



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

Debugging and Profiling your HPC Applications

David Lecomber, CEO and Co-founder

david@allinea.com



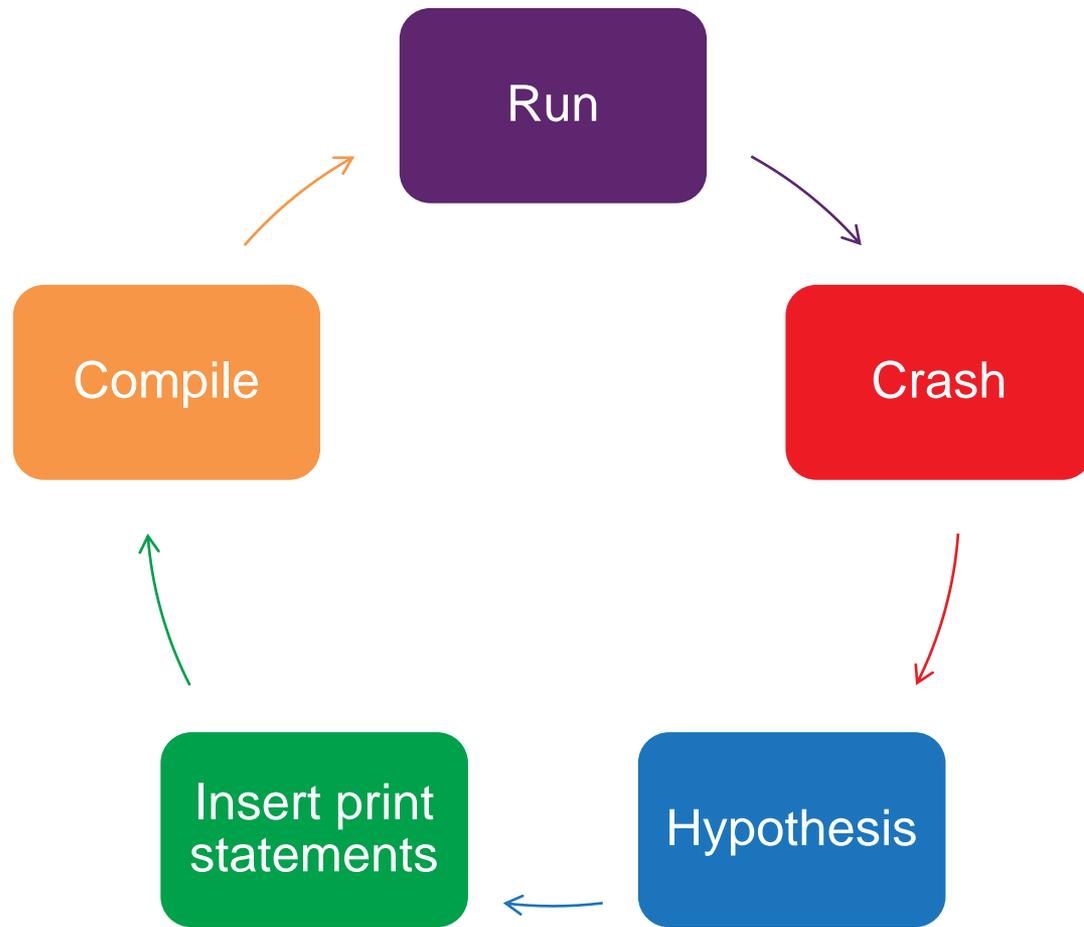
About this talk

- 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

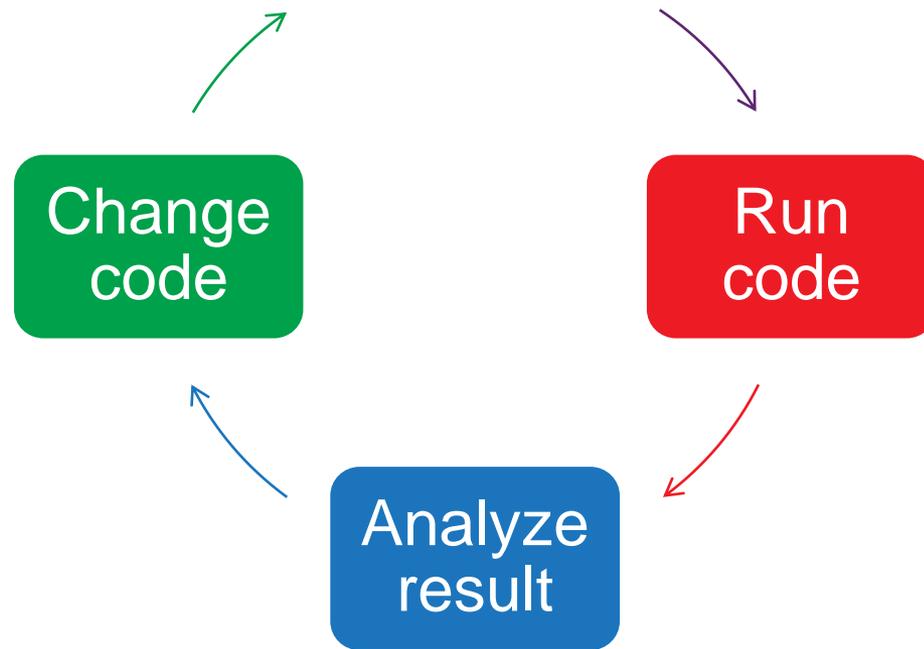
Motivation

- 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...



