



arm

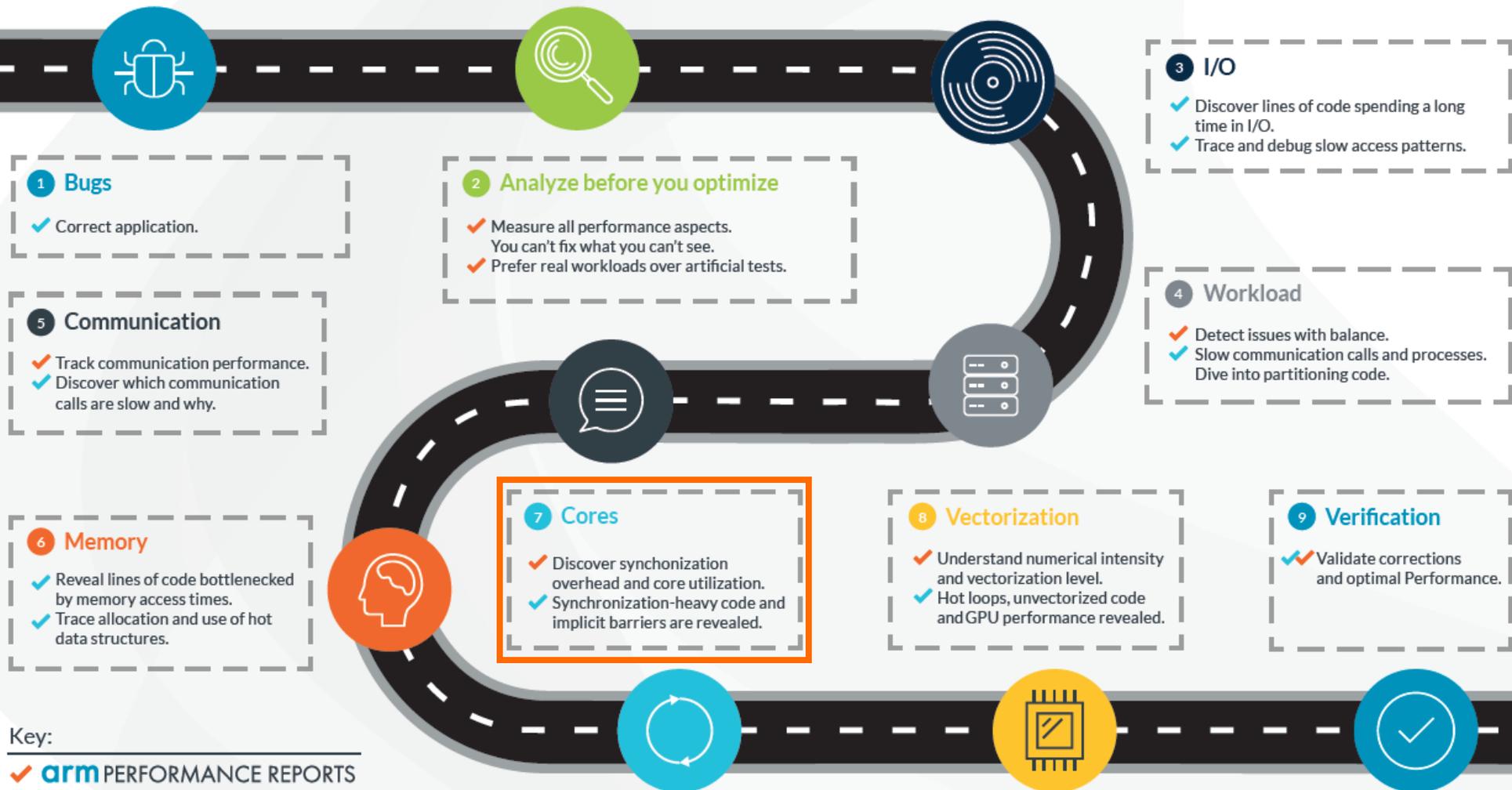
Meet the experts
**Optimize HPC - Application
Efficiency on
Many-Core Systems**

- Florent Lebeau
- 27 March 2018

9 Step guide: optimizing high performance applications



Improving the efficiency of your parallel software holds the key to solving more complex research problems faster. This pragmatic, 9 Step best practice guide will help you identify and focus on application readiness, bottlenecks and optimizations one step at a time.



