



# arm

## Improving your code with Arm Forge

PDC-PRACE Workshop  
“HPC Tools for the Modern Era”

# An introduction to Arm

Arm is the world's leading semiconductor intellectual property supplier

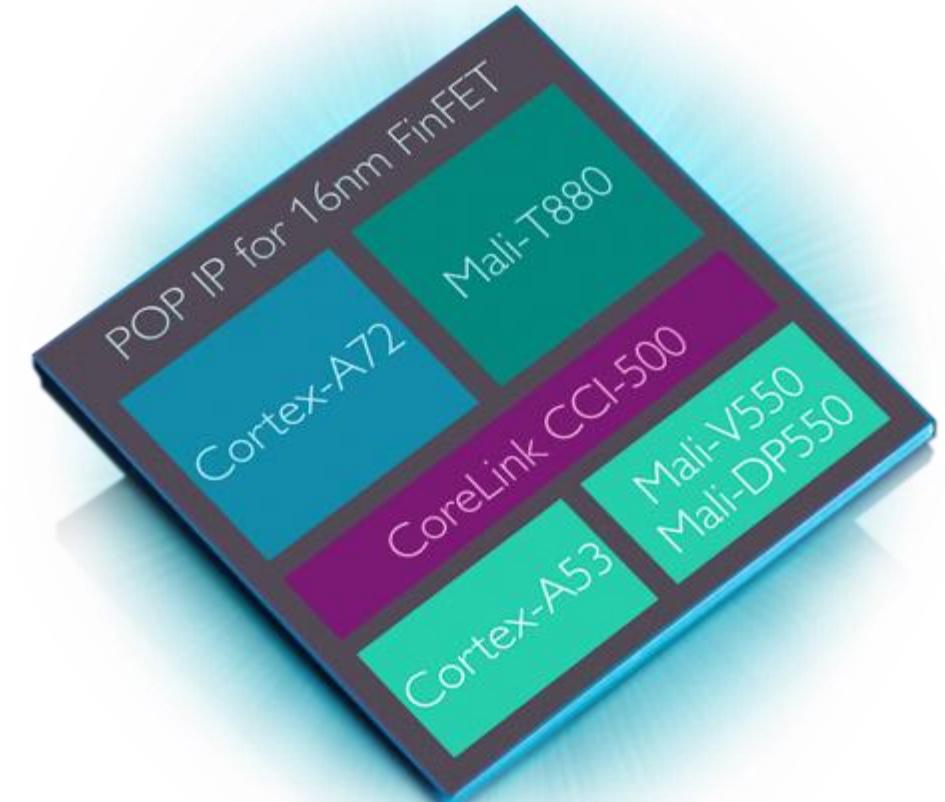
We license to over 350 partners: present in 95% of smart phones, 80% of digital cameras, 35% of all electronic devices, and a total of 60 billion Arm cores have been shipped since 1990

## 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

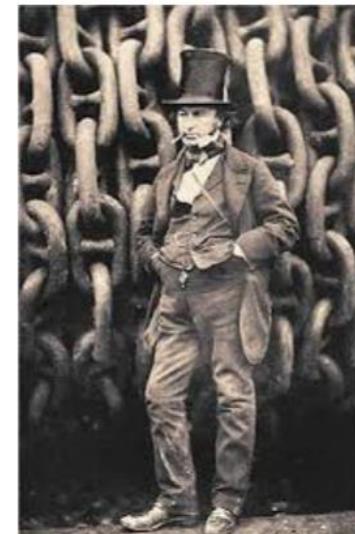


Isambard

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

# Catalyst UK

## Accelerating Arm adoption in the UK

### Sites and Target HPC

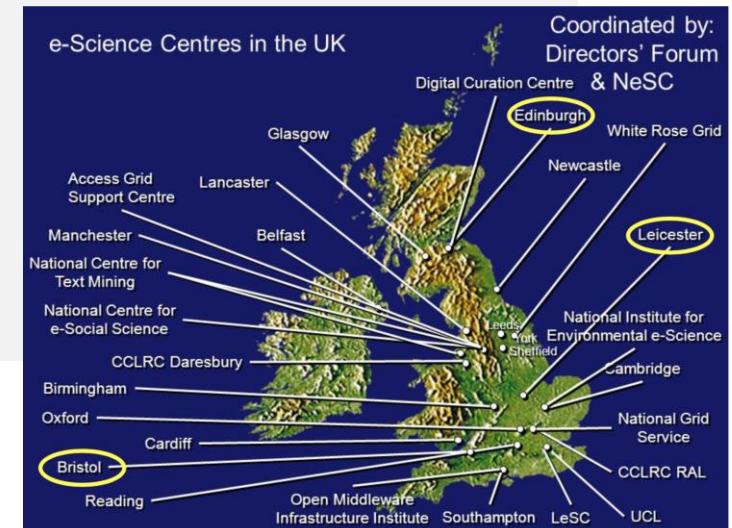
#### Applications:

- **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

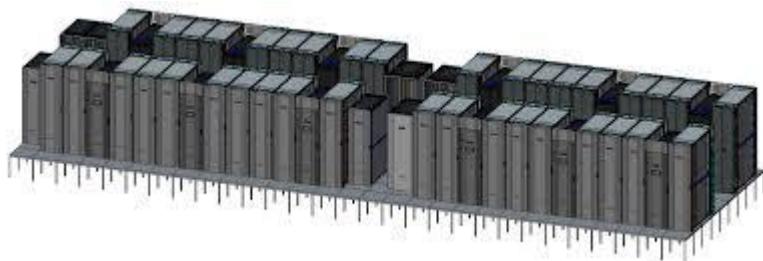
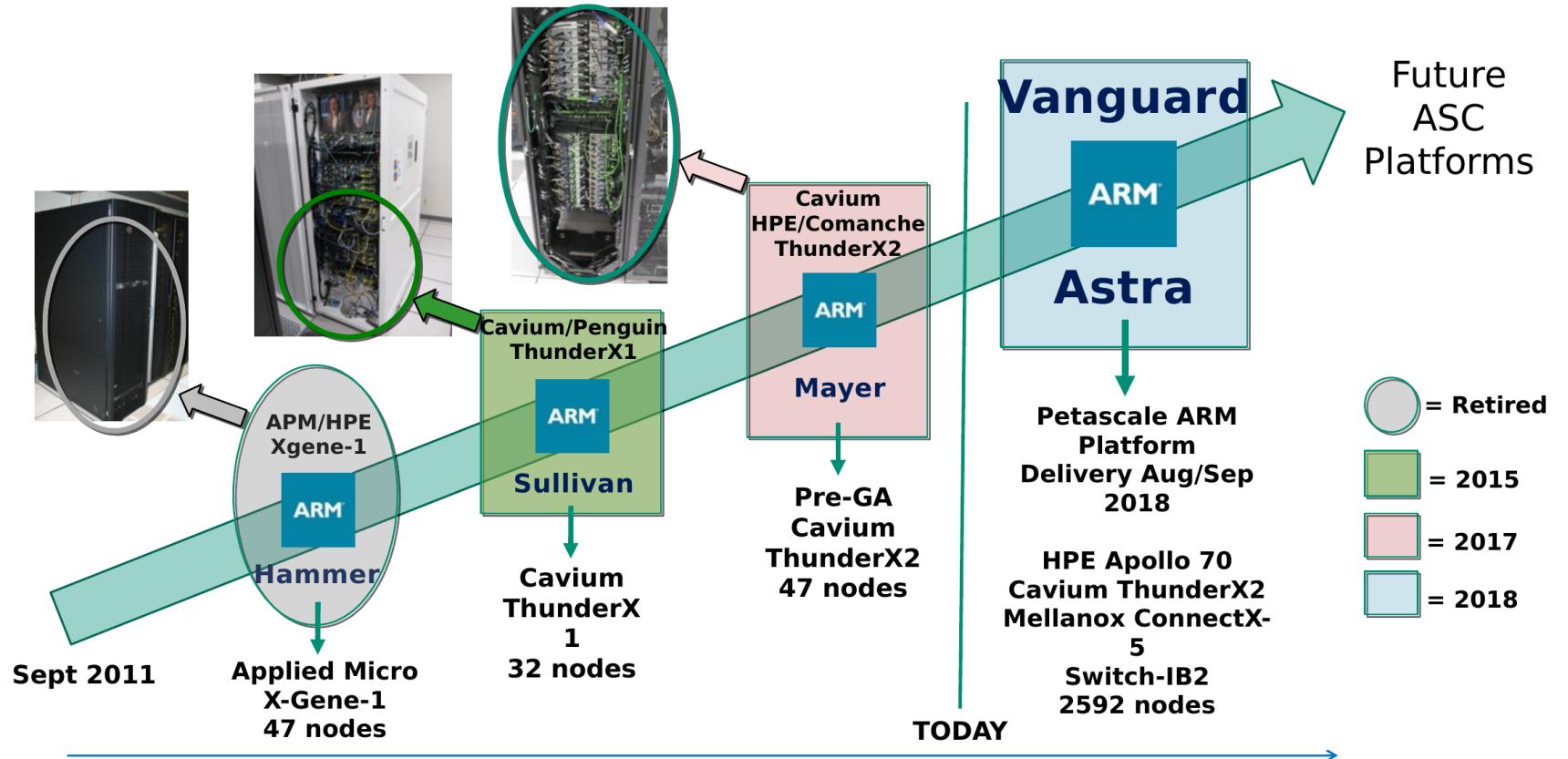


### Typical Cluster for each site:

- 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](#))



## Post-K: Fujitsu HPC CPU to Support ARM v8



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

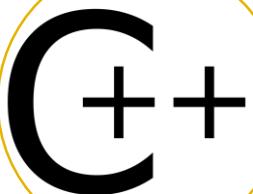
ARM v8 brings out the real strength of Fujitsu's microarchitecture

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...

# Conrad : Support Engineer - Arm Allinea Studio and Arm Forge

A quick glance at what is in Arm Allinea Studio



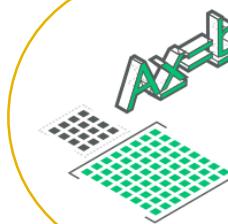
C/C++ Compiler  
AArch64

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

The word "Fortran" inside a yellow circle.

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

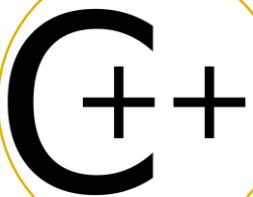


Performance Reports  
Cross Platform

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

# Conrad : Support Engineer - Arm Allinea Studio and Arm Forge

A quick glance at what is in Arm Allinea Studio



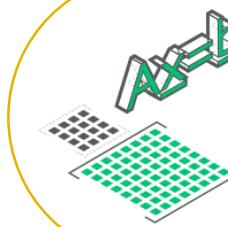
C/C++ Compiler  
AArch64

- 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

# Summary

# Summary :

## Overview

### Introduction

I

III

II

## Arm Performance Reports

### Arm MAP

Hands - On : Launch MAP

Hands - On : Launch Perf-reports

Hands - On : Vectorization

Hands - On : Workload Imbalance

## Arm DDT

Hands - On : Launch DDT

Hands - On : SIGFPE

Hands - On : Memory Debugging

# Overview

# Extra documentation

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

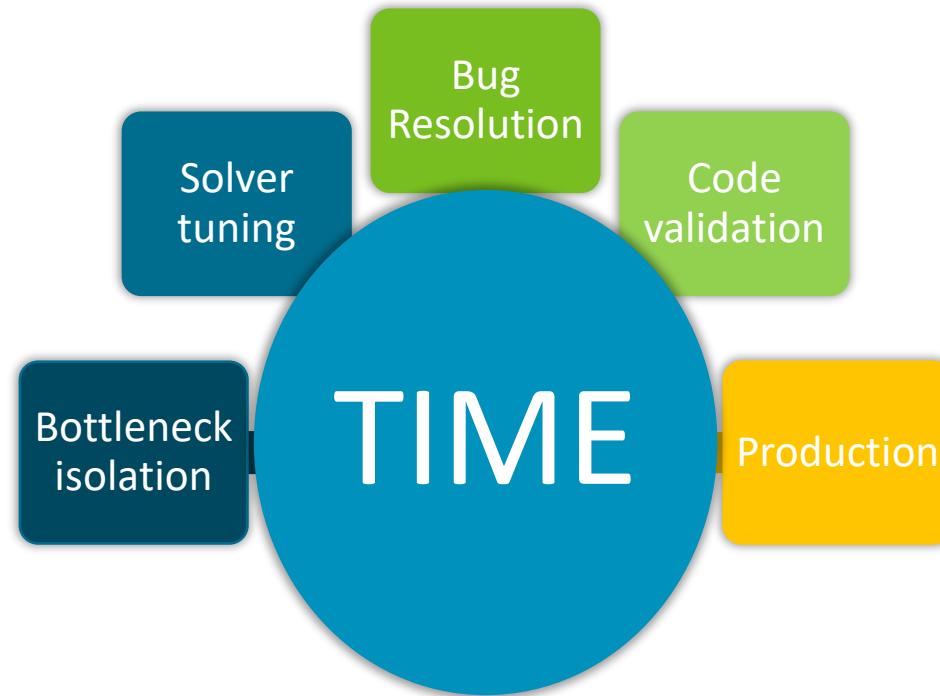
Arm DDT User Guide : <https://developer.arm.com/docs/101136/latest/ddt>

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

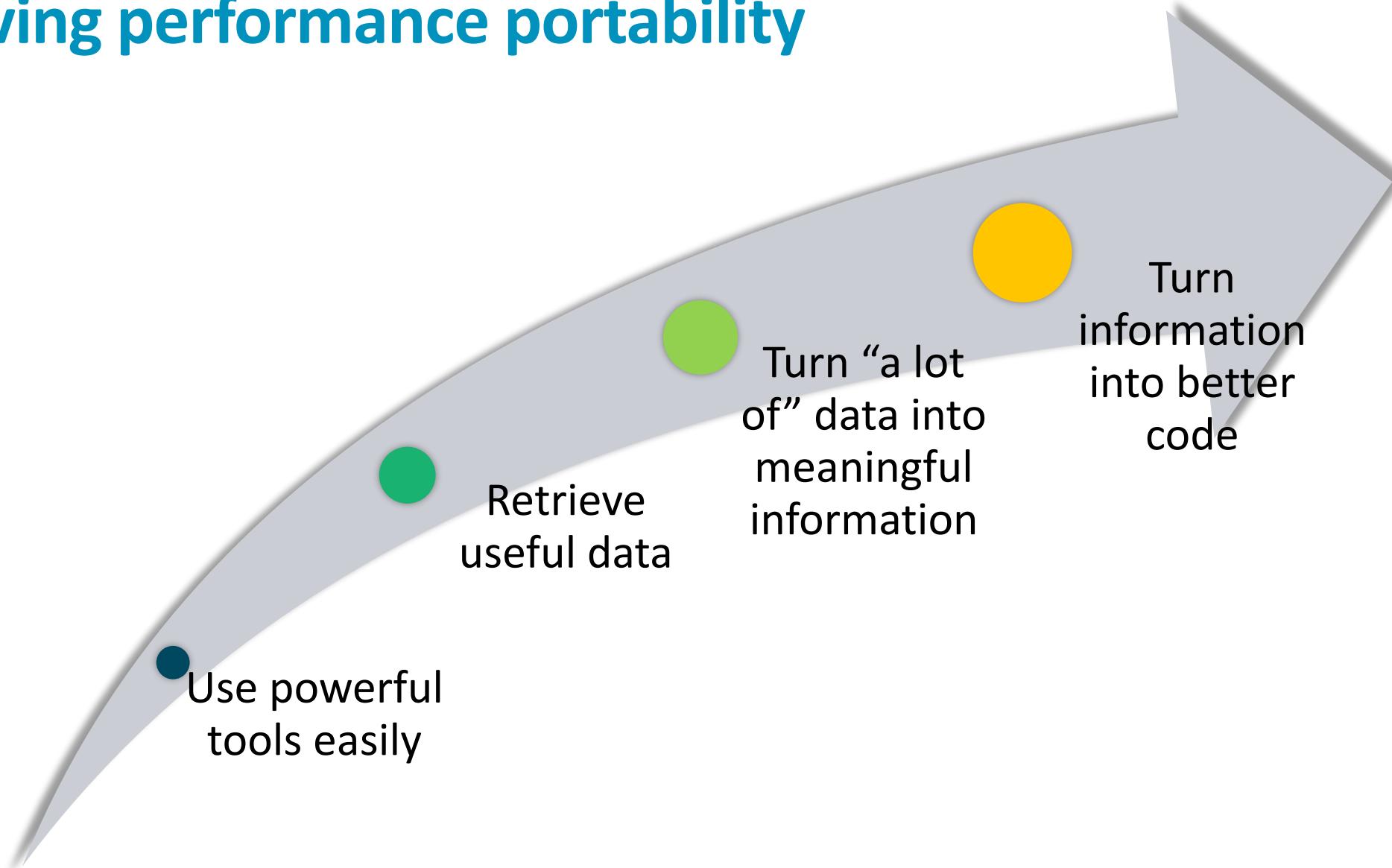
Arm Performance Reports User Guide : <https://developer.arm.com/docs/101137/latest/introduction>

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

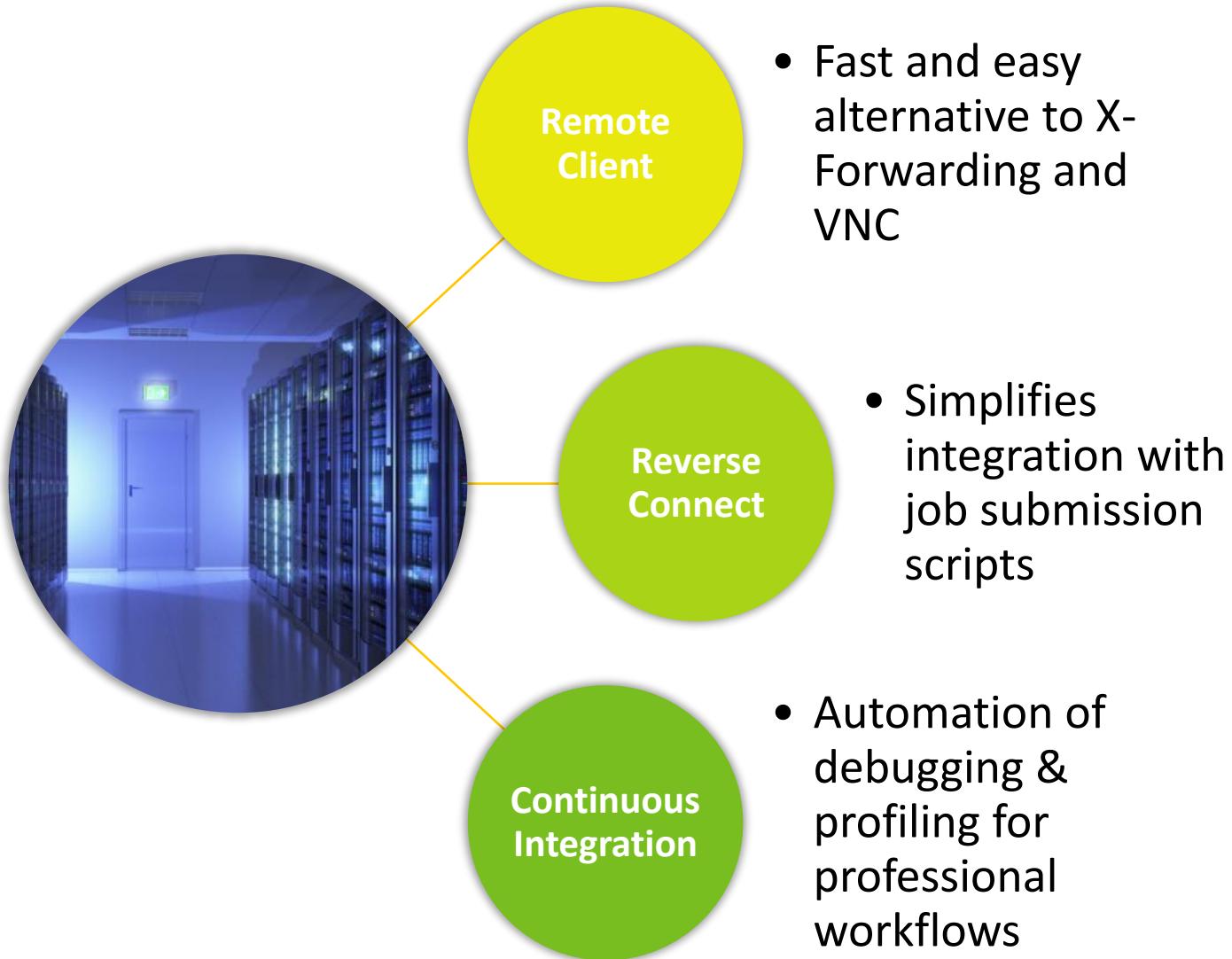
We do tools for a single reason:  
help people save their time.



# Achieving performance portability



# Using powerful tools more easily

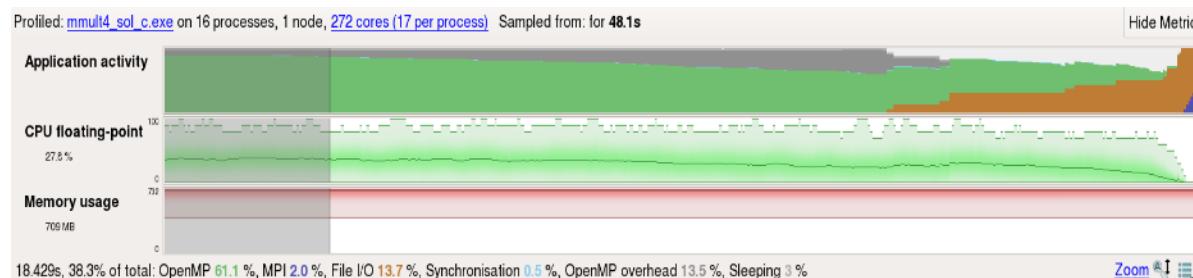
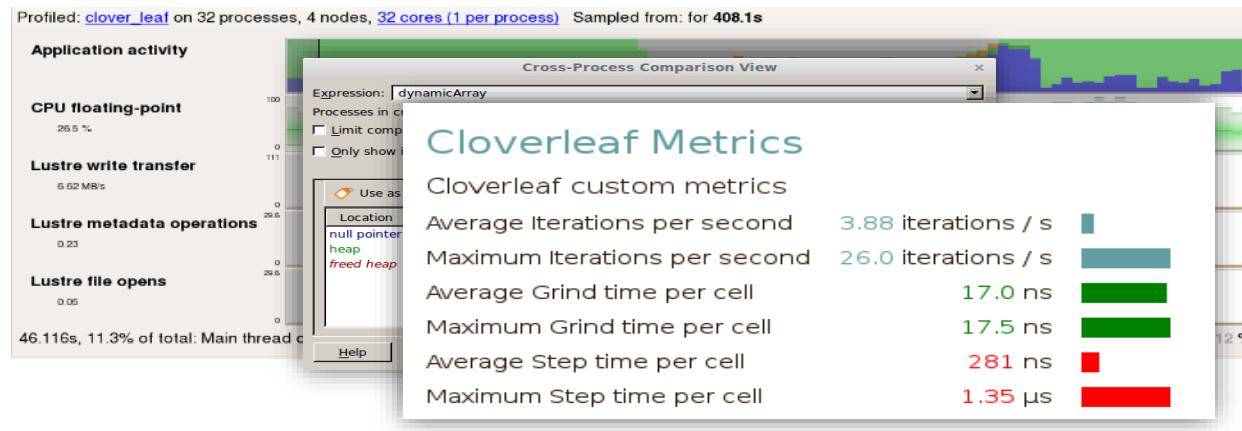
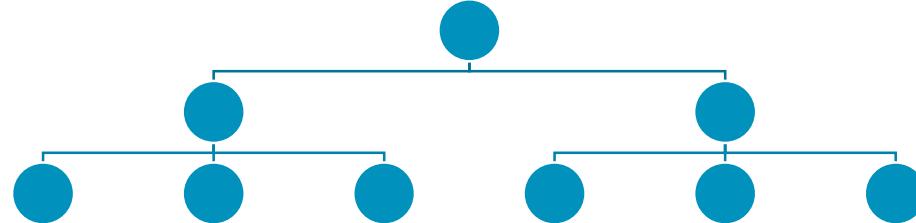


# Generating useful and meaningful information

Scalable &  
Portable

Data collection

Data  
processing



# 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



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.

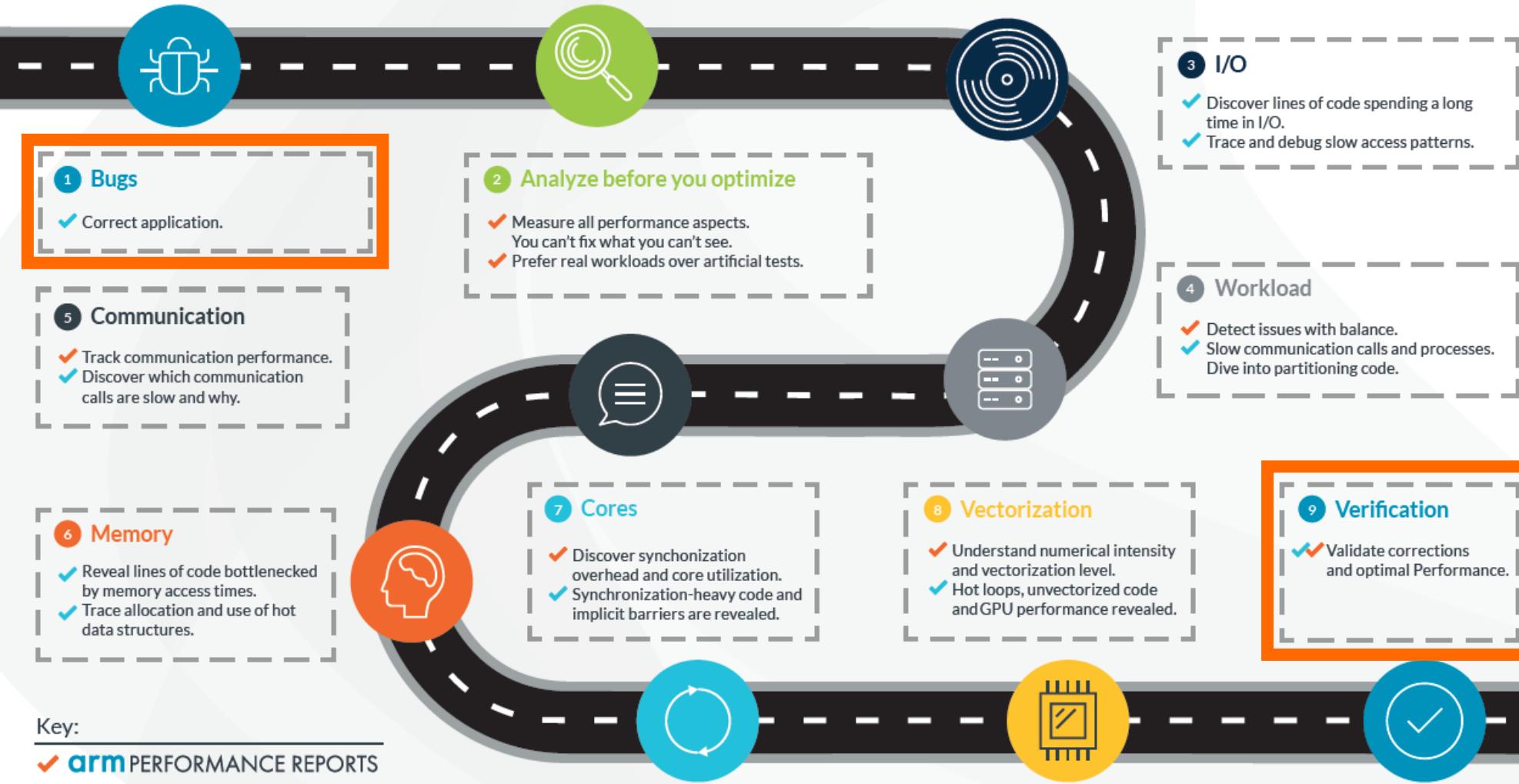


# Arm DDT

# 9 Step guide: optimizing high performance applications

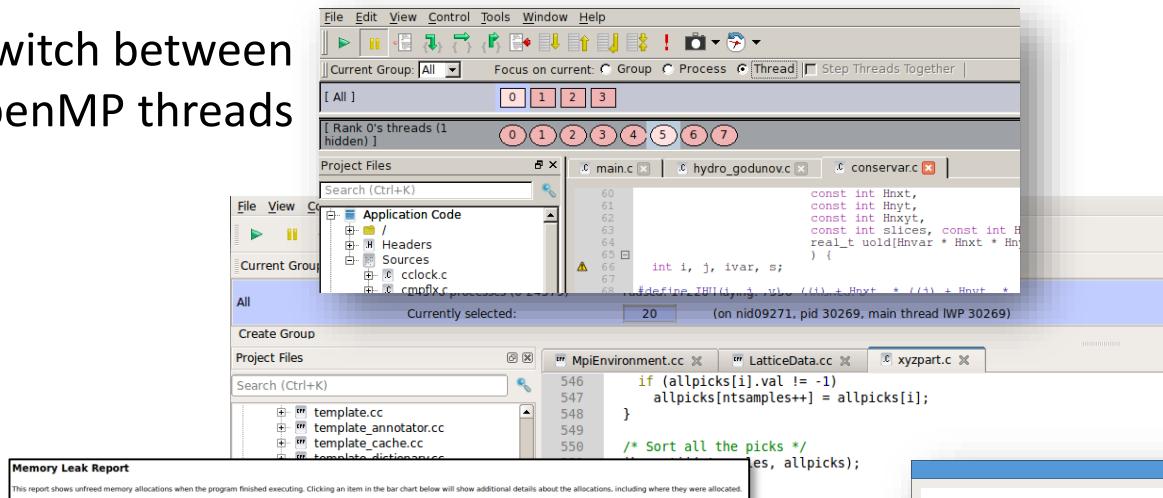


Improving the efficiency of your parallel software holds the key to solving more complex research problems faster. This pragmatic, 9 Step best practice guide will help you identify and focus on application readiness, bottlenecks and optimizations one step at a time.

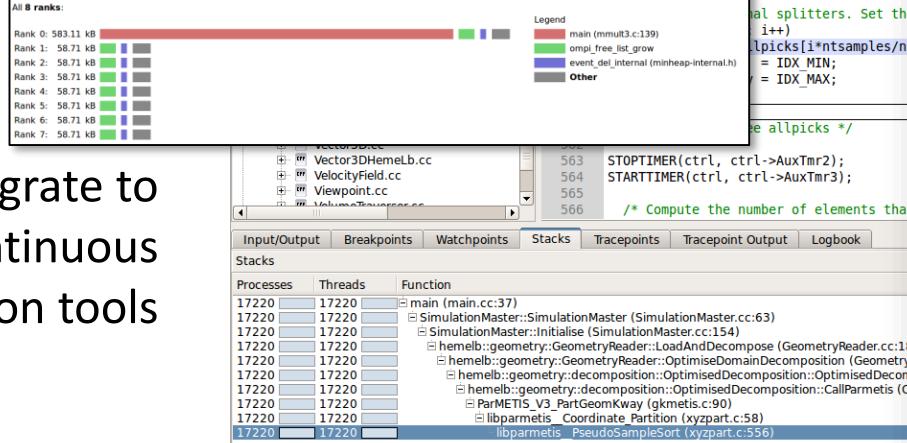


# Migrate and debug application

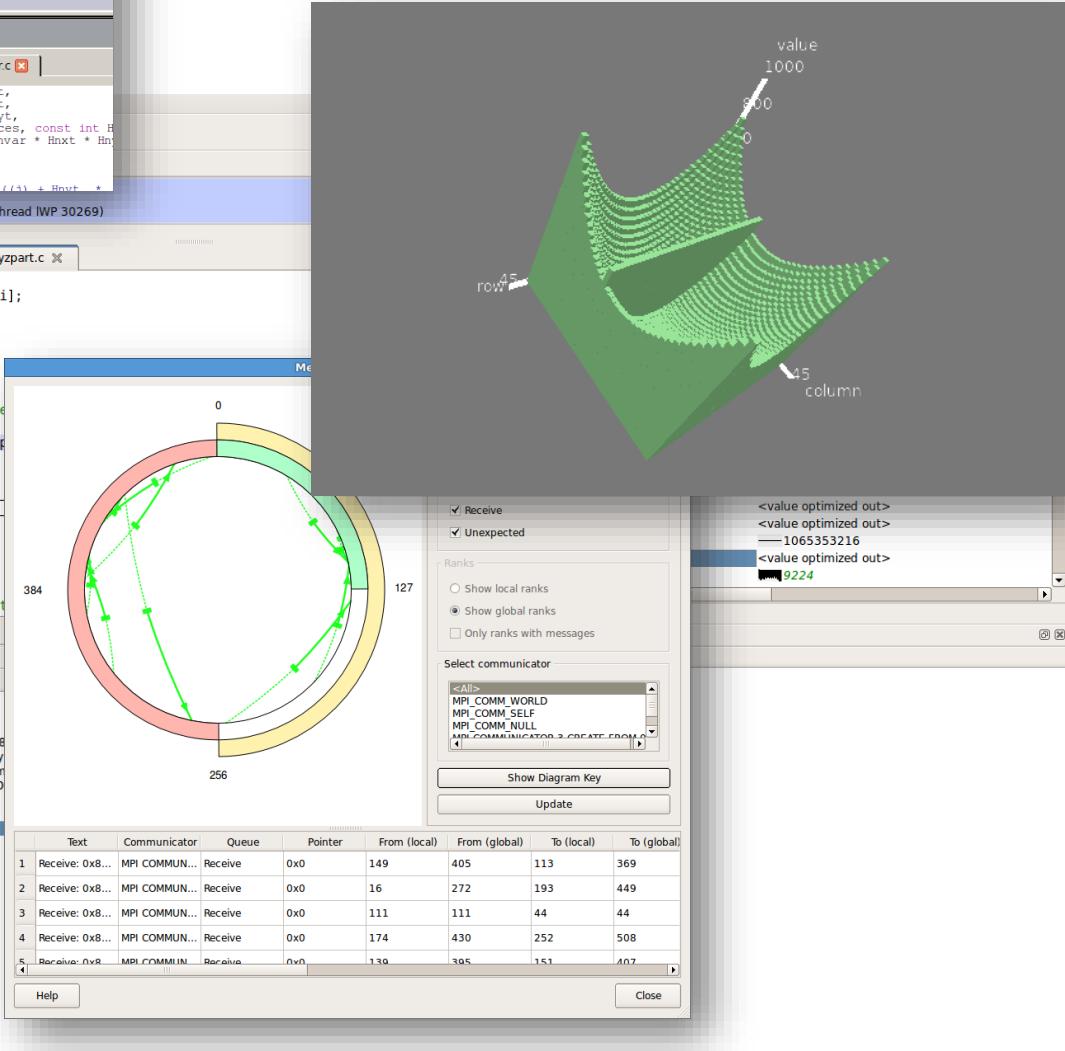
Switch between  
OpenMP threads



Integrate to  
continuous  
integration tools

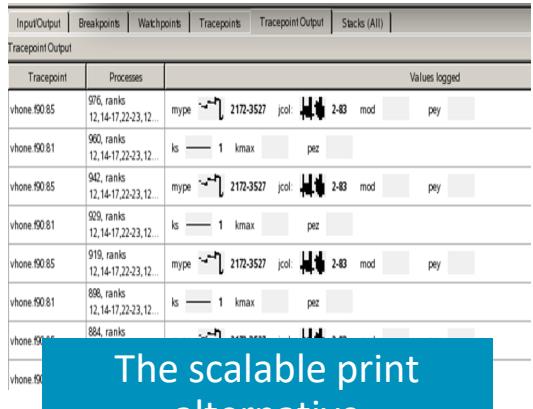


Display pending  
communications



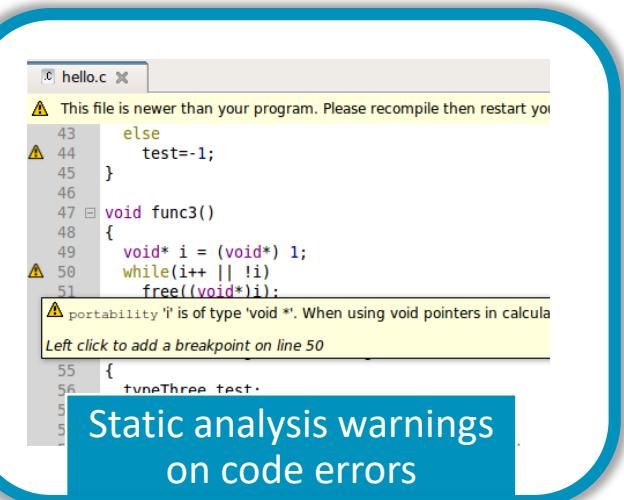
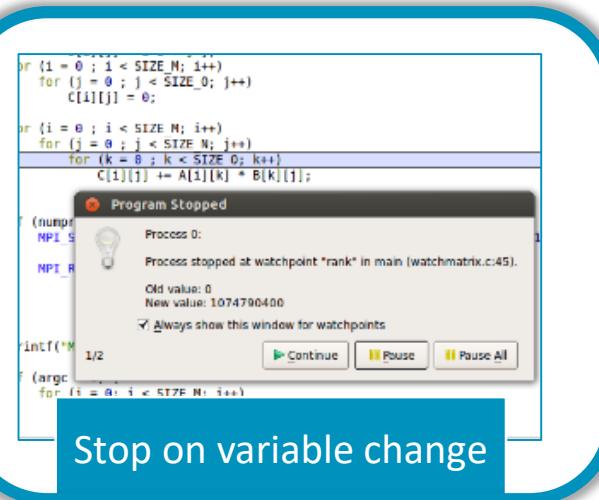
Visualise data  
structures

# Five great things to try with Arm DDT



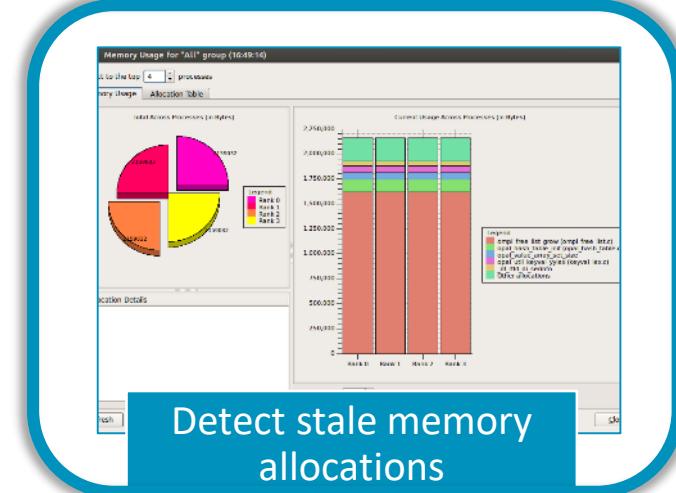
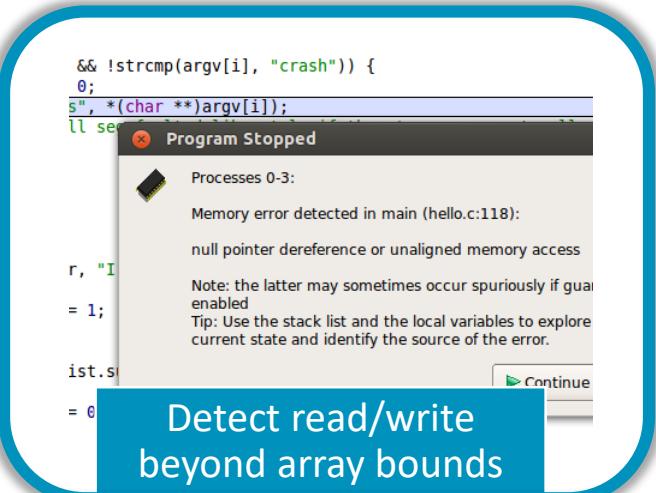
The scalable print alternative

This screenshot shows the Tracepoint Output tab in Arm DDT. It displays a list of log entries from various processes (rank 0, rank 1, rank 2, rank 3) with their corresponding memory addresses and values. The interface is designed to handle multiple parallel processes efficiently.



Static analysis warnings on code errors

This screenshot shows the code editor in Arm DDT with several static analysis warnings highlighted. The warnings are related to portability issues and undefined behavior. A tooltip provides instructions to recompile the program if it is newer than the current code.



# Arm DDT – The Debugger

Who had a rogue behavior ?

- Merges stacks from processes and threads

Where did it happen?

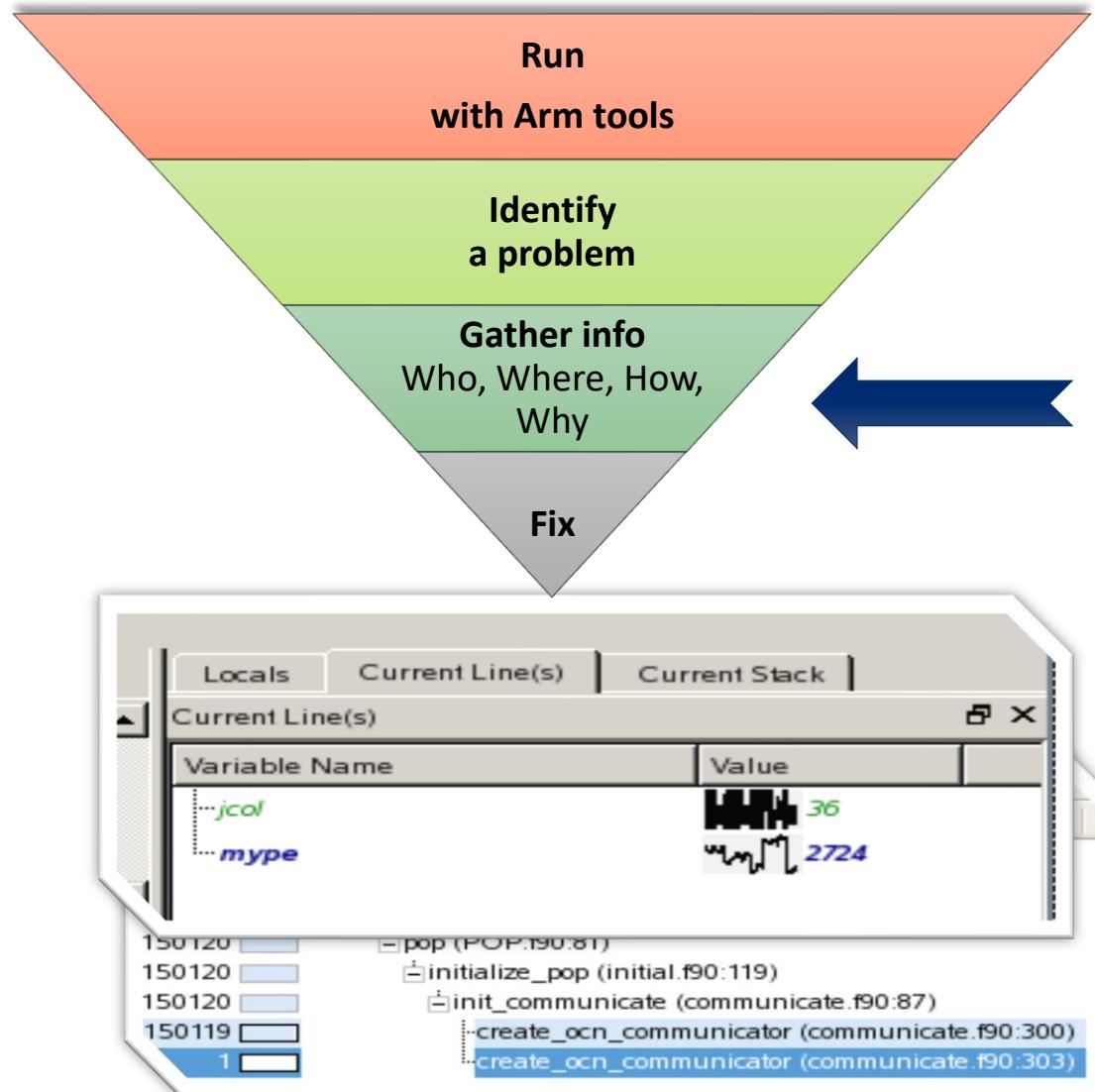
- leaps to source

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

# 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.

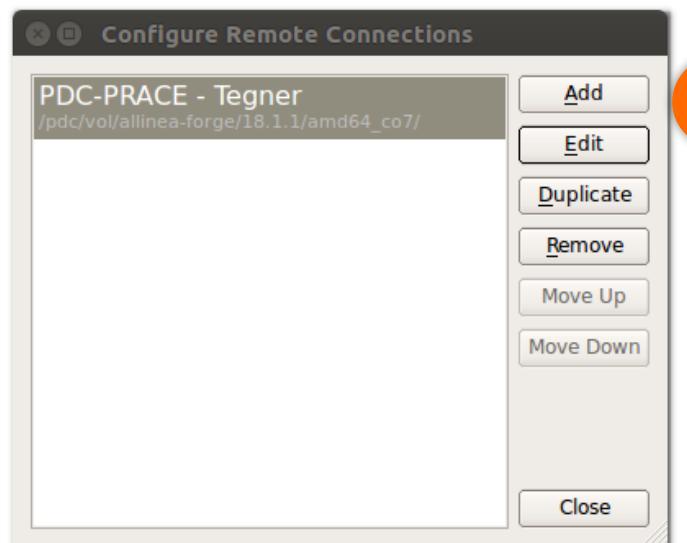
**MANUAL LAUNCH (ADVANCED)**  
Manually launch the backend yourself.

