



**allinea**  
**FORGE**  
DDT + MAP



# Debugging and Profiling your HPC Applications

Srinath Vadlamani, Field Application Engineer

[srinath.vadlamani@arm.com](mailto: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 Alinea DDT
- Benchmarking with Alinea Performance Reports
- Profiling with Alinea MAP
- Go to [www.allinea.com/trials](http://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

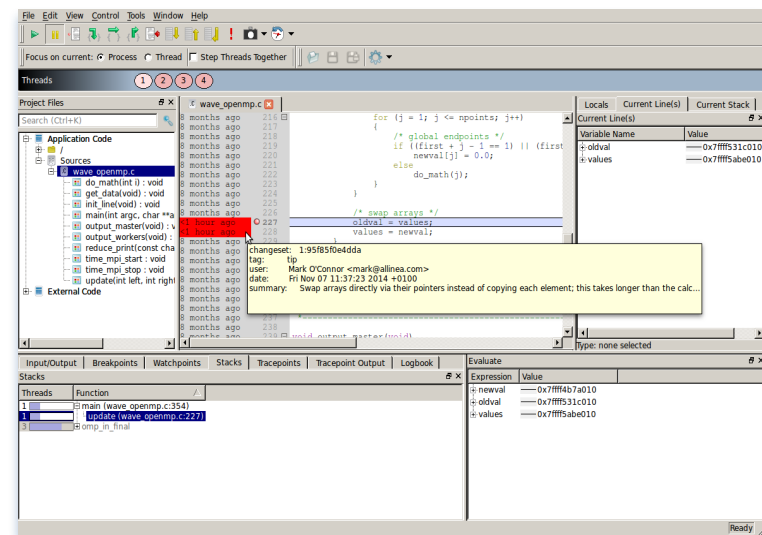
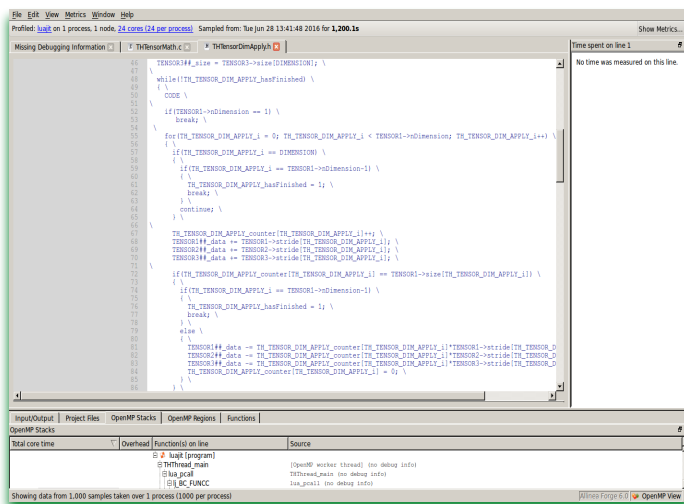
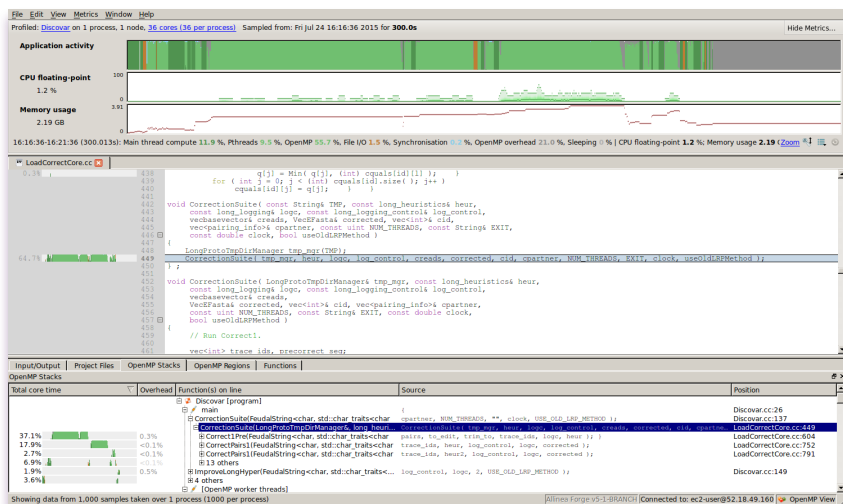
- PhD to submit
- Project to complete
- Paper to write
- Career to develop

## 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  
**arm**

## Real-world example



**Bioinformatics**  
Discover Assembly  
3x speedup  
EC2

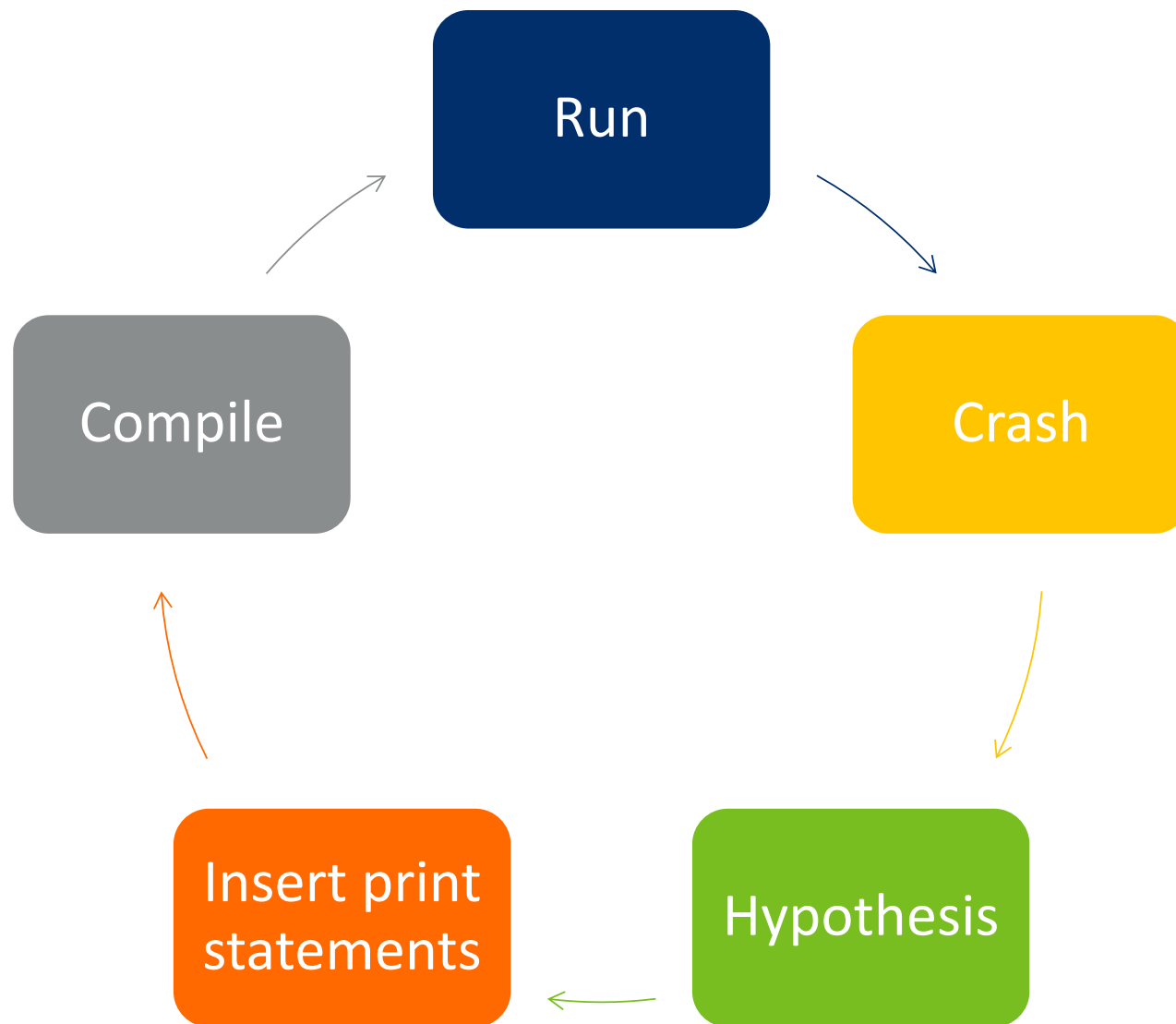
**Deep Learning**  
Torch + DeepMind  
5.3x speedup  
Intel Xeon Phi (KNL)

