arm

Arm Tools Workshop

Nick Forrington <nick.forrington@arm.com>

14th September 2018

Agenda

9:00	Introduction
9:30	Remote Client Setup
9:45	DDT Getting Started
10:30	15-minute break
10:45	Offline Debugging
11:15	Memory Debugging – Leaks and Errors
12:00	Lunch
13:00	Performance Reports and MAP
14:30	15-minute break
14:45	GPU Debugging and Profiling
16:00	Discussion / Finish

Performance Engineering

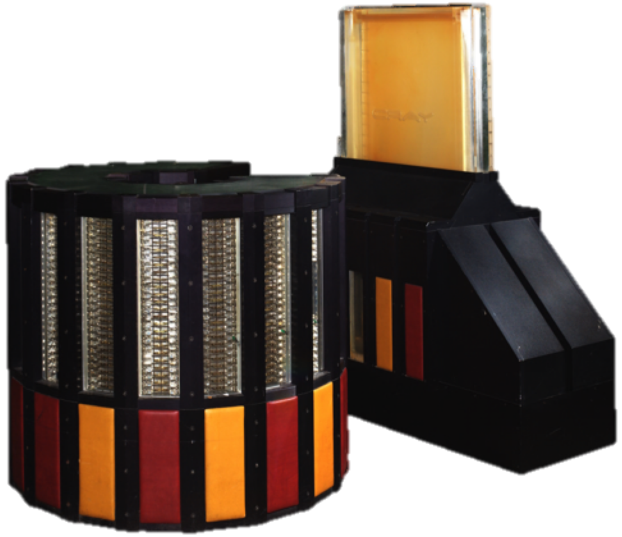
Methodology and Tools

arm

Welcome to the age of machine-scale computing

It's dangerous to go alone! Take this.

30 years ago: human-scale computing



Cray 2:
- 4 vector processors
- 1.9 gigaflops (9.5 mflops/Watt)

Today: machine-scale computing

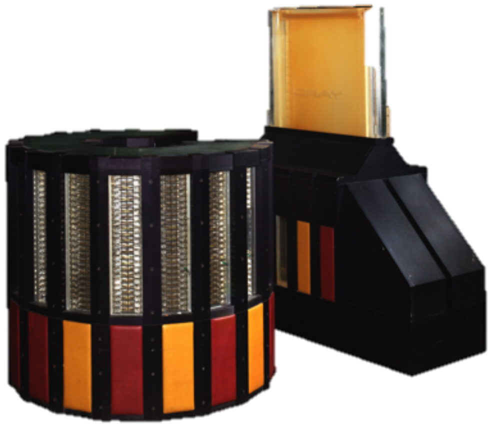
Summit:
- 2,282,544 cores
- 2,000,000 gigaflops (154 mflops/Watt)

arm

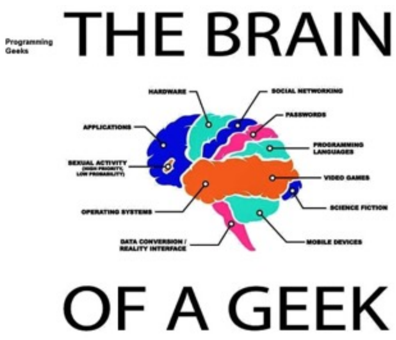
Your brain is no longer enough

No way around it, you need tools to achieve maximum performance.

- Supercomputers are now incomprehensibly complex.
- Naïve optimization may harm performance.
- **Performance engineering tools are essential** for realizing performance at scale.



<



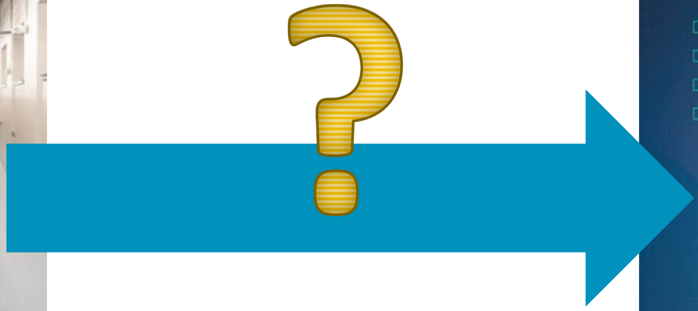
<

arm

Your brain is no longer enough

No way around it, you need tools to achieve maximum performance.

- Supercomputers are now incomprehensibly complex.
- Naïve optimization may harm performance.
- **Performance engineering tools are essential** for realizing performance at scale.



Arm's solution for *any* architecture, at *any* scale

Commercial tools for aarch64, x86_64, ppc64le and accelerators

Cross-platform Tools

arm FORGE
- DDT
- MAP

arm PERFORMANCE REPORTS

+

Arm Architecture Tools

arm C/C++ & FORTRAN COMPILER

arm PERFORMANCE LIBRARIES

=

arm ALLINEA STUDIO

- C/C++ Compiler
- Fortran Compiler
- Performance Libraries
- Forge (DDT and MAP)
- Performance Reports

Arm's solution for *any* architecture, at *any* scale

Commercial tools for aarch64, x86_64, ppc64 and accelerators

Cross-platform Tools

arm FORGE
- DDT
- MAP

arm PERFORMANCE REPORTS

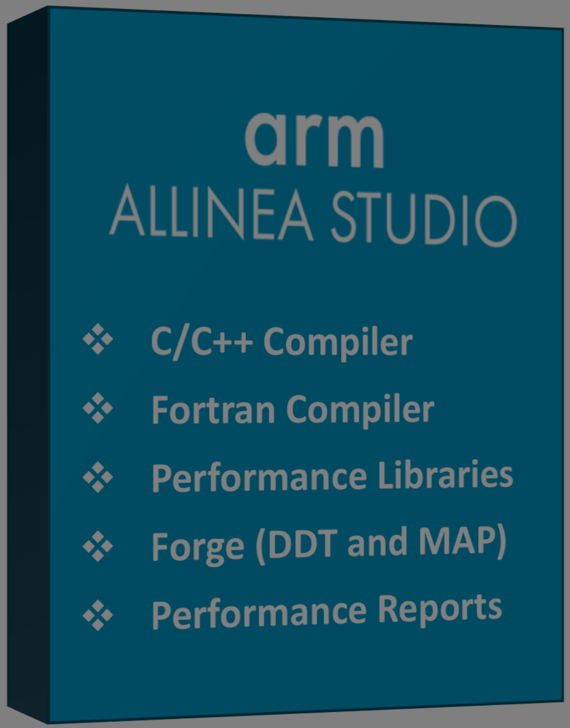
+

Arm Architecture Tools

arm C/C++ & FORTRAN COMPILER

arm PERFORMANCE LIBRARIES

=



Arm Forge = DDT + MAP

An interoperable toolkit for debugging and profiling

Commercially supported by Arm

Fully Scalable

Very user-friendly

The de-facto standard for HPC development

- Available on the vast majority of the Top500 machines in the world
- Fully supported by Arm on x86, IBM Power, Nvidia GPUs, etc.

State-of-the art debugging and profiling capabilities

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

Easy to use by everyone

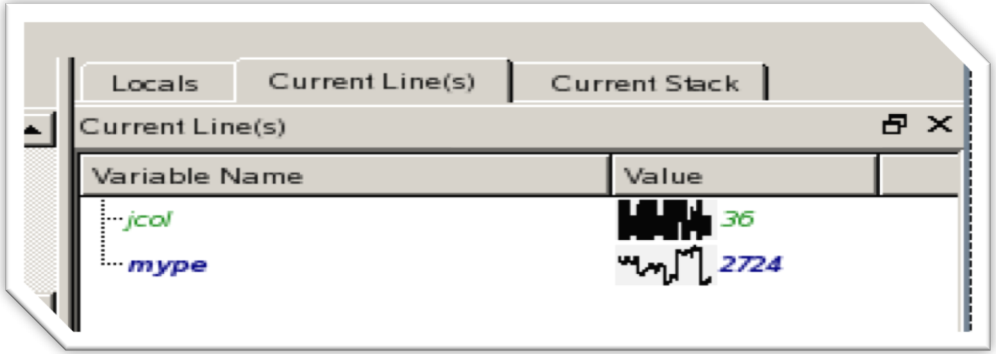
- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

arm

DDT: Production-scale debugging

Isolate and investigate faults at scale

- Which MPI rank misbehaved?
  - Merge stacks from processes and threads
  - Sparklines comparing data across processes
- What source locations are related to the problem?
  - Integrated source code editor
  - Dynamic data structure visualization
- How did it happen?
  - Parse diagnostic messages
  - Trace variables through execution
- Why did it happen?
  - Unique “Smart Highlighting”
  - Experiment with variable values