#### **OPTIONS**

Remote Launch:  
**Configure...**

**QUIT**

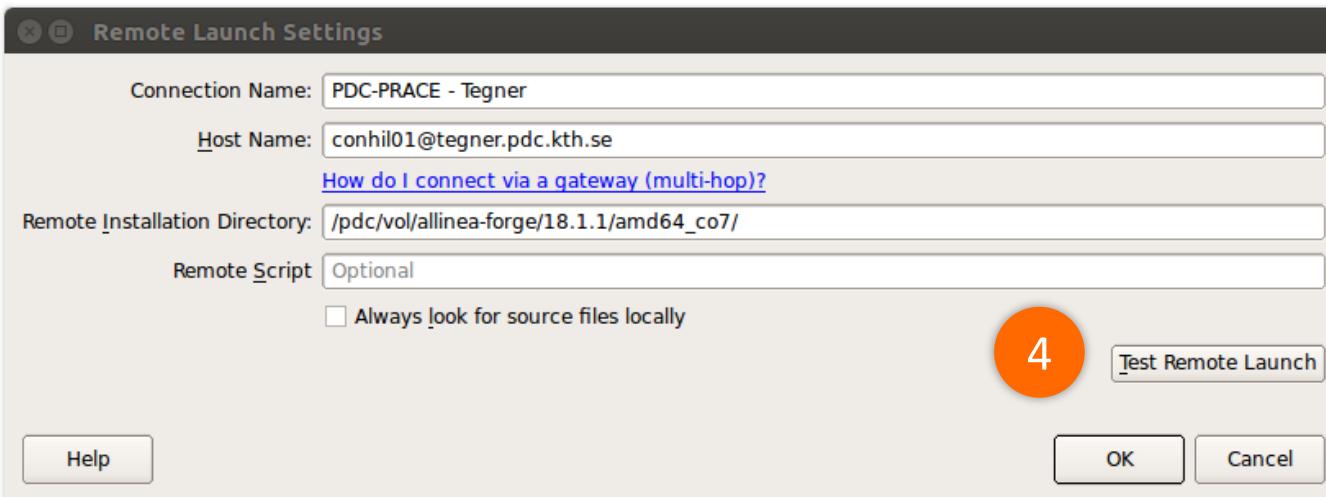
1



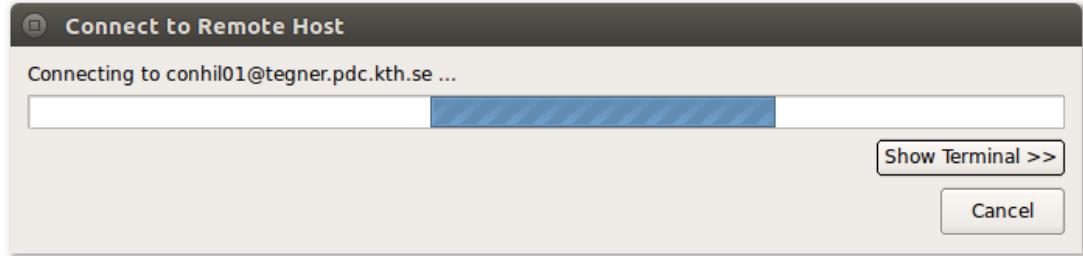
2



5



4



**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:

- Off
- Configure...

PDC-PRACE - Tegner

1

2  
Pop – Up  
Wait

**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:

PDC-PRACE - Tegner

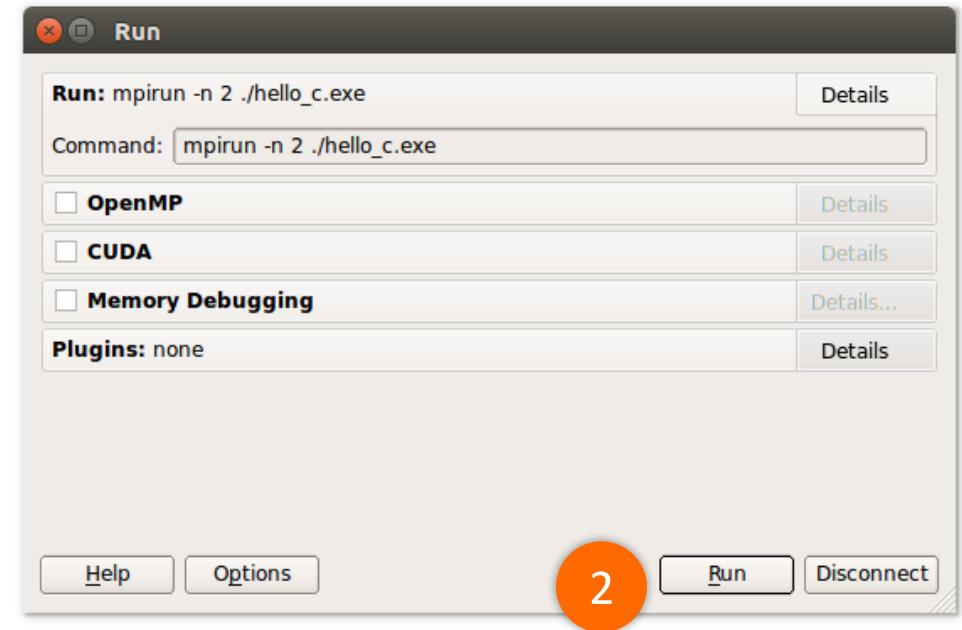
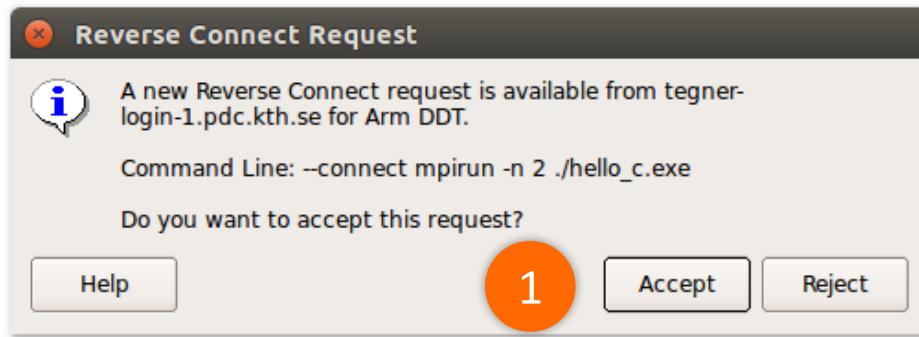
3

Reverse-Connect  
Client ready

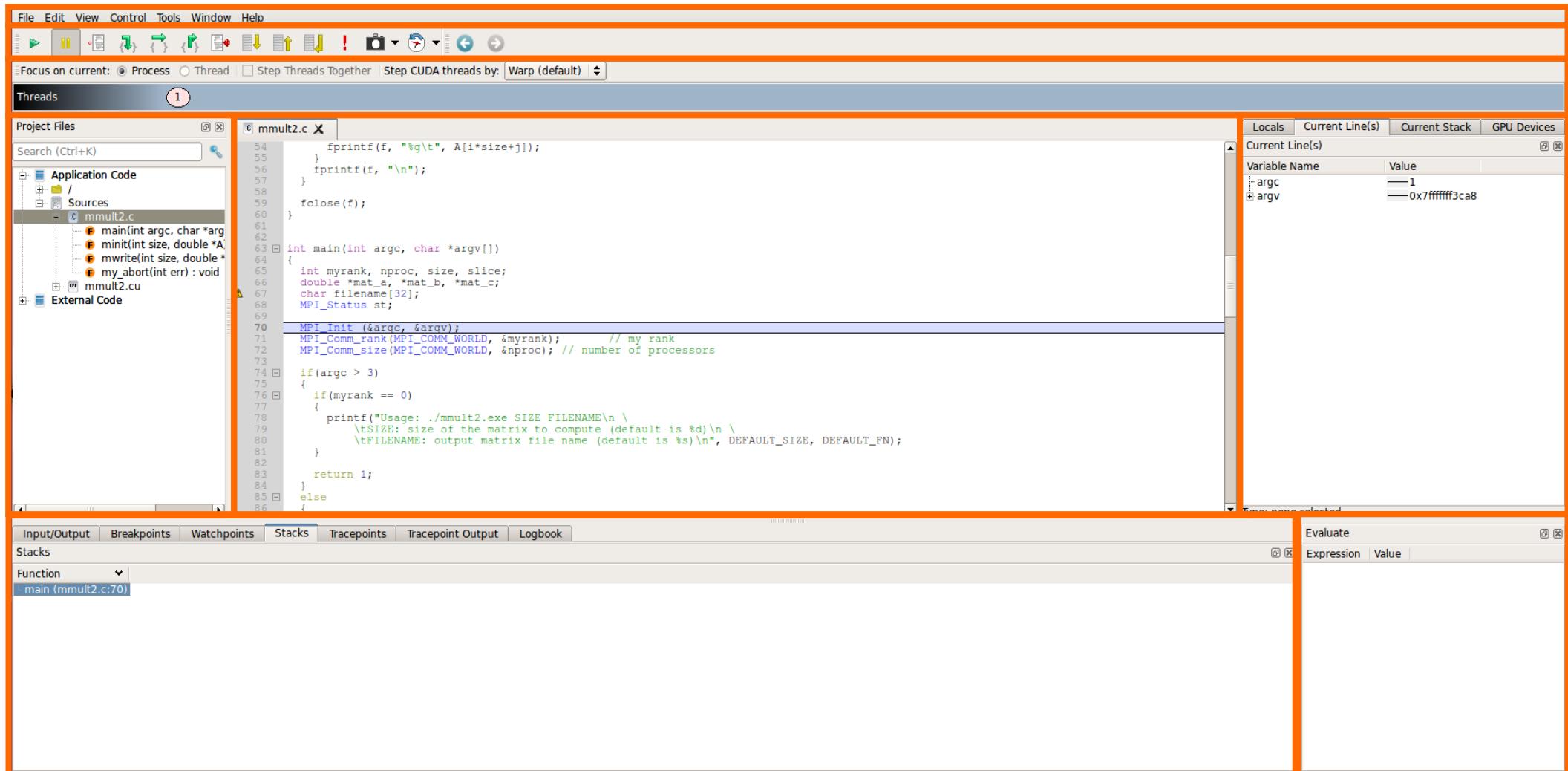
# Reverse-Connect – Server / Cluster side

```
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
dtt --connect mpirun -n 2 ./hello_c.exe
```

# Reverse-Connect – Client / Laptop side



# User Interface



# User Interface – Source code viewer

The screenshot shows a debugger interface with the following components:

- File Edit View Control Tools Window Help**: Standard menu bar.
- Focus on current: Process Thread Step Threads Together Step CUDA threads by: Warp (default)**: Focus settings.
- Threads**: Shows 1 thread.
- Project Files**: Shows Application Code, Sources, and External Code. mmult2.c is selected.
- Search (Ctrl+K)**: Search bar.
- mmult2.c**: Source code editor window with the following code:

```
54     fprintf(f, "%g\t", A[i*size+j]);
55 }
56 fprintf(f, "\n");
57 }
58 }
59 }
60 fclose(f);
61 }
62 }
63 int main(int argc, char *argv[])
64 {
65     int myrank, nproc, size, slice;
66     double *mat_a, *mat_b, *mat_c;
67     char filename[32];
68     MPI_Status st;
69
70     MPI_Init (&argc, &argv);
71     MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // my rank
72     MPI_Comm_size(MPI_COMM_WORLD, &nproc); // number of processors
73
74     if(argc > 3)
75     {
76         if(myrank == 0)
77         {
78             printf("Usage: ./mmult2.exe SIZE FILENAME\n \
79                   \tSIZE: size of the matrix to compute (default is %d)\n \
80                   \tFILENAME: output matrix file name (default is %s)\n", DEFAULT_SIZE, DEFAULT_FN);
81         }
82
83         return 1;
84     }
85     else
86     {
87 }
```
- Locals**: Shows variable values: argc = 1, argv = 0x7fffffff3ca8.
- Current Line(s)**: Shows the current line of code.
- Current Stack**: Shows the current stack.
- GPU Devices**: GPU device information.
- Input/Output Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output Logbook**: Toolbars at the bottom.
- Stacks**: Function stack: main (mmult2.c:70).
- Evaluate**: Evaluate expression tool.

# User Interface – Play/ Pause / Step

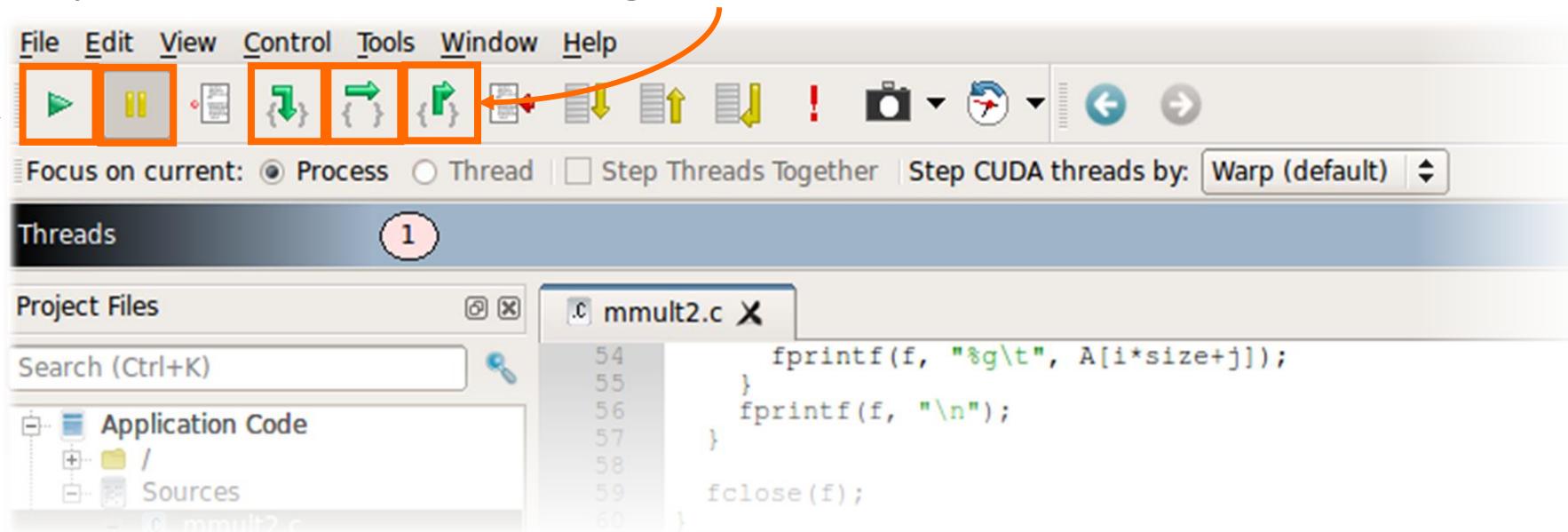
Play : Run everything. Use typically at the beginning or after Pause

Pause : Stops running current kernel

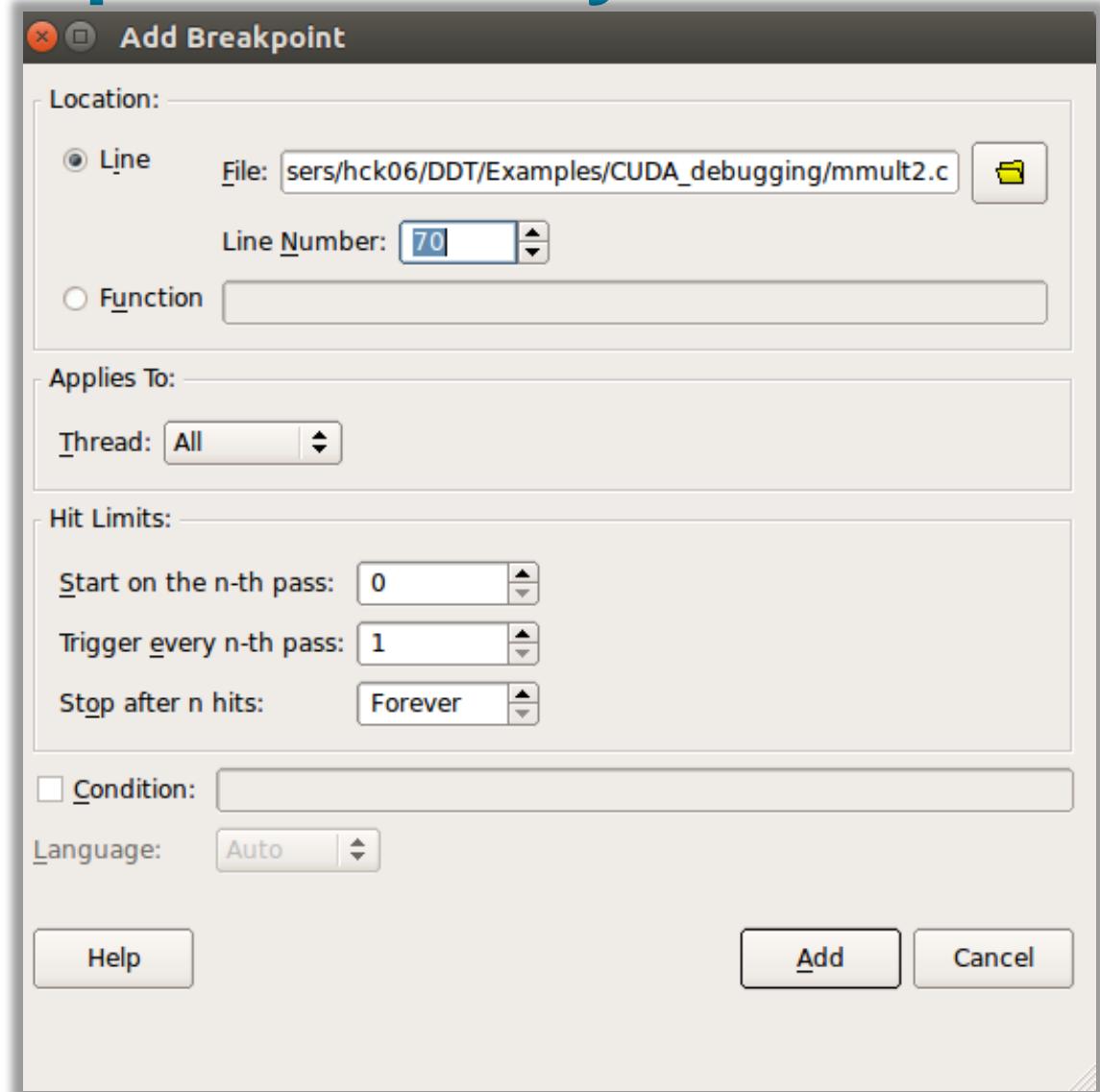
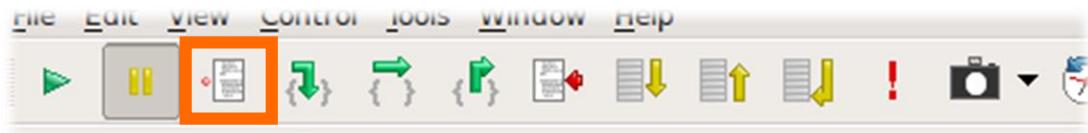
Step In : Enter a function call and display source code of the function

Step Over : Execute current line of code

Step Out : Comes back one stage above current stack

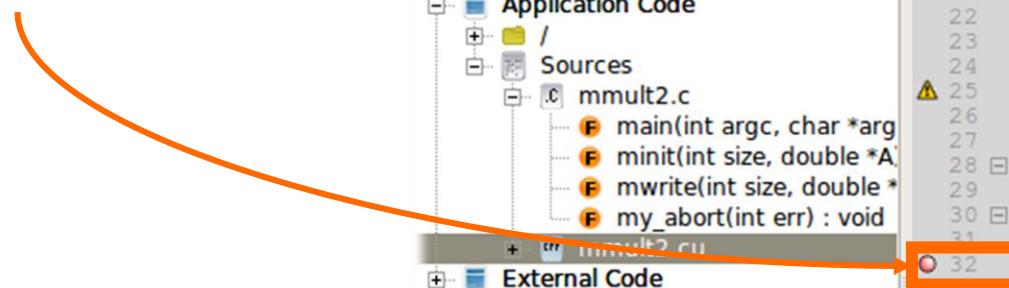


# User Interface – Add Breakpoints – Way 1



# 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



The screenshot shows the Nsight Compute IDE interface. The top menu bar includes File, Edit, View, Control, Tools, Window, Help, and various toolbar icons. The status bar at the bottom indicates 'Focus on current: Process'. The main window has tabs for 'Threads' (with a count of 1) and two code files: 'mmult2.c' (selected) and 'mmult2.cu'. The 'mmult2.c' tab shows the following CUDA code:

```
// CUDA version of mmult: size, pitch_A_nbelem, pitch_B_nbelem, pitch_C_nbelem
__global__ void mmult_kernel(int size, int nslices, double *A, size_t pitch_A, size_t pitch_B, size_t pitch_C)
{
    // Set thread indexes
    int i = blockDim.y*blockIdx.y+threadIdx.y;
    int j = blockDim.x*blockIdx.x+threadIdx.x;
    double res = 0.0;

    // Make sure we don't write any value out of C because the grid can be very large
    if(i < size/nslices & j < size)
    {
        for(int k=0; k<size; k++)
        {
            res += A[i*pitch_A_nbelem+k] * B[k*pitch_B_nbelem+j];
        }

        C[i*pitch_C_nbelem+j] += res;
    }
}

void mmult_cuda(int size, int nslices, double *A, double *B, double *C,
                size_t pitch_A, size_t pitch_B, size_t pitch_C,
                int widthinbytes = size*sizeof(double));
```

