Arm Forge
Debugging and Optimization Tools for HPC

Beau Paisley<Beau.Paisley@arm.com>
**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
HPC Development Solutions from Arm

Best in class commercially supported tools for Linux and high-performance computing

Performance Engineering

_for any architecture, at any scale_

**arm FORGE**

- **arm DDT**
- **arm MAP**
- **arm PERFORMANCE REPORTS**
- **Profiler**
- **Debugger**
- **Reporting**
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. **I/O**
   - Discover lines of code spending a long time in I/O.
   - Trace and debug slow access patterns.

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

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

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

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

9. **Verification**
   - Validate corrections and optimal performance.

**Key:**
- arm PERFORMANCE REPORTS
- arm FORGE
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 high; check the CPU performance section for advice.

- **MPI**: 0.0%
  - Time spent in MPI calls. High values are usually bad.

- **I/O**: 0.0%
  - Time spent in file system I/O. High values are usually bad.

This application ran as Compute-bound. A breakdown of this time and advice for investigating further is in the CPU Metrics section below.

As very little time was spent in MPI calls, this code may also benefit from running at larger scales.

**MPI**

- **A breakdown of the 0.0% MPI time:**
  - Time in collective calls: 0.0%
  - Time in point-to-point calls: 0.0%
  - Effective process collective rate: 0.00 bytes/sec
  - Effective process point to point rate: 0.00 bytes/sec
  - No time spent in MPI operations. There's nothing to optimize here!

**I/O**

- **A breakdown of the 0.0% I/O time:**
  - Time in reads: 0.0%
  - Time in writes: 0.0%
  - Time in seeks: 0.0%
  - Effective process read rate: 0.00 bytes/sec
  - Effective process write rate: 0.00 bytes/sec
  - No time spent in I/O operations. There's nothing to optimize here!

**OpenMP**

- **A breakdown of the 99.7% time in OpenMP regions:**
  - Computation: 99.1%
  - Synchronization: 0.1%
  - Physical core utilization: 0.3%
  - System load: 7.4%
  - Memory
    - Peak process memory usage: 312 MB
    - Peak node memory usage: 214 MB
  - Physical core utilization is low and some cores may be un-used. Try increasing `OMP_NUM_THREADS` to improve performance.

No source code needed

Less than 5% runtime overhead

Fully scalable

Run regularly – or in regression tests

Explicit and usable output

© 2019 Arm Limited
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
MAP Capabilities

- MAP is a sampling based scalable profiler
  - Built on same framework as DDT
  - Parallel support for MPI, OpenMP, CUDA
  - Designed for C/C++/Fortran

- Designed for ‘hot-spot’ analysis
  - Stack traces
  - Augmented with performance metrics

- Adaptive sampling rate
  - Throws data away – 1,000 samples per process
  - Low overhead, scalable and small file size
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)

```sh
call.map --profile jsrun -n 2 python3 ./diffusion-fv-2d.py
```
WFH Technology’, … Remote Connect

https://developer.arm.com/docs/101136/latest/arm-forge/connecting-to-a-remote-system
‘WFH Technology’, ... Offline Debugging

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