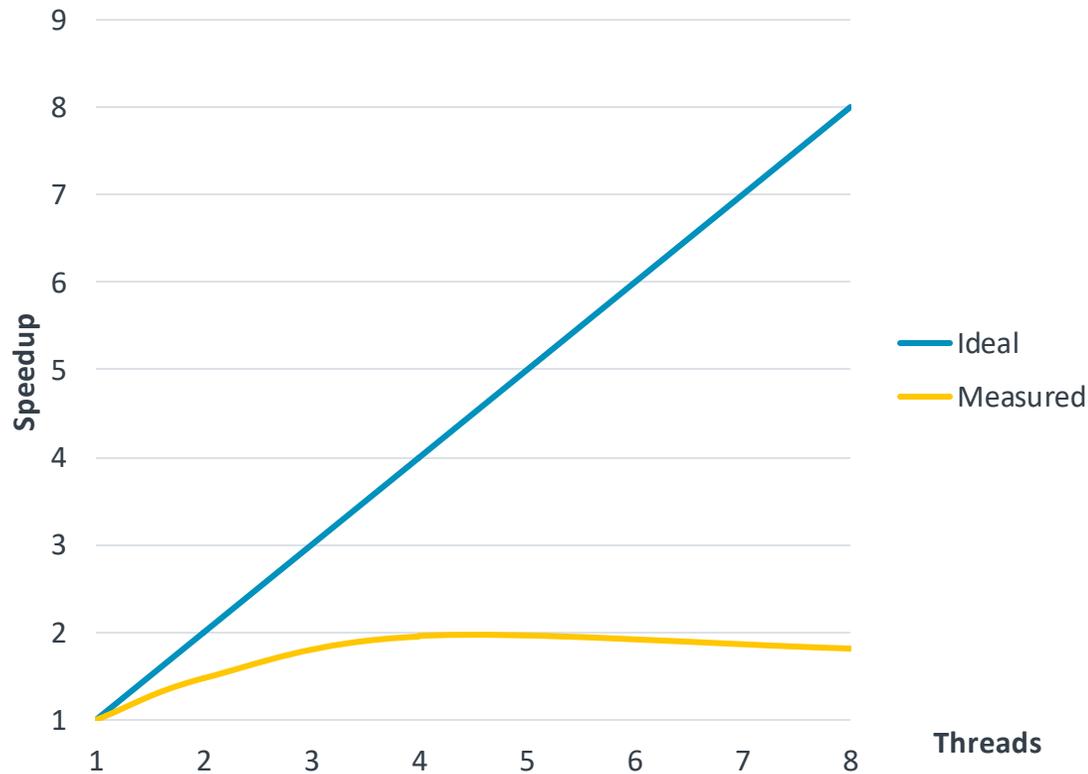
Key:

- ✓ **arm** PERFORMANCE REPORTS
- ✓ **arm** FORGE



Multithreading and scalability

“I wrote my program to run in parallel with a few OpenMP directives... But the performance is not really what I expected.”



