

allinea

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

Application Trapped Capacity and Energy Reporting

Mark O'Connor

VP Product Management



allinea
FORGE

allinea
DDT

allinea
MAP



allinea
PERFORMANCE
REPORTS

Application Trapped Capacity and Energy Reporting

The screenshot shows the allinea website with navigation links for Developer Tools, Performance Reports, Trials, Purchase, Support, Resources, News, Events, and Company. It features several promotional banners: 'allinea PERFORMANCE REPORTS', 'DEVELOPERS THE INDUSTRY STANDARD HPC C++ AND F90 DEVELOPMENT SUITE', 'allinea FORGE', 'ANALYSTS APPLICATION ANALYTICS FOR HPC CLUSTERS', 'THE WORLD'S MOST SCALABLE DEBUGGER allinea DDT', and 'allinea MAP INCREASE APPLICATION PERFORMANCE'. Below these are three columns of text describing the company's focus on climate modeling, application performance, and software development tools.

Today: Application-centric performance tuning and energy metrics

The screenshot shows a summary of a performance report. It includes the allinea logo and the following details:
Command: /home/abowen/work/trapped-examples/mmult-orig 300 4096
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 8 processes
Machine: mars
Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
Total time: 463 seconds (about 8 minutes)
Full path: /home/abowen/work/trapped-examples



Energy Usage

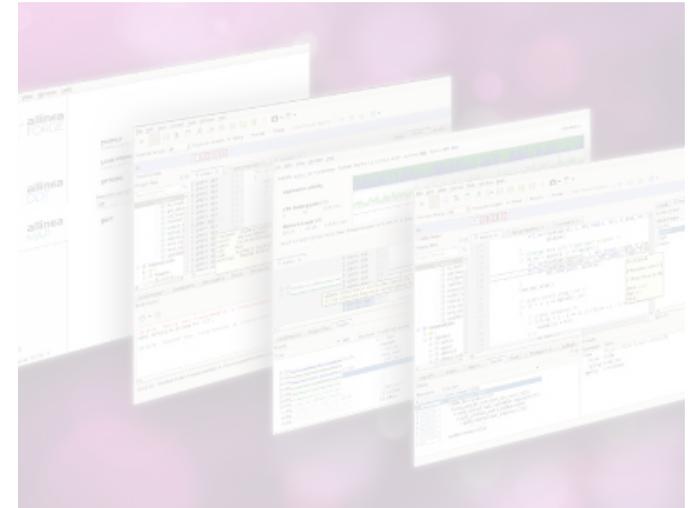


To reduce trapped capacity look at vectorization first, for more information see the CPU section below. In addition, look at memory accesses, also in the CPU section. This job is bound by `compute`, scheduling it alongside jobs bound by `MPI` and `I/O` may reduce overall trapped capacity.

Overall Energy

CPU Energy

New research: Trapped Capacity Reports on a per-application basis



Future research: energy-aware scheduling, superoptimizing compilers, ...

f t in

allinea DEVELOPER TOOLS PERFORMANCE REPORTS TRIALS PURCHASE SUPPORT RESOURCES NEWS EVENTS COMPANY

allinea
The problem we solve...

From climate modeling to astrophysics, from financial modeling to engine design, the power of clusters and supercomputers advances the frontiers of knowledge and delivers results for industry. Writing and deploying software that exploits that computing power is a

allinea PERFORMANCE REPORTS
How well do your applications exploit your hardware? Allinea Performance Reports are the most effective way to characterize and understand the performance of HPC application runs.

allinea FORGE
Allinea Forge is the complete tool suite for software development - with everything needed to debug, profile, optimize, edit and build applications for high performance.

Allinea provides high-performance software tools



f t in

allinea

DEVELOPER TOOLS PERFORMANCE REPORTS TRIALS PURCHASE SUPPORT RESOURCES NEWS EVENTS COMPANY



allinea
PERFORMANCE
REPORTS

DEVELOPERS
THE INDUSTRY
STANDARD HPC C++ AND
F90 DEVELOPMENT SUITE



allinea
FORGE

THE WORLD'S MOST SCALABLE DEBUGGER



allinea
DDT

ANALYSTS
APPLICATION ANALYTICS
FOR HPC CLUSTERS

INCREASE APPLICATION PERFORMANCE



allinea
MAP

allinea
The problem we solve...

From climate modeling to astrophysics, from financial modeling to engine design, the power of clusters and supercomputers advances the frontiers of knowledge and delivers results for industry. Writing and deploying software that exploits that computing power is a

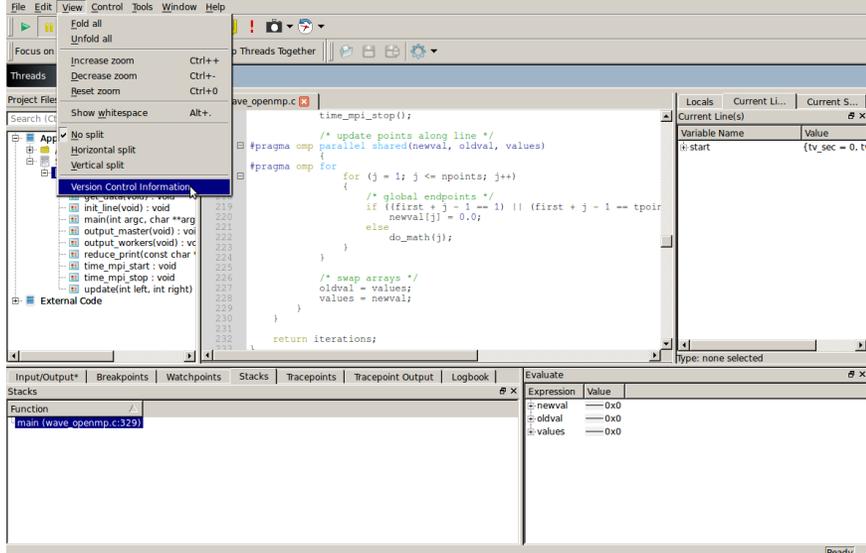


How well do your applications exploit your hardware? Allinea Performance Reports are the most effective way to characterize and understand the performance of HPC application runs.



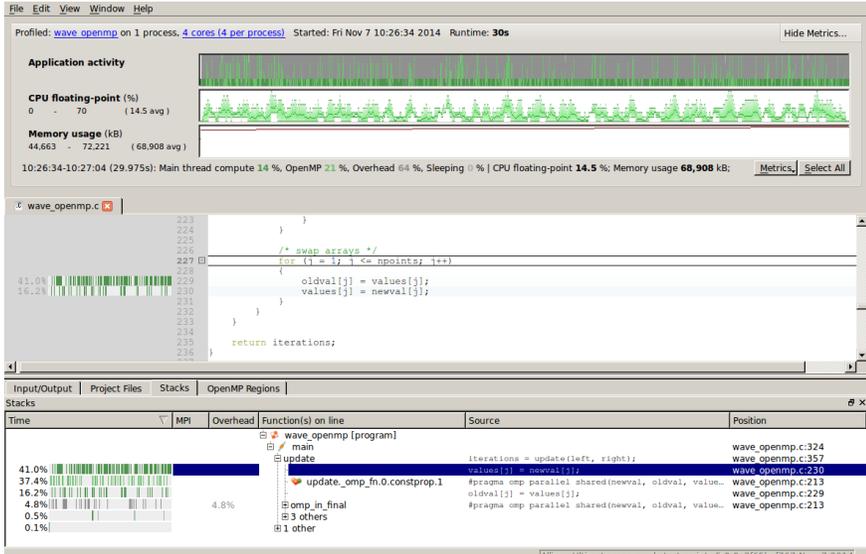
Allinea Forge is the complete tool suite for software development - with everything needed to debug, profile, optimize, edit and build applications for high performance.

allinea
DDT



The screenshot shows the Allinea DDT interface. The main window displays a C++ code snippet with OpenMP pragmas. A 'Locals' window on the right shows the current state of variables: 'i' is 1, 'j' is 1, and 'tv_sec' is 0. The 'Stacks' window at the bottom shows the call stack, with 'main (wave_openmp.c:329)' selected.

