Performance Engineering for Arm Supercomputers

Linaro Connect San Diego 2019

john.linford@arm.com
What is this “supercomputer” of which you speak?

HPC: tightly coupled yet highly flexible computing systems

**Leadership-class computing systems**

- Thousands of CPU cores, 100-300 Watt SoCs
- Network fabric with record-setting bandwidth and latency
- Several megawatts, millions of $.

**The whole machine operates as one**

- Extremely low latency point-to-point communication between any two cores
- Typically the same process image runs on all cores at the same time

NASA Langley

Sandia National Laboratories
Vanguard Astra by HPE: #156 on the Top500

- 2,592 HPE Apollo 70 compute nodes
  - 5,184 CPUs, 145,152 cores, 2.3 PFLOPs (peak)
- Marvell ThunderX2 ARM SoC, 28 core, 2.0 GHz
- Memory per node: 128 GB (16 x 8 GB DR DIMMs)
  - Aggregate capacity: 332 TB, 885 TB/s (peak)
- Mellanox IB EDR, ConnectX-5
  - 112 36-port edges, 3 648-port spine switches
- Red Hat RHEL for Arm
- HPE Apollo 4520 All-flash Lustre storage
  - Storage Capacity: 403 TB (usable)
  - Storage Bandwidth: 244 GB/s
Isambard system specification

- **10,752** Armv8 cores (168n x 2s x 32c)
  - Cavium ThunderX2 32core 2.1→2.5GHz
- Cray XC50 ‘Scout’ form factor
- High-speed **Aries** interconnect
- Cray HPC optimised software stack
  - CCE, Cray MPI, math libraries, CrayPAT, ...

- **Phase 2 (the Arm part):**
  - Delivered Oct 22\(^{\text{nd}}\), handed over Oct 29\(^{\text{th}}\)
  - Accepted Nov 9\(^{\text{th}}\)
  - Upgrade to final B2 TX2 silicon, firmware, CPE completed March 15\(^{\text{th}}\) 2019

http://gw4.ac.uk/isambard/
1. High-Performance Arm CPU A64FX in HPC and AI Areas

- **Architecture features**
  - ISA: Armv8.2-A (AArch64 only) SVE (Scalable Vector Extension)
  - SIMD width: 512-bit
  - Precision: FP64/32/16, INT64/32/16/8
  - Cores: 48 computing cores + 4 assistant cores (4 CMGs)
  - Memory: HBM2: Peak B/W 1,024 GB/s
  - Interconnect: TofuD: 28 Gbps x 2 lanes x 10 ports

- **Peak performance (Chip level)**
  - **HPC**
    - A64FX (Fugaku): 21.6+ TOPS
    - SPARC64 VIIIfx (K computer): 10.8+ TOPS
  - **AI**
    - 64 bits: 0.128
    - 32 bits: 0.128
    - 16 bits: N/A
    - 8 bits: N/A

Copyright 2019 FUJITSU LIMITED
Himeno Benchmark (Fortran90)

<table>
<thead>
<tr>
<th></th>
<th>Gflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Platinum</td>
<td>85</td>
</tr>
<tr>
<td>FX100 1 CPU</td>
<td>103</td>
</tr>
<tr>
<td>Post-K 1 CPU</td>
<td>346</td>
</tr>
<tr>
<td>SX-Aurora 1 VE †</td>
<td>286</td>
</tr>
<tr>
<td>Tesla V100 1 GPU †</td>
<td>305</td>
</tr>
</tbody>
</table>

† “Performance evaluation of a vector supercomputer SX-aurora TSUBASA”, SC18, https://dl.acm.org/citation.cfm?id=3291728
Arm HPC SoC Theoretical Roofline Comparison

The diagram illustrates the theoretical peak Gflops (GFlop/s) versus arithmetic intensity (Flop/Byte) for various processors. The processors are arranged along the horizontal and vertical axes, with the theoretical peak Gflops on the vertical axis and arithmetic intensity on the horizontal axis.