## Fluid Dynamics

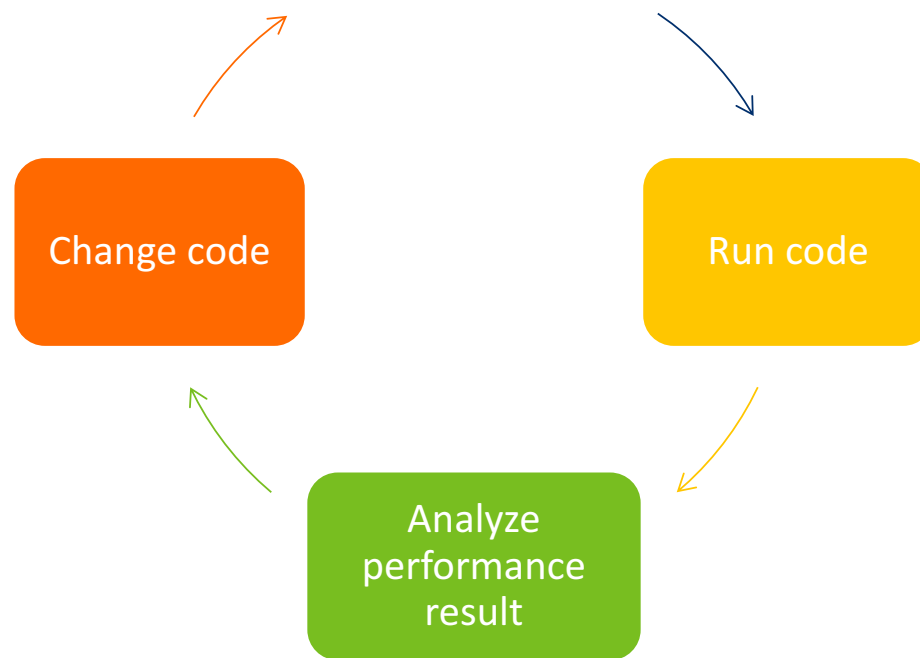
HemeLB blood flow  
16.8x capability boost  
50k core crash fixed

Aug. 8, ATPESC\_2017  
grm

# Debugging in practice...



# Optimization in Practice



## 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

---

Bohrbug

Steady, dependable bug

---

Heisenbug

Vanishes when you try to debug (observe)

---

Mandelbug

Complexity and obscurity of the cause is so great that it appears chaotic

---

Schroedinbug

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

---



# 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

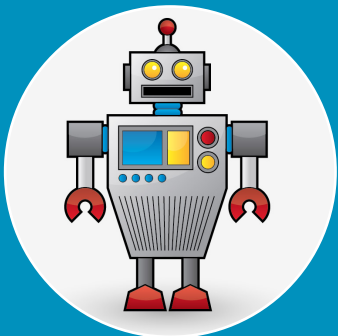
Suggested Reading:

- Andreas Zeller, “Why Programs Fail”, 2nd Edition, 2009

What you will read:

- Crowd sources like *stack overflow*

# 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?

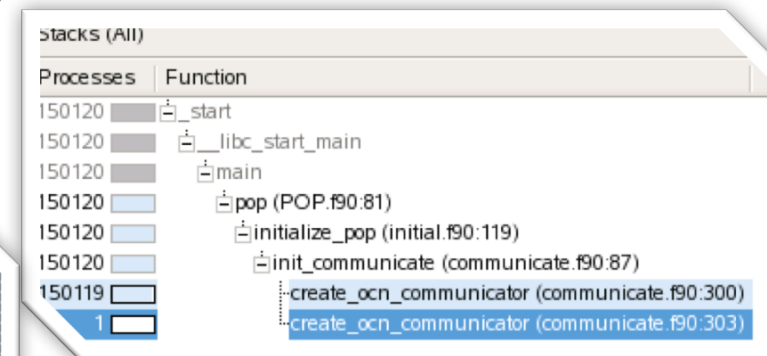
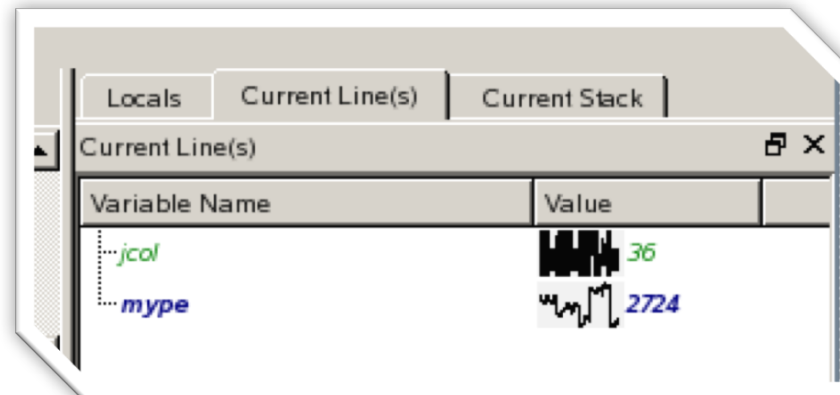
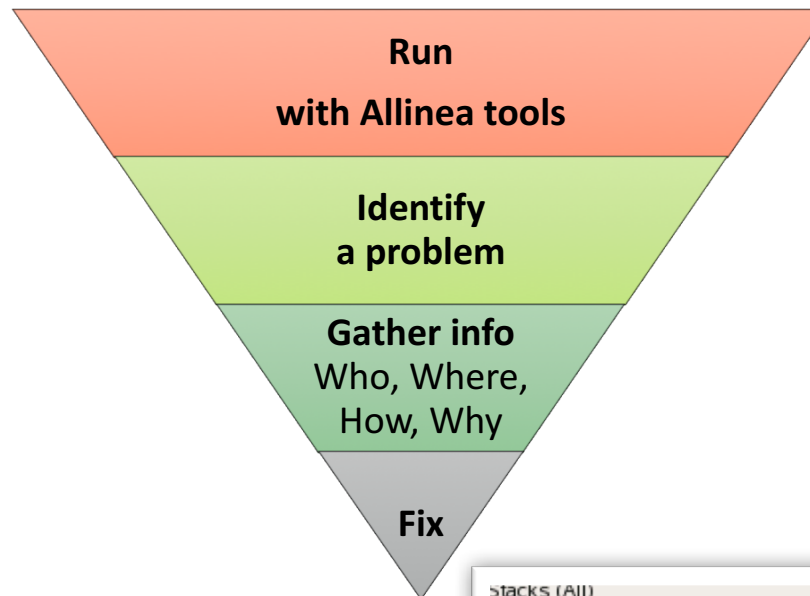
- leaps to source

How did it happen?

- Diagnostic messages
- Some faults evident instantly from source

Why did it happen?

- Unique “Smart Highlighting”
- Sparklines comparing data across processes



Aug. 8, ATPESC\_2017  
**arm**

# Favorite Allinea DDT Features for Scale

The first screenshot shows the 'Stacks (All)' window with a list of processes and functions. A blue cloud labeled 'Parallel' is overlaid on the list. Below it is a dark blue box with the text 'Parallel stack view'.

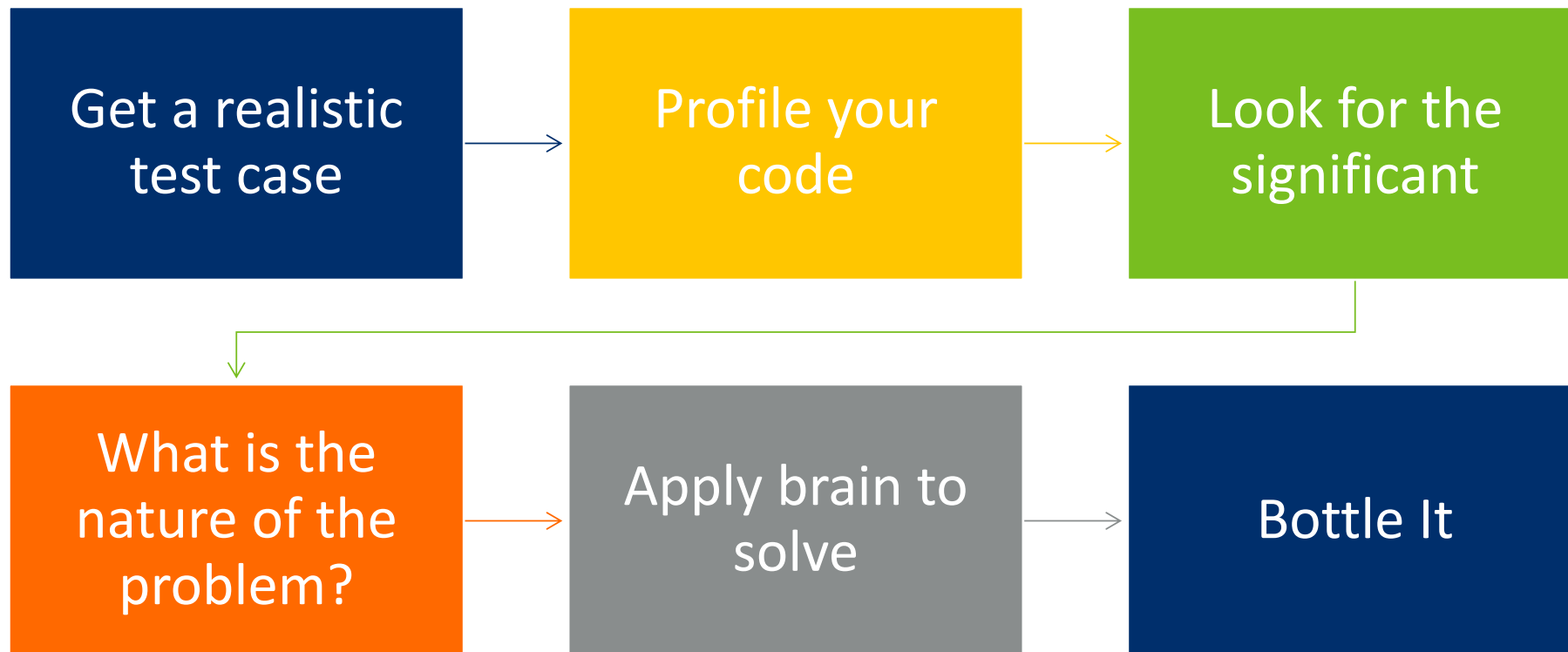
The second screenshot shows the 'Current Line(s)' and 'Current Stack' windows. A blue cloud labeled 'Parallel' is overlaid on the 'Current Line(s)' window. Below it is a yellow box with the text 'Automated data comparison: sparklines'.