Allinea Forge Debug and profile codes



The screenshot shows the Allinea MAP interface. The top section displays 'Application activity' with a bar chart showing CPU floating-point usage (14.5% avg) and memory usage (68,908 kB avg). Below this, a code editor shows a snippet of code with a performance bar overlaid on the 'update' function. The bottom section shows 'OpenMP Regions' with a table of execution times and overheads for different regions.

Time	MPI	Overhead	Function(s) on line	Source	Position
			main	iterations = update(left, right);	wave_openmp.c:324
			update	values[i] = newval[i];	wave_openmp.c:357
41.0%			update_omp_fn_0.constprop.1	#pragma omp parallel shared(newval, oldval, value...	wave_openmp.c:213
37.4%				oldval[i] = values[i];	wave_openmp.c:229
16.2%				#pragma omp parallel shared(newval, oldval, value...	wave_openmp.c:213
4.8%		4.8%	omp_in_final		
0.5%			others		
0.1%			other		

allinea
MAP

f t in

allinea DEVELOPER TOOLS PERFORMANCE REPORTS TRIALS PURCHASE SUPPORT RESOURCES NEWS EVENTS COMPANY

ANALYSTS
APPLICATION ANALYTICS
FOR HPC CLUSTERS

DEVELOPERS
THE INDUSTRY
STANDARD HPC C++ AND
F90 DEVELOPMENT SUITE

THE WORLD'S MOST SCALABLE DEBUGGER

INCREASE APPLICATION PERFORMANCE

allinea
The problem we solve...

From climate modeling to astrophysics, from financial modeling to engine design, the power of clusters and supercomputers advances the frontiers of knowledge and delivers results for industry. Writing and deploying software that exploits that computing power is a

How well do your applications exploit your hardware? Allinea Performance Reports are the most effective way to characterize and understand the performance of HPC application runs.

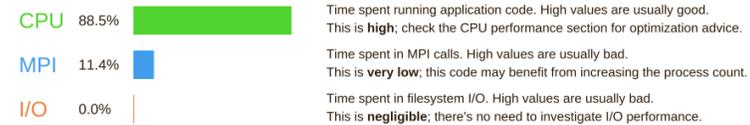
Allinea Forge is the complete tool suite for software development - with everything needed to debug, profile, optimize, edit and build applications for high performance.

Performance Reports

Monitor and tune applications

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:

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 synchronization 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

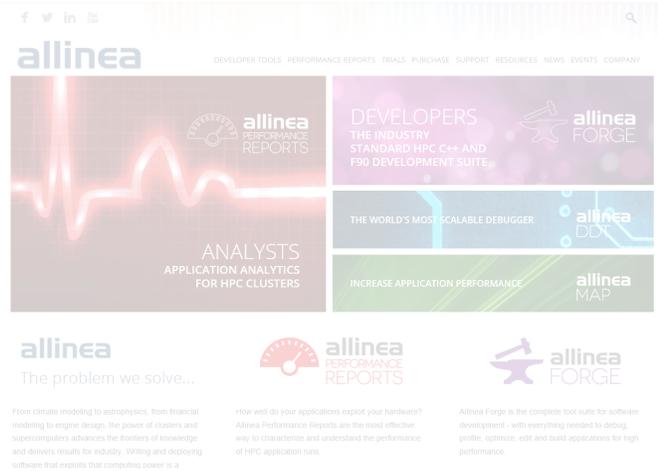
A breakdown of how the total **588 J** energy was spent:

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.

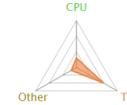
Application Trapped Capacity and Energy Reporting



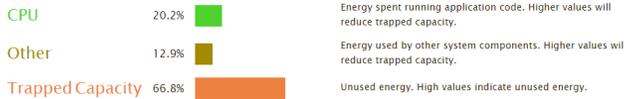
Today: Application-centric performance tuning and energy metrics



Command: /home/abowen/work/trapped-examples/mmult-orig 300 4096
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 8 processes
Machine: mars
Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
Total time: 463 seconds (about 8 minutes)
Full path: /home/abowen/work/trapped-examples



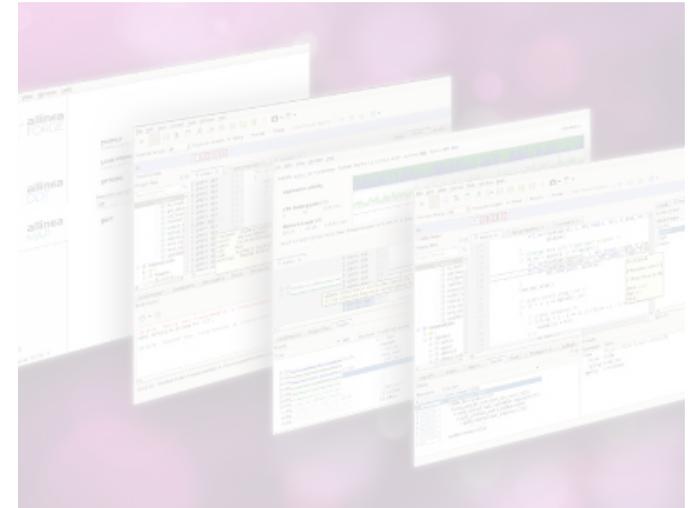
Energy Usage



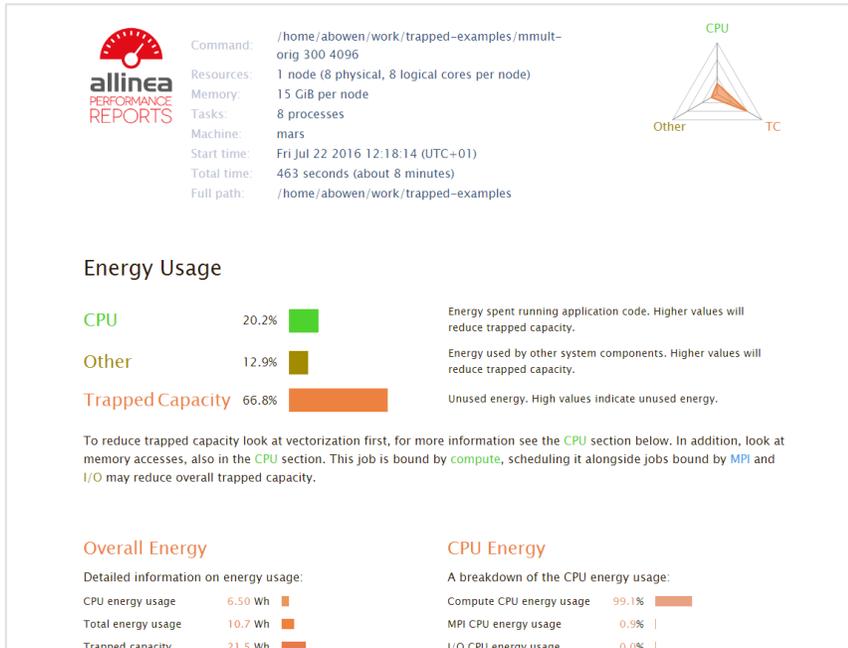
To reduce trapped capacity look at vectorization first, for more information see the CPU section below. In addition, look at memory accesses, also in the CPU section. This job is bound by `compute`, scheduling it alongside jobs bound by `MPI` and `I/O` may reduce overall trapped capacity.