Example with modified version of Cloverleaf

- Multi-threaded version with OpenMP
- No MPI, no IO

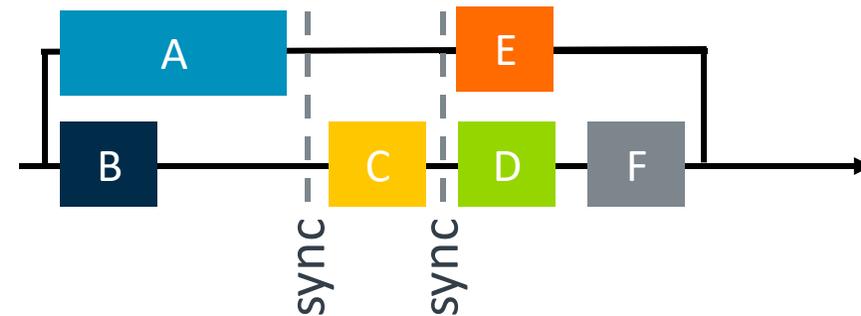
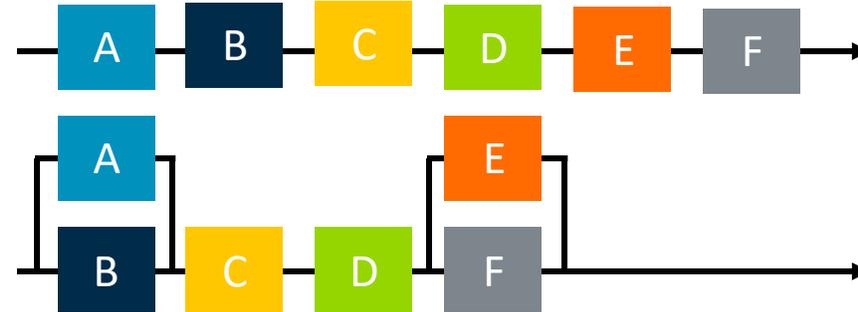
Number of threads	Runtime (seconds)
1	281
2	192
4	144
8	155

Outline

- What are the challenges of multithreaded applications?
- How to identify issues and act quickly?
- How to understand the performance and optimise the code?

Challenges

- Sequential sections
 - Regions outside of parallel sections
 - Master or single OpenMP sections
- Synchronisation overhead
 - Load imbalance
 - Implicit and explicit barriers
 - Scheduling policy
 - Communications
 - Hardware contention



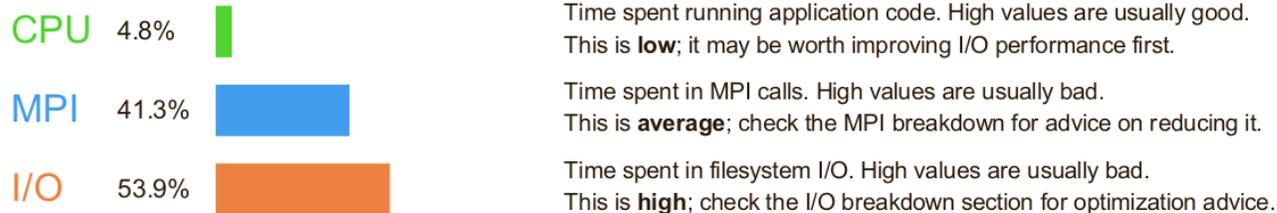
Identifying the amount of sequential code

- Arm Performance Reports is an application reporting tool for HPC
 - Easy to use: no re-compiling required
 - Gives a comprehensible and readable summary of the application behavior

arm PERFORMANCE Executable: MADbench2 Resources: 16 processes, 1 node Machine: sandybridge2 Start time: Mon Nov 4 12:27:50 2013

Summary: MADbench2 is I/O-bound in this configuration

The total wallclock time was spent as follows:



This application run was **I/O-bound**. A breakdown of this time and advice for investigating further is in the **I/O** section below.

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance. No time was spent in **vectorized instructions**. Check the compiler's vectorization advice to see why key loops could not be vectorized.

I/O
A breakdown of how the **53.9%** total I/O time was spent:

Time in reads	3.7%	
Time in writes	96.3%	
Estimated read rate	272 Mb/s	
Estimated write rate	7.06 Mb/s	

Most of the time is spent in **write operations**, which have a very low 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.

All of the time is spent in **collective calls** with a very low transfer rate. This suggests a significant load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

Memory
Per-process memory usage may also affect scaling:

Mean process memory usage	160 Mb	
Peak process memory usage	173 Mb	
Peak node memory usage	17.2%	

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.

CPU

A breakdown of how the **4.8%** total CPU time was spent:



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

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

Sequential sections and scalability

- Running Performance Reports on the example using 1 thread indicates that:
 - Time spent in serial code is 11%
 - Theoretical speedup of 4.5 with 8 threads
- With 8 threads:
 - Speedup is only 1.8

CPU

A breakdown of the 100.0% CPU time:

Single-core code	11.4%	█
OpenMP regions	88.6%	█
Scalar numeric ops	6.8%	
Vector numeric ops	44.9%	█
Memory accesses	48.3%	█

The CPU performance appears well-optimized for numerical computation. The biggest gains may now come from running at larger scales.

