Debugging and Profiling your HPC Applications

David Lecomber, CEO and Co-founder
david@allinea.com
About this talk

• Learn how to debug and profile your code
  – Techniques to take home

• Tools we will use: Allinea Forge
  – Debugging with Allinea DDT
  – Profiling with Allinea MAP
  – NB. Allinea MAP is not supported on BG/Q

• Where to find Allinea’s tools
  – > 70% of Top 500 have at least one Allinea tool
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
  – Unrivaled productive and easy-to-use development environment…
  – … To help reach the highest level of performance and scalability

• High performance parallel code needs tools designed for the challenge
Use the right software tool to be faster

- What parts of the code would benefit most from being rewritten?
- How should I modify a code to make it better (or work at all)?
Debugging in practice...

Compile → Run → Crash → Insert print statements → Hypothesis → Compile
Optimization in Practice

Insert timers

Run code

Change code

Analyse 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
Application Development Workflow

- Coding
- Profiling
- Optimization
- Debugging
- Execution
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
Profiling for performance

- Code optimisation can be time-consuming...

  ![Diagram showing time cost comparison between Strategy A and Strategy B. Strategy B is more efficient. The reason I am so inefficient.](image courtesy of xkcd.com)
6 steps to improve performance

Get a realistic test case
- Performance on real data matters
- Keep the test case for reference and re-use

Profile your code
- Add "-g" flag to your compilation
- Run "map –profile mpirun –np 4 application.exe"

Look for the significant
- Which part/phase of the code dominates time?
- Is there any unexpected significant time use?

What is the nature of the problem?
- Compute? I/O? MPI? Thread synchronization?
- Display the metrics that show the problem best

Apply brain to solve
- MPI – can you balance the work better?
- Compute – is memory time dominant – can you improve layout?

Think of the future
- Try larger process or thread counts to watch for scalability problems
- Keep the profile (.map file) for future comparison
MAP in a nutshell

- Small data files
- <5% slowdown
- No instrumentation
- No recompilation
How Allinea MAP is different

Adaptive sampling
- Sample frequency decreases over time
- Data never grows too much
- Run for as long as you want

Scalable
- Same scalable infrastructure as Allinea DDT
- Merges sample data at end of job
- Handles very high core counts, fast

Instruction analysis
- Categorizes instructions sampled
- Knows where processor spends time
- Shows vectorization and memory bandwidth

Thread profiling
- Core-time not thread-time profiling
- Identifies lost compute time
- Detects OpenMP issues

Integrated
- Part of Forge tool suite
- Zoom and drill into profile
- Profiling within your code
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!
<table>
<thead>
<tr>
<th>Bug Type</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

• Transforming a broken program to a working one

• How?
  – 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
Print statement debugging

• The first debugger: print statements
  – Each process prints a message or value at defined locations
  – Diagnose the problem from evidence and intuition

• A long slow process
  – Analogous to bisection root finding

• Broken at modest scale
  – Too much output – too many log files
While still connected to the server we switch to the debugger
It’s already configured to reproduce the profiling run.
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
HPC could be brain surgery

- **Brain aneurysms**
  - 2-5% of population – most are undiagnosed
  - 30,000 rupture in US each year – 40% fatal
  - Early discovery and treatment increases survival rates

- **Neurosurgery as HPC**
  - MRI provides the blood vessel structure
  - Intra-cranial blood flow and pressures is just complex CFD
  - Full brain 3D model is 2-10GB geometry

- **Individualized HPC**
  - Patient’s MRI scan enables surgical decision: whether to operate, how to operate, …
  - Circle of Willis requires super-Petascale machine software
  - Need answer in minutes or hours

- ... but it crashes at 49152 cores
Run at problem size (49,152 processes)

Ah… Integer overflow!
Debugging by Inspiration

• Some errors are harder than others to diagnose
  – A bug that occurs 1% of the time is many times harder to fix than one that occurs 100% of the time
  – Often caused by incorrect memory usage

• Get help from Allinea DDT’s memory debugging
  – Checks double frees
  – Checks use of dangling (freed) pointers
  – Can force O/S to check for read/write beyond bounds
    • Make some random bugs deterministic and occur 100% of the time (“Guard Pages”)
  – showing memory leaks (group by allocation point)

• Other tips – get inspiration from a colleague
  – try explaining your code
    • The very act of explaining your thinking in the code can help
  – don’t have a colleague to hand?
    • Follow http://www.rubberduckdebugging.com
Favorite Allinea DDT Features for Scale

- Parallel stack view
- Automated data comparison: sparklines
- Parallel array searching
- Step, play, and breakpoints
- Offline debugging
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
A Productive HPC Development Workflow
Tour/Demo of Forge

- Debugging with DDT
  - Walkthrough example – cstartmpi.exe
- Profiling with MAP
  - Matrix-multiplication
Hands on Session

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

• Use Allinea MAP on NERSC Edison or ANL Cooley to see your code performance

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

• Can you beat a 50% speed up?
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.
Performance and debugging challenge

• Obtain the code:
  – git clone https://github.com/estrabd/2d-heat.git

• The challenge
  – Improve run time by 50%
  – Parameters: –h 1000 –w 1000 set the dimensions

• Submit/run
  • mpirun –np 16 2d-heat.x –h 1000 –w 1000

• Use MAP to find the lowest hanging fruits
• Use DDT to look at behaviour - confirm what is safe to change
• Edit and build in the GUI: File/Configure Build/ and re-run
Initial Performance Report

- Using Forge – profile, debug, edit, build, repeat!
  - profiler to find targets
  - debugger to reveal usage
  - Make changes
  - Configure the Build command
    - make CFLAGS=‘-O3 -g’
    - **NB: single quotes**..
  - Build, repeat!

Summary: shmem_2dheat.x is Compute-bound in this configuration

CPU
- A breakdown of the 99.8% CPU time:
  - Vector numerical ops: 79.1%
  - Vector memory ops: 43.9%

MPI
- A breakdown of the 0.1% MPI time:
  - Time in collective calls: 0.0%
  - Effective process collective rate: 0.00 bytes/sec
  - Effective process point to point rate: 0.00 bytes/sec

I/O
- A breakdown of the 0.0% I/O time:
  - Time in reads: 0.0%
  - Time in writes: 0.0%

Threads
- A breakdown of how multiple threads were used:
  - Computation: 0.0%
  - Synchronization: 0.0%
  - Physical core utilization: 100.0%

Memory
- Response memory usage may also affect scaling:
  - Peak process memory usage: 246.4 MB
  - Peak process memory usage: 415.8 MB
  - Peak node memory usage: 15.0 MB
  - Peak node memory usage: 10.0 MB

This application ran was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below. As very little time is spent in MPI calls, this code may also benefit from running at larger scales.