Overall Energy CPU Energy

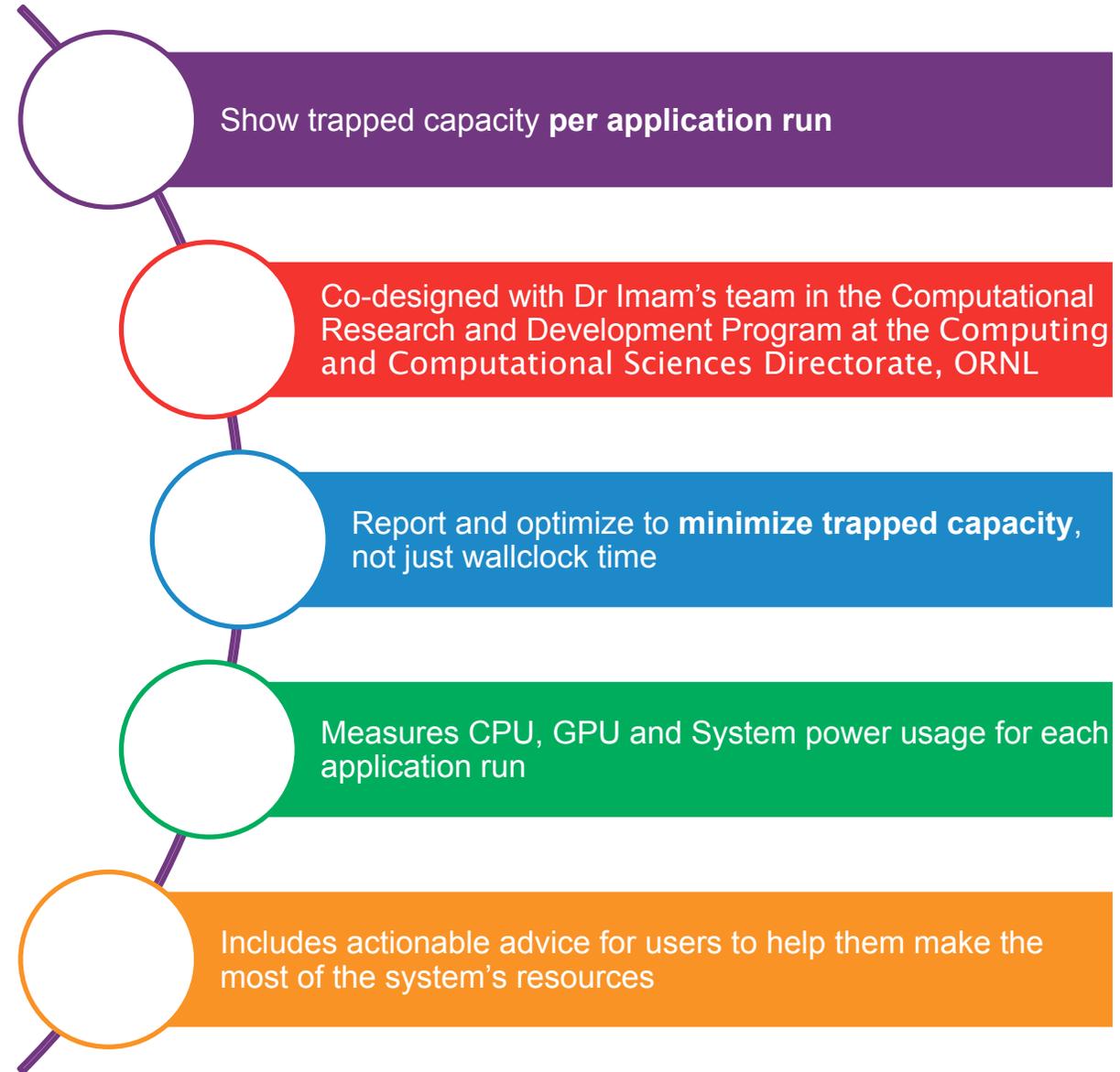
New research: Trapped Capacity Reports on a per-application basis



Future research: energy-aware scheduling, superoptimizing compilers, ...

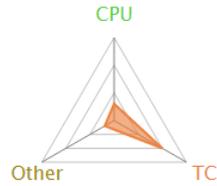


Application-centric Trapped Capacity Reports

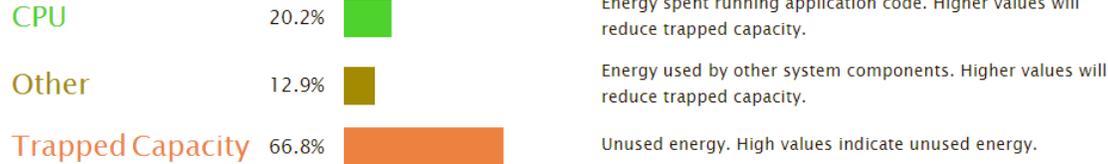




Command: /home/abowen/work/trapped-examples/mmult-orig 300 4096
 Resources: 1 node (8 physical, 8 logical cores per node)
 Memory: 15 GiB per node
 Tasks: 8 processes
 Machine: mars
 Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
 Total time: 463 seconds (about 8 minutes)
 Full path: /home/abowen/work/trapped-examples



Energy Usage



To reduce trapped capacity look at vectorization first, for more information see the [CPU](#) section below. In addition, look at memory accesses, also in the [CPU](#) section. This job is bound by [compute](#), scheduling it alongside jobs bound by [MPI](#) and [I/O](#) may reduce overall trapped capacity.

Overall Energy

Detailed information on energy usage:

CPU energy usage	6.50 Wh	<div style="width: 10%; height: 10px; background-color: #FF8C00;"></div>
Total energy usage	10.7 Wh	<div style="width: 15%; height: 10px; background-color: #FF8C00;"></div>
Trapped capacity	21.5 Wh	<div style="width: 30%; height: 10px; background-color: #FF8C00;"></div>
Total energy budget	32.1 Wh	<div style="width: 50%; height: 10px; background-color: #FF8C00;"></div>
Mean node power budget	250 W	<div style="width: 40%; height: 10px; background-color: #FF8C00;"></div>
Mean node power	82.8 W	<div style="width: 15%; height: 10px; background-color: #FF8C00;"></div>
Peak node power	98.0 W	<div style="width: 20%; height: 10px; background-color: #FF8C00;"></div>

The **peak power** draw is significantly higher than the mean, use a profiler to identify times when the power draw is low.

CPU Energy

A breakdown of the CPU energy usage:

Compute CPU energy usage	99.1%	<div style="width: 99.1%; height: 10px; background-color: #FF8C00;"></div>
MPI CPU energy usage	0.9%	<div style="width: 0.9%; height: 10px; background-color: #FF8C00;"></div>
I/O CPU energy usage	0.0%	<div style="width: 0.0%; height: 10px; background-color: #FF8C00;"></div>
Mean power during compute	50.5 W	<div style="width: 60%; height: 10px; background-color: #FF8C00;"></div>
Mean power during MPI	54.0 W	<div style="width: 65%; height: 10px; background-color: #FF8C00;"></div>
Mean power during I/O	0.00 W	<div style="width: 0.0%; height: 10px; background-color: #FF8C00;"></div>

Note: Power and energy values during MPI, I/O and compute are estimated from available information and may not be accurate.