The third screenshot shows the 'Array Expression' window with a table of data. A blue cloud labeled 'Parallel' is overlaid on the table. Below it is a green box with the text 'Parallel array searching'.

The first screenshot shows the 'All' window with a list of processes and threads. A blue cloud labeled 'Parallel' is overlaid on the list. Below it is an orange box with the text 'Step, play, and breakpoints'.

The second screenshot shows the 'Stacks' window with a list of processes and threads. A blue cloud labeled 'Parallel' is overlaid on the list. Below it is a grey box with the text 'Offline debugging'.

# 6 steps to help improve performance



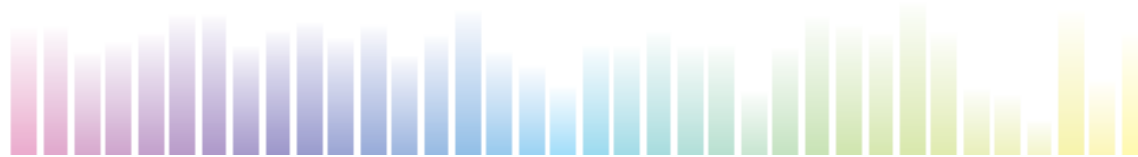
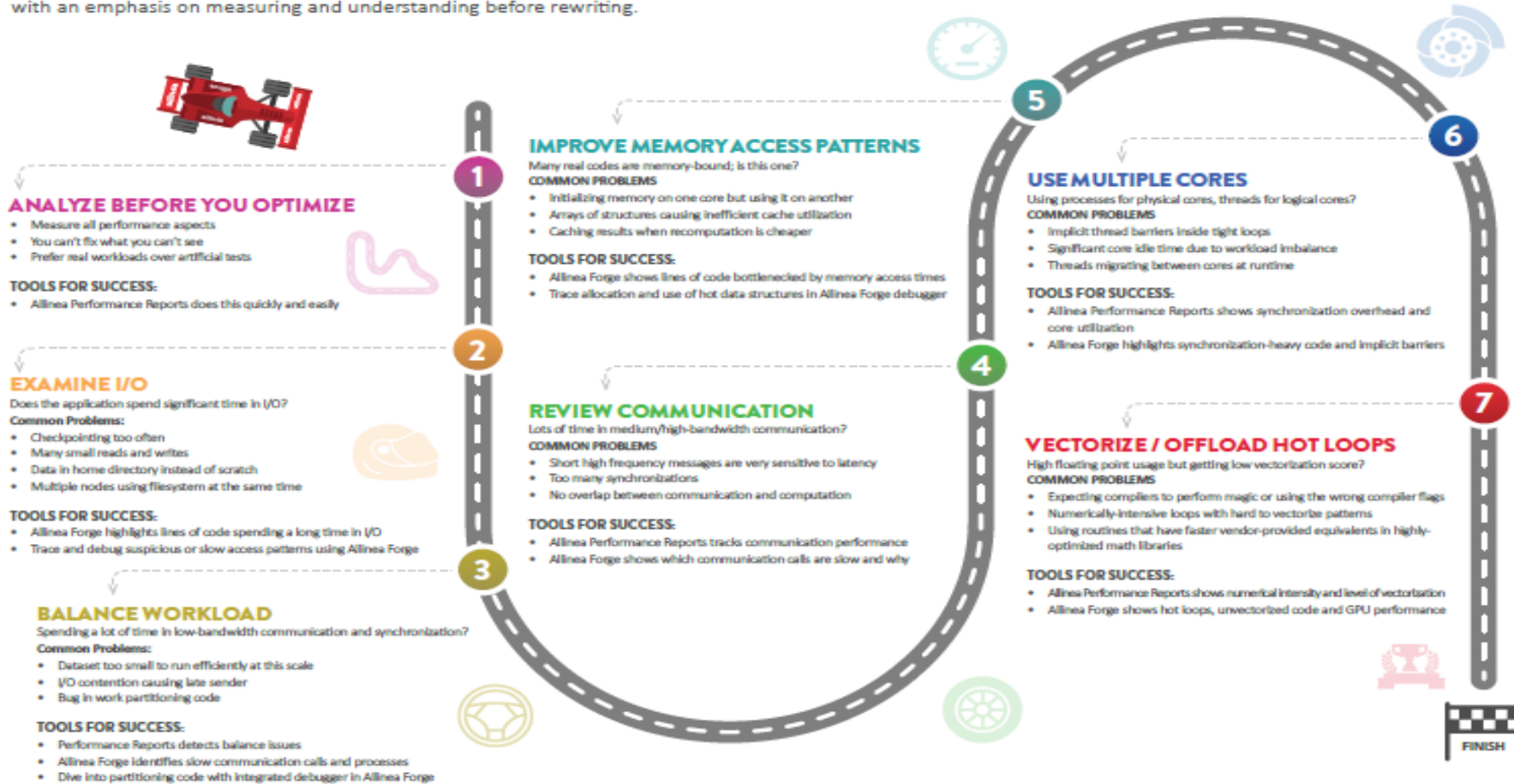
Logging like an experiment is useful.

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

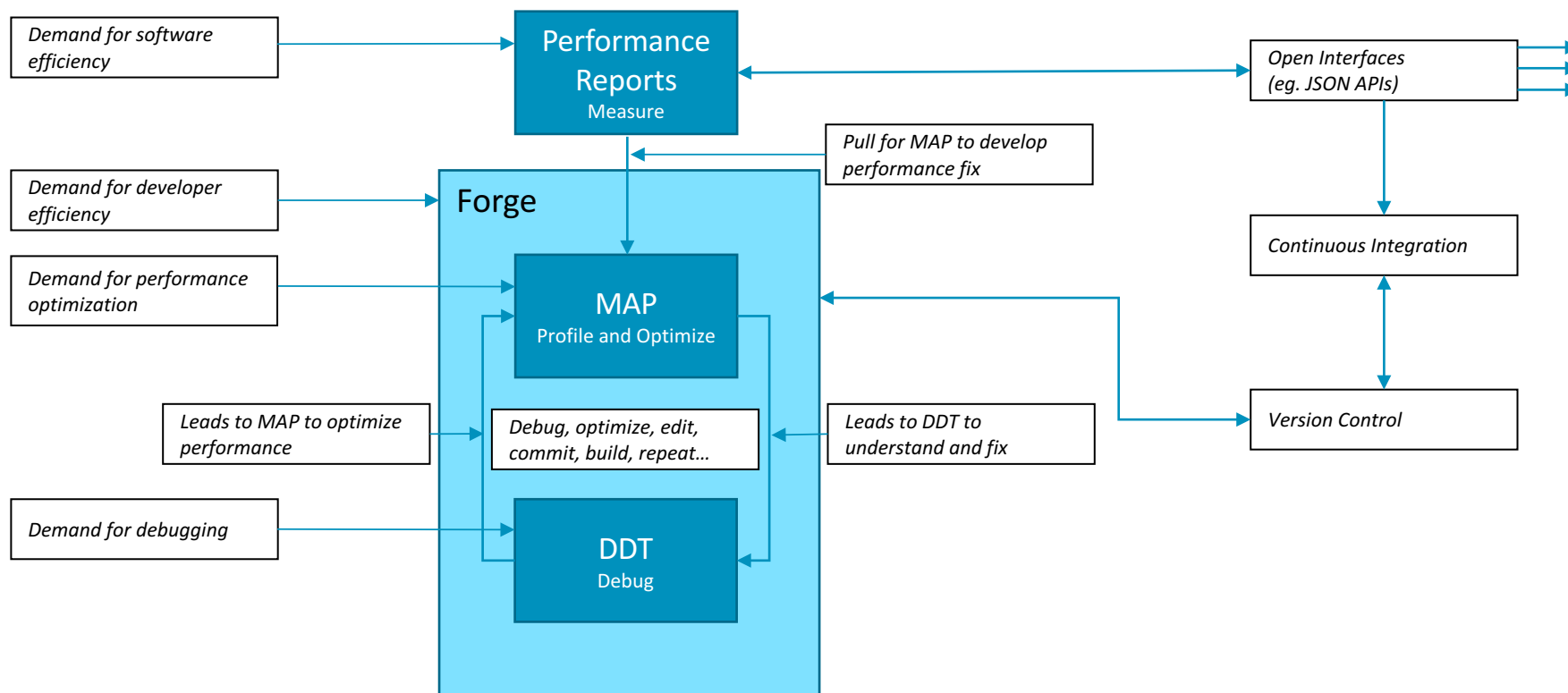


**allinea**

Contact us:  
w: [www.allinea.com](http://www.allinea.com) t: +1 (408) 600 2786 e: [sales@allinea.com](mailto:sales@allinea.com)  
Allinea Software Inc. 2033 Gateway Place, Suite 500, San Jose, CA 95110

