



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

Accelerating Real Applications

Best Practices for Profiling and Debugging Complex Code

Beau Paisley

Senior Solutions Architect US



Allinea: The industry standard tools for HPC



(and hundreds more)

allinea



We have enjoyed a long and productive relationship with Allinea to scale and deploy DDT on Titan and previous systems. We now see MAP as a performance tool that will help our users with the transition from Titan to Summit by providing a portable performance analysis solution.

— Buddy Bland, Project Director for the Oak Ridge Leadership Computing Facility

Best Practices for Profiling and Debugging Complex Code

In the beginning

- Offloading a simple kernel

Real-world complexity

- Understanding and analysing real application performance

Science: it works

- Profiling and debugging in extreme conditions

Best Practices for Profiling and Debugging Complex Code

In the beginning

- Offloading a simple kernel



Real-world complexity

- Understanding and analysing real application performance

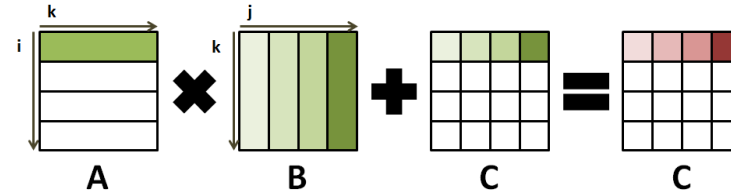


Science: it works

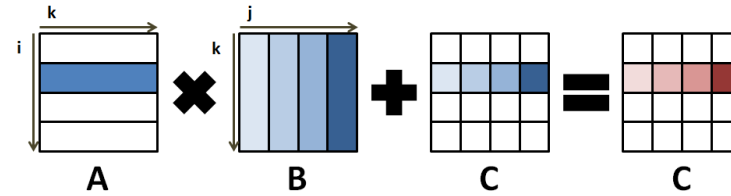
- Profiling and debugging in extreme conditions

In the beginning: offloading a simple multiplication kernel

Process master:



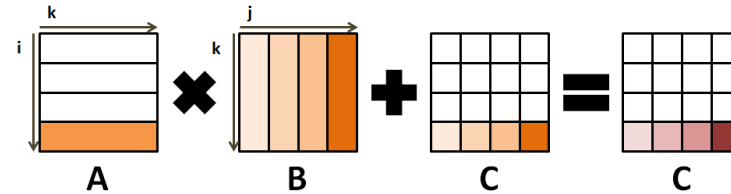
Process slave 1:



...

...

Process slave n:



In the beginning: offloading a simple multiplication kernel

```
53 void mmult(int size, int nslices, double *A, double *B, double *C)
54 {
55     for(int i=0; i<size/nslices; i++)
56     {
57         for(int j=0; j<size; j++)
58         {
59             double res = 0.0;
60
61             for(int k=0; k<size; k++)
62             {
63                 res += A[i*size+k]*B[k*size+j];
64             }
65
66             C[i*size+j] += res;
67         }
68     }
69 }
```

Phase 1: Profile our simple matrix multiplication kernel

Running the example program:

```
$ mpiexec -n 8 ./mmult1.exe
```

Profiling the example program:

```
$ map mpiexec -n 8 ./mmult1.exe
```


Phase 3: A correctly-implemented matrix multiplication kernel!

