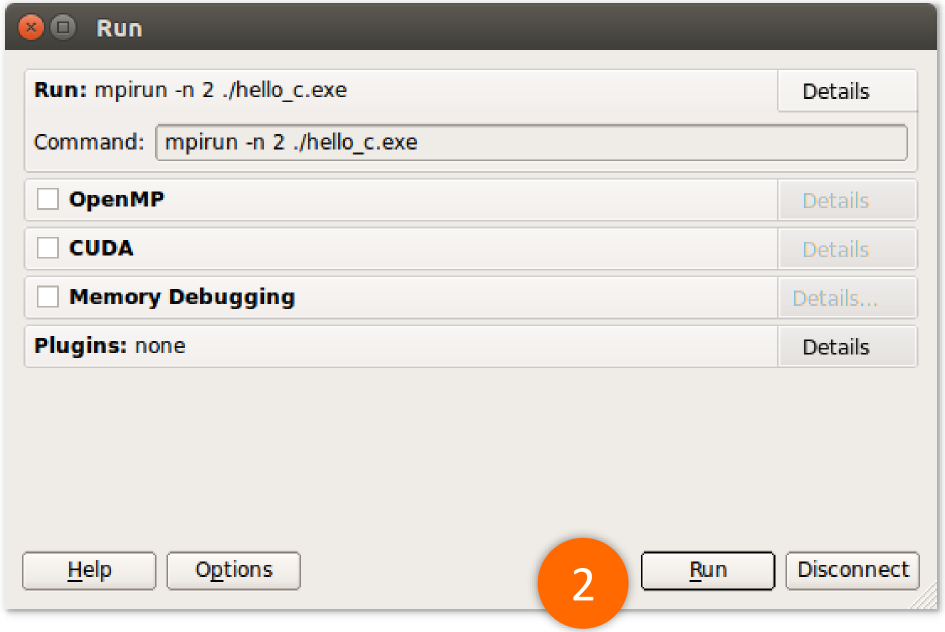
arm

Reverse-Connect – Server / Cluster side

```
ssh conhil01@tegner.pdc.kth.se
module load i-compilers
module load intelmpi
module load allinea-forge
cd /cfs/klemming/nobackup/c/conhil01
cp /afs/pdc.kth.se/home/c/conhil01/Public/arm_trial.tar.gz .
tar -xvf arm_trial.tar.gz
cp /afs/pdc.kth.se/home/c/conhil01/Public/Licence_kth .
unset ALLINEA_LICENSE_FILE_modshare
unset ALLINEA_LICENSE_FILE
export ALLINEA_FORCE_LICENCE_FILE=$PWD/Licence_kth
cd arm_trial
cd 0_test_reverse_connect
make
salloc -nodes=1 -t 00:10:00 -A pdc-test-2018
ddt --connect  mpirun -n 2 ./hello_c.exe
```

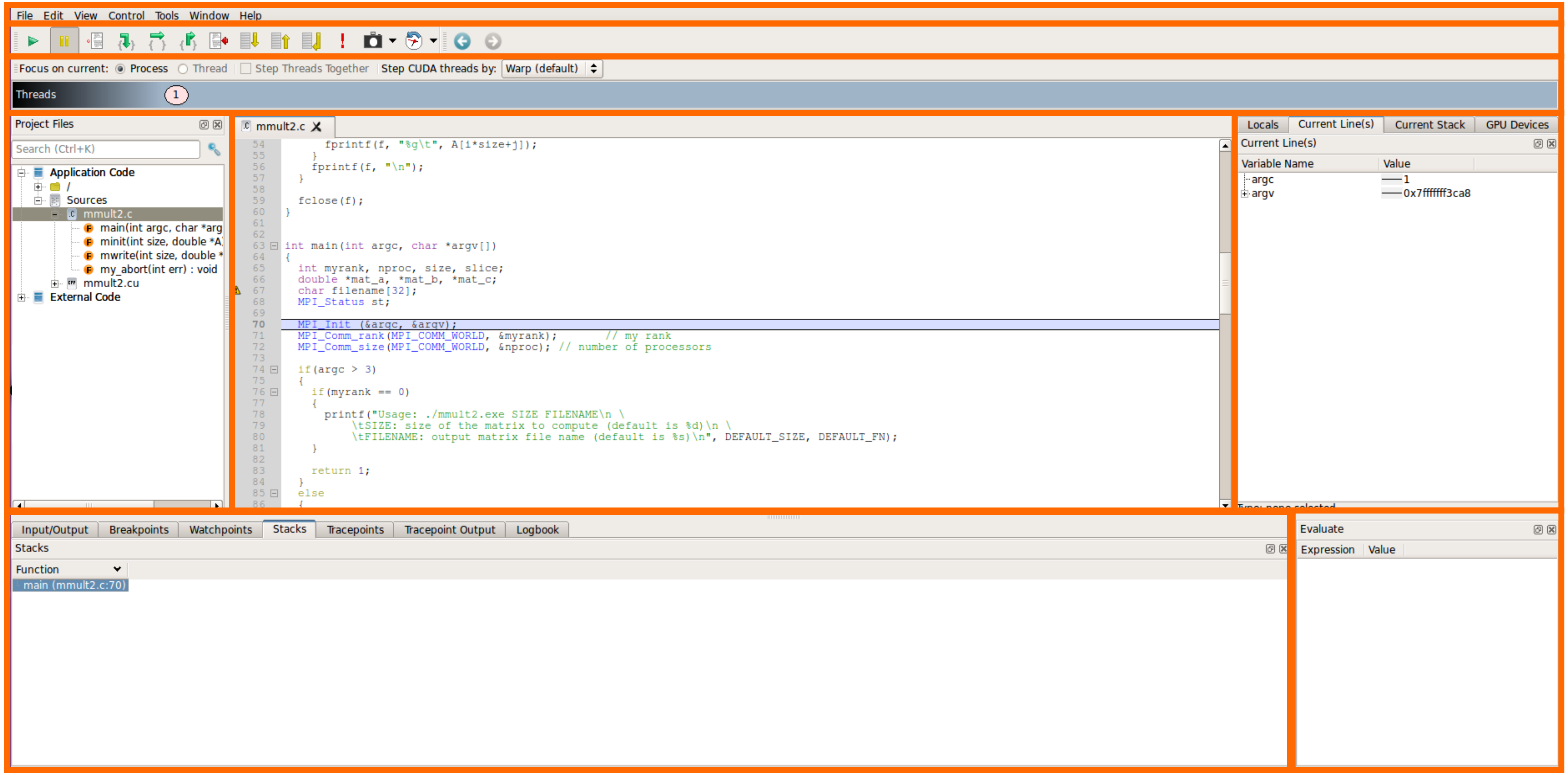
Reverse-Connect – Client / Laptop side





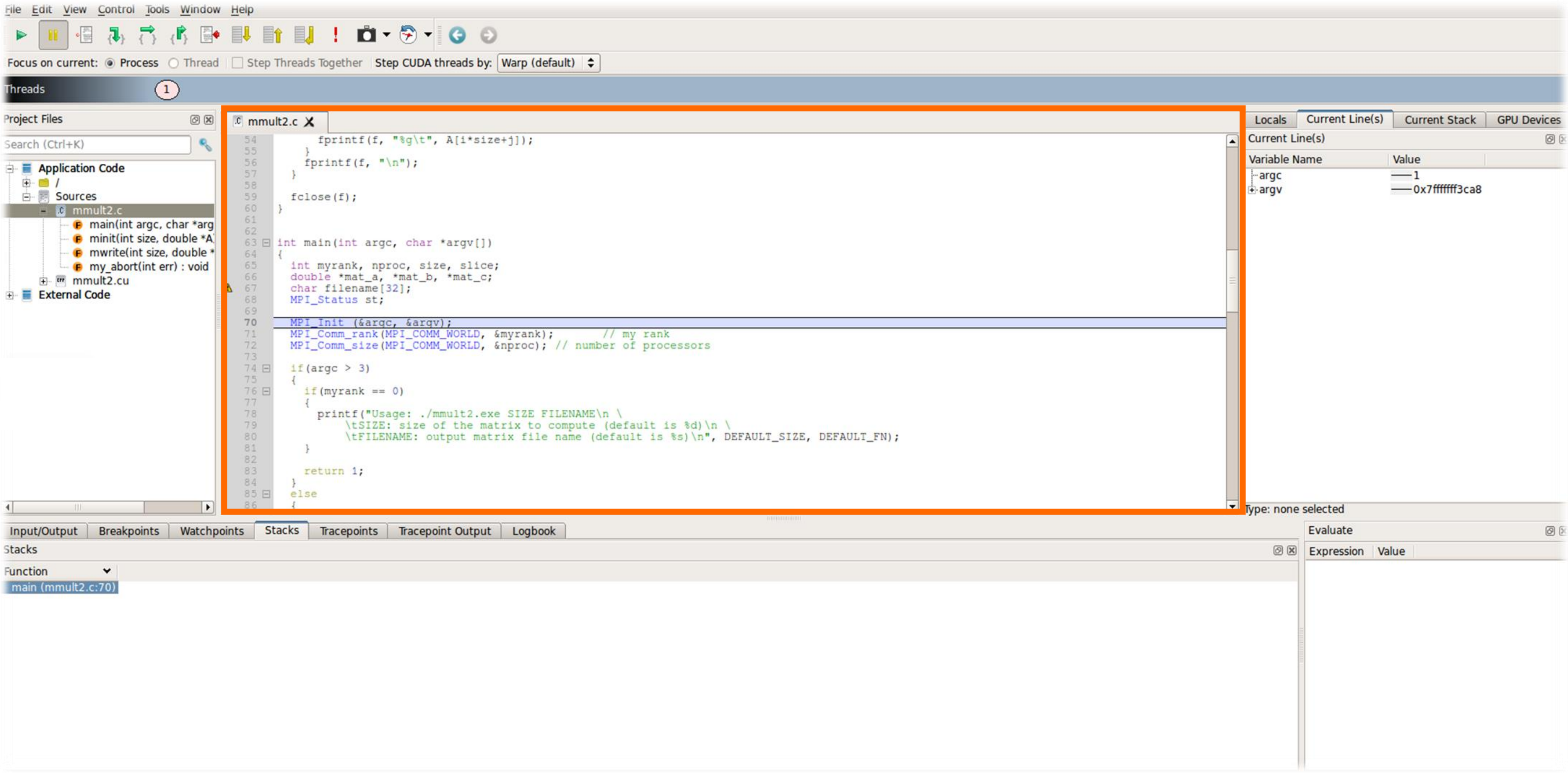
arm

User Interface



arm

User Interface – Source code viewer

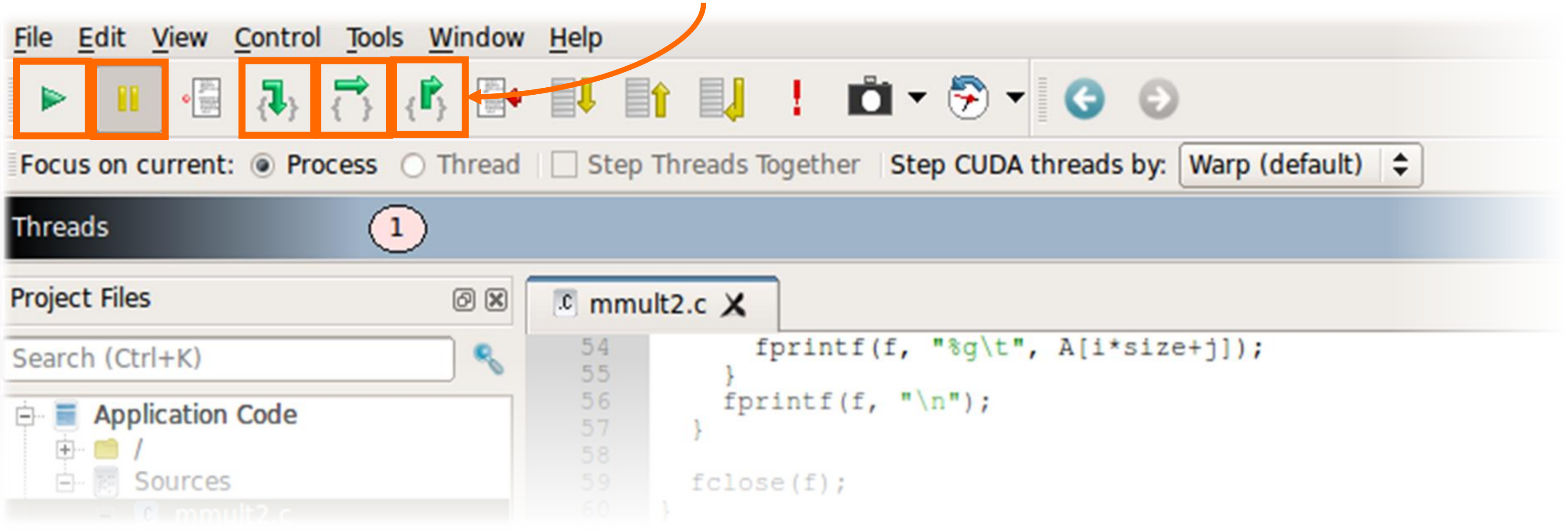


arm

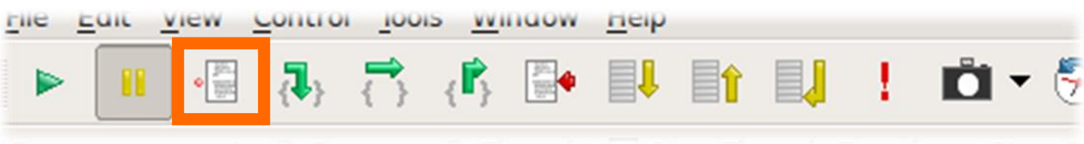
User Interface – Play/ Pause / Step

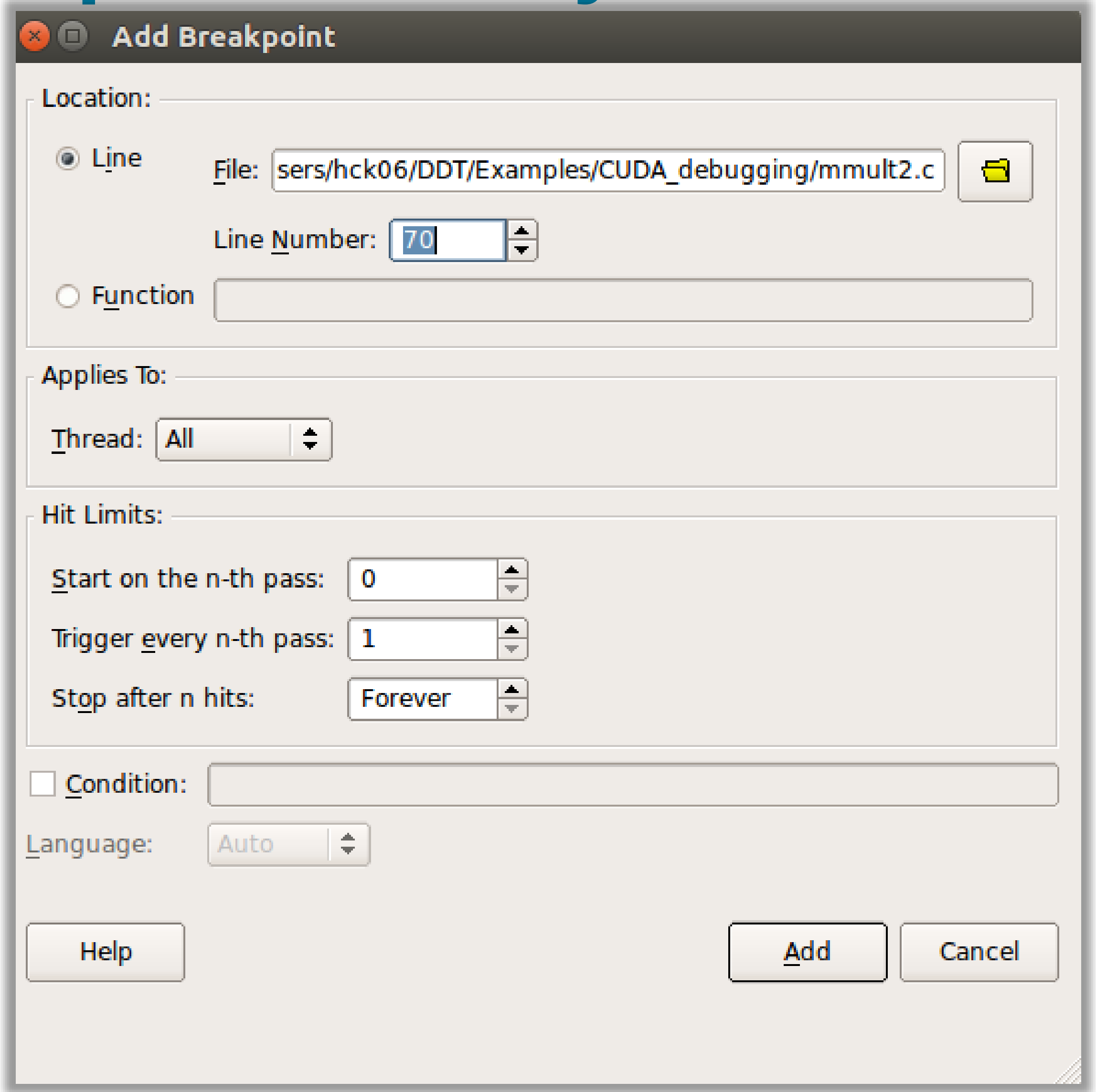
Play : Run everything. Use typically at the beginning or after Pause
Pause : Stops running current kernel