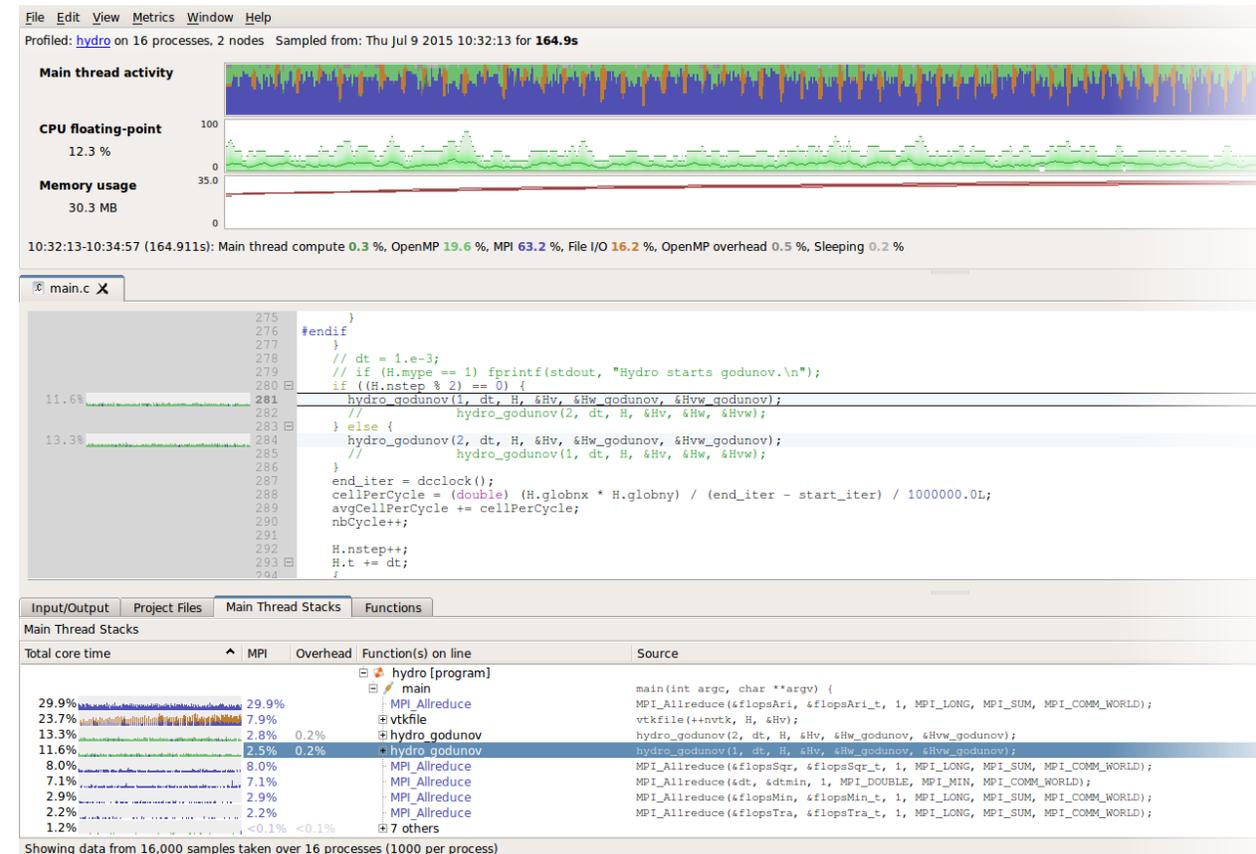
CPU

A breakdown of the 99.8% CPU time:

Single-core code	25.5%	█
OpenMP regions	74.5%	█
Scalar numeric ops	2.8%	
Vector numeric ops	14.6%	█
Memory accesses	68.2%	█

Where does code run serially?

