arm
Cross-Platform Performance Engineering with Arm Allinea Studio
John C. Linford
28 May 2019

Outline

Slides [http://bit.ly/arm-alcf-may19]

- Performance Engineering Concepts
- Arm Forge (DDT and MAP)
- Arm Performance Reports
- Arm Compilers and Libraries
- Real World Case Studies
- Hands on kickoff

Hands-on

- **AWS Graviton cluster available for 24hrs!**
- Pick a student number 1 … 20
- Replace XX with your student number
- `ssh student0XX@108.128.237.67`
  - *Password:* `Tr@ining0XX`
- Grab a compute node (8 cores per node):
  - `srun -n 8 --pty $SHELL`
- Remember to zero-pad your student number to three places, e.g. "3" becomes "`003`"
- We strongly recommend you download and install the Arm Forge Remote Client.

© 2019 Arm Limited

OpenFOAM and ParaView across the Arm ecosystem

Cross-platform ecosystem and standards make this possible

Hewlett Packard Enterprise

OpenFOAM

ParaView

CAVIUM THUNDERX2™

CAVIUM

ubuntu

VNC

Windows 10

snapdragon by Qualcomm

hp

arm

© 2019 Arm Limited

Identifying and resolving performance issues

Identify Hotspots

Focus Optimization

Profile

-50x File I/O — Yes → Buffers, data formats, in-memory filesystems

No

-10x Communication — Yes → Collectives, blocking, non-blocking, topology, load balance

No

-5x Memory — Yes → Bandwidth/latency, cache utilization

No

-2x Compute — Yes → Vectors, branches, integer, floating point

No → Refine the Profile

© 2019 Arm Limited

Arm's solution for HPC application development and porting

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

=

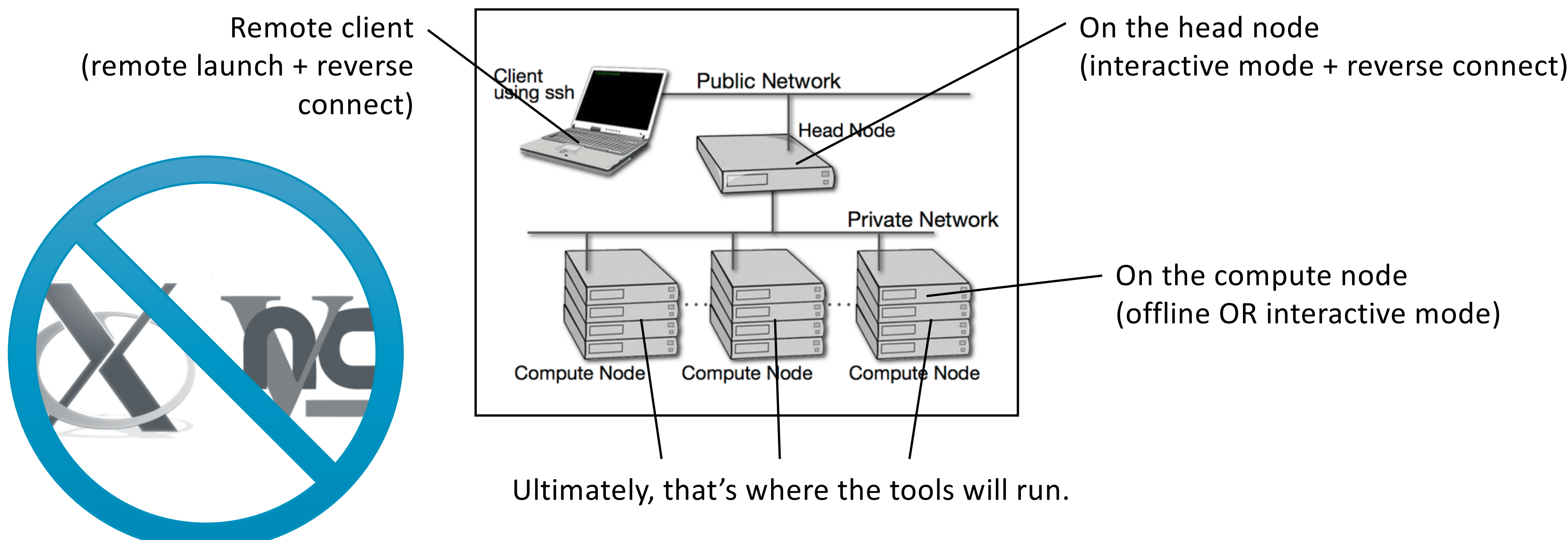
arm ALLINEA STUDIO

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

© 2019 Arm Limited

Use the Arm Forge Remote Client

https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/downloads/download-arm-forge



© 2019 Arm Limited

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

PING
759
ms

DOWNLOAD
0.42
Mbps

UPLOAD
0.34
Mbps

SHARE

Result ID 7229830690

Hibernia Networks
104.153.224.168

GO

Telstra
Perth
Change Server

© 2019 Arm Limited

Launching the Arm Forge Remote Client

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

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

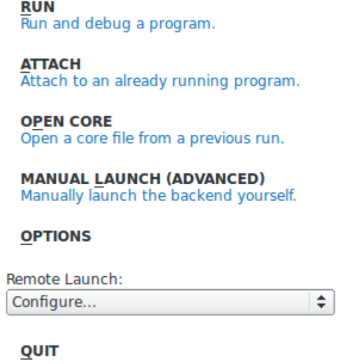
- https://developer.arm.com/products/software-development-tools/hpc/downloads/download-arm-forge

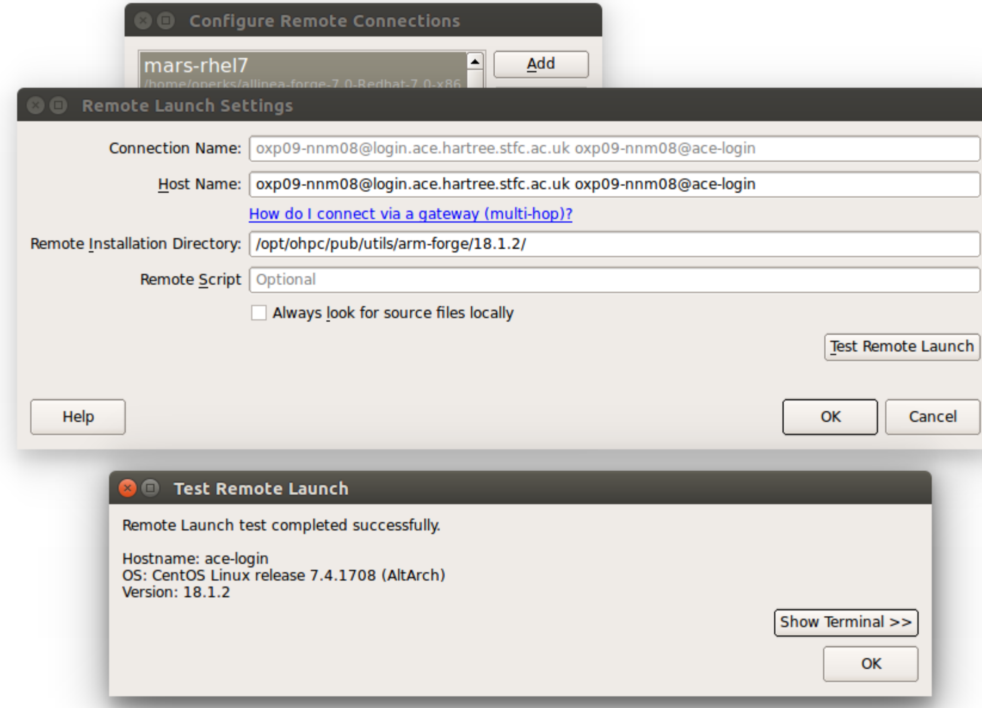
Connect to the cluster with the remote client

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

© 2019 Arm Limited

Remote connect





© 2019 Arm Limited

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

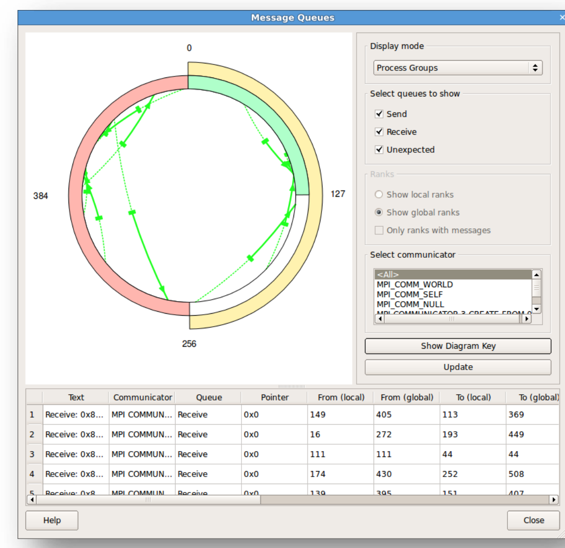
- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users

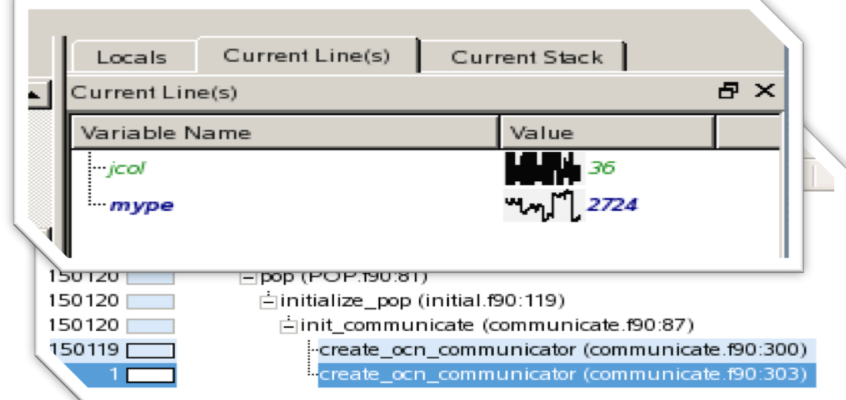
© 2019 Arm Limited

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





© 2019 Arm Limited

DDT: Feature Highlights

Switch between MPI ranks and OpenMP threads

Visualise data structures

Connect to continuous integration

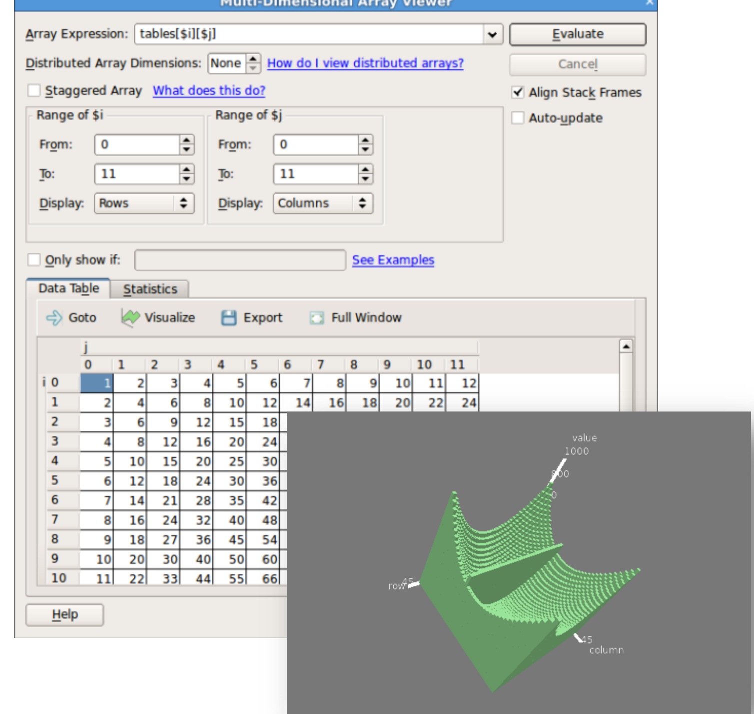
Display pending communications

© 2019 Arm Limited

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



© 2019 Arm Limited

Memory debugging menu in Arm DDT

Run

Run: mpirun -n 8 ./mmult2_c.exe Details

Command: mpirun -n 8 ./mmult2_c.exe

OpenMP Details

CUDA: Track allocations: enabled, Detect invalid accesses: disabled Details

Track GPU allocations (also enables CPU memory debugging)

Detect invalid accesses (memcheck)

Memory Debugging: Fast, 1 guard page after, Backtraces, Preload Details...

Plugins: none Details

Help Options Run Qu

Preload the memory debugging library Language: C++, threads

Note: Preloading only works for programs linked against shared libraries. If your program is statically linked, you must relink it against the dmalloc library manually.

Heap Debugging

Fast Balanced Thorough Custom

Enabled Checks: basic More Information

Heap Overflow/Underflow Detection

Add guard pages to detect out of bounds heap access

Guard pages: 1 Add guard pages: After

Advanced

Check heap consistency every 100 heap operations

Store stack backtraces for memory allocations

Only enable for these processes:

100% Select All x2 x0.5 1%

Help OK Cancel

When manual linking is used, untick “Preload” box

© 2019 Arm Limited

Arm DDT Feature Details

- Scalable debugging of threaded codes (with OpenMP or pthreads)
  - Support for asynchronous thread control
- Memory debugging: error detection, OOB detection (guard pages), leak detection
- Single or multiple Linux corefiles.
  - Core files are well supported on aarch64,
  - Can selectively dump core memory from specified processes or threads.
  - Standard core files as generated by all major Linux distributions. Lightweight core files not supported.
- Scalable launch via many vendor specific launch infrastructures, e.g. PMIx or MPIR

© 2019 Arm Limited

Application Debug Information

- DWARF 4 is fully supported on aarch64.
  - Arm and partners regularly verify debug information against GDB and Arm regression test suites.
- The GCC and Arm compilers produce debug information for applications that are compiled with at least the “-O -g” code optimization.
- Addition of architecture-specific flags like -march, -mcpu etc. further optimize the produced binaries to take full advantage of distinguishing features of the ThunderX2 CPU without interfering with debugging.
- The debugging information supports source context including program variables and stack traces. The runtime libraries of C/C++ and Fortran also retain key debugging information such as stack frame information.

