Agenda

• Arm Software for Debugging and Profiling
• Debugging with DDT
• Generating Performance Reports
• Profiling with MAP
• Using Arm tools with Python
**Arm Forge**

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 parallel applications running at petascale)

**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**
Arm Performance Reports

Characterize and understand the performance of HPC application runs

Gathers a rich set of data

- Analyzes metrics around CPU, memory, IO, hardware counters, etc.
- Possibility for users to add their own metrics

Build a culture of application performance & efficiency awareness

- Analyzes 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)
Run and ensure application correctness

Combination of debugging and re-compilation

- Ensure application correctness with **Arm DDT scalable debugger**
- Integrate with continuous integration system.
- Use version control to track changes and leverage Forge’s built-in VCS support.

**Examples:**

```
$> ddt --offline aprun -n 48 ./example
$> ddt --connect aprun -n 48 ./example
```
Visualize the performance of your application

- Measure all performance aspects with Arm MAP parallel profiler
- Identify bottlenecks and rewrite some code for better performance

Examples:
```bash
$> map --profile -n 48 ./example
```
Debugging with DDT
Arm DDT – The Debugger

Who had a rogue behaviour?
- Merges stacks from processes and threads

Where did it happen?
- Leaps to source

How did it happen?
- Diagnostic messages
- Some faults evident instantly from source

Why did it happen?
- Unique “Smart Highlighting”
- Sparklines comparing data across processes
Preparing Code for Use with DDT

As with any debugger, code must be compiled with the debug flag typically \texttt{--g}

It is recommended to turn off optimization flags i.e. \texttt{--O0}

Leaving optimizations turned on can cause the compiler to \textit{optimize out} some variables and even functions making it more difficult to debug
Segmentation Fault

In this example, the application crashes with a segmentation error outside of DDT.

What happens when it runs under DDT?
Segmentation Fault in DDT

DDT takes you to the exact line where Segmentation fault occurred, and you can pause and investigate.
Invalid Memory Access

The array tab is a 13x13 array, but the application is trying to write a value to tab(4198128,0) which causes the segmentation fault.

i is not used, and x and y are not initialized
Track Your Changes in a Logbook
New Bugs from Latest Changes
Arm DDT Demo
It works… Well, most of the time

A strange behaviour where the application “sometimes” crashes is a typical sign of a memory bug.

Arm DDT is able to force the crash to happen.

SCHRODIN BUG
Advanced Memory Debugging

The image shows a screenshot of a software interface for memory debugging. The interface includes options for selecting debugging methods such as OpenMP, CUDA, and Memory Debugging. There are also settings for heap debugging, overflow/underflow detection, and advanced options. The interface also includes a note explaining that preloading only works for programs linked against shared libraries, and if the program is statically linked, the user must relink it against the dmalloc library manually.
# Heap debugging options available

<table>
<thead>
<tr>
<th>Fast</th>
<th>Balanced</th>
<th>Thorough</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>basic</strong></td>
<td>• Detect invalid pointers passed to memory functions (e.g. <code>malloc</code>, <code>free</code>, <code>ALLOCATE</code>, <code>DEALLOCATE</code>,...)</td>
<td><strong>free-blank</strong></td>
</tr>
<tr>
<td><strong>check-fence</strong></td>
<td>• Check the end of an allocation has not been overwritten when it is freed.</td>
<td><strong>alloc-blank</strong></td>
</tr>
<tr>
<td><strong>free-protect</strong></td>
<td>• Protect freed memory (using hardware memory protection) so subsequent read/writes cause a fatal error.</td>
<td><strong>check-heap</strong></td>
</tr>
<tr>
<td><strong>Added goodness</strong></td>
<td>• Memory usage, statistics, etc.</td>
<td><strong>realloc-copy</strong></td>
</tr>
</tbody>
</table>

See user-guide: Chapter 12.3.2
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
Five great things to try with Allinea DDT

- The scalable print alternative
- Stop on variable change
- Static analysis warnings on code errors
- Detect read/write beyond array bounds
- Detect stale memory allocations
Arm DDT cheat sheet

Load the environment module

• $ module load allinea-forg

Prepare the code

• $ cc -O0 -g myapp.c -o myapp.exe

Start Arm DDT in interactive mode

• $ ddt srun -n 8 ./myapp.exe arg1 arg2

Or use the reverse connect mechanism

• On the login node:
  • $ ddt &
  • (or use the remote client) ← Preferred method

• Then, edit the job script to run the following command and submit:
  • ddt --connect srun -n 8 ./myapp.exe arg1 arg2
Generating Performance Reports
Profiling is central to understanding and improving application performance.

**Identify Hotspots**
- File I/O (50x)
- Communication (10x)
- Memory (5x)
- CPU (2x)
- Refine the Profile

**Focus Optimization**
- Buffers, data formats, in-memory filesystems
- Collectives, blocking, non-blocking, topology, load balance
- Bandwidth/latency, cache utilization
- Vectors, branches, integer, floating point
Arm Performance Reports

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

Summary: hydro is MPI-bound in this configuration

<table>
<thead>
<tr>
<th>Compute</th>
<th>20.6%</th>
<th>Time spent running application code. High values are usually good. This is very low; focus on improving MPI or I/O performance first</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>63.2%</td>
<td>Time spent in MPI calls. High values are usually bad. This is high; check the MPI breakdown for advice on reducing it</td>
</tr>
<tr>
<td>I/O</td>
<td>16.2%</td>
<td>Time spent in filesystem I/O. High values are usually bad. This is average; check the I/O breakdown section for optimization advice</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.
After the fix, write rate has improved 41.6x

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

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

<table>
<thead>
<tr>
<th>Compute</th>
<th>23.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>75.5%</td>
</tr>
<tr>
<td>I/O</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

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.
Performance Reports Demo
LAMMPS IO Performance Report Suggests Using MPI Profiler

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

**CPU**
A breakdown of the 22.1% CPU time:
- Single-core code: 96.2%
- OpenMP regions: 3.8%
- Scalar numeric ops: 34.4%
- Vector numeric ops: 0.0%
- Memory accesses: 65.6%

Per-process performance is dominated by serial sections of computation. Use a profiler to find these or run with fewer threads and more processes.