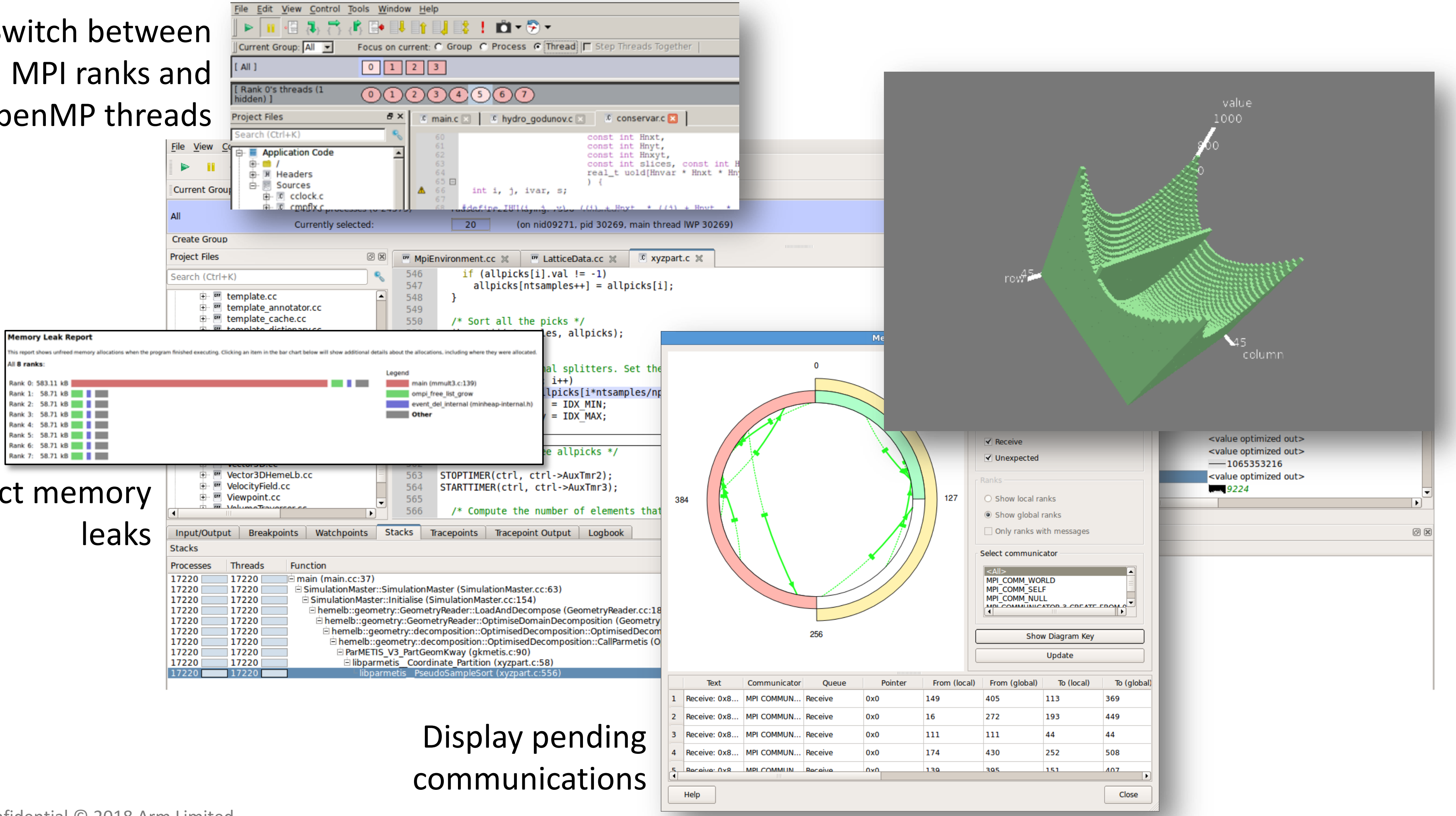
DDT: Feature Highlights

Switch between MPI ranks and OpenMP threads

Visualise arrays

Detect memory leaks

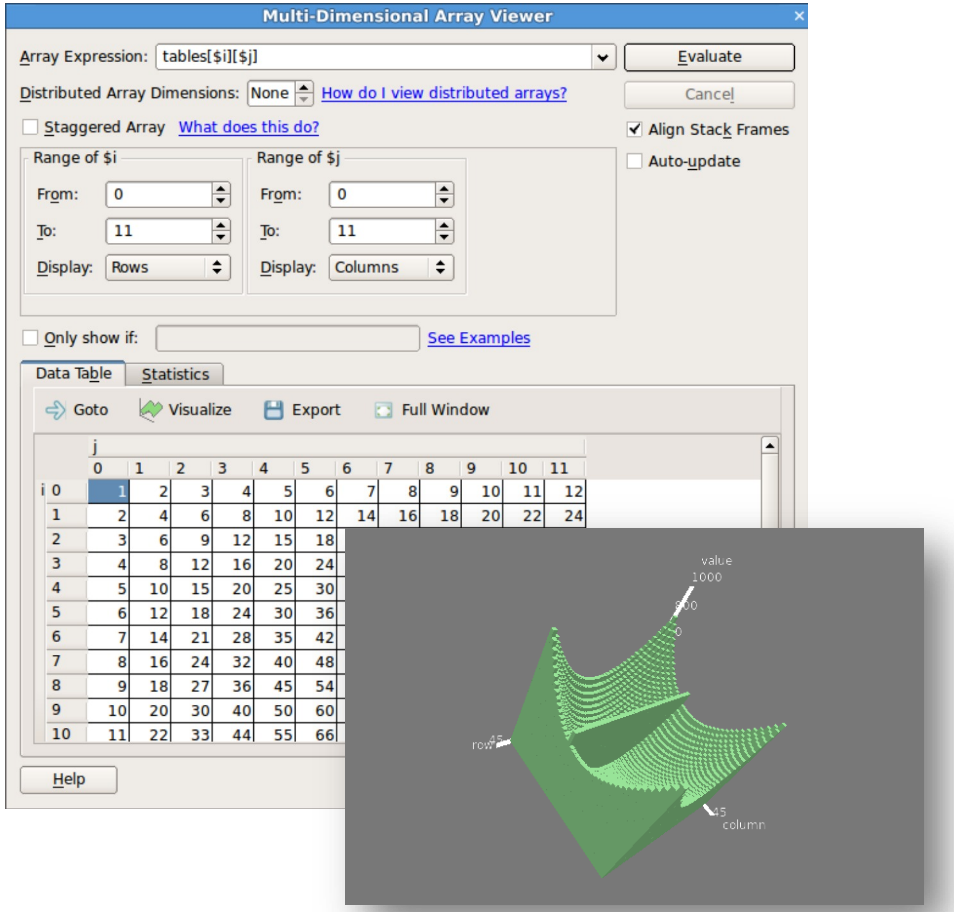
Display pending communications



Multi-dimensional Array Viewer

What does your data look like at runtime?

- View arrays
  - On a single process
  - Or distributed on many ranks
- Use metavariables to browse the array
  - Example: $i and $j
  - Metavariables are unrelated to the variables in your program.
  - The bounds to view can be specified
  - Visualise draws a 3D representation of the array
- Data can also be filtered
  - “Only show if”: $value > 0 for example $value being a specific element of the array



arm

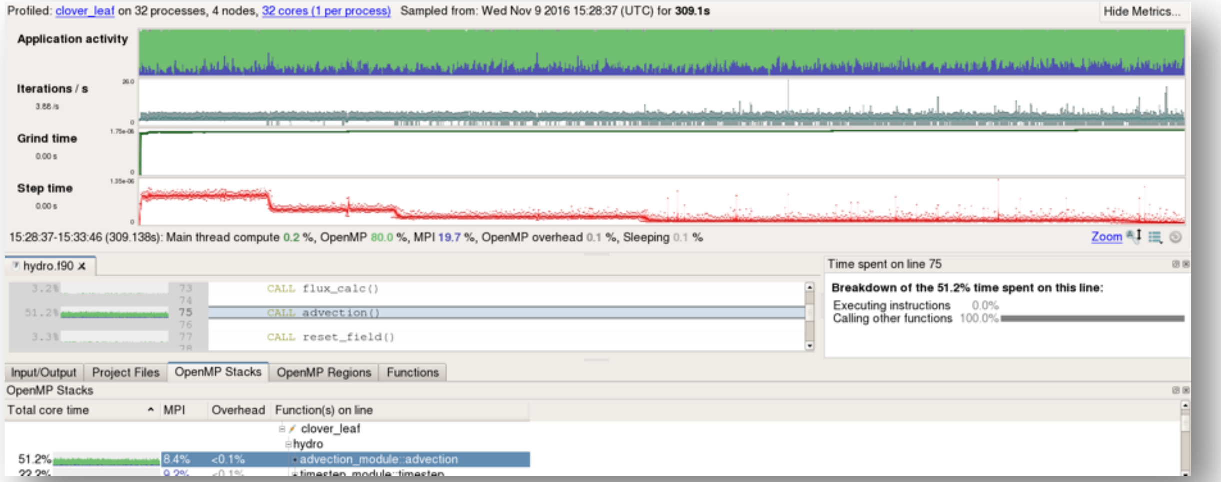
MAP: Production-scale application profiling

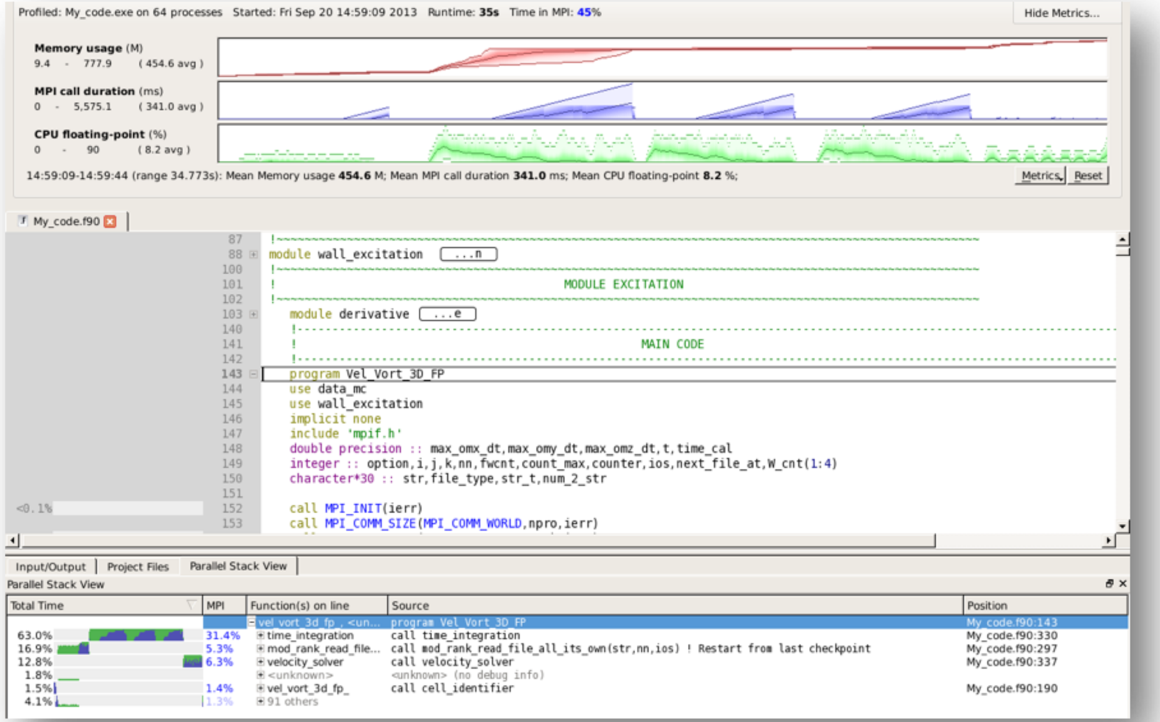
Identify bottlenecks and rewrite code for better performance

- Run with the representative workload you started with
- Measure all performance aspects with Arm Forge Professional

Examples:

```
$> map -profile aprun –n 8 ./example
$> map -profile jsrun –n 6 ./example
```





arm

How MAP is different

MAP's flagship feature is lightweight, highly scalable performance profiling

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**

arm

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)

Arm Performance Reports

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

arm
PERFORMANCE
REPORTS

Command:	mpiexec.hydra -host node-1,node-2 -map-by socket -n 16 -ppn 8 ./Bin/low_freq/../../Src//hydro -i ./Bin/low_freq/../../../../Input/input_250x125_corner.nml
Resources:	2 nodes (8 physical, 8 logical cores per node)
Memory:	15 GiB per node
Tasks:	16 processes, OMP_NUM_THREADS was 1
Machine:	node-1
Start time:	Thu Jul 9 2015 10:32:13
Total time:	165 seconds (about 3 minutes)
Full path:	Bin/../Src

Summary: hydro is MPI-bound in this configuration

Compute	20.6%	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	63.2%	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	16.2%	Time spent in filesystem I/O. High values are usually bad. This is **average**; check the I/O breakdown section for optimization advice

I/O

A breakdown of the 16.2% I/O time:

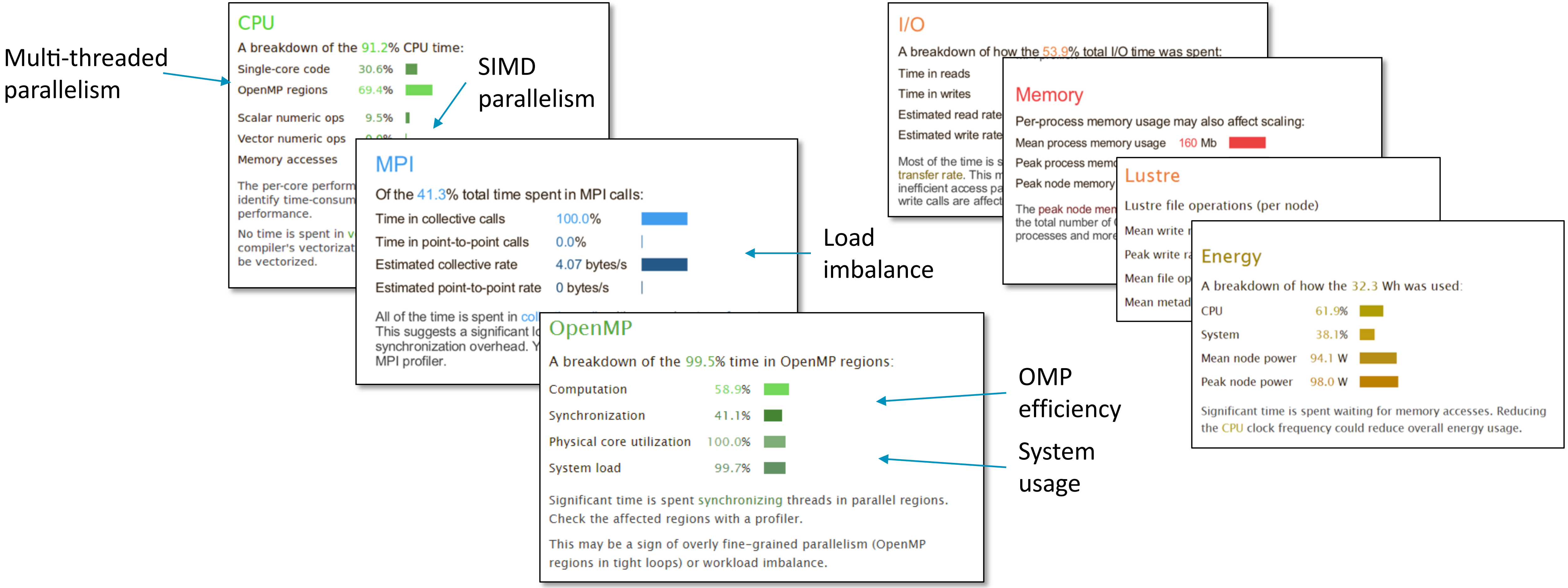
Time in reads	0.0%
Time in writes	100.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	1.38 MB/s

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

arm

Arm Performance Reports Metrics

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



Forge and Performance Reports at ORNL

- Machines
  - Titan
  - Summit
  - Wombat
  - Your laptop
  - …
- User Guides
  - https://www.olcf.ornl.gov/software_package/forge/
  - https://www.olcf.ornl.gov/software_package/arm-performance-reports/

Arm Forge Quick Start

Tool cheat sheets

arm

Arm DDT cheat sheet

Start DDT interactively, remotely, or from a batch script.

- Load the environment module:
  - `$ module load forge`
- Prepare the code:
  - `$ cc  -O0 -g myapp.c -o myapp.exe`
  - `$ ftn -O0 -g myapp.f -o myapp.exe`
- Start DDT in interactive mode:
  - `$ ddt aprun -n 8 ./myapp.exe arg1 arg2 …`
- Or use reverse connect:
  - Connect the remote client (or launch "**ddt**" on the login node)
  - Run the follow command, or edit a job script and submit:
    - `$ ddt --connect aprun -n 8 ./myapp.exe arg1 arg2 …`
- Offline mode
  - `$ ddt --offline aprun -n 8 ./myapp.exe arg1 arg2 …` (see ddt --help for more options)

Arm MAP cheat sheet

Generate profiles and view offline

- Load the environment module
  - $ module load **forge**
- Prepare the code
  - `$ cc  -O3 … -g myapp.c -o myapp.exe`
  - `$ ftn -O3 … -g myapp.f -o myapp.exe`
- Interactive (Collect and View)
  - `$ map aprun –n8 ./myapp.exe arg1 arg2`
- Offline: edit the job script to run Arm MAP in “profile” mode
  - `$ map --profile aprun –n8 ./myapp.exe arg1 arg2`
- View profile in MAP:
  - On the login node:
    - `$ map myapp_Xp_Yn_YYYY-MM-DD_HH-MM.map`
  - (or load the corresponding file using the remote client connected to the remote system or locally)

arm

Arm Performance Reports cheat sheet

Generate text and HTML reports from application runs or MAP files

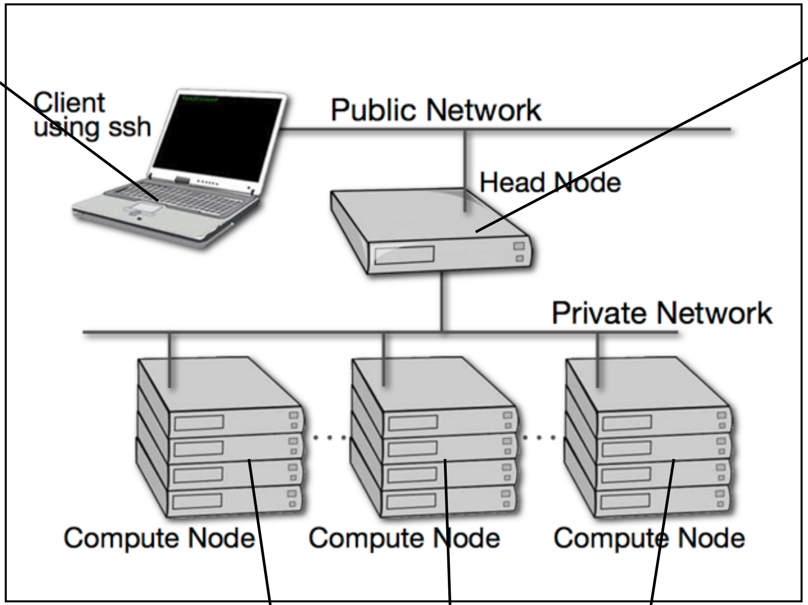
- Load the environment module:
  - `$ module load` **`perf-reports`**
- No need to prepare application
- Run the application:
  - **`perf-report`** `aprun -n 8 ./myapp.exe`
- … or, if you already have a MAP file:
  - **`perf-report`** `myapp_8p_1n_YYYY-MM-DD_HH:MM.txt`
- Analyze the results
  - `$ cat myapp_8p_1n_YYYY-MM-DD_HH:MM.txt`
  - `$ firefox myapp_8p_1n_YYYY-MM-DD_HH:MM.html`

Forge Remote Client

The Forge GUI and where to run it

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

Remote client
(remote launch + reverse connect)



On the head node
(interactive mode + reverse connect)

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

After connecting the client

Three options to proceed

Click run and launch via the the GUI

- Works well simple jobs
- DDT can launch a batch job for you
- Can be tricky to replicate complicated launch environments or flags

Edit a batch script to use ddt --connect

- Best option for complex batch scripts
- Also for long running non-interactive jobs
-
```
. $MODULESHOME/init/bash
module load forge
ddt --connect aprun …
```

Use ddt --connect from an interactive session

- Useful if you want to try many runs within different launch options/environments

Launching the Forge Remote Client

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

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

- https://developer.arm.com/products/software-development-tools/hpc/downloads/download-arm-forge
  - Searching for “Arm Forge Download” will typically take you here
- https://www.olcf.ornl.gov/tutorials/forge-remote-client-setup-and-usage/

Connect to the cluster with the remote client

- Open Forge Remote Client
- Create a new connection: Remote Launch ➔ Configure ➔ Add
  - Hostname: `<username>@titan.ccs.ornl.gov`
  - Remote installation directory: `/sw/xk6/forge/18.2.2/sles11_binary`
    - You can also get the above path by: `module load forge/18.2.2; echo $DDT_HOME`
- Connect!

- Training material: `~nforr/training/arm-tools-workshop.tar.gz`

Working with the queue

- Connect the remote client
- In a terminal, SSH to Titan and launch and interactive session
  - `qsub -I -A <account> -q debug -l nodes=1,walltime=01:00:00`
- `module load forge/18.2.2`
- Launch aprun command prefixed with `ddt --connect`

DDT Getting Started

Crash and hang

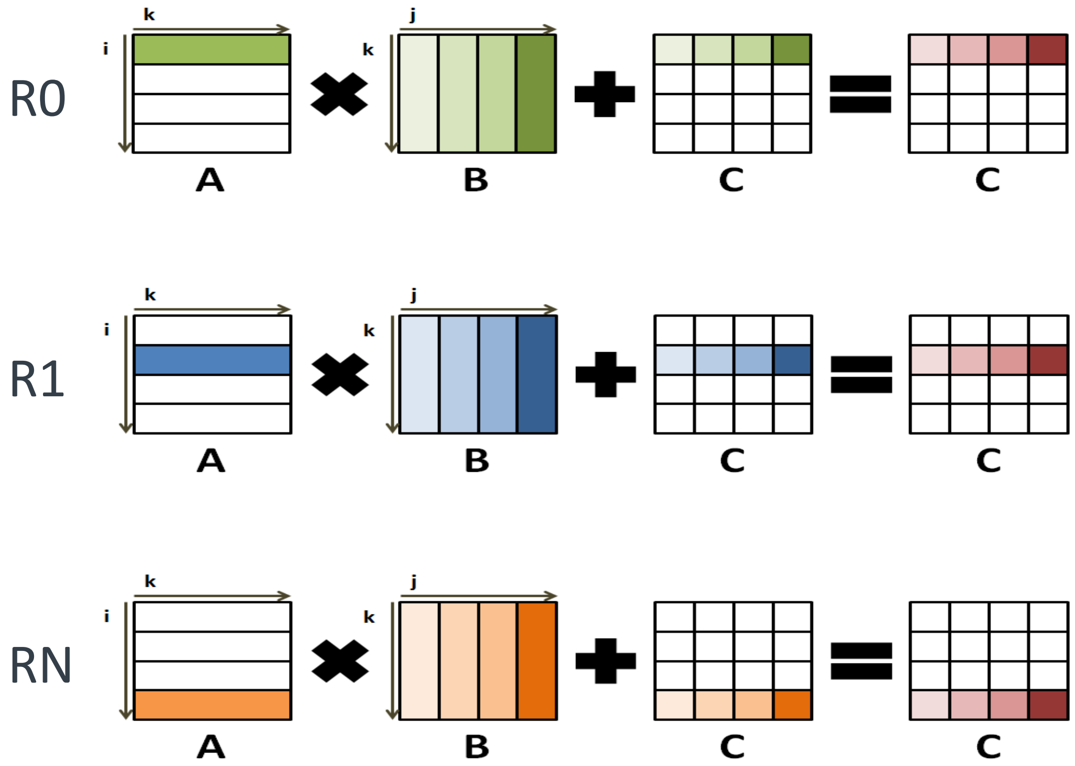
arm

C = A x B + C

Simply multiply and add two matrices

Algorithm

1. Rank 0 (R0) initialises matrices A, B & C
2. R0 slices the matrices A & C and sends them to Rank 1…N (R1+)
3. R0 and R1+ perform the multiplication
4. R1+ send their results back to R0
5. R0 writes the result matrix C to file



arm

Fix a simple crash in a MPI code

Simple matrix multiply and add? No problem! Except that it crashes…

Exercise Outline

- **Objectives**
  - Discover Arm DDT's interface
  - Interactively debug a crash in a MPI application
- **Commands**

```
$ make
$ aprun –n 4 ./mmult1_c.exe
# Observe crash
$ ddt --connect ./mmult1_c.exe
# Observe cause of crash
```