 © 2019 Arm Limited

Arm DDT cheat sheet

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

- Load the environment module:
  - $ module load **forge**
- Prepare the code:
  - $ mpicc **-O0 -g** myapp.c -o myapp.exe
  - $ mpfort **-O0 -g** myapp.f -o myapp.exe
- Start DDT in interactive mode:
  - $ **ddt** mpirun -n 8 ./myapp.exe arg1 arg2 …
- Or use reverse connect:
  - On the login node:
    - $ ddt &
  - (or use the remote client)
  - Then, edit the job script to run the following command and submit:
    - **ddt --connect** mpirun -n 8 ./myapp.exe arg1 arg2 …

© 2019 Arm Limited

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** mpirun ./jacobi_omp_mpi_gnu.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**

```
ddt --offline -o jacobi_omp_mpi_gnu_debug.txt \
           --trace-at _jacobi.F90:83,residual \
           srun ./jacobi_omp_mpi_gnu.exe
```


© 2019 Arm Limited

DDT command line options

```
$ ddt --help
Arm Forge 18.2.1 - Arm DDT

Usage: ddt [OPTION...] [PROGRAM [PROGRAM_ARGS]]
       ddt [OPTION...] (mpirun|mpiexec|aprun|...) [MPI_ARGS] PROGRAM [PROGRAM_ARGS]

  --connect                                                Reverse Connect (launch as a server and wait)
  --attach=[host1:]pid1,[host2:]pid2... [PROGRAM]          attach to PROGRAM being run by list of host:pid
  --attach-mpi=MPI_PID [--subset=rank1,rank2,rank3,...] [PROGRAM]   attach to processes in an MPI program.
  --break-at=LOCATION[,START:EVERY:STOP] [if CONDITION]    set a breakpoint at LOCATION
  --trace-at=LOCATION[,START:EVERY:STOP],VAR1,VAR2,...     set a tracepoint at LOCATION
  --cuda                                                   enable CUDA
  --mem-debug[=(fast|balanced|thorough|off)]               configure memory debugging (defaults to fast)
  --mpiargs=ARGUMENTS                                      command line arguments to pass to mpirun
  -n, --np, --processes=NUMPROCS                           specify the number of MPI processes
  --nodes=NUMNODES                                         configure the number of nodes for MPI jobs
  --procs-per-node=PROCS                                   configure the number of processes per node
  --offline                                                run through program without user interaction
  -s, --silent                                             don't write unnecessary output to the command line
```


© 2019 Arm Limited

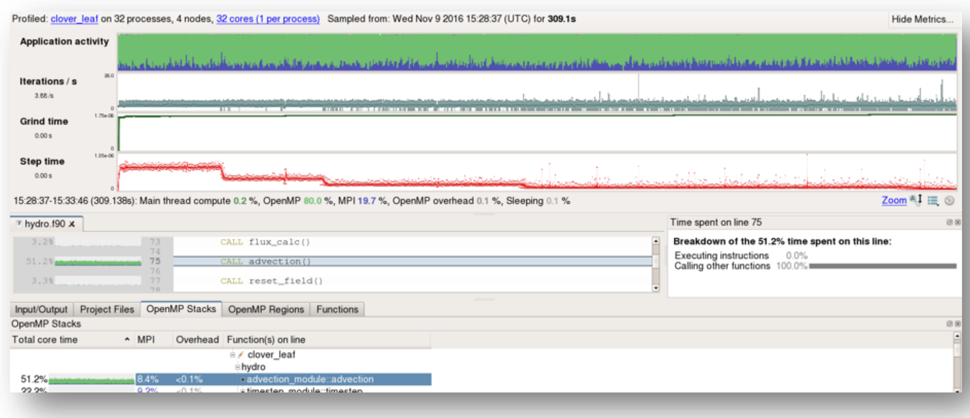
Arm MAP: Production-scale application profiling

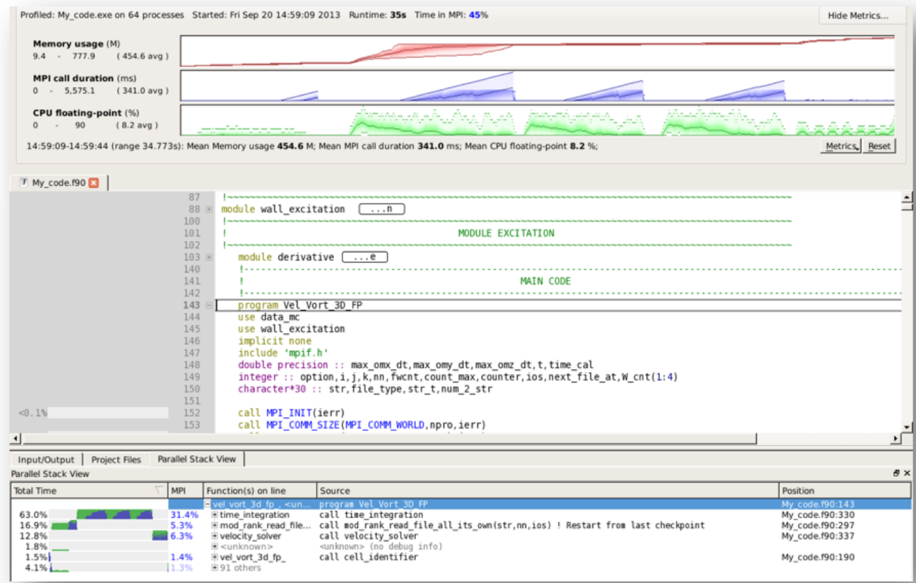
Identify bottlenecks and rewrite code for better performance

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

Examples:

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





© 2019 Arm Limited

Arm MAP Overview

A lightweight sampling-based profiler for large scale jobs

Core Features

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

Performance Metrics

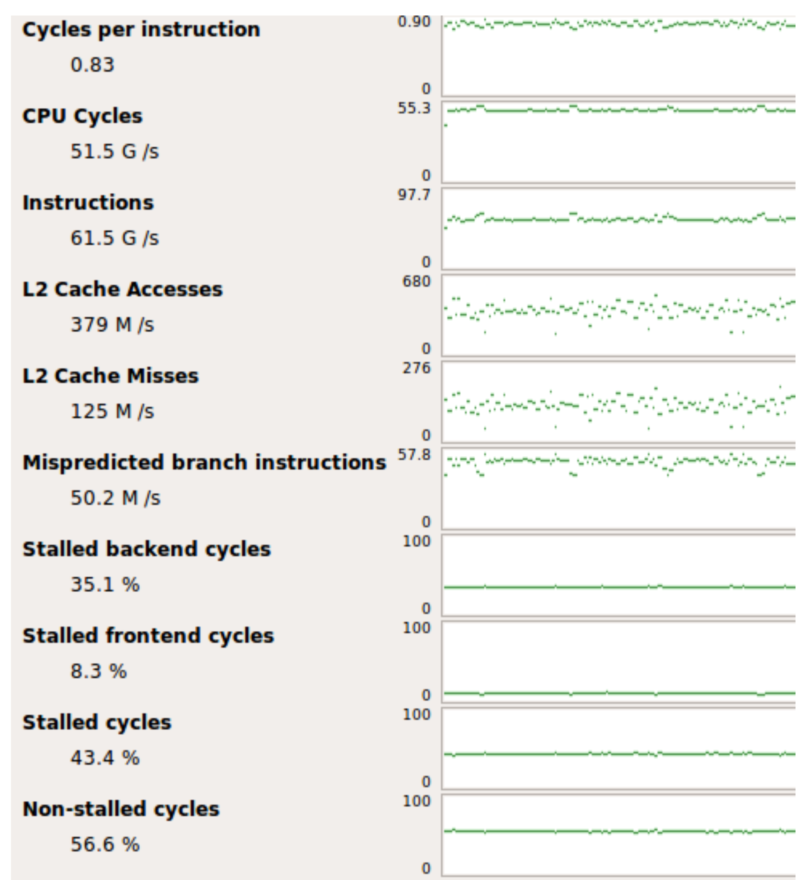
- Time classification
  - Based on call stacks
  - MPI, OpenMP, I/O, Synchronization
- Feature-specific metrics
  - MPI call and message rates
    - (P2P and collective bandwidth)
  - I/O data rates (POSIX or Lustre)
  - Energy data (IPMI or RAPL for Intel)
- Instruction information (hardware counters)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

© 2019 Arm Limited

Hardware Performance Metrics on Arm

MAP uses perf or PAPI to gather data.

- On x86 MAP reports on instruction mix
  - CPU, vectorization, memory, etc
  - Arm are researching ways to provide the same
- Instruction activity via perf
  - Harder to read / action
  - Raw rates presented – not interpolated
- Welcome your feedback to improve this



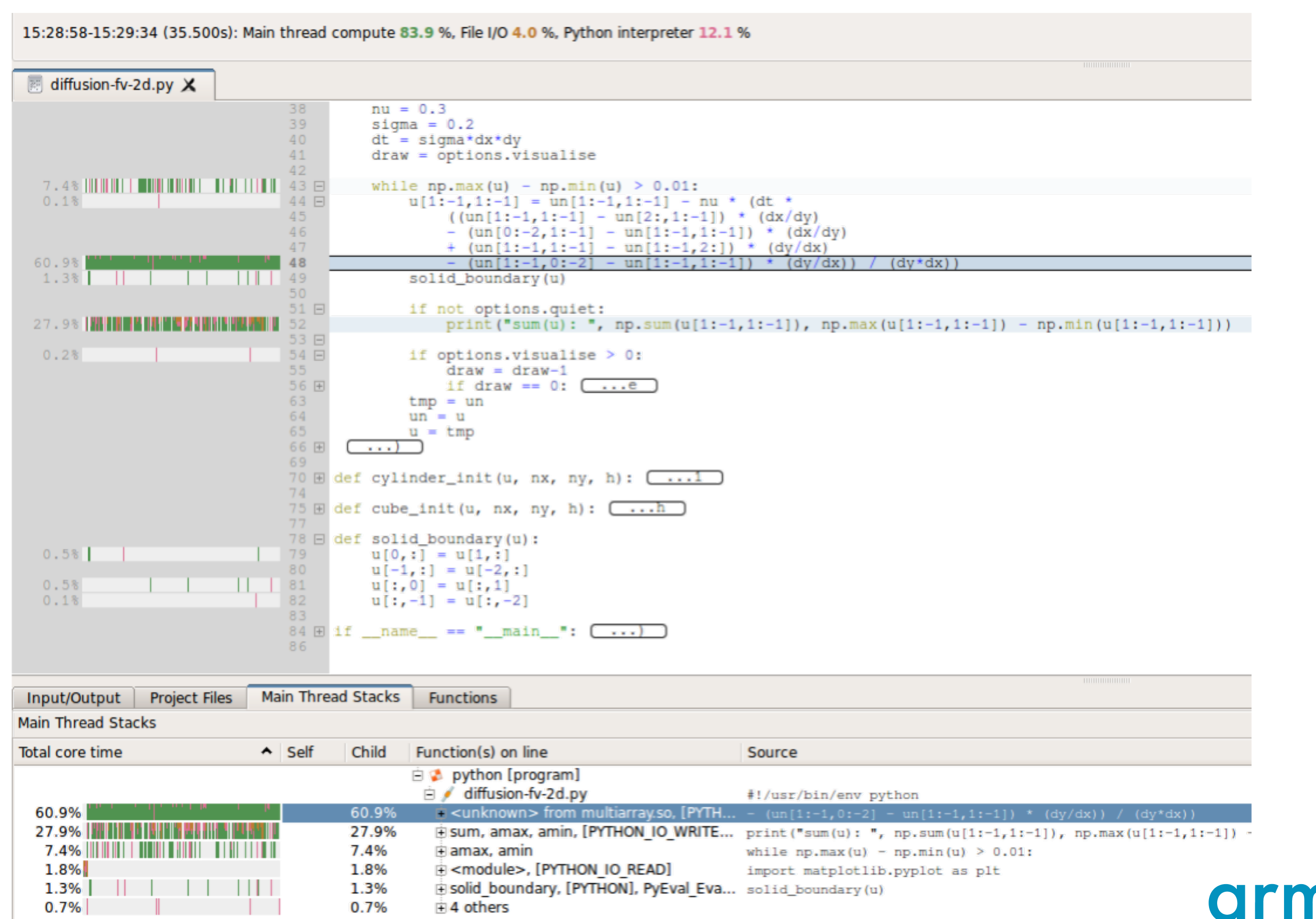
© 2019 Arm Limited

Python Profiling

From 19.1

New support for Python applications

- Native Python
- Cython Interpreter
- Called C/C++ code



© 2019 Arm Limited

arm

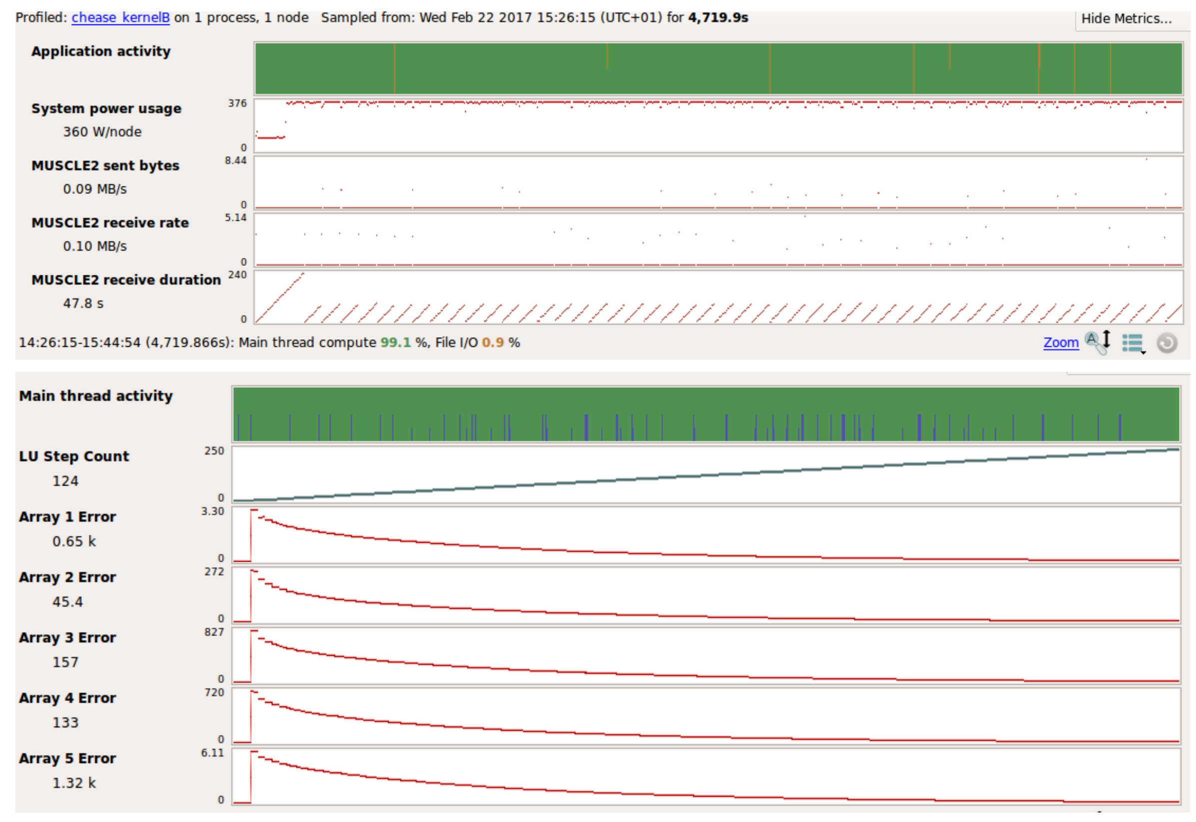
Custom metrics interface

- MAP supports the development of user metrics
- We provide a custom metric interface
  - API for safe calls to common functions
- Let's you develop your own metrics of interest
  - Link to application metrics (units / s, error values)
  - Link to libraries (specialist communication or I/O)
  - System metrics (custom energy monitors)
- Integrates directly into MAP and Performance Reports
  - XML files for aggregation methods
- Need to consider overheads and thread safety

© 2019 Arm Limited

Custom metric example: MUSCLE2 & LU error terms

https://github.com/arm-hpc/custom-metrics



- Customized application instrumentation, e.g. NPB LU
- Record error terms of solve
- Plot over time and step count for optimisation

© 2019 Arm Limited

Arm MAP cheat sheet

Generate profiles and view offline

- Load the environment module
  - $ module load **forge**

- Prepare the code
  - $ mpicc **-O -g** myapp.c -o myapp.exe
  - $ mpfort **-O -g** myapp.f -o myapp.exe

