Debugging and Profiling your HPC Applications
David Lecomber, CEO and Co-founder
david@allinea.com
About this talk

• Techniques not tools
  – Learn how to debug and profile your code

• Use tools to apply techniques
  – Debugging
    • Allinea DDT (BlueGene/Q and Linux)
  – Performance
    • Benchmarking with Allinea Performance Reports (Linux)
    • Profiling with Allinea MAP (Linux)

• 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
Debugging in practice...

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

- Change code
- Run code
- Analyze result
Motivation

- “Without capable highly parallel software, large supercomputers are less useful”
  – Council on Competitiveness

- “1% of HPC application codes can exploit 10,000 cores”
  – IDC, 2011
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 it never worked, after which the program ceases to work</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:
  – Zen and the Art of Motorcycle Maintenance, Robert M. Pirsig
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
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

---

**Run with Allinea tools**

**Identify a problem**

**Gather info Who, Where, How, Why**

**Fix**
Live demo
Favorite Allinea DDT Features for Scale

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

"Premature optimization is the root of all evil"

Donald Knuth, 1974
Profiling for performance

- Code optimization can be time-consuming...

![Diagram showing time cost comparison between Strategy A and Strategy B.](image courtesy of xkcd.com)
6 steps to 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
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 identify and focus on bottlenecks and optimizations one at a time with an emphasis on measuring and understanding before rewriting.

1. **ANALYZE BEFORE YOU OPTIMIZE**
   - Measure all performance aspects
   - You can’t fix what you can’t see
   - Prefer real workloads over artificial tests

   **TOOLS FOR SUCCESS:**
   - Allinea Performance Reports does this quickly and easily

2. **EXAMINE I/O**
   - Does the application spend significant time in I/O?
   - Common Problems:
     - Checkpointing too often
     - Many small reads and writes
     - Data in home directory instead of scratch
     - Multiple nodes using filesystem at the same time

   **TOOLS FOR SUCCESS:**
   - Allinea Forge highlights lines of code spending a long time in I/O
   - Trace and debug suspicious or slow access patterns using Allinea Forge

3. **BALANCE WORKLOAD**
   - Spending a lot of time in low-bandwidth communication and synchronization?
   - Common Problems:
     - Dataset too small to run efficiently at this scale
     - I/O contention causing late sender
     - Bug in work partitioning code

   **TOOLS FOR SUCCESS:**
   - Performance Reports detects balance issues
   - Allinea Forge identifies slow communication calls and processes
   - Dive into partitioning code with integrated debugger in Allinea Forge

4. **REVIEW COMMUNICATION**
   - Lots of time in medium/high-bandwidth communication?
   - Common Problems:
     - Short high-frequency messages are very sensitive to latency
     - Too many synchronizations
     - No overlap between communication and computation

   **TOOLS FOR SUCCESS:**
   - Allinea Performance Reports tracks communication performance
   - Allinea Forge shows which communication calls are slow and why

5. **IMPROVE MEMORY ACCESS PATTERNS**
   - Many real codes are memory-bound; is this one?
   - Common Problems:
     - Initializing memory on one core but using it on another
     - Arrays of structures causing inefficient cache utilization
     - Caching results when recomputation is cheaper

   **TOOLS FOR SUCCESS:**
   - Allinea Forge shows lines of code bottlenecked by memory access times
   - Trace allocation and use of hot data structures in Allinea Forge debugger

6. **USE MULTIPLE CORES**
   - Using processes for physical cores, threads for logical cores?
   - Common Problems:
     - Implicit thread barriers inside tight loops
     - Significant core idle time due to workload imbalance
     - Threads migrating between cores at runtime

   **TOOLS FOR SUCCESS:**
   - Allinea Performance Reports shows synchronization overhead and core utilization
   - Allinea Forge highlights synchronization-heavy code and implicit barriers

7. **VECTORIZE/OFFLOAD HOT LOOPS**
   - High floating point usage but getting low vectorization score?
   - Common Problems:
     - Expecting compilers to perform magic or using the wrong compiler flags
     - Numerically-intensive loops with hard to vectorize patterns
     - Using routines that have faster vendor-provided equivalents in highly-optimized math libraries

   **TOOLS FOR SUCCESS:**
   - Allinea Performance Reports shows numerical intensity and level of vectorization
   - Allinea Forge shows hot loops, unvectorized code and GPU performance

FINISH
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
How The Tools Fit…

Demand for software efficiency → Performance Reports → Forge → MAP (Profile and Optimize) → DDT (Debug) → Demand for debugging

Demand for developer efficiency → Forge

Demand for performance optimization → MAP → DDT → Pull for MAP to develop performance fix

Open Interfaces (e.g., JSON APIs) → Continuous Integration

Version Control

Leads to MAP to optimize performance

Debug, optimize, edit, commit, build, repeat… Leads to DDT to understand and fix
How to help scientific developers best?

You can teach a man to fish
But first he must realize he is hungry

Image © Kanani CC-BY
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
Show performance they understand

**CPU**

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Percentage</th>
<th>Bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-core code</td>
<td>100.0%</td>
<td></td>
</tr>
<tr>
<td>Scalar numeric ops</td>
<td>22.4%</td>
<td></td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td><strong>0.0%</strong></td>
<td></td>
</tr>
<tr>
<td>Memory accesses</td>
<td>77.6%</td>
<td></td>
</tr>
</tbody>
</table>

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.

“Vectorization, how does it work?”
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:

<table>
<thead>
<tr>
<th>Component</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-core code</td>
<td>100.0%</td>
</tr>
<tr>
<td>Scalar numeric ops</td>
<td>22.4%</td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td>0.0%</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>77.6%</td>
</tr>
</tbody>
</table>

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
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.

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.
Application Development Workflow

- Profiling
- Optimization
- Execution
- Debugging
- Coding
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
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!
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 is largest profiling tried to date (and was Very Successful)
  – … and active users with just 1 process too

• **Deployed on**
  – 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
Getting started on Mira/Cooley

• Install local client on your laptop
    • 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@cetus.alcf.anl.gov
    • username@cooley.alcf.anl.gov
  – Remote installation directory: /soft/debuggers/ddt
  – Click Test

• Congratulations you are now ready to debug on Mira/Vesta/Cetus – or debug and profile on Cooley.
Using the Performance Reports on Cooley

There is no GUI – command line only

Usual command:

```
mpirun -np 4 a.out
```

Becomes:

```
/soft/debuggers/allinea-reports-6.0.6-2016-08-03/bin/perf-report -np 4 a.out
```

Email yourself the “.html” file at the end:

```
mail -a {report.html} me@gmail.com
```
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

• How much speed up can you get?
Thanks for watching!

• Contact:
  – david@allinea.com

• Download a trial for ATPESC (or later)
  – http://www.allinea.com/trials