allinea

High performance tools to debug, profile, and analyze your applications

# Tools and Methodology for Ensuring HPC Programs Correctness and Performance

Beau Paisley
bpaisley@allinea.com

allinea FORGE allinea DDT allinea MAP allinea PERFORMANCE REPORTS

# About 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

Oil – X
Confidential

(and hundreds more)

allinea

# 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 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

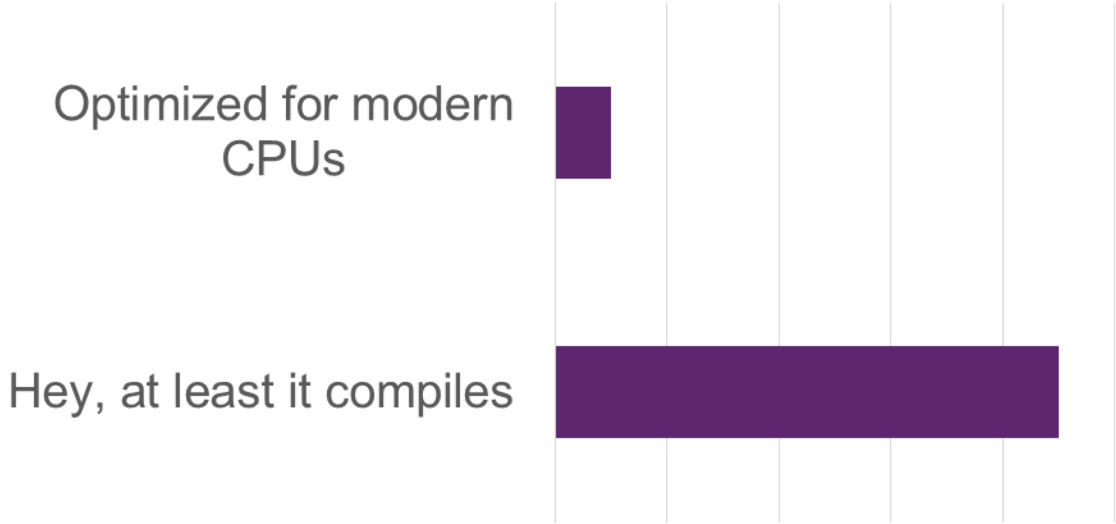
**TOOLS FOR SUCCESS:**

- Allinea Performance Reports shows numerical intensity and level of vectorization
- Allinea Forge shows hot loops, unvectorized code and GPU performance