- Offline: edit the job script to run Arm MAP in “profile” mode
  - $ **map --profile** mpirun ./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)

© 2019 Arm Limited

MAP command line options

```
$ map --help
Arm Forge 18.2.1 - Arm MAP

Usage: map [OPTION...] [PROGRAM [PROGRAM_ARGS]]
       map [OPTION...] (mpirun|mpiexec|aprun|...) [MPI_ARGS] PROGRAM [PROGRAM_ARGS]
       map [OPTION...] [MAP_FILE]

  --connect                          Reverse Connect (launch as a server and wait for the GUI to connect)
  --cuda-kernel-analysis             Analysis of the CUDA kernel source code lines
  --list-metrics                     Display metrics IDs which can be explicitly enabled or disabled.
  --disable-metrics=METRICS          Explicitly disable metrics specified by their metric IDs.
  --enable-metrics=METRICS           Explicitly enable metrics specified by their metric IDs.
  --export=FILE.json                 Exports a specified .map file as JSON
  --export-functions=FILE            Export all the available columns in the functions view to a CSV file (use --profile)
  --select-ranks=RANKS               Select ranks to profile.
  --mpiargs=ARGUMENTS                command line arguments to pass to mpirun
  -n, --np, --processes=NUMPROCS     specify the number of MPI processes
  --nodes=NUMNODES                   configure the number of nodes for MPI jobs
  --procs-per-node=PROCS             configure the number of processes per node
  --profile                          run through program without user interaction
```

 © 2019 Arm Limited

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

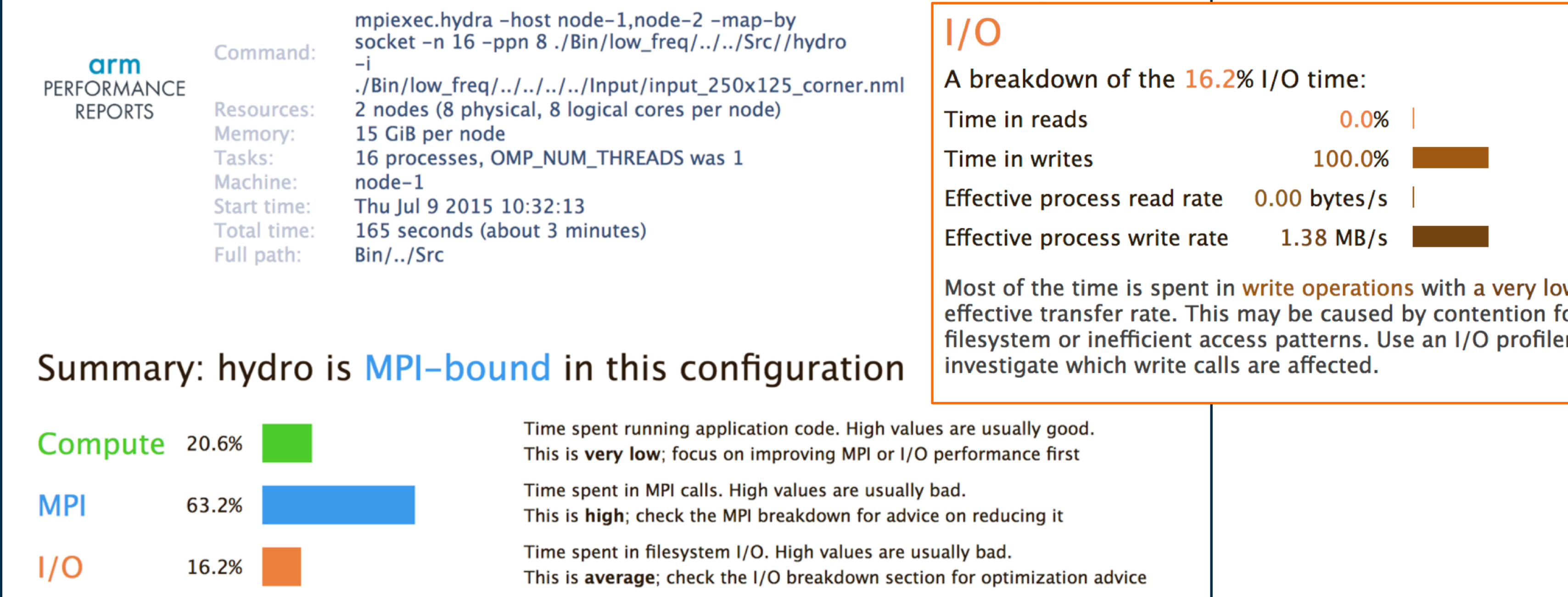
Adds value to typical users' workflows

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

© 2019 Arm Limited

Arm Performance Reports

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



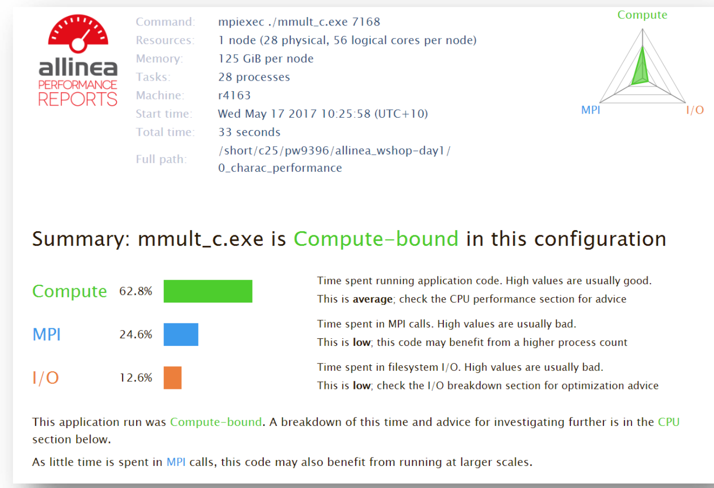
© 2019 Arm Limited

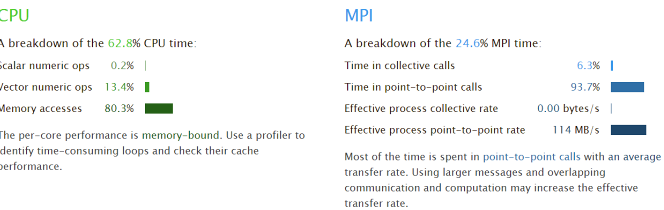
Understand application behaviour now

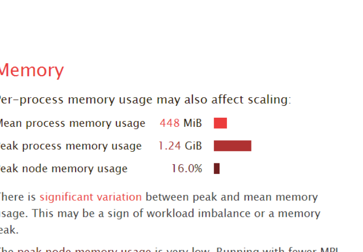
Set a reference for future work

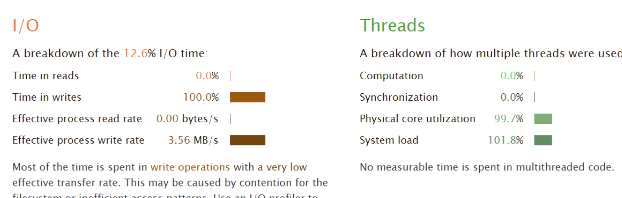
- Choose a representative test cases with known results
- Analyse performance on existing hardware (e.g. x86)
- with **Arm Performance Reports**
- Test scaling and note compiler flags

- <u>Example:</u>
- `$> perf-report mpirun –n 16 mmult.exe`





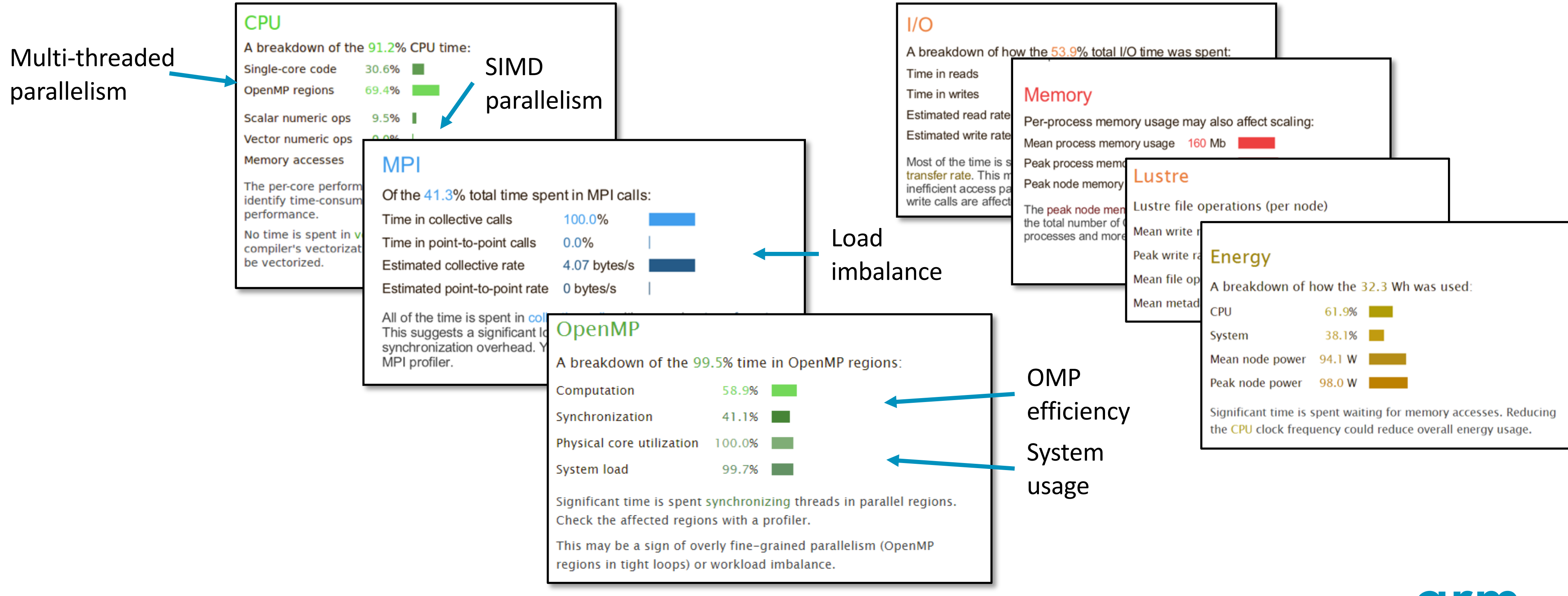




© 2019 Arm Limited

Arm Performance Reports Metrics

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



© 2019 Arm Limited
arm

Arm Performance Reports cheat sheet

Generate text and HTML reports from application runs or MAP files

- Load the environment module:
  - $ module load **reports**
- Run the application:
  - **perf-report** mpirun -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

© 2019 Arm Limited

Performance Reports command line options

```
$ perf-report --help
Arm Performance Reports 18.2.1 - Arm Performance Reports

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

  --list-metrics                      Display metrics IDs which can be explicitly enabled or disabled.
  --disable-metrics=METRICS           Explicitly disable metrics specified by their metric IDs.
  --enable-metrics=METRICS            Explicitly enable metrics specified by their metric IDs.
  --mpiargs=ARGUMENTS                 command line arguments to pass to mpirun
  --nodes=NUMNODES                    configure the number of nodes for MPI jobs
  -o, --output=FILE                   writes the Performance Report to FILE instead of an auto-generated name.
  -n, --np, --processes=NUMPROCS      specify the number of MPI processes
  --procs-per-node=PROCS              configure the number of processes per node for MPI jobs
  --select-ranks=RANKS                Select ranks to profile.
```


© 2019 Arm Limited

arm

Compute Optimization with Arm MAP

CCC and the ORNL GPU Hackathon @ Pawsey

Quantum collisions in atomic and molecular physics

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

© 2019 Arm Limited

Initial Profile

/home/igor/group/B3+/PRODUCTION/FINE_ENERGY_SCAN/402.2000000eV/cccd_profile_192p_8n_24t_2018-04-18_19-19.map - Arm MAP - Arm Forge 18.1.2 [Trial Version]

Profiled: cccd_profile on 192 processes, 8 nodes, 192 cores (1 per process) Sampled from: Wed Apr 18 2018 19:19:10 (UTC+08) for **1,453.0s**

Hide Metrics...

Application activity

CPU floating-point 100 12.8 % 0

Memory usage 23.0 0.71 GB 0

19:19:40-19:34:09 (868.918s, 59.8% of total): Main thread compute 0.5 %, OpenMP 88.4 %, MPI in OpenMP 0.5 %, MPI 0.0 %, File I/O 0.0 %, OpenMP overhead 10.5 %

Zoom

main.f | vmat.f | redistribute.f

```
668         save first_time, states, nchpmax
669
670         if(i_sw_ng .eq. 1) then
671
672            iparity = (-1)**(lg + ipar)
673            ncht = 0
674            nchp = 1
675
676            do N=1,Nstate_str_to_scat
677               la = latom(N)
678               lapar = lpar(N)
679               na = na_ng(N)
680
681   c     This is a horrible "hack", its purpose is to include "unnatural" parity states,
682   c     like positive parity P-state, into channel looping for lg=0 symmetry.
683   c     It produces additional channels for lg=0 that in principel should not be there
684   c     but they are coming out with zero V-matrix due to the parity and ang.mom. considerations anyway...
685               lapar = (-1)**la
686               if (lpar(nchp).ne.0.and.lg.ne.0) lapar = lpar(N)
687   c
```

Time spent on line 677

Breakdown of the 0.0% time spent on this line:

Executing instructions	100.0%
Calling other functions	0.0%

Time in instructions executed:

Scalar floating-point	0.0%
Vector floating point	0.0%
Scalar integer	0.0%
Vector integer	0.0%
Memory access	100.0%
Branch	0.0%
Other instructions	0.0%

Input/Output | Project Files | OpenMP Stacks | OpenMP Regions | Functions

Functions

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

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

Arm Forge 18.1.2 Connected to: igor@athena.pawsey.org.au OpenMP View

© 2019 Arm Limited

Load balancer is imbalanced?

- Before:
- `0 8 0 -10 199 329 492 1.21 13530 0 89 -1` **91%** LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
- `1 8 0 -7 591 573 872 1.97 45150 0 350 0` **80%** LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
- `2 8 0 -16 894 762 1153 2.28 77028 0 607 1` **86%** LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
- `3 8 0 -24 916 886 1331 2.05 99681 0 766 2` **91%** LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
-

© 2019 Arm Limited

Initial Profile

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

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

Surprise! Didn’t expect that.

© 2019 Arm Limited

Results and Final Profile

/home/igor/group/B3+/PRODUCTION/FINE_ENERGY_SCAN/402.4000000eV/cccd_profile_192p_8n_24t_2018-04-19_17-04.map - Arm MAP - Arm Forge 18.1.2 [Trial Version]

Profiled: cccd_profile on 192 processes, 8 nodes, 192 cores (1 per process) Sampled from: Thu Apr 19 2018 17:04:41 (UTC+08) for **986.0s**

Hide Metrics...

Application activity

CPU floating-point 20.8 %

Memory usage 0.79 GB

17:04:52-17:13:44 (531.448s, 53.9% of total): Main thread compute 0.7 %, OpenMP 83.4 %, MPI in OpenMP 0.3 %, MPI 0.0 %, File I/O 0.0 %, OpenMP overhead 15.2 %, Sleeping 0.0 %

Zoom

main.f vmat.f

```
679            na = na_ng(N)
680
681   c     This is a horrible "hack", its purpose is to include "unnatural" parity states,
682   c     like positive parity P-state, into channel looping for lg=0 symmetry.
683   c     It produces additional channels for lg=0 that in principel should not be there
684   c     but they are coming out with zero V-matrix due to the parity and ang.mom. considerations anyway...
685            lapar = (-1)**la
686            if (lpar(nchp).ne.0.and.lg.ne.0) lapar = lpar(N)
687   c
688            do lp = abs(lg-la), lg+la
689               if (((-1)**lp)*lapar.ne.iparity) cycle
690               ncht = ncht + 1
691               if (nch.eq.ncht) then ...
709
710            enddo
711         enddo
712   c      print*, 'Could not find this channel: nch=',nch,
713   c     >        ', lg,ipar:',lg,ipar
714         nch = 0
715         return
```

Time spent on line 688

Breakdown of the 0.0% time spent on this line:

Executing instructions	100.0%
Calling other functions	0.0%

Time in instructions executed:

Scalar floating-point	0.0%
Vector floating point	0.0%
Scalar integer	0.0%
Vector integer	0.0%
Memory access	100.0%
Branch	0.0%
Other instructions	0.0%

Input/Output | Project Files | OpenMP Stacks | OpenMP Regions | Functions

Functions

Self	Total	MPI	Child	Overhead	Function
7.3%	16.8%		9.6%		form
<0.1%	0.3%		0.3%		getchchar
5.2%	5.6%		0.4%		getchinfo
<0.1%	<0.1%				getchinfo
0.1%	1.4%		1.3%		getformout
1.3%	16.3%		15.0%		getformout2 [inlined]
2.7%	3.2%		0.5%		getformout3 [inlined]
<0.1%	<0.1%				getprod
<0.1%	<0.1%				jfirst1 [inlined]
<0.1%	<0.1%				lagpol8 [inlined]
<0.1%	0.3%	0.3%	0.3%		MAIN__ [OpenMP region 1]
0.3%	0.4%		0.1%		makechil
<0.1%	<0.1%		<0.1%		makegreen
<0.1%	<0.1%		<0.1%		makeps
1.4%	1.9%		0.5%		maketail

Showing data from 103,488 samples taken over 192 processes (539 per process)

Arm Forge 18.1.2 Connected to: igor@athena.pawsey.org.au OpenMP View

© 2019 Arm Limited

Results and Final Profile

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

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

© 2019 Arm Limited

Balanced load balancer

- Before:

```
    0    8    0 -10    199    329    492   1.21  13530      0     89  -1  91% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
    1    8    0  -7    591    573    872   1.97  45150      0    350   0  80% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
    2    8    0 -16    894    762   1153   2.28  77028      0    607   1  86% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
    3    8    0 -24    916    886   1331   2.05  99681      0    766   2  91% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
```

- After:

```
    0    8    0 -10    174    329    492   1.06  13530      0     85  -1  93% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
    1    8    0 -11    415    577    872   1.40  43956      0    340   0  97% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
    2    8    0 -11    616    757   1153   1.55  79003      0    592   1  97% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
    3    8    0 -12    667    874   1331   1.46 105111      0    734   2  96% LG,node,ipar,inc,vt,i1,i2,tperi,nch,naps,mt,prev LG,eff
```

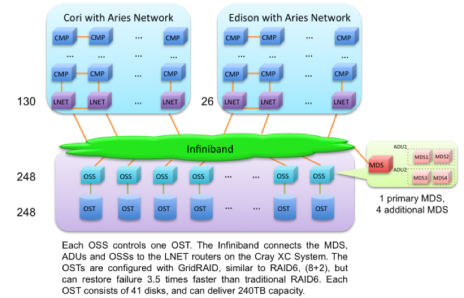

© 2019 Arm Limited

I/O Optimization with Arm MAP

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

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

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

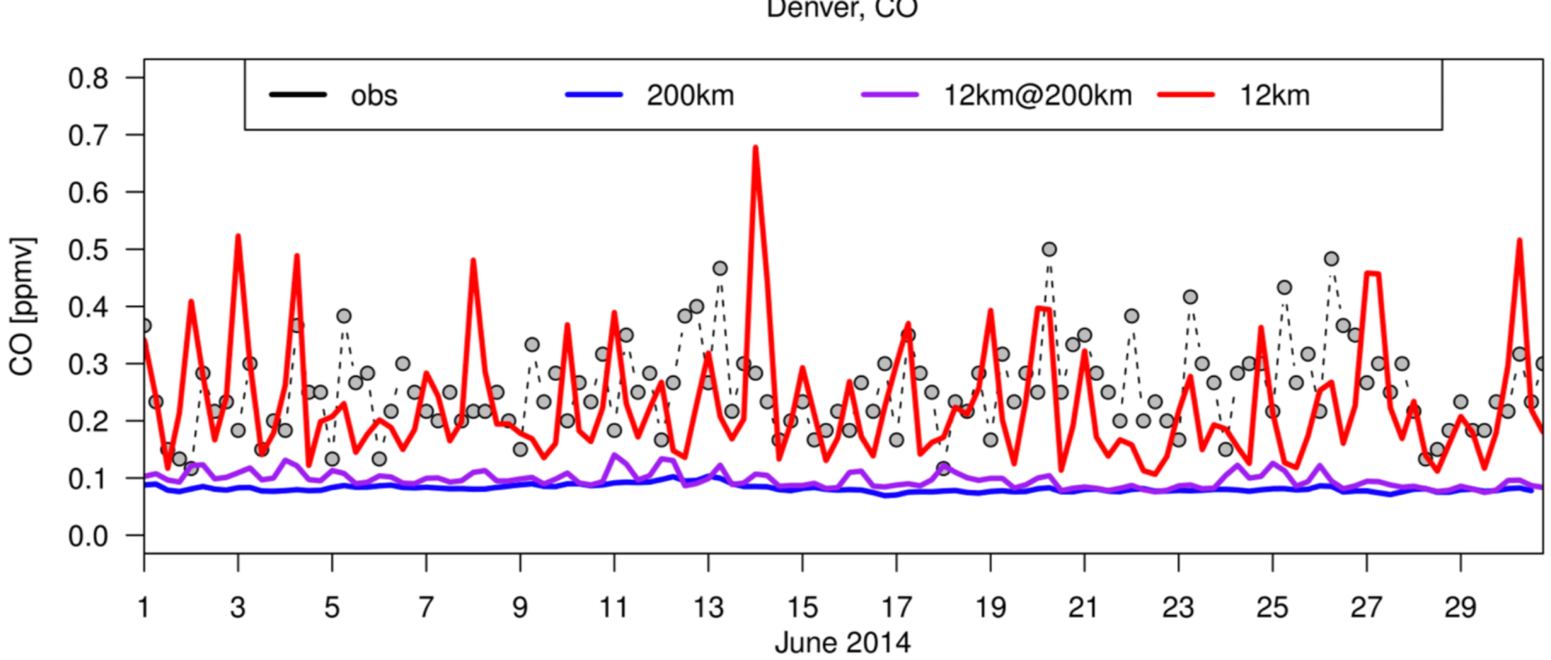


Each OSS controls one OST. The Infiniband connects the MDS, ADUs and OSSs to the LNET routers on the Cray XC System. The OSTs are configured with GridRAID, similar to RAID6, (8+2), but can restore failure 3.5 times faster than traditional RAID6. Each OST consists of 41 disks, and can deliver 240TB capacity.

© 2019 Arm Limited

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

Models require high resolutions to accurately describe physical conditions.



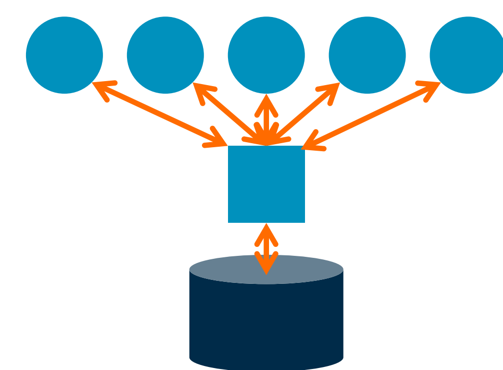
Credit: NASA GMAO, Christoph Keller.

© 2019 Arm Limited

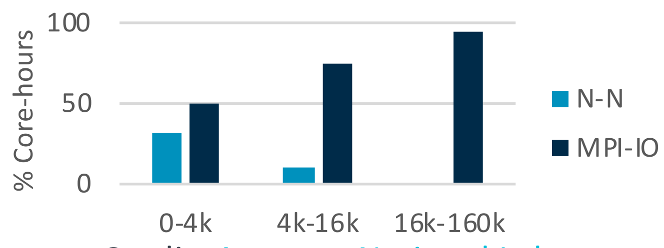
Simple approaches to parallel I/O

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

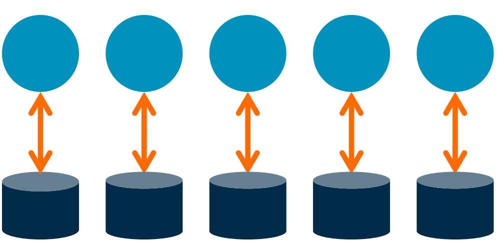
- 1 – 1: Master and workers
  - A master process performs I/O on behalf of many workers.
  - Collective operations (e.g. MPI_Gather, MPI_Scatter) move data to/from workers.
  - Performance bottleneck at the master.



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



Credit: Argonne National Lab

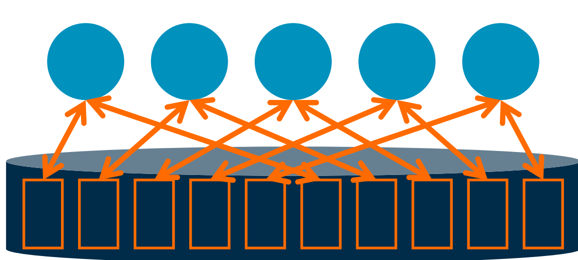


© 2019 Arm Limited

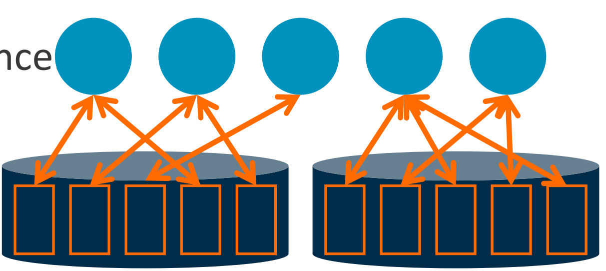
Treating parallel I/O like shared memory

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

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



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



© 2019 Arm Limited

Understand your I/O system

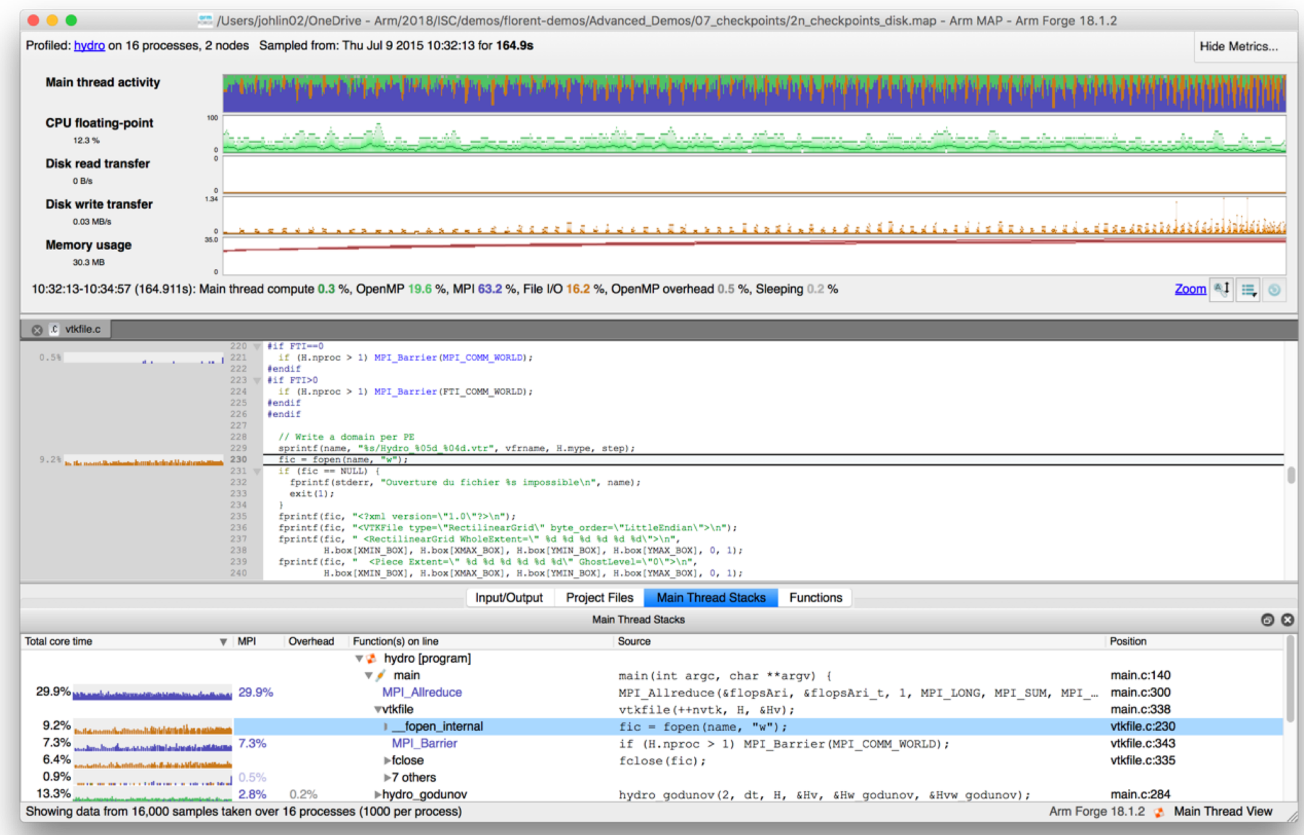
Use portable, cross-platform tools and libraries.

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

© 2019 Arm Limited

Initial profile shows 9.2% of runtime spent just opening files

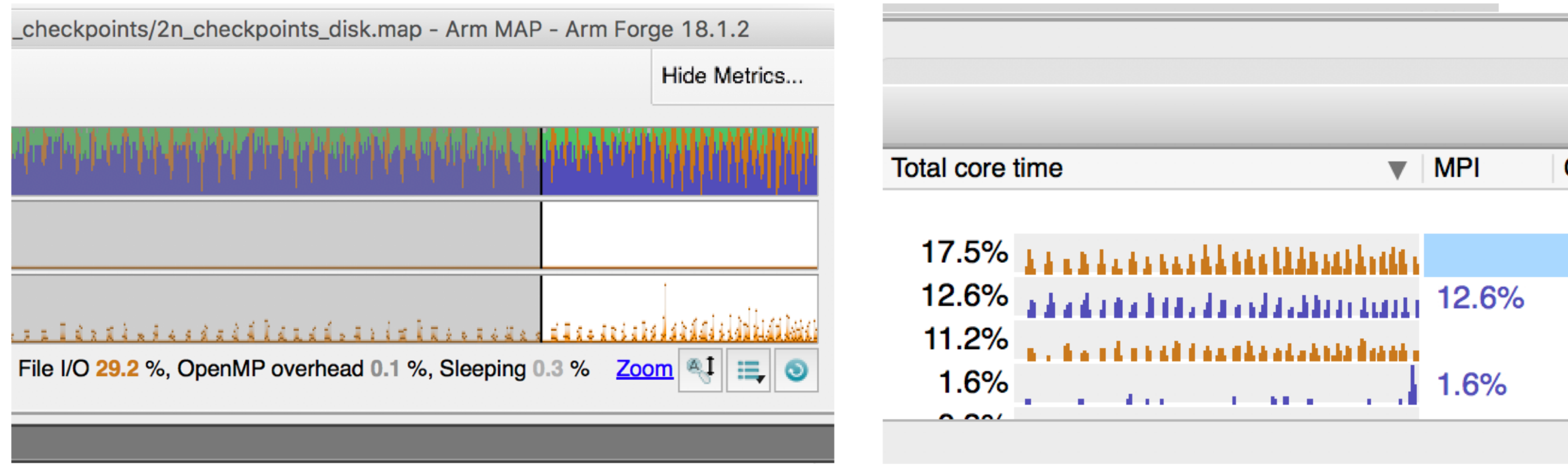
16.2% of runtime is I/O, but only 5% is spent in read/write operations.



© 2019 Arm Limited

Almost 30% of hotspot runtime is I/O

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

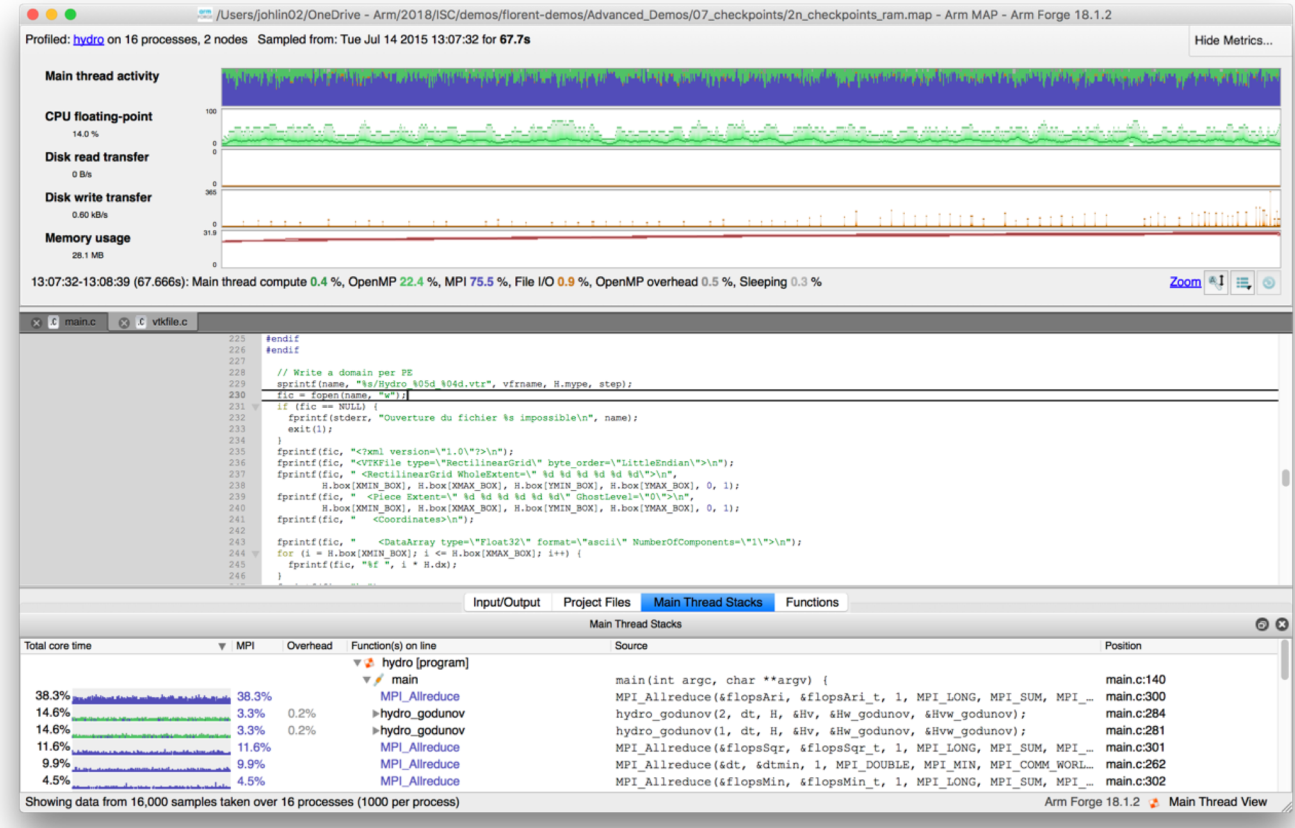


- Intermediate files for visualization are being written to disk.
- Fix: write intermediate files to an in-memory filesystem, e.g. /dev/shm.

© 2019 Arm Limited

Easy fix: write intermediate files to /dev/shm

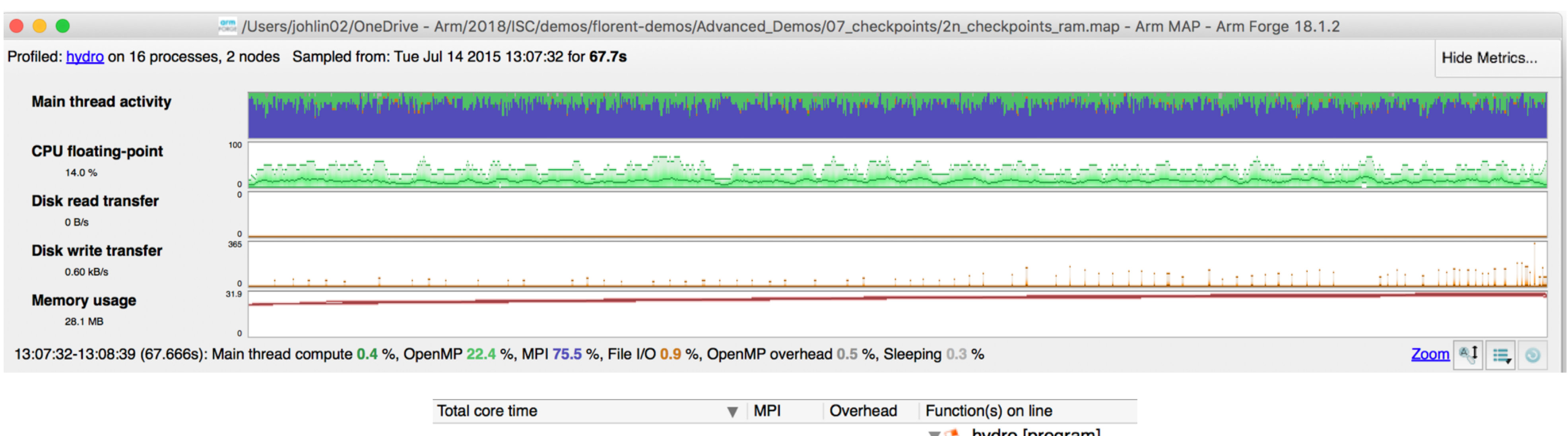
Writing temporary files to in-memory filesystem can dramatically improve performance.

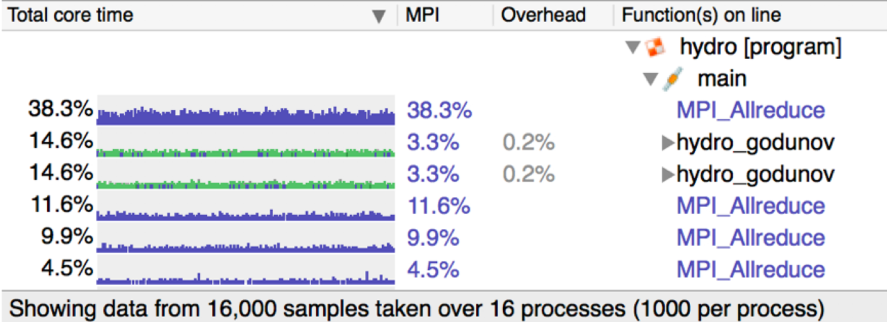


© 2019 Arm Limited

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

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





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

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

© 2019 Arm Limited

Arm Performance Reports

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

arm PERFORMANCE REPORTS

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

Summary: hydro is MPI-bound in this configuration

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

I/O

A breakdown of the 16.2% I/O time:

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

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

© 2019 Arm Limited

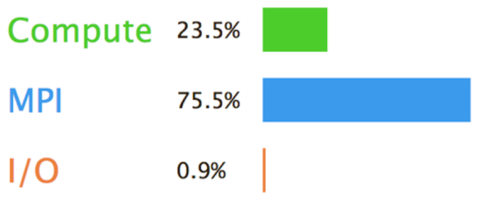
After the fix, write rate has improved 41.6x

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

arm PERFORMANCE REPORTS

Command:	mpiexec.hydra -host node-1,node-2 -map-by socket -n 16 -ppn 8 ./Bin/../Src//hydro -i ./Bin/../../../Input/input_250x125_corner.nml
Resources:	2 nodes (8 physical, 8 logical cores per node)
Memory:	15 GiB per node
Tasks:	16 processes, OMP_NUM_THREADS was 1
Machine:	node-1
Start time:	Tue Jul 14 2015 13:07:32
Total time:	68 seconds (about 1 minutes)
Full path:	Src

Summary: hydro is MPI-bound in this configuration



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

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

Time spent in filesystem I/O. High values are usually bad.
This is **very low**; however single-process I/O may cause MPI wait times

I/O

A breakdown of the 0.9% I/O time:

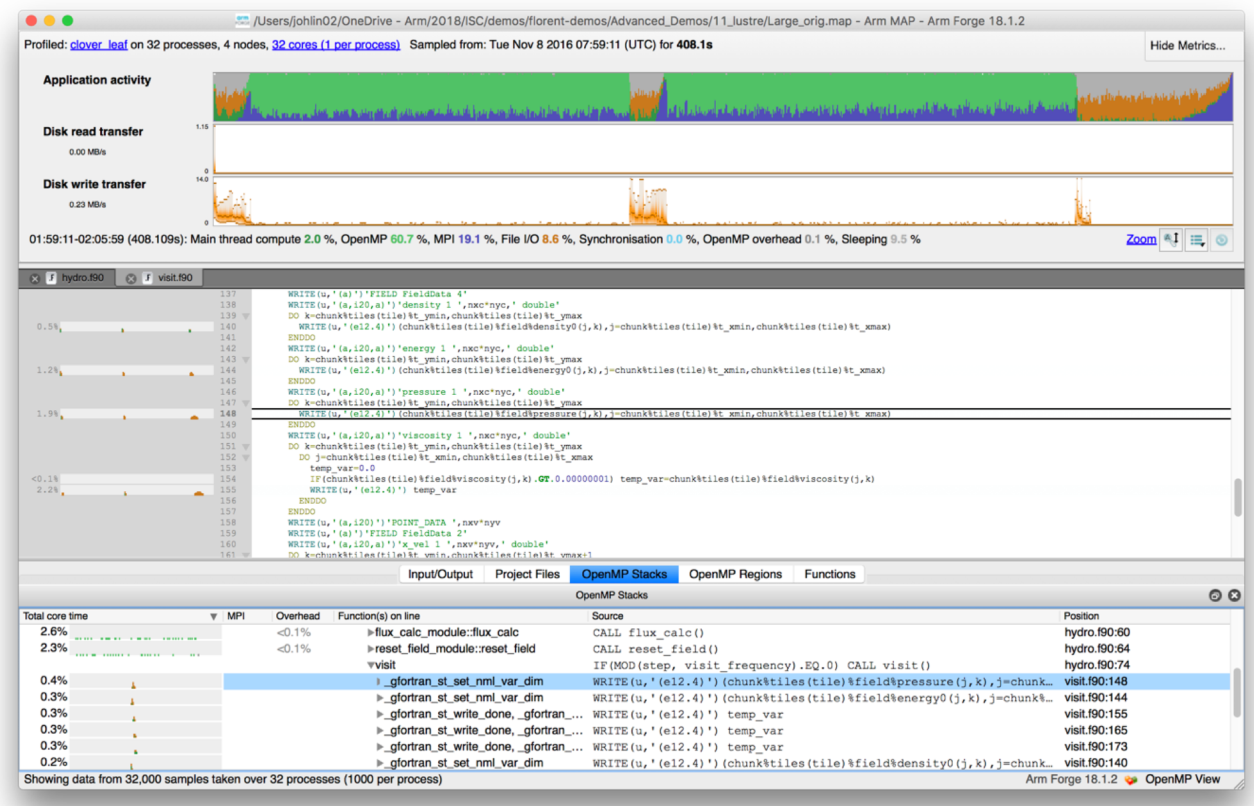
Time in reads	0.0%
Time in writes	100.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	57.5 MB/s

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

© 2019 Arm Limited

Initial profile of CloverLeaf shows surprisingly unequal I/O

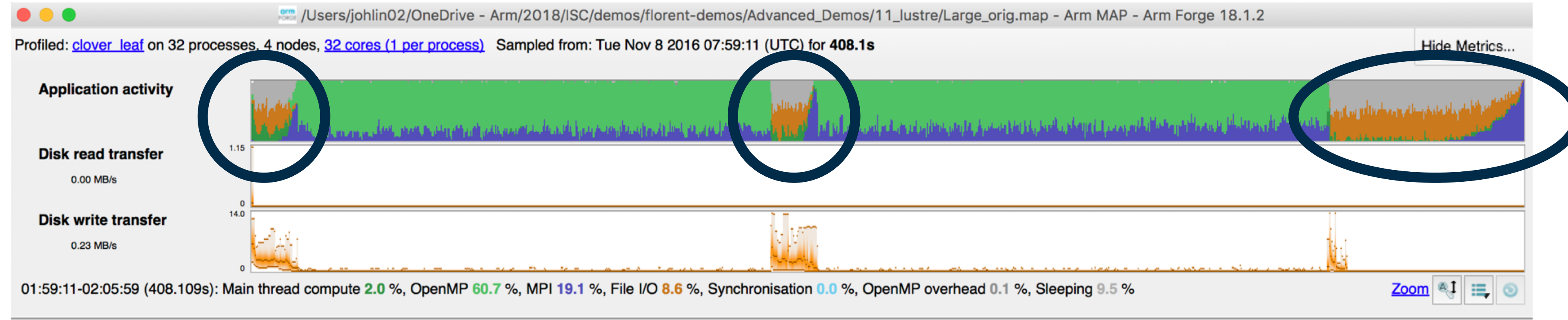
Each I/O operation should take about the same time, but it’s not the case.



© 2019 Arm Limited

Symptoms and causes of the I/O issues

Sub-optimal file format and surprise buffering.

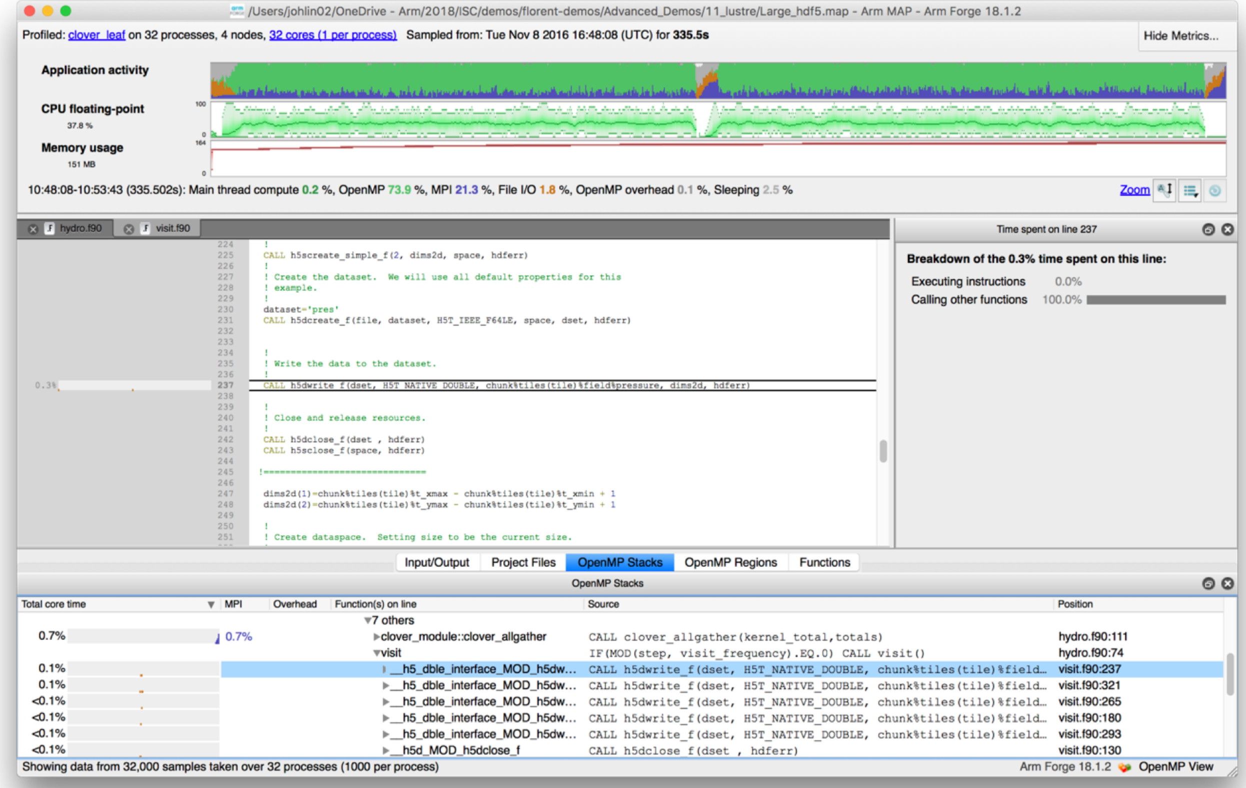


- Write rate is less than 14MB/s.
- Writing an ASCII output file.
- Writes not being flushed until buffer is full.
  - Some ranks have much less buffered data than others.
  - Ranks with small buffers wait in barrier for other ranks to finish flushing their buffers.

© 2019 Arm Limited

Solution: use HDF5 to write binary files

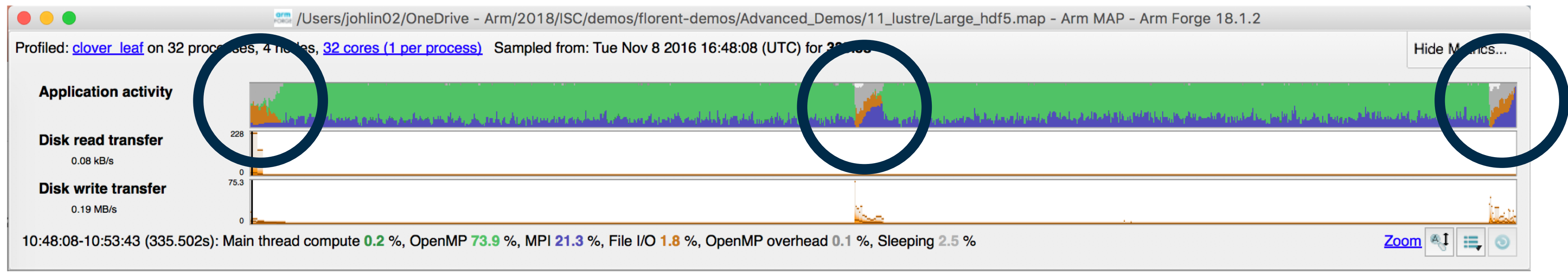
Using a library optimized for HPC I/O improves performance and portability.



© 2019 Arm Limited

Solution: use HDF5 to write binary files

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



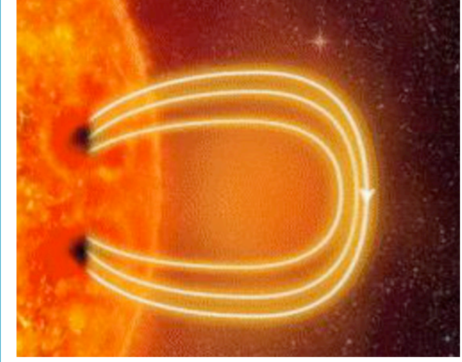
- Replace Fortran write statements with HDF5 library calls.
  - Binary format reduces write volume and can improve data precision.
  - Maximum transfer rate now 75.3 MB/s, over 5x faster.
- Note MPI costs (blue) in the I/O region, so room for improvement.

© 2019 Arm Limited

Advanced I/O investigation of Lustre on Archer

Simultaneously view system-level and application-level performance.

- Show data from Lustre client logs along with application data
- iPIC3D: kinetic simulation of plasma
  - Fully 3D implicit particle-in-cell (PIC)
  - C++ and MPI
  - Intermediate simulation results saved in VTK binary files, single file per quantity
  - Checkpointing done through HDF5 to individual files per process
  - Field values saved using collective MPI-IO to single file



© 2019 Arm Limited

Available performance data

Use MAP's ability to measure filesystem performance at the system and application levels

System level performance data

- Lustre logs: each read, write, or metadata operation recorded from each Lustre client.
- Aggregate I/O data for precise bandwidth figures for read/write at any moment in time.
- Max/min/mean bandwidth.
- Scheduler logs: application run start and end time and assigned nodes.

Application level performance data

- Approximate I/O bandwidth in a timeline.
- Approximate classification of I/O instructions (methods).
- In block-synchronous approach, it is possible to identify different I/O phases.

© 2019 Arm Limited

MAP aligns the system timeline with the application timeline

Lustre data is read from the lustre client's log files, while application data is read directly.

Profiled: iPIC3D on 1024 processes, 64 nodes Sampled from: Wed Feb 7 2018 20:02:56 (UTC) for **730.3s**

Main thread activity

Lustre read rate 1.57 MB/s (19.5 / 0)

Lustre write rate 0.01 GB/s (2.26 / 0)

Lustre metadata operations 1.38 k /s (16.8 / 0)

Lustre file opens 0.68 k /s (8.39 / 0)

20:02:56-20:15:06 (730.285s): Main thread compute 13.0 %, OpenMP 36.5 %, MPI 21.4 %, File I/O 29.1 %, OpenMP overhead 0.0 %, Sleeping 0.0 %

N-N file read shows spike in file open/read operations.

Checkpoint I/O corresponds to spike in Lustre write rate

© 2019 Arm Limited

We can focus on each I/O operation individually

Select a portion of the application timeline to view the source code performing I/O.

File Edit View Metrics Window Help

Profiled: iPIC3D on 1024 processes, 64 nodes Sampled from: Wed Feb 7 2018 20:02:56 (UTC) for **730.3s**

Main thread activity

Lustre read rate 1.78 MB/s (19.5 – 0)

Lustre write rate 0.00 GB/s (2.26 – 0)

Lustre metadata operations 1.61 k /s (16.8 – 0)

Lustre file opens 0.80 k /s (8.39 – 0)

20:05:26-20:05:41 (15.336s, 2.1% of total): Main thread compute 9.2 %, OpenMP 21.4 %, MPI 18.7 %, File I/O 50.7 %, OpenMP overhead 0.0 %, Sleeping 0.0 %

Input/Output | Project Files | Main Thread Stacks | Functions

Main Thread Stacks

Total core time	MPI	Overhead	Function(s) on line
			iPIC3D [program]
			main
			iPic3D::c_Solver::WriteOutput(int)
43.4%			WriteMomentsVTK(Grid3DCU*, EMfields3D*, Collective*, VCtopol...
7.4%			WriteFieldsVTK(Grid3DCU*, EMfields3D*, Collective*, VCtopology...
0.4%	0.2%		1 other
20.6%	3.1%		iPic3D::c_Solver::ParticlesMover()
15.5%	9.1%		iPic3D::c_Solver::CalculateField(int)
12.3%	6.0%		iPic3D::c_Solver::CalculateMoments()
0.3%	0.3%		2 others

 © 2019 Arm Limited

MAP’s timeline shows I/O overlapping with communication

We see elevated Lustre write rate when writing checkpoint restart files in HDF5.

File Edit View Metrics Window Help

Profiled: iPIC3D on 1024 processes, 64 nodes Sampled from: Wed Feb 7 2018 20:02:56 (UTC) for 730.3s

Main thread activity

Lustre read rate 1.04 MB/s (19.5 / 0)

Lustre write rate 0.05 GB/s (2.26 / 0)

Lustre metadata operations 0.96 k /s (16.8 / 0)

Lustre file opens 0.38 k /s (8.39 / 0)

Input/Output | Project Files | Main Thread Stacks | Functions

Main Thread Stacks

Total core time	MPI	Overhead	Function(s) on line
			iPIC3D [program]
			main
37.8%	30.7%		iPic3D::c_Solver::CalculateField(int)
28.3%	4.7%	<0.1%	iPic3D::c_Solver::ParticlesMover()
			iPic3D::c_Solver::WriteOutput(int)
17.5%			iPic3D::c_Solver::WriteRestart(int)
<0.1%			1 other
16.0%	4.7%	<0.1%	iPic3D::c_Solver::CalculateMoments(), __cray_memcpy_SNB
0.4%	0.2%		2 others

Time

ipic3d/iPIC3D.cpp
ipic3d/iPIC3D.cpp
ipic3d/iPIC3D.cpp
ipic3d/iPIC3D.cpp
ipic3d/main/iPIC3Dlib
ipic3d/iPIC3D.cpp

© 2019 Arm Limited

It’s possible to overlap different I/O approaches

HDF5 and VTK I/O operations occur at the same time on different ranks.

File Edit View Metrics Window Help

Profiled: iPIC3D on 1024 processes, 64 nodes Sampled from: Wed Feb 7 2018 20:02:56 (UTC) for 730.3s

Main thread activity

Lustre read rate 0.52 MB/s

Lustre write rate 0.04 GB/s

Lustre metadata operations 0.55 k /s

Lustre file opens

Input/Output | Project Files | Main Thread Stacks | Functions

Main Thread Stacks

Total core time	MPI	Overhead	Function(s) on line
			iPIC3D [program]
			main
			iPic3D::c_Solver::WriteOutput(int)
26.3%			WriteMomentsVTK(Grid3DCU*, EMfields3D*, Collective*, VCtopol...
25.8%			WriteFieldsVTK(Grid3DCU*, EMfields3D*, Collective*, VCtopology...
13.1%			iPic3D::c_Solver::WriteRestart(int)
0.4%	<0.1%		2 others
16.5%	2.1%	<0.1%	iPic3D::c_Solver::ParticlesMover()
10.4%	2.1%	<0.1%	iPic3D::c_Solver::CalculateMoments()
			iPic3D::c_Solver::CalculateField(int)
			EMfields3D::calculateE(int)
6.9%	3.0%		GMRES(void (EMfields3D::*)(double*, double*), double*, int, d...

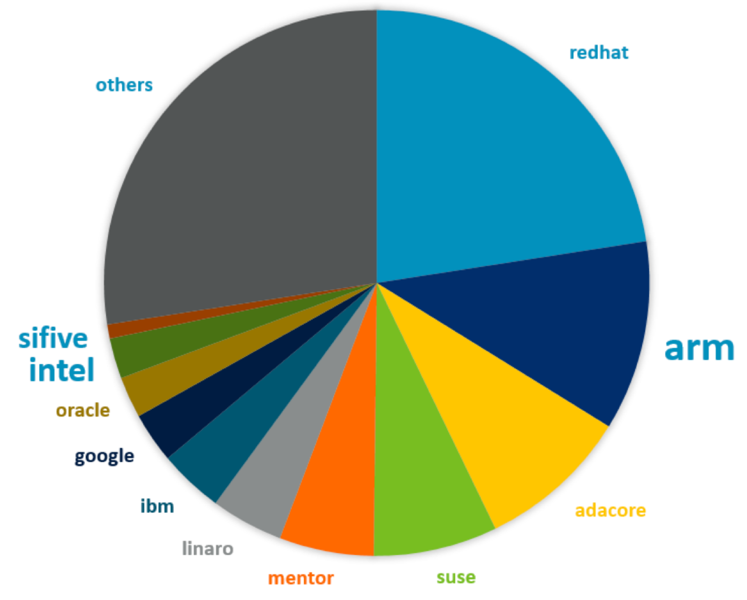
© 2019 Arm Limited

Arm Compilers for HPC

GCC is a first-class compiler in the Arm ecosystem

Arm the second largest contributor to the GCC project

- On Arm, GCC is a first class compiler alongside commercial compilers.
- GCC ships with Arm Compiler for HPC.
- Use GCC 7 or later! GCC 8+ preferred.