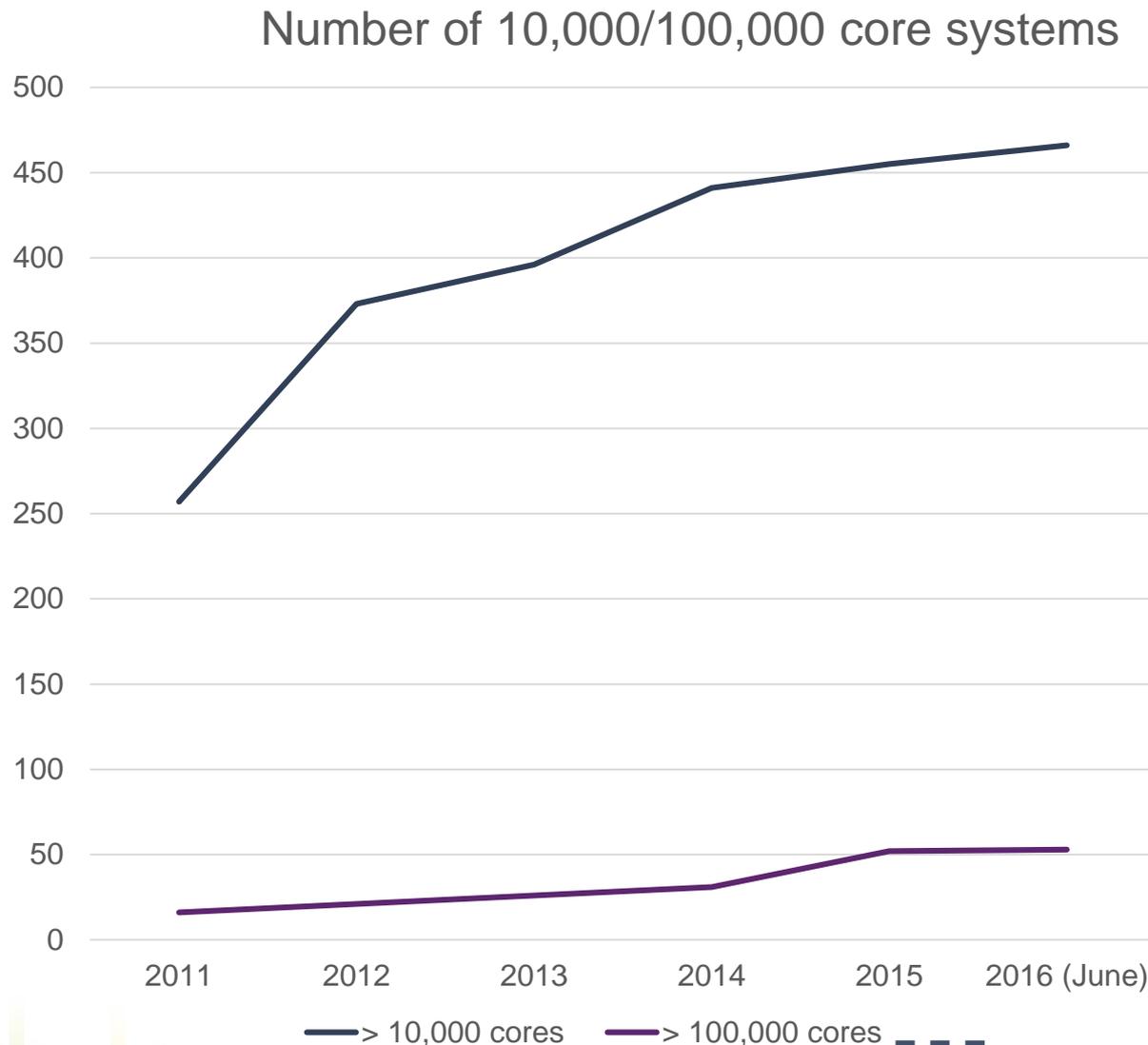
Optimization in Practice



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

Bohrbug

Steady, dependable bug

Heisenbug

Vanishes when you try to debug (observe)

Mandelbug

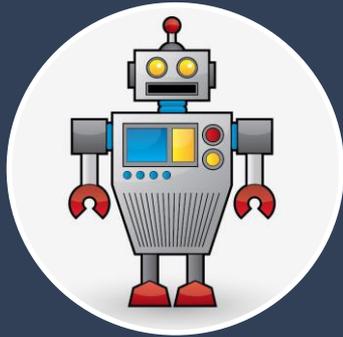
Complexity and obscurity of the cause is so great that it appears chaotic

Schroedinbug

First occurs after someone reads the source file and deduces that it never worked, after which the program ceases to work

- 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
 - 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

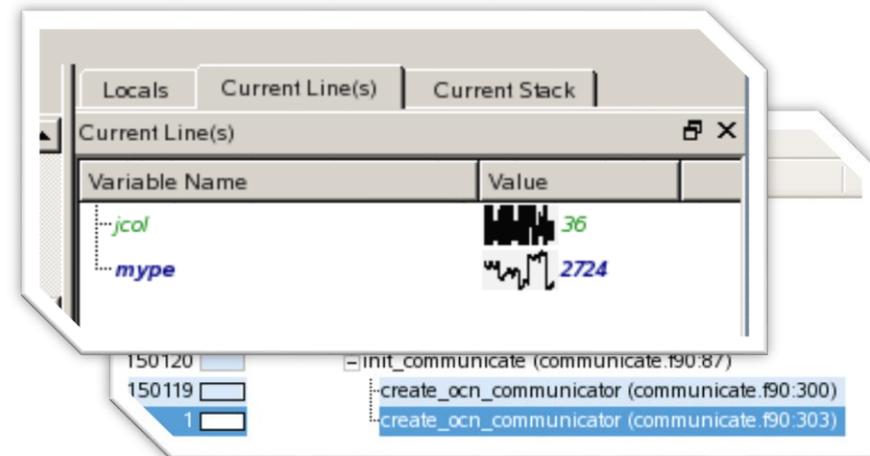
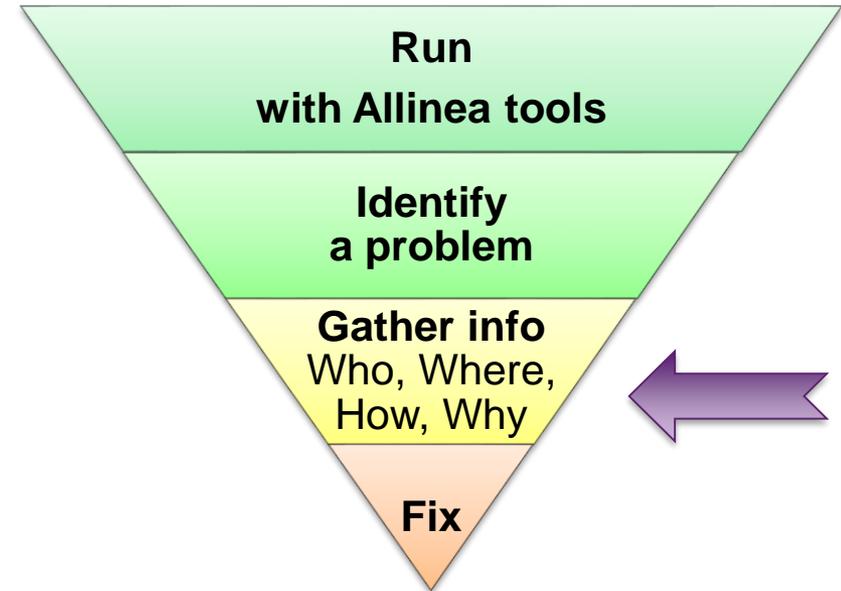


Magic

- Static analysis
- Memory debugging



- 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





Live demo

allinea
DDT



allinea

Favorite Allinea DDT Features for Scale

Stacks (All)

Processes	Function
150119	start
150120	lib_start_main
150120	main
150120	pop (POP.f90 81)
150120	initiate_pop (initial.f90 119)
150120	init_communicate (communicate.f90 87)
150119	create_ocn_communicator (communicate.f90 300)
150119	create_ocn_communicator (communicate.f90 303)

Parallel

Parallel stack view

Current Line(s) | Current Stack

Variable Name	Value
icol	36
myp	2724

Parallel

Automated data comparison: sparklines

Array Expression: bigArray(i,j)

Distributed Array Dimensions: 1

Range of Sx (Distributed): Range of Sj

From: 0 To: 9999

Display: Rows Display: Columns

Only show if: Svalue == 1

	2444	2733	3011	3185	4704	5343	6795	7881	9108	9467
x 0										
1									1	
2										1
3										
4										
5										

Parallel

Parallel array searching

All: 0 1 2

ddt.bin: 1 2

licen_reserver: 0

Current Group: All

Focus on current: Group Process Thread

200004 processes (0-200003) Paused: 200004 Running: 0

Currently selected: 0

Parallel

Step, play, and breakpoints

Process stopped in sched_yield (syscall-template.S:82) with signal SIGSEGV (Segmentation fault). Reason: Origin: kill, signal or raise. The process will probably be terminated if you continue. You can use the stack controls to see what the process was doing at the time.