- Arm MAP is a lightweight multi-node profiling tool
 - Compiling with debugging flag required
 - Shows processes and threads activity over time
 - See source code is annotated
 - Information aggregated by stacks and function
- **Compute**, **IO** and **MPI**



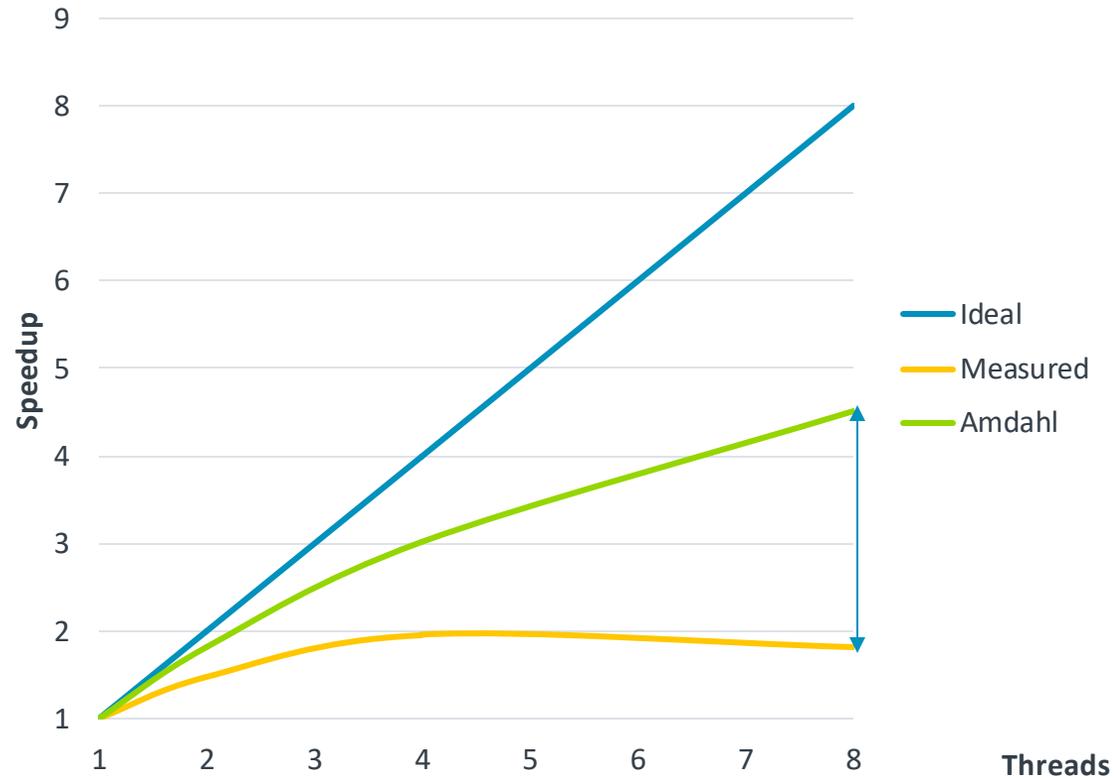
Critical sections

- Profiling our example using 8 threads with Arm MAP shows
 - OpenMP activity in **Light Green**
 - Single thread activity in **Dark Green**
 - Synchronisation in **Grey**
- The tool shows serial execution happens in the `pdv_module::pdv` subroutine
 - In a OpenMP **single** section
 - Can be replaced with OpenMP **parallel do** section

```
62 REAL(KIND=8) :: recip_volume,energy_change,min_cell_volume
63 REAL(KIND=8) :: right_flux,left_flux,top_flux,bottom_flux,total_flux
64 REAL(KIND=8) :: volume_change_s
65
66
67 !$OMP SINGLE
68
69 IF(predict)THEN
70
71 DO k=y_min,y_max
72 DO j=x_min,x_max
73
74 left_flux= (xarea(j ,k )*(xvel0(j ,k )+xvel0(j ,k+1)
75 +xvel0(j ,k )+xvel0(j ,k+1)))*0.25_8*dt*0.5
76 right_flux= (xarea(j+1,k )*(xvel0(j+1,k )+xvel0(j+1,k+1)
77 +xvel0(j+1,k )+xvel0(j+1,k+1)))*0.25_8*dt*0.5
78 bottom_flux=(yarea(j ,k )*(yvel0(j ,k )+yvel0(j+1,k )
79 +yvel0(j ,k )+yvel0(j+1,k )))*0.25_8*dt*0.5
80 top_flux= (yarea(j ,k+1)*(yvel0(j ,k+1)+yvel0(j+1,k+1)
81 +yvel0(j ,k+1)+yvel0(j+1,k+1)))*0.25_8*dt*0.5
82 total_flux=right_flux-left_flux+top_flux-bottom_flux
83
84 volume_change_s=volume(j,k)/(volume(j,k)+total_flux)
85
86 min_cell_volume=MIN(volume(j,k)+right_flux-left_flux+top_flux-bottom_flux & .....
87
88 recip_volume=1.0/volume(j,k)
89
90 energy_change=(pressure(j,k)/density0(j,k)+viscosity(j,k)/density0(j,k))*total_flux*recip_volume
91
92 energy1(j,k)=energy0(j,k)-energy_change
93
94 density1(j,k)=density0(j,k)*volume_change_s
95
96
97 ENDDO
98 ENDDO
99
100 ELSE
101 DO k=y_min,y_max
102 DO j=x_min,x_max
103
104 left_flux= (xarea(j ,k )*(xvel0(j ,k )+xvel0(j ,k+1)