© 2019 Arm Limited

arm COMPILER

Arm's commercially-supported C/C++/Fortran compiler

Compilers tuned for Scientific Computing and HPC

Latest features and performance optimizations

Commercially supported by Arm

Tuned for Scientific Computing, HPC and Enterprise workloads

- Processor-specific optimizations for various server-class platforms
- Optimal shared-memory parallelism via Arm's optimized OpenMP runtime

Linux user-space compiler with latest features

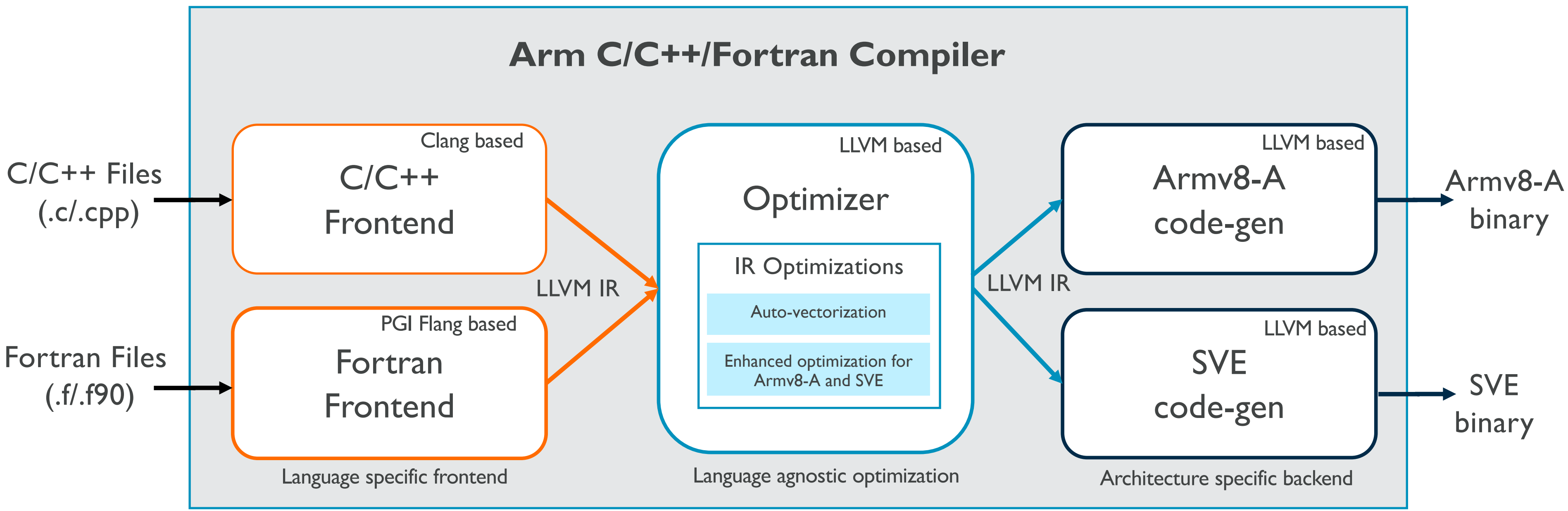
- C++ 14 and Fortran 2003 language support with OpenMP 4.5
- Support for Armv8-A and SVE architecture extension
- Based on LLVM and Flang, leading open-source compiler projects

Commercially supported by Arm

- Available for a wide range of Arm-based platforms running leading Linux distributions – RedHat, SUSE and Ubuntu

© 2019 Arm Limited

Arm Compiler is built on LLVM, Clang and Flang



© 2019 Arm Limited

Arm Compiler for HPC: Back-end

LLVM7

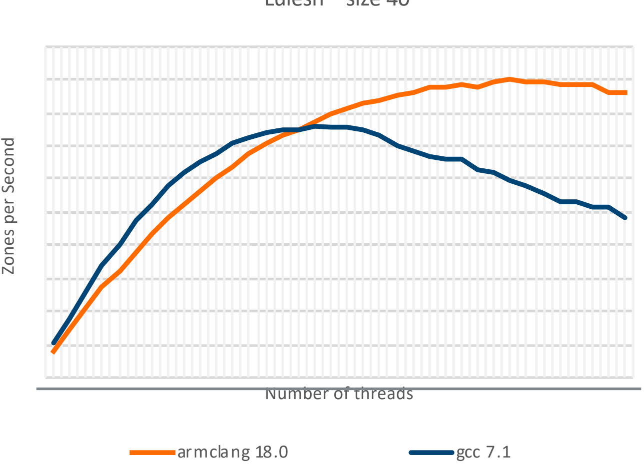
- Arm pulls all relevant cost models and optimizations into the downstream codebase.
  - Marvell have committed to upstreaming the cost models for future cores to LLVM.
- Auto-vectorization via LLVM **vectorizers:**
  - Use cost models to drive decisions about what code blocks can and/or should be vectorized.
  - As of October 2018, two different vectorizers from LLVM: Loop Vectorizer and SLP Vectorizer.
- Loop Vectorizer support for NEON (ThunderX2) and SVE:
  - Loops with unknown trip count
  - Runtime checks of pointers
  - Reductions
  - Inductions
  - "If" conversion
  - Pointer induction variables
  - Reverse iterators
  - Scatter / gather
  - Vectorization of mixed types
  - Global structures alias analysis

© 2019 Arm Limited

Arm's Optimized OpenMP Runtime

Arm actively optimizes OpenMP runtime libraries for high thread counts

- Large System Extension (LSE) atomic update instructions
- Atomics dramatically reduce runtime overhead, especially at high thread counts.
  - Used extensively in the OpenMP runtime shipped with the Arm HPC Compiler.
  - Also available in GNU's runtime.
- Synchronization constructs optimized for high thread counts.
  - Designed with hundreds of threads in mind.
  - Uses hardware features whenever available.



© 2019 Arm Limited

Compile and link your application on Arm

Application porting is a boring, immediate task

- Modify the Makefile/installation scripts to ensure compilation for aarch64 happens
- Compile the code with the **Arm Compiler for HPC**
- Link the code with the **Arm Performance Libraries**

- Examples:

```
$> armclang -c –I/path/armpl/include example.c -o example.o
$> armclang example.o -L/path/armpl/lib -larmpl_lp64 -o example.exe -lflang -lflangrti -lm
```

Arm Compiler for HPC	GNU Compiler
armclang	gcc
armclang++	g++
armflang	gfortran

© 2019 Arm Limited

arm PERFORMANCE LIBRARIES

Optimized BLAS, LAPACK and FFT

Commercially supported by Arm

Best in class performance

Validated with NAG test suite

Commercial 64-bit Armv8-A math libraries

- Commonly used low-level math routines - BLAS, LAPACK and FFT
- Provides FFTW compatible interface for FFT routines
- Batched BLAS support

Best-in-class serial and parallel performance

- Generic Armv8-A optimizations by Arm
- Tuning for specific platforms like Cavium ThunderX2 in collaboration with silicon vendors

Validated and supported by Arm

- Available for a wide range of server-class Arm-based platforms
- Validated with NAG's test suite, a de-facto standard

© 2019 Arm Limited

Validated with NAG's test suite

NAG, the Numerical Algorithms Group are a company from Oxford, UK, specialising in developing mathematical routines

They have been around for almost 50 years and have been involved with almost all vendor maths libraries

They provided us with their validation test suite

- This enables us to test every build of the library to ensure that all changes we make still provide numerical accuracy to the end-user

NAG are also under contract with us to provide support if we discover any issues with code they have supplied

They also provide us updated code-drops when new versions of the base libraries are released

© 2019 Arm Limited

Commonly used low-level math routines

The libraries we include are known as BLAS, LAPACK and FFT

Most routines come in a four varieties (where appropriate)

- Single precision real : Routines prefixed by ‘S’
- Double precision real : Routines prefixed by ‘D’
- Single precision complex : Routines prefixed by ‘C’
- Double precision complex : Routines prefixed by ‘Z’

- The rest of the name (normally) describes something about what the routine does
  - E.g. the matrix-matrix multiplication routine DGEMM is a
    - D – Double precision
    - GE – Matricies given in GEneral format
    - MM – Matrix-Matrix multiplication is performed

© 2019 Arm Limited

BLAS

BLAS, the Basic Linear Algebra Subroutines, is a standard API

- It is provided on all systems, used by a wealth of scientific codes for vector and matrix maths
- It was designed for Fortran, but is callable from all languages

These routines are come in three levels

- BLAS level 1 – vector-vector operations, e.g. DCOPY, DAXPY, DDOT
- BLAS level 2 – matrix-vector operations, e.g. DGEMV, DTRMV, DGER
- BLAS level 3 – matrix-matrix operations, e.g. DGEMM, DTRMM, DTRSM

42 BLAS routines in total

Providing incredibly high performing versions of these routines is the team’s main work

© 2019 Arm Limited

LAPACK

LAPACK, the Linear Algebra Package, is a another standard API

- It is provided on all systems, used by a wealth of scientific codes for solving equation systems
- It was designed for Fortran, but is callable from all languages
- We currently support LAPACK 3.7.0

The routines in LAPACK are normally build on BLAS routines so work we do on BLAS routines increases performance of particular LAPACK routines, too

There are now around 1700 LAPACK routines

- Most we do not touch, just using the reference version from Netlib
- Certain ones are **very widely used**, and these are where we focus our attention
- The key names to look out for are:
  - Cholesky factorization : ?POTRF
  - LU factorization : ?GETRF
  - QR factorization : ?GETQR

© 2019 Arm Limited

Fast Fourier Transforms

FFTs are very commonly used in a wide variety of applications. They allow some hard problems to be transformed into a way that can be solved much more easily.

We ship 44 separate FFT routines, in 1-d, 2-d, 3-d and n-d

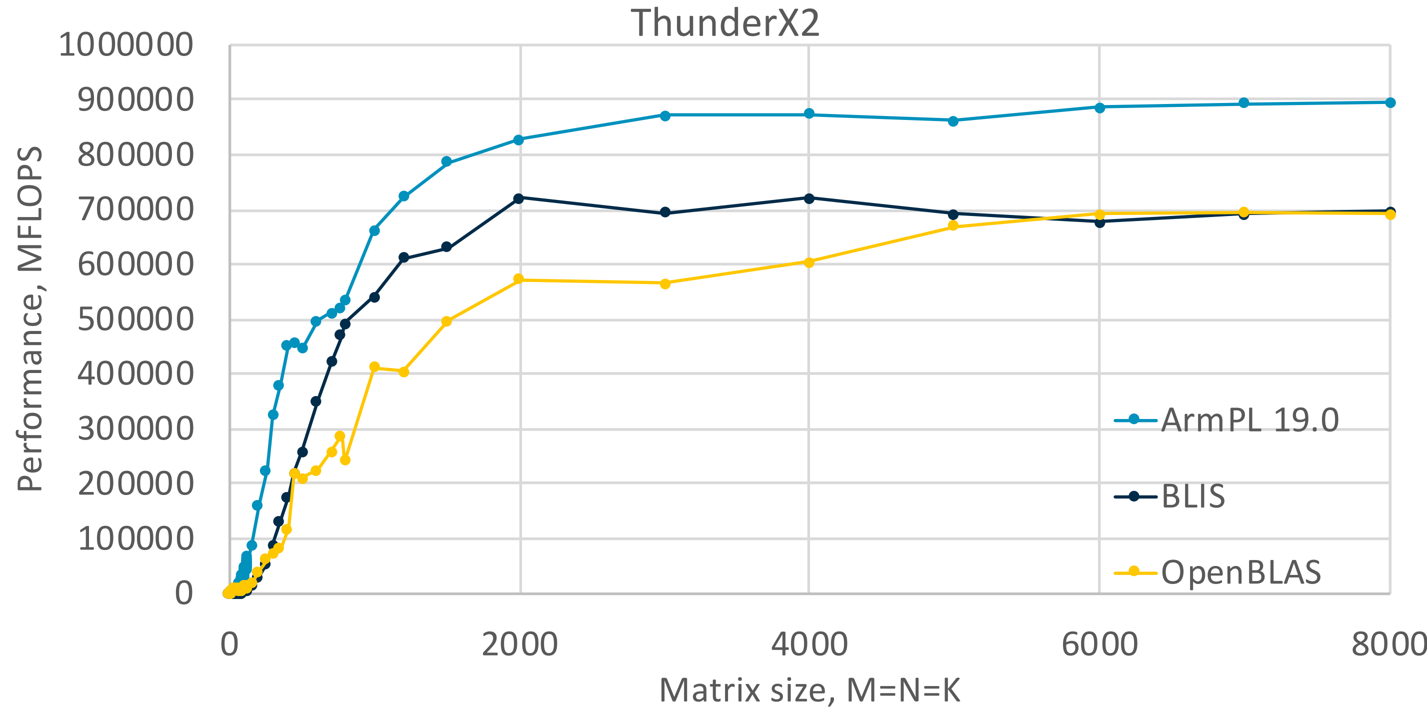
FFTs have no standard interface, unlike BLAS and LAPACK

- Instead we have an interface that matches that used by AMD's ACML library
- We also provide a compatibility layer to allow users to call using the FFTW3 interface
- Support for Basic, Advanced, Guru and MPI interfaces

Our interface is therefore documented on the website (only in a PDF still) and that PDF is also included in the installation

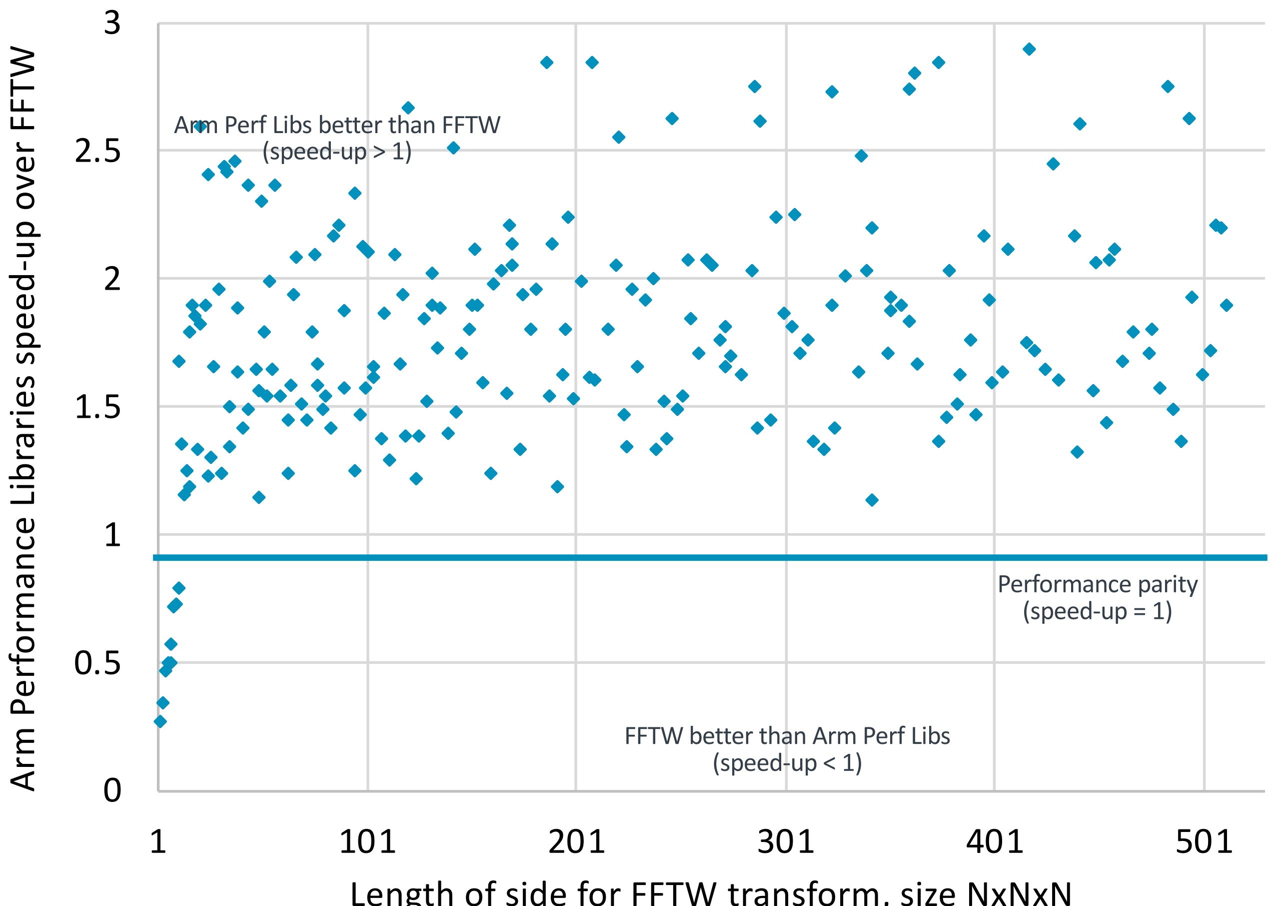
© 2019 Arm Limited

DGEMM – ArmPL 19.0 vs BLIS vs OpenBLAS : Parallel



© 2019 Arm Limited

ArmPL 19.0 FFT 3D complex-to-complex DP vs FFTW 3.3.7



© 2019 Arm Limited

Micro-architectural tuning

In order to achieve the best performance possible on all partner systems we need to do different micro-architectural tuning

All BLAS kernels are handwritten in *assembly code* in order to maximise overall performance

Different micro-architectures sometimes need fundamental differences in the instruction ordering – or even the instructions used