Key processors and their theoretical peak Gflops include:
- Intel Skylake 8168 (2.08 TFlops)
- Marvell ThunderX2 (0.56 TFlops)
- Nvidia tesla V100 (7.5 TFlops)
- Intel KNL-7250 (3.04 TFlops)
- Fujitsu-A64FX (2.99 TFlops)
- Huawei-Kunpeng920 (1.33 TFlops)
- Marvell TX2-CN9980 (560 GFlops)
- AMD EPYC Naples/7601 (1.13 TFlops)
- AMD EPYC Rome (2.41 TFlops)
- Intel Skylake-8168 (2.08 TFlops)
- Amazon Graviton (294 GFlops)
- Intel KNL-7250 (3.04 TFlops)
- Intel Skylake-8168 (2.08 TFlops)
- Amazon Graviton (294 GFlops)
- Intel KNL-7250 (3.04 TFlops)
- Intel KNL-7250 (3.04 TFlops)
- Fujitsu-A64FX (2.99 TFlops)
- Fujitsu-A64FX (2.99 TFlops)
- Fujitsu-A64FX (2.99 TFlops)
- Fujitsu-A64FX (2.99 TFlops)
- Fujitsu-A64FX (2.99 TFlops)
Arm HPC SoC Theoretical Roofline Comparison

Theoretical Peak Gflops vs. Arithmetic Intensity (Flop/Byte)

- Fujitsu A64FX (2.99 TFlops)
- Intel Skylake 8168 (2.08 TFlops)
- Marvell ThunderX2 (0.56 TFlops)
HPC Software Environment Characteristics
Familiar software stacks at extreme scales with strong legacy support

Your typical HPC environment

**Specialized Linux Environment**
- RHEL, SLES, or Debian derivative

**Legacy Support**
- Fortran (even FORTRAN)
- Message Passing Interface (MPI)

**Queue system, few users, no root**
- Slurm, PBS, Torque, etc.
- Some/all of the machine is allocated to one user

HPC software trends

**Python rapidly becoming mainstream**

**Containers gaining popularity**

**Many new codes are not written in Fortran**

**Enormous interest in ML**
- E.g. apply high resolution solution to a reduced search space

**Things you won’t usually see**
Java, Ruby, Rust, Nginx, MS Windows, macOS, virtual machines, ...
Silicon Suppliers
Marvell, Fujitsu, Mellanox, NVIDIA, ...

OEM/ODM’s
Cray, HPE, ATOS-Bull, Fujitsu, Gigabyte, ...

Applications
Open-source, owned, commercial ISV codes, ...

Containers, Interpreters, etc.
Singularity, PodMan, Docker, Python, ...

Performance Engineering
Arm Forge (DDT, MAP), Rogue Wave, HPC Toolkit, Scalasca, Vampir, TAU, ...

Middleware
Mellanox IB/OFED/HPC-X, OpenMPI, MPICH, MVAPICH2, OpenSHMEM, OpenUCX, HPE MPI

Compilers
Arm, GNU, LLVM, Clang, Flang, Cray, PGI/NVIDIA, Fujitsu, ...

Libraries
ArmPL, FFTW, OpenBLAS, NumPy, SciPy, Trilinos, PETSc, Hypre, SuperLU, ScaLAPACK, ...

Filesystems
BeeGFS, Lustre, ZFS, HDF5, NetCDF, GPFS, ...

Schedulers
SLURM, IBM LSF, Altair PBS Pro, ...

Cluster Management
Bright, HPE CMU, xCat, Warewulf, ...

OS
RHEL, SUSE, CentOS, Ubuntu, Debian, ...