That little demo is *nothing like the real world* at all

In the beginning

- Offloading a simple kernel



Real-world complexity

- Understanding and analysing real application performance



Science: it works

- Profiling and debugging in extreme conditions

Introducing a real application: Discover DeNovo

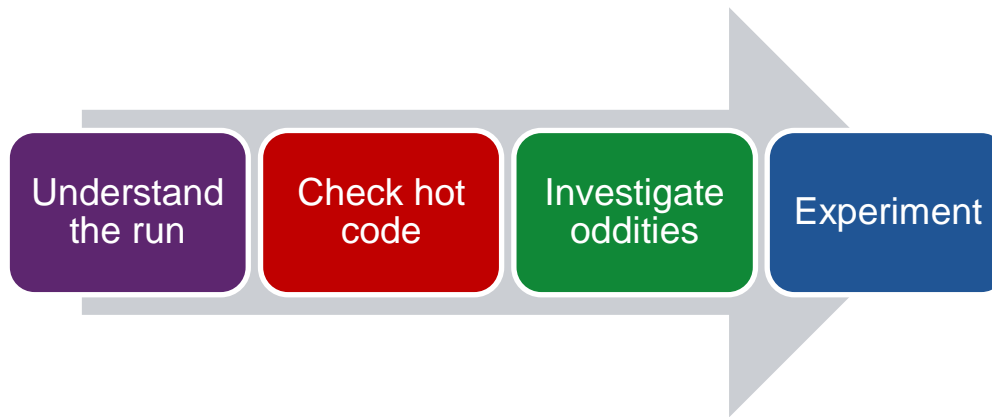
Matrix multiply example:

Language	files	blank	comment	code
C	1	39	0	151

Discover DeNovo, a genome assembly code:

Language	files	blank	comment	code
C++	312	15898	14797	99857
C/C++ Header	405	15219	15718	47118
Bourne Shell	9	5107	5878	32283
m4	12	971	100	8456
make	4	651	1600	3580
SUM:	742	37846	38093	191294

Introducing a real application: Discover DeNovo



Phases

- **Stacks** and **OpenMP regions**
- What application **intends** and **does**

Low-level

- **Functions**: low-level time
- **Memory** or **FPU** bound? **Vectorized**?

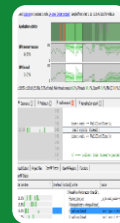
Metrics

- Look for **slopes**, **spread** and **trends**

Which
lines of
code **are**
hot?



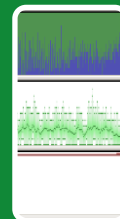
Should
they be?



Spread implies
task imbalance



Slope implies
workload
imbalance



Trends over time
are often leaks or
algorithmic
oversights

Observation

Hypothesis

Experiment

On the subject of making mistakes, what about “Phase 2...”?

Demo output from our matrix multiplication example:

```
2: Receiving matrices...
3: Receiving matrices...
...
6: Processing...
7: Processing...
0: Processing...
...
0: Receiving result matrix...
7: Sending result matrix...
0: Done.
```

```
real    0m2.675s
user    0m7.490s
sys     0m2.561s
```

On the subject of making mistakes, what about “Phase 2...”?

More typical output after offloading a real-world kernel:

```
1: Receiving matrices...
```

```
7: Receiving matrices...
```

```
0: Sending matrices...
```

```
...
```

```
7: Processing...
```

```
0: Processing...
```

```
CUDA error
```

```
-----  
MPI_ABORT was invoked on rank 1 in communicator MPI_COMM_WORLD  
with errorcode 77.
```

```
NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.  
You may or may not see output from other processes, depending on  
exactly when Open MPI kills them.
```

Shared interface with integrated GPU + CPU memory debugging

The screenshot displays a debugger interface with the following components:

- Menu Bar:** File, Edit, View, Control, Tools, Window, Help.
- Toolbar:** Contains icons for running, stepping, and other debugging actions.
- Threads Panel:** Shows two threads, 1 and 2, with thread 1 selected.
- Project Files Panel:** Shows the project structure, including Application Code, Sources, and External Code. The file mmult2.c is selected.
- Source Code Panel:** Displays the source code of mmult2.c. The current line is 72, which is `MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // my rank`. The code includes headers for MPI and defines variables for rank, size, and filename.
- Locals Panel:** Shows the current line(s) and variable names, but no values are displayed.
- Stacks Panel:** Shows the stack frames. The top frame is `main (mmult2.c:72)`, and the bottom frame is `orte_progress_thread_engine`.
- Evaluate Panel:** Shows the expression and value, but no expression is entered.

```
63
64 int main(int argc, char *argv[])
65 {
66     int myrank, nproc, size, slice;
67     double *mat_a, *mat_b, *mat_c, *tmp;
68     char filename[32];
69     MPI_Status st;
70
71     MPI_Init (&argc, &argv);
72     MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // my rank
73     MPI_Comm_size(MPI_COMM_WORLD, &nproc); // number of processors
74
75     if(argc > 3)
76     {
77         if(myrank == 0)
78         {
79             printf("Usage: ./mmult3_c.exe SIZE FILENAME\n \
80                  \tSIZE: size of the matrix to compute (default is %d)\n \
81                  \tFILENAME: output matrix file name (default is %s)\n", DEFAULT_SIZE, DEFAULT_FN);
82         }
83     }
```


Just hit Play!