At run-time this work should all be transparent to the user

However multiple packages are typically available for users to choose from, and they need to load the appropriate module to set up their paths

Currently available are versions for:

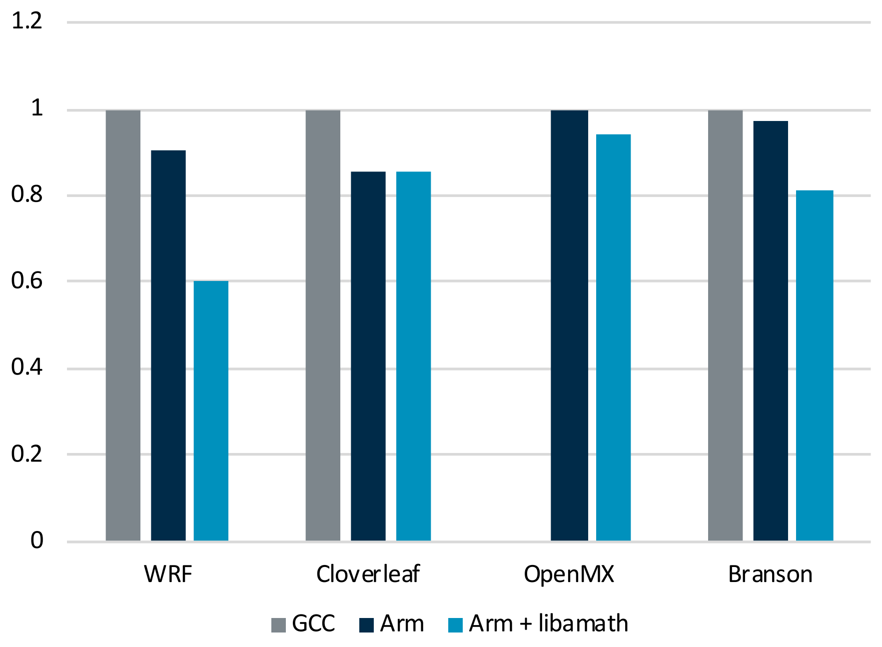
- A72
- Cavium ThunderX2
- Generic AArch64

© 2019 Arm Limited

Math Routine Performance

Distribution of https://github.com/ARM-software/optimized-routines

Normalised runtime



Arm PL provides libamath

- With Arm PL module loaded, include `-lamath` in the link line.
- Algorithmically better performance than standard library calls
- No loss of accuracy
- single and double precision implementations of: `exp()`, `pow()`, and `log()`
- single precision implementations of: `sin()`, `cos()`, `sincos()`, `tan()`

...more to come.

© 2019 Arm Limited

Open source libraries for improved performance

Arm Optimized Routines

https://github.com/ARM-software/optimized-routines

These routines provide high performing versions of many math.h functions

- Algorithmically better performance than standard library calls
- No loss of accuracy

SLEEF library

https://github.com/shibatch/sleef/

Vectorized math.h functions

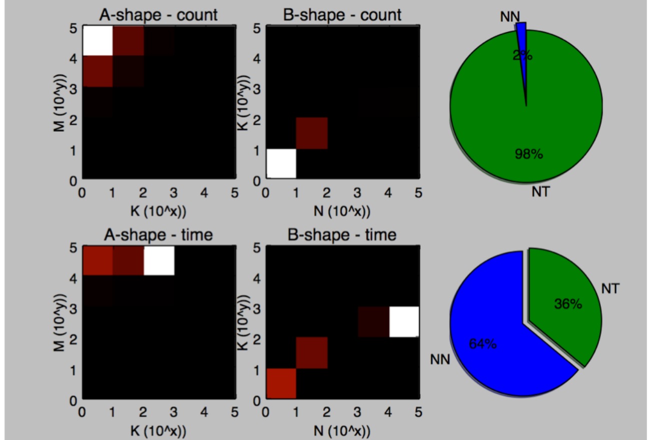
- Provided as an option for use in Arm Compiler

Perf-libs-tools

https://github.com/ARM-software/perf-libs-tools

Understanding an application's needs for BLAS, LAPACK and FFT calls

- Used in conjunction with Arm Performance Libraries can generate logging info to help profile applications for specific case breakdowns



Example visualization: DGEMM cases called

© 2019 Arm Limited

arm

Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

شكرًا

תודה

© 2019 Arm Limited

AWS Graviton Cluster: `108.128.237.67`

Available for the next 24 hours

- Pick a student number 1 … 20
- Replace XX with your student number
- `ssh student0XX@108.128.237.67`
  - *Password:* `Tr@ining0XX`
- Grab a compute node (8 cores per node):
  - `srun -n 8 --pty $SHELL`
- Remember to zero-pad your student number to three places, e.g. “`3`” becomes “`003`”
- We strongly recommend you download and install the Arm Forge Remote Client.

© 2019 Arm Limited

Tutorial Problem

- Matrix-matrix multiplication in C/C++:
  - Focus not on understanding the problem, but how to use the Arm toolchain
  - Naïve to optimized performance
- Program flow
  - Initialize random data
  - Perform multiply
- Use of compiler for single core and multi-core application
- Understanding of application performance
- Finish with your codes or other examples
  - Mini-apps etc.

© 2019 Arm Limited

Matrix-Matrix Multiply: Version 1

- Naïve implementation works on single core
- No consideration of underlying hardware

```
for (i= 0; i < n; ++i){
    for (j= 0; j < l; ++j){
        for (k= 0; k < m; ++k){
            matC[i][j] += matA[i][k] * matB[k][j];
        }
    }
}
```

- How bad is performance? Easier to evaluate when compared to other code

© 2019 Arm Limited

Matrix-Matrix Multiply: Version 2

- Add blocking to be able to take advantage of cache

```
for (kk= 0; kk < m; kk+= blockSize){
    for(jj= 0; jj < l; jj+= blockSize){
        for(i= 0; i < n; i+= 2){
            for(j= jj; j < min(l, jj + blockSize); j+= 2){
                for(k =kk; k < min(m, kk + blockSize); ++k){
                    matC[i][j] += matA[i][k] * matB[k][j];
                    matC[i][j+1] += matA[i][k] * matB[k][j+1];
                    matC[i+1][j] += matA[i+1][k] * matB[k][j];
                    matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                }
            }
        }
    }
}
```

- Have only added one level of blocking, which can take advantage of a single level of cache only
- Also little bit of loop unrolling to allow re-use of registers

© 2019 Arm Limited

Matrix-Matrix Multiply: Version 2

- Add blocking to be able to take advantage of cache

```
for (kk= 0; kk < m; kk+= blockSize){
    for(jj= 0; jj < l; jj+= blockSize){
        for(i= 0; i < n; i+= 2){
            for(j= jj; j < min(l, jj + blockSize); j+= 2){
                for(k =kk; k < min(m, kk + blockSize); ++k){
                    matC[i][j] += matA[i][k] * matB[k][j];
                    matC[i][j+1] += matA[i][k] * matB[k][j+1];
                    matC[i+1][j] += matA[i+1][k] * matB[k][j];
                    matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                }
            }
        }
    }
}
```

- Performance improvement of greater than 2x at 1024x1024 with 128 block size
- Still tens of seconds to perform a small multiply – bring out the big guns

© 2019 Arm Limited

Matrix-Matrix Multiply: Version 3

- Use Arm Performance Libraries
  - Addition of –larmpl flag to build command
  - Standard BLAS interface

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, m, l, 1, dataA,
        m, dataB, l, 1, dataC, m);
```

- Approximately 17x speed-up over blocked multiply and 40x over naïve implementation

 © 2019 Arm Limited

Matrix-Matrix Multiply: Version 4

- Have multiple physical cores available on the system – make use of them
- OpenMP directives added to loops
- Link to parallel version of the Arm Performance Libraries (-larmpl_mp)

```
#pragma omp parallel for private(i, j, jj, k, kk)
    for (kk= 0; kk < m; kk+= blockSize){
        for(jj= 0; jj < l; jj+= blockSize){
            for(i= 0; i < n; i+= 2){
                for(j= jj; j < min(l, jj + blockSize); j+= 2){
                    for(k =kk; k < min(m, kk + blockSize); ++k){
                        matC[i][j] += matA[i][k] * matB[k][j];
                        matC[i][j+1] += matA[i][k] * matB[k][j+1];
                        matC[i+1][j] += matA[i+1][k] * matB[k][j];
                        matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                    }
                }
            }
        }
    }
```

 © 2019 Arm Limited

Matrix-Matrix Multiply: Version 4

- Have multiple physical cores available on the system – make use of them
- Small matrix size means not all threads can be used with simple OpenMP directive
- Arm Performance library utilizes all threads – approximately 100x faster than blocking approach

```
#pragma omp parallel for private(i, j, jj, k, kk)
    for (kk= 0; kk < m; kk+= blockSize){
        for(jj= 0; jj < l; jj+= blockSize){
            for(i= 0; i < n; i+= 2){
                for(j= jj; j < min(l, jj + blockSize); j+= 2){
                    for(k =kk; k < min(m, kk + blockSize); ++k){
                        matC[i][j] += matA[i][k] * matB[k][j];
                        matC[i][j+1] += matA[i][k] * matB[k][j+1];
                        matC[i+1][j] += matA[i+1][k] * matB[k][j];
                        matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                    }
                }
            }
        }
    }
```


© 2019 Arm Limited

Matrix-Matrix Multiply: Version 4 (Interlude)

- Time to populate matrices with random data much higher when using OpenMP
- Use Arm Forge (performance analysis toolset) to investigate

```
        112  #pragma omp parallel for private(i, j)
        113      for (i= 0; i < n; ++i){
        114          for (j= 0; j < m; ++j){
37.5%   115              matA[i][j]= ((double)rand()) / RAND_MAX;
        116          }
        117
        118          for (j= 0; j < l; ++j){
        119              matC[i][j]= 0.0;
        120          }
        121      }
        122
        123  #pragma omp parallel for private(i, j)
        124      for (i= 0; i < m; ++i){
        125          for (j= 0; j < l; ++j){
35.6%   126              matB[i][j]= ((double)rand()) / RAND_MAX;
        127          }
        128      }
```

- Lots of time spent waiting on the kernel in glibc rand function
  - Kernel lock being held means that parallel performance can’t be obtained

 © 2019 Arm Limited

Matrix-Matrix Multiply: Version 5 (Interlude)

- Update random function to be thread safe implementation (e.g. C++ stdlib functions)

```
        117      for (i= 0; i < n; ++i){
        118          for (j= 0; j < m; ++j){
5.0%    119              matA[i][j]= ((double)uniform_dist(el)) / maxRandNum;
        120          }
        121
        122          for (j= 0; j < l; ++j){
<0.1%   123              matC[i][j]= 0.0;
        124          }
        125      }
        126
        127  #pragma omp parallel for private(i, j)
        128      for (i= 0; i < m; ++i){
        129          for (j= 0; j < l; ++j){
6.4%    130              matB[i][j]= ((double)uniform_dist(el)) / maxRandNum;
        131          }
        132      }
        133
```

- Can make use of all cores in parallel

© 2019 Arm Limited

Matrix-Matrix Multiply: Version 6

- Allow compiler to make better decisions on loop parallelism

```
#pragma omp parallel for collapse(2) private(i, j, jj, k, kk)
    for (kk= 0; kk < m; kk+= blockSize){
        for(jj= 0; jj < l; jj+= blockSize){
            for(i= 0; i < n; i+= 2){
                for(j= jj; j < min(l, jj + blockSize); j+= 2){
                    for(k =kk; k < min(m, kk + blockSize); ++k){
                        matC[i][j] += matA[i][k] * matB[k][j];
                        matC[i][j+1] += matA[i][k] * matB[k][j+1];
                        matC[i+1][j] += matA[i+1][k] * matB[k][j];
                        matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                    }
                }
            }
        }
    }
```

- Allows better use of threads – 8x speed-up over not using collapse directive
- Still greater than 10x slower than Arm Performance Libraries

© 2019 Arm Limited

arm

Debug

Exercise 1: Fix a simple crash in MPI

Objectives:

- Discover Arm DDT’s interface
- Debug a simple crash in a MPI application interactively
- Use the tool in a cluster environment

Key commands:

- $ cd 01_*/
- Compile the application
  - $ make
- Run it!
  - $ ./Run.sh
- Accept the incoming connection!
- Can you find out and fix the bug?

© 2019 Arm Limited

Exercise 2: Debug a fatal memory crash

Objectives:

- Use the memory debugging feature
- Diagnose and fix a memory problem

Key commands:

- $ cd 02_*/
- Compile the application with debugging flags
  - $ make
- Run it!
  - $ ./Run.sh
- Enable memory debugging in the “Run window”
- Change the amount of checks, enable guard pages
- Can you see the memory issue can you fix it?

© 2019 Arm Limited

Exercise 3: Detect memory leaks

Objectives:

- Use the memory debugging feature
- Diagnose and fix a memory leak problem

Key commands:

- $ cd 03_*/
- Compile the application for debugging
  - $ make
- Run it!
  - $ ./Run.sh
- Open the resulting *.html file
- Can you see the memory leak?
- Restart the debugger in interactive mode. Can you see any hint from the debugger?

© 2019 Arm Limited

Exercise 4: Offline debugging

Objectives:

- Use Arm DDT’s offline mode
- Use the memory debugging feature
- Diagnose and fix a memory leak problem

Key commands:

- $ cd 04_*/
- Compile the application for debugging
  - $ make
- Run it!
  - $ ./Run.sh
- Open the resulting *.html file
- Can you see the memory leak?
- Restart the debugger in interactive mode. Can you see any hint from the debugger?

© 2019 Arm Limited

Exercise 5: Debug a deadlock

Objectives:

- Witness a deadlock and attach to the running processes
- Use Arm DDT Stack feature
- Use Arm DDT evaluation window

Key commands:

- $ cd 05_deadlock/
- Compile with:
  - $ make
- Run the job with 10 processes: it works.
- Run it with 8 processes: it hangs!
- Leave the application run in the queue and attach to it with the debugger
- Observe where it hangs. Can you fix the problem?

© 2019 Arm Limited

Exercise 6: Computation Error

Objectives:

- Fix a computation error using the multi-dimensional array viewer (MDA)
- Use the debugger on a MPMD application
- Use breakpoints

Key commands:

- $ cd 06_*/
- Compile the application for debugging
  - $ make
- Run it!
  - $ ./Run.sh
- Look at the output log files, you should see NaN results which indicate a computation error
- Use the MDA to visualise the result data and find the source of the problem.

© 2019 Arm Limited

Arm Forge and MVAPICH2

- To use DDT’s memory debugging features, **set the environment variable MV2_ON_DEMAND_THRESHOLD to the maximum job size you expect.** This setting should ***not*** be a system wide default; it should be set as needed.
- To use mpirun_rsh with DDT, from *File → Options* go to the *System* page, check *Override default mpirun path* and enter mpirun_rsh. You should also add -hostfile <hosts>, where <hosts> is the name of your hosts file, within the *mpirun_rsh arguments* field in the *Run* window.
- To enable message Queue Support MVAPICH 2 must be compiled with the flags --enable-debug --enable-sharedlib. These are not set by default.
- MVAPICH2 MPI programs cannot be started using Express Launch syntax.
  - Do use: “ddt ./a.out” and configure MPI launch parameters in the GUI.
  - ~~Don’t use: “ddt mpirun <mpi_args> ./a.out”~~

© 2019 Arm Limited