FINISH

allinea

# The Uncomfortable Truth about Applications




allinea FORGE

allinea DDT

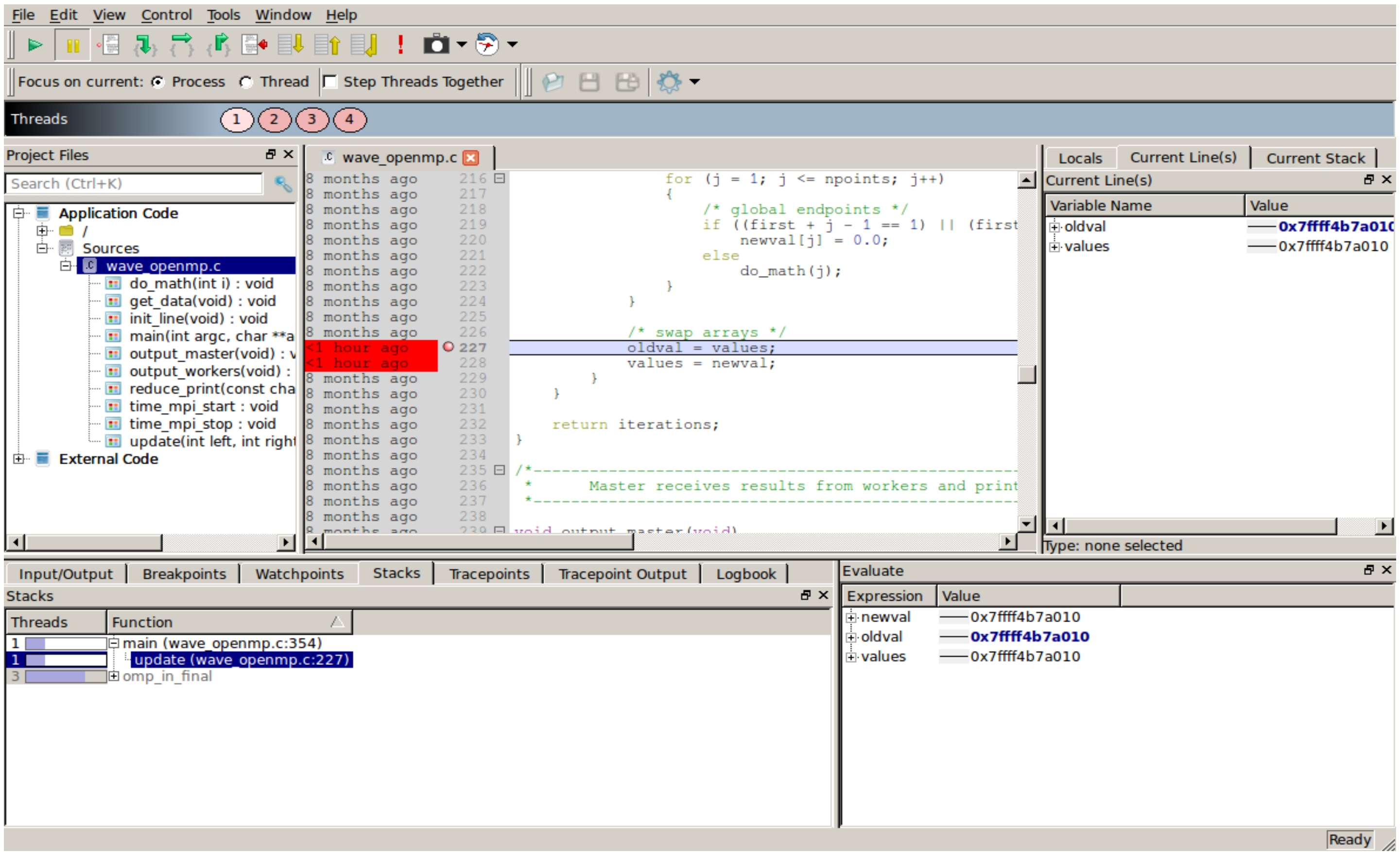
allinea MAP

allinea