Step In : Enter a function call and display source code of the function
Step Over : Execute current line of code
Step Out : Comes back one stage above current stack



User Interface – Add Breakpoints – Way 1

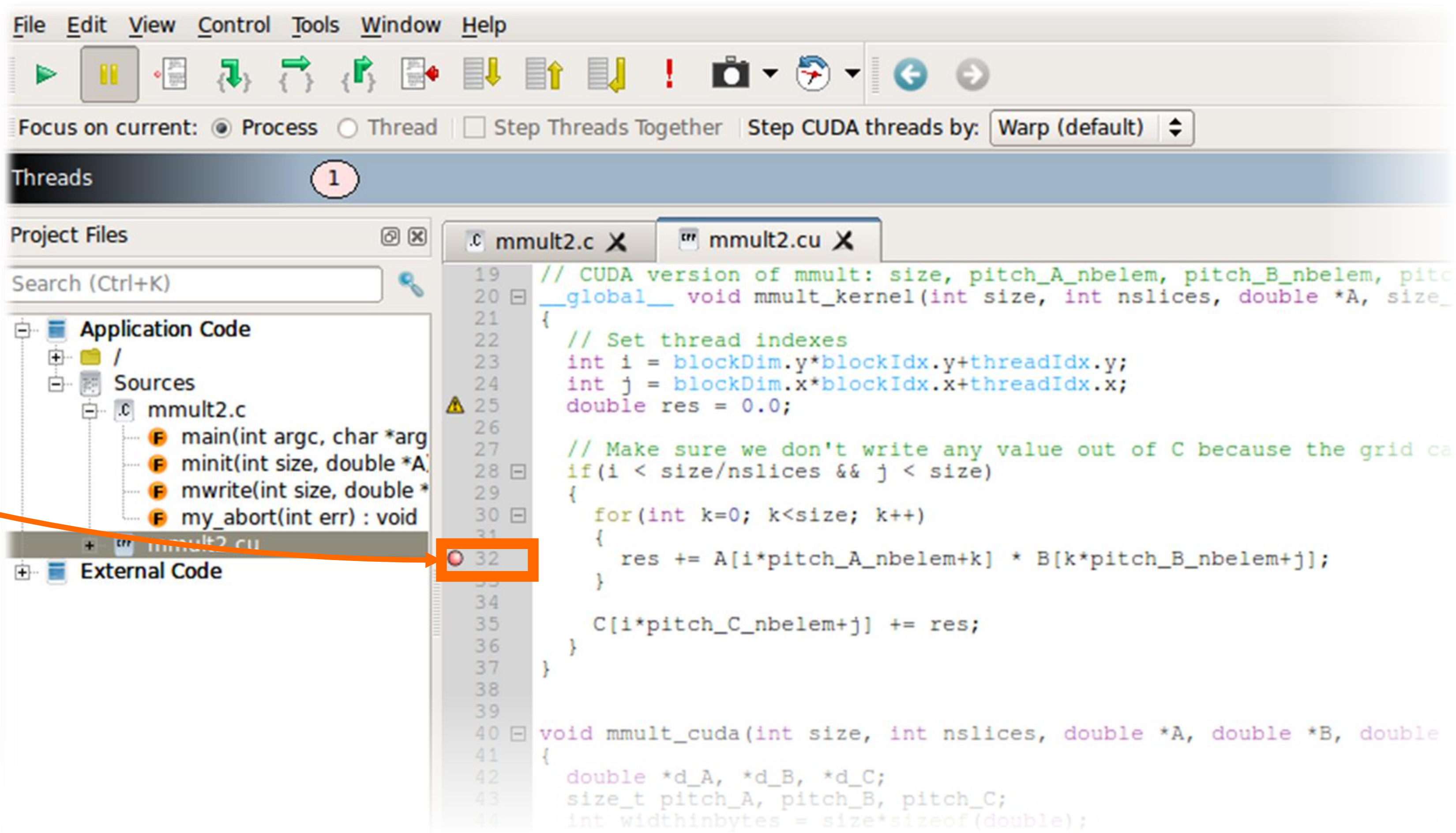




User Interface – Add Breakpoints – Way 2

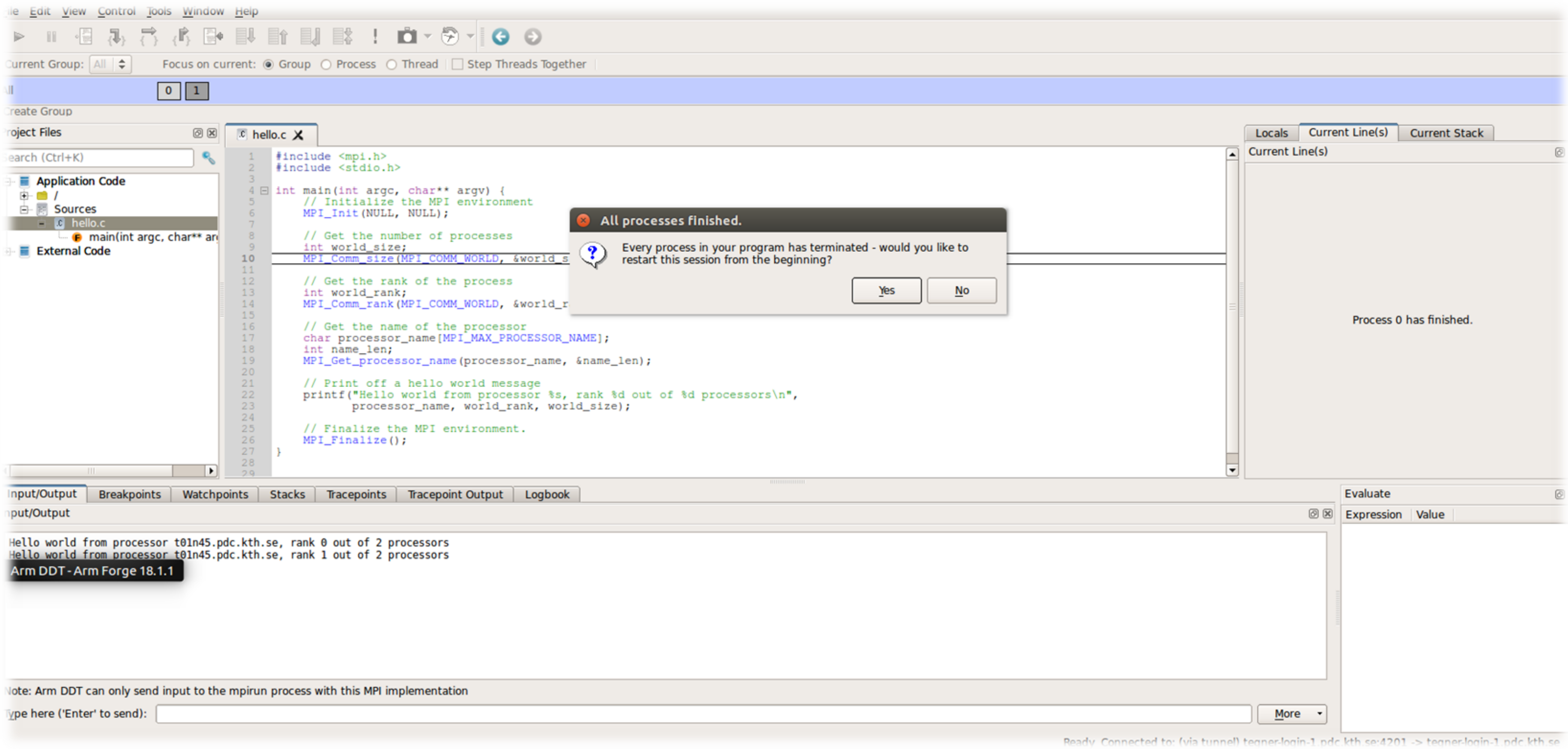
In the source code viewer, on the left, left click on the line to add a Breakpoint

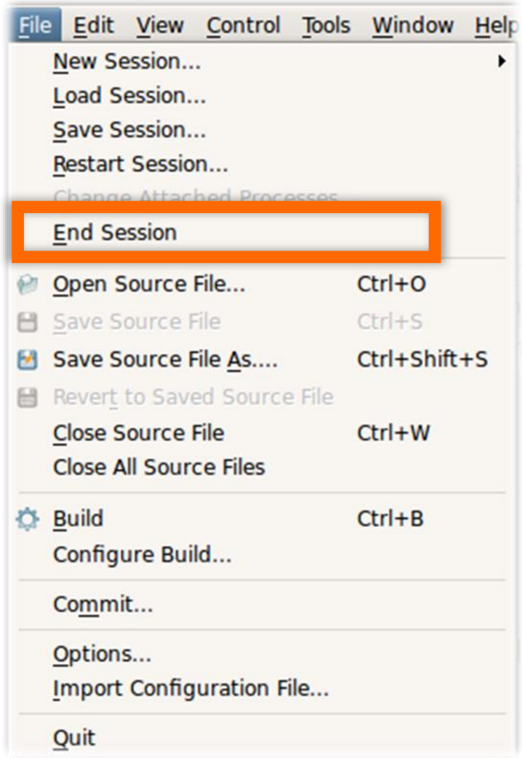
Typical next action : Play



arm

Reverse-Connect – Client / Laptop Side

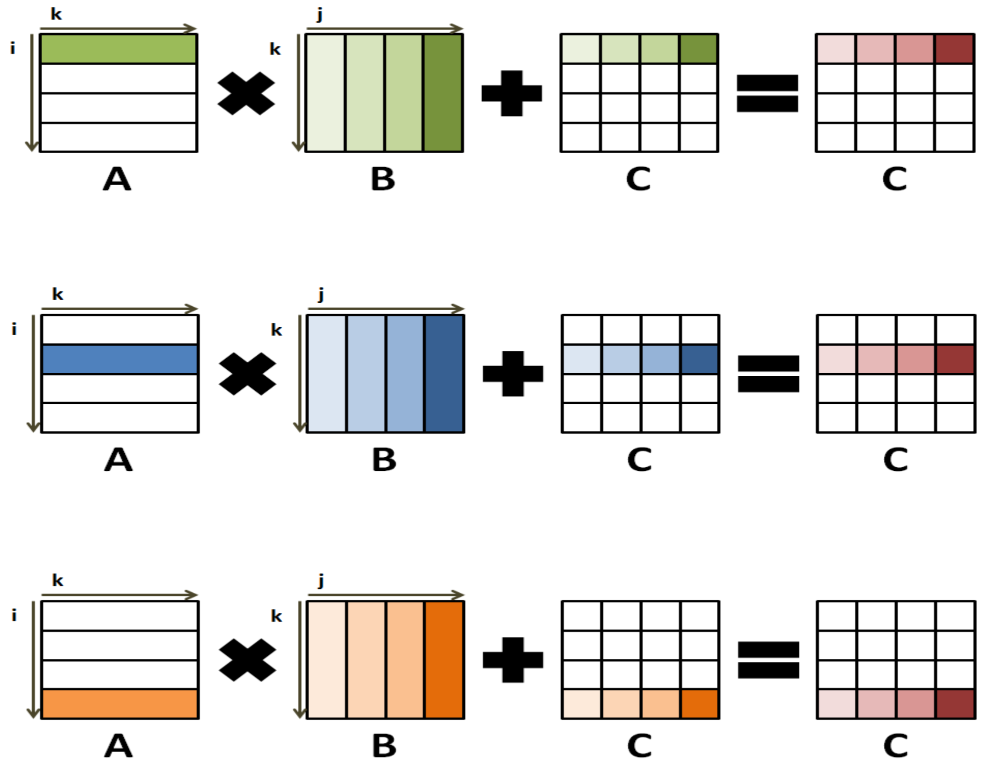




Hands – On :
SIGFPE (Arithmetic Exception)

Matrix Multiplication Example

$$C = A \times B + C$$



Environment configuration (reminder)

```
ssh conhil01@tegner.pdc.kth.se

module load i-compilers
module load intelmpi
module load allinea-forge

cp /afs/pdc.kth.se/home/c/conhil01/Public/arm_trial.tar.gz .
tar -xvf arm_trial.tar.gz

cp /afs/pdc.kth.se/home/c/conhil01/Public/Licence_kth .
unset ALLINEA_LICENSE_FILE_modshare
unset ALLINEA_LICENSE_FILE
export ALLINEA_FORCE_LICENCE_FILE=$PWD/Licence_kth
```


Hands – On : SIGFPE

- 1_interactive_debugging
- Compile the program
- Run one of the binaries. What do you see ?
- Let's debug it then !
- Recompile with DEBUG=1, launch DDT and … debug !
- Can you find where the problem comes from ?
- Modify the code and recompile (in DDT)
- Relaunch the program.

Hands – On : Memory Debugging

arm

Heap debugging options available

Fast

basic

- Detect invalid pointers passed to memory functions (e.g. malloc, free, ALLOCATE, DEALLOCATE,...)

check-fence

- Check the end of an allocation has not been overwritten when it is freed.

free-protect

- Protect freed memory (using hardware memory protection) so subsequent read/writes cause a fatal error.

Added goodiness

- Memory usage, statistics, etc.

Balanced

free-blank

- Overwrite the bytes of freed memory with a known value.

alloc-blank

- Initialise the bytes of new allocations with a known value.

check-heap

- Check for heap corruption (e.g. due to writes to invalid memory addresses).

realloc-copy

- Always copy data to a new pointer when re-allocating a memory allocation (e.g. due to realloc)

Thorough

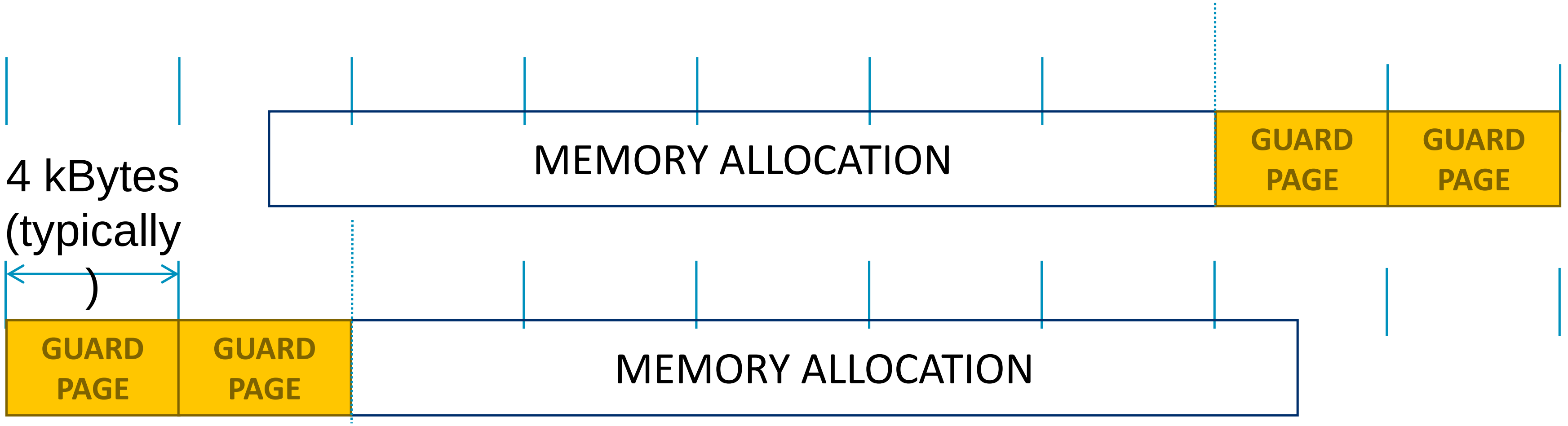
check-blank

- Check to see if space that was blanked when a pointer was allocated/freed has been overwritten.

check-funcs

- Check the arguments of addition functions (mostly string operations) for invalid pointers.