The screenshot shows a debugger interface with a central error message window. The error message is from 'Process 0' and states: 'Kernel 1 Thread <<<(0,25,0),(0,0,0)>>> stopped in mmult_kernel (mmult2.cu:32) with signal CUDA_EXCEPTION_1 (Lane Illegal Address). Your program will probably be terminated if you continue. You can use the stack controls to see what the process was doing at the time.' There are 'Continue' and 'Pause' buttons at the bottom of the error window. The 'Always show this window for signals' checkbox is checked.

The debugger interface includes a menu bar (File, Edit, View, Control, Tools, Window, Help), a toolbar with various icons, and a status bar at the bottom showing 'Ready'. The 'Threads' panel at the top shows two threads, 1 and 2. The 'Project Files' panel on the left shows a tree view of the project structure, including 'Application Code', 'Sources', and 'External Code'. The 'mmult2.c' file is open in the editor, showing lines 63 to 83. The 'Locals' panel on the right shows the current line(s) and variable names. The 'Stacks' panel at the bottom shows the current stack frame, 'main (mmult2.c:72)', and the 'orte_progress_thread_engine' function.

File Edit View Control Tools Window Help

Focus on current: ☒ Process ☐ Thread ☐ Step Threads Together Step CUDA threads by: Warp (default)

Threads 1 2

Project Files

Search (Ctrl+K)

Application Code

Sources

mmult2.c

main(int argc, char *arg

minit(int size, double *A

mwrite(int size, double *

my_abort(int err) : void

mmult2.cu

External Code

mmult2.c

63

64 int r

65 {

66

67 int

68 do

69 ch

70 MP

71 MP

72 MP

73 MP

74

75 if(a

76 {

77 if

78 {

79

80

81

82 }

83

Process 0:

Kernel 1 Thread <<<(0,25,0),(0,0,0)>>> stopped in mmult_kernel (mmult2.cu:32) with signal CUDA_EXCEPTION_1 (Lane Illegal Address).

Your program will probably be terminated if you continue.

You can use the stack controls to see what the process was doing at the time.

☒ Always show this window for signals

Continue Pause

Locals Current ... Curren... GPU ...

Current Line(s)

Variable Name Value

Type: none selected

Input/Output* Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output Logbook Evaluate

Stacks

Threads Function

1 main (mmult2.c:72)

1 orte_progress_thread_engine

Ready

This is the exact line the program crashed on – now look at GPU variables to see why

The screenshot shows a CUDA debugger interface with the following components:

- Threads:** 1, 2, 3, K4 (selected)
- CUDA Threads (mmult_kernel):** Block 0, Thread 0, Grid size: 1x4x1, Block size: 64x1x1
- Project Files:** mmult2.c, mmult2.cu
- Code Editor:** mmult2.cu, line 29: `res += A[i*pitch_A_nbelem+k] * B[k*pitch_B_nbelem+j];` (highlighted in red)
- Locals:**

Variable Name	Value
A	0x900200000
B	0x900200800
C	0x900201000
i	3
j	0
k	0
nslices	1
pitch_A_nbelem	512
pitch_B_nbelem	512
pitch_C_nbelem	512
res	<optimized out>
size	4
- Stacks:**

Threads	CUDA Thre	Function
1	0	main (mmult2.c:148)
1	128	mmult_kernel (mmult2.cu:19)
1	16	mmult_kernel (mmult2.cu:29)
1	112	mmult_kernel (mmult2.cu:34)
1	0	orte_progress_thread_engine
1	0	cudbgGetAPIVersion
- Evaluate:**

Expression	Value
<code>i*pitch_A_nbelem+k</code>	1536
<code>k*pitch_B_nbelem+j</code>	0