Stacks

Processes	Threads	Function
1-10	25	main (f:118)
1-10	25	User (f:132)
1-10	25	lib_start_main (f:36)
1-3-7-9-11-13-15-17	11	exchange_1 (f:37)
1-3-7-9-11-13-15-17	11	mpi_recv
1-3-7-9-11-13-15-17	11	mpi_send
1-3-7-9-11-13-15-17	11	mpi_wait
1-3-7-9-11-13-15-17	11	mpi_finalize
1-3-7-9-11-13-15-17	11	exchange_2 (syscall-template.S:82)
1-3-7-9-11-13-15-17	11	exchange_1 (f:35)
1-3-7-9-11-13-15-17	11	mpi_recv
1-3-7-9-11-13-15-17	11	mpi_send
1-3-7-9-11-13-15-17	11	mpi_wait
1-3-7-9-11-13-15-17	11	mpi_finalize

Stack for process 1

Every process in your program has terminated.

Tracepoints

#	Time	Tracepoint	Processes	Values
1	00:13:45	libdomain.f66	libdomain.f66	libdomain.f66
2	00:13:45			
3	00:13:45			

Parallel

Offline debugging



Analyze before optimizing

"Premature optimization is the root of all evil"

Donald Knuth, 1974

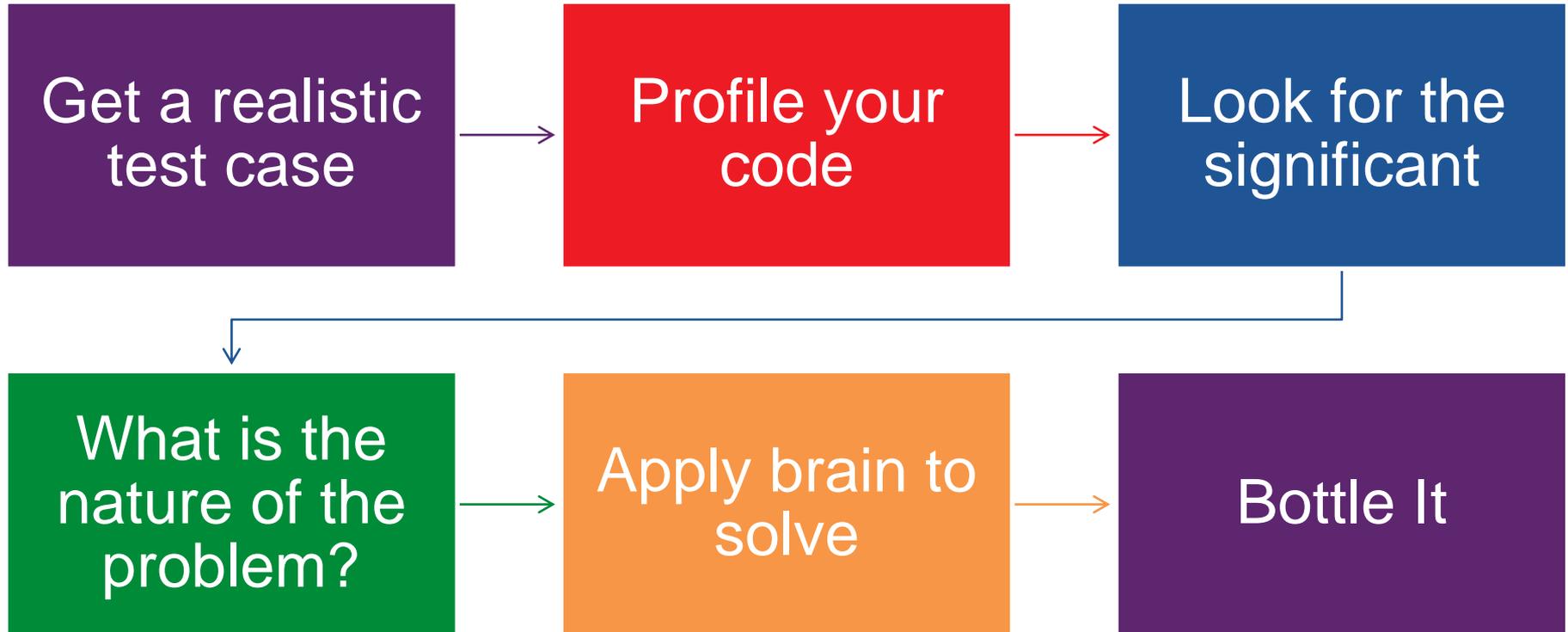
Profiling for performance

- Code optimization can be time-consuming...



– (image courtesy of xkcd.com)

6 steps to improve performance



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 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

5 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

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

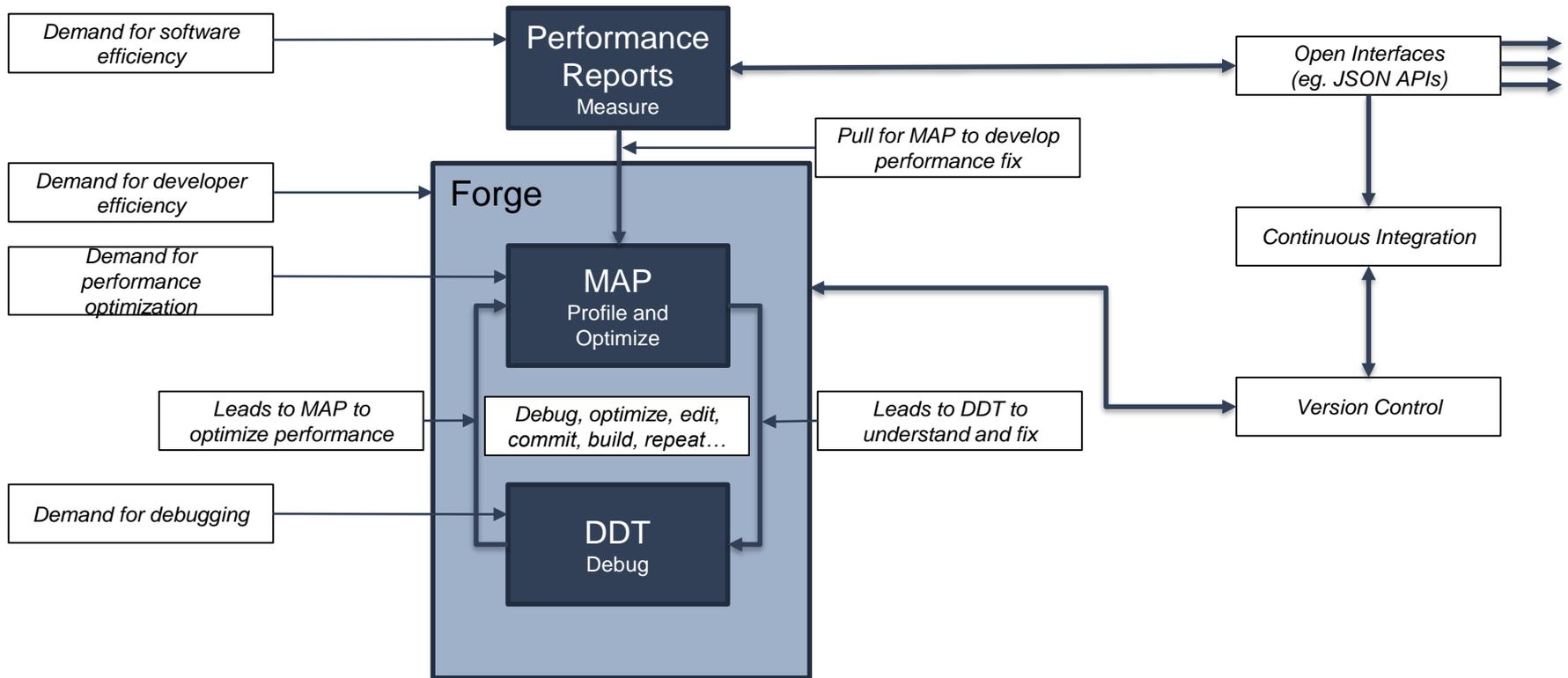
FINISH

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...



