Tutorial Schedule

• [15] Performance Engineering: Methodology and Tools
• [15] Arm Forge Quick Start: DDT, MAP, and Performance Reports
• [30] Exercises
  • Interactive debugging
  • Profiling from the command line
  • Detect memory leaks
  • Debug invalid memory access
• [30] Break
• [20] Exercises and Examples
  • Explore I/O imbalance with MAP and performance reports
  • Real-world success story
  • Custom metrics for Lustre profiling
• [10] Q&A
Performance Engineering
Methodology and Tools
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)
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.
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.
Identifying and resolving performance issues

Identify Hotspots

-50x

File I/O

No

-10x

Communication

No

-5x

Memory

No

-2x

Compute

No

Refine the Profile

Yes

Focus Optimization

Buffers, data formats, in-memory filesystems

Collectives, blocking, non-blocking, topology, load balance

Bandwidth/latency, cache utilization

Vectors, branches, integer, floating point

Refine the Profile
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

---

**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 ALLinea Studio

- C/C++ Compiler
- Fortran Compiler
- Performance Libraries
- Forge (DDT and MAP)
- Performance Reports
Arm Forge = DDT + MAP
An interoperable toolkit for debugging and profiling

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

Commercially supported by Arm
Fully Scalable
Very user-friendly
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

Connect to continuous integration

Display pending communications

Visualise data structures
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”: $\text{value} > 0$ for example $\text{value}$ being a specific element of the array
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
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 Performance Reports

Characterize and understand the performance of HPC application runs

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)

Commercially supported by Arm

Accurate and astute insight

Relevant advice to avoid pitfalls
Arm Performance Reports

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

**Summary: hydro is MPI-bound in this configuration**

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>20.6%</td>
</tr>
<tr>
<td>MPI</td>
<td>63.2%</td>
</tr>
<tr>
<td>I/O</td>
<td>16.2%</td>
</tr>
</tbody>
</table>

**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 Performance Reports Metrics

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

- Multi-threaded parallelism
- SIMD parallelism
- Load imbalance
- OMP efficiency
- System usage

- CPU
  - A breakdown of the 91.2% CPU time:
  - Single-core code: 30.6%
  - OpenMP regions: 69.4%
  - Scalar numeric ops: 9.5%
  - Vector numeric ops: 1.3%
  - Memory accesses: 0.0%

- MPI
  - Of the 41.3% total time spent in MPI calls:
    - Time in collective calls: 100.0%
    - Time in point-to-point calls: 0.0%
    - Estimated collective rate: 4.07 bytes/s
    - Estimated point-to-point rate: 0 bytes/s

- OpenMP
  - A breakdown of the 99.5% time in OpenMP regions:
    - Computation: 58.9%
    - Synchronization: 41.1%
    - Physical core utilization: 100.0%
    - System load: 99.7%

- I/O
  - A breakdown of how the 53.0% total I/O time was spent:
    - Time in reads: 38.9%
    - Time in writes: 14.1%
    - Estimated read rate: 2.1 GB/s
    - Estimated write rate: 2.1 GB/s

- Memory
  - Per-process memory usage may also affect scaling:
    - Mean process memory usage: 160 Mb

- Lustre
  - Lustre file operations (per node)

- Energy
  - A breakdown of how the 32.3 Wh was used:
    - CPU: 61.5%
    - System: 38.1%
    - Mean node power: 94.1 W
    - Peak node power: 98.0 W

Significant time is spent waiting for memory accesses. Reducing the CPU clock frequency could reduce overall energy usage.
VI-HPS and the tools ecosystem

See the http://www.vi-hps.org/tools/ for an excellent view of the tools ecosystem.
Arm Forge Quick Start

Tool cheat sheets
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 ...`
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 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
--attach=[host1:]pid1,[host2:]pid2... [PROGRAM]
--attach-mpi=mpi_PID [--subset=rank1,rank2,rank3,...] [PROGRAM]
--break-at=LOCATION[,START:EVERY:STOP] [if CONDITION]
--trace-at=LOCATION[,START:EVERY:STOP],VAR1,VAR2,...
--cuda
--mem-debug=[(fast|balanced|thorough|off)]
--mpiargs=ARGUMENTS
-n, --np, --processes=NUMPROCS
--nodes=NUMNODES
--procs-per-node=PROCS
--offline
-s, --silent

Reverse Connect (launch as a server and wait)
attach to PROGRAM being run by list of host:pid
attach to processes in an MPI program.
set a breakpoint at LOCATION
set a tracepoint at LOCATION
enable CUDA
configure memory debugging (defaults to fast)
command line arguments to pass to mpirun
specify the number of MPI processes
configure the number of nodes for MPI jobs
configure the number of processes per node
run through program without user interaction
don't write unnecessary output to the command line
Arm MAP cheat sheet

Generate profiles and view offline

• 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

• 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)
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
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
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.
The Forge GUI and where to run it

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

On the head node
(interactive mode + reverse connect)

On the compute node
(offline OR interactive mode)

Remote client
(remote launch + reverse connect)