Guard pages (aka “Electric Fences”)



- **A powerful feature…:**
  - Forbids read/write on guard pages throughout the whole execution

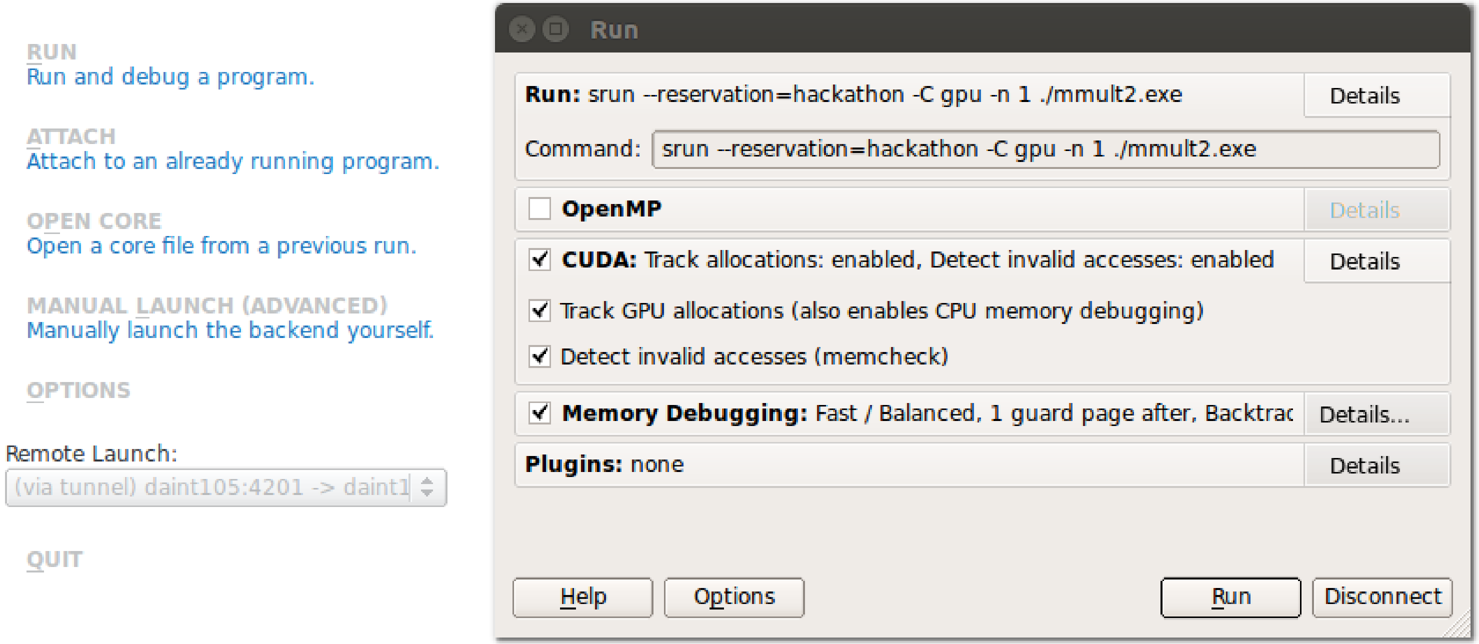
    (because it overrides C Standard Memory Management library)
- **… to be used carefully:**
  - Kernel limitation: up to 32k guard pages max (“mprotect fails” error)
  - Beware the additional memory usage cost

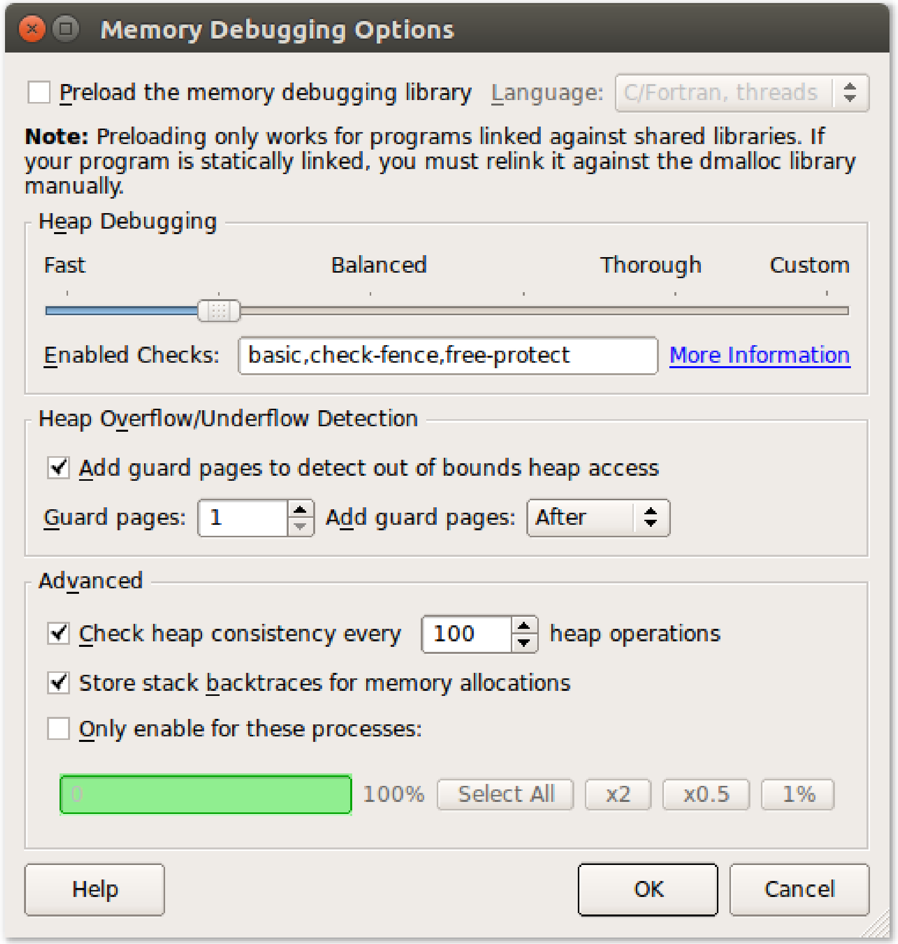
Compilation flags for memory debugging

Compiler : `-O0 -g`

Linking : `-L<path_to_DDT_install>/lib/64 -Wl,--allow-multiple-definition,--undefined=malloc,--undefined=_ZdaPv -ldmallocthcxx`

Memory debugging





Environment configuration (reminder)

```
ssh conhil01@tegner.pdc.kth.se

module load i-compilers
module load intelmpi
module load allinea-forge

cp /afs/pdc.kth.se/home/c/conhil01/Public/arm_trial.tar.gz .
tar -xvf arm_trial.tar.gz

cp /afs/pdc.kth.se/home/c/conhil01/Public/Licence_kth .
unset ALLINEA_LICENSE_FILE_modshare
unset ALLINEA_LICENSE_FILE
export ALLINEA_FORCE_LICENCE_FILE=$PWD/Licence_kth
```


Hands – On : Memory debugging

- 3_offline_debugging
- Compile the program
- Run one of the binaries. What do you see ?
- No problem ? Are you sure ? Let's launch DDT, just in case!
- Recompile with DEBUG=1
- Launch the application with DDT
- Check memory debugging and guard-pages
- Run the program … Any problem ?
- Can you resolve it ?
- Modify the code and recompile (in DDT)
- Relaunch the program.

arm

Hands – On : Memory debugging

- Are you sure we are done with hidden issues ?
- Use DDT offline report with "--offline --mem-debug" flags
- Have a look to the report, anything suspicious ?
- Do you see how to fix this ?

Arm Performance Reports

arm

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.

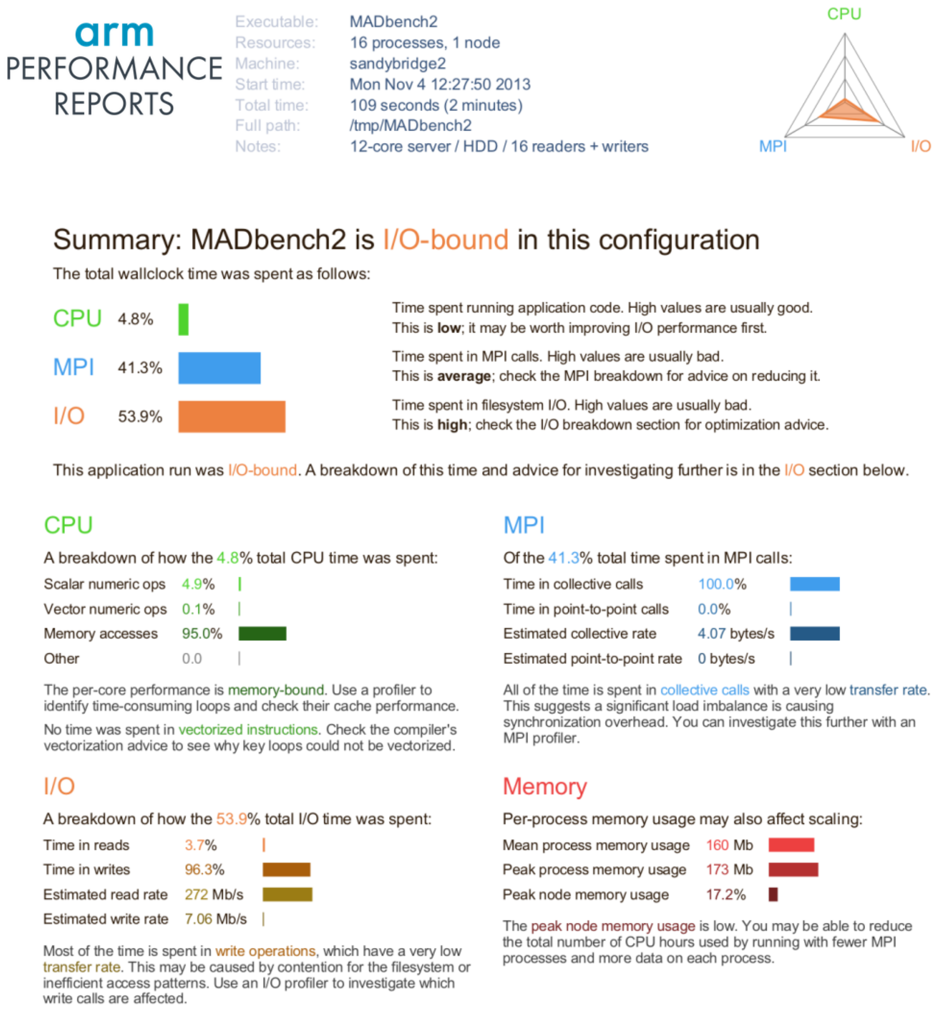
9 Verification

- ✔✔ Validate corrections and optimal Performance.

Key:

- ✔ arm PERFORMANCE REPORTS
- ✔ arm FORGE

“Learn” with Arm Performance Reports



arm PERFORMANCE REPORTS

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

CPU MPI I/O

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

The total wallclock time was spent as follows:

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

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

CPU

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

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

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

No time was spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

Of the 41.3% total time spent in MPI calls:

Time in collective calls	100.0%
Time in point-to-point calls	0.0%
Estimated collective rate	4.07 bytes/s
Estimated point-to-point rate	0 bytes/s

All of the time is spent in collective calls with a very low transfer rate. This suggests a significant load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

I/O

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

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

Most of the time is spent in write operations, which have a very low transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Memory

Per-process memory usage may also affect scaling:

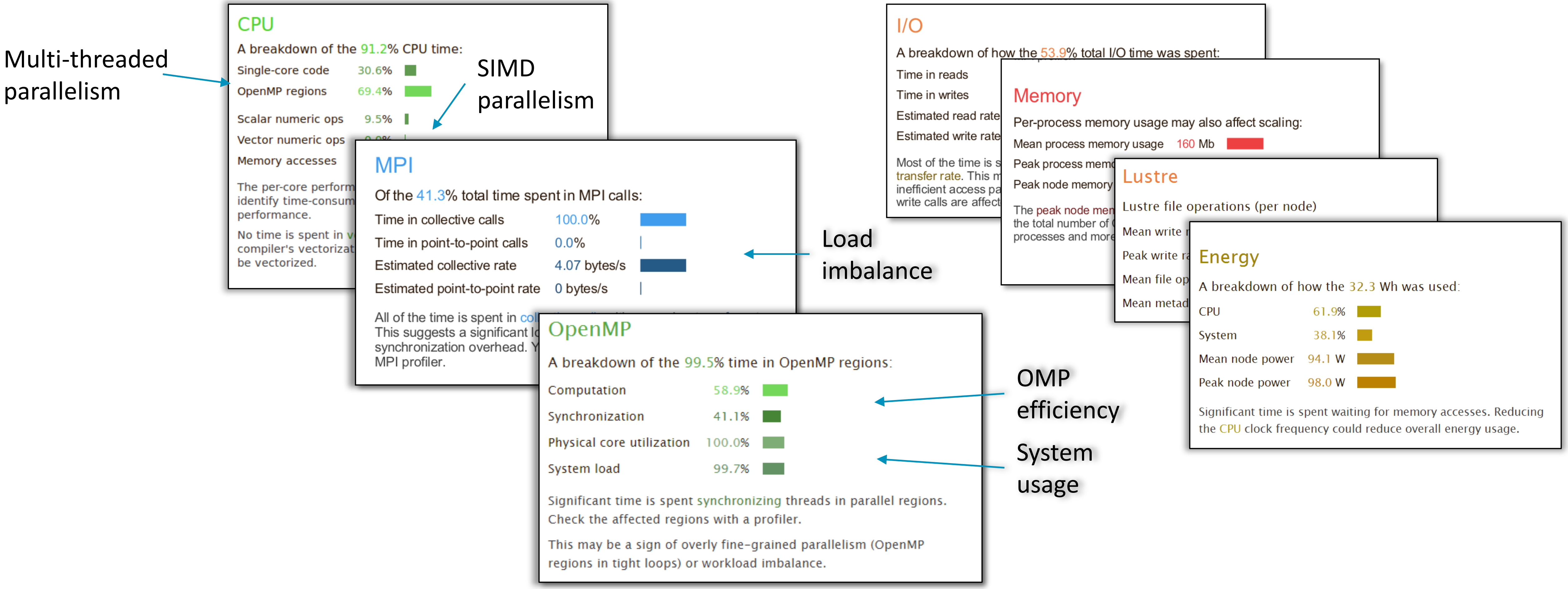
Mean process memory usage	160 Mb
Peak process memory usage	173 Mb
Peak node memory usage	17.2%