Initial Result: Crash!

```
johlin02@johlin02-VM: ~/MUG18/01_walkthrough/1_crash
johlin02@johlin02-VM:~/MUG18/01_walkthrough/1_crash$ make
mpicc -g -ffast-math -O0 -DDEBUG -std=c99 mmult1.c -o mmult1_c.exe -lm
mpif90 -g -ffast-math -O0 -DDEBUG -cpp mmult1.f90 -o mmult1_f90.exe -lm
johlin02@johlin02-VM:~/MUG18/01_walkthrough/1_crash$ mpirun -np 4 ./mmult1_c.exe
0: Size of the matrices: 64x64
0: Initializing matrices...
0: Sending matrices...
0: Processing...
[johlin02-VM:mpi_rank_0][error_sighandler] Caught error: Segmentation fault (signal 11)
3: Receiving matrices...
2: Receiving matrices...
1: Receiving matrices...
2: Processing...
[johlin02-VM:mpi_rank_2][error_sighandler] Caught error: Segmentation fault (signal 11)
1: Processing...
[johlin02-VM:mpi_rank_1][error_sighandler] Caught error: Segmentation fault (signal 11)

===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 9160 RUNNING AT johlin02-VM
=   EXIT CODE: 139
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Segmentation fault (signal 11)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
johlin02@johlin02-VM:~/MUG18/01_walkthrough/1_crash$ $
```

Answer: Fix incorrect limits on k-loop

Incorrect limits lead to invalid memory access

Before

```
do i=0,size/nslices-1
  do j=0,size-1
    res=0.0
    do k=size,size*size
     res=A(i*size+k)*B(k*size+j)+res
    end do
    C(i*size+j)=res+C(i*size+j)
  end do
end do
```

After

```
do i=0,size/nslices-1
  do j=0,size-1
    res=0.0
    do k=0,size-1
     res=A(i*size+k)*B(k*size+j)+res
    end do
    C(i*size+j)=res+C(i*size+j)
  end do
end do
```


Problem #2

Fixing the crash reveals another issue

- Run the program again, and found out why the program now hangs
- Either launch again with DDT
- Or launch without, and attach
  - Ensure your nodes file is set to $DDT_HOME/titan.nodes in the options dialog
  - Click attach, from the welcome page. This will may result in SSH prompts as DDT scans the other Titan login/batch nodes, before detecting your job
  - Alternatively, launch: ddt --connect --attach-mpi=<aprun-pid>

Program now hangs

```
0 : Size of the matrices:          64 x          64
0 : Initializing matrices...
1 : Receiving matrices...
2 : Receiving matrices...
3 : Receiving matrices...
0 : Sending matrices...
1 : Processing...
0 : Processing...
2 : Processing...
1 : Sending result matrix...
2 : Sending result matrix...
0 : Receiving result matrix...
```


arm

Answer: Fix incorrect limits on i-loop

Incorrect limits on i-loop lead to unmatched MPI_Send

Before

```
    do i=1,nproc-2
      call MPI_Send(mat_a(slice*i), slice, &
                    MPI_DOUBLE, i, 100+i, &
                    MPI_COMM_WORLD, ierr)
      call MPI_Send(mat_b, size*size, &
                    MPI_DOUBLE, i, 200+i, &
                    MPI_COMM_WORLD, ierr)
      call MPI_Send(mat_c(slice*i), slice, &
                    MPI_DOUBLE, i, 300+i, &
                    MPI_COMM_WORLD, ierr)
    end do
```

After

```
    do i=1,nproc-1
      call MPI_Send(mat_a(slice*i), slice, &
                    MPI_DOUBLE, i, 100+i, &
                    MPI_COMM_WORLD, ierr)
      call MPI_Send(mat_b, size*size, &
                    MPI_DOUBLE, i, 200+i, &
                    MPI_COMM_WORLD, ierr)
      call MPI_Send(mat_c(slice*i), slice, &
                    MPI_DOUBLE, i, 300+i, &
                    MPI_COMM_WORLD, ierr)
    end do
```


Offline Debugging

Run DDT in offline mode

Run the application under DDT and halt or report when a failure occurs.

- You can run the debugger in non-interactive mode
  - For long-running jobs
  - For automated testing, continuous integration…

- To do so, use the following arguments:
  - `$ ddt` **`--offline --output=report.html`** `aprun ./myapp.exe`
    - **`--offline`** enable non-interactive debugging
    - **`--output`** specifies the name and output of the non-interactive debugging session
      - Html
      - Txt
    - Add **`--mem-debug`** to enable memory debugging **and memory leak detection**
    - Add **`--break-at=<location>`** to report stacks and variables at certain locations
    - Add **`--trace-at=<location>,variable1,variable2`** to evaluate variables/expressions at certain locations
    - See **`--help`** for more information

Offline Log

Snippet from an earlier crash

Process stopped in mmult (mmult1.f90:168) with signal SIGSEGV (Segmentation fault).
Reason/Origin: address not mapped to object (attempt to access invalid address)

Additional Information

▼ Stacks

Processes	Function	Source	Variables
	mmult2 (mmult1.f90:92)	▶ call mmult(size, nproc, mat_a, mat_b, mat_c)	▶ Rank 0, thread 1
	mmult (mmult1.f90:168)	▼ res=A(i*size+k)*B(k*size+j)+res	▼ Rank 0, thread 1

```
165.        do j=0,size-1
166.          res=0.0
167.          do k=size,size*size
168.           res=A(i*size+k)*B(k*size+j)+res
169.          end do
170.          C(i*size+j)=res+C(i*size+j)
171.        end do
```

Name	Value
a	<aggregate value>
b	<aggregate value>
c	<aggregate value>
i	0
j	0
k	*260 (from 260 to 262)*
nslices	4
res	*5380641 (from 4189752 to 13189176)*
size	64

arm

When to use offline debugging

- If you're not available
  - e.g. when you have a long wait in the queue
- Scriptable
  - Debug many jobs
  - Nightly builds / Continuous integration

Memory Debugging

Allocation tracking and guard pages

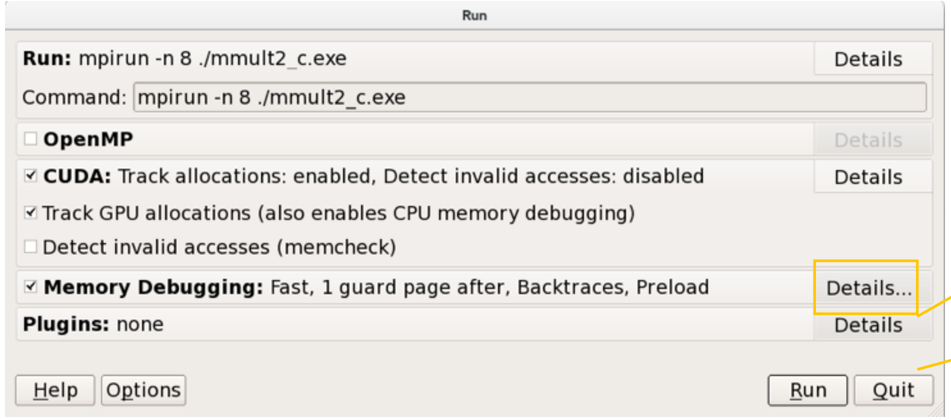
DDT's heap memory debugging framework

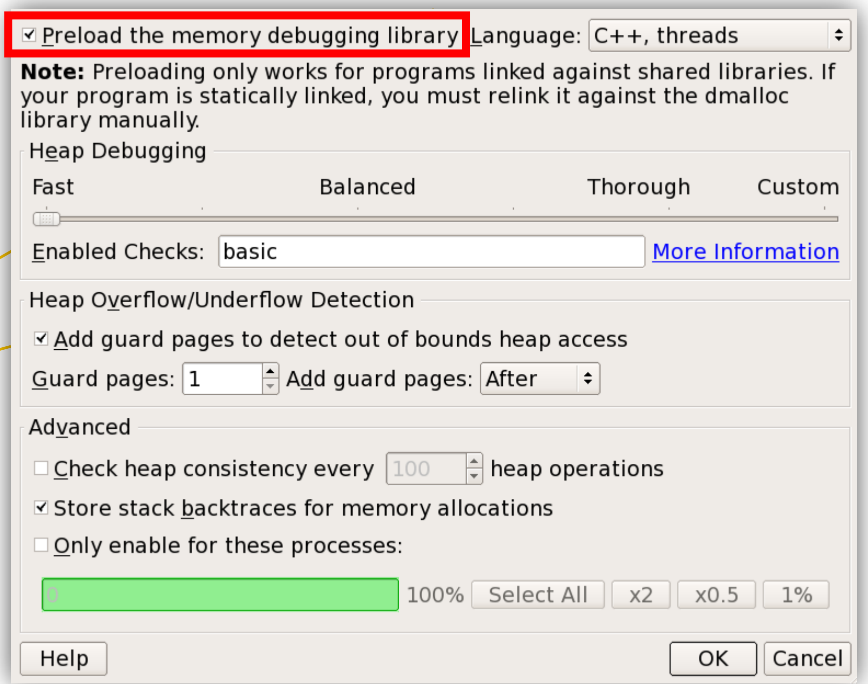
On Titan, we need to link DDT's memory debugging libraries

- Caveat: Does not work with PGI and Fortran
- Handled by helper module loaded after the forge module
- `module load forge/18.2.2; module load ddt-memdebug`

Other systems (including Summit)

- No linking required for dynamically linked binaries (handled by LD_PRELOAD)
- For static binaries, check the Forge user guide





When manual linking is used, untick "Preload" box

Three levels of heap debugging overhead

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 goodness

- 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.

See user-guide:

Chapter 12.3.2

arm

Tri-diagonal solve: segmentation fault

Crashing with invalid memory reference. Sounds like a job for a memory debugger!