105 +xvel1(j ,k )+xvel1(j ,k+1))*0.25_8*dt
106 right_flux= (xarea(j+1,k )*(xvel0(j+1,k )+xvel0(j+1,k+1)
107 +xvel1(j+1,k )+xvel1(j+1,k+1))*0.25_8*dt
108 bottom_flux=(yarea(j ,k )*(yvel0(j ,k )+yvel0(j+1,k )
109 +yvel1(j ,k )+yvel1(j+1,k )))*0.25_8*dt
110 top_flux= (yarea(j ,k+1)*(yvel0(j ,k+1)+yvel0(j+1,k+1)
111 +yvel1(j ,k+1)+yvel1(j+1,k+1))*0.25_8*dt
112 total_flux=right_flux-left_flux+top_flux-bottom_flux
113
114 volume_change_s=volume(j,k)/(volume(j,k)+total_flux)
115
116 min_cell_volume=MIN(volume(j,k)+right_flux-left_flux+top_flux-bottom_flux & .....
117
118 recip_volume=1.0/volume(j,k)
119
120 energy_change=(pressure(j,k)/density0(j,k)+viscosity(j,k)/density0(j,k))*total_flux*recip_volume
121
122 energy1(j,k)=energy0(j,k)-energy_change
123
124 density1(j,k)=density0(j,k)*volume_change_s
125
126
127 ENDDO
128 ENDDO
129
130 ENDDO
131 ENDDO
132
133 ENDIF
134
135 !$OMP END SINGLE
136
137 END SUBROUTINE PdV_kernel
138
```



Synchronization overhead



Theoretical speedup: 4.5
Measured: 1.8

Identifying the amount of synchronization

- Performance Reports on the example using 8 threads shows:
 - Low amount of computation
- System load is only 78%
- Possible reasons:
 - Load imbalance
 - Implicit and explicit barriers
 - Scheduling policy
 - Communications
 - Hardware contention

OpenMP

A breakdown of the 74.5% time in OpenMP regions:

Computation	53.6%	
Synchronization	46.4%	
Physical core utilization	100.0%	
System load	78.0%	

System Load and thread binding

- Dynamic adjustment of the number of threads is enabled
 - `OMP_DYNAMIC='TRUE'`
- Threads are not bound
 - `OMP_PROC_BIND='FALSE'`
- Binding the threads and disabling dynamic adjustment slightly improve performance
- Performance Reports detects a sign of overly fine-grained parallelism

CPU

A breakdown of the 99.9% CPU time:

Single-core code	29.8%	■
OpenMP regions	70.2%	■
Scalar numeric ops	2.5%	
Vector numeric ops	16.9%	■
Memory accesses	59.1%	■