CPU

A breakdown of the 99.2% CPU time:

Scalar numeric ops	50.8%	<div style="width: 50.8%; height: 10px; background-color: #008000;"></div>
Vector numeric ops	0.0%	<div style="width: 0.0%; height: 10px; background-color: #008000;"></div>
Memory accesses	49.2%	<div style="width: 49.2%; height: 10px; background-color: #008000;"></div>

No time is spent in **vectorized instructions**. Vectorized instructions generally require more power than scalar computation. To reduce trapped capacity check the compiler's vectorization advice to see if key loops can be vectorized.

Significant time is spent on **memory accesses**. Memory operations generally require less power than computation. To reduce trapped capacity use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 0.8% MPI time:

Time in collective calls	2.9%	<div style="width: 2.9%; height: 10px; background-color: #000080;"></div>
Time in point-to-point calls	97.1%	<div style="width: 97.1%; height: 10px; background-color: #000080;"></div>
Effective process collective rate	0.00 bytes/s	<div style="width: 0.0%; height: 10px; background-color: #000080;"></div>
Effective process point-to-point rate	33.4 MB/s	<div style="width: 40%; height: 10px; background-color: #000080;"></div>

Optimising [MPI](#) communication will have little impact on the trapped capacity.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	210 MiB	<div style="width: 10%; height: 10px; background-color: #FF0000;"></div>
Peak process memory usage	406 MiB	<div style="width: 20%; height: 10px; background-color: #FF0000;"></div>
Peak node memory usage	14.0%	<div style="width: 14.0%; height: 10px; background-color: #FF0000;"></div>

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	<div style="width: 0.0%; height: 10px; background-color: #008000;"></div>
Synchronization	0.0%	<div style="width: 0.0%; height: 10px; background-color: #008000;"></div>
Physical core utilization	100.0%	<div style="width: 100.0%; height: 10px; background-color: #008000;"></div>
System load	100.6%	<div style="width: 100.6%; height: 10px; background-color: #008000;"></div>

I/O

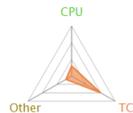
A breakdown of the 0.0% I/O time:

Time in reads	0.0%	<div style="width: 0.0%; height: 10px; background-color: #000080;"></div>
Time in writes	0.0%	<div style="width: 0.0%; height: 10px; background-color: #000080;"></div>
Effective process read rate	0.00 bytes/s	<div style="width: 0.0%; height: 10px; background-color: #000080;"></div>
Effective process write rate	0.00 bytes/s	<div style="width: 0.0%; height: 10px; background-color: #000080;"></div>

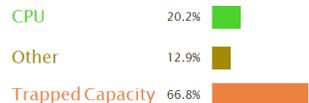
No time is spent in [I/O](#), there's nothing to optimise here!



Command: /home/abowen/work/trapped-examples/mmult-orig 300 4096
 Resources: 1 node (8 physical, 8 logical cores per node)
 Memory: 15 GiB per node
 Tasks: 8 processes
 Machine: mars
 Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
 Total time: 463 seconds (about 8 minutes)
 Full path: /home/abowen/work/trapped-examples



Energy Usage

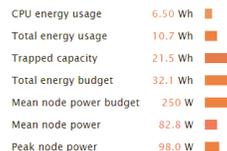


Energy spent running application code. Higher values will reduce trapped capacity.
 Energy used by other system components. Higher values will reduce trapped capacity.
 Unused energy. High values indicate unused energy.

To reduce trapped capacity look at vectorization first, for more information see the CPU section below. In addition, look at memory accesses, also in the CPU section. This job is bound by compute, scheduling it alongside jobs bound by MPI and I/O may reduce overall trapped capacity.

Overall Energy

Detailed information on energy usage:



The peak power draw is significantly higher than the mean, use a profiler to identify times when the power draw is low.

CPU Energy

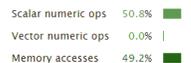
A breakdown of the CPU energy usage:



Note: Power and energy values during MPI, I/O and compute are estimated from available information and may not be accurate.

CPU

A breakdown of the 99.2% CPU time:



No time is spent in vectorized instructions. Vectorized instructions generally require more power than scalar computation. To reduce trapped capacity check the compiler's vectorization advice to see if key loops can be vectorized.

Significant time is spent on memory accesses. Memory operations generally require less power than computation. To reduce trapped capacity use a profiler to identify time-consuming loops and check their cache performance.

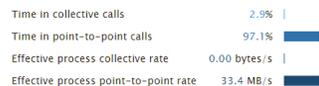
Threads

A breakdown of how multiple threads were used:



MPI

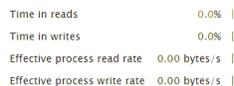
A breakdown of the 0.8% MPI time:



Optimising MPI communication will have little impact on the trapped capacity.

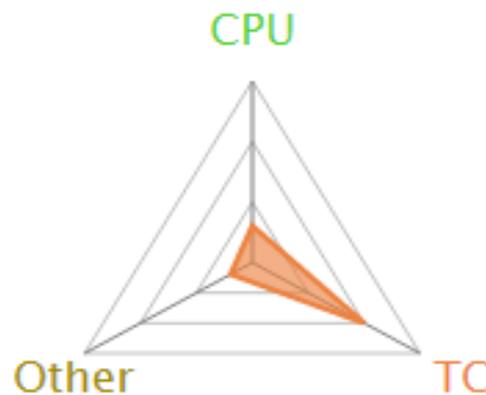
I/O

A breakdown of the 0.0% I/O time:



No time is spent in I/O, there's nothing to optimise here!

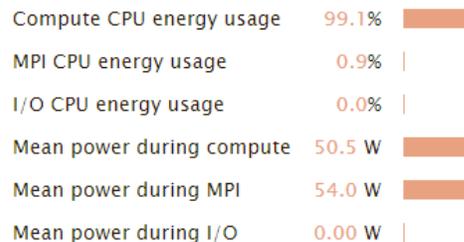
Trapped Capacity at a glance



Track CPU and GPU energy use

CPU Energy

A breakdown of the CPU energy usage:



Note: Power and energy values during MPI, I/O and compute are estimated from available information and may not be accurate.

Actionable advice

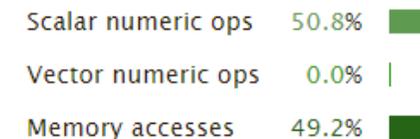


To reduce trapped capacity look at vectorized memory accesses, also in the CPU section. I/O may reduce overall trapped capacity.

Identify inefficient applications

CPU

A breakdown of the 99.2% CPU time:

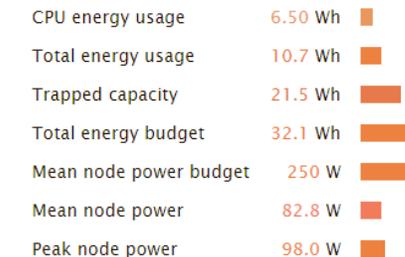


No time is spent in vectorized instructions: instructions generally require more power than computation. To reduce trapped capacity vectorization advice to see if key loops can be vectorized. Significant time is spent on memory accesses.

Detailed breakdown

Overall Energy

Detailed information on energy usage:

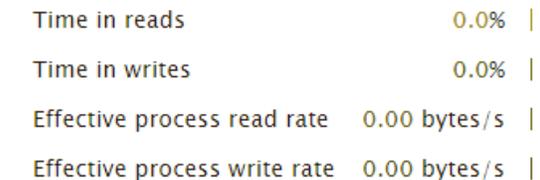


The peak power draw is significantly higher than the mean, profiler to identify times when the power draw is low.