Exercise Outline

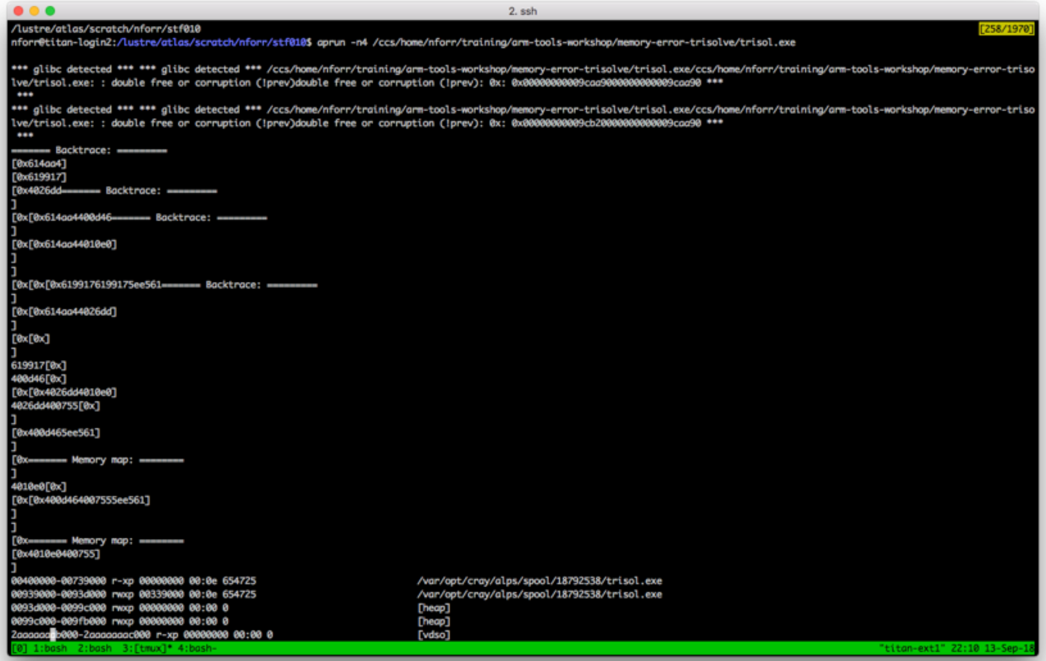
- **Objectives**
  - Use DDT's memory debugging features
  - Use guard pages to find out-of-bounds access
- **First lets run without DDT**

```
$ module swap PrgEnv-pgi PrgEnv-gnu
$ make
$ aprun -n 4 ./trisol.exe
```

- **Now let's see where it crashes in DDT (without memory debugging)**

```
$ ddt --connect aprun -n 4
./trisol.exe
```

Invalid memory access



arm

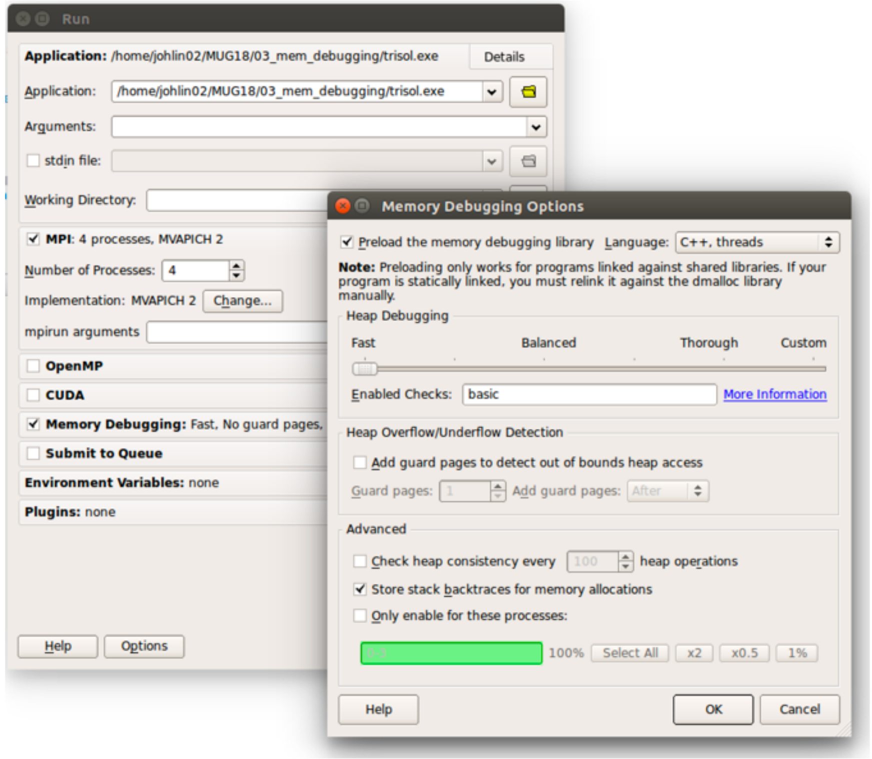
Let’s try memory debugging

Relink

- `module load ddt-memdebug`
- `make clean; make`
- `ddt --connect aprun -n 4 ./trisol.exe`

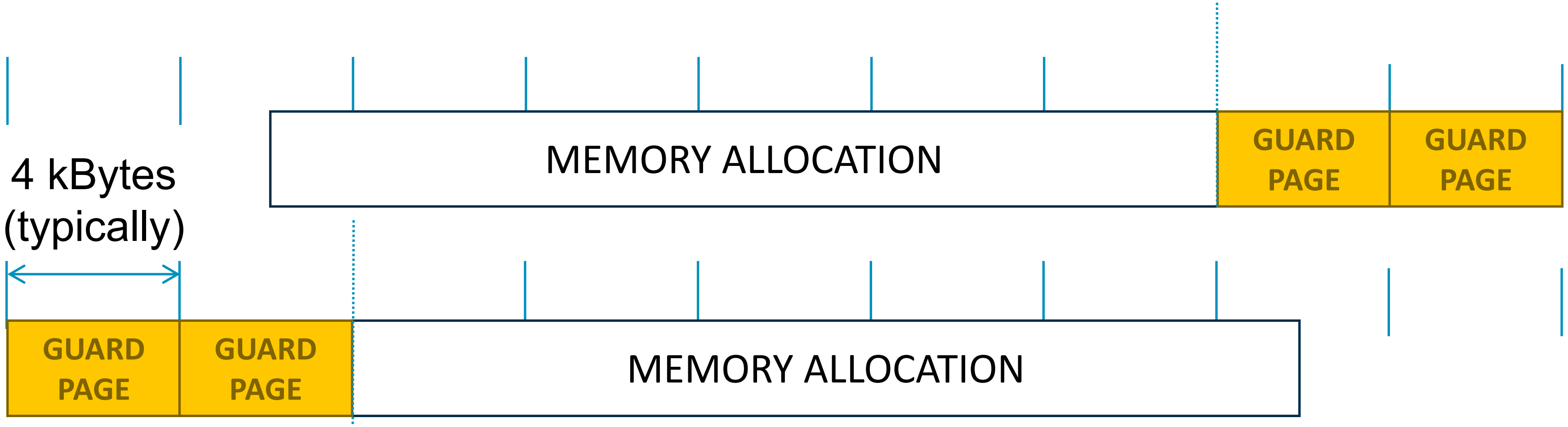
- Launch without guard pages enabled and “Fast” heap debugging.
- The program seems to run fine now - why?

And launch in DDT



arm

Guard pages (aka “electric fences”)



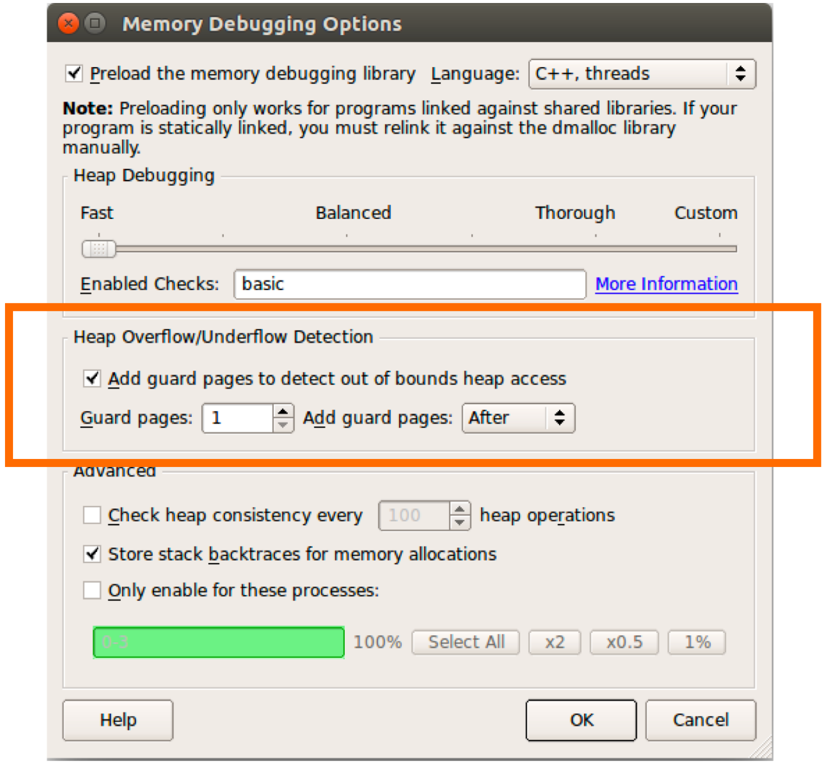
- **A powerful feature…:**
  - Forbids read/write on guard pages throughout the whole execution

    (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

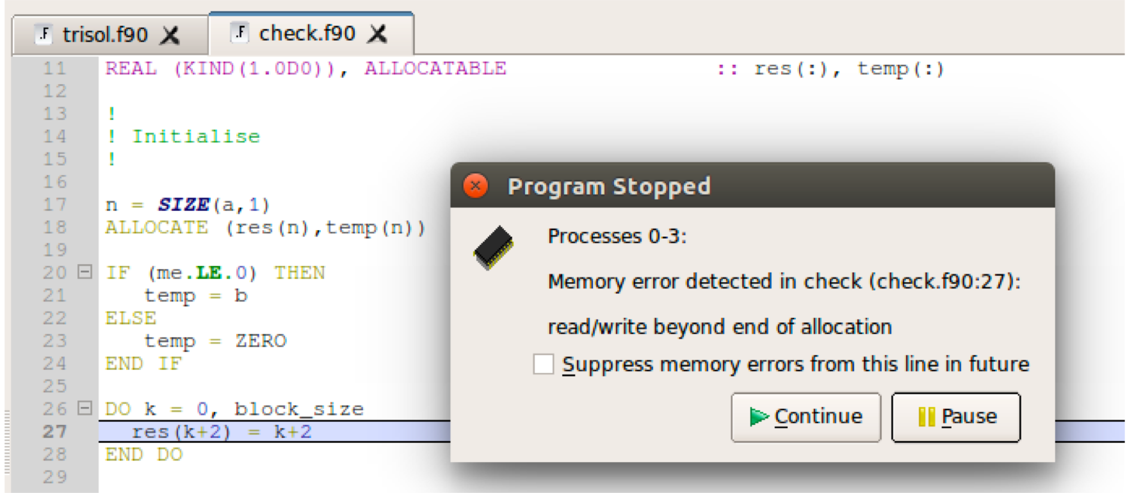
OK, this time enable guard pages

The code appears to run fine when launched from the debugger! Why?

Add one guard page after every allocation



Gotcha! Write OOB at res(k+2)



Memory Leak Detection

… and DDT in Offline Mode

arm

Three levels of heap debugging overhead

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 goodness
- 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.

See user-guide:

Chapter 12.3.2

arm

Possible memory leak

Program is working great, but sometimes I run out of memory?