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

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

arm

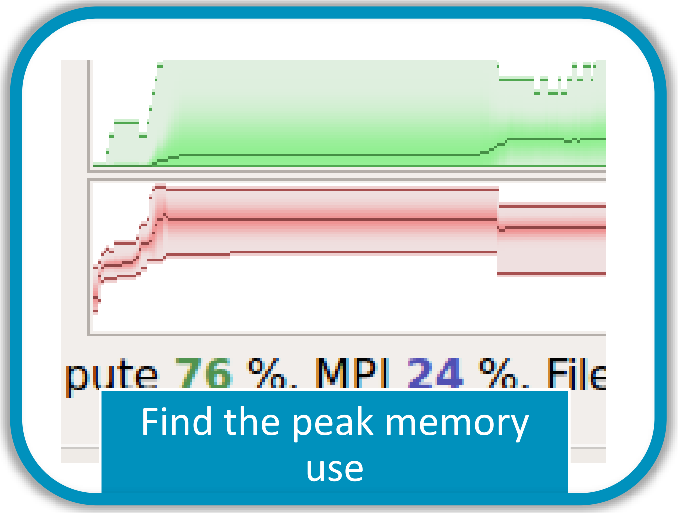
Metrics Overview



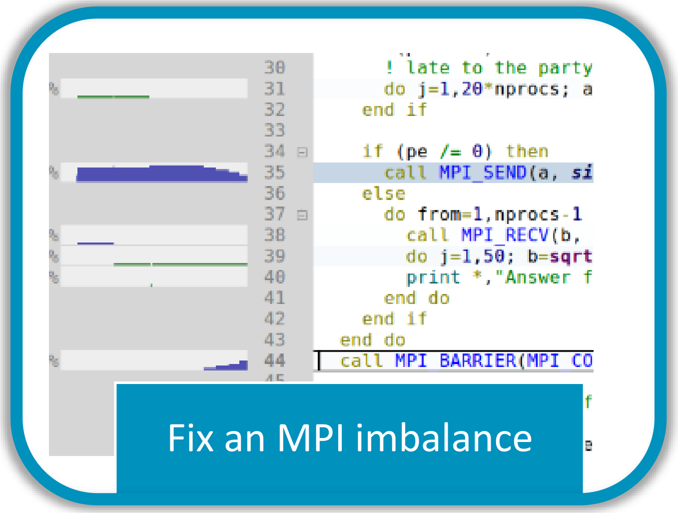
Arm MAP

arm

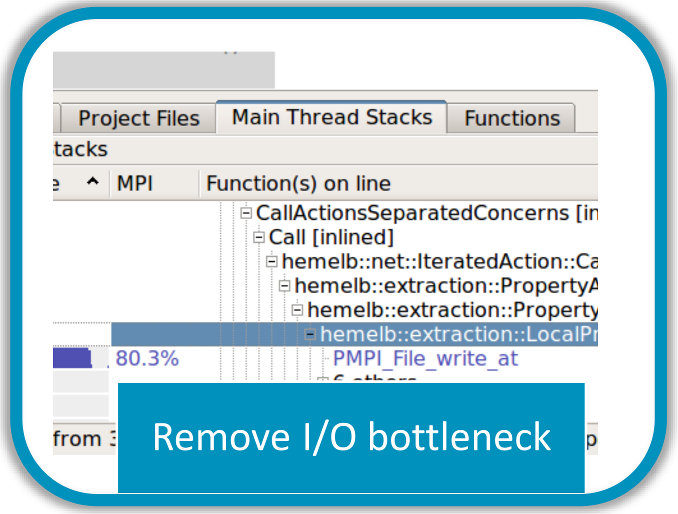
Six Great Things to Try with Arm MAP



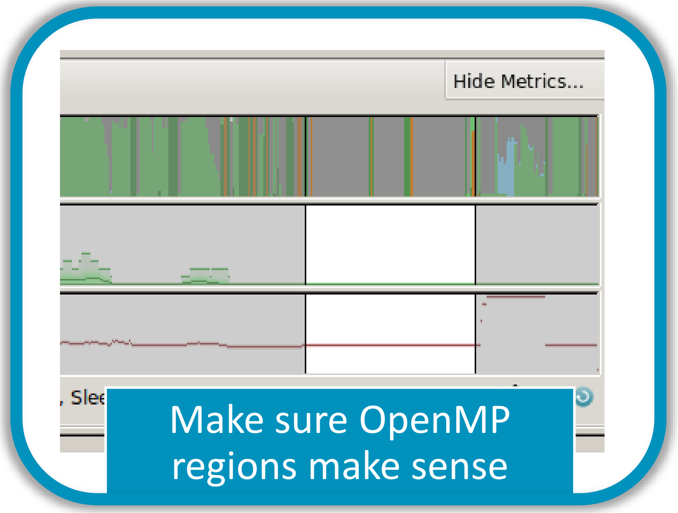
Find the peak memory use



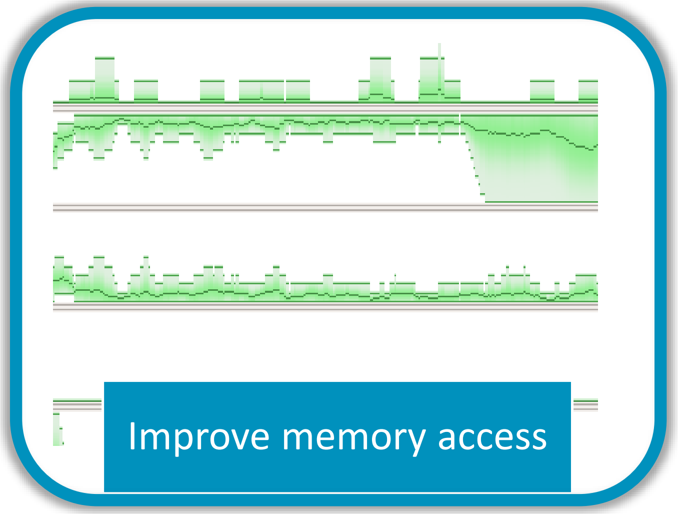
Fix an MPI imbalance



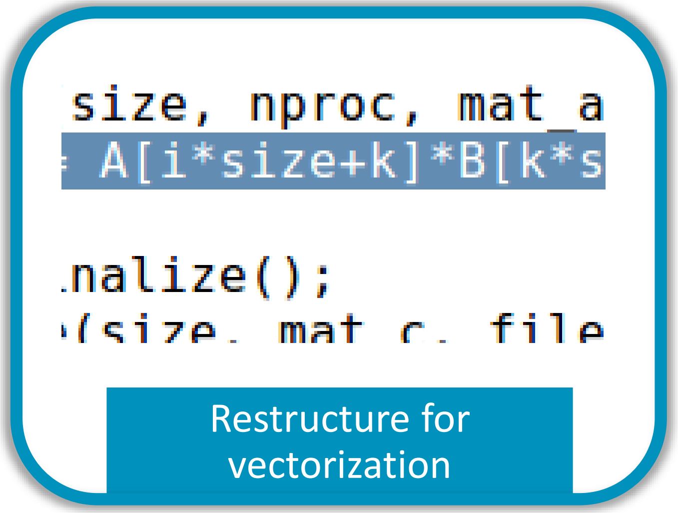
Remove I/O bottleneck



Make sure OpenMP regions make sense

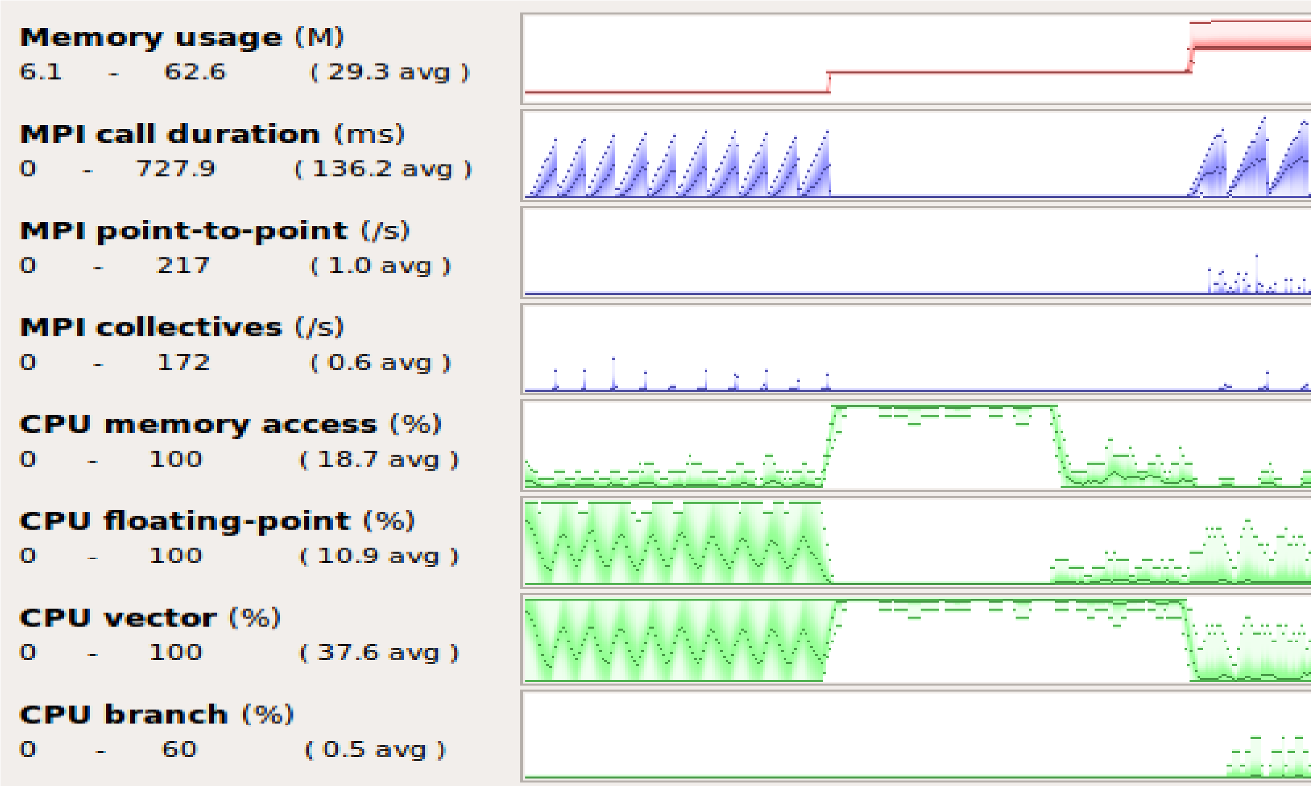


Improve memory access



Restructure for vectorization

Glean Deep Insight from our Source-Level Profiler



Track memory usage across the entire application over time

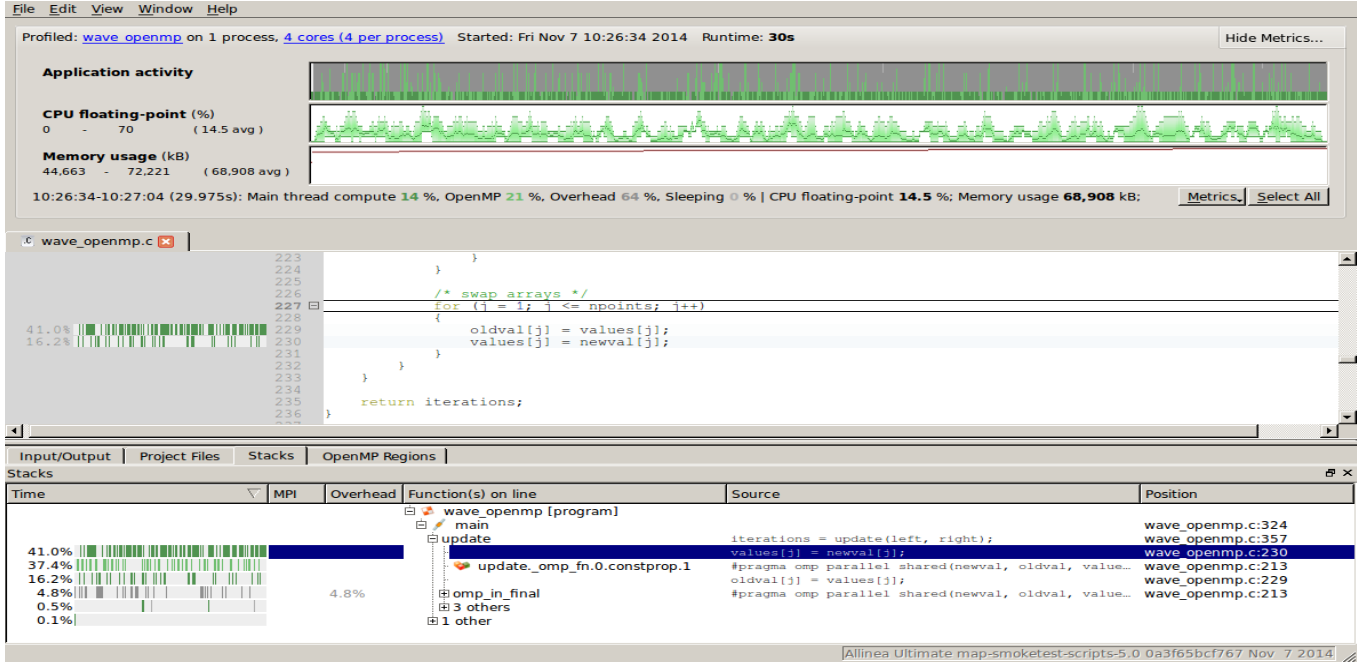
Spot MPI and OpenMP imbalance and overhead

Optimize CPU memory and vectorization in loops

Detect and diagnose I/O bottlenecks at real scale

Allinea MAP – The Profiler

- Small data files
- <5% slowdown
- No instrumentation
- No recompilation



How Arm MAP is different

Adaptive sampling	Sample frequency decreases over time	Data never grows too much	**Run for as long as you want**
Scalable	Same scalable infrastructure as Allinea DDT	Merges sample data at end of job	**Handles very high core counts, fast**
Instruction analysis	Categorizes instructions sampled	Knows where processor spends time	**Shows vectorization and memory bandwidth**
Thread profiling	Core-time not thread-time profiling	Identifies lost compute time	**Detects OpenMP issues**
Integrated	Part of Forge tool suite	Zoom and drill into profile	**Profiling within your code**

Preparing Code for Use with MAP

To see the source code, the application should be compiled with the debug flag typically **-g**

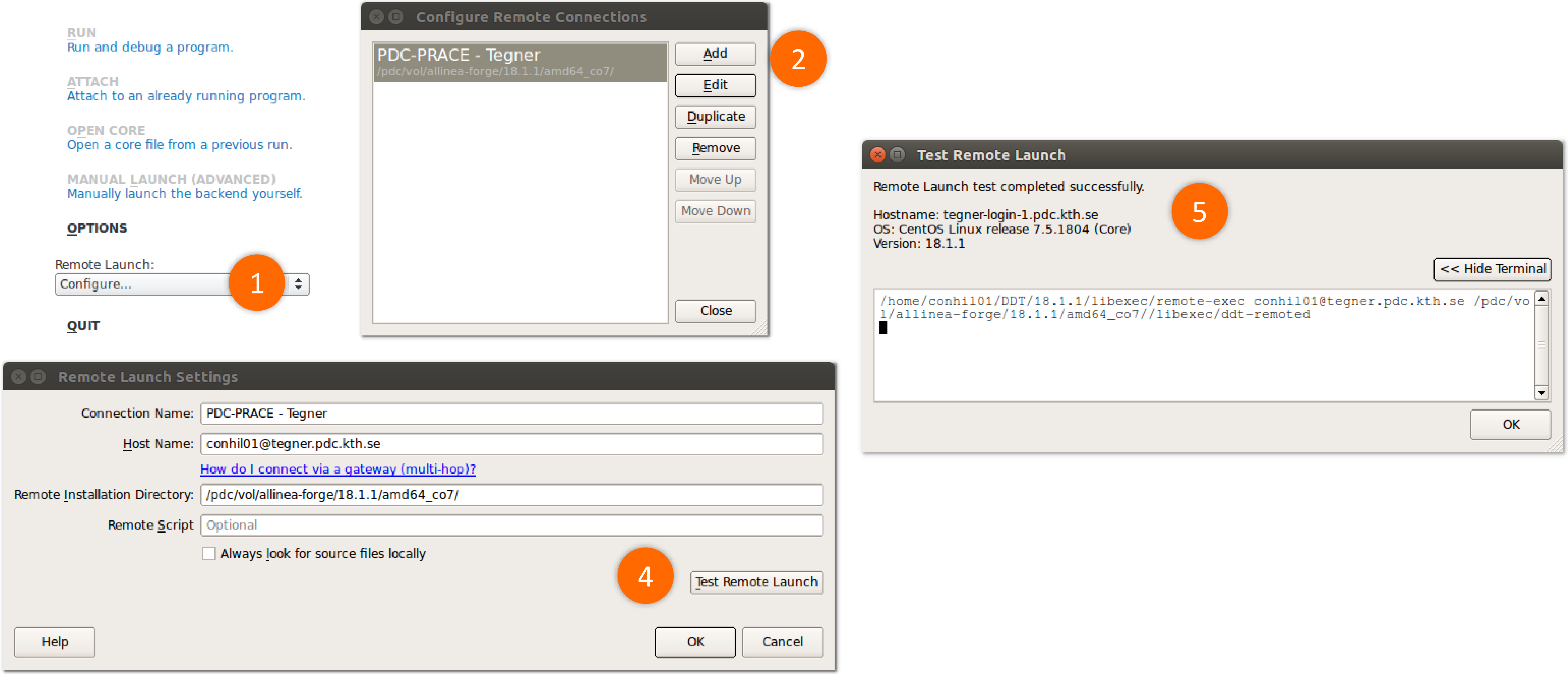
It is recommended to *always* keep optimization flags on when profiling

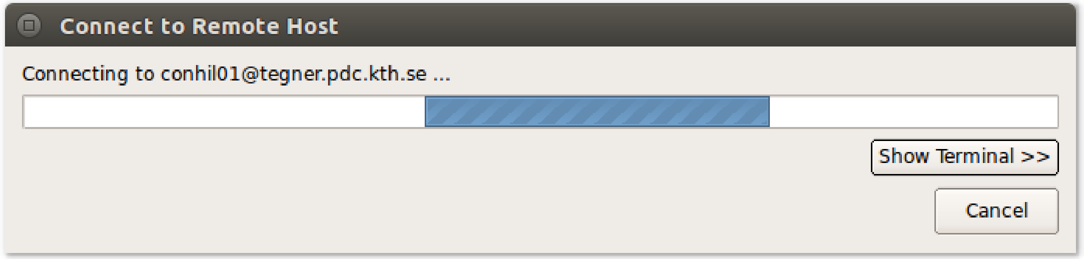
Hands – On : Launch MAP

arm

Reverse-Connect – Client / Laptop side

```
kinit -f <userName>@NADA.KTH.SE

klist -f

export PATH=$PATH:<pathToForgeInstall>/bin

map
```

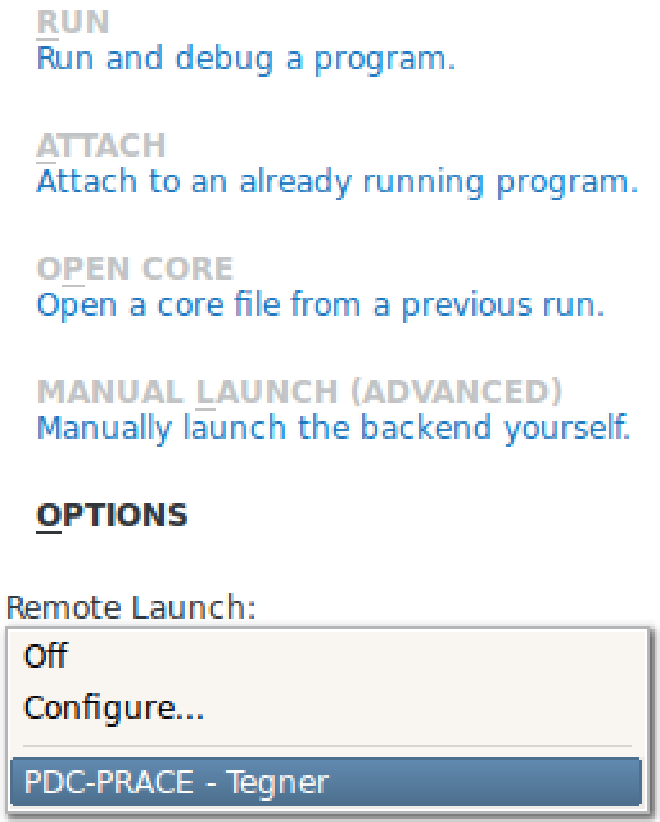


RUN
Run and debug a program.
ATTACH
Attach to an already running program.
OPEN CORE
Open a core file from a previous run.
MANUAL LAUNCH (ADVANCED)
Manually launch the backend yourself.
OPTIONS
Remote Launch:
Configure...
QUIT
1
Configure Remote Connections
PDC-PRACE - Tegner
/pdc/vol/allinea-forge/18.1.1/amd64_co7/
Add
Edit
Duplicate
Remove
Move Up
Move Down
Close
2
Remote Launch Settings
Connection Name: PDC-PRACE - Tegner
Host Name: conhil01@tegner.pdc.kth.se
How do I connect via a gateway (multi-hop)?
Remote Installation Directory: /pdc/vol/allinea-forge/18.1.1/amd64_co7/
Remote Script Optional
Always look for source files locally
4
Test Remote Launch
Help
OK
Cancel
Test Remote Launch
Remote Launch test completed successfully.
Hostname: tegner-login-1.pdc.kth.se
OS: CentOS Linux release 7.5.1804 (Core)
Version: 18.1.1
5
<< Hide Terminal
/home/conhil01/DDT/18.1.1/libexec/remote-exec conhil01@tegner.pdc.kth.se /pdc/vol/allinea-forge/18.1.1/amd64_co7//libexec/ddt-remoted
OK

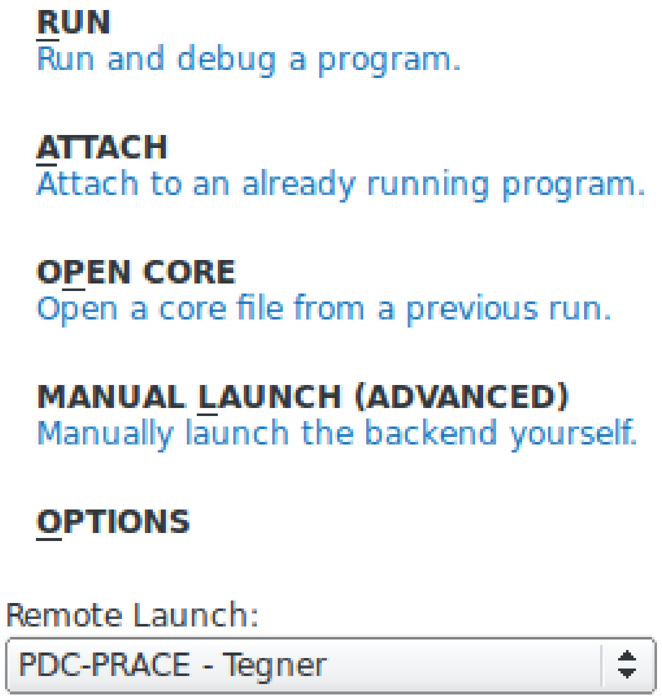


2

Pop – Up
Wait

1





3

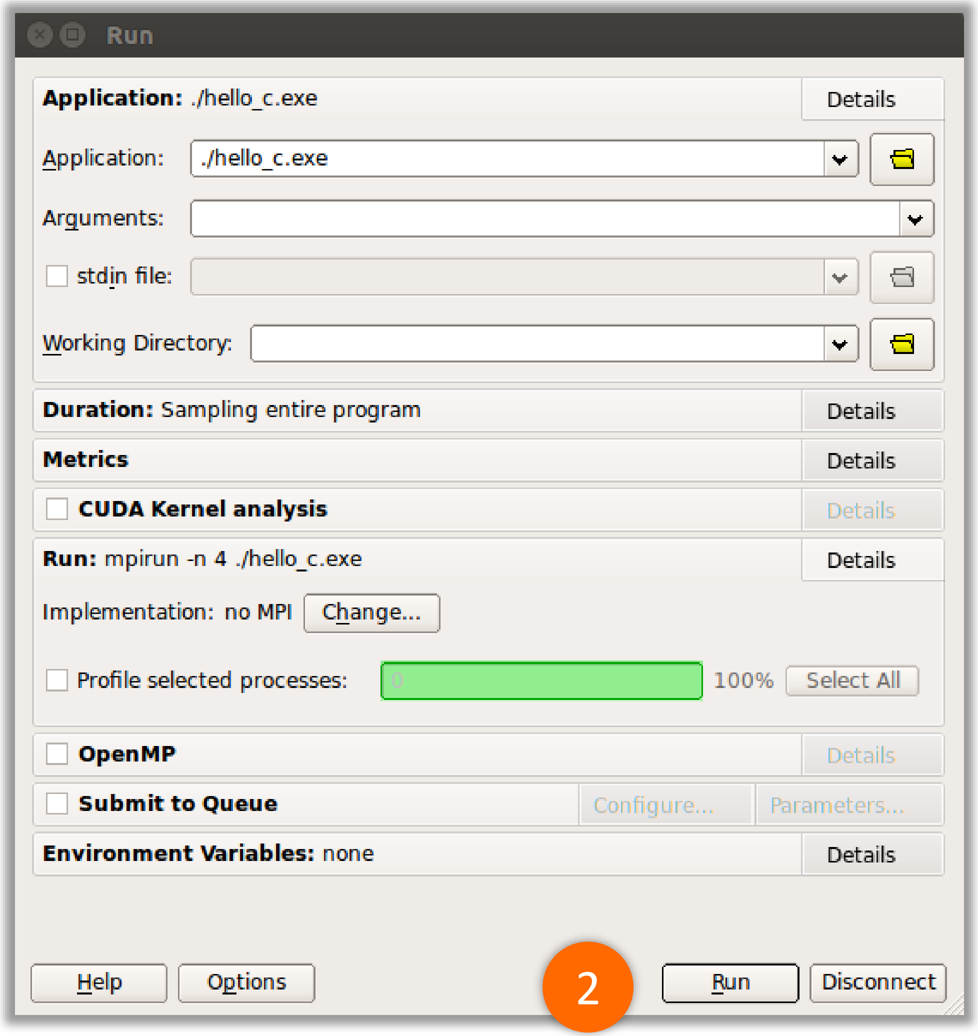
Reverse-Connect
Client ready

arm

Reverse-Connect – Server / Cluster side

```
ssh conhil01@tegner.pdc.kth.se
module load i-compilers
module load intelmpi
module load allinea-forge
cp /afs/pdc.kth.se/home/c/conhil01/Public/arm_trial.tar.gz .
tar -xvf arm_trial.tar.gz
cp /afs/pdc.kth.se/home/c/conhil01/Public/Licence_kth .
unset ALLINEA_LICENSE_FILE_modshare
unset ALLINEA_LICENSE_FILE
export ALLINEA_FORCE_LICENCE_FILE=$PWD/Licence_kth
cd arm_trial
cd 0_test_reverse_connect
make
salloc -nodes=1 -t 00:10:00 -A pdc-test-2018
map --connect  mpirun -n 2 ./hello_c.exe
```

Reverse-Connect – Client / Laptop side



arm

Hands – On :
Launch Perf-Reports

arm

Launch Performance Reports

```
ssh conhil01@tegner.pdc.kth.se
module load i-compilers
module load intelmpi
module load allinea-reports
cp /afs/pdc.kth.se/home/c/conhil01/Public/arm_trial.tar.gz .
tar -xvf arm_trial.tar.gz
cp /afs/pdc.kth.se/home/c/conhil01/Public/Licence_kth .
unset ALLINEA_LICENSE_FILE_modshare
unset ALLINEA_LICENSE_FILE
export ALLINEA_FORCE_LICENCE_FILE=$PWD/Licence_kth
cd arm_trial
cd 0_test_reverse_connect
make
salloc -nodes=1 -t 00:10:00 -A pdc-test-2018
perf-report mpirun -n 2 ./hello_c.exe
```


Visualize Performance Reports outputs

- Two files outputted : .txt and .html
- .txt can be visualized on the cluster with file editor
- Use scp to copy the .html file back to your laptop
- Open it with a Web Browser

Hands – On : Vectorization

arm

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

Computational Intensity

“My program is doing a lot of computation … How do I make it go faster”

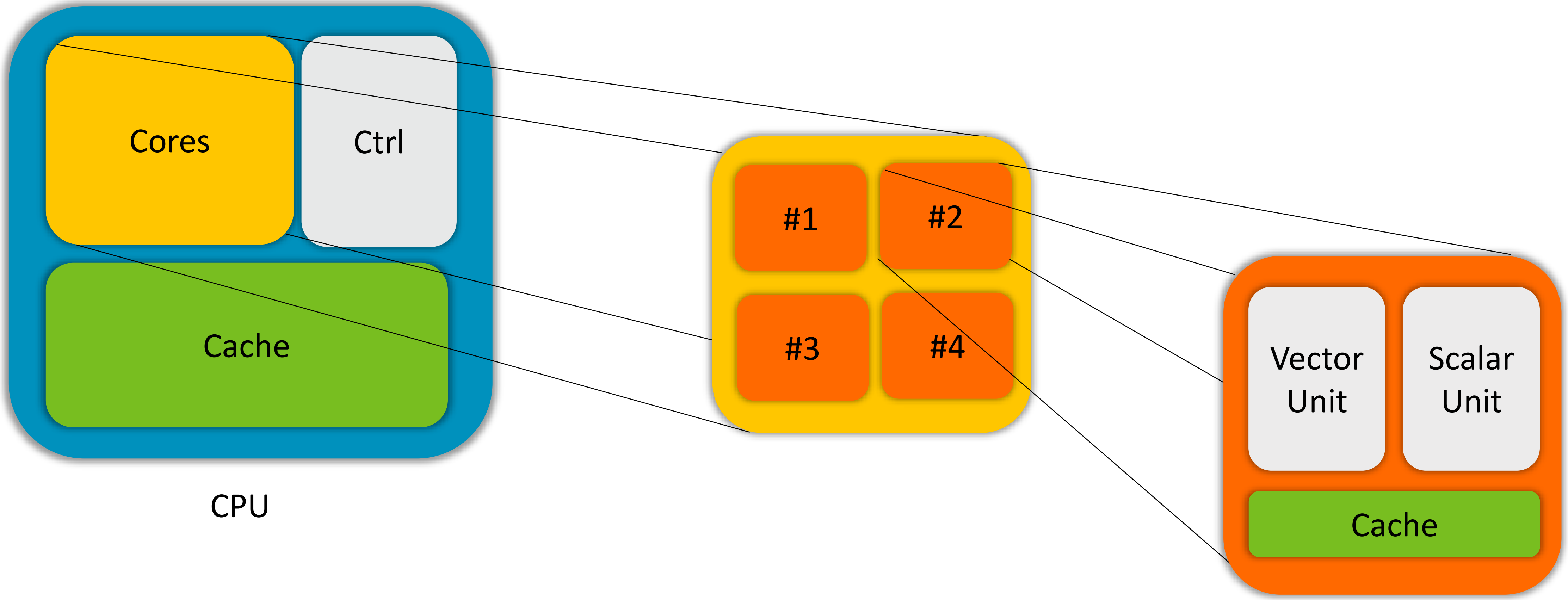
```
…

DO k=y_min-2,y_max+2

     DO j=x_min-2,x_max+2

      pre_vol(j,k)=volume(j,k)+(vol_flux_x(j+1,k  )-vol_flux_x(j,k)+vol_flux_y(j  ,k+1)-vol_flux_y(j,k))

      post_vol(j,k)=pre_vol(j,k)-(vol_flux_x(j+1,k  )-vol_flux_x(j,k))

     ENDDO

ENDDO

…
```

Example with modified version of CloverLeaf

- non-threaded version without OpenMP
- MPI, no IO

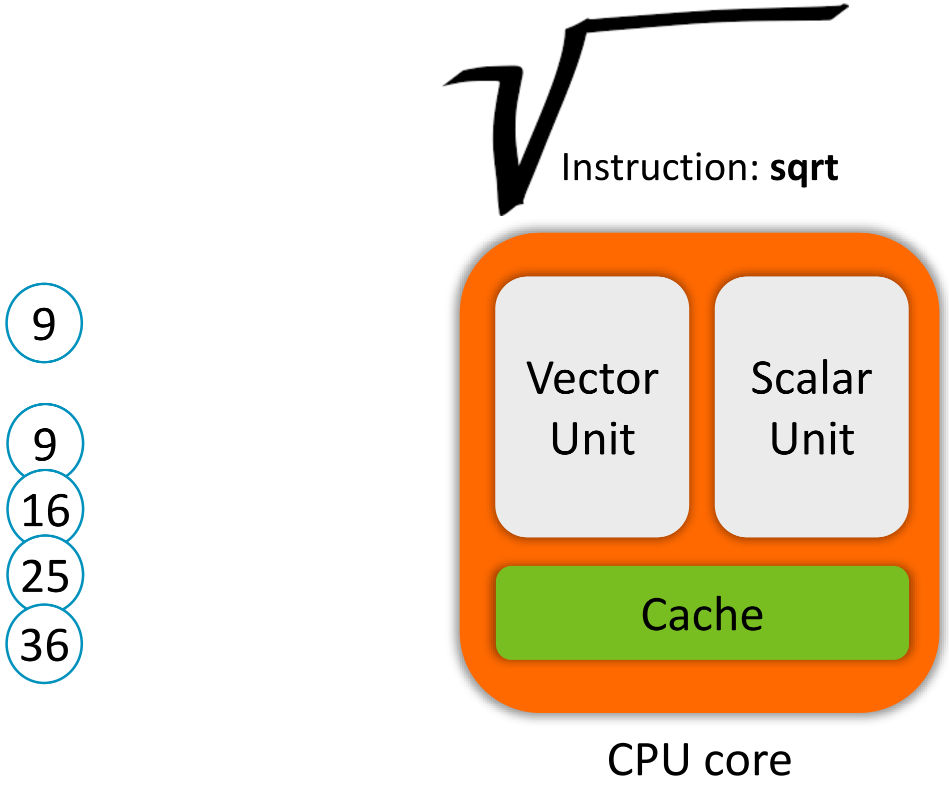
arm

Vector Units

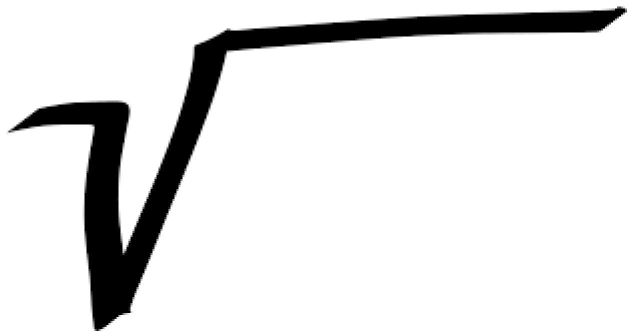


arm

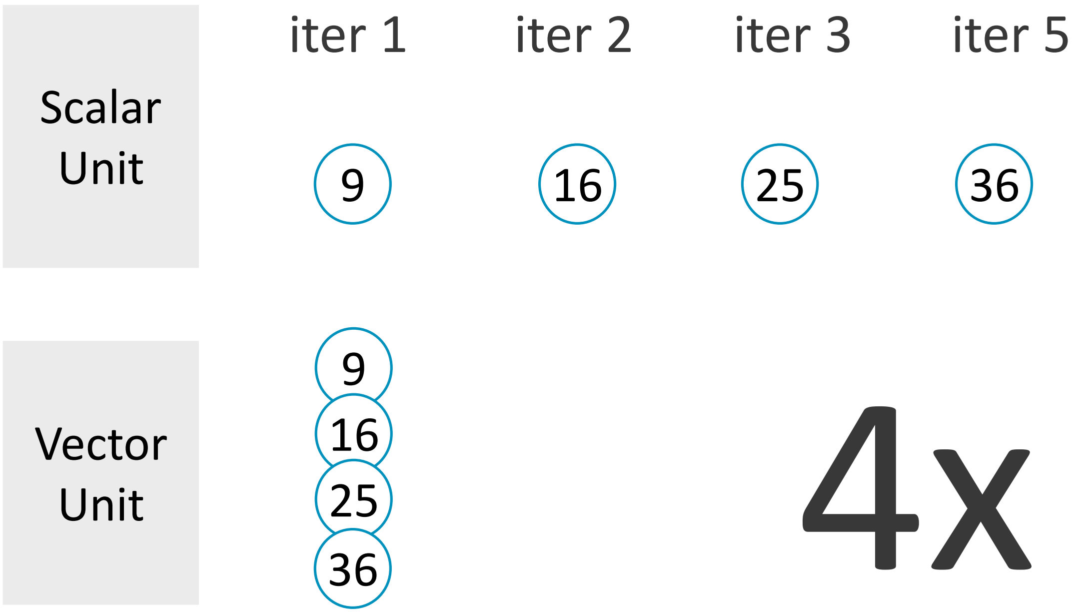
Vectorization / SIMD



Vectorization / SIMD

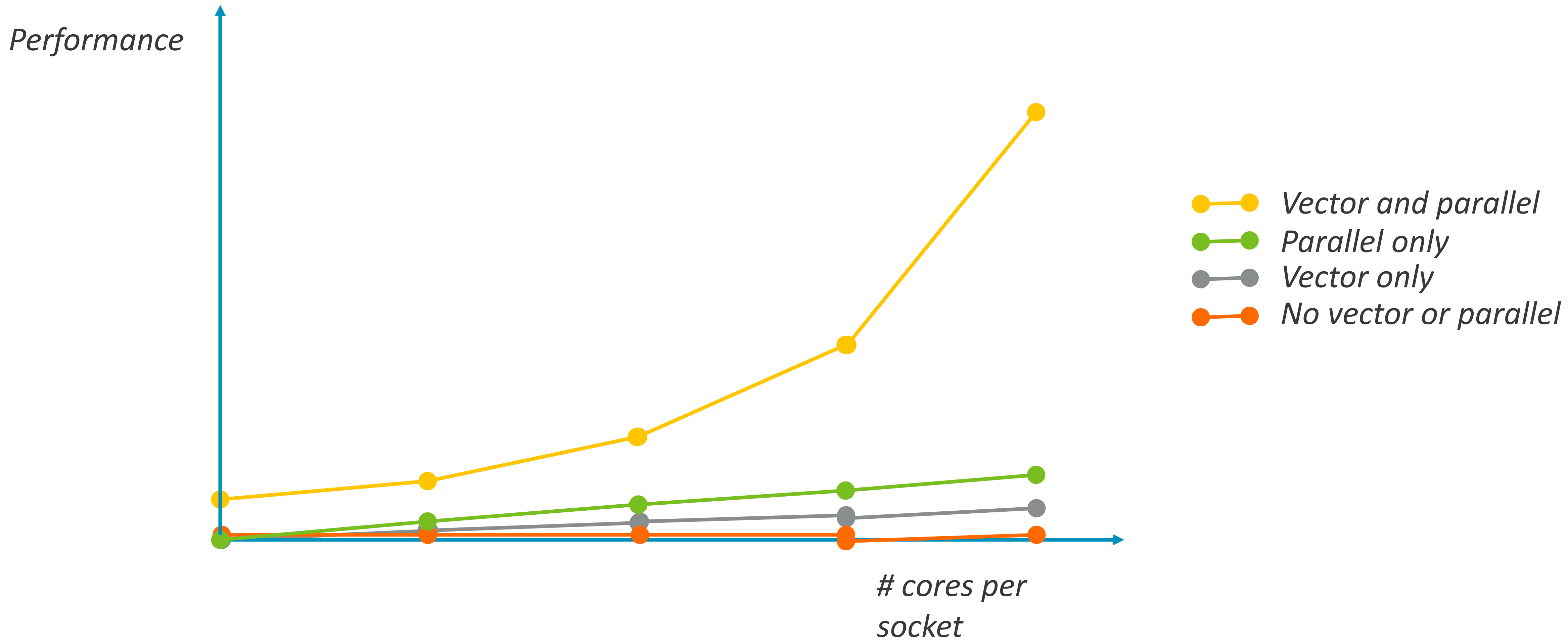


```
do i=1,n
  a(i) = sqrt(b(i))
end do
```



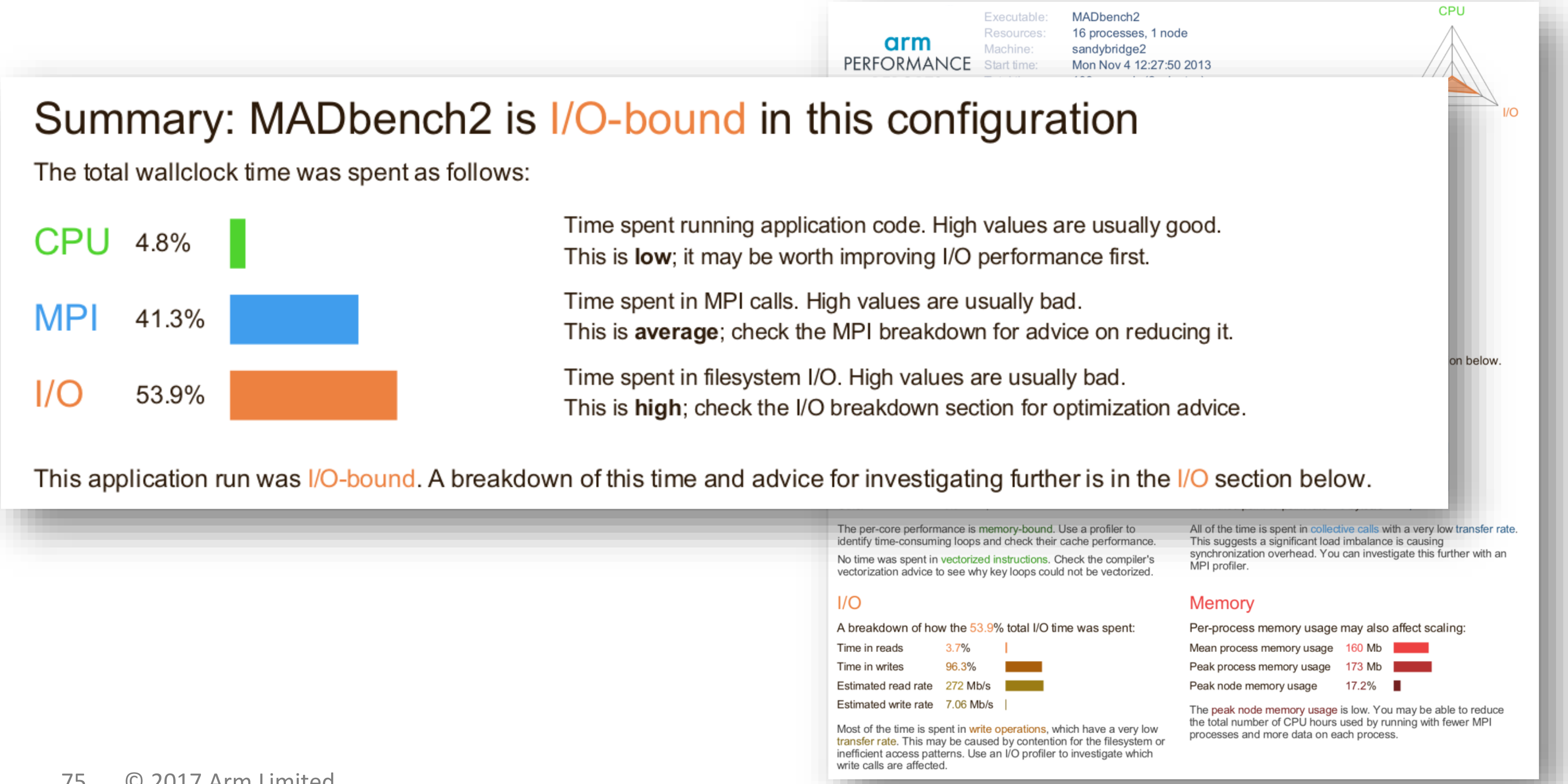
Intel® AVX2: 256-bit vector unit → 8 SP / 4 DP
Intel® AVX-512: 512-bit vector unit → 16 SP / 8 DP
Arm® NEON: 128-bit vector unit → 4 SP / 2 DP

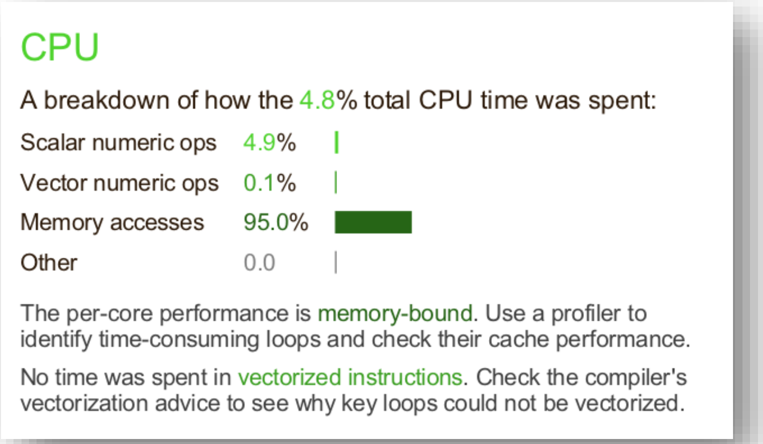
Why? Performance lies in the software



Identifying the amount of vectorized code

- Arm Performance Reports is an application reporting tool for HPC
  - Easy to use: no re-compiling required
  - Gives a comprehensible and readable summary of the application behavior





Analyze the results

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

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

Summary: clover_leaf is Compute-bound in this configuration

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

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

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

CPU

A breakdown of the 93.4% CPU time:

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

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

Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 6.6% MPI time:

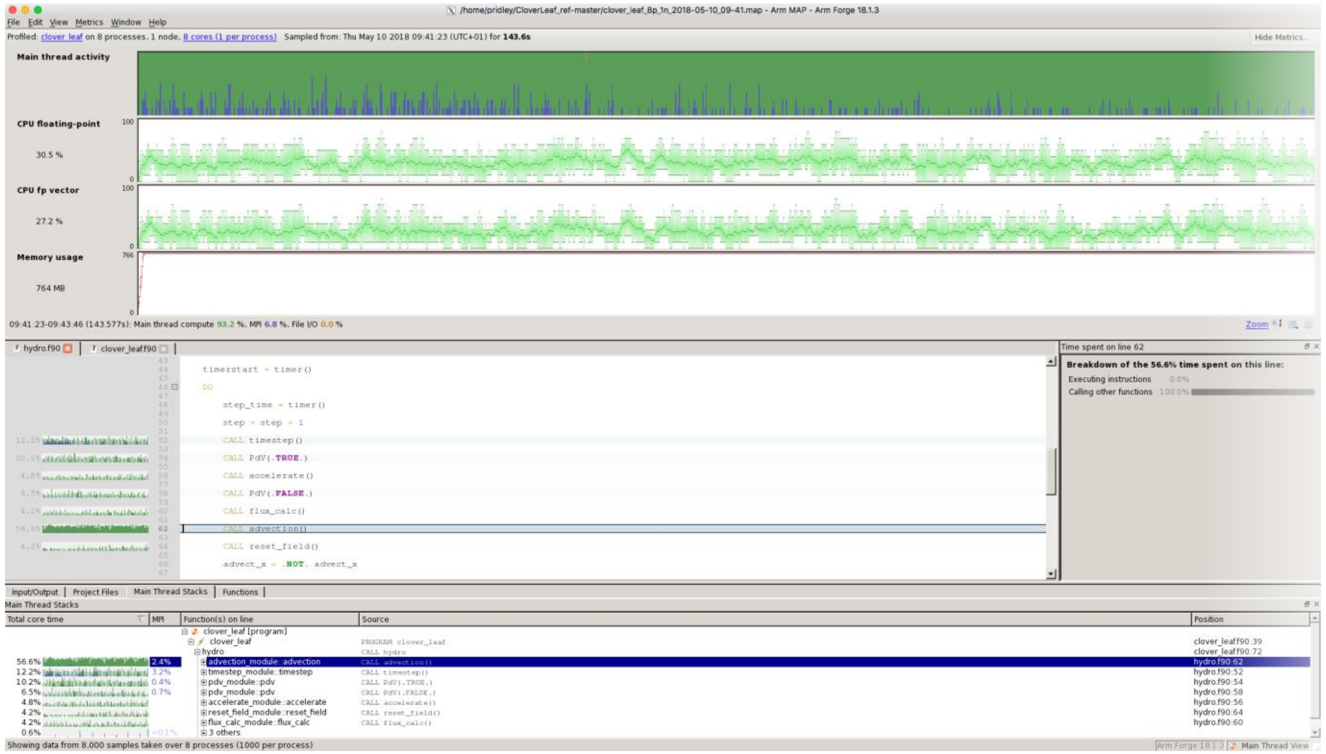
Time in collective calls	20.9%
Time in point-to-point calls	79.1%
Effective process collective rate	1.55 kB/s
Effective process point-to-point rate	33.1 MB/s

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

When? Time to use a profiler

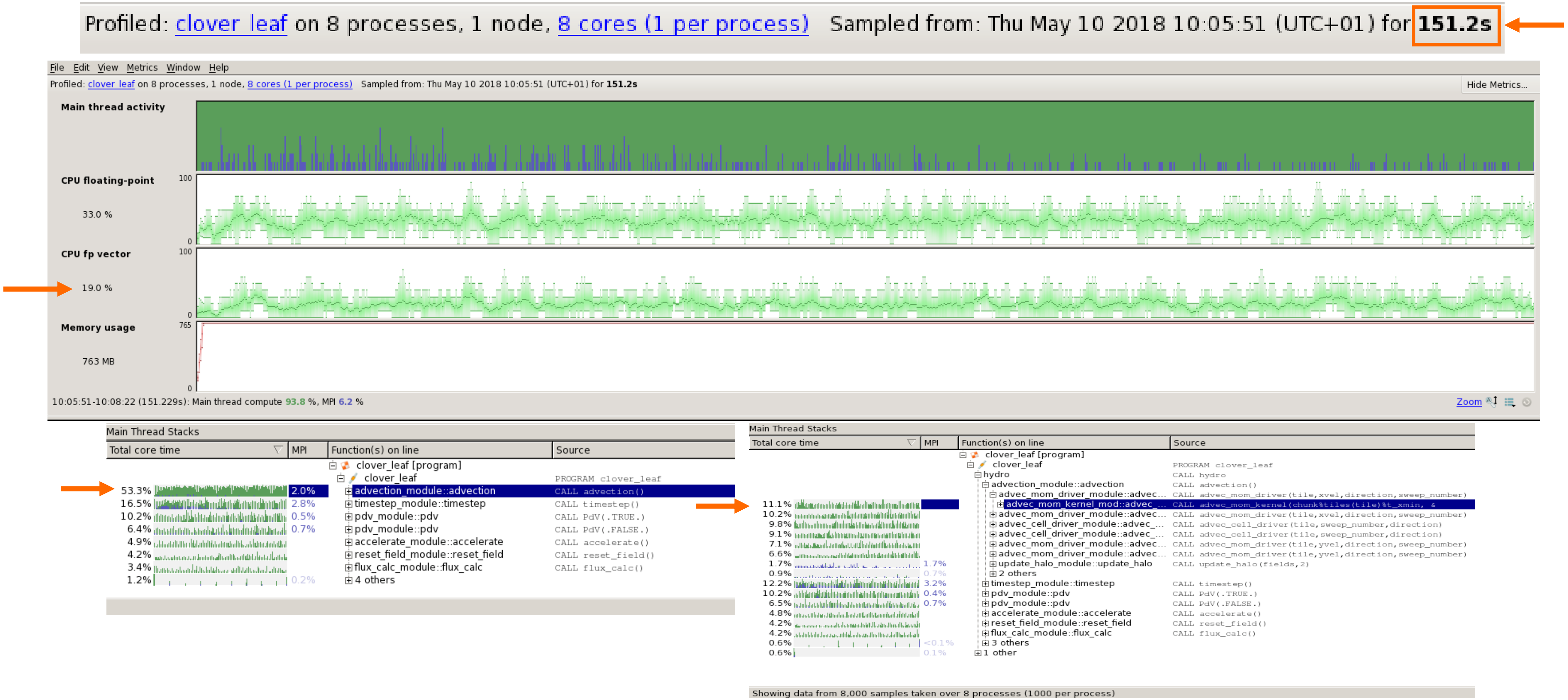
Arm MAP is a lightweight multi-node profiling tool

- Compiling with debugging flag required
- Shows processes and threads activity over time
- Source code is annotated
- Information aggregated by stacks and function

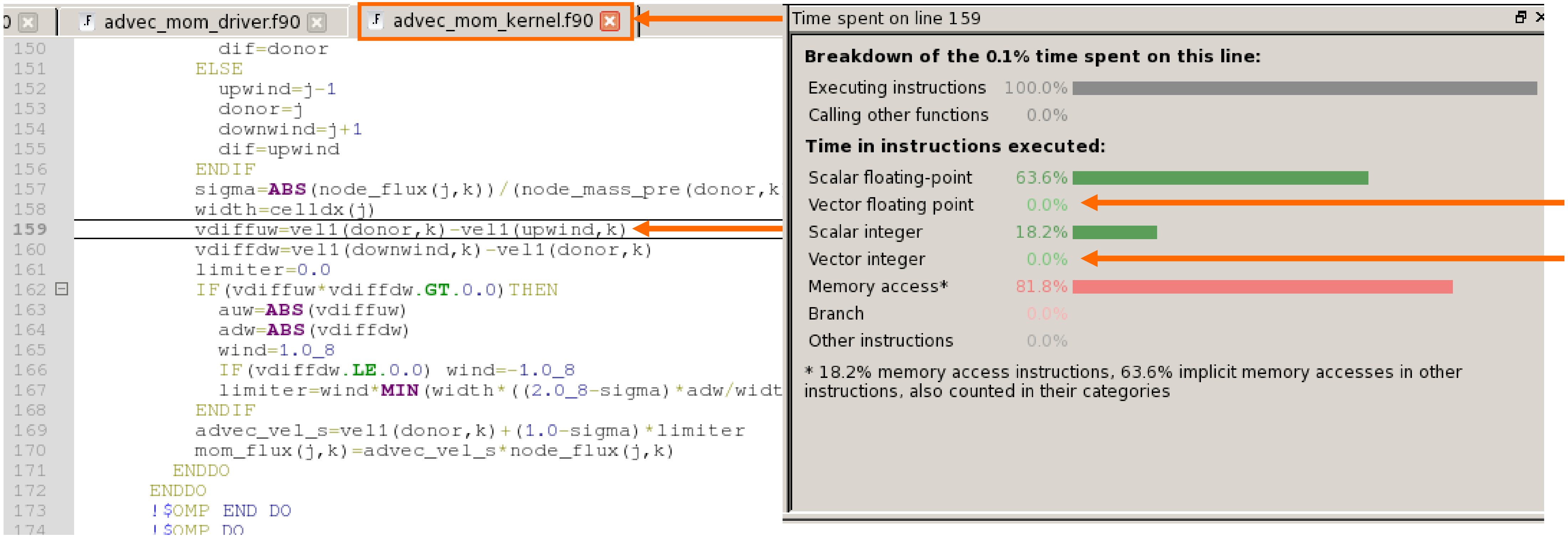


Compute, **IO** and **MPI**

How much of the code is vectorized?