# Reverse-Connect – Client / Laptop Side

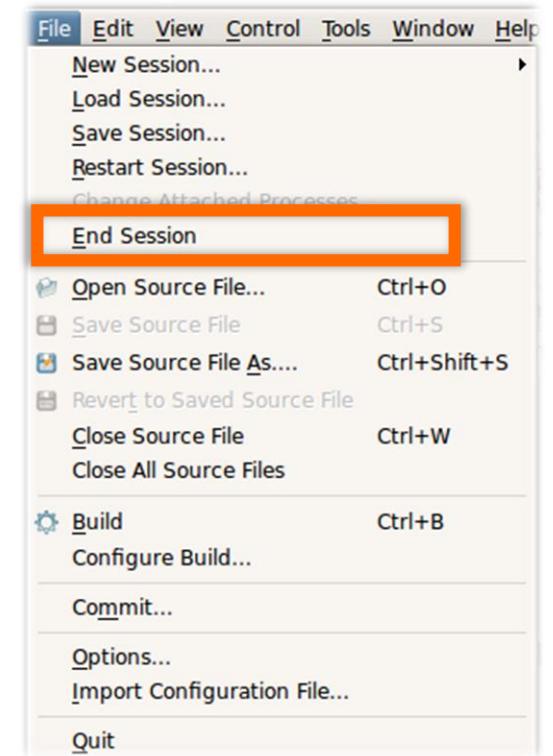
The screenshot shows the Arm DDT (Debugging and Tracing Tool) interface. On the left, the project tree shows a single file 'hello.c'. The code window contains the following MPI code:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

A message box in the center says "All processes finished." and asks "Every process in your program has terminated - would you like to restart this session from the beginning?". The log window at the bottom shows:

```
Hello world from processor t0in45.pdc.kth.se, rank 0 out of 2 processors
Hello world from processor t0in45.pdc.kth.se, rank 1 out of 2 processors
```

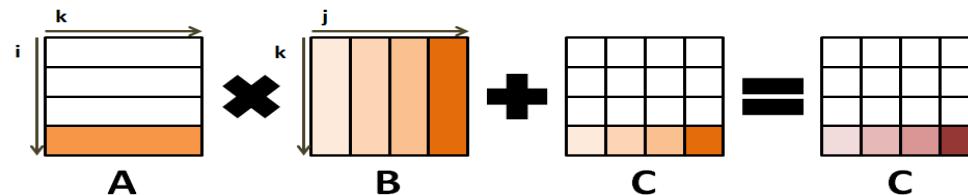
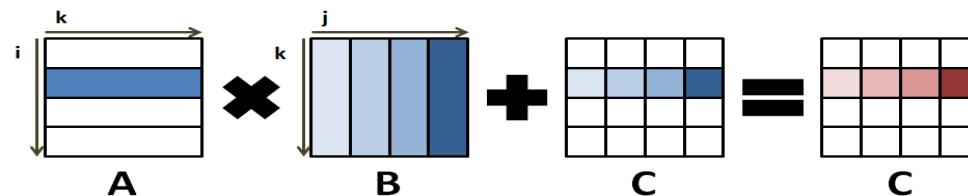
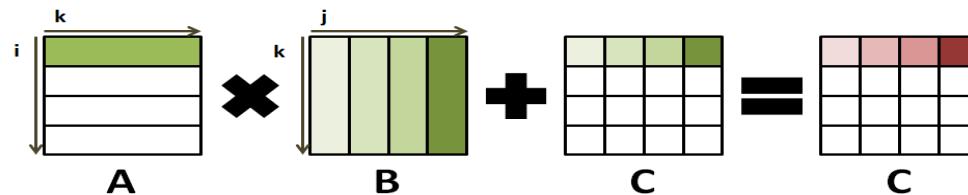
The status bar at the bottom right says "Ready. Connected to: (via tunnel) tennerlogin-1.pdc.kth.se:4201 -> tennerlogin-1.pdc.kth.se".



# 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

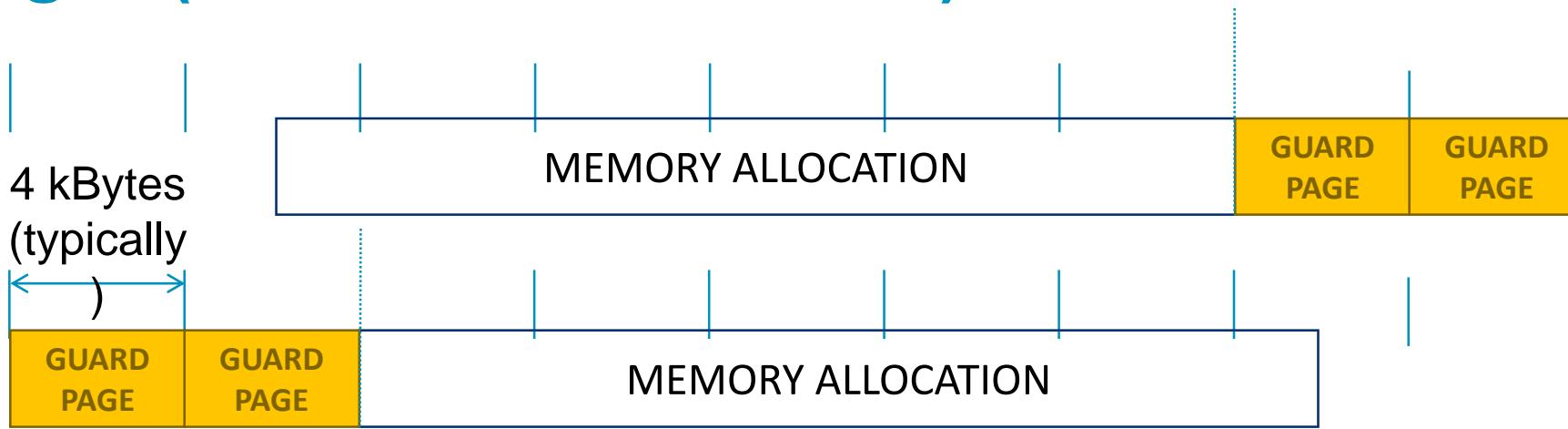
- 1\_interactive\_debugging
- Compile the program
- Run one of the binaries. What do you see ?
- Let's debug it then !
- Recompile with DEBUG=1, launch DDT and ... debug !
- Can you find where the problem comes from ?
- Modify the code and recompile (in DDT)
- Relaunch the program.

# Hands – On : Memory Debugging

# Heap debugging options available



# 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 -lmallocthcxx

# Memory debugging

**RUN**  
Run and debug a program.

**ATTACH**  
Attach to an already running program.

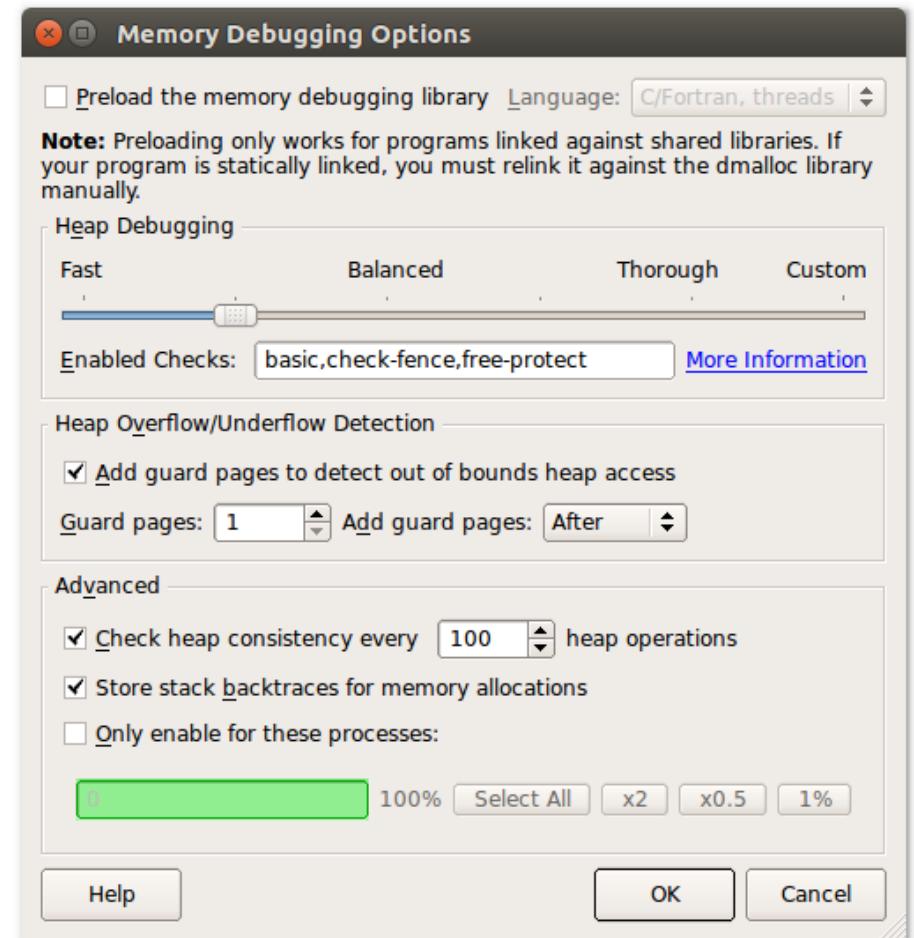
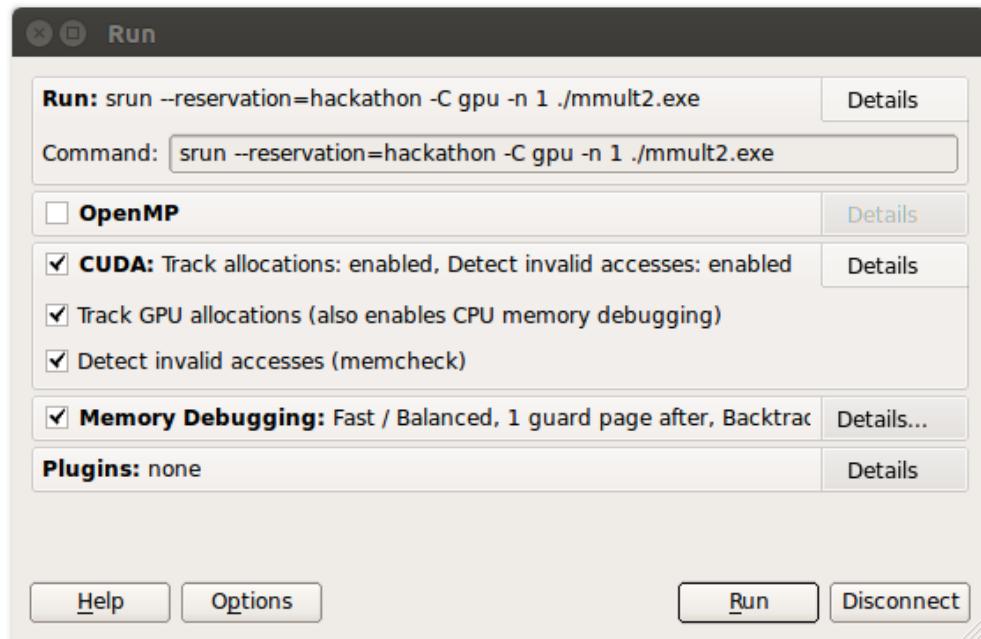
**OPEN CORE**  
Open a core file from a previous run.

**MANUAL LAUNCH (ADVANCED)**  
Manually launch the backend yourself.

**OPTIONS**

Remote Launch:  
(via tunnel) daint105:4201 -> daint1

**QUIT**



# 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.

# Hands – On : Memory debugging

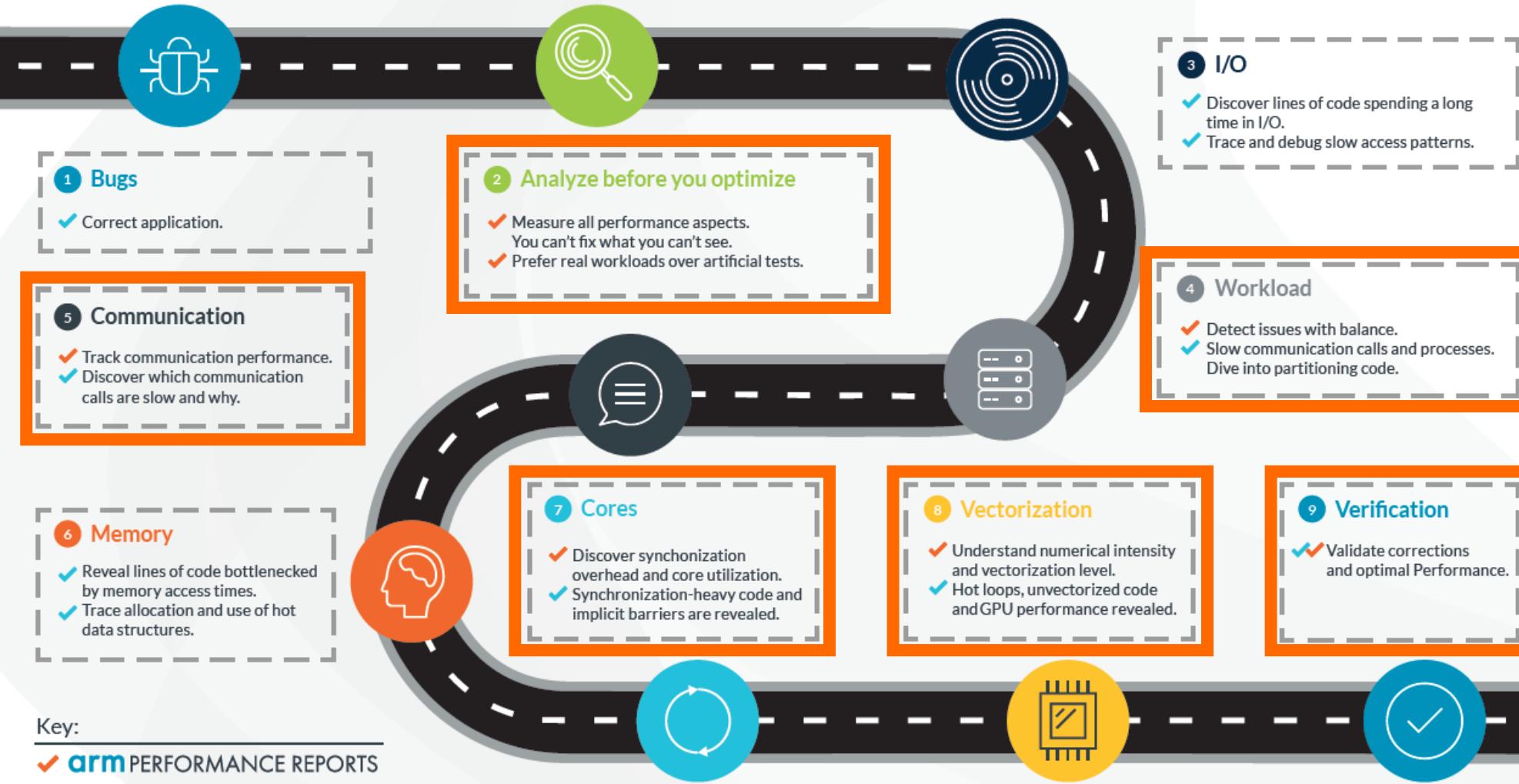
- Are you sure we are done with hidden issues ?
- Use DDT offline report with “--offline --mem-debug” flags
- Have a look to the report, anything suspicious ?
- Do you see how to fix this ?

# Arm Performance Reports

# 9 Step guide: optimizing high performance applications



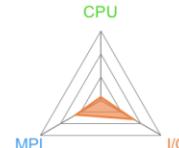
Improving the efficiency of your parallel software holds the key to solving more complex research problems faster. This pragmatic, 9 Step best practice guide will help you identify and focus on application readiness, bottlenecks and optimizations one step at a time.



# “Learn” with Arm Performance Reports

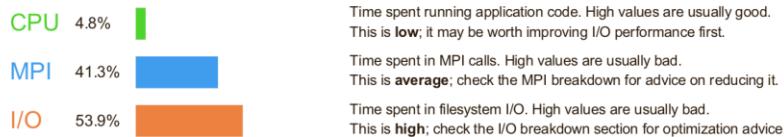


Executable: MADbench2  
Resources: 16 processes, 1 node  
Machine: sandybridge2  
Start time: Mon Nov 4 12:27:50 2013  
Total time: 109 seconds (2 minutes)  
Full path: /tmp/MADbench2  
Notes: 12-core server / HDD / 16 readers + writers



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

The total wallclock time was spent as follows:



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

## CPU

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



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

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

## I/O

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



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

## MPI

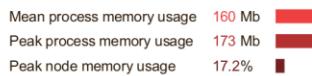
Of the **41.3%** total time spent in MPI calls:



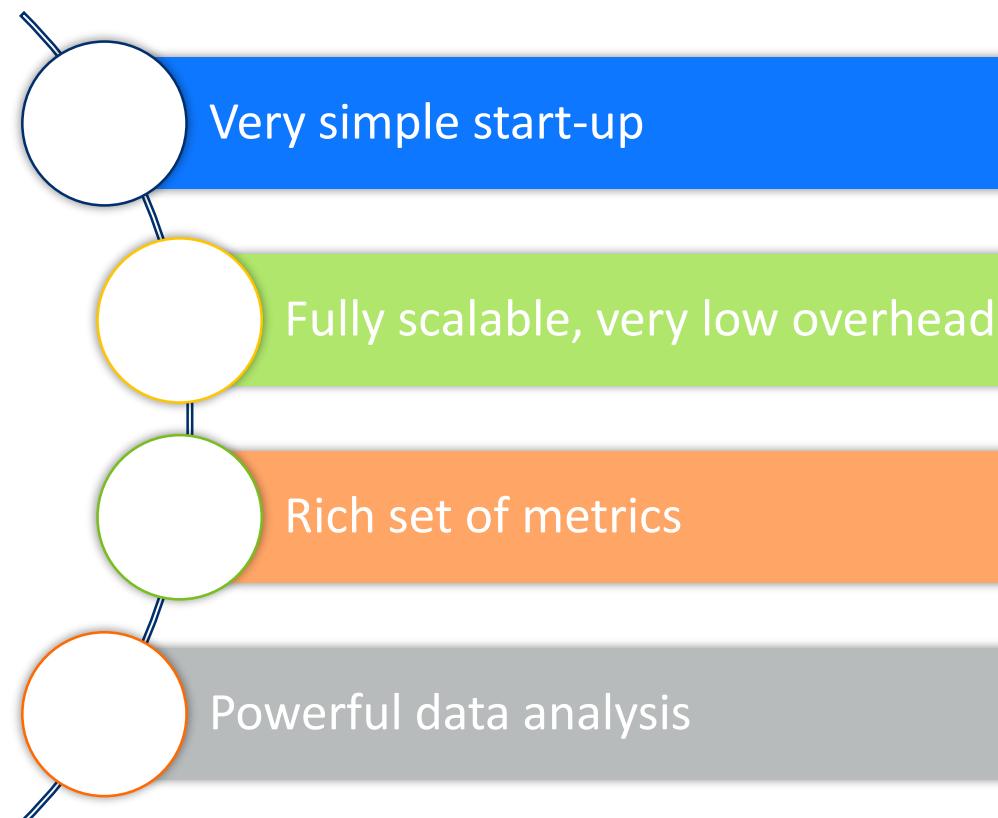
All of the time is spent in **collective calls** with a very low **transfer rate**. This suggests a significant load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

## Memory

Per-process memory usage may also affect scaling:

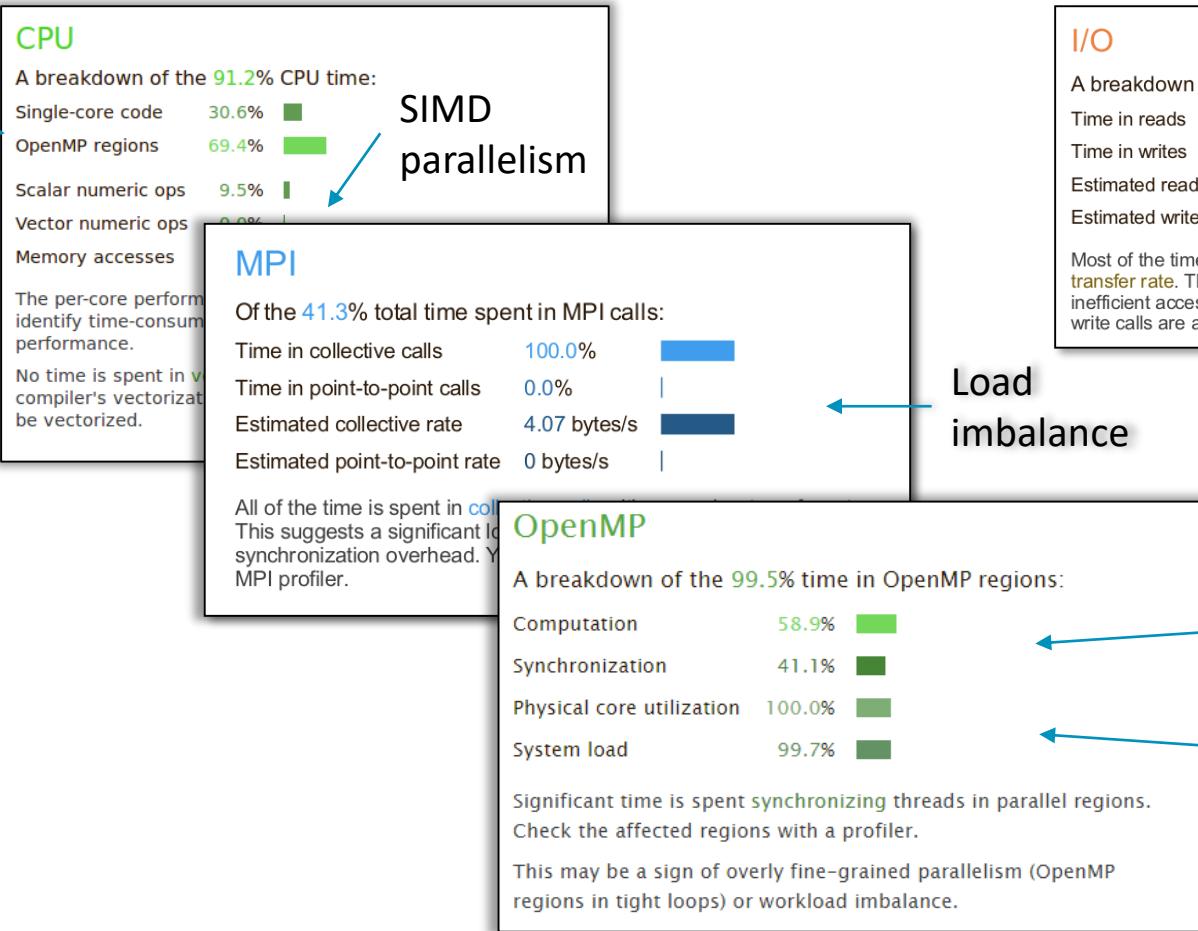


The **peak node memory usage** is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.