Exercise Outline

- **Objectives**
  - Use DDT in offline mode
  - Explore DDT's report logbook
- **Commands**

```
$ make
$ ddt --offline \
      --output=report.html \
      aprun -n 4 \
      ./mmult3_f90.exe
$ xdg-open leak-report.html
# Observe report
```

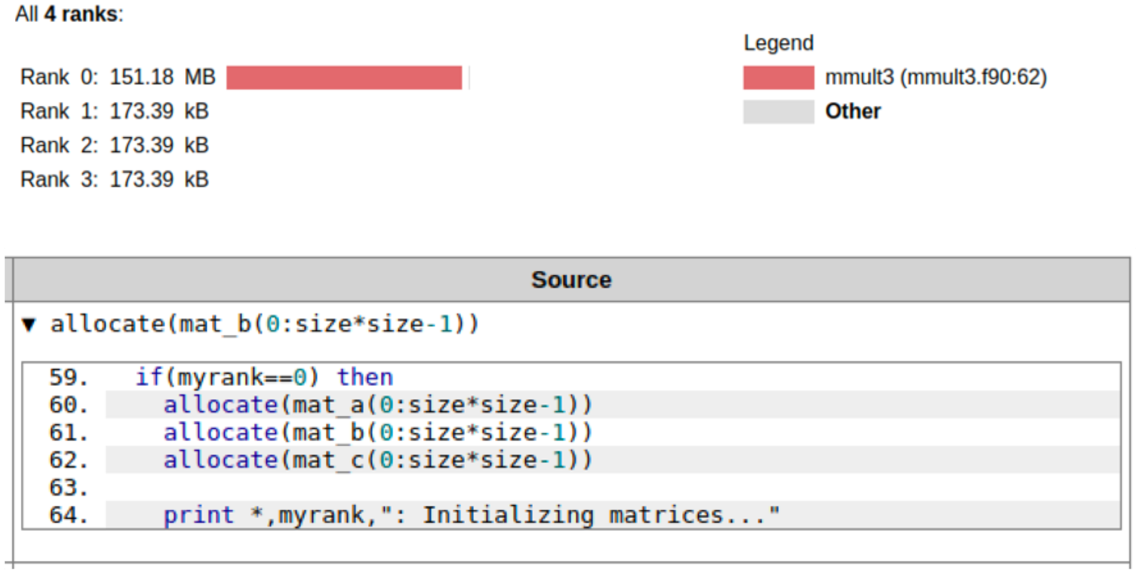
DDT in offline mode (--offline)

```
johlin02@johlin02-VM: ~/MUG18/01_walkthrough/3_memory_leak
johlin02@johlin02-VM:~/MUG18/01_walkthrough/3_memory_leak$ ddt --offline --output=report.html --mem-
debug -n 4 ./mmult3_f90.exe
Arm Forge 18.2.1 - Arm DDT

Debugging                : /home/johlin02/MUG18/01_walkthrough/3_memory_leak/mmult3_f90.exe
MPI implementation       : Auto-Detect (MVAPICH 2)
* number of processes    : 4
* number of nodes        : 1
Memory debugging enabled : Yes
* setting                : Fast
* check bounds           : Off

           0 : Size of the matrices:         3072 x        3072
           0 : Initializing matrices...
           1 : Receiving matrices...
           2 : Receiving matrices...
           3 : Receiving matrices...
           0 : Sending matrices...
           1 : Processing...
           2 : Processing...
           0 : Processing...
           3 : Processing...
           1 : Sending result matrix...
           2 : Sending result matrix...
           0 : Receiving result matrix...
           3 : Sending result matrix...
           0 : Writing results...
           0 : Done.

Offline log written to: '/home/johlin02/MUG18/01_walkthrough/3_memory_leak/report.html'

johlin02@johlin02-VM:~/MUG18/01_walkthrough/3_memory_leak$ xdg-open report.html
```

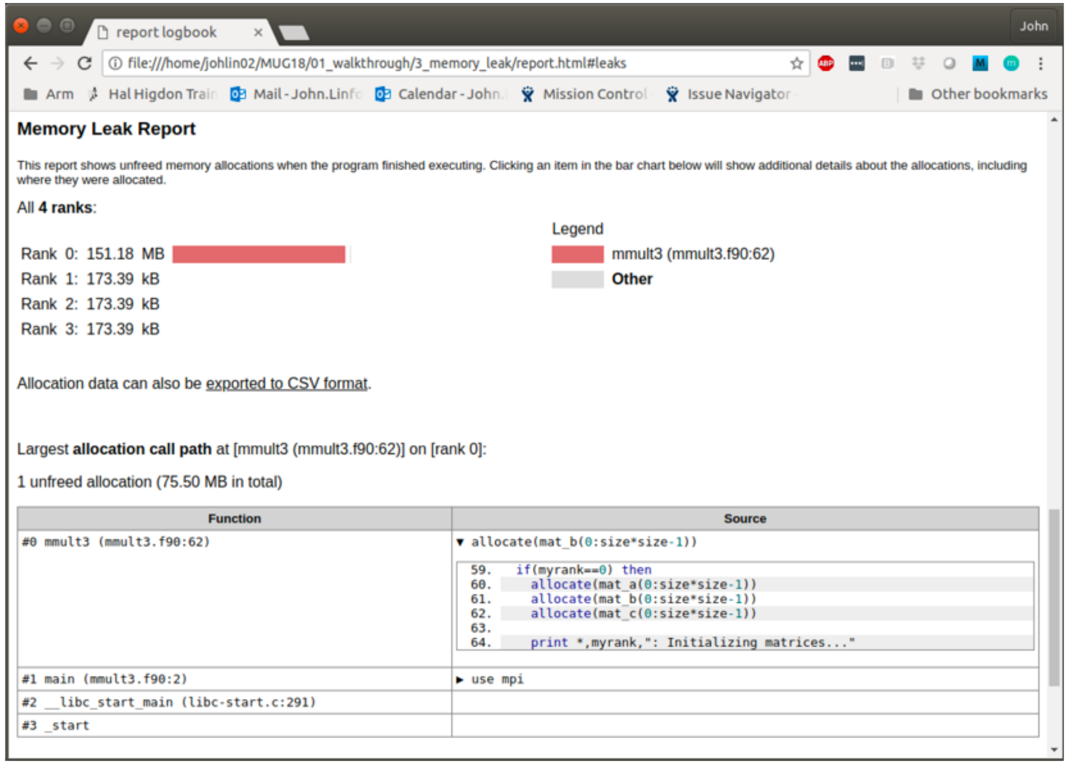
View the memory leak report to see unfreed allocations

Allocations that are not freed when the program exits *could* be leaks

Click allocation to see function source



Review source code to verify leak



arm

DDT can also track leaks via the GUI

To see the equivalent of a leak report

- "Current Memory Usage" in the GUI shows all current, unfreed allocations
- To see something like the offline leak report, stop the program just before exit
  - Enable Control -> Default Breakpoints -> Exit
  - Run program to "exit"
  - Open "Current Memory Usage"

Also...

- "View pointer details" allows you to see where pointers were allocated, freed, and whether they point to a valid memory location
- Memory tracking also works for GPU allocations made with cudaMalloc

arm

Another leak…

- Use either the GUI or a leak report to track down and fix the memory leak in the “memory-leak-mandel” exercise.

Profiling with MAP

…and Performance Reports

Profiling on Titan

Static binaries

- Need to link MAP libraries
- $ module load forge/18.2.2
- $ make-profile-libraries
- Generates libraries for your MPI and outputs instructions on how to link.

Dynamic binaries

- No need to link
- MAP will preload libraries into the binaries automatically
- We'll use this method today by adding `-dynamic` to the link line

arm

Improve performance

Efficient memory access

Fix inefficient memory access pattern

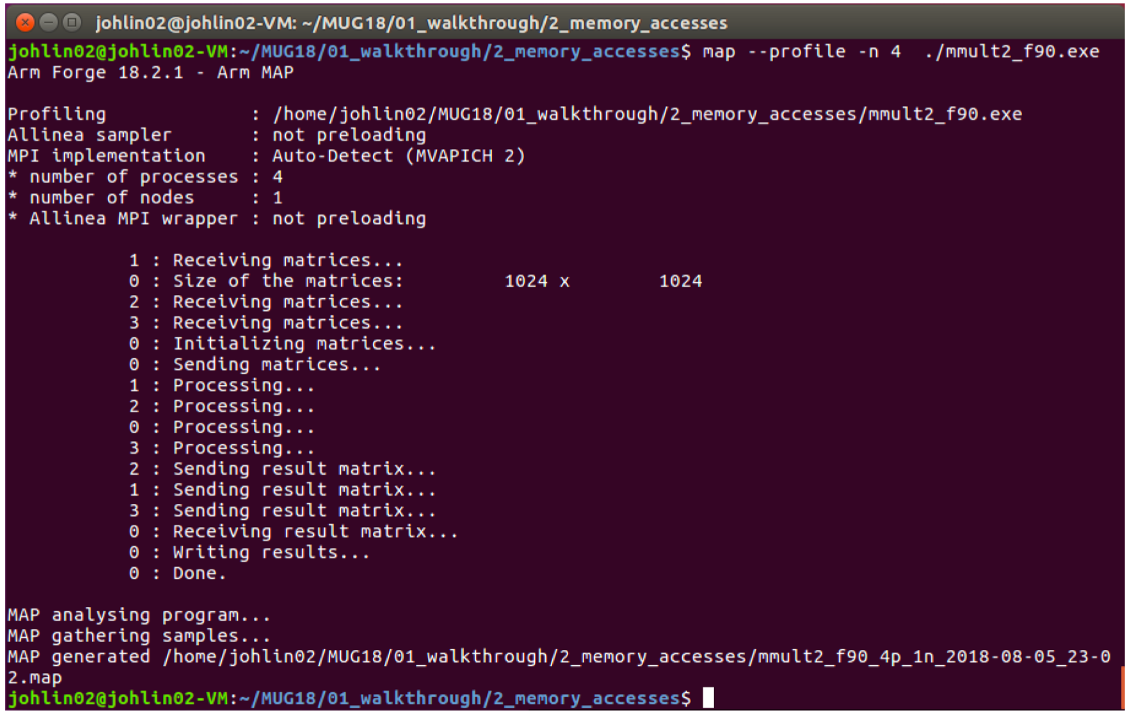
Revisiting the matrix multiply crash example

Exercise Outline

- **Objectives**
  - Discover Arm MAP's interface
  - Gather initial profiles of a MVAPICH2 application
- **Commands**

```
$ make
$ map --profile aprun -n 4 \
      ./mmult2_f90.exe
$ map mmult2_f90_4p*.map
# Observe profile
```