Aug. 8, ATPESC\_2017  
**arm**

## How The Tools Fit...



Aug. 8, ATPESC\_2017  
grm



# 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”, ie. no optimizations

# Show performance they understand

## CPU

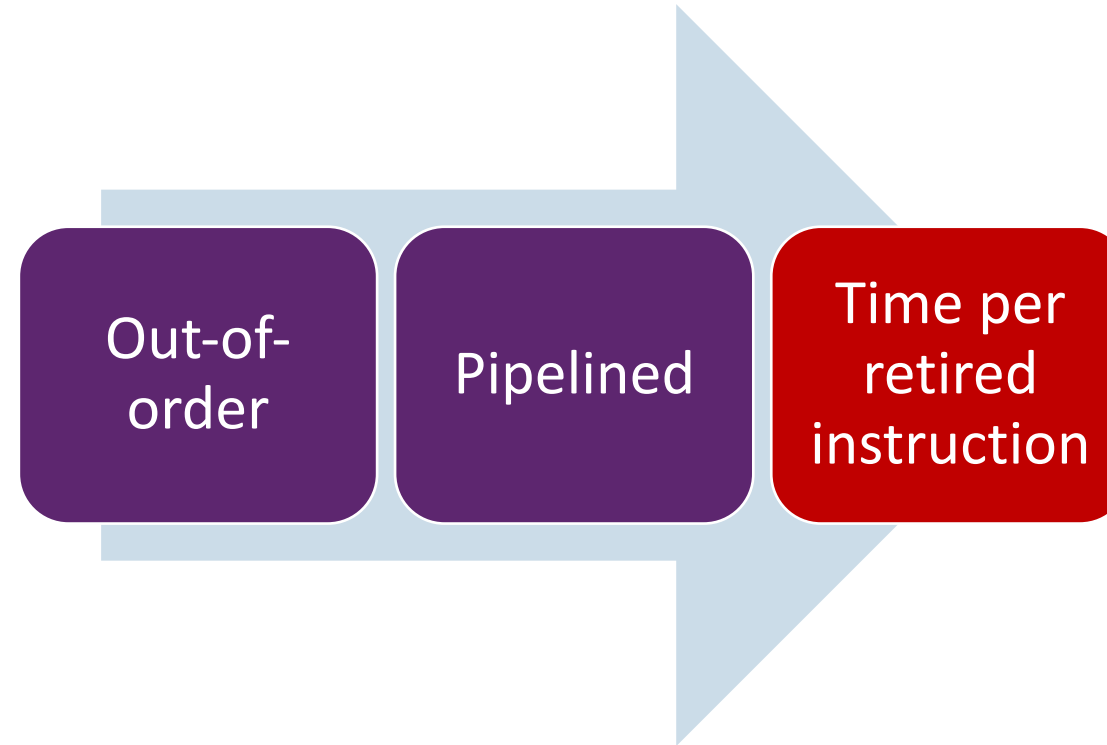
A breakdown of the 88.5% CPU time:

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



# Explaining performance at the right level

## CPU

A breakdown of the 88.5% CPU time:

Single-core code	100.0%	<div></div>
Scalar numeric ops	22.4%	<div></div>
Vector numeric ops	0.0%	<div></div>
Memory accesses	77.6%	<div></div>

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.

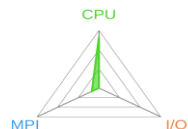
↗  
+ simple, actionable advice

Compiler advice is your friend.

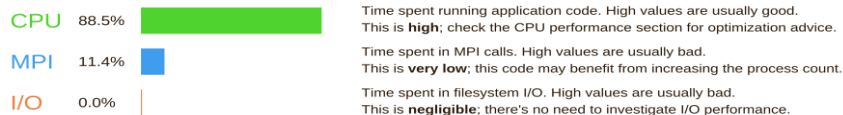
# Vectorization, MPI, I/O, memory, energy...



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



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:

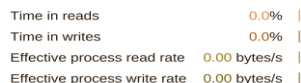


The per-core performance is **memory-bound**. Use a profiler to identify time-consuming loops and check their cache performance.

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

## I/O

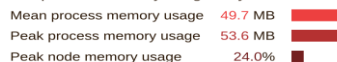
A breakdown of the **0.0%** I/O time:



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

## Memory

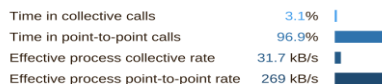
Per-process memory usage may also affect scaling:



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

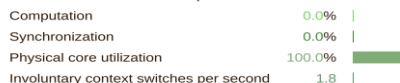
A breakdown of the **11.4%** MPI time:



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

## Threads

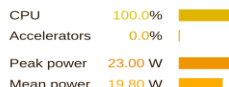
A breakdown of how multiple threads were used:



No measurable time is spent in multithreaded code.

## Energy

A breakdown of how the total **588 J** energy was spent:



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:

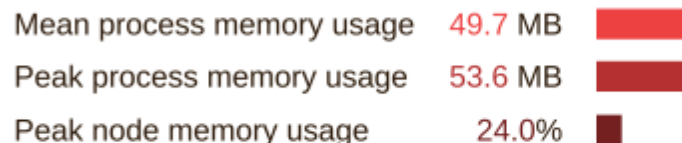


The per-core performance is **memory-bound**. Use a profiler to identify time-consuming loops and check their cache performance.

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:



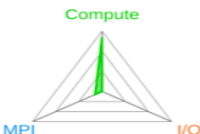
The **peak node memory usage** is very low. You may be able to reduce the amount of allocation time used by running with fewer MPI processes and more data on each process.



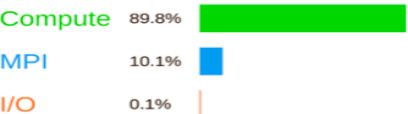
# Accelerator support...



Command: /scratch/mark/miniMD\_OpenCL/miniMD\_nvidia -t 128  
Resources: 2 processes, 1 node (12 physical, 24 logical cores per node)  
Machine: kaze  
Start time: Mon Nov 3 11:52:14 2014  
Total time: 110 seconds (2 minutes)  
Full path:  
Input file:  
Notes:



Summary: miniMD\_nvidia is **Compute-bound** in this configuration

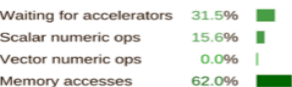


Time spent running application code. High values are usually good. This is **high**; check the CPU and accelerator sections for optimization advice.  
Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from increasing the process count.  
Time spent in filesystem 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 **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

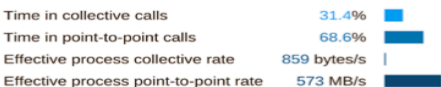
A breakdown of the **89.8%** CPU time:



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 **10.1%** MPI time:



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.

## I/O

A breakdown of the **0.1%** I/O time:



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

## Threads

A breakdown of how multiple threads were used:



**Physical core utilization** is low. Try increasing the number of threads or processes to improve performance.  
Significant time is spent **synchronizing** threads. Check which locks cause the most overhead with a profiler.

## Memory

Per-process memory usage may also affect scaling:



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.

## Accelerators

A breakdown of how accelerators were used:

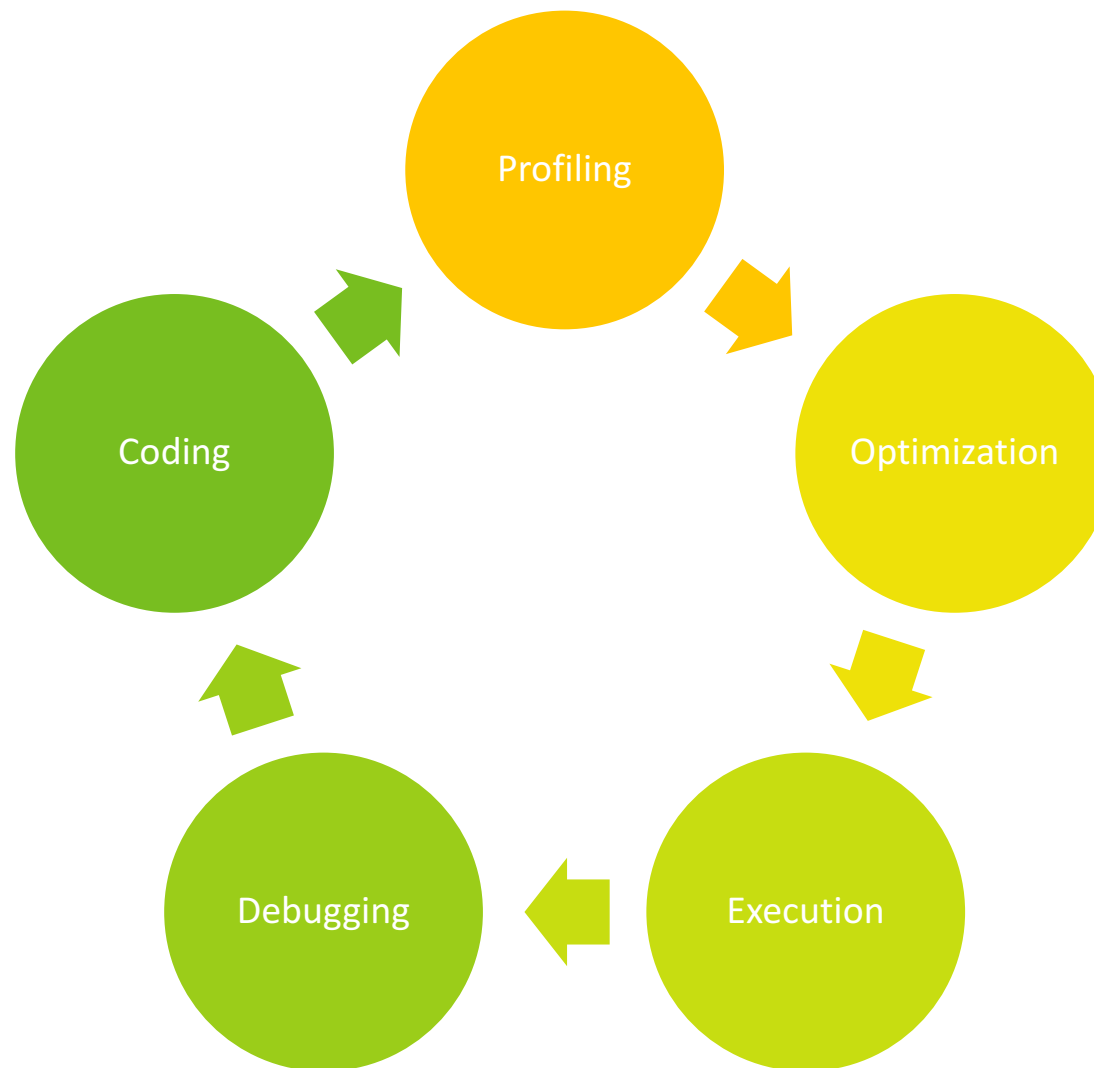


High **compute** and low **memory** utilization suggests compute throughput is limiting GPU performance.  
Use a profiler to find the hottest kernels and check them for divergent branches and over-subscribed function units.



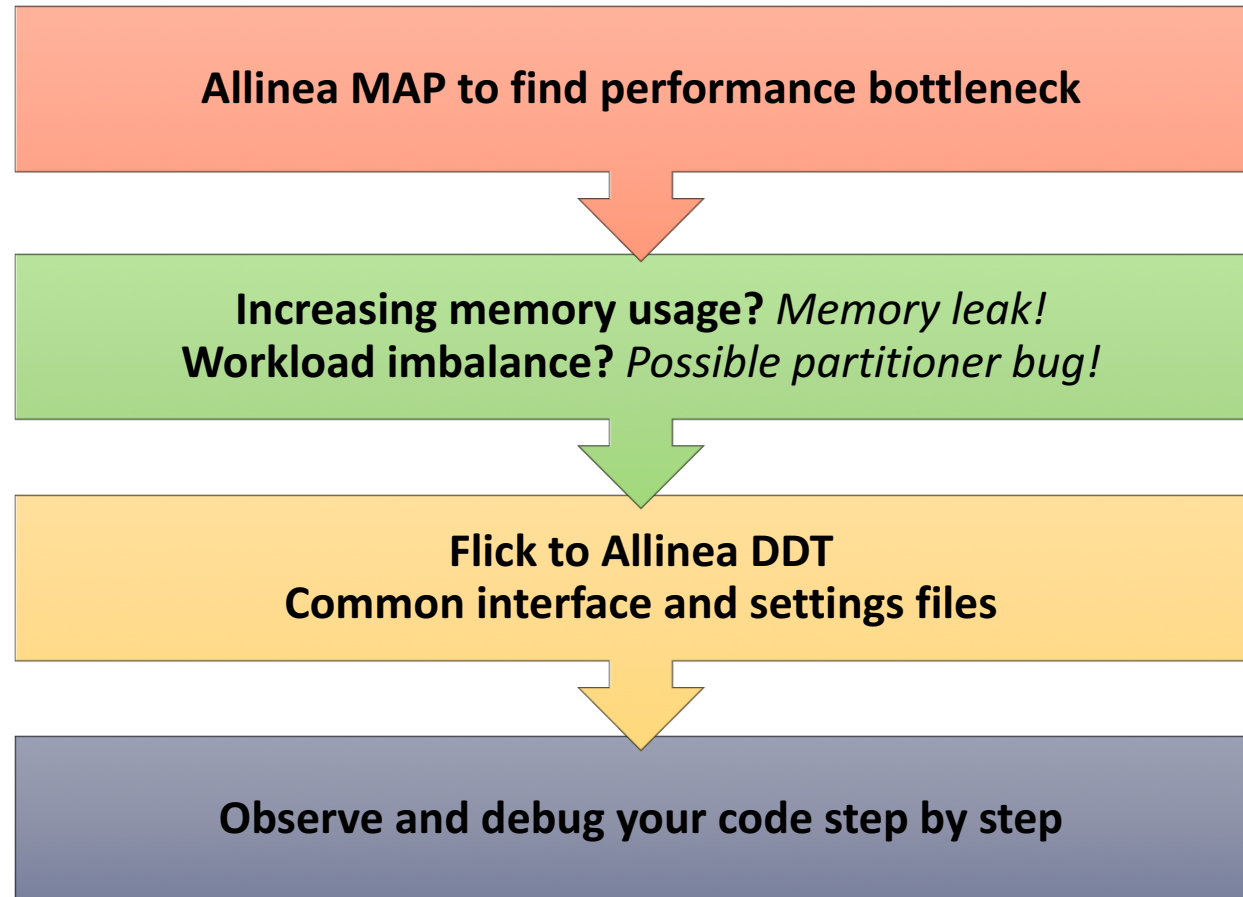
Aug. 8, ATPESC\_2017  
**arm**

# Application Development Workflow



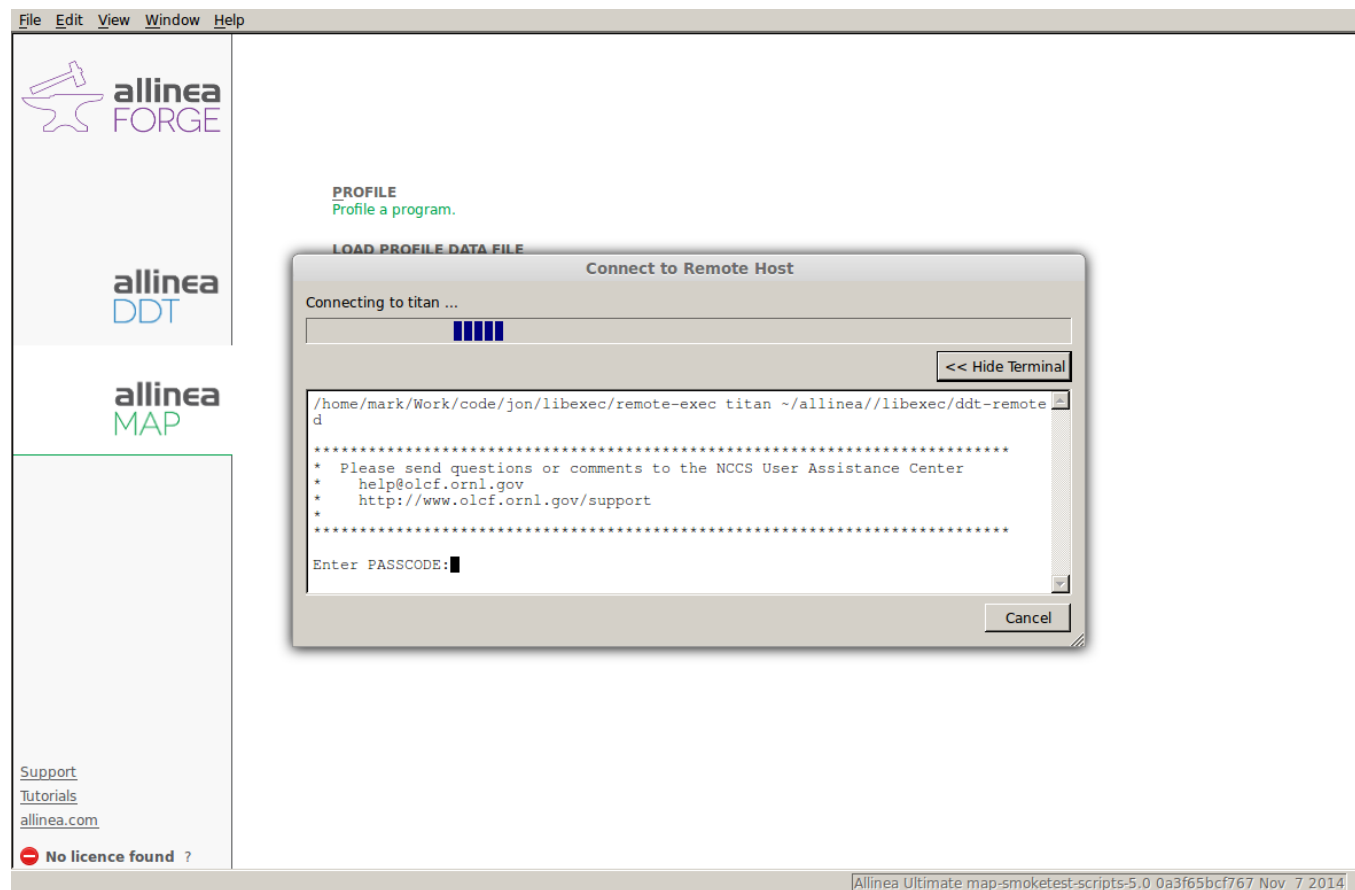


# Hello Allinea Forge!



# HPC means being productive on remote machines

- ✓ Linux
- ✓ OS/X
- ✓ Windows
- ✓ Multiple hop SSH
- ✓ RSA + Cryptocard
- ✓ Uses server license