Real-world debugging requires a systematic approach

In the beginning

- Offloading a simple kernel

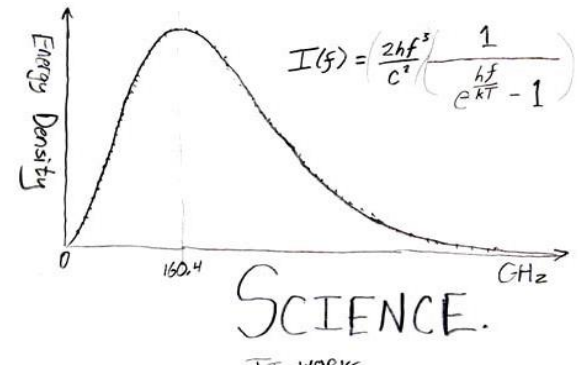
Real-world complexity

- Understanding and analysing real application performance

Science: it works

- Profiling and debugging in extreme conditions

Real-world debugging requires a systematic approach



Discipline Magic Inspiration Science

Debugging by Discipline



Simple techniques,
rigorously applied,
will dramatically
improve your life.

(At least when it's
time to debug)

Discipline #3: Continuous Integration and Regression Testing



Simple

- Sanity and performance checks
- Reliability is crucial – no false positives allowed

Regular

- Run on every code commit
- Speed is important – don't run entire cases

Auto

- Use source control hooks to submit test jobs
- OSS to view and manage runs (<http://jenkins-ci.org>)

Discipline #3: Continuous Integration and Regression Testing

DDT

- Prefix sanity tests with `ddt --offline $REV.html ...`
- Integrate debug reports into Jenkins/CI system

MAP

- Prefix performance tests with: `map --profile ...`
- MAP's editor highlights source lines changed

PR

- Generate HTML reports directly or from MAP files
- Integrate into Jenkins/CI & graph metrics over time

Debugging by Magic

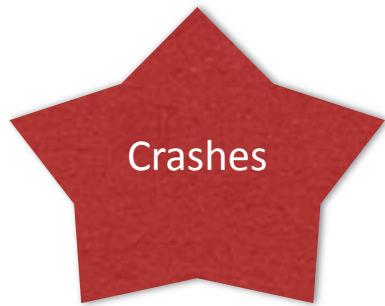


Any technology sufficiently advanced is indistinguishable from magic.

Unpredictable, dangerous, irresistible.

Debugging by Magic

Some problems are perfect for investigating with a debugging tool:



Learn to use the bisect command with a test script to isolate the revision that failed:

```
$ hg bisect --bad
$ hg bisect --good 4
$ hg bisect -c logs/my-test.sh
$ hg log -pr <changeset id>
```

Bonus - static analysis
(integrated into DDT)

Debugging by Inspiration



Look at the problem,
see the solution.

Trust your instincts.

Test whether they're
right.

Debugging by Inspiration

When you have a sense for what the problem is:

Test it: `$ ddt -offline log.html -trace-at mmult.c:412,rx,ry,rz`

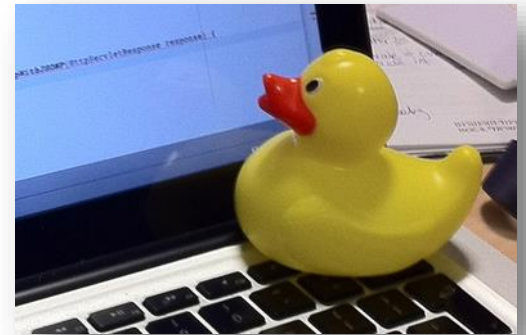
Log it: `$ cat >> logs/short-problem-name`

Suspect rx is out of bounds in mmult.c:412.

Testing with `-trace-at mmult.c:412,rx,ry,rz` showed...