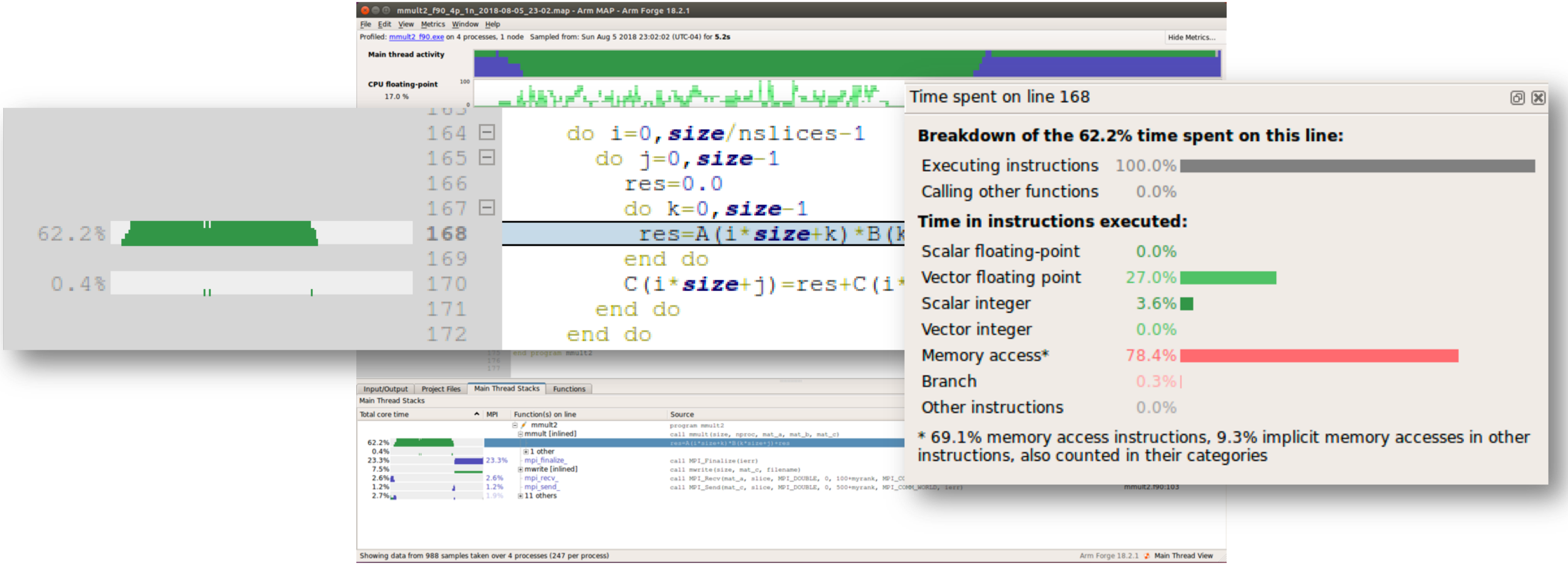
Initial Result: SLOW



arm

Initial profile

Find the hotspot: look for the line with the highest core time.



arm

Memory access patterns

- Data locality
  - Temporal locality: use of data within a short time of its last use
  - Spatial locality: use memory references close to memory already referenced

Temporal locality example

```
for (i=0 ; i < N; i++) {
    for (loop=0; loop < 10; loop++) {
        … = … x[i] …
    }
}
```

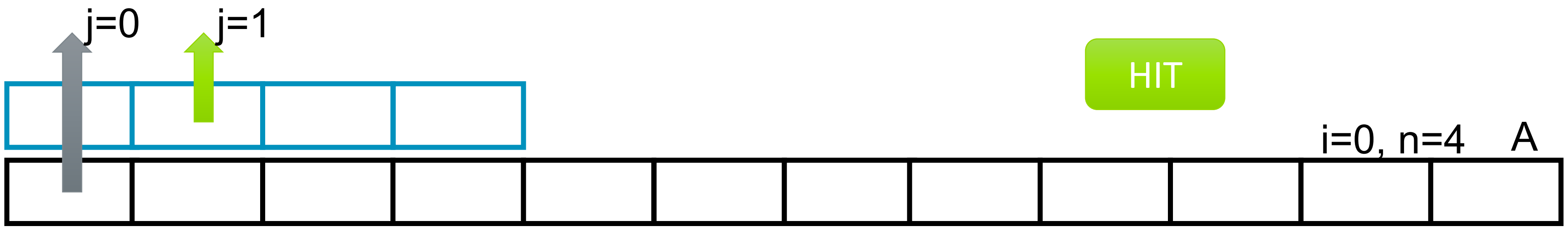
Spatial locality example

```
for (i=0 ; i < N*s; i+=s) {
    … = … x[i] …
}
```

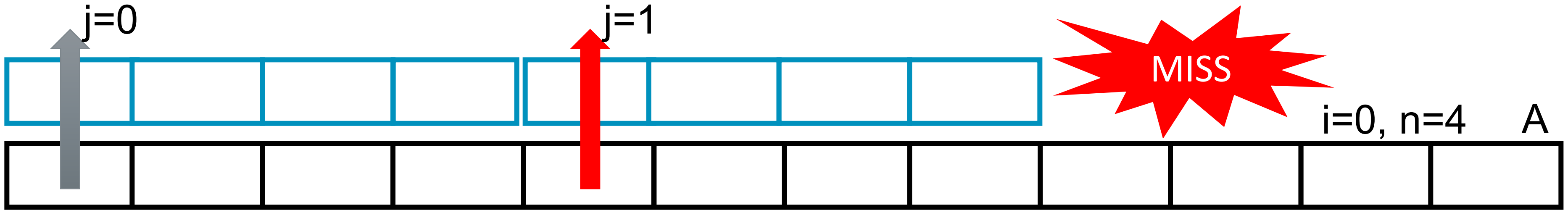

arm

Memory Accesses and Cache Misses

```
for(i=0; i<n; i++) {
  for(j=0; j<n; j++) {
    A[i*n+j]=…
  }
}
```




```
for(i=0; i<n; i++) {
  for(j=0; j<n; j++) {
    A[j*n+i]=…
  }
}
```



Answer: Transpose matrix and interchange loops

Transposing the matrix improves locality → performance

Before

```
      do i=0,size/nslices-1
        do j=0,size-1
          res=0.0
          do k=0,size-1
           res=A(i*size+k)*B(k*size+j)+res
          end do
          C(i*size+j)=res+C(i*size+j)
        end do
      end do
```

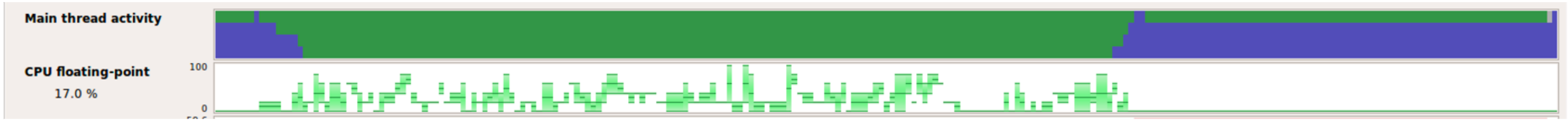
After

```
      do i=0,size/nslices-1
        do j=0,size-1
          res=0.0
          do k=0,size-1
           res=A(i*size+k)*transB(j*size+k)+res
          end do
          C(i*size+j)=res+C(i*size+j)
        end do
      end do
```

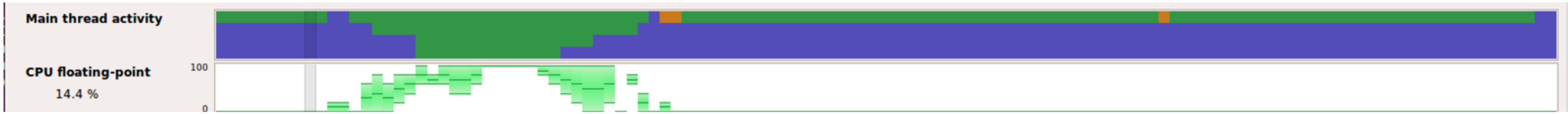

Final profile

About 3x faster

Before



After



arm

Debugging Imbalance

MPI I/O

arm

Can we improve I/O performance?

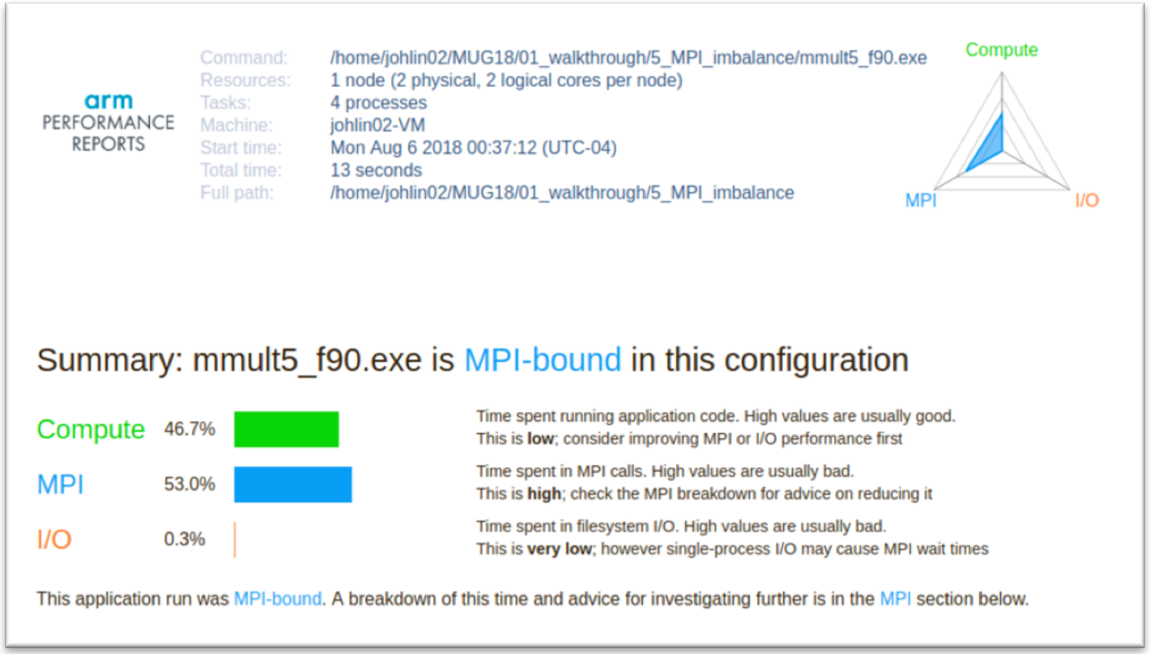
R0 responsible for all file I/O after R1+ return results. Surely we can do better?