Arm Server Ready Platform
Standard firmware and RAS
A Rich and Growing Application Ecosystem
Integrating HPC into the Arm Ecosystem

Open standards, open source
Performance Engineering
Methodology and Tools
Arm Allinea Studio: HPC Development Solutions from Arm

Best in class “Arm on Arm”, commercially supported tools for Linux 64-bits

**Code Generation**
*for Arm servers*

**Performance Engineering**
*for any architecture, at any scale*

**Server & HPC Solution**
*for Arm servers*

---

**Arm Compiler for Linux**
- arm C/C++ Compiler
- arm FORTRAN Compiler
- arm PERFORMANCE LIBRARIES

**Arm FORGE**
- arm DDT
- arm MAP
- arm PERFORMANCE REPORTS

**Commercially Supported Toolkit**
for applications development on Linux

- C/C++ Compiler for Linux
- Fortran Compiler for Linux
- Performance Libraries
- Performance Reports
- Debugger
- Profiler

© 2019 Arm Limited
Step 0: Compile and Link
Arm’s commercially-supported C/C++/Fortran compiler

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
Arm Compiler: 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.
  • Two different vectorizers from LLVM: Loop Vectorizer and SLP Vectorizer.

• Loop Vectorizer support for NEON (e.g. 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
Arm’s Optimized OpenMP Runtime

Arm actively optimizes runtime libraries for high thread counts

- Large System Extension (LSE) atomic update instructions
  - Used extensively in the OpenMP runtime shipped with the Arm HPC Compiler.
  - Also available in GNU’s runtime.

- Atomics dramatically reduce runtime overhead, especially at high thread counts.
  - Designed with hundreds of threads in mind.
  - Uses hardware features whenever available.

- Synchronization constructs optimized for high thread counts.
  - Also available in GNU’s runtime.

![Graph showing performance of Lulesh with different compilers](image)
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
- Arm’s commercial compiler vs GCC:
  - Response time to customer needs
  - Optimization for partner SoCs
  - Commercial support
Optimized BLAS, LAPACK and FFT

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
Arm Performance Libraries
Optimized BLAS, LAPACK and FFT

DGEMM: ArmPL vs BLIS vs OpenBLAS

3D Complex/Complex FFT: ArmPL vs FFTW
Optimized Math Routines

Included with the Arm Performance Libraries – automatically linked with Arm Compiler

Normalised runtime

ArmPL includes libamath

- Algorithmically better performance than standard library calls
- No loss of accuracy
- Single and double precision implementations of: `exp()`, `pow()`, `log()`, `sin()`, `cos()`, ...
- To use with Arm Compiler:
  - Just compile! Included automatically
- To use with GCC:
  - Load ArmPL module and add “-lamath” to the link line.
Step 1: Debug
Arm Forge = DDT + MAP + Performance Reports
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
DDT Feature Highlights

Switch between MPI ranks and OpenMP threads

Display pending communications

Visualise data structures

Connect to continuous integration
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
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
MVAPICH2 Compatibility Hint
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.

When manual linking is used, untick “Preload” box
Step 2: Characterize Performance
Identifying and resolving performance issues

Profile

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

Identify Hotspots

File I/O

Yes

Memory

Yes

Compute

Yes

Communication

Yes

© 2019 Arm Limited

33
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
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
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
Python Profiling

From 19.1

New support for Python applications

- Native Python
- Cython Interpreter
- Called C/C++ code
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>Resource</th>
<th>Percentage</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>20.6%</td>
<td>Green</td>
</tr>
<tr>
<td>MPI</td>
<td>63.2%</td>
<td>Blue</td>
</tr>
<tr>
<td>I/O</td>
<td>16.2%</td>
<td>Red</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.

**Compute**

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

**MPI**

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

**I/O**

Time spent in filesystem I/O. High values are usually bad. This is **average**; check the I/O breakdown section for optimization advice.
Arm Performance Reports Metrics
Lowers expertise requirements by explaining everything in detail right in the report.
The Arm Forge GUI and where to run it

Forge includes a powerful GUI that can be run in a variety of configurations.

- **Remote client** (remote launch + reverse connect)
- **On the head node** (interactive mode + reverse connect)
- **On the compute node** (offline OR interactive mode)
- Ultimately, that’s where the tools will run. But what about the GUI?
DDT somewhere over the Pacific at 41,000ft and 550MPH
Performance Engineering for SVE
SVE: The Scalable Vector Extension

The vector length is **LEN x 128-bit** up to 2048

- There is no preferred vector length
- No need to recompile
- No need to rewrite hand-coded SVE assembler or C intrinsics
- The programmer’s intent is expressed in the binary ➔ easier to optimize
- **Predicate Registers** indicate active vector lanes

![Diagram showing vector lengths and predicate registers](image)
How can you program when the vector length is unknown?

SVE provides features to enable VLA programming from the assembly level and up

```
+  
    pred
 1 2 3 4
5 5 5 5
1 0 1 0
= 6 2 8 4
```

**Per-lane predication**
Operations work on individual lanes under control of a predicate register.

```plaintext
for (i = 0; i < n; ++i)
INDEX i n-2 n-1 n n+1
CMPLT n 1 1 0 0
```

**Predicate-driven loop control and management**
Eliminate scalar loop heads and tails by processing partial vectors.

```
+  
    pred
 1 2 0 0
1 1 0 0
```

**Vector partitioning & software-managed speculation**
First Faulting Load instructions allow memory accesses to cross into invalid pages.
SVE vs Traditional ISA
How do we compute data which has ten chunks of 8-bytes?

**Aarch64 (scalar)**
- Ten iterations over an 8-byte register

**NEON (128-bit vector engine)**
- Four iterations over a 16-byte register + two iterations of a drain loop over a 8-byte register

**SVE (VLA vector engine)**
- Three iterations over a 32-byte VLA register with an adjustable predicate
SVE Programming Approaches
Libraries > Auto-vectorization > compilers directives > intrinsics > assembly

• Libraries:
  • ArmPL 19.3 supports SVE!

• Compilers:
  • Auto-vectorization: GCC, Arm Compiler for HPC, Cray, Fujitsu
  • Compiler directives, e.g. OpenMP
    – #pragma omp parallel for simd
    – #pragma vector always

• Intrinsics:
  • Arm C Language Extensions for SVE
  • Arm Scalable Vector Extensions and Application to Machine Learning

• Assembly:
  • Full ISA Specification: The Scalable Vector Extension for Armv8-A
  • Lots of worked examples: A Sneak Peek Into SVE and VLA Programming
## SVE Compiler Support:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Assembly / Disassembly</th>
<th>Inline Assembly</th>
<th>ACLE</th>
<th>Auto-vectorization</th>
<th>Math Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm Compiler for Linux</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2</td>
<td>SVE</td>
</tr>
<tr>
<td>LLVM/Clang</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2 in LLVM 10</td>
<td>SVE + SVE2 in LLVM 11</td>
<td></td>
</tr>
<tr>
<td>GNU</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2</td>
<td>SVE + SVE2 in GNU 10</td>
<td>SVE now SVE2 in GNU10</td>
<td></td>
</tr>
</tbody>
</table>

Note: Yellow boxes are planned and work in progress. Timelines may change.
Targeting SVE with both Arm compiler and GNU (8+)

• Compilation targets a specific architecture based on an architecture revision
  • -mcu=native -march=armv8.1-a+sve
    – Learn more: https://community.arm.com/.../compiler-flags-across-architectures-march-mtune-and-mcpu
  • -march=armv8-a
    • Target V8-a
    • Will generate NEON instructions
    • No SVE
  • -march=armv8-a+sve
    • Will add SVE instruction generations
• Check the assembly (-S)
  • armclang++ -S -o code.s -Ofast -g -march=armv8-a+sve code.cpp
  • gcc++ -S -o code.s -Ofast -g -march=armv8-a+sve code.cpp
Running and Analyzing SVE Binaries with ArmIE
Arm Instruction Emulator

Develop your user-space applications for future hardware today

Start porting and tuning for future architectures early

- Reduce time to market
- Save development and debug time with Arm support

Run 64-bit user-space Linux code that uses new hardware features on current Arm hardware

- SVE support available now. Support for 8.x planned.
- Tested with Arm Architecture Verification Suite (AVS)