Aug. 8, ATPESC\_2017  
**arm**

# MAP in a nutshell



Small data files



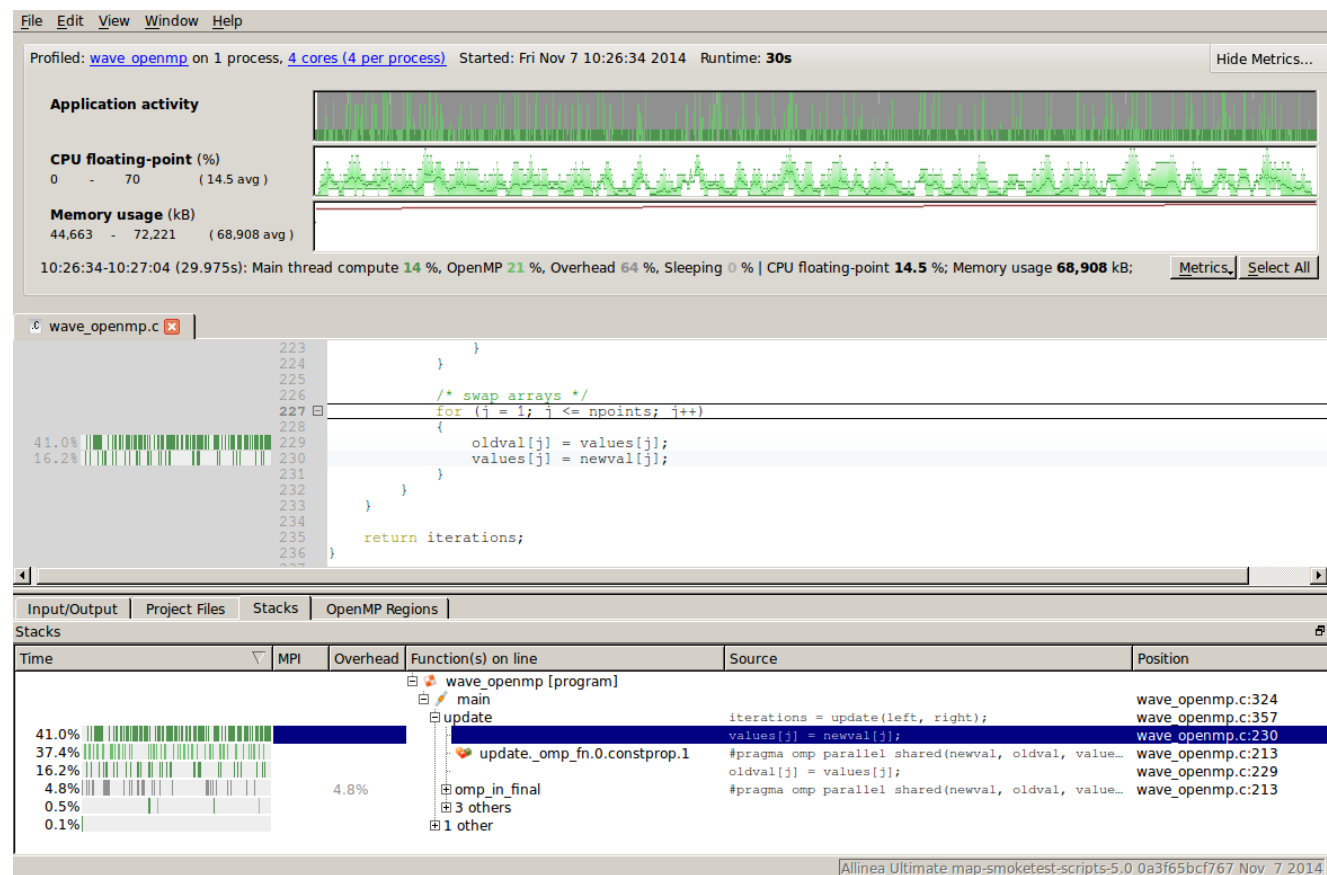
<5% slowdown



No instrumentation



No recompilation



# Above all...

Aimed at any performance problem that matters

- MAP focuses on time

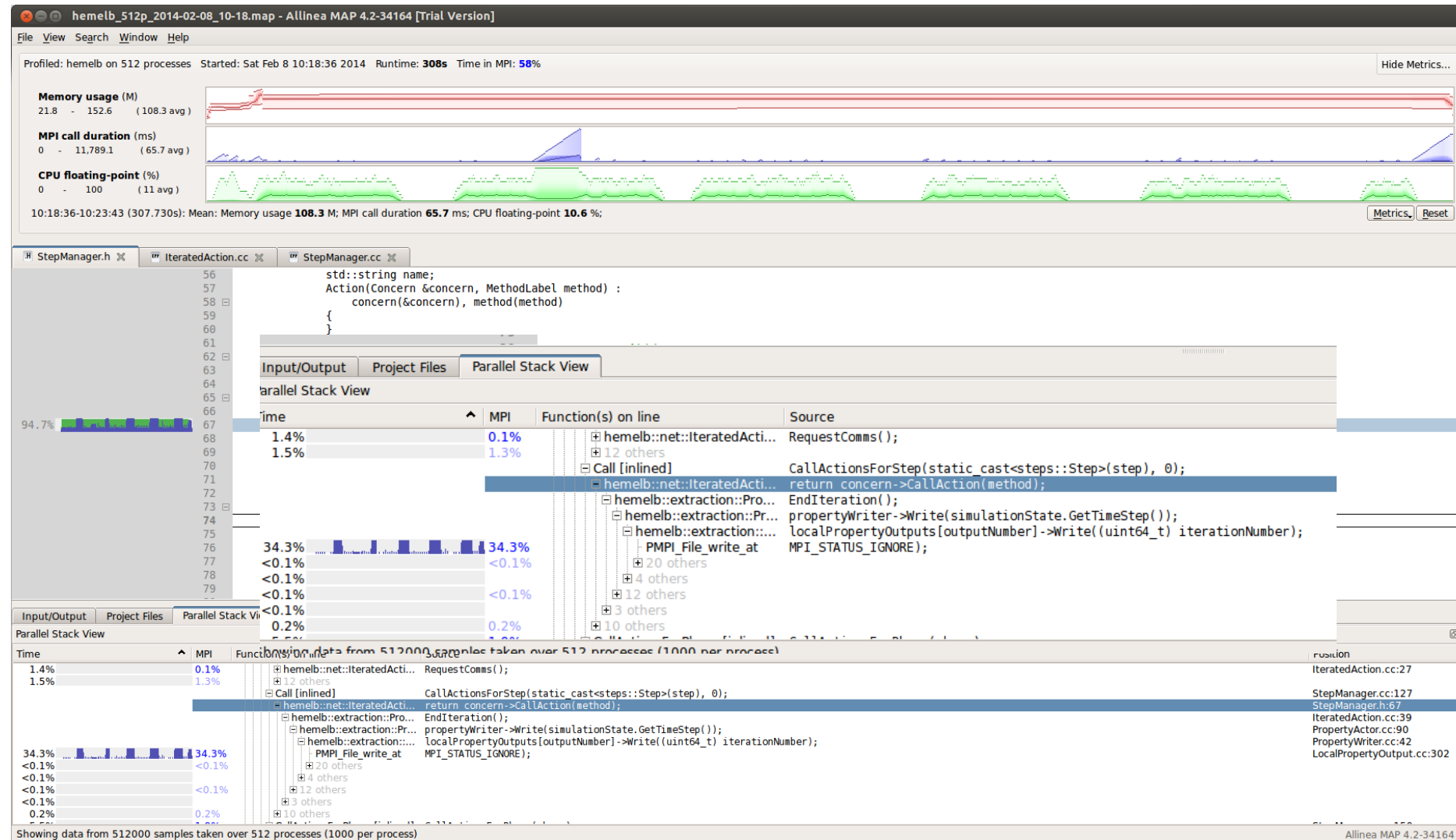
Does not prejudge the problem