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:

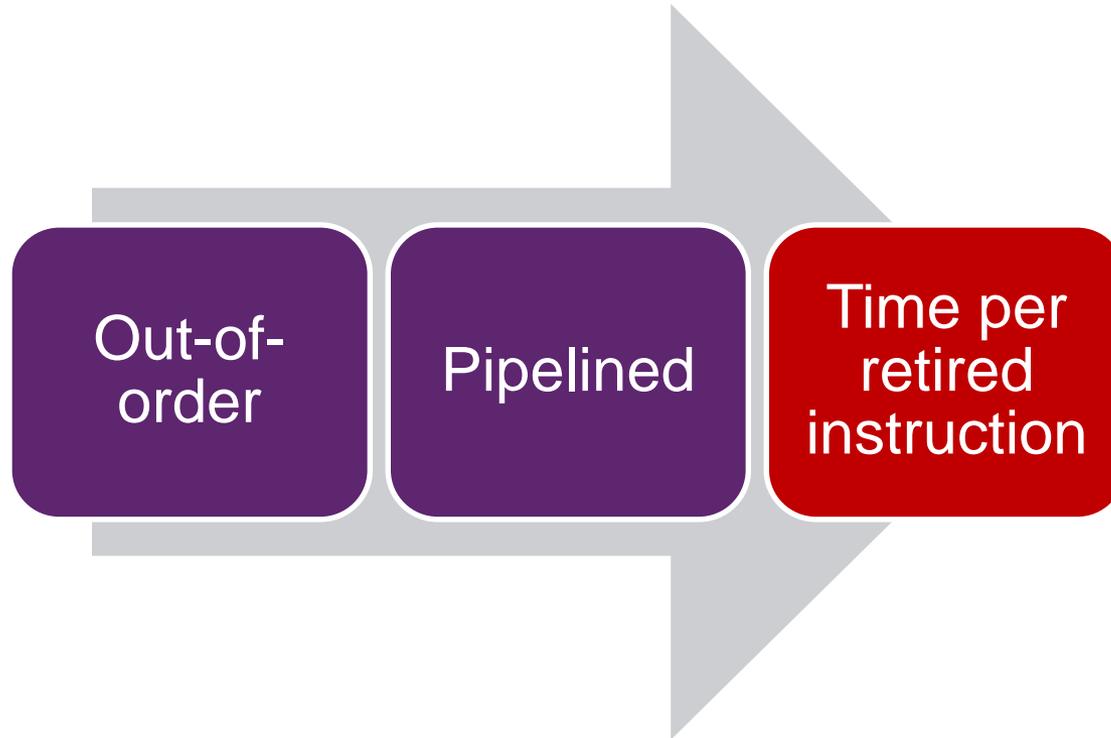
Single-core code	100.0%	
Scalar numeric ops	22.4%	
Vector numeric ops	0.0%	
Memory accesses	77.6%	

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.

“Vectorization, how does it work?”

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%	
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.

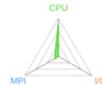
No time is spent in **vectorized instructions**. Check the compiler's vectorization advice to see why key loops could not be vectorized.

↗
+ simple, actionable advice

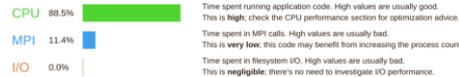
Vectorization, MPI, I/O, memory, energy...



Command: mpixexec -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

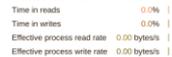
A breakdown of the 88.5% CPU time:



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

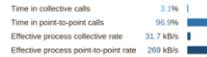
Per-process memory usage may also affect scaling:



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:

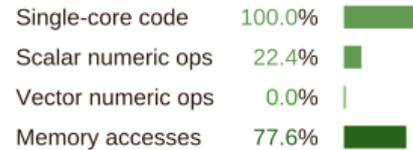


The CPU is responsible for all measured energy usage. Check the CPU breakdown section to see if it is being well-used.

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

CPU

A breakdown of the 88.5% CPU time:

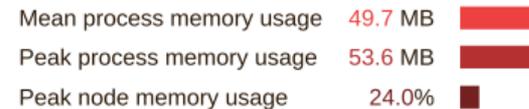


The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

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

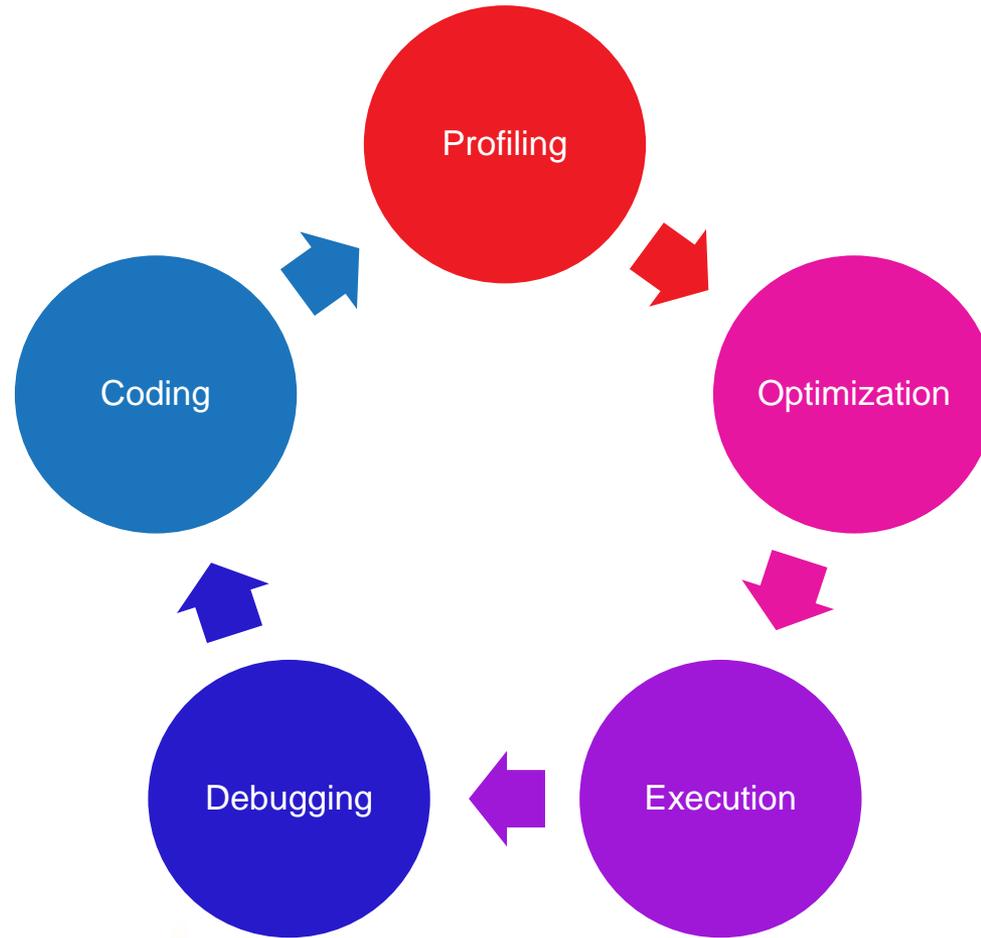
Memory

Per-process memory usage may also affect scaling:

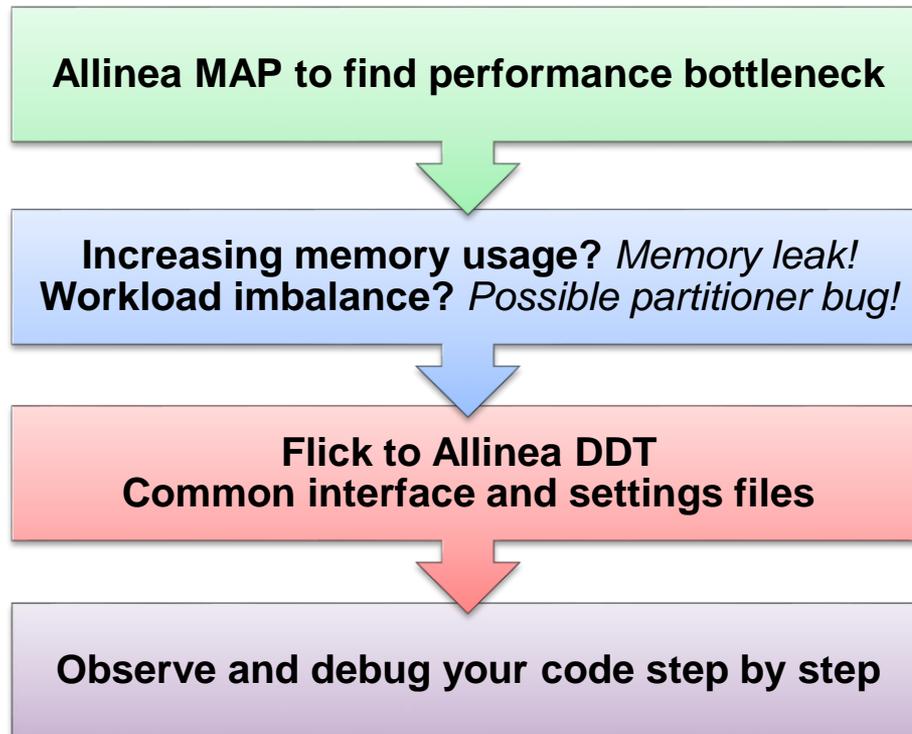


The peak node memory usage is very low. You may be able to reduce the amount of allocation time used by running with fewer MPI processes and more data on each process.

Application Development Workflow



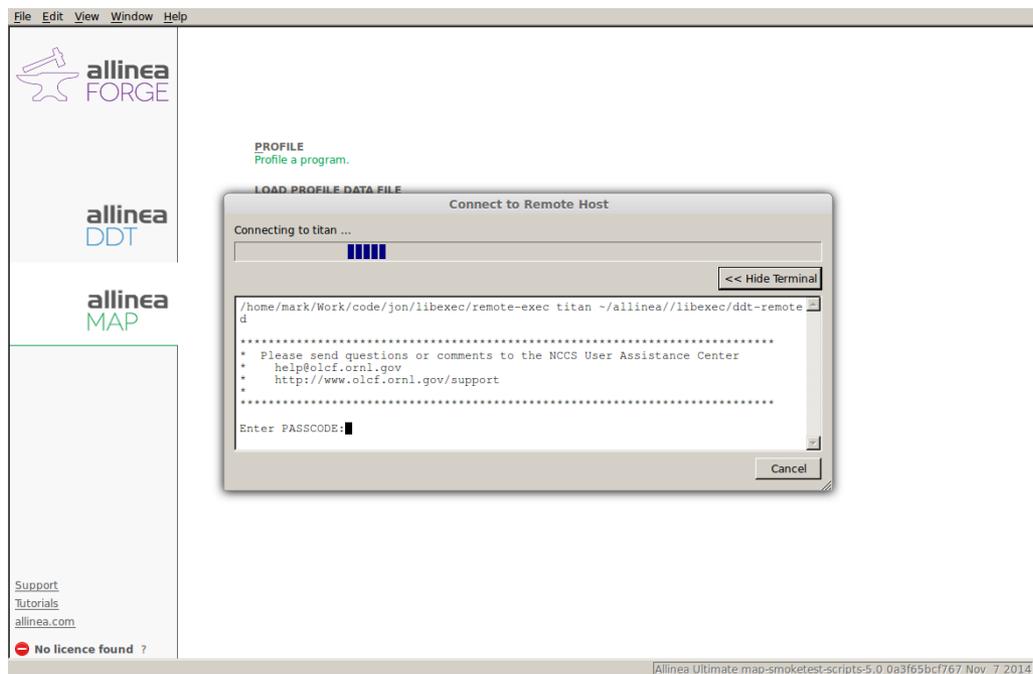
Hello Alinea Forge!



HPC means being productive on remote machines



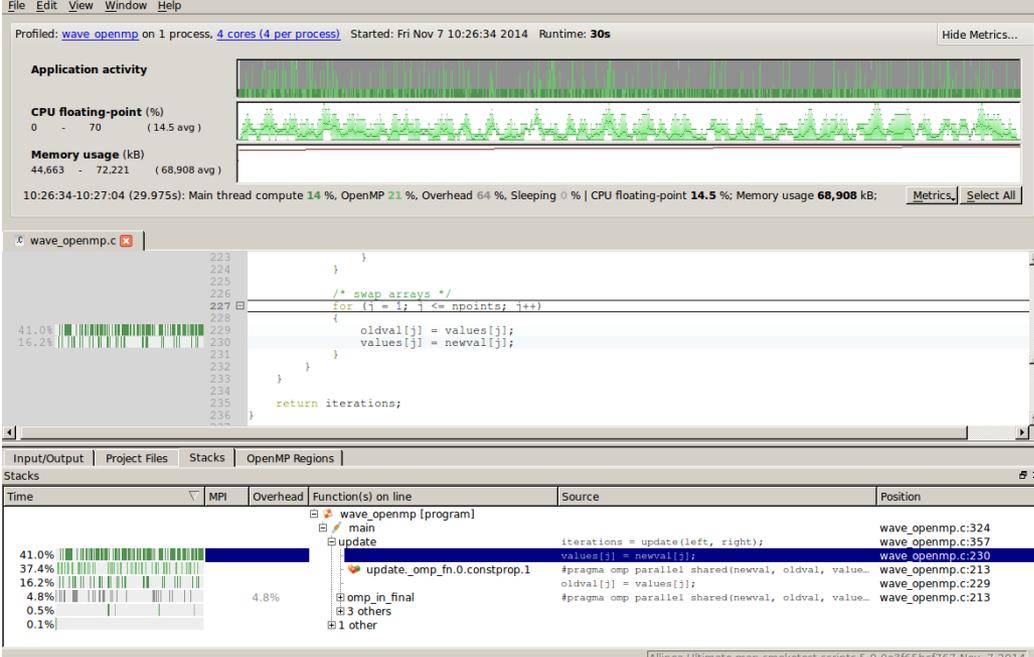
- ✓ Linux
- ✓ OS/X
- ✓ Windows
- ✓ Multiple hop SSH
- ✓ RSA + Cryptocard
- ✓ Uses server license