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

**MPI**
A breakdown of the 76.7% MPI time:
- Time in collective calls: 38.2%
- Time in point-to-point calls: 61.8%
- Effective process collective rate: 293 kB/s
- Effective process point-to-point rate: 80.8 MB/s

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

The collective transfer rate is very low. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.
Built-in Timers vs Arm MAP

Loop time of 21.7979 on 32 procs for 25000 steps with 80331 atoms

Performance: 297276.400 tau/day, 1146.900 timesteps/s
94.1% CPU use with 32 MPI tasks x 1 OpenMP threads

MPI task timing breakdown:

<table>
<thead>
<tr>
<th>Section</th>
<th>min time</th>
<th>avg time</th>
<th>max time</th>
<th>%varavg</th>
<th>%total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td>2.8332</td>
<td>2.9174</td>
<td>3.177</td>
<td>6.3</td>
<td>13.38</td>
</tr>
</tbody>
</table>

MAP analysing program...
MAP gathering samples...
MAP generated /global/u2/rhulguin/cori/lammps/Lammps_io_32.map

Missing debugging information

Normally MAP will display your source files here. This time, that hasn’t been possible. There are a few possible explanations for this:
1. You didn’t compile your program with debugging information. Normally this means passing an extra -g flag to the compiler. If your executable does not contain debugging information then MAP will not be able to show you your source files.
2. Your program may have debugging information, but MAP was...
Break
Profiling with MAP
Arm MAP – The Profiler

- Small data files
- <5% slowdown
- No instrumentation
- No reccompilation
Glean Deep Insight from our Source-Level Profiler

- Track memory usage across the entire application over time
- Spot MPI and OpenMP imbalance and overhead
- Optimize CPU memory and vectorization in loops
- Detect and diagnose I/O bottlenecks at real scale
Profile of 2d Laplace Solver with Jacobi Iteration
Tracking Largest Change

// Compare newly computed value with old value
diff = fabs(grid2[i][j] - grid1[i][j]);

// Track largest change between new and old values
maxDiff = diff > maxDiff ? Diff : maxDiff;

If (diff > maxDiff)
    then maxDiff = diff;
Else
    maxDiff = maxDiff;
Conditional Removal from Innermost Loop

20 % faster, also operation is now vectorized
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.
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.
Solution: use HDF5 to write binary files

Using a library optimized for HPC I/O improves performance and portability.
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.
Arm Map Handson
Arm MAP: Python profiling

• Launch command
  • $ python ./laplace1.py slow 100 100

• Profiling command
  • $ map --profile python ./laplace1.py slow 100 100
    • --profile: non-interactive mode
    • --output: name of output file

• Display profiling results
  • $ map laplace1.map

Laplace1.py

[...]
err = 0.0
for i in range(1, nx-1):
    for j in range(1, ny-1):
        tmp = u[i,j]
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                  (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
        diff = u[i,j] - tmp
        err += diff*diff
return numpy.sqrt(err)
[...]
Naïve Python loop (laplace1.py slow 100 1000)
Optimizing computation on NumPy arrays

Naïve Python loop

```python
err = 0.0
for i in range(1, nx-1):
    for j in range(1, ny-1):
        tmp = u[i,j]
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy + (u[i, j-1] + u[i, j+1])*dx)*dnr_inv
        diff = u[i,j] - tmp
        err += diff*diff
return numpy.sqrt(err)
```

NumPy loop

```python
u[1:-1, 1:-1] =
    ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
     (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
return g.computeError()
```
NumPy array notation (laplace1.py numeric 1000 1000)

This is 10 times more iterations than was computed in the previous profile.
Arm MAP cheat sheet

Load the environment module (manually specify version)

- $ module load allinea-forge

Follow the instructions displayed to prepare the code

- $ cc -O3 -g myapp.c -o myapp.exe
- Edit the job script to run Arm MAP in “profile” mode
- $ map --profile -n 8 ./myapp.exe arg1 arg2

Open the results

- 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 --connect myapp_Xp_Yn_YYYY-MM-DD_HH-MM.map
Six Great Things to Try with Allinea 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
Cori Specific Settings
Configure the remote client

Install the Arm Remote Client


Connect to the cluster with the remote client

- Open your Remote Client
- Create a new connection: Remote Launch ➔ Configure ➔ Add
  - Hostname: <username>@cori.nersc.gov
  - Remote installation directory:
    /usr/common/software/allinea-forge/20.1-Suse-15.0-x86_64
Examples for hands-on session

Examples are available at /global/cfs/cdirs/training/2020/arm-tools/

ddt/ddt_demo

ddt/memory_debugging

perf-report/

map/

python/
Questions?
Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद