Debugging and Profiling your HPC Applications

Srinath Vadlamani, Field Application Engineer

srinath.vadlamani@arm.com
About this talk

Techniques not tools

• Learn ways to debug and profile your code

Use tools to apply techniques

• Debugging with Allinea DDT
• Benchmarking with Allinea Performance Reports
• Profiling with Allinea MAP

• Go to www.allinea.com/trials

Tools are available on the ATPESC machines
Motivation

HPC systems are finite

- Limited lifetime to achieve most science possible
- Sharing a precious resource means your limited allocation needs to be used well

Your time is finite

- PhD to submit
- Project to complete
- Paper to write
- Career to develop

Doing good things with HPC means creating better software, faster

- Being smart about what you’re doing
- Using the tools that help you apply smart techniques

Aug. 8, ATPESC_2017
Real-world example

**Bioinformatics**
Discover Assembly
3x speedup
EC2

**Deep Learning**
Torch + DeepMind
5.3x speedup
Intel Xeon Phi (KNL)

**Fluid Dynamics**
HemeLB blood flow
16.8x capability boost
50k core crash fixed

Aug. 8, ATPESC_2017
Debugging in practice...

1. Compile
2. Run
3. Crash
4. Hypothesis
5. Insert print statements

Aug. 8, ATPESC_2017
Optimization in Practice

Change code

Run code

Analyze performance result

Aug. 8, ATPESC_2017
About those techniques…

“No-one cares how quickly you can compute the wrong answer”

• Old saying of HPC performance experts

Let’s start with debugging then…
## Some types of bug

<table>
<thead>
<tr>
<th>Bug</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bohrbug</td>
<td>Steady, dependable bug</td>
</tr>
<tr>
<td>Heisenbug</td>
<td>Vanishes when you try to debug (observe)</td>
</tr>
<tr>
<td>Mandelbug</td>
<td>Complexity and obscurity of the cause is so great that it appears chaotic</td>
</tr>
<tr>
<td>Schroedinbug</td>
<td>First occurs after someone reads the source file and deduces that the code should have never worked, after which the program ceases to work until fixed</td>
</tr>
</tbody>
</table>
Debugging

The art of transforming a broken program to a working one:

Debugging requires thought – and discipline:

• Track the problem
• Reproduce
• Automate – (and simplify) the test case
• Find origins – where could the “infection” be from?
• Focus – examine the origins
• Isolate – narrow down the origins
• Correct – fix and verify the testcase is successful

Suggested Reading:


What you will read:

• Crowd sources like stack overflow

Aug. 8, ATPESC_2017
Popular techniques

- **Automation**
  - Test cases
  - Bisection via version control

- **Observation**
  - Print statements
  - Debuggers

- **Inspiration**
  - Explaining the source code to a duck

- **Magic**
  - Static analysis
  - Memory debugging

Aug. 8, ATPESC_2017
Solving Software Defects

Who had a rogue behavior?
- 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

Aug. 8, ATPESC_2017
Favorite Allinea DDT Features for Scale

- Parallel stack view
- Automated data comparison: sparklines
- Parallel array searching
- Step, play, and breakpoints
- Offline debugging

Aug. 8, ATPESC_2017
6 steps to help improve performance

1. Get a realistic test case
2. Profile your code
3. Look for the significant
4. What is the nature of the problem?
5. Apply brain to solve
6. Bottle It

Logging like an experiment is useful.
Bottling it…

• Lock in performance once you have won it
• Save your nightly performance
• Tie your performance results to your continuous integration server

• Lock in the bug fixes
• Save the test cases
• Tie the test cases to your continuous integration server

• Regression tests do help you from regressing!!!
PERFORMANCE ROADMAP

Improving the efficiency of your parallel software holds the key to solving more complex research problems faster. This pragmatic, step-by-step guide will help you to identify and focus on bottlenecks and optimizations at a time with an emphasis on measuring and understanding before rewriting.

1. IMPROVE MEMORY ACCESS PATTERNS
   - Many real-world parallel applications suffer from memory access issues.
   - Common Problems:
     - Improper memory allocation.
     - Inefficient cache utilization.
   - Tools for Success:
     - Use tools like profilers to identify memory access patterns.
     - Optimize memory access patterns using cache-friendly algorithms.

2. EXAMINE I/O
   - Does your application spend a significant amount of time on I/O?
   - Common Problems:
     - Overhead in input/output
     - Inefficient file formatting.
   - Tools for Success:
     - Use profiling tools to identify I/O bottlenecks.
     - Optimize I/O operations using efficient file formats.

3. REVIEW COMMUNICATION
   - Is your program spending too much time in communication?
   - Common Problems:
     - High communication overhead.
     - Inefficient data partitioning.
   - Tools for Success:
     - Use profiling tools to identify communication bottlenecks.
     - Optimize communication patterns using efficient algorithms.

4. BALANCE WORKLOAD
   - Is your program spending too much time in low-bandwidth communication and synchronization?
   - Common Problems:
     - Inefficient data distribution.
     - Uneven workload distribution.
   - Tools for Success:
     - Use profiling tools to identify load imbalance.
     - Optimize workload distribution using efficient algorithms.

5. USE MULTIPLE CORES
   - Using multiple cores can significantly improve performance.
   - Common Problems:
     - Inefficient use of cores.
     - Overhead in thread creation.
   - Tools for Success:
     - Use synchronization and communication tools to optimize core utilization.
     - Optimize data partitioning using efficient algorithms.

6. ANALYZE BEFORE YOU OPTIMIZE
   - Measure performance before optimization.
   - Common Problems:
     - Inefficient use of resources.
     - Overhead in measurement.
   - Tools for Success:
     - Use profiling tools to measure performance before optimization.
     - Optimize performance using efficient algorithms.

7. VECTORIZE / OFFLOAD HOT LOOPS
   - Vectorizing and offloading hot loops can significantly improve performance.
   - Common Problems:
     - Inefficient use of vector instructions.
     - Overhead in offloading.
   - Tools for Success:
     - Use vectorizing compilers to optimize hot loops.
     - Offload computations to accelerators using efficient APIs.

Contact us:
- www.allinea.com
- +1 (408) 600 2215
- sales@allinea.com
- Allinea Software Inc, 3303 Alameda Place, Suite 700, San Jose CA 95110
How The Tools Fit...

Demand for software efficiency

Demand for developer efficiency

Demand for performance optimization

Leads to MAP to optimize performance

Demand for debugging

Performance Reports
Measure

Forge

MAP
Profile and Optimize

Debug, optimize, edit, commit, build, repeat...

DDT
Debug

Pull for MAP to develop performance fix

Pull for MAP to develop performance fix

Leads to DDT to understand and fix

Open Interfaces (eg. JSON APIs)

Continuous Integration

Version Control

Demand for performance optimization

Demand for debugging

Demand for software efficiency

Aug. 8, ATPESC_2017
How to help scientific developers best?

You *can* teach a man to fish
But first he must realize **he is hungry**
Communicate the benefits of optimization

Show, don’t tell…

CPU

A breakdown of the 84.4% CPU time:

Scalar numeric ops 27.4%
Vector numeric ops 0.0%
Memory accesses 72.6%
Waiting for accelerators 0.0%

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

No time is spent in vectorized instructions. Check the compiler’s vectorization advice to see why key loops could not be vectorized.

… this is your code on “–O0”, ie. no optimizations
Show performance they understand

**CPU**

A breakdown of the **88.5%** CPU time:

- Single-core code: 100.0%
- **Scalar numeric ops**: 22.4%
- Vector numeric ops: 0.0%
- Memory accesses: 77.6%

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

No time is spent in **vectorized instructions**. Check the compiler’s vectorization advice to see why key loops could not be vectorized.
Communicating at the right level

Out-of-order  Pipelined  Time per retired instruction
Explaining performance at the right level

CPU

A breakdown of the 88.5% CPU time:

- Single-core code: 100.0%
- Scalar numeric ops: 22.4%
- Vector numeric ops: 0.0%
- Memory accesses: 77.6%

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

No time is spent in vectorized instructions. Check the compiler’s vectorization advice to see why key loops could not be vectorized.

+ simple, actionable advice

Compiler advice is your friend.

Aug. 8, ATPESC_2017
**Vectorization, MPI, I/O, memory, energy…**

**CPU**
A breakdown of the 88.5% CPU time:
- Single-core code: 100.0%
- Scalar numeric ops: 22.4%
- Vector numeric ops: 0.0%
- Memory accesses: 77.6%

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

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

**I/O**
A breakdown of the 0.0% I/O time:
- Time in reads: 0.0%
- Time in writes: 0.0%
- Effective process read rate: 0.00 bytes/sec
- Effective process write rate: 0.00 bytes/sec

No time is spent in I/O operations. There's nothing to optimize here.

**Memory**
Per-process memory usage may also affect scaling:
- Mean process memory usage: 49.7 MB
- Peak process memory usage: 53.6 MB
- Peak node memory usage: 24.0%

The peak node memory usage is very low. You may be able to reduce the amount of allocation time used by running with fewer MPI processes and more data on each process.

**Energy**
A breakdown of how the total 58.0 J energy was spent:
- CPU: 30.0 J
- Accelerators: 0.0 J
- Peak power: 23.00 W
- Mean power: 13.80 W

The CPU is responsible for all measured energy usage. Check the CPU breakdown section to see if it is being well-used.

No system-level measurements were available on this run.

---

**Aug. 8, ATPESC_2017**
Accelerator support...

Summary: miniMD_nvidia is Compute-bound in this configuration

- **Compute**: 89.8%
- **MPI**: 10.1%
- **I/O**: 0.1%

Time spent running application code. High values are usually good. This is high, check the CPU and accelerator sections for optimization advice. Time spent in MPI calls. High values are usually bad. This is very low, this code may benefit from increasing the process count. Time spent in lqsys I/O. High values are usually bad. This is very low, however single-process I/O often causes large MPI wait times.

This application run was compute-bound. Investigate further with the CPU and accelerator sections below. As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

**CPU**
A breakdown of the 89.8% CPU time:
- Waiting for accelerators: 21.1%
- Scalar numeric ops: 15.0%
- Vector numeric ops: 0.0%
- Memory accesses: 62.9%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance. No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

**I/O**
A breakdown of the 0.1% I/O time:
- Time in reads: 100.0%
- Time in writes: 0.0%
- Effective process read rate: 12.7 MiB/s
- Effective process write rate: 0.0 MiB/s

Most of the time is spent in read operations with a low effective transfer rate. This may be caused by contention for the I/O system or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

**Memory**
Per-process memory usage may also affect scaling:
- Mean process memory usage: 200 MiB
- Peak process memory usage: 212 MiB
- Peak node memory usage: 6.0%

The peak node memory usage is very low. You may be able to reduce the amount of allocation time used by running with fewer MPI processes and more data on each process.

**MPI**
A breakdown of the 10.1% MPI time:
- Time in collective calls: 31.4%
- Time in point-to-point calls: 68.6%
- Effective process collective rate: 859 bytes/s
- Effective process point-to-point rate: 573 MiB/s

Most of the time is spent in point-to-point calls with an average transfer rate. Using larger messages and overlapping communication and computation may increase the effective transfer rate. The collective transfer rate is very low. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate further.

**Threads**
A breakdown of how multiple threads were used:
- Computation: 62.1%
- Synchronization: 31.9%
- Physical core utilization: 16.7%

Synchronization time is spent synchronizing threads. Check which locks cause the most overhead with a profiler.

**Accelerators**
A breakdown of how accelerators were used:
- Compute utilization: 93.4%
- Memory utilization: 20.4%
- Device memory used: 98.4%

High compute and low memory utilization suggests compute throughput is limiting GPU performance. Use a profiler to find the fastest kernels and check them for divergent branches and over-subscribed function units.
Application Development Workflow

- Coding
- Profiling
- Optimization
- Debugging
- Execution

Aug. 8, ATPESC_2017
Hello Allinea Forge!

- Allinea MAP to find performance bottleneck
- Increasing memory usage? *Memory leak!*
  Workload imbalance? *Possible partitioner bug!*
- Flick to Allinea DDT
  Common interface and settings files
- Observe and debug your code step by step

Aug. 8, ATPESC_2017
HPC means being productive on remote machines

- Linux
- OS/X
- Windows
- Multiple hop SSH
- RSA + Cryptocard
- Uses server license
MAP in a nutshell

- Small data files
- <5% slowdown
- No instrumentation
- No recompilation
Above all…

Aimed at any performance problem that matters

- MAP focuses on time

Does not prejudge the problem

- Doesn’t assume it’s MPI messages, threads or I/O

If there’s a problem..

- MAP shows you it, next to your code
Scaling issue – 512 processes

Simple fix... reduce periodicity of output
Deeper insight into CPU usage

Runtime of application still unusually slow

Allinea MAP identifies vectorization close to zero

Why? Time to switch to a debugger!

Aug. 8, ATPESC_2017
While still connected to the server we switch to the debugger.
It’s already configured to reproduce the profiling run
Today’s Status on Scalability

Debugging and profiling

• Active users at 100,000+ cores debugging
• 50,000 cores was largest profiling tried to date (and was Very Successful)
• ... and active users with just 1 process too

Deployed on

• NERSC Cori, ORNL’s Titan, NCSA Blue Waters, ANL Mira etc.
• Hundreds of much smaller systems – academic, research, oil and gas, genomics, etc.

Tools help the full range of programmer ambition

• Very small slow down with either tool (< 5%)
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
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

Aug. 8, ATPESC_2017
Getting started on Theta

Install local client on your laptop

- www.allinea.com/products/forge/downloads
  - Linux – installs full set of tools
  - Windows, Mac – just a remote client to the remote system

- Run the installation and software

- “Connect to remote host”

- Hostname:
  - username@theta.alcf.anl.gov

- Remote installation directory: /soft/debuggers/forge-7.0.6-2017-08-07/

- Click Test

Congratulations you are now ready to debug Theta.
Hands on Session

Use Allinea DDT on your favorite system to debug your code – or example codes

Use Allinea MAP or Performance Reports on Cooley to see your code performance

Use Allinea DDT and Allinea MAP together to improve our test code

• Download examples from www.allinea.com - Trials menu, Resources – “trial guide”
Thank you for your attention!

Contact:

• support@allinea.com
• support@arm.com

Download a trial for ATPESC (or later)

• http://www.allinea.com/trials