allinea

MAP in a nutshell

-  Small data files
-  <5% slowdown
-  No instrumentation
-  No recompilation



The screenshot displays the Alinea Ultimate MAP interface. At the top, it shows application activity for 'wave_openmp' on 1 process, 4 cores (4 per process), started on Fri Nov 7 10:26:34 2014, with a runtime of 30s. Below this, there are graphs for CPU floating-point usage (0-70% with a 14.5% avg) and memory usage (44,663 - 72,221 KB with a 68,908 avg). A status bar indicates: '10:26:34-10:27:04 (29.975s): Main thread compute 14 %, OpenMP 21 %, Overhead 64 %, Sleeping 0 % | CPU floating-point 14.5 %; Memory usage 68,908 kB; Metrics Select All'.

The code editor shows a snippet of C code with OpenMP pragmas:

```

223     }
224
225
226     /* swap arrays */
227     for (i = 1; i <= npoints; i++)
228     {
229         oldval[j] = values[j];
230         values[j] = newval[j];
231     }
232
233     }
234
235     return iterations;
236 }
    
```

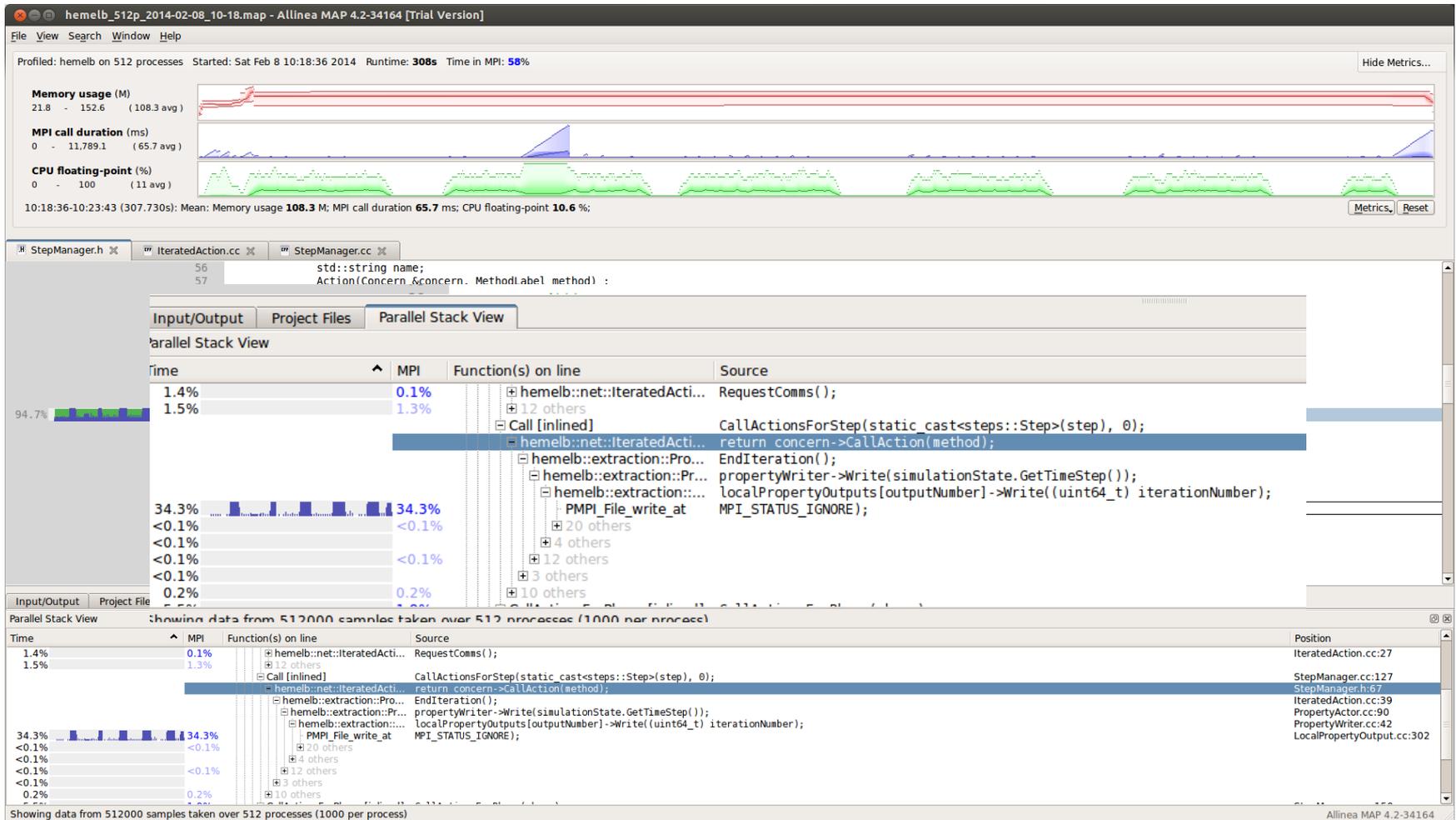
The 'Stacks' table at the bottom provides a detailed breakdown of execution time:

Time	MPI	Overhead	Function(s) on line	Source	Position
41.0%			main	iterations = update(left, right);	wave_openmp.c:324
37.4%			update	values[j] = newval[j];	wave_openmp.c:230
16.2%			update_omp_fn.0.constprop.1	#pragma omp parallel shared(newval, oldval, value...	wave_openmp.c:213
4.8%		4.8%	omp_in_final	oldval[j] = values[j];	wave_openmp.c:229
0.5%			others	#pragma omp parallel shared(newval, oldval, value...	wave_openmp.c:213
0.1%			other		