I/O and MPI breakdowns

I/O

A breakdown of the 0.0% I/O time:

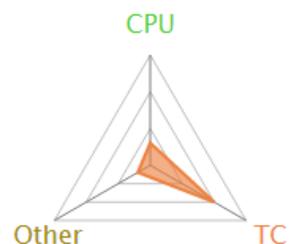


No time is spent in I/O, there's nothing to optimise here!

Example 1: application binary not optimized for CPU architecture



Command: `/home/abowen/work/trapped-examples/mmult-orig 300 4096`
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 8 processes
Machine: mars
Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
Total time: 463 seconds (about 8 minutes)
Full path: `/home/abowen/work/trapped-examples`

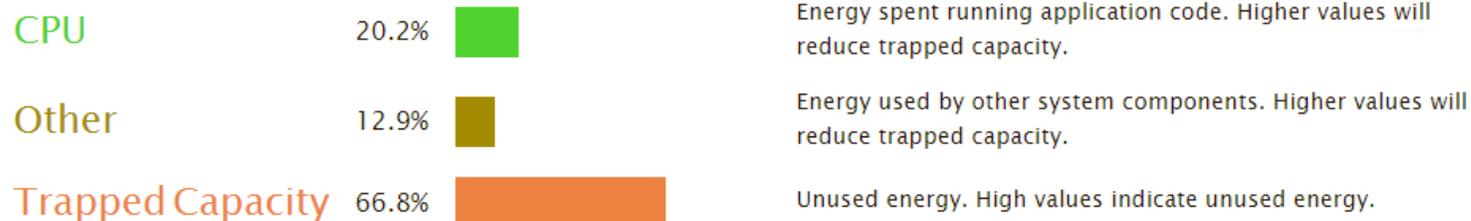


Overall Energy

Detailed information on energy usage:

CPU energy usage	6.50 Wh	■
Total energy usage	10.7 Wh	■
Trapped capacity	21.5 Wh	■
Total energy budget	32.1 Wh	■
Mean node power budget	250 W	■
Mean node power	82.8 W	■
Peak node power	98.0 W	■

Energy Usage



To reduce trapped capacity look at vectorization first, for more information see the [CPU](#) section below. In addition, look at memory accesses, also in the [CPU](#) section. This job is bound by [compute](#), scheduling it alongside jobs bound by [MPI](#) and [I/O](#) may reduce overall trapped capacity.

CPU

A breakdown of the 99.2% CPU time:

Scalar numeric ops	50.8%	■
Vector numeric ops	0.0%	
Memory accesses	49.2%	■

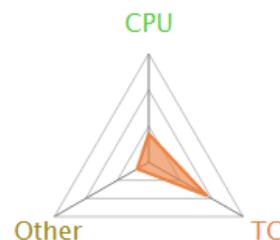
No time is spent in [vectorized instructions](#). Vectorized instructions generally require more power than scalar computation. To reduce trapped capacity check the compiler's vectorization advice to see if key loops can be vectorized.

Significant time is spent on memory accesses. Memory operations generally require less power than computation. To reduce trapped capacity use a profiler to identify time-consuming loops and check their cache performance.

Example 1: recompiled with architecture-specific optimizations



Command: /home/abowen/work/trapped-examples/mmult-vec
300 256
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 8 processes
Machine: mars
Start time: Fri Jul 22 2016 15:19:11 (UTC+01)
Total time: 301 seconds (about 5 minutes)
Full path: /home/abowen/work/trapped-examples

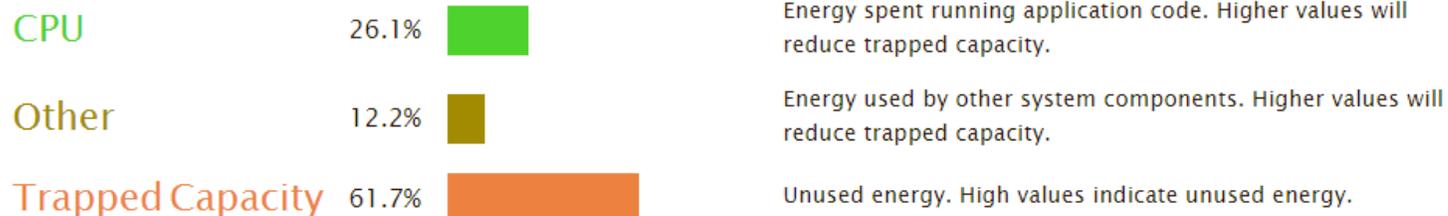


Overall Energy

Detailed information on energy usage:

CPU energy usage	5.43 Wh	■
Total energy usage	7.97 Wh	■
Trapped capacity	12.9 Wh	■
Total energy budget	20.8 Wh	■
Mean node power budget	250 W	■
Mean node power	95.6 W	■
Peak node power	98.0 W	■

Energy Usage



To reduce trapped capacity look at memory accesses first, for more information see the [CPU](#) section below. In addition, look at vectorization, also in the [CPU](#) section. This job is bound by [compute](#), scheduling it alongside jobs bound by [MPI](#) and [I/O](#) may reduce overall trapped capacity.

CPU

A breakdown of the 97.2% CPU time:

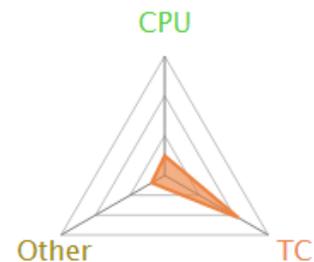
Scalar numeric ops	17.3%	■
Vector numeric ops	37.8%	■
Memory accesses	44.9%	■

Significant time is spent on memory accesses. Memory operations generally require less power than computation. To reduce trapped capacity use a profiler to identify time-consuming loops and check their cache performance.

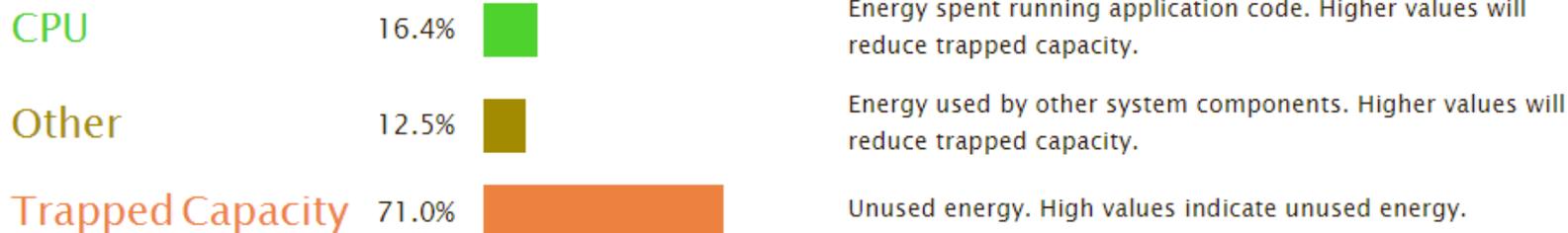
Example 2: CPU idle due to inefficient OpenMP scheduling



Command: `/home/abowen/work/trapped-examples/umult-omp-static 300 1024`
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 1 process, OMP_NUM_THREADS was 8
Machine: node-5
Start time: Fri Jul 22 2016 14:39:01 (UTC+01)
Total time: 301 seconds (about 5 minutes)
Full path: `/home/abowen/work/trapped-examples`



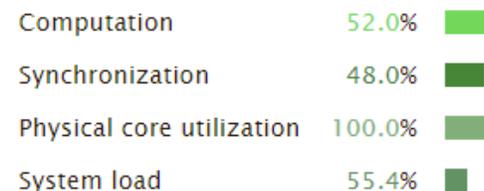
Energy Usage



To reduce trapped capacity look at OpenMP overhead first, for more information see the [OpenMP](#) section below. In addition, look at vectorization, see the [CPU](#) section below. This job is bound by [compute](#), scheduling it alongside jobs bound by [MPI](#) and [I/O](#) may reduce overall trapped capacity.