# Metrics Overview

Multi-threaded parallelism



I/O

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

Time in reads  
Time in writes  
Estimated read rate  
Estimated write rate

Most of the time is spent in reads. This may indicate a slow transfer rate. This metric is also affected by inefficient access patterns. Write calls are affected by the number of processes and more.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage: 160 Mb

Peak process memory usage  
Peak node memory

The peak node memory usage is the total number of processes and more.

Lustre

Lustre file operations (per node)

Mean write rate

Peak write rate

Mean file operations

Mean metadata operations

Energy

A breakdown of how the 32.3 Wh was used:

CPU: 61.9%

System: 38.1%

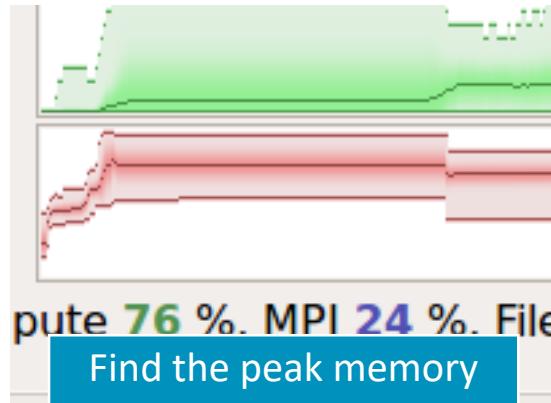
Mean node power: 94.1 W

Peak node power: 98.0 W

Significant time is spent waiting for memory accesses. Reducing the CPU clock frequency could reduce overall energy usage.

# Arm MAP

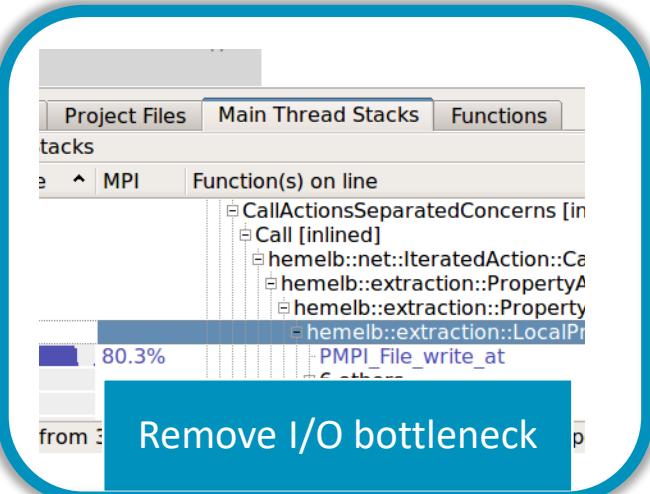
# Six Great Things to Try with Arm MAP



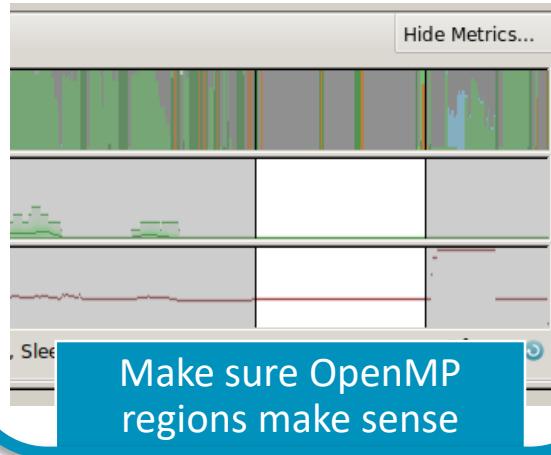
Find the peak memory use

A screenshot of a code editor showing MPI C code. The code includes MPI\_SEND, MPI\_RECV, and MPI\_BARRIER calls. A call tip for MPI\_BARRIER is visible.

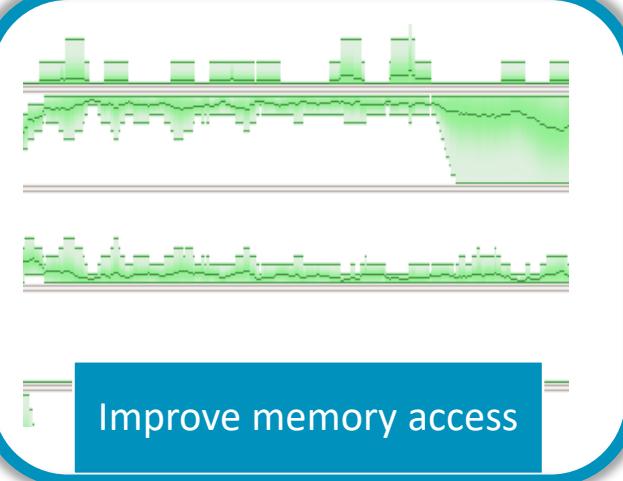
Fix an MPI imbalance



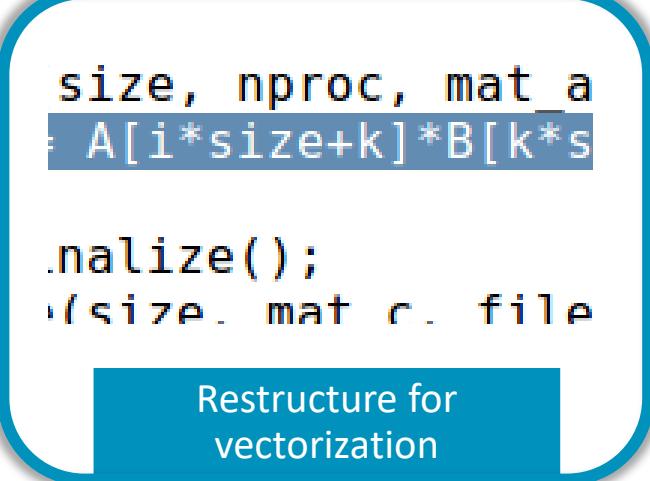
Remove I/O bottleneck



Make sure OpenMP regions make sense

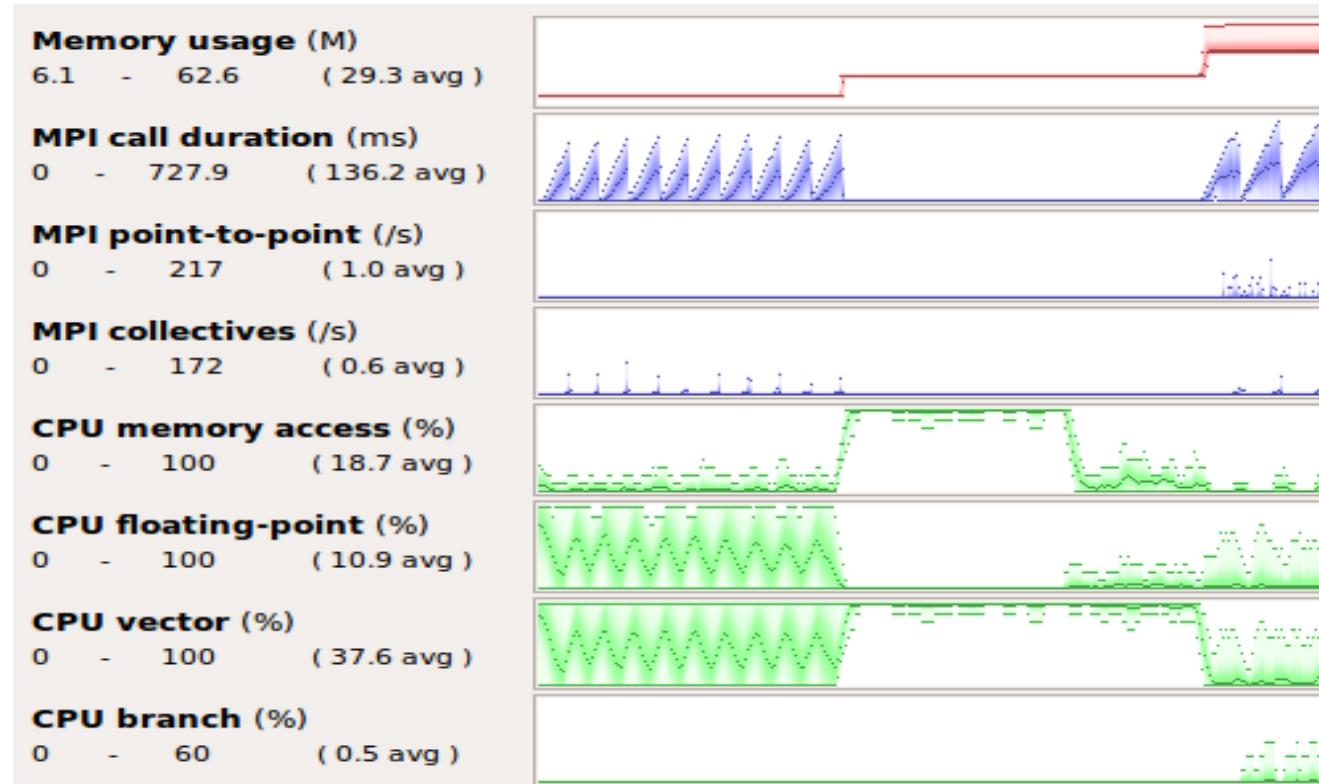


Improve memory access



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



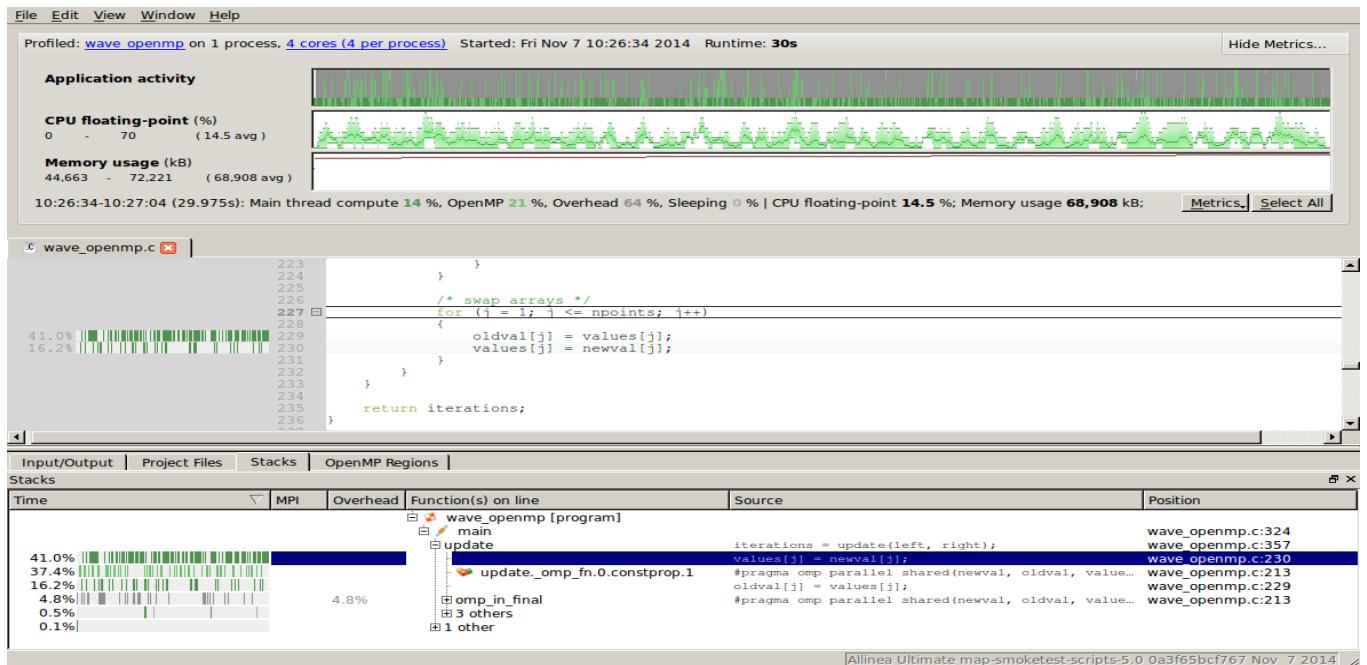
<5% slowdown



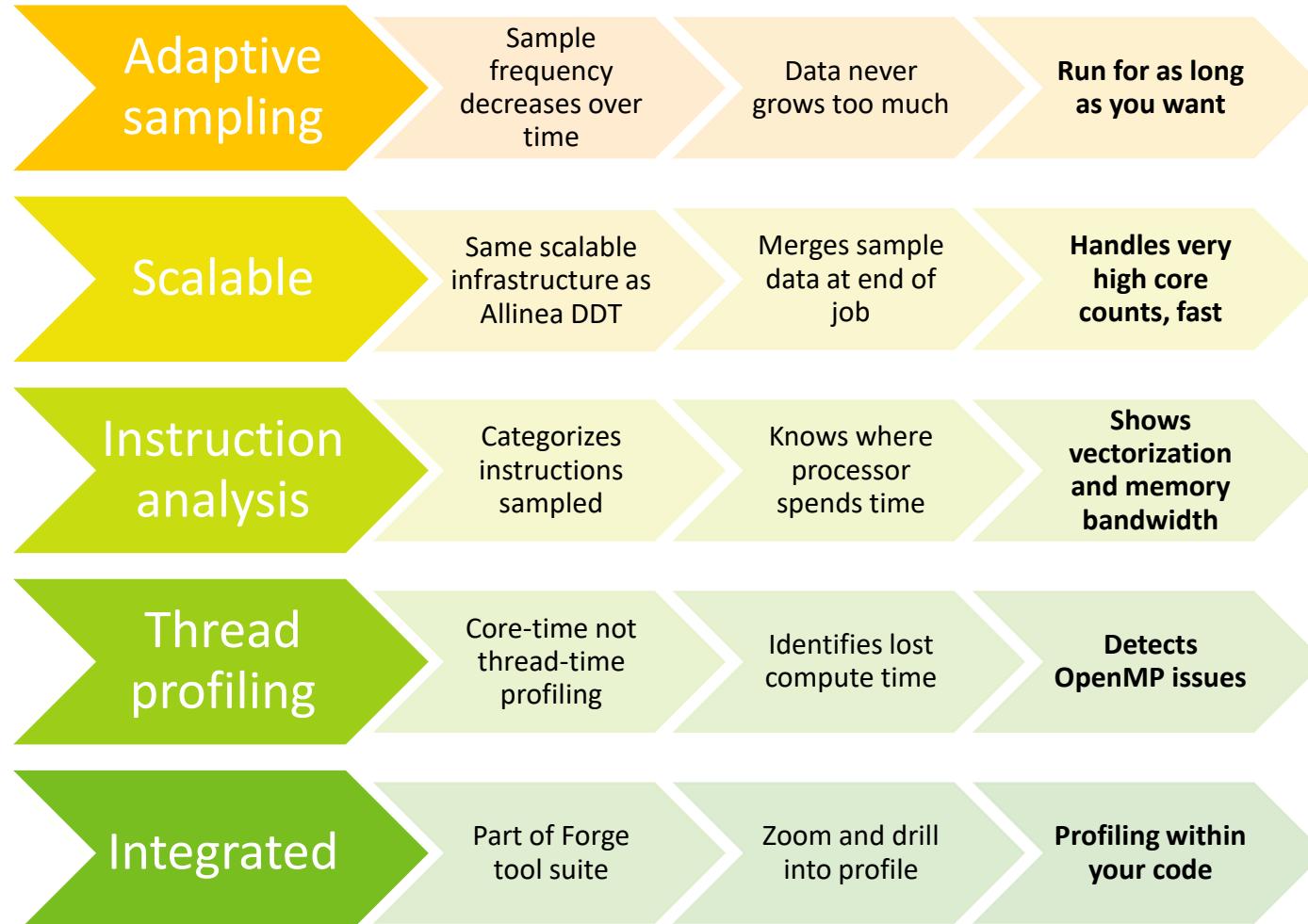
No instrumentation



No recompilation



# How Arm MAP is different



# Preparing Code for Use with MAP

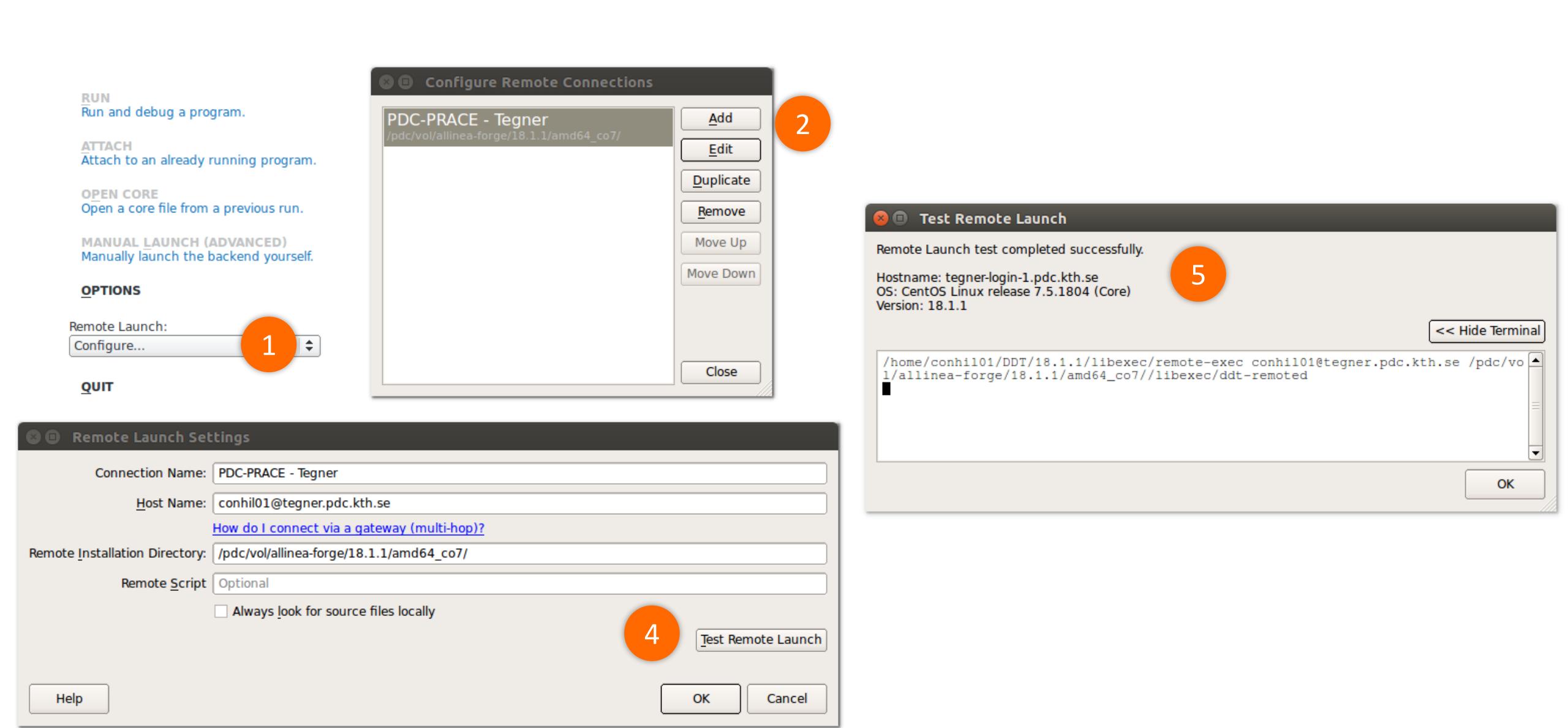
To see the source code, the application should be compiled with the debug flag typically `-g`

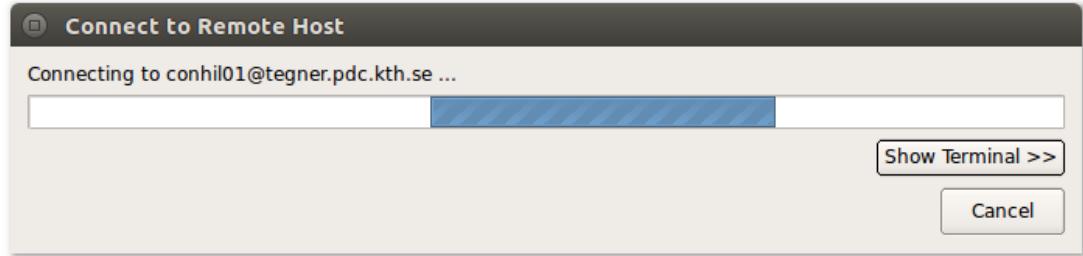
It is recommended to *always* keep optimization flags on when profiling

# Hands – On : Launch MAP

# 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:

- Off
- Configure...

PDC-PRACE - Tegner

1

2  
Pop – Up  
Wait

**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:

PDC-PRACE - Tegner

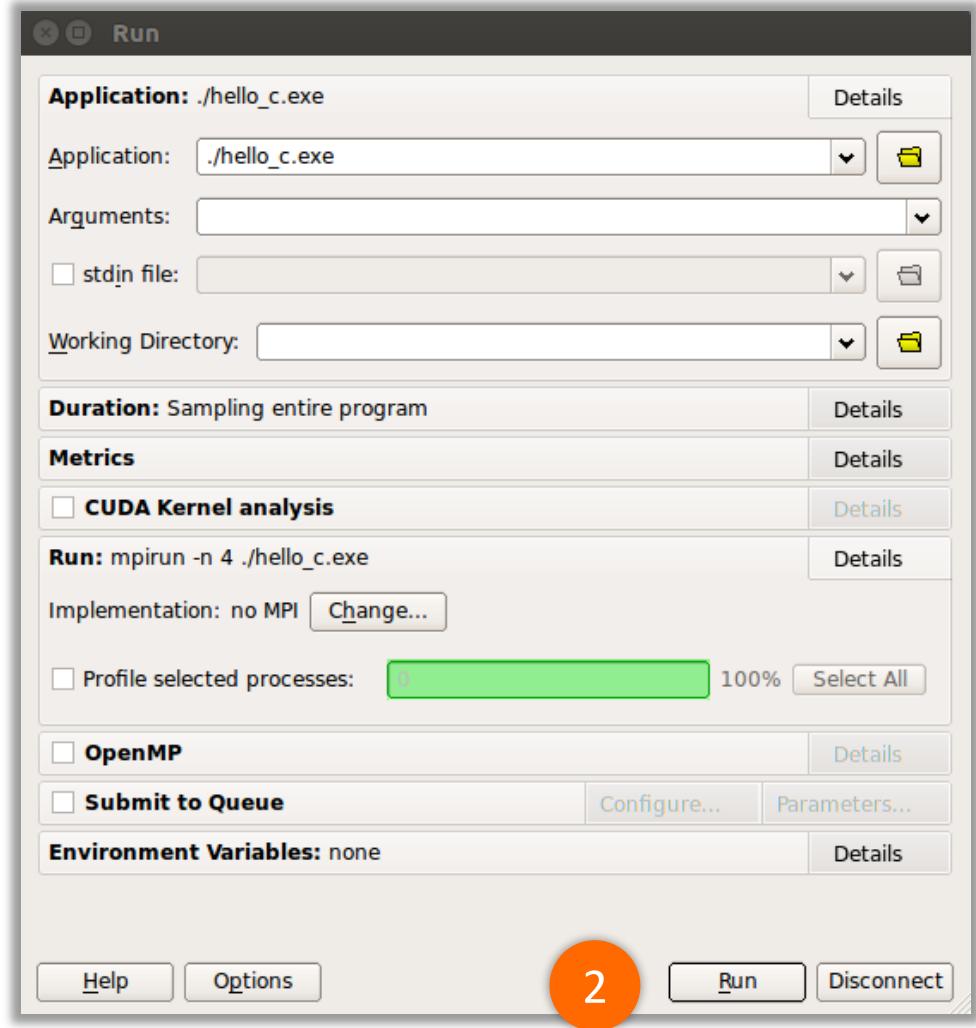
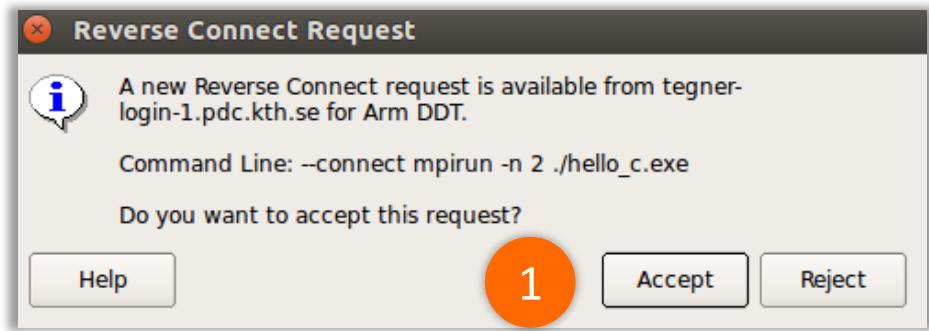
3

Reverse-Connect  
Client ready

# 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



# Hands – On : Launch Perf-Reports

# 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

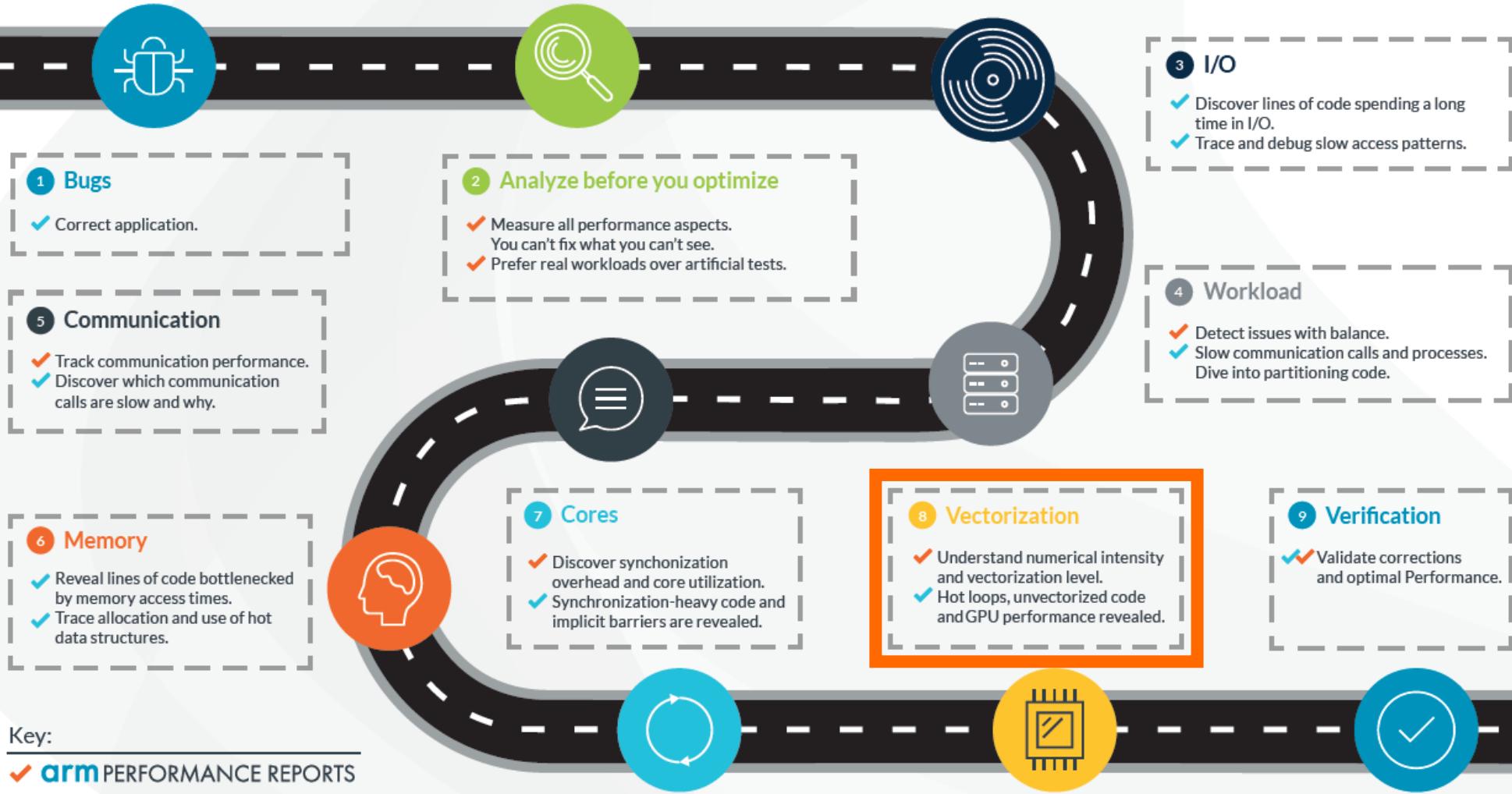
- Two files outputted : .txt and .html
- .txt can be visualized on the cluster with file editor
- Use scp to copy the .html file back to your laptop
- Open it with a Web Browser

# Hands – On : Vectorization

# 9 Step guide: optimizing high performance applications



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.