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

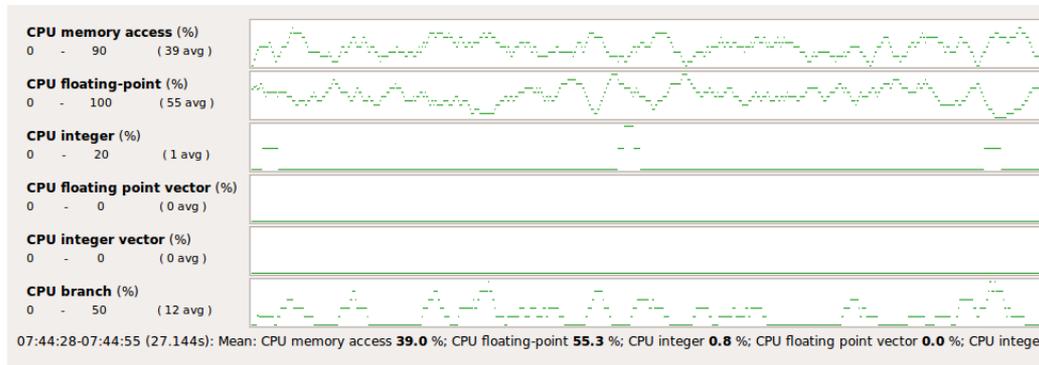
Scaling issue – 512 processes



Simple fix... reduce periodicity of output

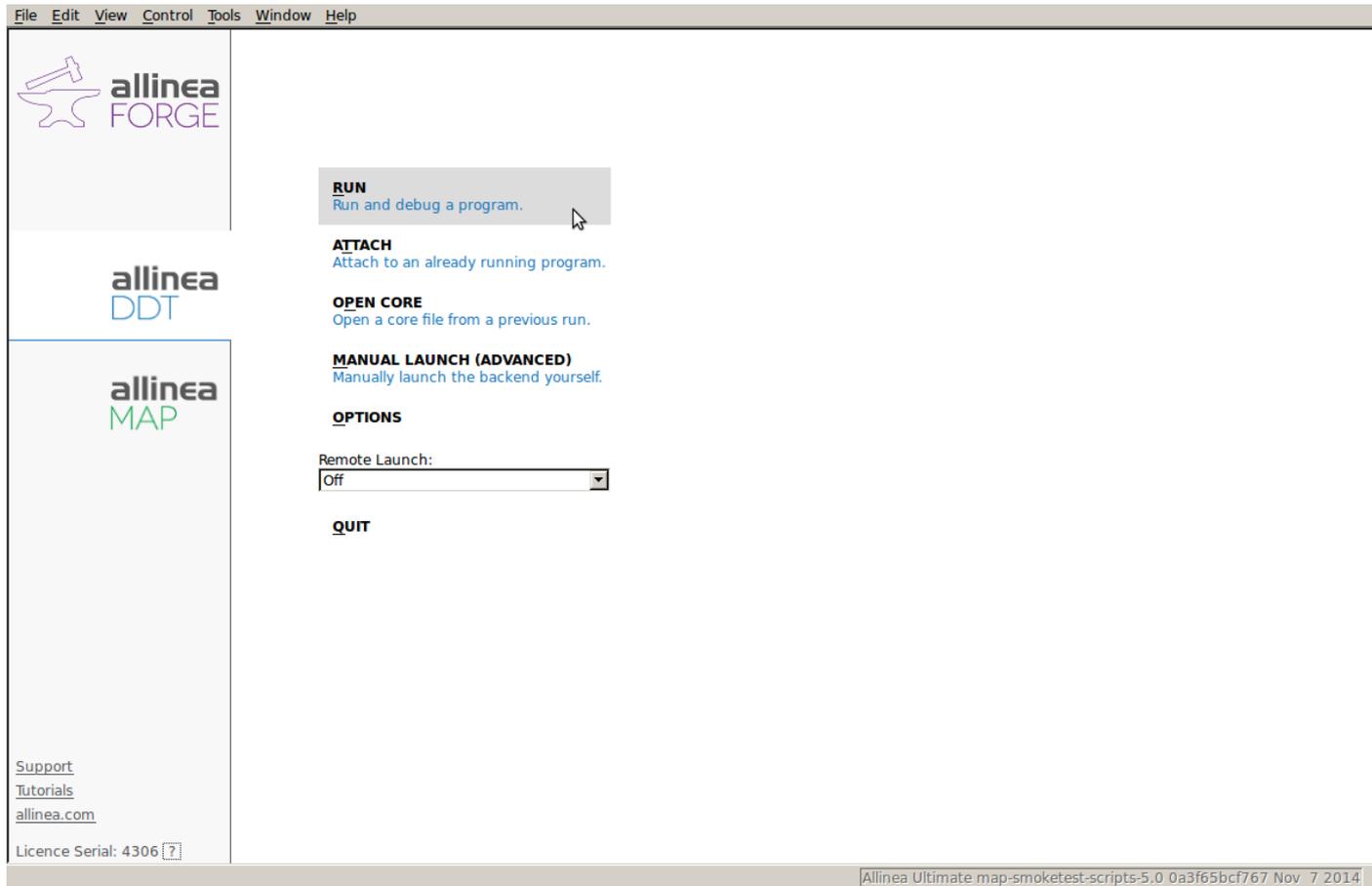
Deeper insight into CPU usage

- Runtime of application still unusually slow

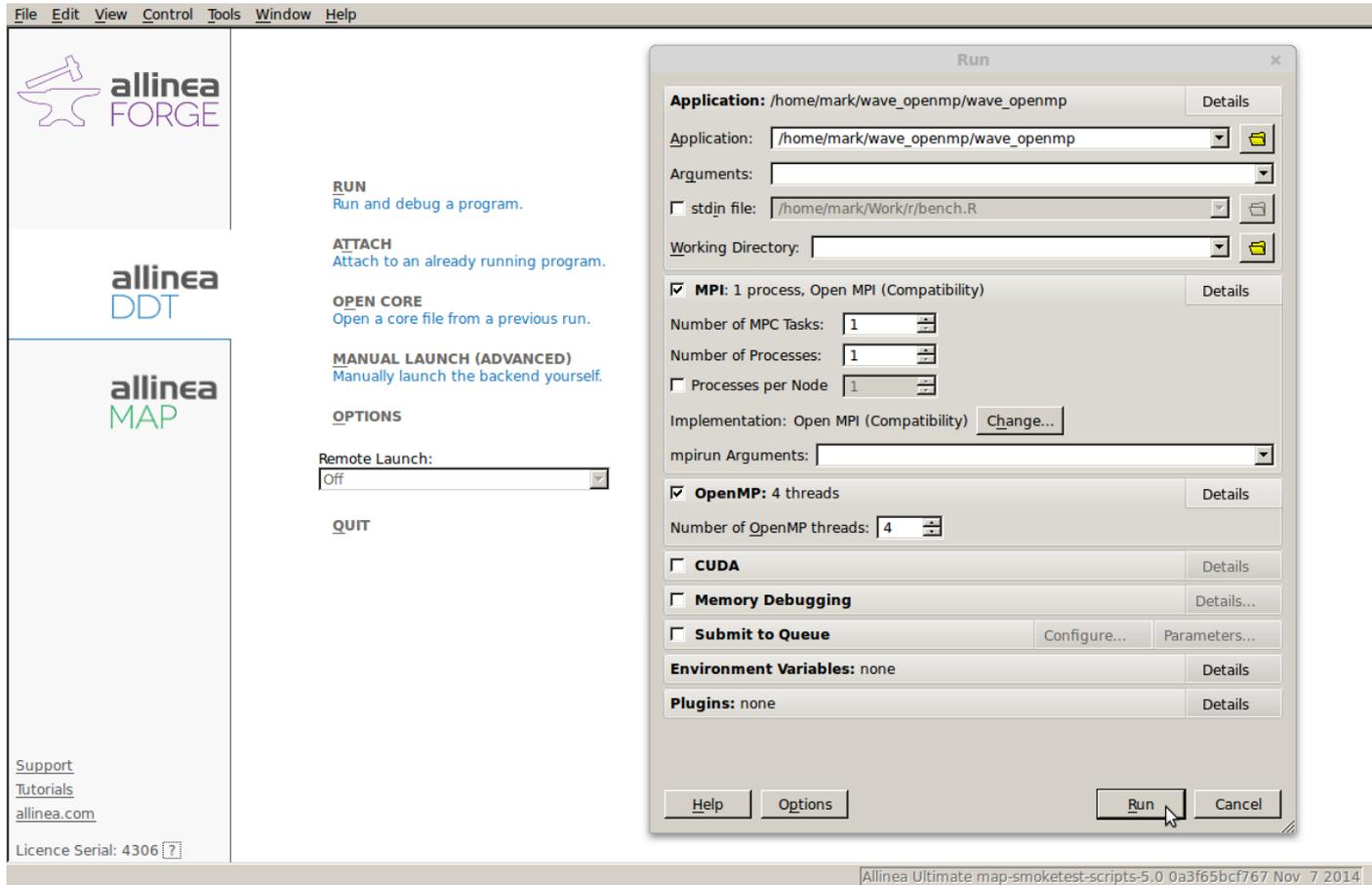


- Allinea MAP identifies vectorization close to zero
- Why? Time to switch to a debugger!