OpenMP

A breakdown of the 100.0% time in OpenMP regions:



Significant time is spent [synchronizing](#) threads. Sleeping CPU cores require minimal power. Check which locks cause the most overhead with a profiler and improve workload balance to reduce trapped capacity.

Example 2: Alinea Forge identifies the inefficient loop

```
75 #pragma omp parallel for schedule(static)
76 #endif
77 for(int i=0; i<size; i++)
78 {
79     for(int j=0; j<size; j++)
80     {
81         double res = 0.0;
82
83         for(int k=i; k<size; k++)
84         {
85             res += A[i*size+k]*Btranspose[j*size+k];
86             res += A[i*size+k]*Btranspose[j*size+k];
87         }
88
89         C[i*size+j] += res;
```

Input/Output | Project Files | OpenMP Stacks | OpenMP Regions | Functions

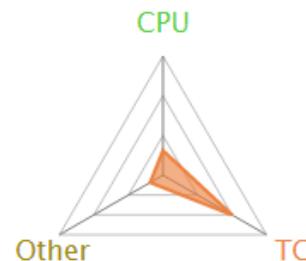
OpenMP Regions

Total core time	Overhead	Function(s) on line	Position
26.5%		umult-omp-static [program]	umult-omp.c:131
25.2%		umult [inlined]	umult-omp.c:75
0.2%		umult [OpenMP region 2]	umult-omp.c:85
0.2%		2 others	umult-omp.c:86
47.9%	<0.1%	1 other	[OpenMP overhead (no region active)]

Example 2: Dynamic scheduling reduces trapped capacity



Command: /home/abowen/work/trapped-examples/umult-omp-dynamic 300 1024
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 1 process, OMP_NUM_THREADS was 8
Machine: node-5
Start time: Fri Jul 22 2016 14:44:34 (UTC+01)
Total time: 301 seconds (about 5 minutes)
Full path: /home/abowen/work/trapped-examples



OpenMP

A breakdown of the 100.0% time in OpenMP regions:

Computation	99.9%	█
Synchronization	<0.1%	
Physical core utilization	100.0%	█
System load	100.1%	█

CPU

A breakdown of the 100.0% CPU time:

Single-core code	0.0%	
OpenMP regions	100.0%	█
Scalar numeric ops	85.1%	█
Vector numeric ops	0.0%	
Memory accesses	14.9%	█

No time is spent in **vectorized instructions**. Vectorized instructions generally require more power than scalar computation. To reduce trapped capacity check the compiler's vectorization advice to see if key loops can be vectorized.

Energy Usage

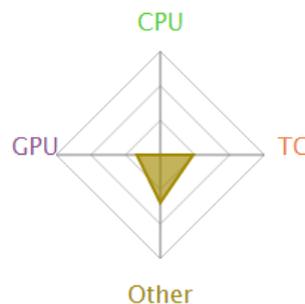
CPU	20.7%	█	Energy spent running application code. Higher values will reduce trapped capacity.
Other	12.3%	█	Energy used by other system components. Higher values will reduce trapped capacity.
Trapped Capacity	66.9%	█	Unused energy. High values indicate unused energy.

To reduce trapped capacity look at vectorization first, for more information see the **CPU** section below. In addition, look at memory accesses, also in the **CPU** section. This job is bound by **compute**, scheduling it alongside jobs bound by **MPI** and **I/O** may reduce overall trapped capacity.

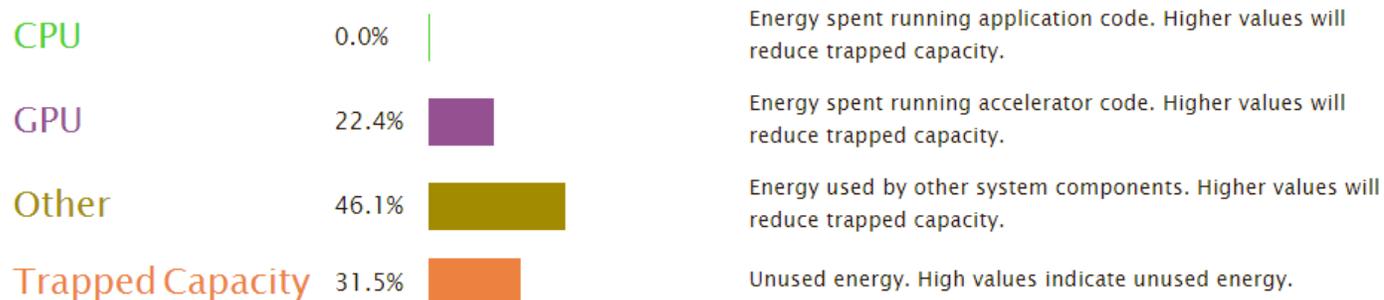
Example 3: GPU accelerators unused by application



Command: /home/abowen/work/trapped-examples/gpu-waiting-nogpu 300 12
Resources: 1 node (12 physical, 24 logical cores per node)
2 GPUs per node available
Memory: 15 GiB per node, 11 GiB per GPU
Tasks: 1 process
Machine: rhel-7-amd64
Start time: Wed Jul 20 2016 11:38:37 (UTC+01)
Total time: 307 seconds (about 5 minutes)
Full path: /home/abowen/work/trapped-examples



Energy Usage

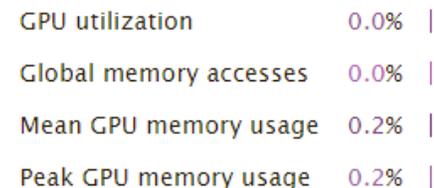


CPU metrics are not supported (no intel_rapl module)

To reduce trapped capacity look at accelerator usage first, for more information see the [Accelerators](#) section below. In addition, look at vectorization, see the [CPU](#) section below. This job is bound by [compute](#), scheduling it alongside jobs bound by [MPI](#), [accelerator usage](#), and [I/O](#) may reduce overall trapped capacity.

Accelerators

A breakdown of how accelerators were used:

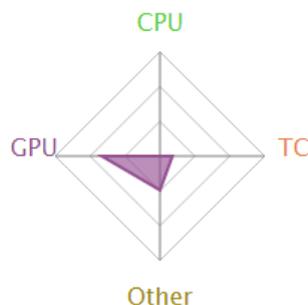


[Accelerators](#) are available but are not used. It may be more efficient to make use of accelerators or run on nodes without them.

Example 3: Developer uses CUDA library. Are we done?



