Arm Forge
Debugging and Optimization Tools for HPC

Beau Paisley<Beau.Paisley@arm.com>
ATPESC2022
Arm Forge

An interoperable toolkit for debugging and profiling

The de-facto standard for HPC development

• Most widely-used debugging and profiling suite in HPC
• Fully supported by Arm on Intel, AMD, Arm, IBM Power, Nvidia GPUs, etc.

State-of-the-art debugging and profiling capabilities

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

Easy to use by everyone

• Unique capabilities to simplify remote interactive sessions
• Innovative approach to present quintessential information to users
DDT Debugger Highlights

- The scalable print alternative
- Stop on variable change
- Static analysis warnings on code errors
- Detect read/write beyond array bounds
- Detect stale memory allocations
9 Step guide: optimizing high performance applications

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

1. Bugs
   - Correct application.

2. Analyze before you optimize
   - Measure all performance aspects.
   - You can’t fix what you can’t see.
   - Prefer real workloads over artificial tests.

3. Communication
   - Track communication performance.
   - Discover which communication calls are slow and why.

4. Workload
   - Detect issues with balance.
   - Slow communication calls and processes.
   - Dive into partitioning code.

5. Memory
   - Reveal lines of code bottlenecked by memory access times.
   - Trace allocation and use of hot data structures.

6. Cores
   - Discover synchronization overhead and core utilization.
   - Synchronization-heavy code and implicit barriers are revealed.

7. Vectorization
   - Understand numerical intensity and vectorization level.
   - Hot loops, unvectorized code and GPU performance revealed.

8. Verification
   - Validate corrections and optimal performance.

Key:

- arm PERFORMANCE REPORTS
- arm FORGE

I/O
- Discover lines of code spending a long time in I/O.
- Trace and debug slow access patterns.
Arm Performance Reports

Summary: clover_leaf is Compute-bound in this configuration

- Compute 100.0%
  - Time spent running application code: High values are usually good. This is very helpful when debugging in OpenMP.
  - Time spent in MPI calls: High values are usually bad. This is very low, so there may be benefits from a higher process count.
  - Time spent in filesystem I/O: High values are usually bad. This is negligible, there’s no need to investigate I/O performance.
  - No time spent in MPI operations. There’s nothing to optimize here!

- MPI 0.0%
  - Time in collective calls: 0.0%
  - Time in point-to-point calls: 0.0%
  - Effective process collective rate: 0.00 bytes/second
  - Effective process point-to-point rate: 0.00 bytes/second
  - No time is spent in MPI operations. There’s nothing to optimize here!

- I/O 0.0%
  - Time in reads: 0.0%
  - Time in writes: 0.0%
  - Effective process read rate: 0.00 bytes/second
  - Effective process write rate: 0.00 bytes/second
  - No time is spent in I/O operations. There’s nothing to optimize here!

- OpenMP
  - A breakdown of the 99.7% time in OpenMP regions:
    - Computation: 95.1%
    - Synchronization: 14.4%
    - Physical core utilization: 0.7%
    - System load: 7.6%
    - Physical core utilization is low and some rates may be missed. Try increasing OMP_NUM_THREADS to improve performance.

- Memory
  - Per-process memory usage may also affect scaling:
  - Mean process memory usage: 312 MB
  - Peak process memory usage: 214 MB
  - Peak node memory usage: 2.0%
  - The peak node memory usage is very low, larger problem sets can be run without starting to multiple nodes.

No source code needed

Less than 5% runtime overhead

Fully scalable

Run regularly – or in regression tests

Explicit and usable output
MAP Source Code Profiler Highlights

- 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
Python Profiling

- 19.0 adds support for Python
  - Call stacks
  - Time in interpreter

- Works with MPI4PY
  - Usual MAP metrics

- Source code view
  - Mixed language support

Note: Green as operation is on numpy array, so backed by C routine, not Python (which would be pink)

map --profile jsrun -n 2 python3 ./diffusion-fv-2d.py
Forge Follow Up Materials

ANL specific references
https://www.alcf.anl.gov/support-center/theta/arm-ddt-theta
https://www.alcf.anl.gov/support-center/theta/arm-map

Getting started videos,

Offline debugging blogs,
https://community.arm.com/developer/tools-software/hpc/b/hpc-blog/posts/debugging-while-you-sleep
https://community.arm.com/developer/tools-software/hpc/b/hpc-blog/posts/more-debugging-while-you-sleep-with-ddt

Topic specific Arm HPC webinars,

Python specific references
https://developer.arm.com/documentation/101136/2102/MAP/Python-profiling

Arm Forge Overview Recorded for the SC Student Cluster Competition
https://www.youtube.com/watch?v=Pe2WDJR2cTg&t=13s

Debugging methodology presentation at Nvidia GTC