While still connected to the server we switch to the debugger



It's already configured to reproduce the profiling run



The screenshot shows the allinea MAP interface with the Run dialog box open. The Run dialog is configured with the following settings:

- Application:** /home/mark/wave_openmp/wave_openmp
- Arguments:** (empty)
- stdin file:** /home/mark/Work/r/bench.R
- Working Directory:** (empty)
- MPI: 1 process, Open MPI (Compatibility)**
 - Number of MPC Tasks: 1
 - Number of Processes: 1
 - Processes per Node: 1
 - Implementation: Open MPI (Compatibility) [Change...](#)
 - mpirun Arguments: (empty)
- OpenMP: 4 threads**
 - Number of OpenMP threads: 4
- CUDA**
- Memory Debugging**
- Submit to Queue** [Configure...](#) [Parameters...](#)
- Environment Variables:** none
- Plugins:** none

The Run dialog has buttons for [Help](#), [Options](#), [Run](#), and [Cancel](#).

The main interface shows the following options:

- RUN** Run and debug a program.
- ATTACH** Attach to an already running program.
- OPEN CORE** Open a core file from a previous run.
- MANUAL LAUNCH (ADVANCED)** Manually launch the backend yourself.
- OPTIONS**
 - Remote Launch: Off
- QUIT**

Support
Tutorials
allinea.com
Licence Serial: 4306 [?]

Allinea Ultimate map-smoketest-scripts-5.0 0a3f65bcf767 Nov 7 2014

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

Tracepoint	Processes	Values logged
vhone 93085	976, ranks 12,14-17,22-23,12	mype 2170-3527 jcol 2-48 mod pay
vhone 93081	900, ranks 12,14-17,22-23,12	ks 1 kmax pec
vhone 93085	942, ranks 12,14-17,22-23,12	mype 2170-3527 jcol 2-48 mod pay
vhone 93081	929, ranks 12,14-17,22-23,12	ks 1 kmax pec
vhone 93085	919, ranks 12,14-17,22-23,12	mype 2170-3527 jcol 2-48 mod pay
vhone 93081	898, ranks 12,14-17,22-23,12	ks 1 kmax pec
vhone 93085	884, r 12,14-	
vhone 93081	860, r 17 14-	

The scalable print alternative

```

for (i = 0 ; i < SIZE M; i++)
  for (j = 0 ; j < SIZE O; j++)
    C[i][j] = 0;

for (i = 0 ; i < SIZE M; i++)
  for (j = 0 ; j < SIZE N; j++)
    for (k = 0 ; k < SIZE O; k++)
      C[i][j] += A[i][k] * B[k][j];
  
```

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

```

hello.c
This file is newer than your program. Please recompile then restart yo

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 }
portability 'i' is of type 'void *'. When using void pointers in calcula
Left click to add a breakpoint on line 50

55 {
56     ty|
57     ty|
58     in|
  
```

Static analysis warnings on code errors

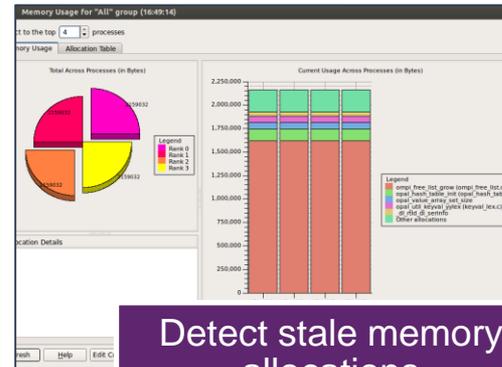
```

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

Detect read/write beyond array bounds



Detect stale memory allocations

Six Great Things to Try with Allinea MAP



compute 76 % MPI 24 % File

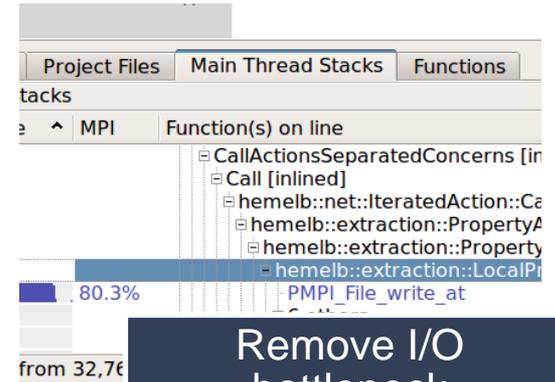
Find the peak memory use

```

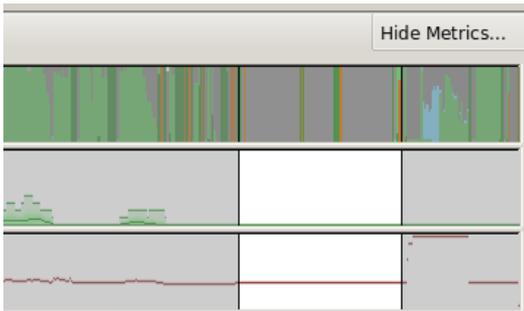
30      ! late to the party
31      do j=1,20*nprocs; a
32
33
34
35      if (pe /= 0) then
36          call MPI_SEND(a, si
37
38      else
39          do from=1,nprocs-1
40              call MPI_RECV(b,
41              do j=1,50; b=sqrt
42              print *, "Answer f
43          end do
44          end if
45      end do
46      call MPI_BARRIER(MPI CO

```

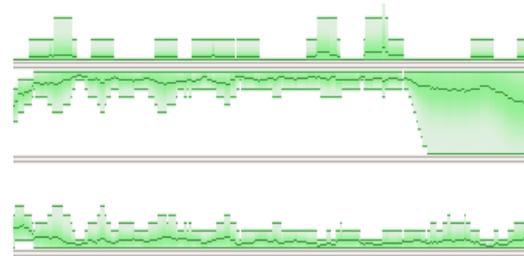
Fix an MPI imbalance



Remove I/O bottleneck



Make sure OpenMP regions make sense



Improve memory access

```

size, nproc, mat a
A[i*size+k]*B[k*s

```

Restructure for vectorization

Getting started on Mira/Cooley

- Install local client on your laptop
 - 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@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
 - Download examples from www.allinea.com - Trials menu, Resources – “trial guide”
- 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>