- 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

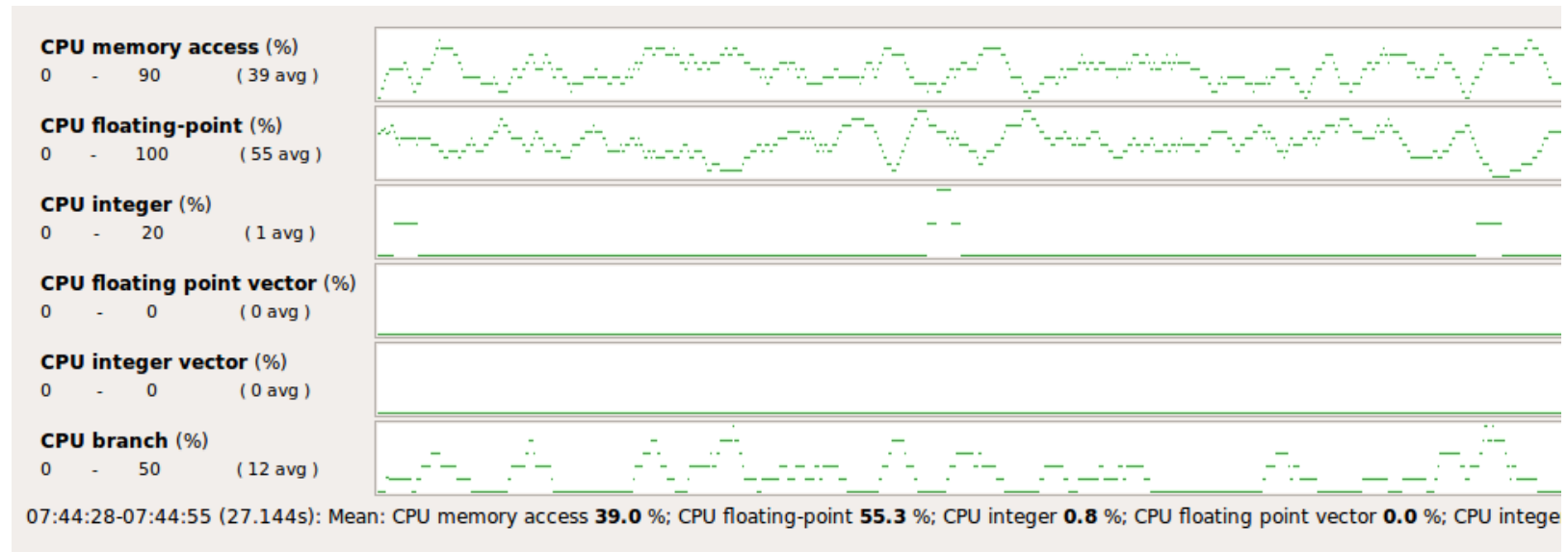


Simple fix... reduce periodicity of output

Aug. 8, ATPESC\_2017  
arm

# Deeper insight into CPU usage

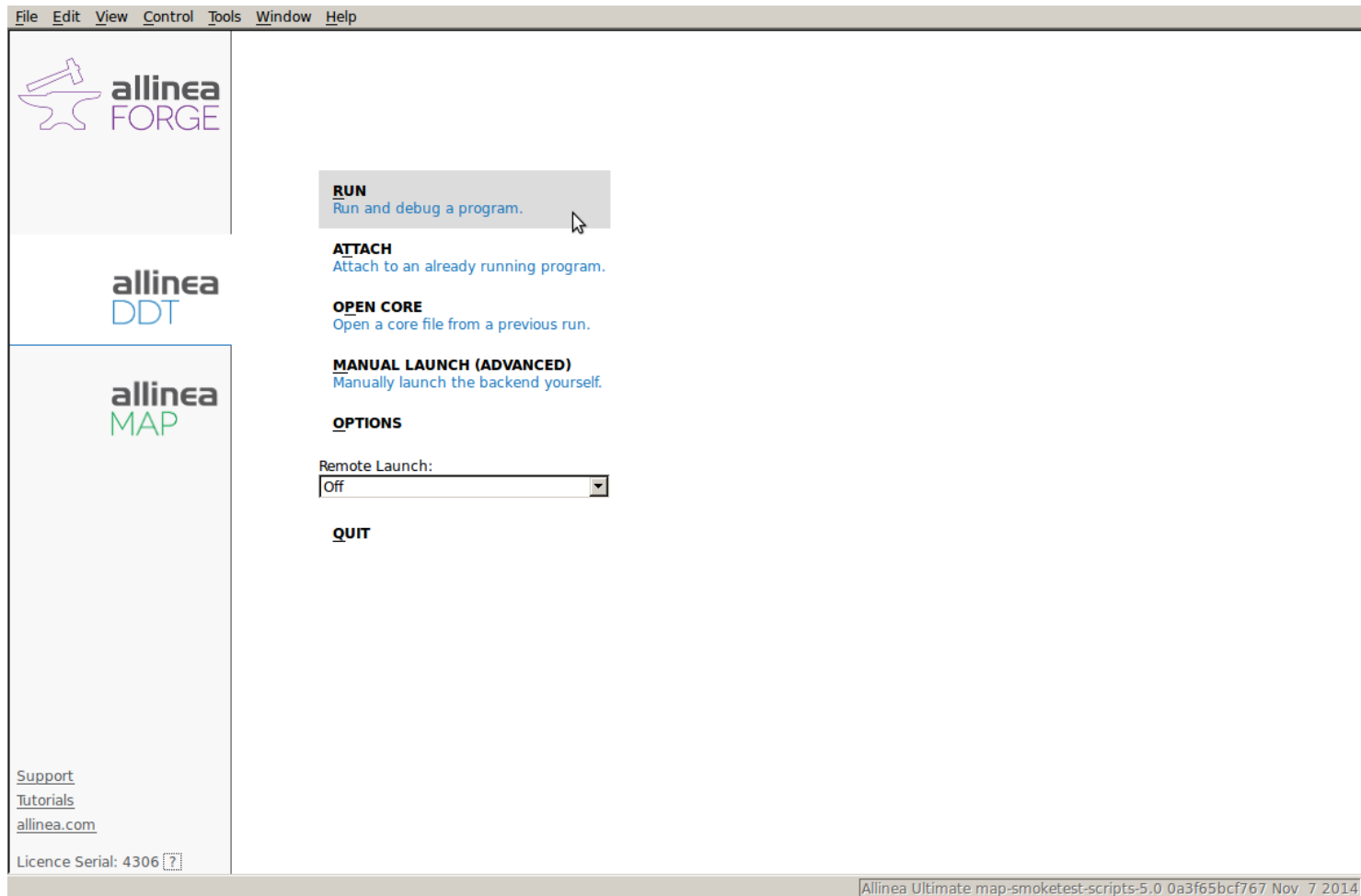
Runtime of application still unusually slow



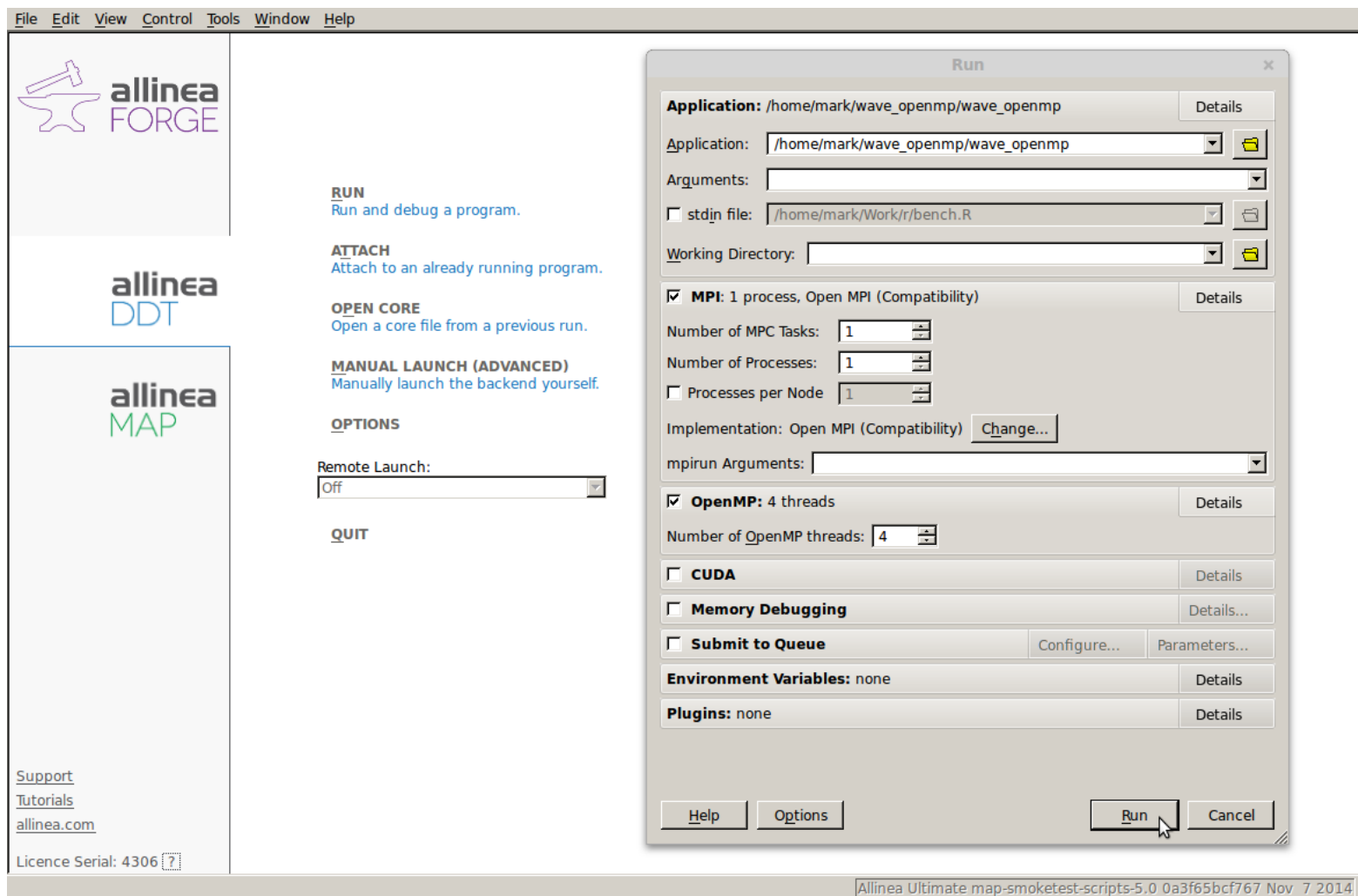
Allinea MAP identifies vectorization close to zero

