Tools and Methodology for Ensuring HPC Programs Correctness and Performance

Beau Paisley
bpaisley@allinea.com
Allinea

- Over 15 years of business focused on parallel programming development tools
- Strong R&D investment to drive innovation in changing landscape
- Committed to giving great support to the HPC community
Where to find Allinea’s tools

- Over 65% of Top 100 HPC systems
  - From small to very large tools provision

- 8 of the Top 10 HPC systems
  - Up to 700,000 core tools usage

- Future leadership systems
  - Millions of cores usage
Allinea: Industry Standard Tools for HPC

(and hundreds more)
Performance in a Nutshell

- Algorithmic Issues
- Balance and Shared Bottlenecks
- The Memory Wall
- Use of Processor Capability
Performance Improvement Workflow

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 with a profiler

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
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 one at a time, with an emphasis on measuring and understanding before rewriting.

1. 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

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. REVIEW COMMUNICATION
   - 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. 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

5. 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

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

BALANCE WORKLOAD
- Spacing 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

FINISH
The Uncomfortable Truth about Applications

Optimized for modern CPUs

Hey, at least it compiles
Obtaining Program Correctness in a Nutshell

- Interactive multi-process and multi-thread debugging at any scale
- Support common architectures and co-processors
- Offline debugging for large runs and non-deterministic bugs
- Support for integration to regression test
Debugging at Scale Requires Powerful Visual Representations
Enable Debugging Across a Range of Architectures

- ARM
- IBM OpenPOWER
- 64-bit ARM
- NVIDIA GPUs
- Intel KNL
- Coming Soon

allinea
Enable Large Scale Debugging and Regression Testing with Offline Debugging
Overview of Allinea Tools

- **Demand for software efficiency**
- **Demand for developer efficiency**
- **Demand for performance optimization**
- **Demand for debugging**

**Forge**

- **Performance Reports Measure**

**MAP**

- **Profile and Optimize**

**DDT**

- **Debug**

**Open Interfaces** (eg. JSON APIs)

**Continuous Integration**

**Version Control**

Leads to MAP to optimize performance

Pull for MAP to develop performance fix

Leads to DDT to understand and fix

Debug, optimize, edit, commit, build, repeat…
Analyze and tune application performance

A single-page report on application performance for users and administrators

Identify configuration problems and resource bottlenecks immediately

Track mission-critical performance over time and after system upgrades

Ensure key applications run at full speed on a new cluster or architecture

Summary: clover_leaf is CPU-bound in this configuration

- **CPU**: 80.6%
  - Time spent running application code. High values are usually good. This is high; check the CPU performance section for optimization advice.
  - Most of the time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

- **MPI**: 19.4%
  - Time spent in MPI calls. High values are usually bad. This is low; this code may benefit from increasing the process count.

- **I/O**: 0.1%
  - Time spent in filesystem 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 CPU-bound. A breakdown of this time and advice for investigating further is in the CPU section below. As little time is spent in MPI calls, this code may also benefit from running at larger scales.

**CPU**
- A breakdown of the 80.6% CPU time:
  - Single-core code: 0.4%
  - OpenMP regions: 99.6%
  - Scalar numeric ops: 42.4%
  - Vector numeric ops: 4.0%
  - Memory accesses: 53.6%

**MPI**
- A breakdown of the 19.4% MPI time:
  - Time in collective calls: 41.7%
  - Time in point-to-point calls: 58.3%
  - Effective process collective rate: 1.68 kbps
  - Effective process point-to-point rate: 24.5 kbps

Most of the time is spent in point-to-point calls with a low transfer rate. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait. The collective transfer rate is very low. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate further.

**I/O**
- A breakdown of the 0.1% 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: 411 kbytes/s

**OpenMP**
- A breakdown of the 99.0% time in OpenMP regions:
  - Computation: 100.0%
  - Synchronization: 0.0%
  - Physical core utilization: 200.0%
  - Inv/undefined context switch per second: 0.0
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.
Allinea MAP – The Profiler

- 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
Why MAP? ...

• Easy to use
• Fast, with low overhead
• Time-based indexing of data
• Extensive metrics, e.g., (I/O, memory, floating-point operations, line-level granularity)
• Customized views into the data
• Extensible with API that can be used to capture custom measurements
• High accuracy
• Professional, responsive support
Allinea DDT – The Debugger

• 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
Bottling it…

• Lock in obtain results; Performance AND Correctness
• Save your results nightly
• Tie your performance results to your continuous integration server
Top Tips for HPC Development Success

• Performance is important
• Software needs performance attention
• Regular profiling pays rewards
• Test correctness and validate performance on real workloads
• Integrate your debugger with program correctness regression testing
• Constant diligence pays off