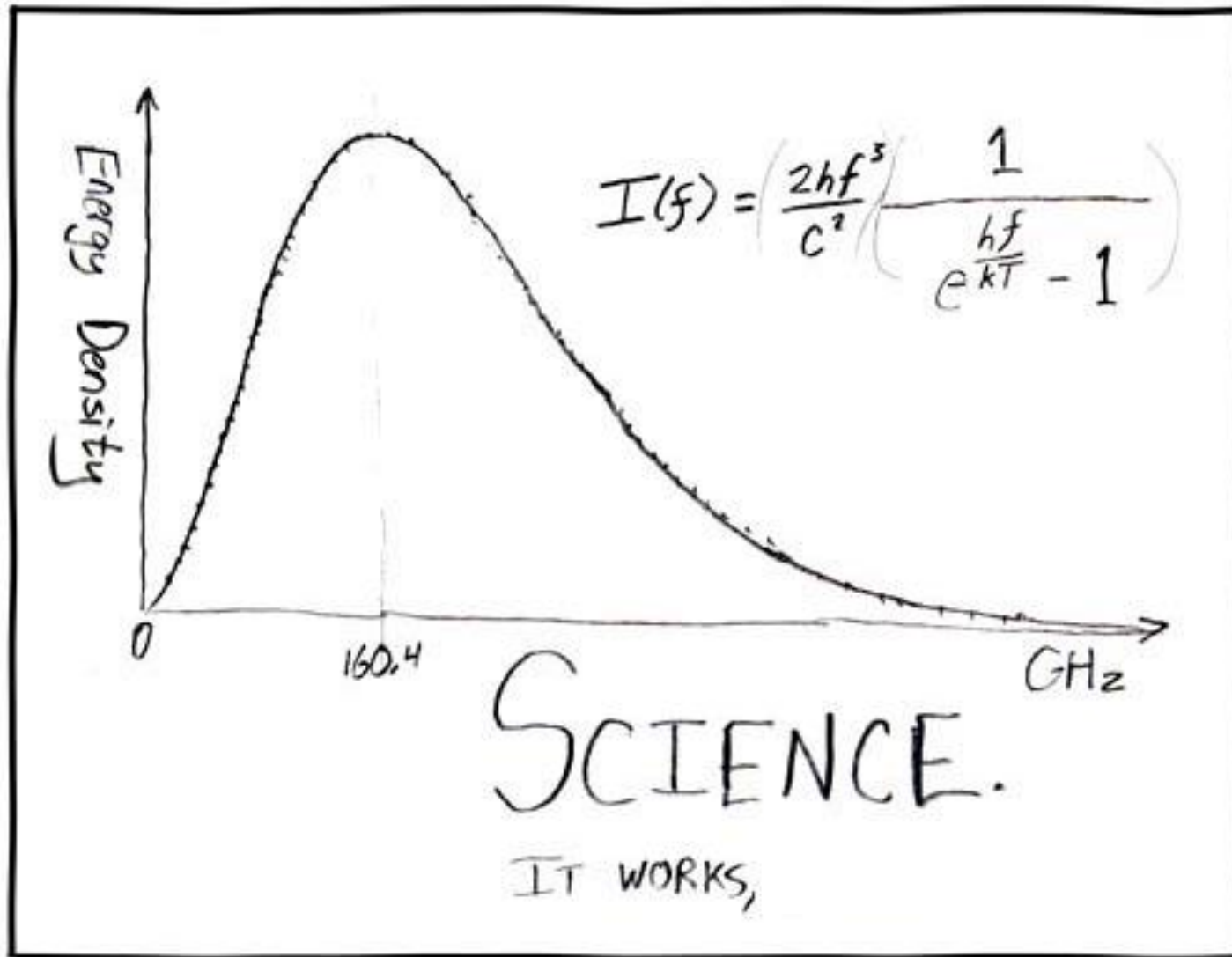
Search your logbooks: `$ grep -ri "out of bounds" logs/*`

If in doubt: explain it to a rubber duck.



Tip - set a **time limit** for debugging by inspiration.
After 15 minutes, try **science**.

Debugging by Science



1. Hypothesis
2. Prediction
3. Experiment
4. Observation
5. Conclusion

There is a **reason** for the bug and you **will** find it!