Near native speed with commercial support

- Integrates with DynamoRIO allowing arbitrary instrumentation extension
- Emulates only unsupported instructions
- Integrated with other commercial Arm tools including compiler and profiler
- Maintained and supported by Arm for a wide range of Arm-based SoCs
Arm Instruction Emulator

Run SVE binaries on today’s CPUs

• Simple tool aimed at software developers
  
  $ armclang hello.c --march=armv8.1-a+sve
  $ ./a.out
  Illegal instruction
  $ armie --msve-vector-bits=256 -- ./a.out
  Hello

• Runs applications at near-native speed
  • Supports dynamic loading and JIT
  • Runs multithreaded applications
  • Transparent to system calls

• Intercepts and emulates use of Arm instructions newer than hardware

Converting unsupported instructions to native Armv8-A instructions
Why ArmIE?
And not <insert tool here>? 

Because ArmIE is:

✓ Fast functional emulator  
  (enables apps with large inputs runs)

✓ Easy to use and develop  
  (allows custom instrumentation and post-processing)

✓ Freely available

✓ Partly open-source  
  (API to build your own instrumentation)

ArmIE is not:

✗ Cycle accurate  
  (no timing information)

✗ A simulator  
  (Requires Armv8 hardware)

✗ Architecture modelling  
  (Is all about the apps)
Instrumenting AArch64 and SVE

DynamoRIO

ArmIE (emulation client)

Armv8-A + SVE Binary

SVE Memtrace Client

SVE Inscount Client

Opcodes Client

SVE custom clients

Emulation API
Types of data we could extract from ArmIE

- Total instruction count*
- SVE/NEON instruction count*
- Memory traces, including gather/scatter ops, with memory addresses
- Average lane utilization in bytes
- Cache statistics based on simple cache model:
  - L1, L2 hit rates, average cycles/access
- These can be obtained per whole application/ROI/function
  - Use of Region-of-Interest (RoI) markers in the code to delimit instrumentation

* All instruction counts are dynamic.

#define __START_TRACE() { asm volatile (".inst 0x2520e020");}
define __STOP_TRACE() { asm volatile (".inst 0x2520e040");}

Resources to help get you started

Communications and Collaborations

gitlab.com/arm-hpc
Gitlab HPC Packages Wiki

arm-hpc.groups.io
AHUG Collaboration Site

developer.arm.com/hpc
Arm Official HPC Marketing Page

community.arm.com/tools/HPC
HPC Blogs and Official Forum

System Access

https://gw4.ac.uk/isambard/
Isambard @ Bristol

https://arm-hpc.groups.io/g/catalyst
Catalyst @ EPCC, Leicester, Bristol
Los Alamos National Laboratory Asteroid Killer Simulation

https://youtu.be/T1-yoiE6U5c?t=72
Thanks!
Arm’s HPC Professional Services

*Porting, Tuning, Training, and Enablement*

- Application performance engineering
  - Get help optimizing for maximum performance
- New system tuning
  - Tune HPC system parameters for your workloads
- Hackathons and tutorials
  - Education, mentoring, and hands-on events to help jumpstart HPC developers

Olly Perks
Fabrice Dupros
Phil Ridley
John Linford
Srinath Vadlamani
Nick Forrington
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 ...
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 -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)
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.
SVE Resources
http://developer.arm.com/hpc

- **Porting and Optimizing Guides**
  - For SVE: [https://developer.arm.com/docs/101726/0110](https://developer.arm.com/docs/101726/0110)
  - For Arm in general: [https://developer.arm.com/docs/101725/0110](https://developer.arm.com/docs/101725/0110)

- **The SVE Specification**

- **ACLE References and Examples**
  - ACLE for SVE: [https://developer.arm.com/docs/100987/latest](https://developer.arm.com/docs/100987/latest)
  - Worked examples: [A Sneak Peek Into SVE and VLA Programming](https://developer.arm.com/docs/100987/latest)
  - Optimized machine learning: [Arm SVE and Applications to Machine Learning](https://developer.arm.com/docs/100987/latest)