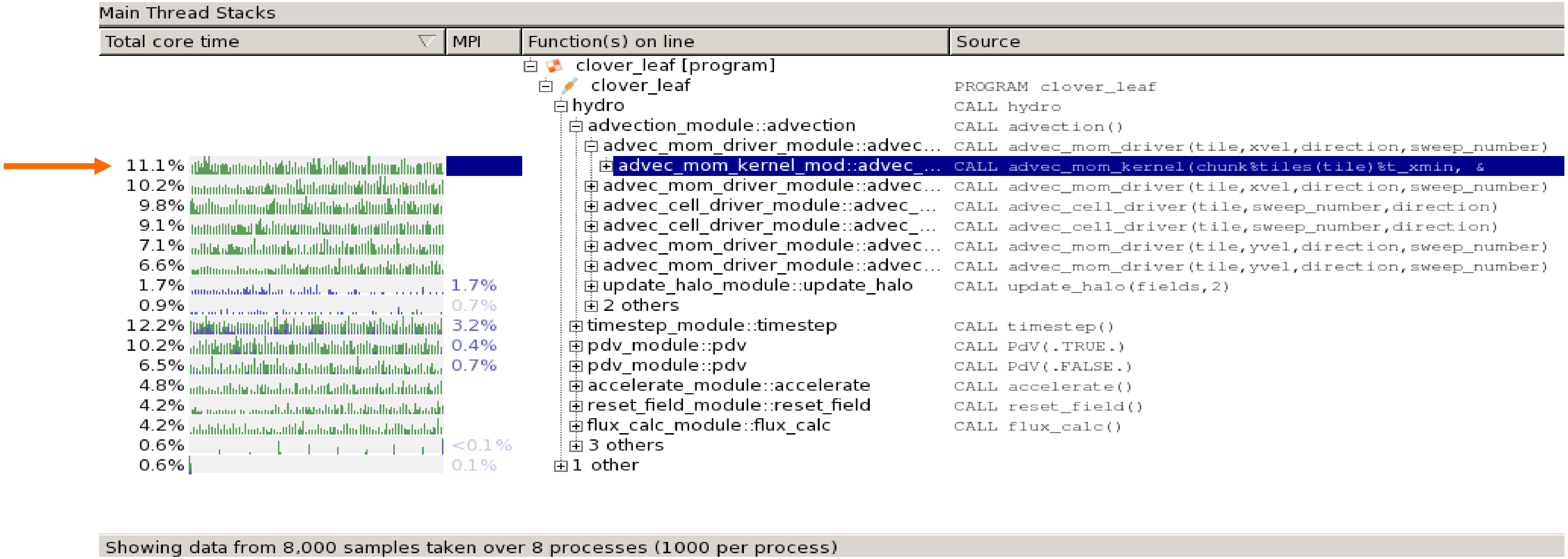


Where is the code vectorized?