# 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

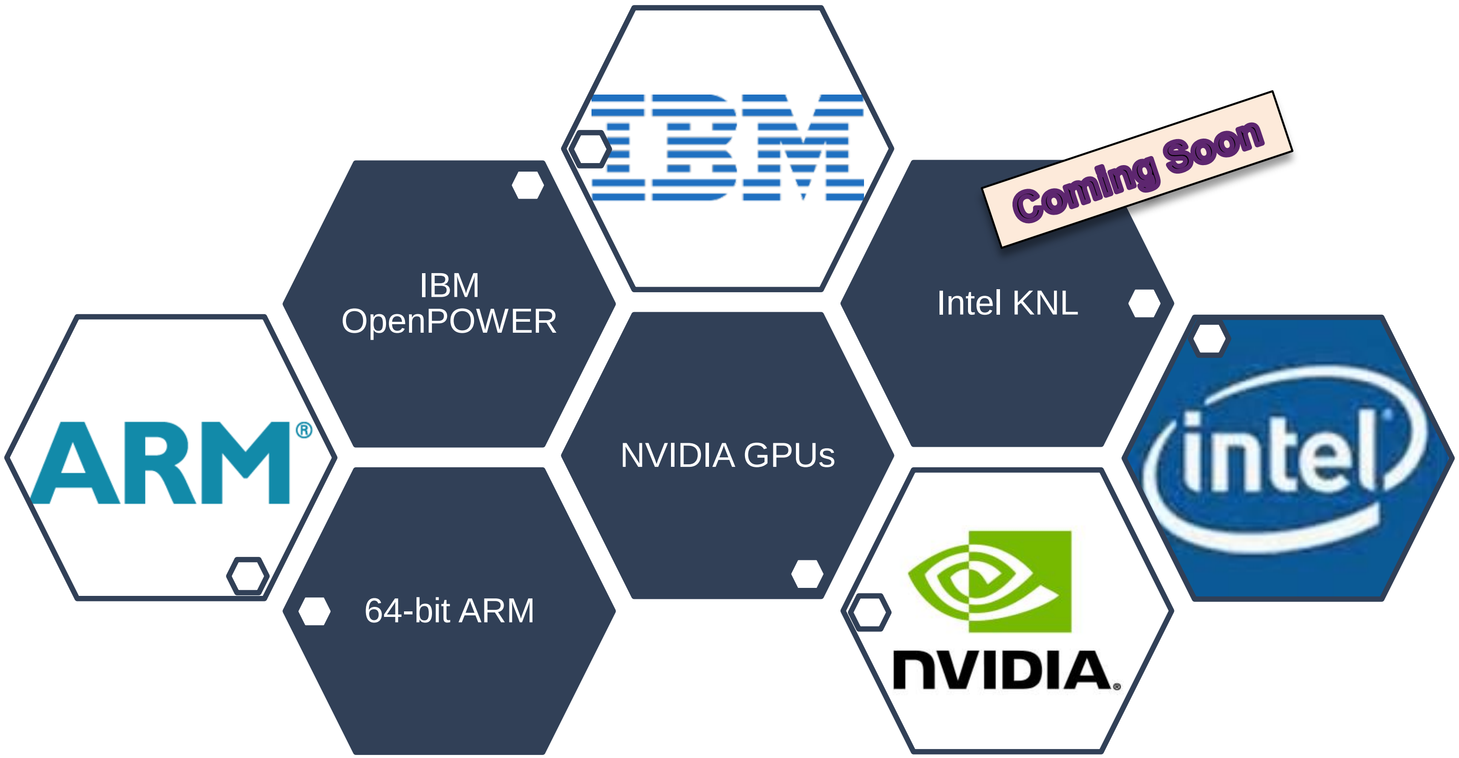
# Debugging at Scale Requires Powerful Visual Representations



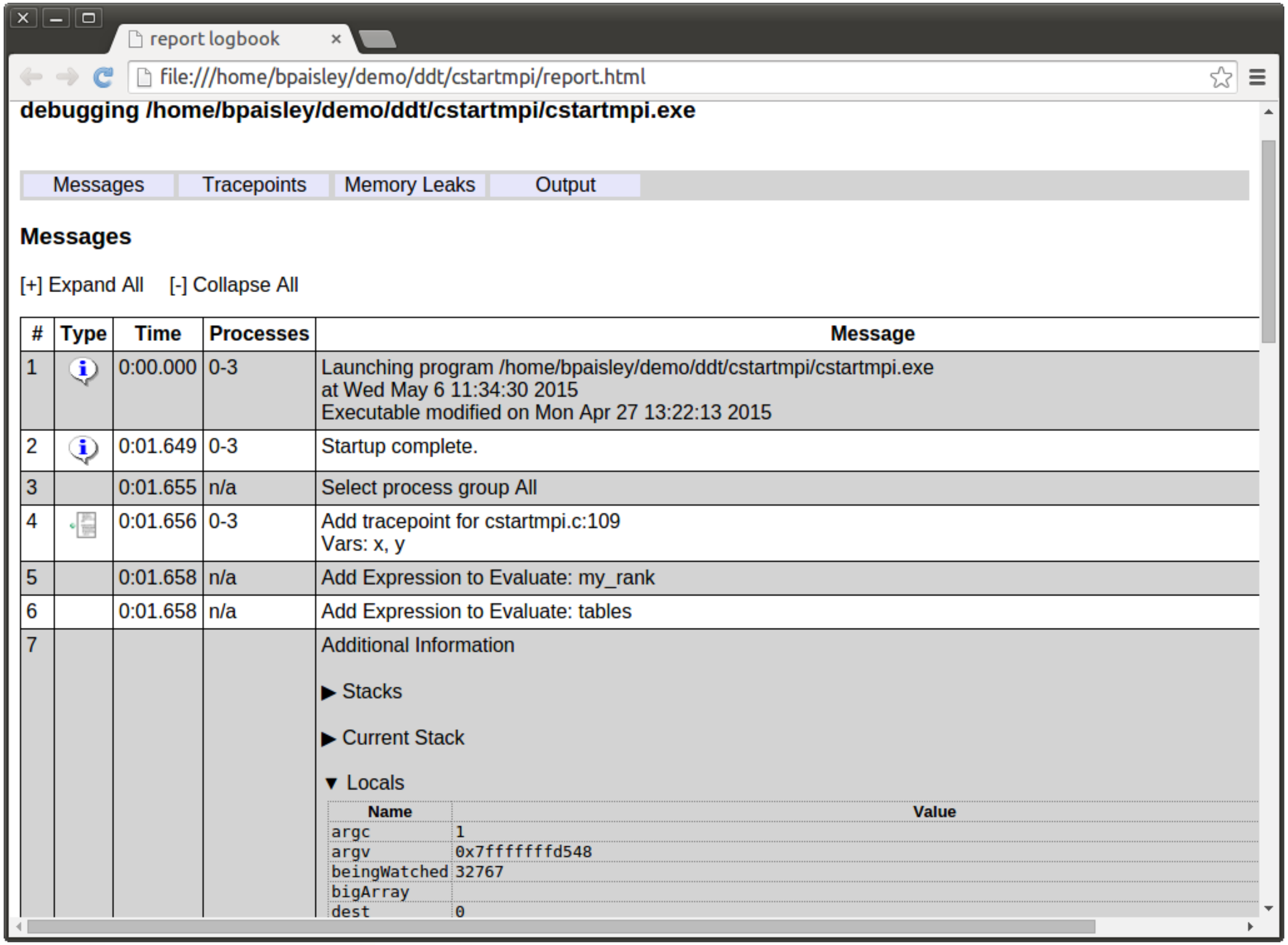

allinea DDT

allinea

# Enable Debugging Across a Range of Architectures



# Enable Large Scale Debugging and Regression Testing with Offline Debugging



report logbook

file:///home/bpaisley/demo/ddt/cstartmpi/report.html

**debugging /home/bpaisley/demo/ddt/cstartmpi/cstartmpi.exe**

Messages | Tracepoints | Memory Leaks | Output

**Messages**

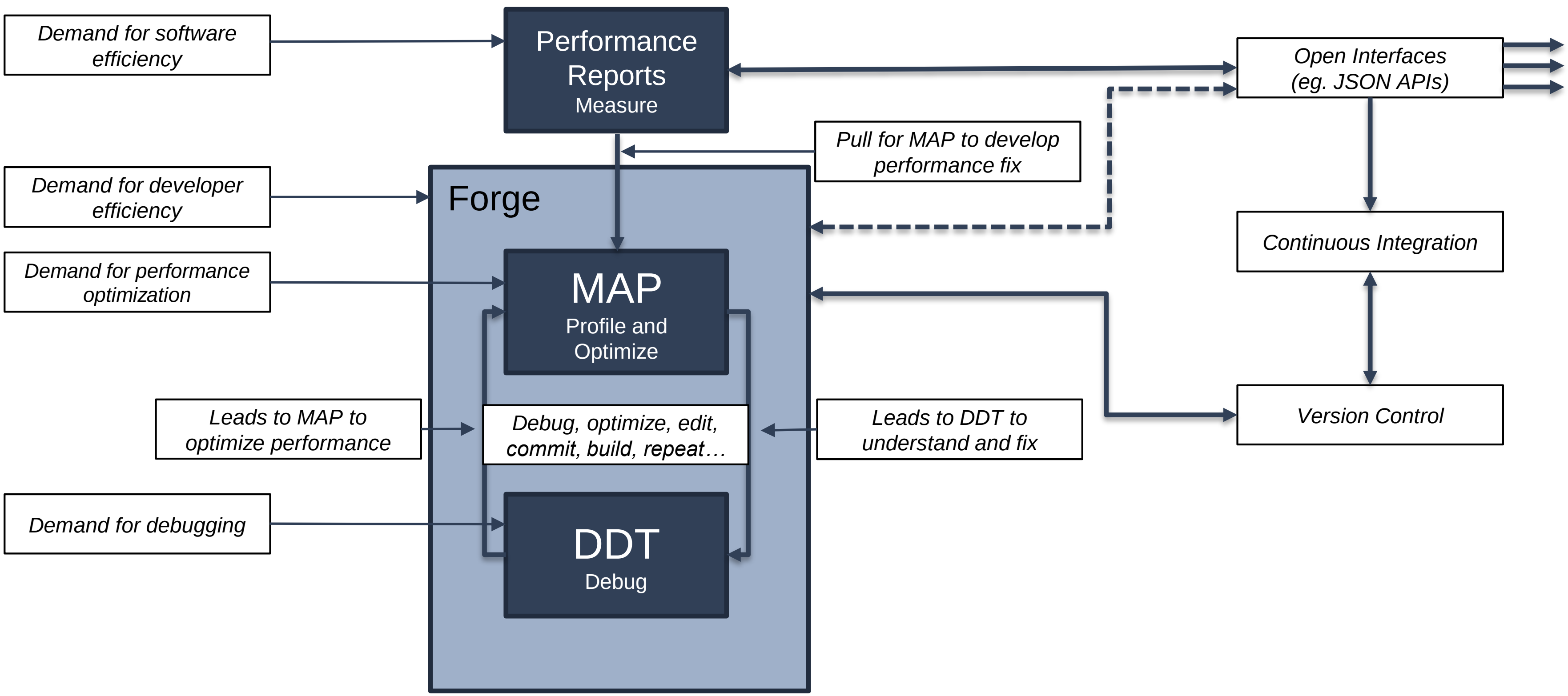
[+] Expand All [-] Collapse All

| # | Type | Time | Processes | Message |
|---|---|---|---|---|
| 1 | | 0:00.000 | 0-3 | Launching program /home/bpaisley/demo/ddt/cstartmpi/cstartmpi.exe<br>at Wed May 6 11:34:30 2015<br>Executable modified on Mon Apr 27 13:22:13 2015 |
| 2 | | 0:01.649 | 0-3 | Startup complete. |
| 3 | | 0:01.655 | n/a | Select process group All |
| 4 | | 0:01.656 | 0-3 | Add tracepoint for cstartmpi.c:109<br>Vars: x, y |
| 5 | | 0:01.658 | n/a | Add Expression to Evaluate: my_rank |
| 6 | | 0:01.658 | n/a | Add Expression to Evaluate: tables |
| 7 | | | | Additional Information<br>▶ Stacks<br>▶ Current Stack<br>▼ Locals |

| Name | Value |
|---|---|
| argc | 1 |
| argv | 0x7fffffffd548 |
| beingWatched | 32767 |
| bigArray | |
| dest | 0 |

allinea

# Overview of Allinea Tools




allinea

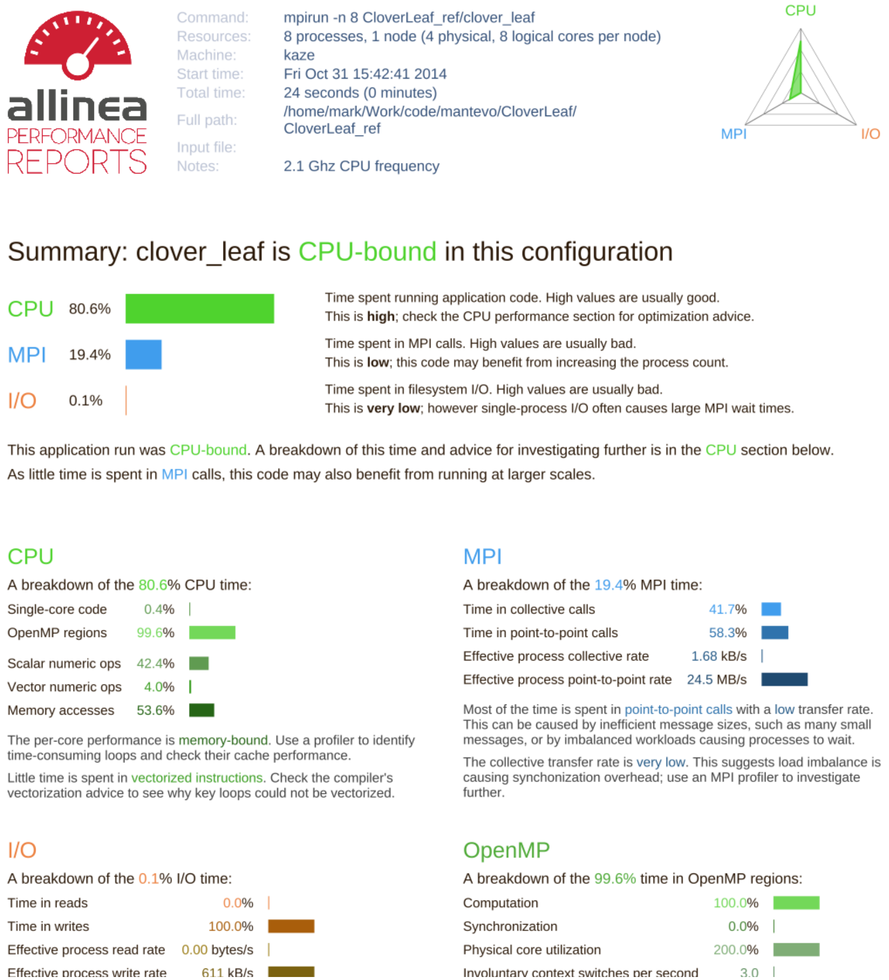
# Analyze and tune application performance

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



allinea PERFORMANCE REPORTS

| | |
|---|---|
| Command: | mpirun -n 8 CloverLeaf_ref/clover_leaf |
| Resources: | 8 processes, 1 node (4 physical, 8 logical cores per node) |
| Machine: | kaze |
| Start time: | Fri Oct 31 15:42:41 2014 |
| Total time: | 24 seconds (0 minutes) |
| Full path: | /home/mark/Work/code/mantevo/CloverLeaf/ CloverLeaf_ref |
| Input file: | |
| Notes: | 2.1 Ghz CPU frequency |

CPU MPI I/O

## 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. |
| 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% |

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

Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

### 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 kB/s |
| Effective process point-to-point rate | 24.5 MB/s |

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 synchonization 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 | 611 kB/s |

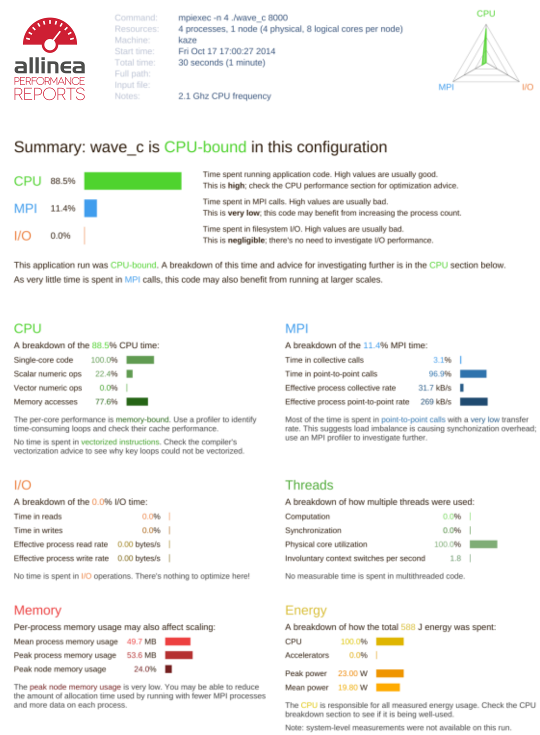
### OpenMP

A breakdown of the 99.6% time in OpenMP regions:

| | |
|---|---|
| Computation | 100.0% |
| Synchronization | 0.0% |
| Physical core utilization | 200.0% |
| Involuntary context switches per second | 3.0 |

allinea

# Vectorization, MPI, I/O, memory, energy…



Command: mpiexec -n 4 ./wave_c 8000
Resources: 4 processes, 1 node (4 physical, 8 logical cores per node)
Machine: kaze
Start time: Fri Oct 17 17:00:27 2014
Total time: 30 seconds (1 minute)
Full path:
Input file:
Notes: 2.1 Ghz CPU frequency



Summary: wave_c is CPU-bound in this configuration

| | | |
|---|---|---|
| CPU | 88.5% | Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for optimization advice. |
| MPI | 11.4% | Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from increasing the process count. |
| I/O | 0.0% | Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance. |

This application run was CPU-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.

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.

MPI

A breakdown of the 11.4% MPI time:

| | |
|---|---|
| Time in collective calls | 3.1% |
| Time in point-to-point calls | 96.9% |
| Effective process collective rate | 31.7 kB/s |
| Effective process point-to-point rate | 269 kB/s |

Most of the time is spent in point-to-point calls with a very low transfer rate. This suggests load imbalance is causing synchonization overhead; use an MPI profiler to investigate further.

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/s |
| Effective process write rate | 0.00 bytes/s |

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

Threads

A breakdown of how multiple threads were used:

| | |
|---|---|
| Computation | 0.0% |
| Synchronization | 0.0% |
| Physical core utilization | 100.0% |
| Involuntary context switches per second | 1.8 |

No measurable time is spent in multithreaded code.

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 588 J energy was spent:

| | |
|---|---|
| CPU | 100.0% |
| Accelerators | 0.0% |
| Peak power | 23.00 W |
| Mean power | 19.80 W |

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

Note: system-level measurements were not available on this run.

## 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

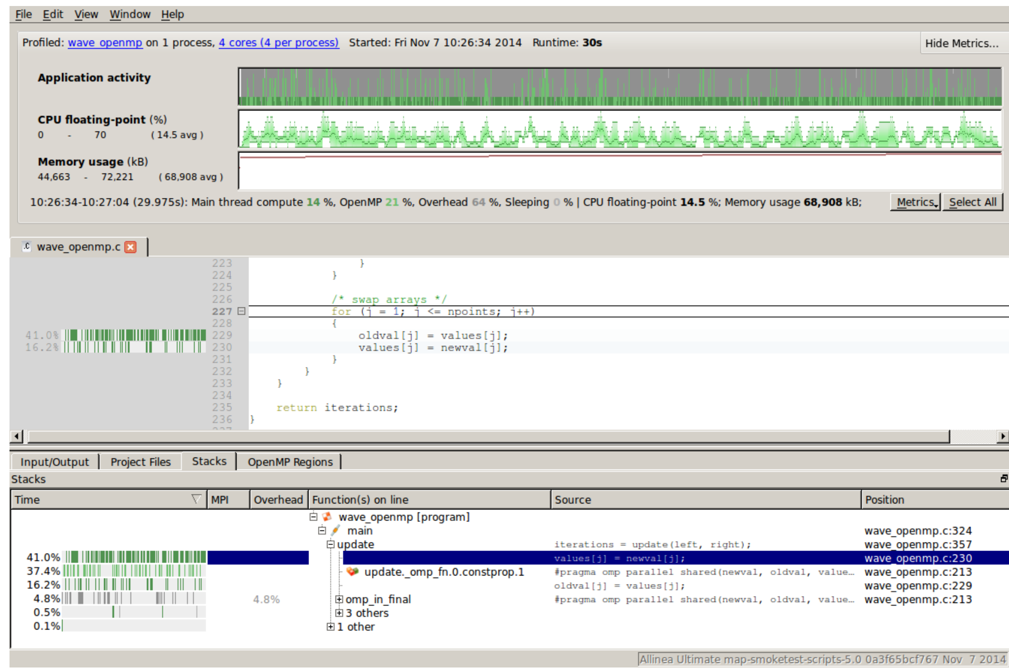
Per-process memory usage may also affect scaling:

| | |
|---|---|
| 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



allinea MAP

# How Allinea MAP is different

allinea FORGE

| | | | |
|---|---|---|---|
| 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** |

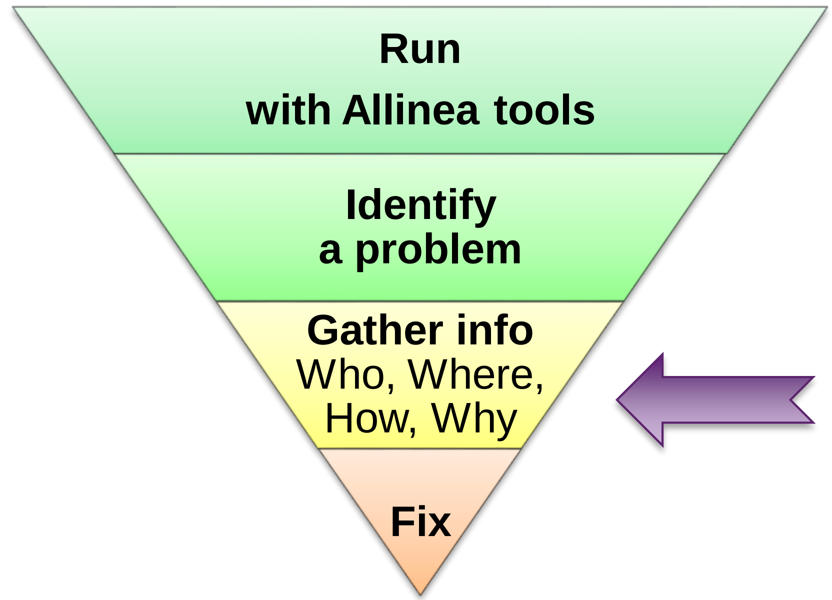
allinea

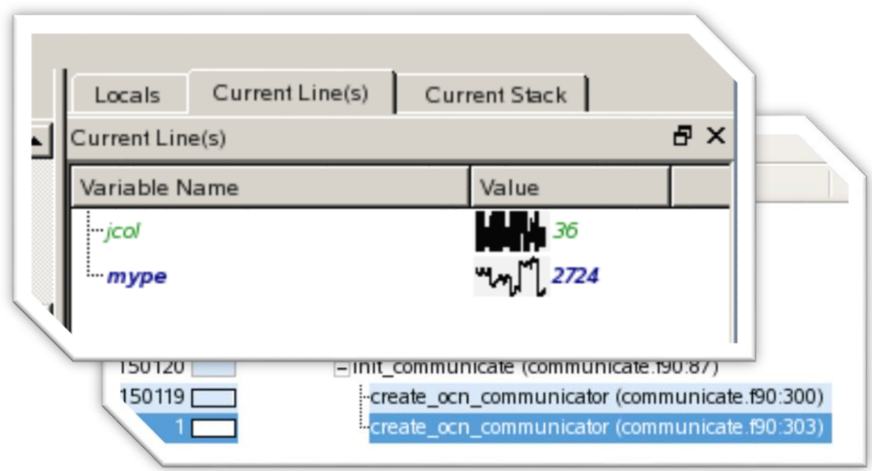
# Why MAP ? …

- 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