# 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
```

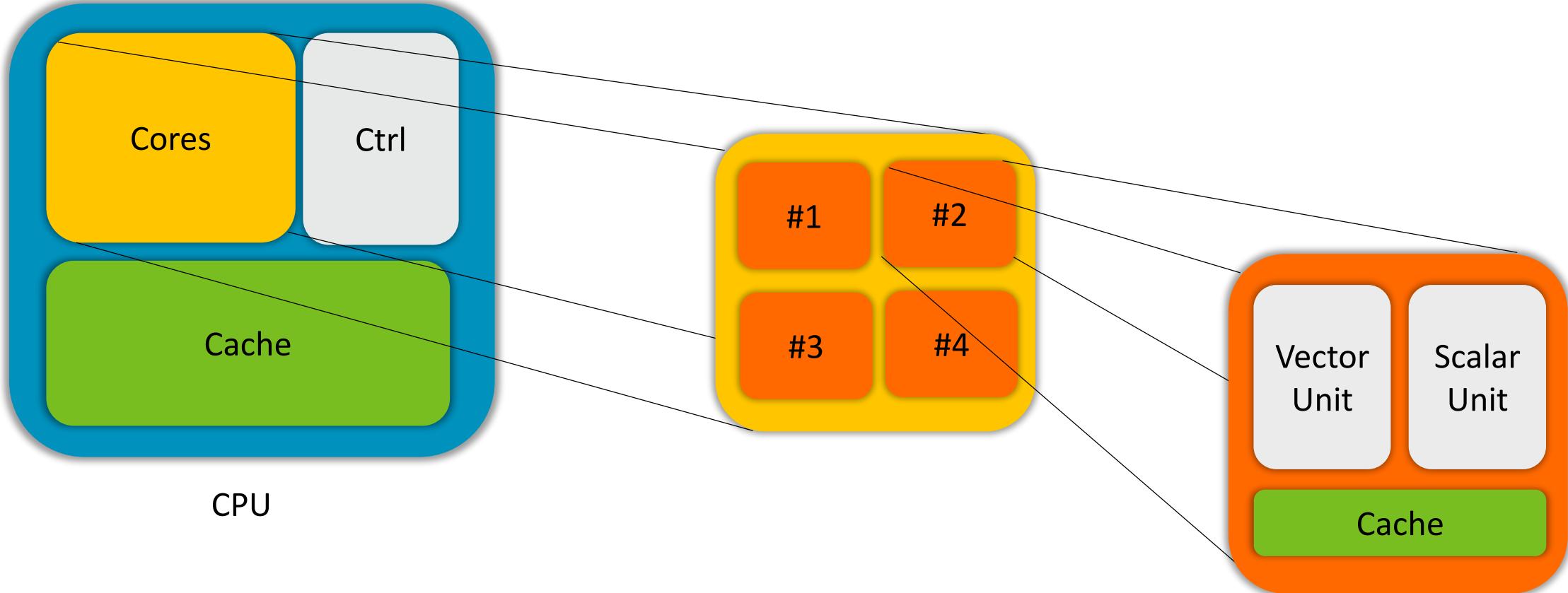
```
ENDDO
```

...

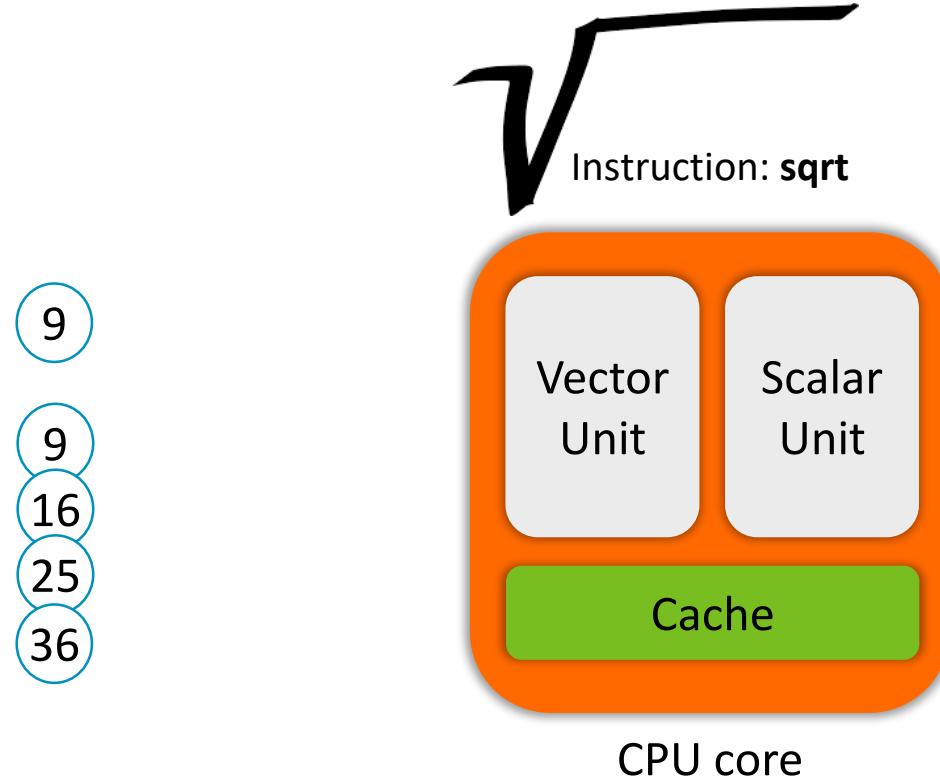
Example with modified version of CloverLeaf

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

# Vector Units



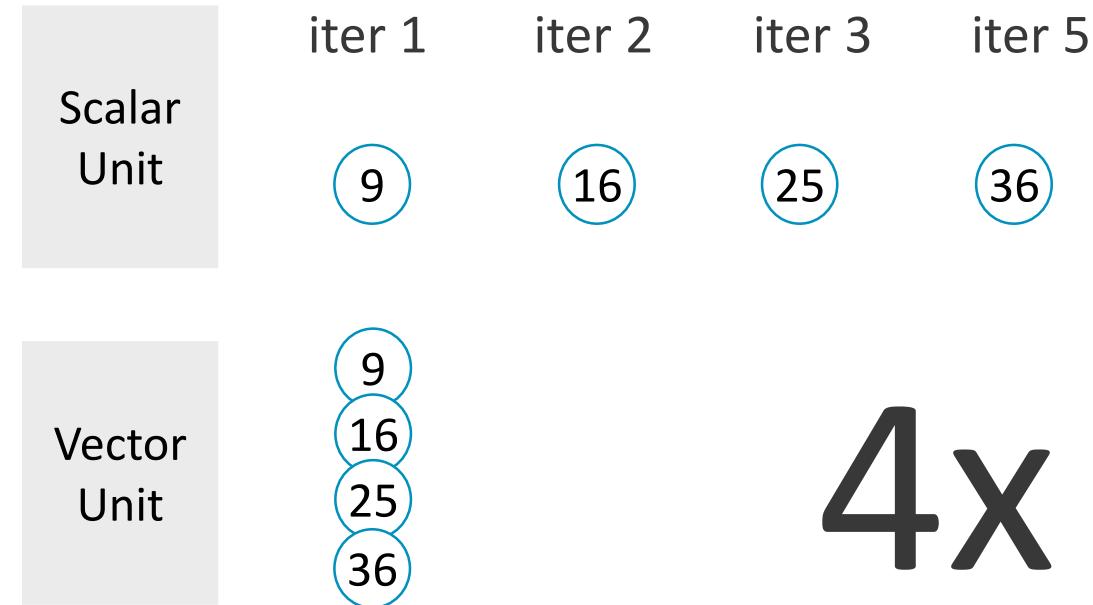
# Vectorization / SIMD



# Vectorization / SIMD



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

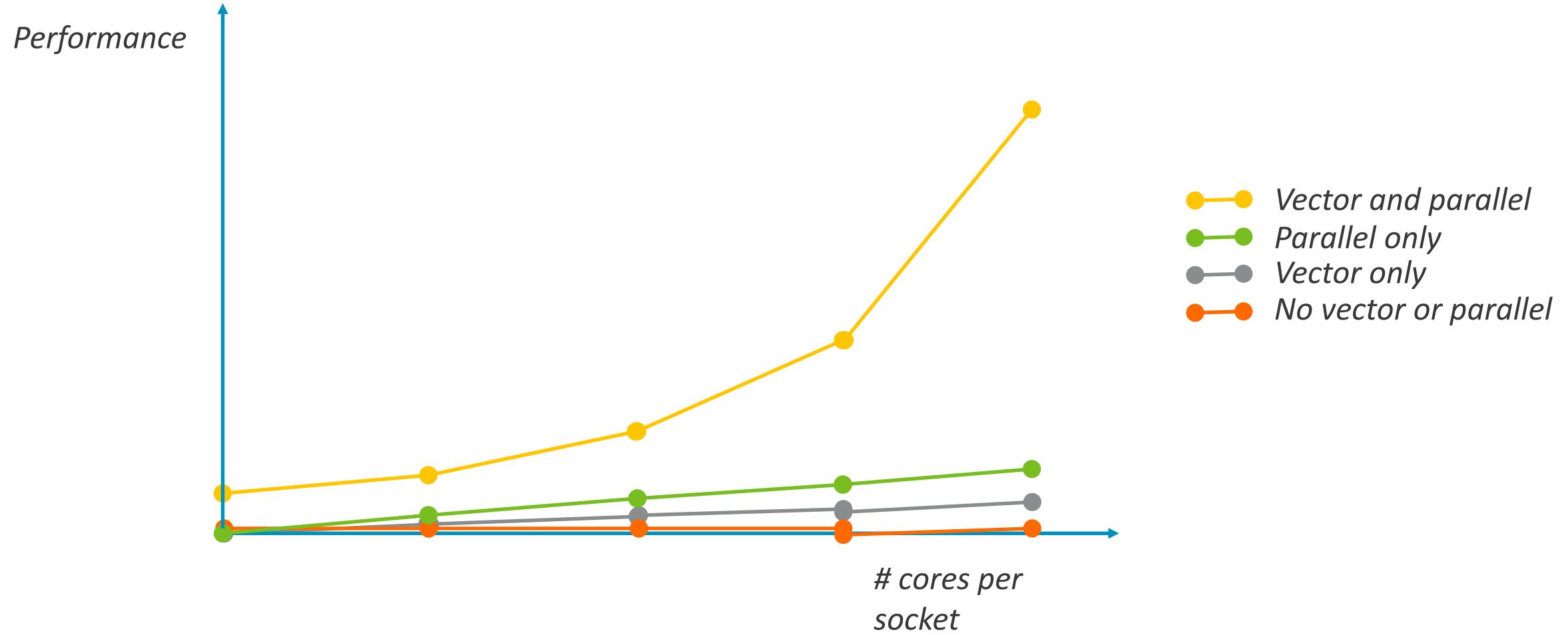


Intel® AVX2: 256-bit vector unit → 8 SP / 4 DP

Intel® AVX-512: 512-bit vector unit → 16 SP / 8 DP

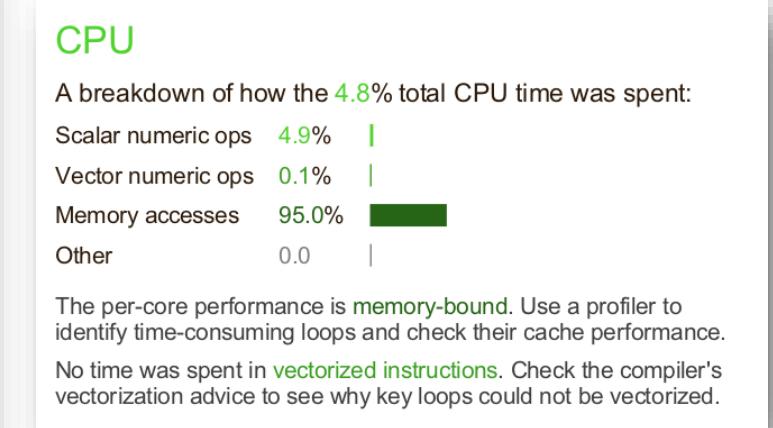
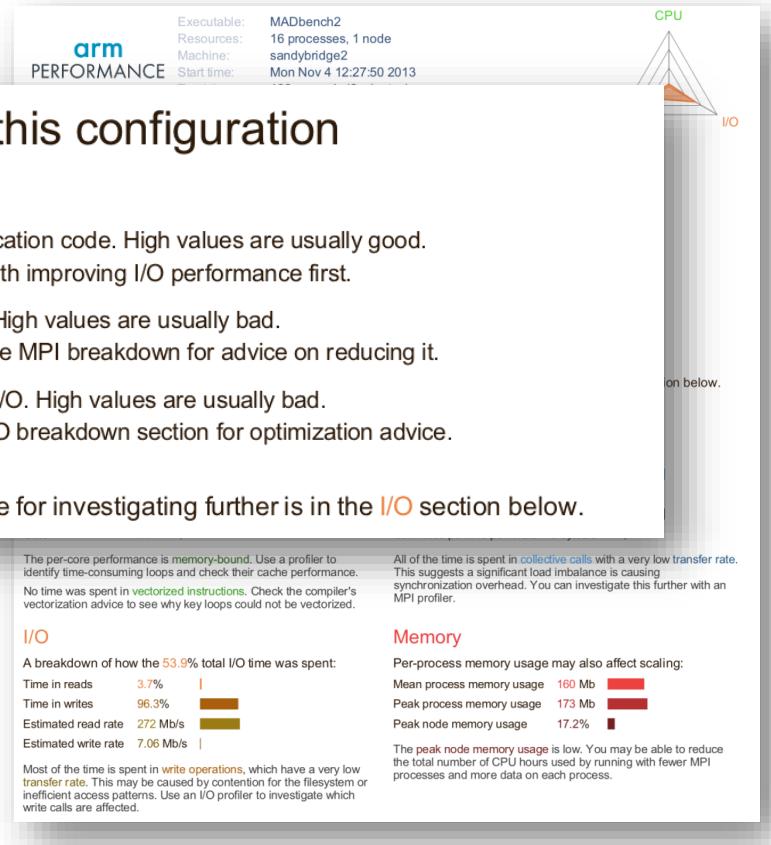
Arm® NEON: 128-bit vector unit → 4 SP / 2 DP

# Why? Performance lies in the software



# Identifying the amount of vectorized code

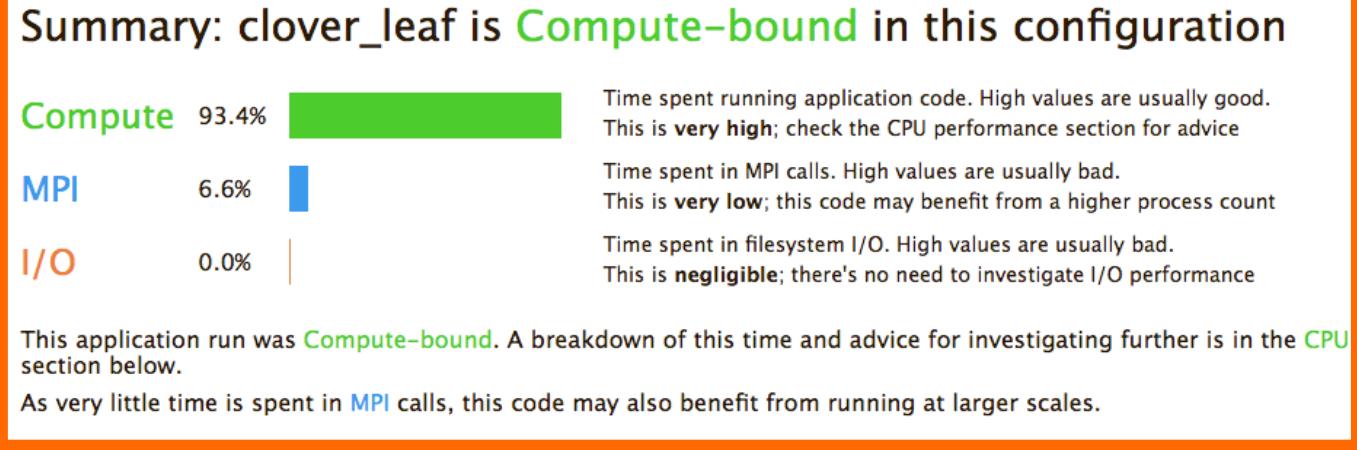
- Arm Performance Reports is an application reporting tool for HPC
  - Easy to use: no re-compiling required
  - Gives a comprehensible and readable summary of the application behavior



# 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%



## 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:

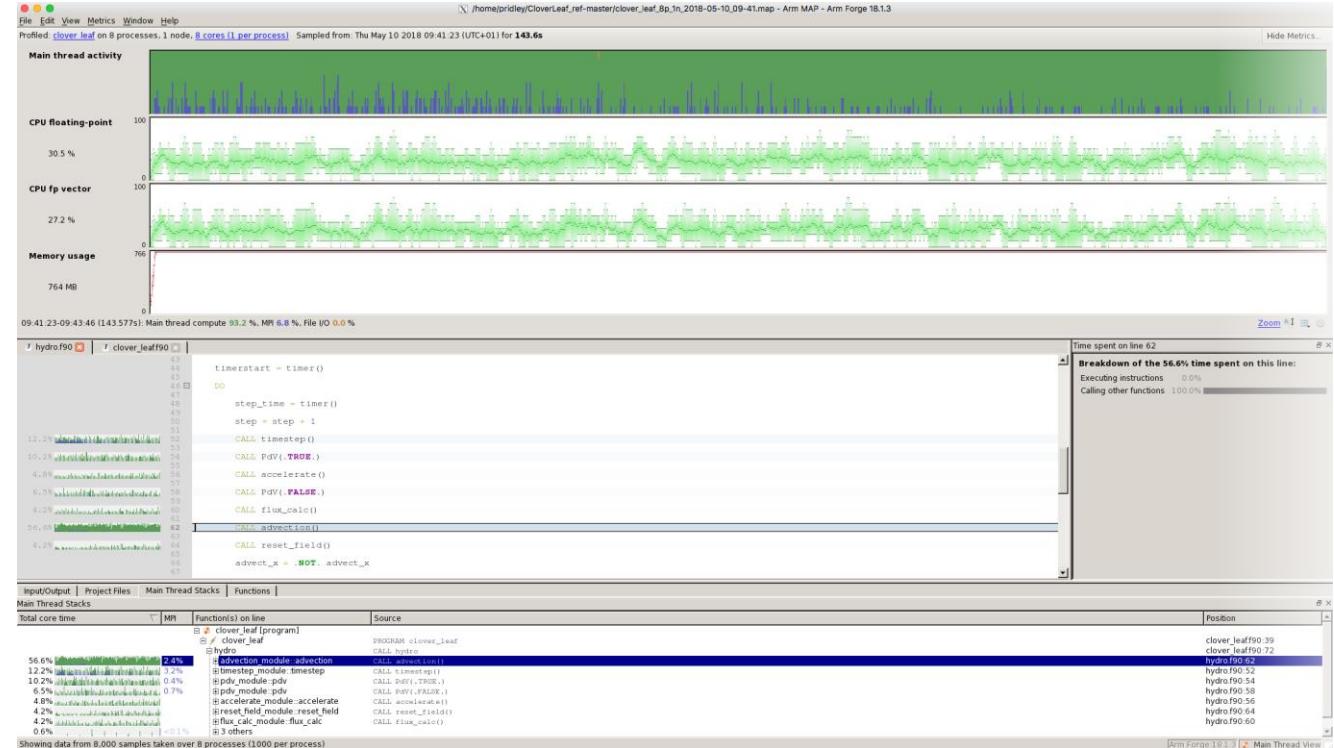
Time in collective calls	20.9%	
Time in point-to-point calls	79.1%	
Effective process collective rate	1.55 kB/s	
Effective process point-to-point rate	33.1 MB/s	

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

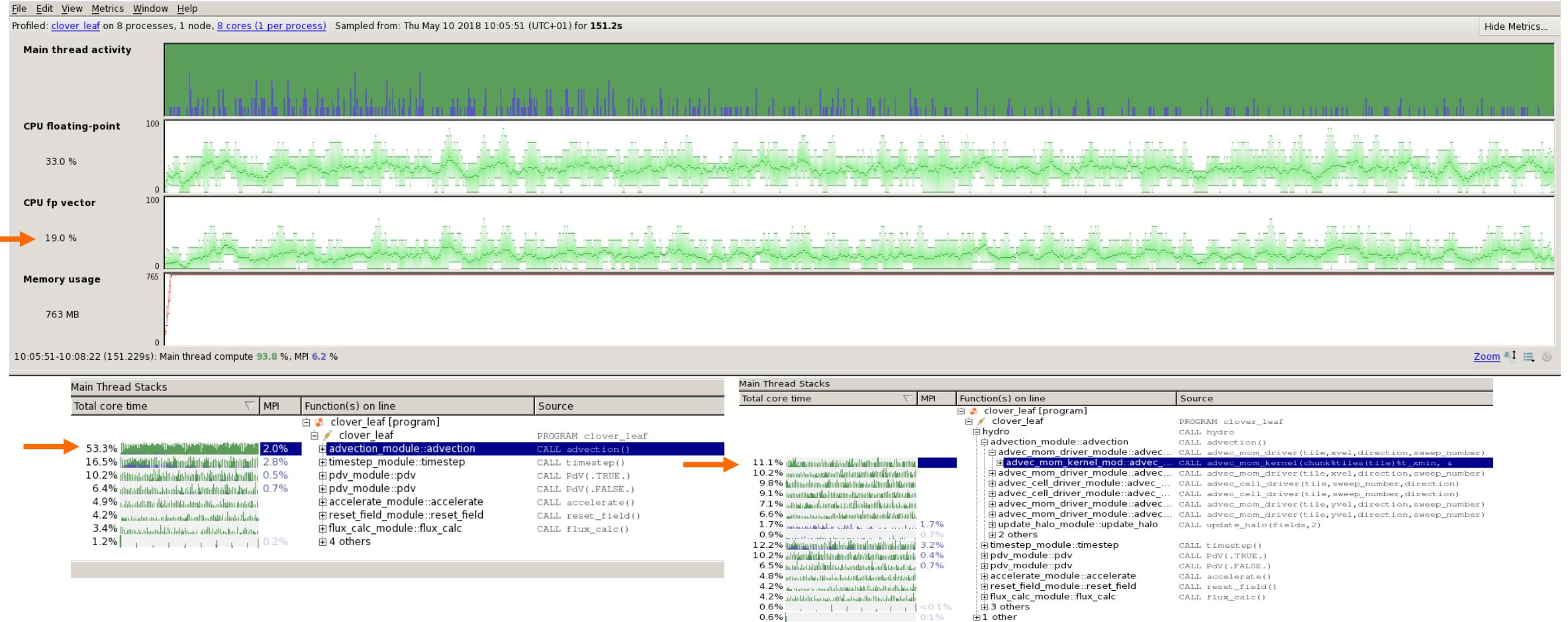
- Compiling with debugging flag required
- Shows processes and threads activity over time
- Source code is annotated
- Information aggregated by stacks and function



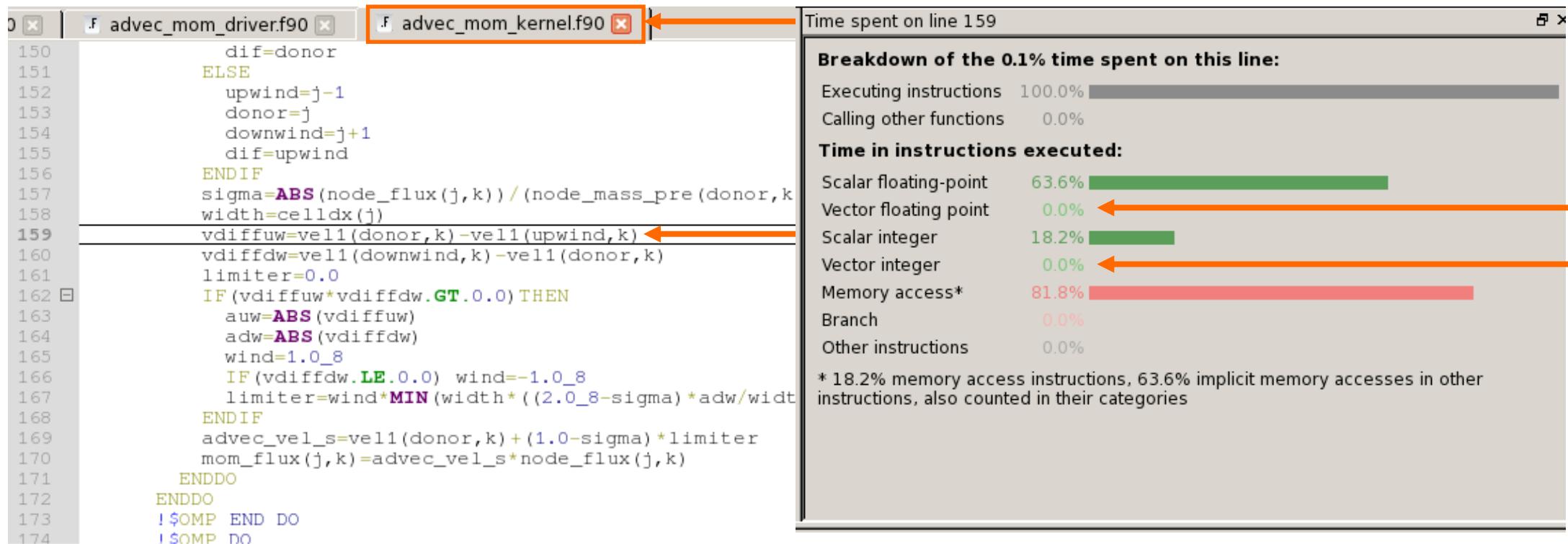
## Compute, IO and MPI

# How much of the code is vectorized?

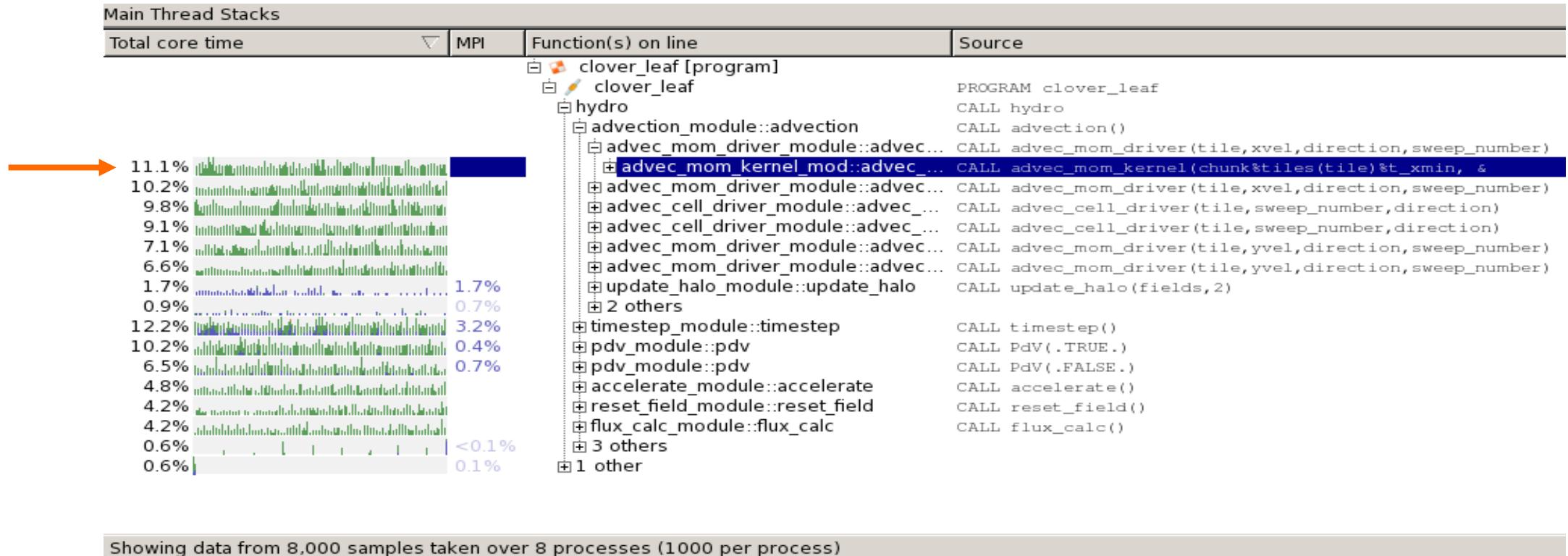
Profiled: [clover leaf](#) on 8 processes, 1 node, [8 cores \(1 per process\)](#) Sampled from: Thu May 10 2018 10:05:51 (UTC+01) for **151.2s** ←



# Where is the code vectorized?



# Follow Performance Reports advice



# Follow Performance Reports advice

advec\_mom\_kernel.f90

```
...
144  DO k=y_min,y_max+1
145  DO j=x_min-1,x_max+1 ←
146    IF(node_flux(j,k).LT.0.0)THEN
147      upwind=j+2
148      donor=j+1
149      downwind=j
150      dif=donor
151    ELSE
152      upwind=j-1
153      donor=j
154      downwind=j+1
155      dif=upwind
156    ENDIF
157    sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
158    width=celldx(j)
159    vdiffuw=vel1(donor,k)-vel1(upwind,k) ←
160    vdiffdw=vel1(downwind,k)-vel1(donor,k)
...

```

-fopt-info-vec-missed

advec\_mom\_kernel.f90:145: note: not vectorized: control flow in loop.  
advec\_mom\_kernel.f90:145: note: bad inner-loop form.  
advec\_mom\_kernel.f90:145: note: not vectorized: Bad inner loop.  
advec\_mom\_kernel.f90:145: note: bad loop form.  
Analyzing loop at advec\_mom\_kernel.f90:145

advec\_mom\_kernel.f90:145: note: not vectorized: control flow in loop.  
advec\_mom\_kernel.f90:145: note: bad loop form.

# How well is the compiler vectorizing?

advec\_mom\_kernel.f90

```
...
144  DO k=y_min,y_max+1
145  DO j=x_min-1,x_max+1 ←
146    IF(node_flux(j,k).LT.0.0)THEN
147      upwind=j+2
148      donor=j+1
149      downwind=j
150      dif=donor
151    ELSE
152      upwind=j-1
153      donor=j
154      downwind=j+1
155      dif=upwind
156    ENDIF
157    sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
158    width=celldx(j)
159    vdiffuw=vel1(donor,k)-vel1(upwind,k)
160    vdiffdw=vel1(downwind,k)-vel1(donor,k)
...

```

-qopt-report=2

LOOP BEGIN at advec\_mom\_kernel.f90(145,9)  
<Peeled loop for vectorization>  
  remark #25456: Number of Array Refs Scalar Replaced In Loop: 2  
LOOP END

LOOP BEGIN at advec\_mom\_kernel.f90(145,9)  
  remark #15300: **LOOP WAS VECTORIZED**  
LOOP END

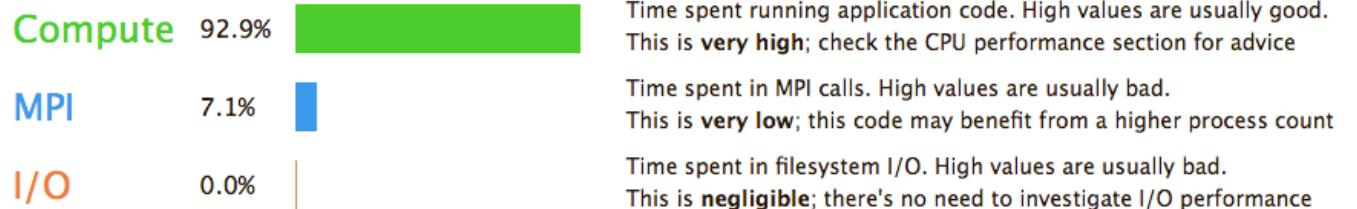
LOOP BEGIN at advec\_mom\_kernel.f90(145,9)  
<Remainder loop for vectorization>  
LOOP END

# Analyze the results

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

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

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



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

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

## CPU

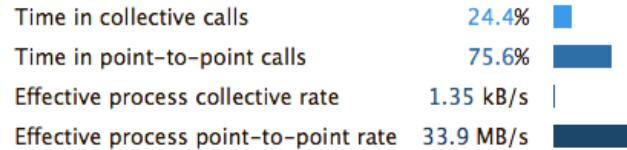
A breakdown of the **92.9%** CPU time:



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

## MPI

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



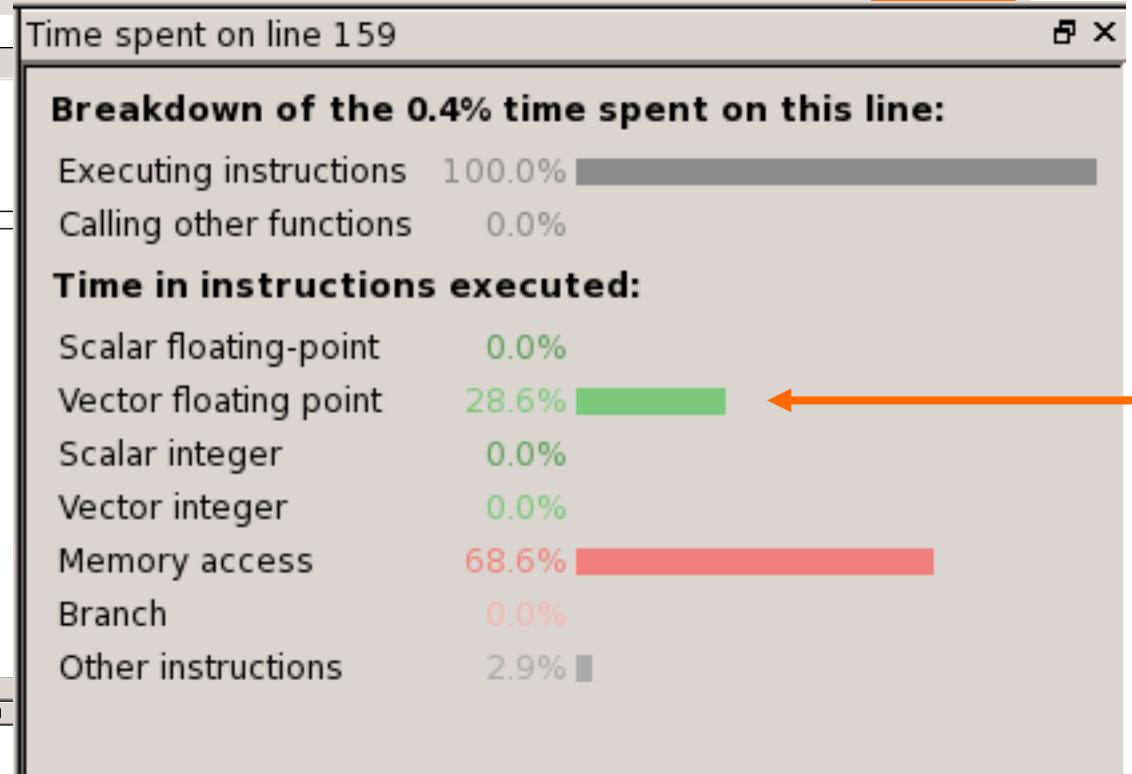
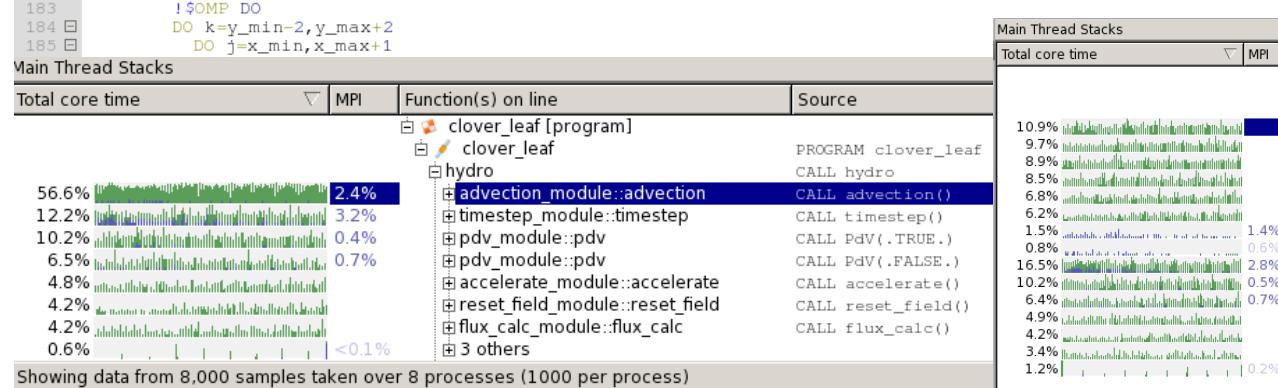
Most of the time is spent in [point-to-point calls](#) with a low transfer rate. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait.

# Where is the code vectorized?

Profiled: [clover leaf](#) on 8 processes, 1 node, [8 cores \(1 per process\)](#)

Sampled from: Thu May 10 2018 09:41:23 (UTC+01) for **143.6s** ←