Debugging by Science

A logbook is at the heart of debugging by science:

hypothesis: cause is in shell_sort()

prediction: At sort.c:6, expect a[] = [11, 4] and size = 2

experiment: -trace-at sort.c:6,a[0],a[1],size

observation: a[] = [11, 14, ?] and size = 3

conclusion: rejected

hypothesis: calling shell_sort with size=3 causes failure

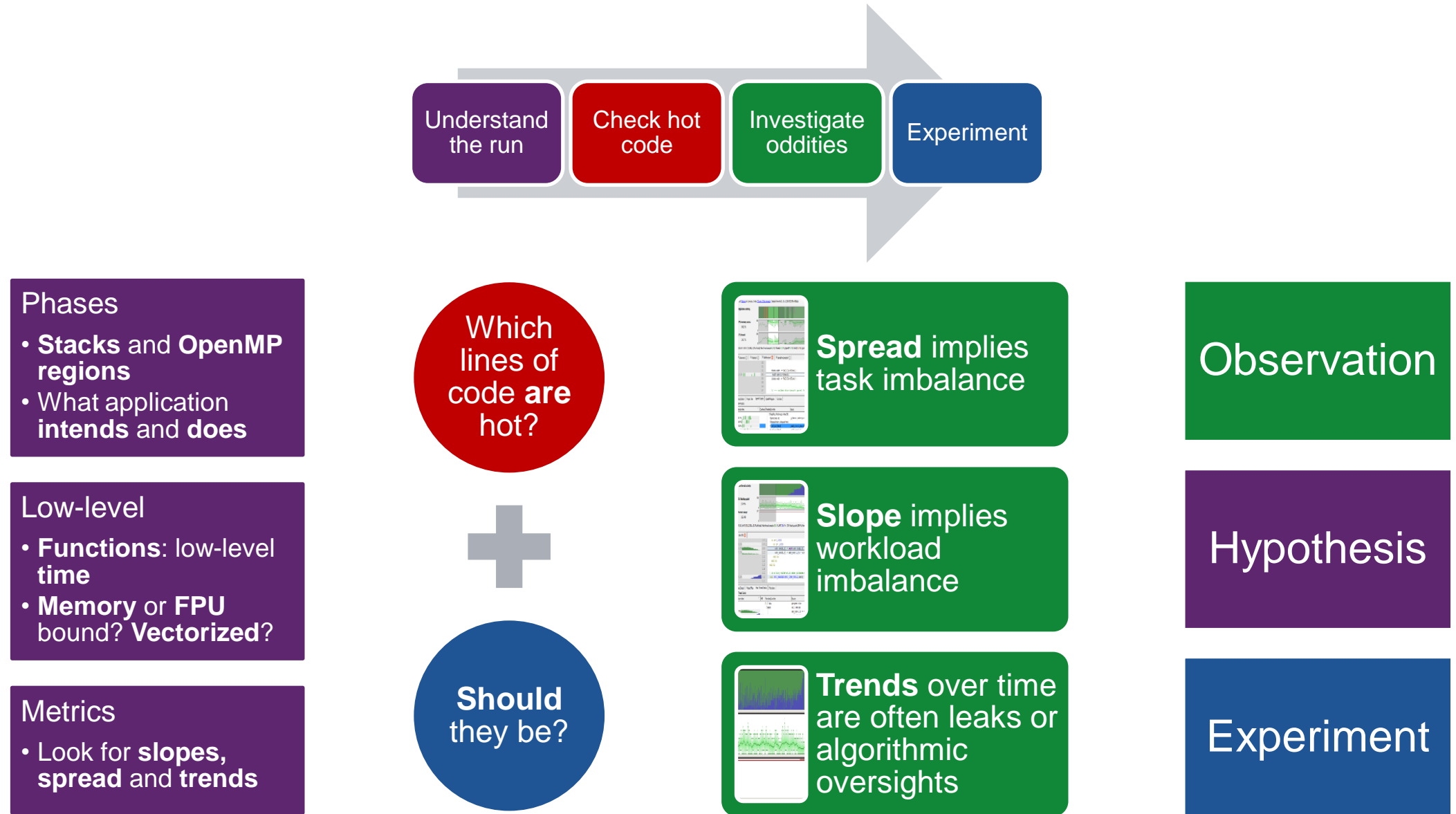
prediction: setting size=2 should make program work

experiment: Set size=2 before call using debugger

observation: As predicted

conclusion: confirmed

Real-world performance optimization is also a process:



Best Practices for Profiling and Debugging Complex Code

In the beginning

- Offloading a simple kernel



Real-world complexity

- Understanding and analysing real application performance



Science: it works

- Profiling and debugging in extreme conditions



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

Accelerating Real Applications

Best Practices for Profiling and Debugging Complex Code

Beau Paisley

Senior Solutions Architect US