arm

Follow Performance Reports advice



arm

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

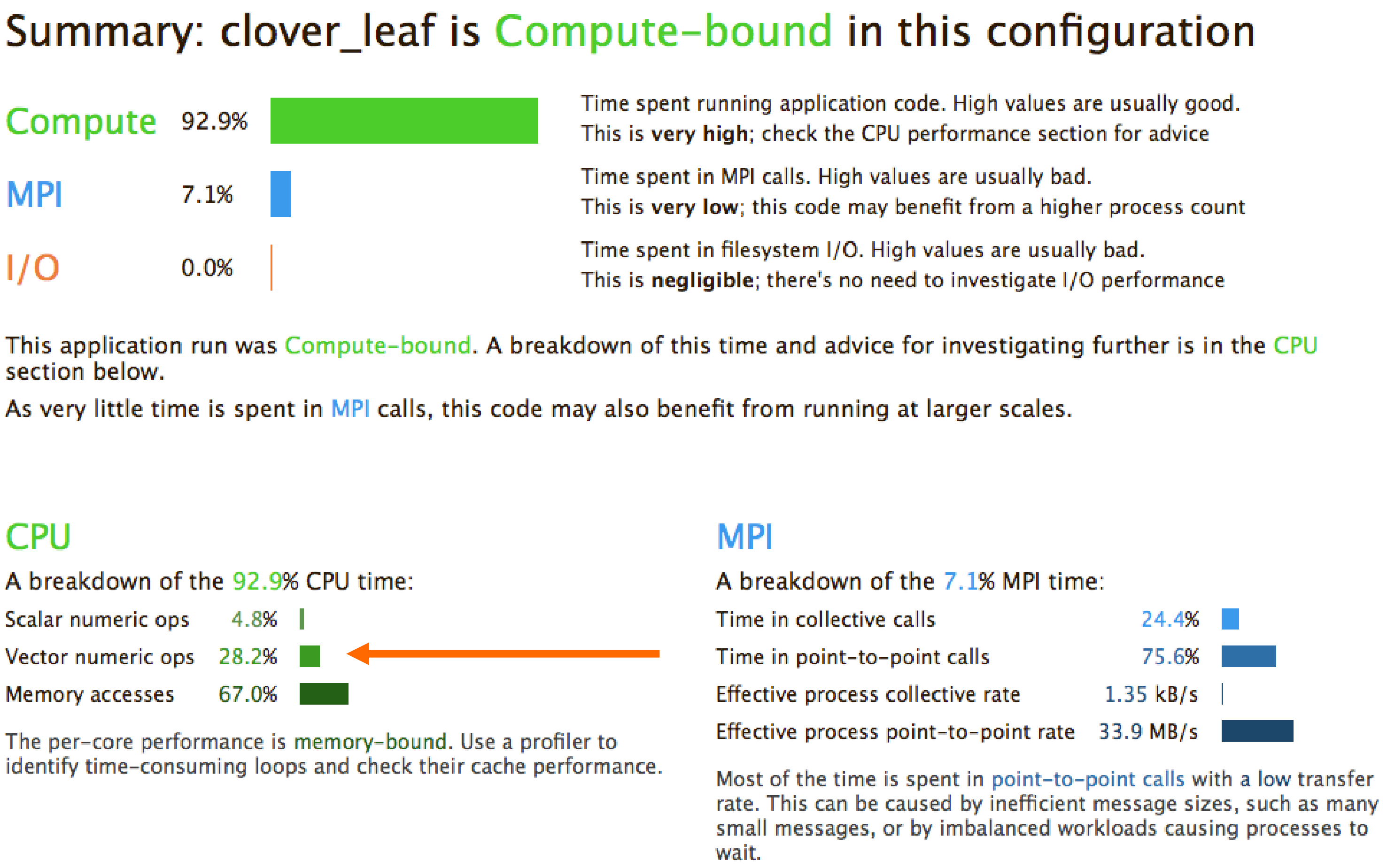
```
...
144     DO k=y_min,y_max+1
145       DO j=x_min-1,x_max+1
146         IF(node_flux(j,k).LT.0.0)THEN
147           upwind=j+2
148           donor=j+1
149           downwind=j
150           dif=donor
151         ELSE
152           upwind=j-1
153           donor=j
154           downwind=j+1
155           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

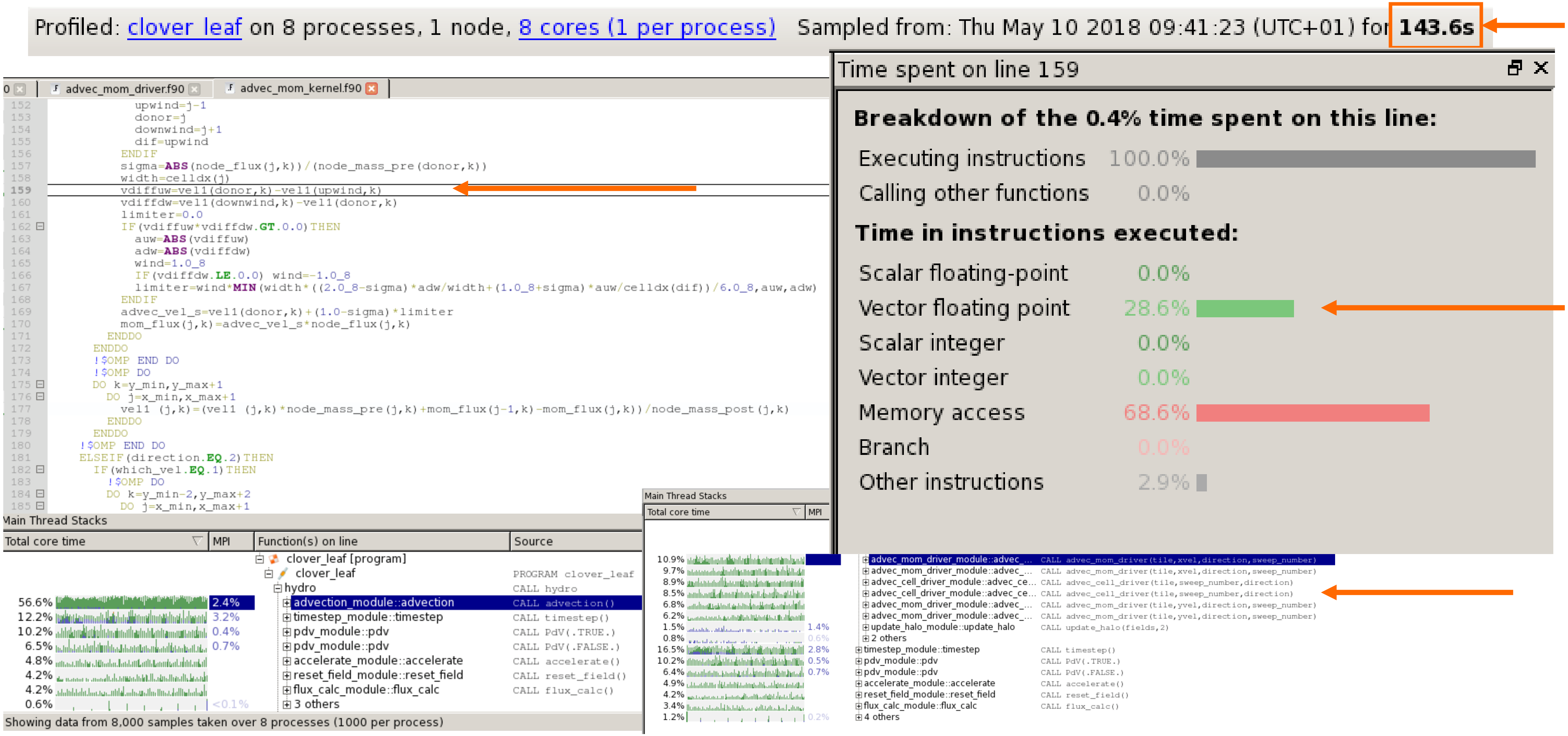
Running Performance Reports with CloverLeaf using 8 MPI tasks indicates that:

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



arm

Where is the code vectorized?



arm

How?

Different compilers may have different capabilities, but here are guidelines

- Remove conditionals inside loop
- Make sure that loop size is known on entry
- Pay attention to work on contiguous, unit-stride arrays
- Remove data dependencies to enable vectorization
- Use compiler directives to force loop vectorization

Conclusion

Vectorizing an application is a difficult task

Arm Performance Reports and Arm MAP make it easier

- Analyze application efficiency and get advices with Performance Reports
- Identify bottlenecks and line by line performance with MAP

Figure out quickly if your application uses vectorization

Find candidates for vectorization

Inspect vectorization over time

Hands – On

- 2_profiling_compute
- Compile the code
- Is the code well vectorized ? (with Arm Performance Reports)
- Identify where and how it can be improved (with Arm MAP)
- Modify the code and recompile
- Has vectorization increased ? Do you see any speed-up ? (with Arm Performance Reports and Arm MAP)

arm

Hands – On : Workload Imbalance

arm

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:

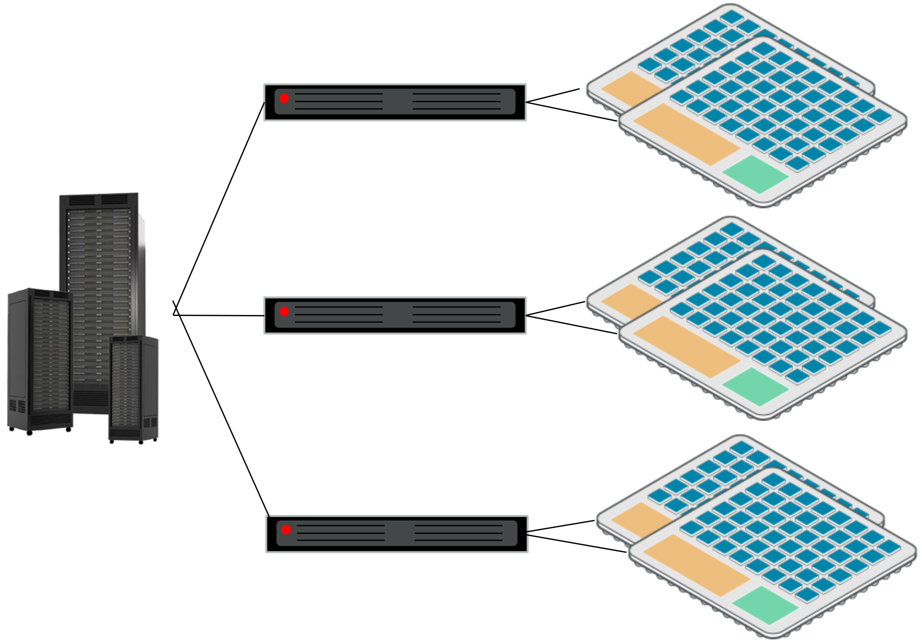
- ✔ arm PERFORMANCE REPORTS
- ✔ arm FORGE

Workload balancing: definition

- *"Aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource."*

(Wikipedia)

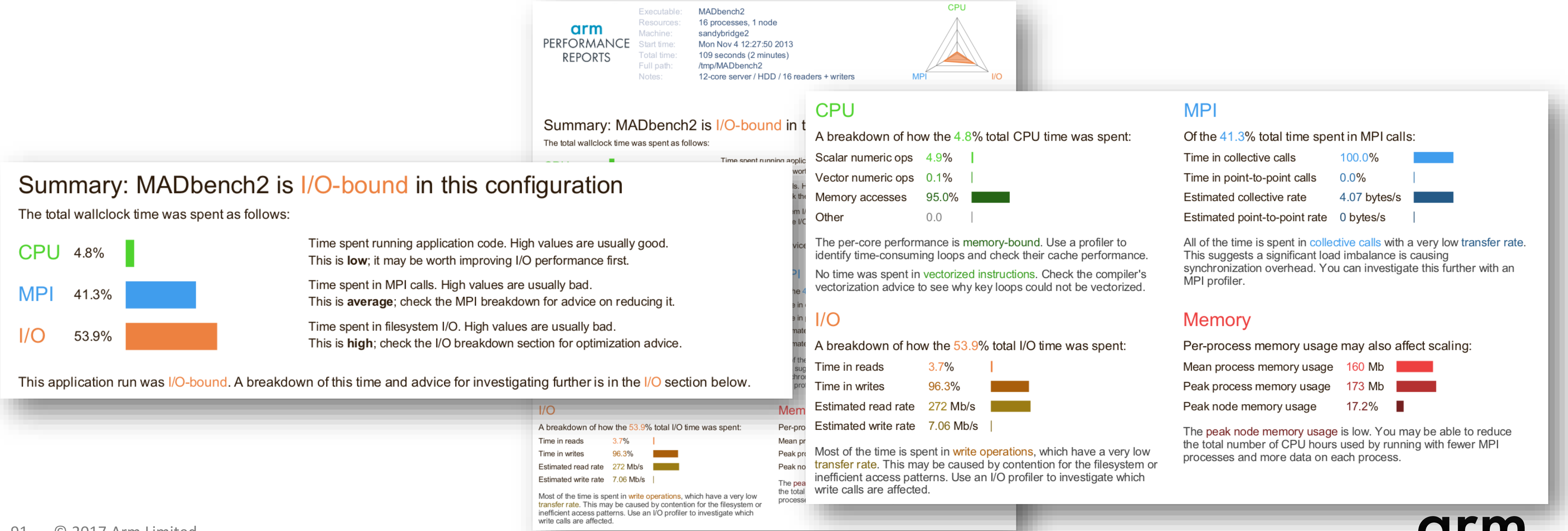
- In HPC, a well balanced workload across:
  - Multiple nodes over a high-speed network,
  - Multiple sockets,
  - Multiple NUMA systems
  - Multiple cores,
  - Multiple accelerators,
  - Multiple disk drives,
- Is critical for application performance



arm

Identify workload imbalance

- Arm Performance Reports is an application reporting tool for HPC
  - Easy to use: no re-compiling required
  - Gives a comprehensible and readable summary of the application behavior



MPI and OpenMP imbalance

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

MPI

Of the 41.3% total time spent in MPI calls:

Time in collective calls	100.0%
Time in point-to-point calls	0.0%
Estimated collective rate	4.07 bytes/s
Estimated point-to-point rate	0 bytes/s

All of the time is spent in collective calls with a very low transfer rate. This suggests a significant load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

OpenMP

A breakdown of the 74.5% time in OpenMP regions:

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

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

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

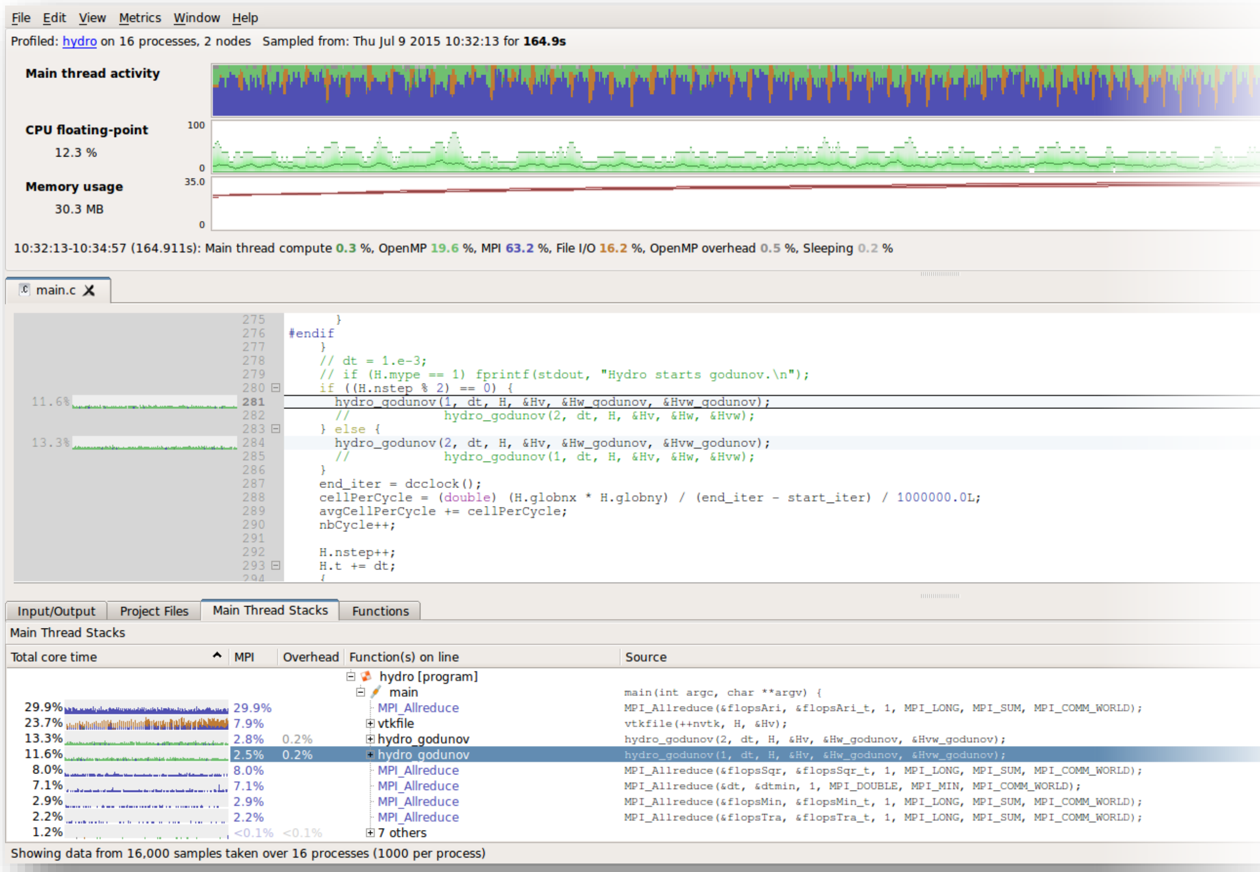
arm

Locate imbalance in your code

arm MAP

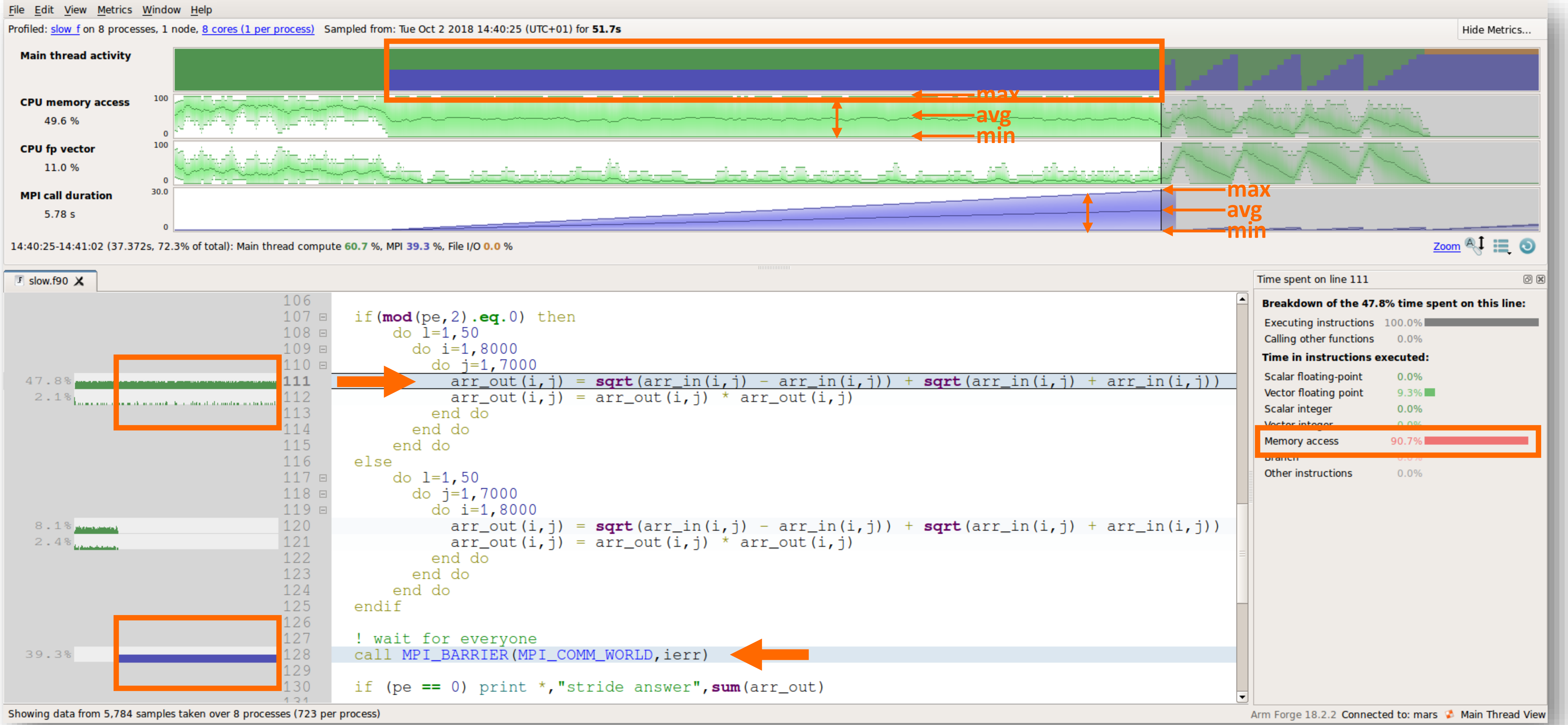
Arm MAP is a lightweight multi-node profiling tool

- Compiling with debugging flag required
- Shows processes and threads activity over time
- Source code is annotated
- Information aggregated by stacks and function



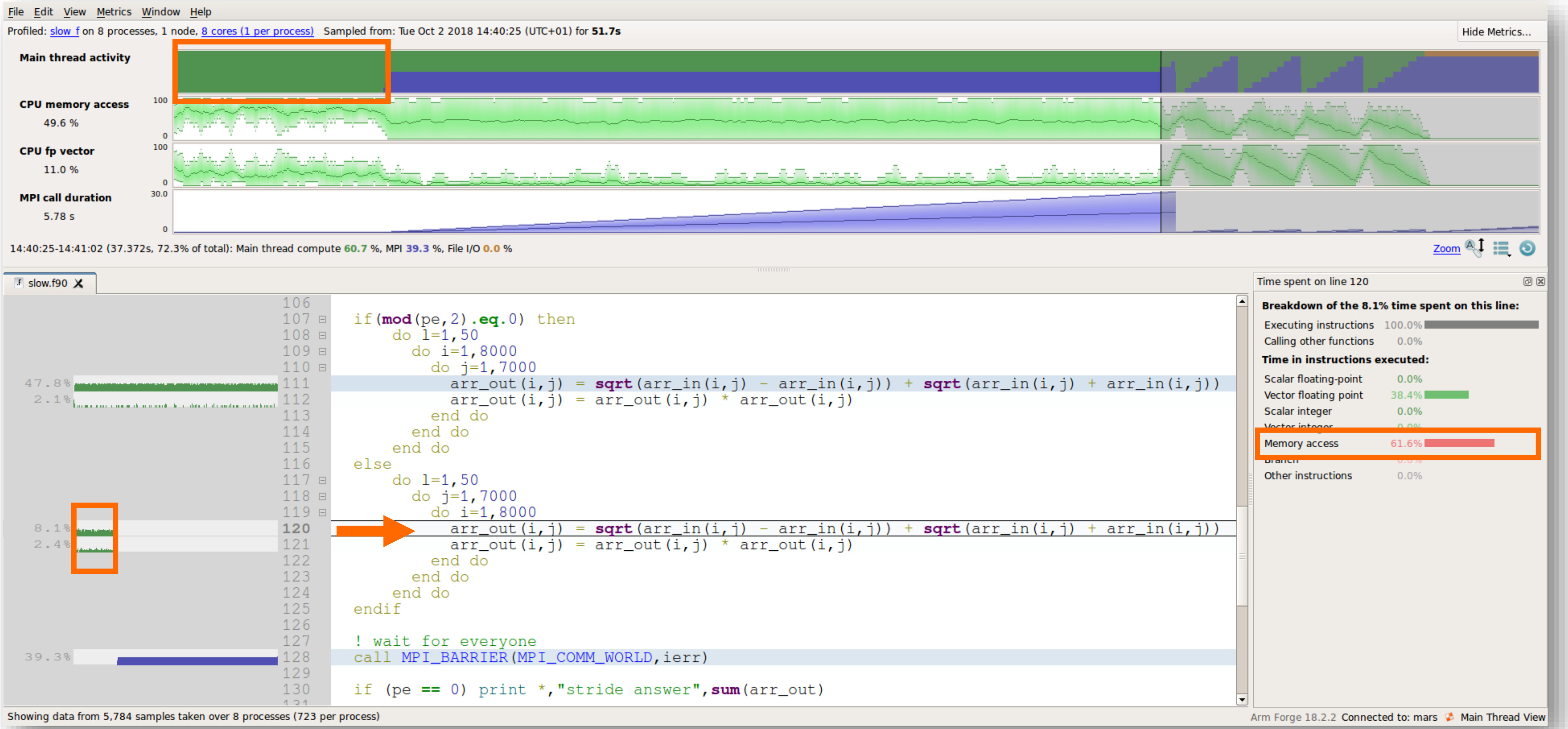
Compute, **IO** and **MPI**

MPI imbalance: barrier



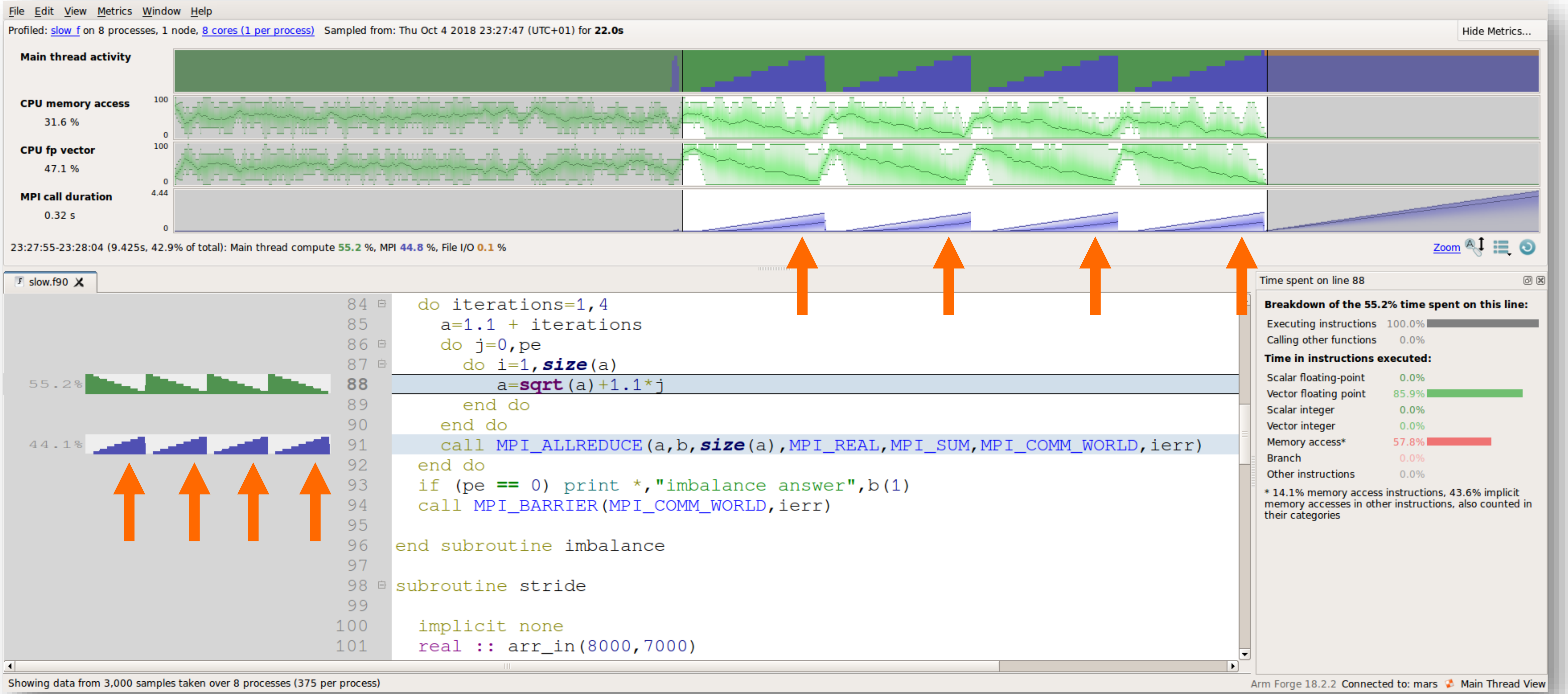
arm

MPI imbalance: barrier

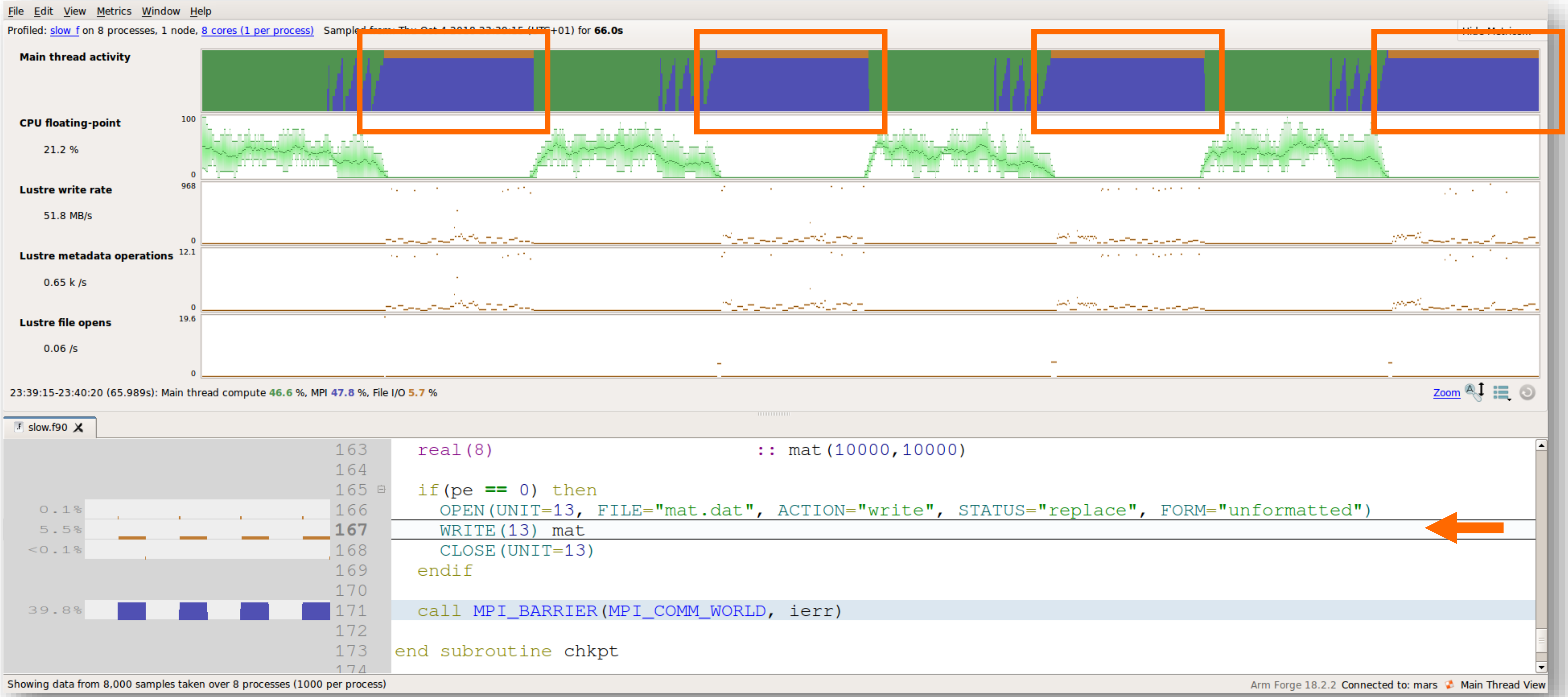


arm

MPI imbalance: all reduce



IO imbalance



Hands – On

- 4_profiling_imbalance
- Compile the code
- Are the MPI communications heavy ? (with Arm Performance Reports)
- Are the IOs efficient ? (with Arm Performance Reports)
- Identify where and how it can be improved (with Arm MAP)
- Modify the code and recompile
- Are the performances better ? (with Arm Performance Reports and Arm MAP)

Contact Support

arm

Issues with Arm Forge ? Our support team is here to help !

For any questions :
support-hpc-sw@arm.com

CC : conrad.hillairet@arm.com

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