Exercise Outline

- **Objectives**
  - Use MAP's I/O profiling features
  - Use performance reports to quantify speedup
- **Commands**

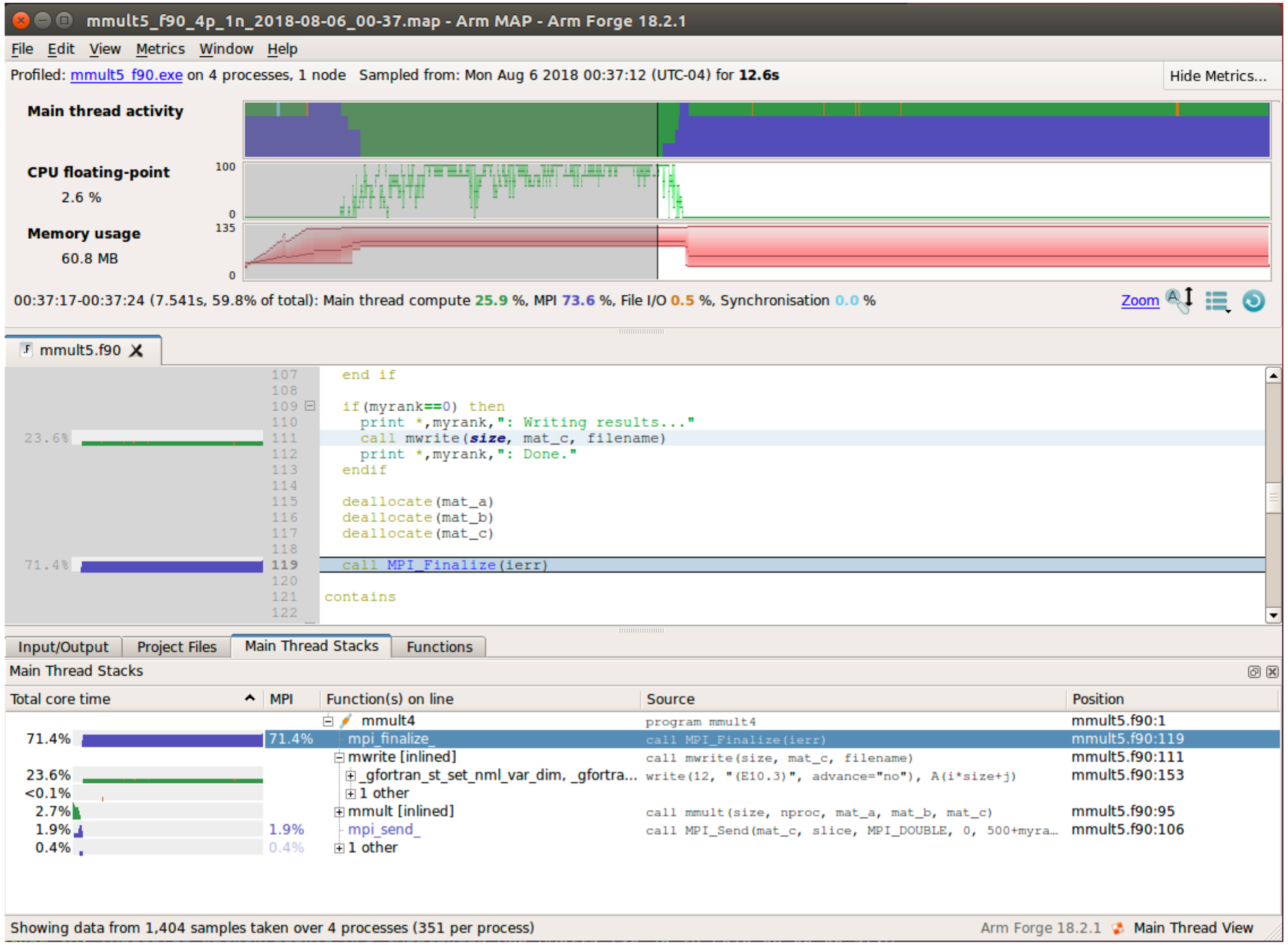
```
$ make
$ map --profile aprun -n 4 \
    ./mmult5_f90.exe
$ perf-report mmult5_f90_4p*.map
$ xdg-open mmult5_f90_4p*.html
```

Performance report shows MPI bound



Initial profile shows MPI_Finalize dominates

Time spent in MPI_Finalize is due to load imbalance in file I/O



Answer: improve scalability of I/O routines

Use MPI-IO to let all MPI ranks write their results to file simultaneously.

Before

```
if(myrank==0) then
  do i=1,nproc-1
    call MPI_Recv(mat_c(slice*i), slice, &
                  MPI_DOUBLE, &i, 500+i, &
                  MPI_COMM_WORLD, st, ierr)
  end do
else
  call MPI_Send(mat_c, slice, MPI_DOUBLE, &
                0, 500+myrank, &
                MPI_COMM_WORLD, ierr)
end if
if(myrank==0) then
  call mwrite(size, mat_c, filename)
endif
```

After

```
call MPI_FILE_OPEN(MPI_COMM_WORLD, &
        filename, &
        MPI_MODE_CREATE+MPI_MODE_WRONLY, &
        MPI_INFO_NULL, fh, ierr)
call MPI_FILE_SET_VIEW(fh, &
        0_MPI_OFFSET_KIND, MPI_DOUBLE, &
        MPI_DOUBLE, 'native', &
        MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE_AT(fh, disp, mat_c, &
        slice, MPI_DOUBLE, st, ierr)
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
call MPI_FILE_CLOSE(fh, ierr)
```

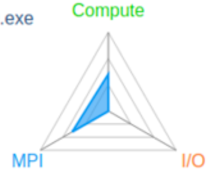
New approach: use MPI-IO for file output

Each MPI rank writes its results to it's own part of the output file

Before: runtime 13 seconds

arm PERFORMANCE REPORTS

Command:	/home/johlin02/MUG18/01_walkthrough/5_MPI_imbalance/mmult5_f90.exe
Resources:	1 node (2 physical, 2 logical cores per node)
Tasks:	4 processes
Machine:	johlin02-VM
Start time:	Mon Aug 6 2018 00:37:12 (UTC-04)
Total time:	13 seconds
Full path:	/home/johlin02/MUG18/01_walkthrough/5_MPI_imbalance



Summary: mmult5_f90.exe is MPI-bound in this configuration

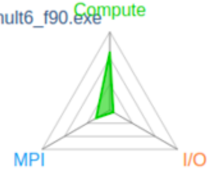
Compute	46.7%	Time spent running application code. High values are usually good. This is **low**; consider improving MPI or I/O performance first
MPI	53.0%	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	0.3%	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

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

After: runtime 5 seconds (2.6x speedup)

arm PERFORMANCE REPORTS

Command:	/home/johlin02/MUG18/01_walkthrough/5_MPI_imbalance/solution/mmult6_f90.exe
Resources:	1 node (2 physical, 2 logical cores per node)
Tasks:	4 processes
Machine:	johlin02-VM
Start time:	Mon Aug 6 2018 00:34:17 (UTC-04)
Total time:	5 seconds
Full path:	/home/johlin02/MUG18/01_walkthrough/5_MPI_imbalance/solution



Summary: mmult6_f90.exe is Compute-bound in this configuration

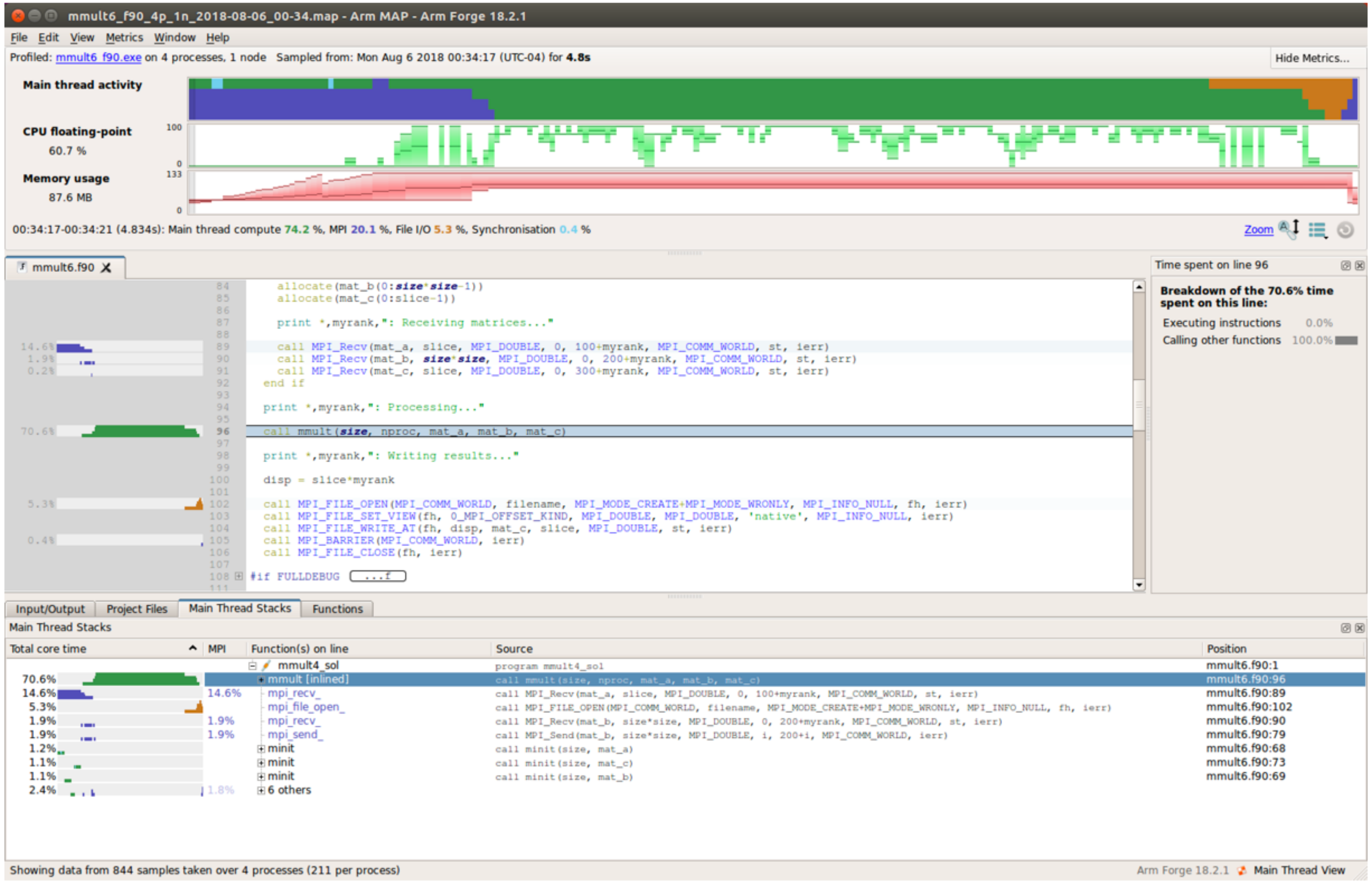
Compute	74.5%	Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for advice
MPI	20.1%	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	5.3%	Time spent in filesystem I/O. High values are usually bad. This is **low**; check the I/O breakdown section for optimization advice

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below. As little time is spent in MPI calls, this code may also benefit from running at larger scales.

arm

Final profile shows balanced I/O and compute dominates

New approach is about 3x faster



arm

GPU Debugging and Profiling

With DDT and MAP

GPU Debugging

- For many aspects, debugging on the GPU is very similar to debugging on the host
  - Adding breakpoints
  - Stepping through code
  - Inspecting variables, arrays, etc
  - Tracking memory
  - Memory error checking
- But there are important differences
  - Stepping will step the entire warp
  - Memory error checking is provided via cuda-memcheck

GPU Profiling

- Time spent waiting for accelerators
  - Determined by time spent in the CUDA (OpenACC, etc) API calls.
- GPU metrics - include:
  - Percentage of time spent in global memory accesses
  - GPU temperature
  - Power consumption
- CUPTI data
  - Which kernels were running and when
  - On-GPU profile data

Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

תודה

arm