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


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!
Arm Forge 18.1.2 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”
Interactive Debugging
Crash and hang
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
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
  $ mpirun -np 4 ./mmult1_c.exe
  # Observe crash
  $ ddt ./mmult1_c.exe
  # Observe cause of crash

Initial Result: Crash!
Answer: Fix incorrect limits on k-loop

Incorrect limits lead to invalid memory access

Before

164    do  i=0, size/nslices-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
172     end do

After

164    do  i=0, size/nslices-1
165    do  j=0, size-1
166     res=0.0
167     do  k=0, size-1
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
172     end do
Answer: Fix incorrect limits on i-loop

Incorrect limits on i-loop lead to unmatched MPI_Send

Before

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

After

```fortran
73    do i=1,nproc-1
74       call MPI_Send(mat_a(slice*i), slice, &
                  MPI_DOUBLE, i, 100+i, &
                  MPI_COMM_WORLD, ierr)
75       call MPI_Send(mat_b, size*size, &
                  MPI_DOUBLE, i, 200+i, &
                  MPI_COMM_WORLD, ierr)
76       call MPI_Send(mat_c(slice*i), slice, &
                  MPI_DOUBLE, i, 300+i, &
                  MPI_COMM_WORLD, ierr)
77    end do
```
Improve performance
Efficient memory access
Fix inefficient memory access pattern

It works! But wow it’s slow.

Exercise Outline

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

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

Find the hotspot: look for the line with the highest core time.
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] ...
}
```
Memory Accesses and Cache Misses

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

For `i=0, n=4`, `j=0` is a hit and `j=1` is a miss.

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

For `i=0, n=4`, `j=0` is a hit and `j=1` is a miss.
Answer: Transpose matrix and interchange loops

Transposing the matrix improves locality $\rightarrow$ performance

**Before**

164  do  i=0,size/nslices-1
165  do  j=0,size-1
166    res=0.0
167    do  k=0,size-1
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
172  end do

**After**

165  do  i=0,size/nslices-1
166  do  j=0,size-1
167    res=0.0
168    do  k=0,size-1
169      res=A(i*size+k)*transB(j*size+k)+res
170    end do
171    C(i*size+j)=res+C(i*size+j)
172  end do
173  end do
Final profile

About 3x faster

Before

After
Leak Detection

... and DDT in Offline Mode
Possible memory leak

Transpose 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
  $ ddt --offline --output=report.html -n 4 ./mmult3_f90.exe
  $ xdg-open report.html
  # Observe report

DDT in offline mode (--offline)
DDT Debugging Report

Use DDT’s reporting feature to debug long-running applications
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
Memory Debugging

Allocation tracking and guard pages
Three levels of heap debugging overhead

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

**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.

**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.

- **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.

- **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 reallocating a memory allocation (e.g. due to realloc)

See user-guide:
Chapter 12.3.2
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

- **Commands**
  
  $ make
  $ ddt -n 4 ./trisol.exe
  # Enable fast memory debugging
  # Do **not** enable guard pages

Invalid memory access
## DDT’s heap memory debugging framework

### Dynamically linked binaries
- LD_PRELOAD is usually used automatically
- Not on static binaries, not on all Crays or old SLURMs

### Statically linked binaries
- If not, manual linking is required
  
  LFLAGS = -dynamic -L/path/to/forge/lib/64/ -zmuldefs -Wl,--undefined=malloc -ldmalloc

When manual linking is used, untick “Preload” box
It works in DDT??????

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

**DDT launch configuration**

**Uh oh, program output looks great**

It should have crashed! What changed?
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)
Debugging Imbalance

MPI I/O
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 -n 4 \n  ./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**

```fortran
97  if(myrank==0) then
100  do i=1,nproc-1
101    call MPI_Recv(mat_c(slice*i), slice, &
102      MPI_DOUBLE, &
103      i, 500+i, &
104      MPI_COMM_WORLD, st, ierr)
102  end do
103  else
106    call MPI_Send(mat_c, slice, MPI_DOUBLE, &
107      0, 500+myrank, &
108      MPI_COMM_WORLD, ierr)
107  end if
109  if(myrank==0) then
111    call mwrite(size, mat_c, filename)
113  endif
```

**After**