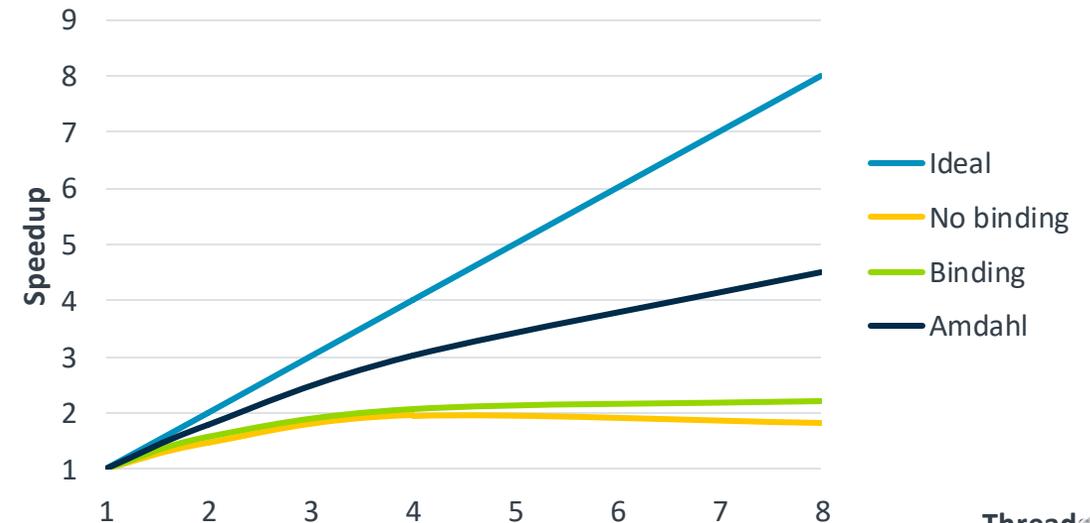
OpenMP

A breakdown of the 70.2% time in OpenMP regions:

Computation	68.0%	■
Synchronization	32.0%	■
Physical core utilization	100.0%	■
System load	96.8%	■

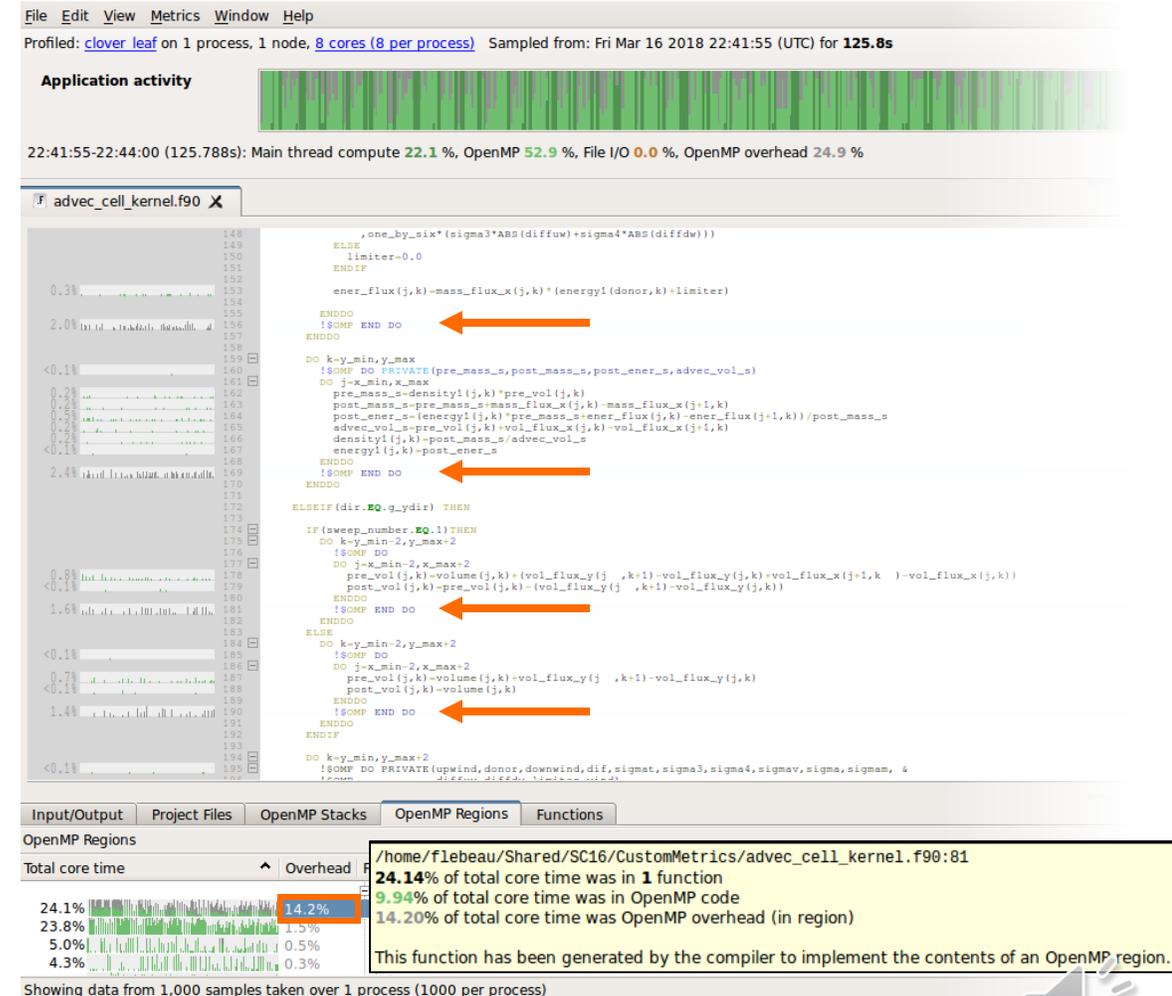
Significant time is spent **synchronizing** threads in parallel regions. Check the affected regions with a profiler.