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 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

InputOutput	Breakpoints	Watchpoints	Tracepoints	Tracepoint Output	Stacks (All)
Tracepoint Output					
Tracepoint	Processes	Values logged			
phone:ISO 85	976, ranks 12, 14-17-22-23, 12	mype		2172-3527 jcol	 2-43 mod    pey
phone:ISO 81	960, ranks 12, 14-17-22-23, 12	ks		1 kmax	pez
phone:ISO 85	942, ranks 12, 14-17-22-23, 12	mype		2172-3527 jcol	 2-43 mod    pey
phone:ISO 81	929, ranks 12, 14-17-22-23, 12	ks		1 kmax	pez
phone:ISO 85	919, ranks 12, 14-17-22-23, 12	mype		2172-3527 jcol	 2-43 mod    pey
phone:ISO 81	898, ranks 12, 14-17-22-23, 12	ks		1 kmax	pez
phone:ISO 85	864, ra 12, 14-				
phone:ISO 81	880, ra 12, 14-				

# The scalable print

for the modern application

## The scalable print alternative

Program Stopped

Process 0:  
Process stopped at watchpoint "rank" in main (watchmatrix.c:45).

Old value: 0  
New value: 1074790400

☒ Always show this window for watchpoints

Continue Pause Pause All

Stop on variable change

## Stop on variable change

```

hello.c
43     else
44         test=-1;
45     }
46
47     void func3()
48     {
49         void* i = (void*) 1;
50         while(i++ || !i)
51             free((void*)i);
52     }
53
54     int main()
55     {
56         typ
57         in'
58     }

```

⚠ This file is newer than your program. Please recompile then restart y

⚠ portability 'i' is of type 'void \*'. When using void pointers in calcul

Left click to add a breakpoint on line 50

Static analysis warning

## Static analysis warnings on code errors

```
&& !strcmp(argv[i], "crash")) {
0;
s", *(char **)argv[i]);
ll se
```

**Program Stopped**

Processes 0-3:

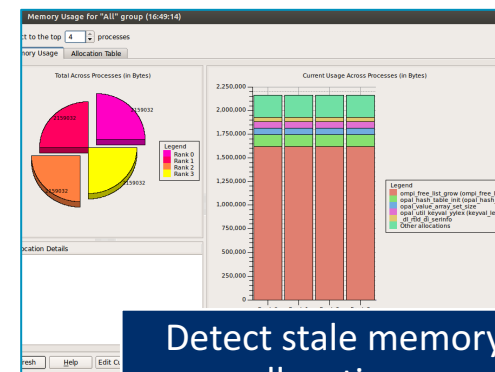
Memory error detected in main (hello.c:118):  
null pointer dereference or unaligned memory access

Note: the latter may sometimes occur spuriously if guard pages are not enabled

Tip: Use the stack list and the local variables to explore the current state and identify the source of the error.

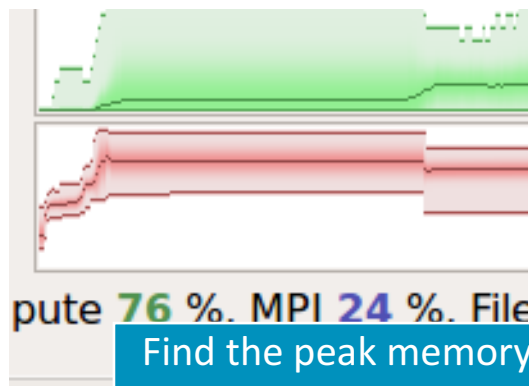
**Detect read/write**

## Detect read/write beyond array bounds

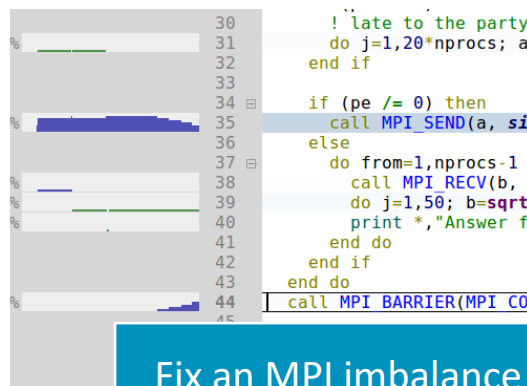


## Detect stale memory allocations

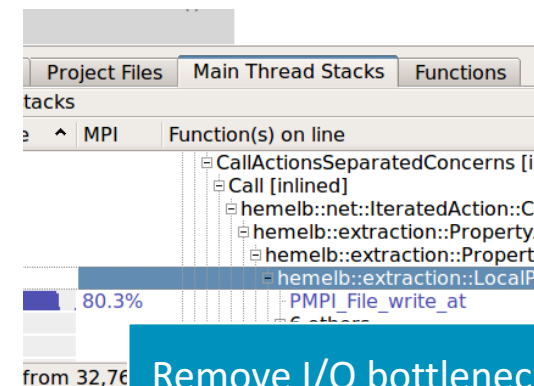
# Six Great Things to Try with Allinea MAP



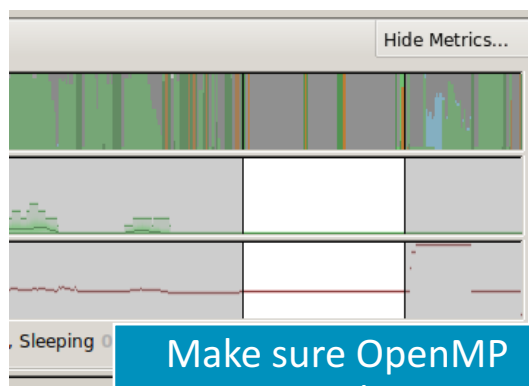
Find the peak memory use



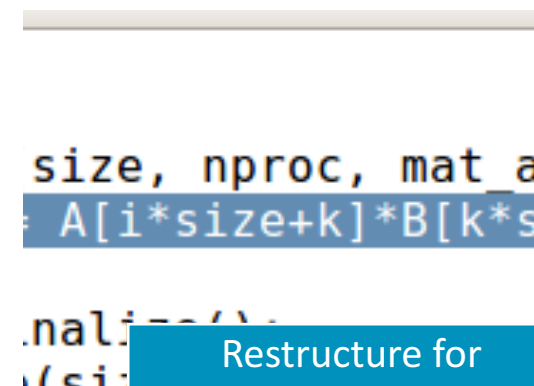
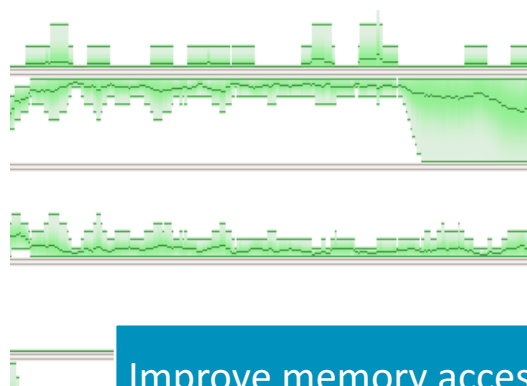
Fix an MPI imbalance



Remove I/O bottleneck



Make sure OpenMP regions make sense



Restructure for vectorization

# Getting started on Theta

Install local client on your laptop

- [www.allinea.com/products/forge/downloads](http://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](mailto: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](http://www.allinea.com) - Trials menu, Resources – “trial guide”

# Thank you for your attention!

Contact:

- [support@allinea.com](mailto:support@allinea.com)
- [support@arm.com](mailto:support@arm.com)

Download a trial for ATPESC (or later)

- <http://www.allinea.com/trials>