```fortran
102  call MPI_FILE_OPEN(MPI_COMM_WORLD, &
103      filename, &
104      MPI_MODE_CREATE+MPI_MODE_WRONLY, &
105      MPI_INFO_NULL, fh, ierr)
103  call MPI_FILE_SET_VIEW(fh, &
104      0_MPI_OFFSET_KIND, MPI_DOUBLE, &
105      MPI_DOUBLE, 'native', &
106      MPI_INFO_NULL, ierr)
104  call MPI_FILE_WRITE_AT(fh, disp, mat_c, &
105      slice, MPI_DOUBLE, st, ierr)
105  call MPI_BARRIER(MPI_COMM_WORLD, ierr)
106  call MPI_FILE_CLOSE(fh, ierr)
```
New approach: use MPI-IO for file output

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

Before: runtime 13 seconds

After: runtime 5 seconds (2.6x speedup)
Final profile shows balanced I/O and compute dominates

New approach is about 3x faster
Success at Scale
Curtin Quantum Collisions
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
Initial profile at production scale
Load balancer is imbalanced?

Customized load balancing algorithm wasn’t delivering expected results

<table>
<thead>
<tr>
<th>LG</th>
<th>node</th>
<th>ipar</th>
<th>inc</th>
<th>vt</th>
<th>i1</th>
<th>i2</th>
<th>tperi</th>
<th>nch</th>
<th>naps</th>
<th>mt</th>
<th>prev</th>
<th>LG,eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
<td>-10</td>
<td>199</td>
<td>329</td>
<td>492</td>
<td>1.21</td>
<td>13530</td>
<td>0</td>
<td></td>
<td>89</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
<td>-7</td>
<td>591</td>
<td>573</td>
<td>872</td>
<td>1.97</td>
<td>45150</td>
<td>0</td>
<td>350</td>
<td>0</td>
<td>80%</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0</td>
<td>-16</td>
<td>894</td>
<td>762</td>
<td>1153</td>
<td>2.28</td>
<td>77028</td>
<td>0</td>
<td>607</td>
<td>1</td>
<td>86%</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>0</td>
<td>-24</td>
<td>916</td>
<td>886</td>
<td>1331</td>
<td>2.05</td>
<td>99681</td>
<td>0</td>
<td>766</td>
<td>2</td>
<td>91%</td>
</tr>
</tbody>
</table>
“That makes no sense!”
Computing one grid point takes as much time as computing the entire grid.

Surprise! Didn’t expect that.
Final profile, again at production scale

Found an unbounded array copy \(a(:)\) that should have been \(a(1:N)\)
Balanced the load balancer
Load can be balanced now that work blocks are of expected sizes

Before:

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

After:

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
<td>-10</td>
<td>174</td>
<td>329</td>
<td>492</td>
<td>1.06</td>
<td>13530</td>
<td>0</td>
<td>85</td>
<td>-1</td>
<td>93%</td>
<td>LG, node, ipar, inc, vt, i1, i2, tperi, nch, naps, mt, prev</td>
<td>LG, eff</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
<td>-11</td>
<td>415</td>
<td>577</td>
<td>872</td>
<td>1.40</td>
<td>43956</td>
<td>0</td>
<td>340</td>
<td>0</td>
<td>97%</td>
<td>LG, node, ipar, inc, vt, i1, i2, tperi, nch, naps, mt, prev</td>
<td>LG, eff</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0</td>
<td>-11</td>
<td>616</td>
<td>757</td>
<td>1153</td>
<td>1.55</td>
<td>79003</td>
<td>0</td>
<td>592</td>
<td>1</td>
<td>97%</td>
<td>LG, node, ipar, inc, vt, i1, i2, tperi, nch, naps, mt, prev</td>
<td>LG, eff</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>0</td>
<td>-12</td>
<td>667</td>
<td>874</td>
<td>1331</td>
<td>1.46</td>
<td>105111</td>
<td>0</td>
<td>734</td>
<td>2</td>
<td>96%</td>
<td>LG, node, ipar, inc, vt, i1, i2, tperi, nch, naps, mt, prev</td>
<td>LG, eff</td>
</tr>
</tbody>
</table>
Custom metrics for Lustre

Combine I/O performance data from system and application
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
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.
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.

| Main thread activity          |  
|-------------------------------|-------------------------------|
| **Lustre read rate**          | 1.57 MB/s                     |
| **Lustre write rate**         | 0.01 GB/s                     |
| **Lustre metadata operations**| 1.38 k/s                      |
| **Lustre file opens**         | 0.68 k/s                      |

**Checkpoint I/O** corresponds to spike in Lustre write rate

N-N file read shows spike in file open/read operations.
We can focus on each I/O operation individually

Select a portion of the application timeline to view the source code performing I/O.
MAP’s timeline shows I/O overlapping with communication

We see elevated Lustre write rate when writing checkpoint restart files in HDF5.
It’s possible to overlap different I/O approaches

HDF5 and VTK I/O operations occur at the same time on different ranks.
Wrap Up
Five great things to try with Arm DDT

- The scalable print alternative
- Stop on variable change
- Static analysis warnings on code errors
- Detect read/write beyond array bounds
- Detect stale memory allocations
Six Great Things to Try with Arm MAP

- Find the peak memory use
- Fix an MPI imbalance
- Remove I/O bottleneck
- Make sure OpenMP regions make sense
- Improve memory access
- Restructure for vectorization
Wrap Up

Visit arm.com/hpc to learn more about Arm Forge and download a free trial.

- Tools are a must-have when programming HPC systems
- Use a structured, profile-driven optimization methodology
- Arm DDT can help improve code correctness
- Arm MAP can help improve code performance
- Arm Forge = DDT + MAP is a great choice at scale

Download at arm.com/hpc
Thank You
Danke
Merci
谢谢
ありがとうございます
Gracias
Kiitos
감사합니다
धन्यवाद
תודה