This may be a sign of overly fine-grained parallelism (OpenMP regions in tight loops) or workload imbalance.



Where are threads waiting?

- Profiling Cloverleaf using 8 threads (bound to cores) with Arm MAP shows
 - Overhead in one OpenMP region
- Implicit barriers
 - Inner loop parallelization only



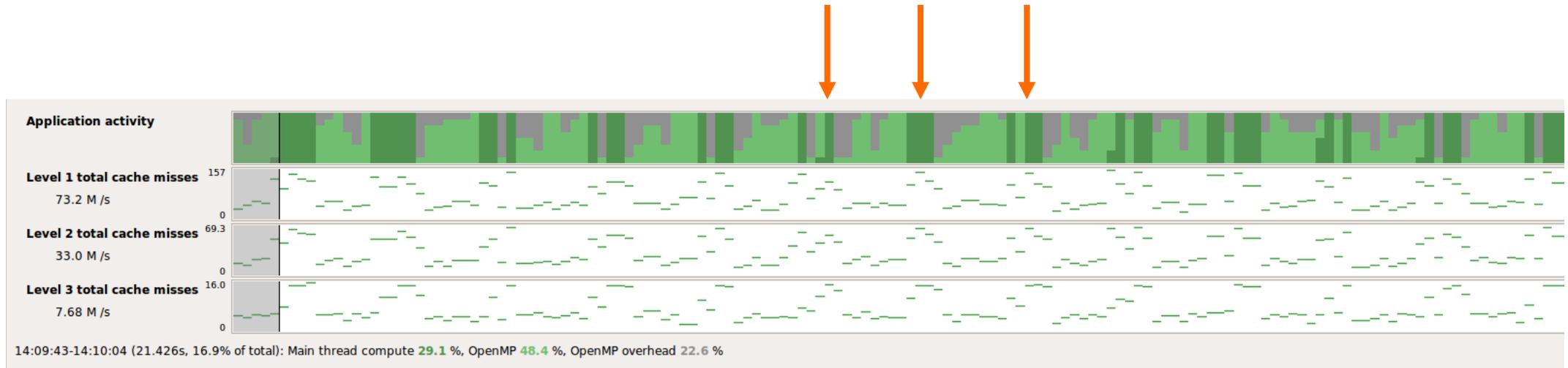
Understanding resource usage

- Memory accesses



Additional metrics

- PAPI



Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

תודה

arm