```
0 x | .f advec_mom_driverf90 x | .f advec_mom_kernelf90 x |
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=cellidx(j)
159         vdiffuw=vell(donor,k)-vell(upwind,k) ←
160         vdiffdw=vell(downwind,k)-vell(donor,k)
161         limiter=0.0
162         IF(vdiffuw*vdiffdw.GT.0.0) THEN
163             auw=ABS(vdiffuw)
164             adw=ABS(vdiffdw)
165             wind=1.0_8
166             IF(vdiffdw.LE.0.0) wind=-1.0_8
167             limiter=wind*MIN(width*((2.0_8-sigma)*adw/width+(1.0_8+sigma)*auw/cellidx(dif))/6.0_8,auw,adw)
168             ENDIF
169             advec_vel_s=vell(donor,k)+(1.0-sigma)*limiter
170             mom_flux(j,k)=advec_vel_s*node_flux(j,k)
171         ENDDO
172     ENDDO
173     !$OMP END DO
174     !$OMP DO
175     DO k=y_min,y_max+1
176         DO j=x_min,x_max+1
177             vell(j,k)=(vell(j,k)*node_mass_pre(j,k)+mom_flux(j-1,k)-mom_flux(j,k))/node_mass_post(j,k)
178         ENDDO
179     ENDDO
180     !$OMP END DO
181     ELSEIF(direction.EQ.2) THEN
182     IF(which_vel.EQ.1) THEN
183         !$OMP DO
184         DO k=y_min-2,y_max+2
185             DO j=x_min,x_max+1
```



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

# 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)

# Hands – On : Workload Imbalance

# 9 Step guide: optimizing high performance applications



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.

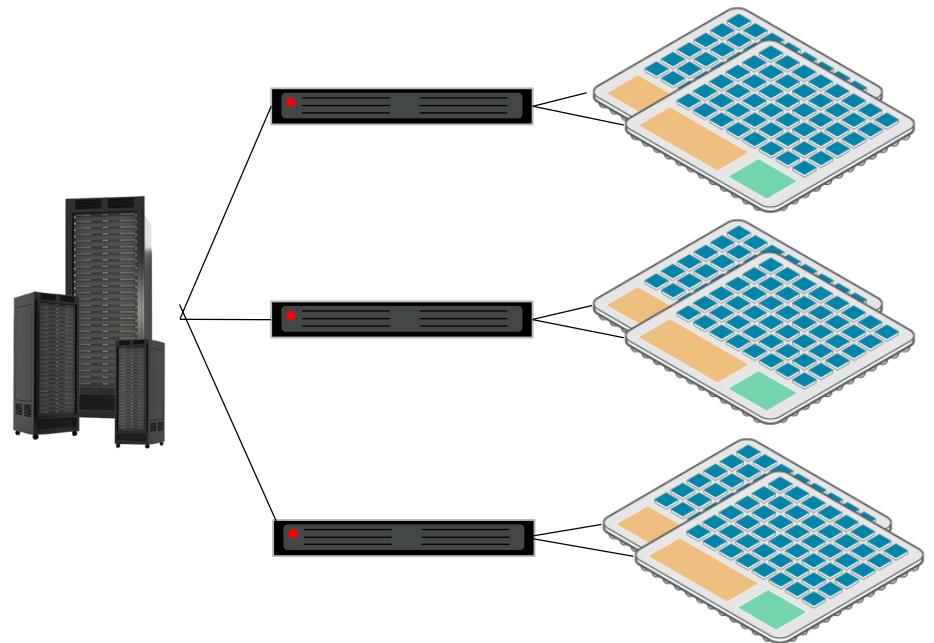


# Workload balancing: definition

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

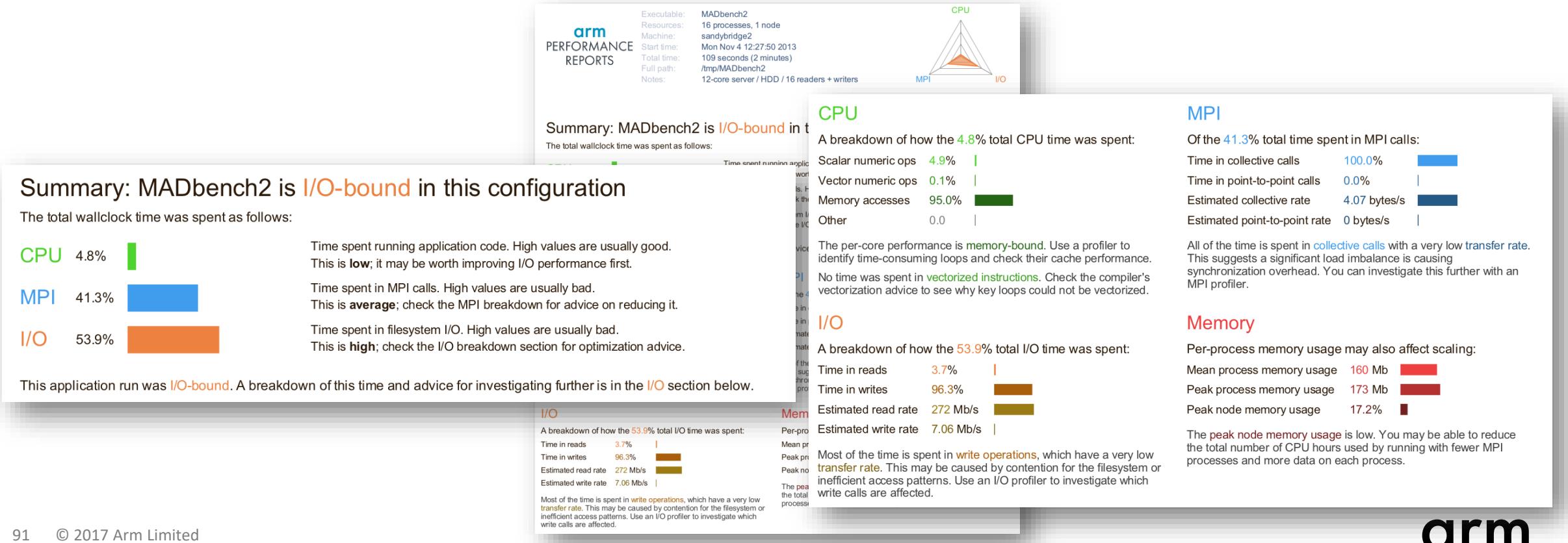
(Wikipedia)

- 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



# 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.

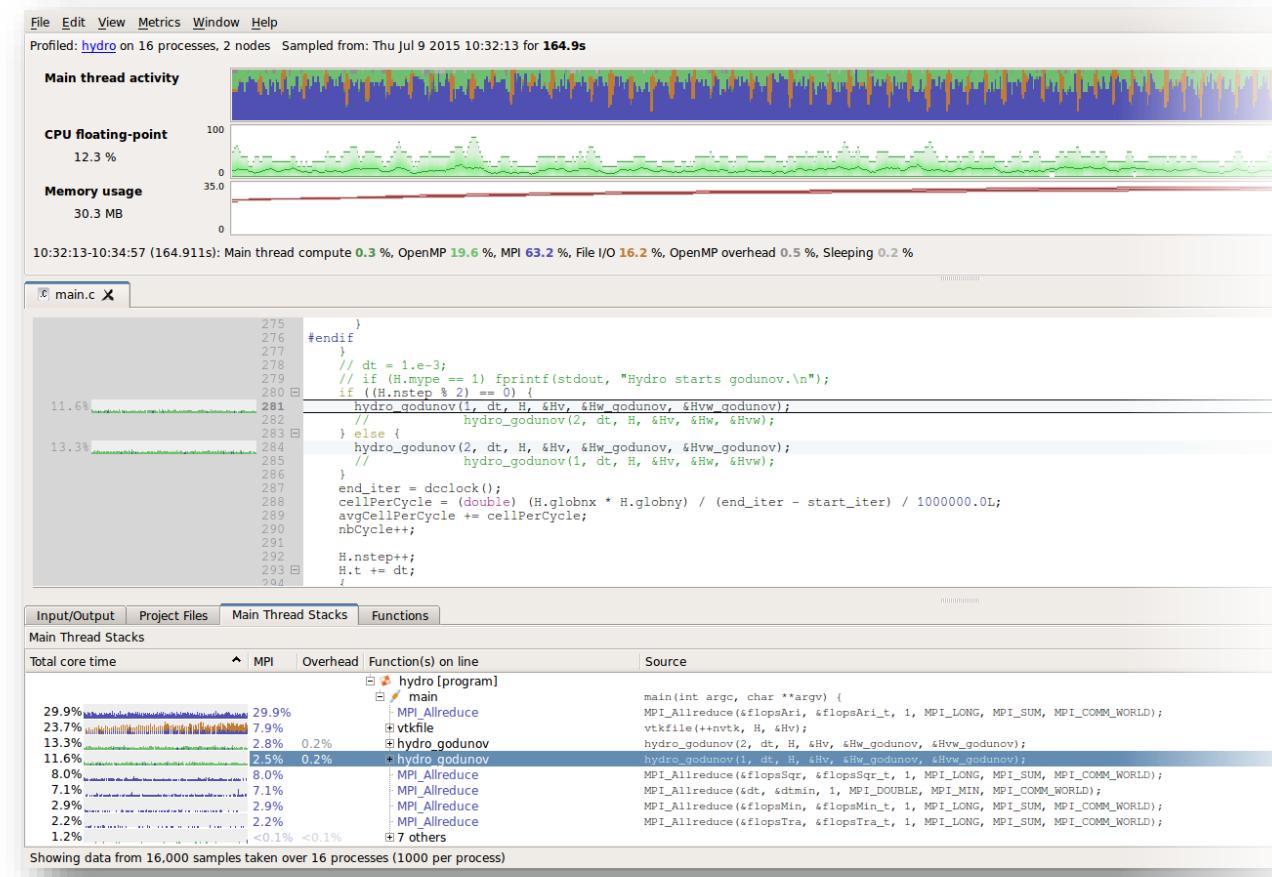
# Locate imbalance in your code



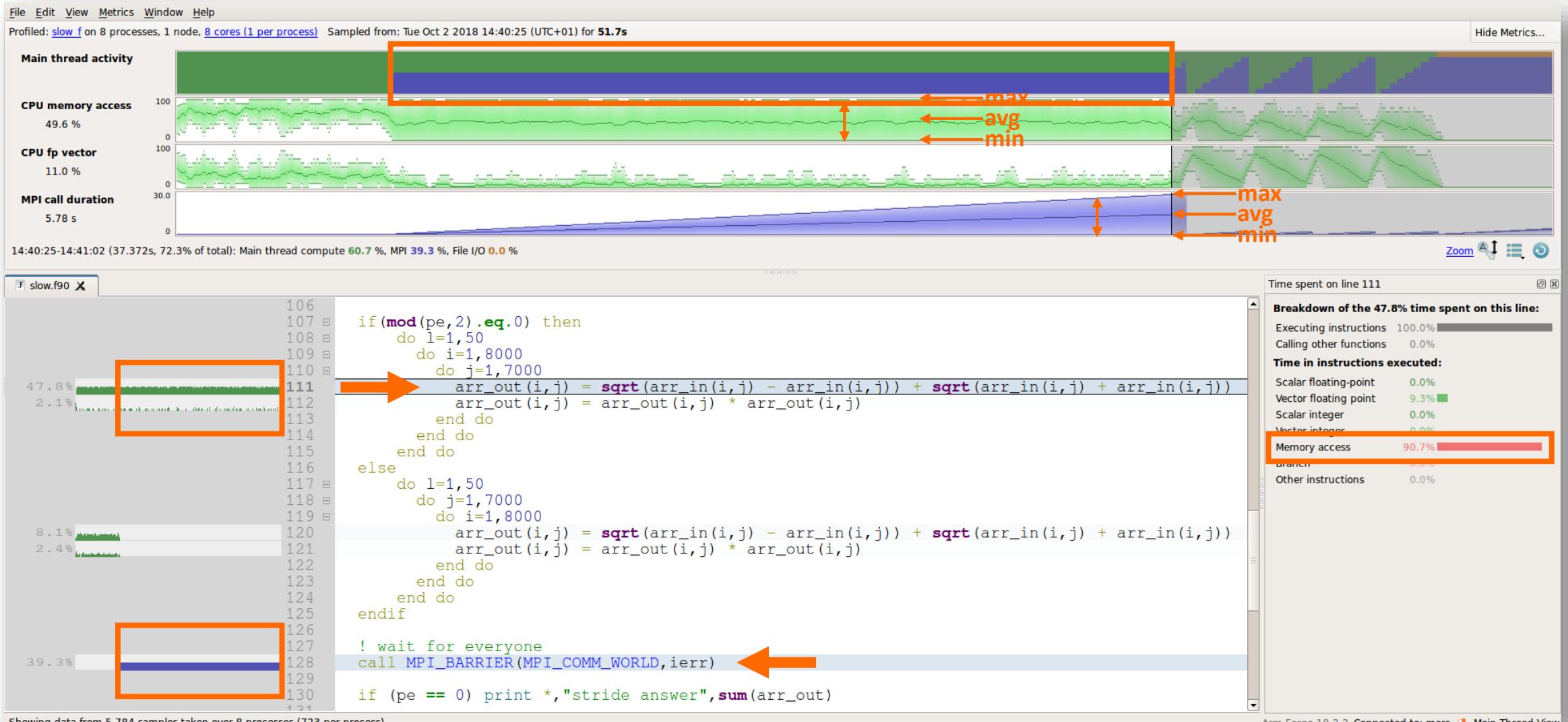
Arm MAP is a lightweight multi-node profiling tool

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

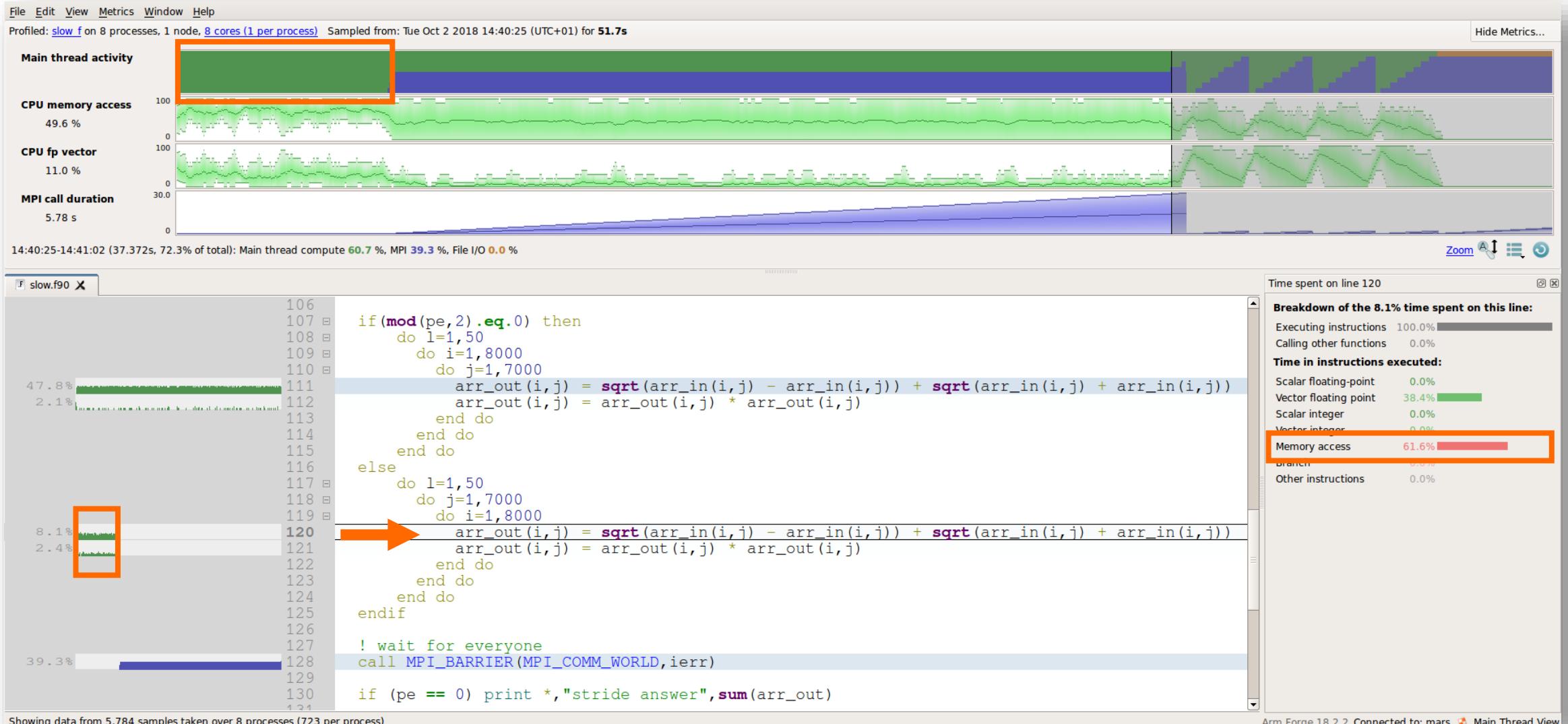
## Compute, IO and MPI



# MPI imbalance: barrier



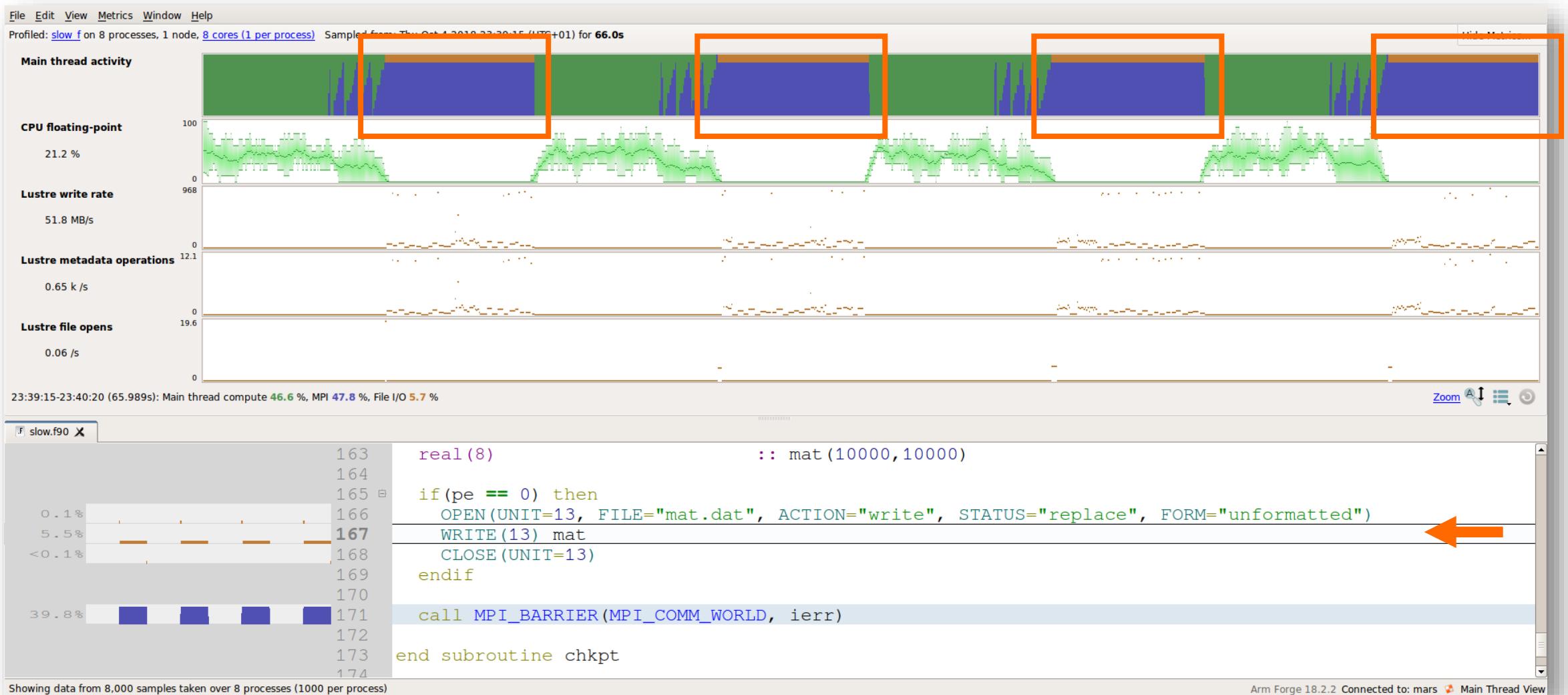
# MPI imbalance: barrier



# 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

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

For any questions :

[support-hpc-sw@arm.com](mailto:support-hpc-sw@arm.com)

CC : [conrad.hillairet@arm.com](mailto:conrad.hillairet@arm.com)

Thank You!

Danke!

Merci!

謝謝!

ありがとう!

Gracias!

Kiitos!

arm