Command: /home/abowen/work/trapped-examples/gpu-waiting 300 12
Resources: 1 node (12 physical, 24 logical cores per node)
2 GPUs per node available
Memory: 15 GiB per node, 11 GiB per GPU
Tasks: 1 process
Machine: rhel-7-amd64
Start time: Wed Jul 20 2016 11:27:05 (UTC+01)
Total time: 301 seconds (about 5 minutes)
Full path: /home/abowen/work/trapped-examples



CPU

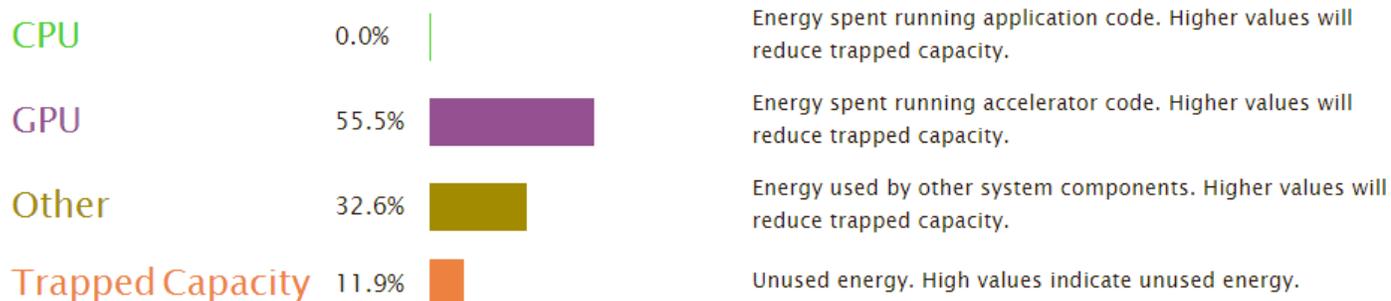
A breakdown of the 100.0% CPU time:

Scalar numeric ops	0.0%
Vector numeric ops	0.0%
Memory accesses	0.0%
Waiting for accelerators	99.9%

Most of the time is spent **waiting for accelerators**. Sleeping CPU cores require minimal power. To reduce trapped capacity use asynchronous calls to overlap CPU and accelerator workloads.

Significant time is spent on **memory accesses**. Cache-inefficient memory operations generally require less power than computation. To reduce trapped capacity use a profiler to identify time-consuming loops and check their cache performance.

Energy Usage



CPU metrics are not supported (no intel_rapl module)

To reduce trapped capacity look at time spent waiting for accelerators first, for more information see the [CPU](#) section below. In addition, look at accelerator usage, see the [Accelerators](#) section below.

Accelerators

A breakdown of how accelerators were used:

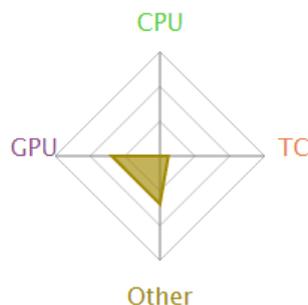
GPU utilization	48.6%
Global memory accesses	13.7%
Mean GPU memory usage	0.9%
Peak GPU memory usage	0.9%

The **accelerator** usage is low. Increasing the accelerator usage could reduce the trapped capacity.

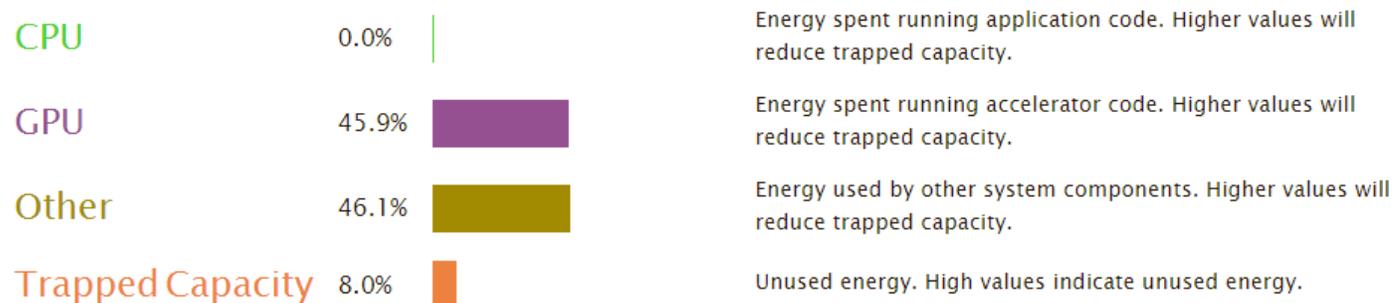
Example 3: Expert helps user overlap CPU and GPU work



Command: /home/abowen/work/trapped-examples/gpu-waiting-async 300 12
Resources: 1 node (12 physical, 24 logical cores per node)
2 GPUs per node available
Memory: 15 GiB per node, 11 GiB per GPU
Tasks: 1 process
Machine: rhel-7-amd64
Start time: Wed Jul 20 2016 11:32:57 (UTC+01)
Total time: 304 seconds (about 5 minutes)
Full path: /home/abowen/work/trapped-examples



Energy Usage

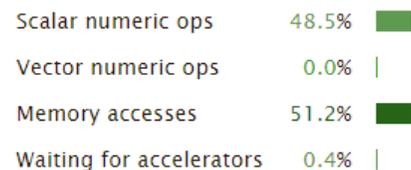


CPU metrics are not supported (no intel_rapl module)

To reduce trapped capacity look at accelerator usage first, for more information see the [Accelerators](#) section below. In addition, look at vectorization, see the [CPU](#) section below. This job is bound by [compute](#), scheduling it alongside jobs bound by [MPI](#), [accelerator usage](#), and [I/O](#) may reduce overall trapped capacity.

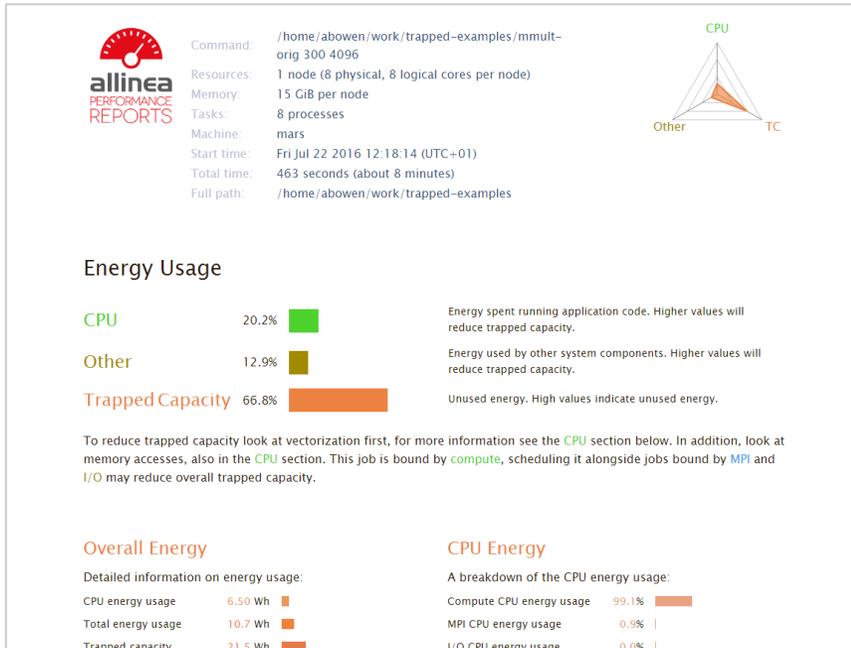
CPU

A breakdown of the 100.0% CPU time:

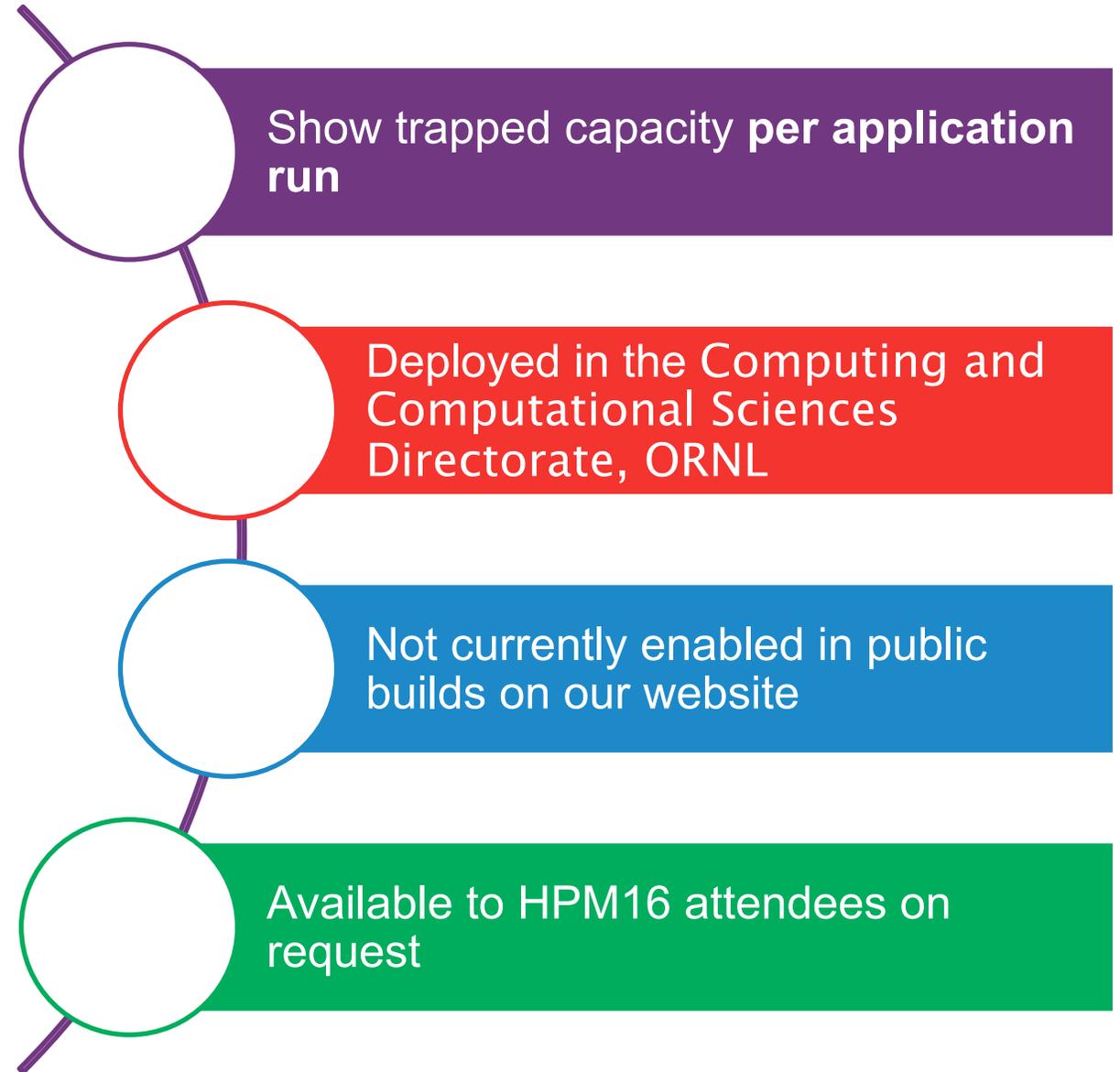


Significant time is spent on [memory accesses](#). Cache-inefficient memory operations generally require less power than computation. To reduce trapped capacity use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in [vectorized instructions](#). Vectorized instructions generally require more power than scalar computation. To reduce trapped capacity check the compiler's vectorization advice to see if key loops can be vectorized.



Application-centric Trapped Capacity Reports



Application Trapped Capacity and Energy Reporting

allinea
DEVELOPER TOOLS PERFORMANCE REPORTS TRIALS PURCHASE SUPPORT RESOURCES NEWS EVENTS COMPANY

allinea PERFORMANCE REPORTS
DEVELOPERS THE INDUSTRY STANDARD HPC C++ AND F90 DEVELOPMENT SUITE
allinea FORGE

ANALYSTS APPLICATION ANALYTICS FOR HPC CLUSTERS
THE WORLD'S MOST SCALABLE DEBUGGER
allinea DDV
INCREASE APPLICATION PERFORMANCE
allinea MAP

allinea
The problem we solve...

allinea PERFORMANCE REPORTS
How well do your applications exploit your hardware? Allinea Performance Reports are the most effective way to characterize and understand the performance of HPC application runs.

allinea FORGE
Allinea Forge is the complete tool suite for software development - with everything needed to debug, profile, optimize, edit and build applications for high performance.

Today: Application-centric performance tuning and energy metrics

allinea PERFORMANCE REPORTS

```
Command: /home/abowen/work/trapped-examples/mmult-orig 300 4096
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 8 processes
Machine: mars
Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
Total time: 463 seconds (about 8 minutes)
Full path: /home/abowen/work/trapped-examples
```



Energy Usage

CPU 20.2% Energy spent running application code. Higher values will reduce trapped capacity.

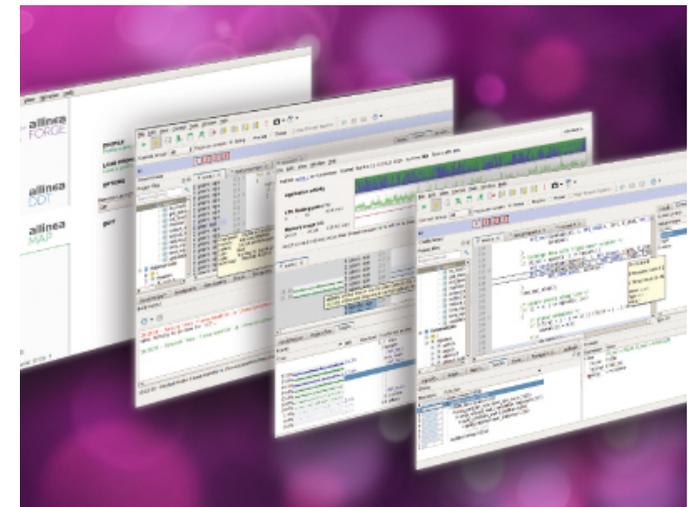
Other 12.9% Energy used by other system components. Higher values will reduce trapped capacity.

Trapped Capacity 66.8% Unused energy. High values indicate unused energy.

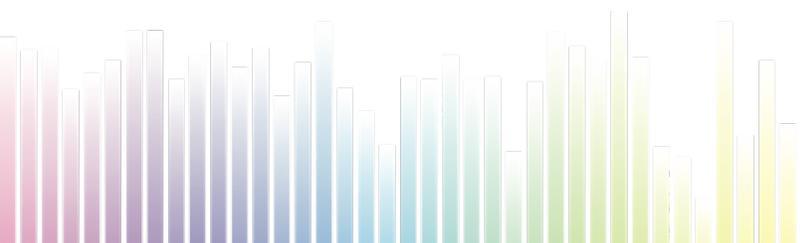
To reduce trapped capacity look at vectorization first, for more information see the CPU section below. In addition, look at memory accesses, also in the CPU section. This job is bound by `compute`, scheduling it alongside jobs bound by `MPI` and `I/O` may reduce overall trapped capacity.

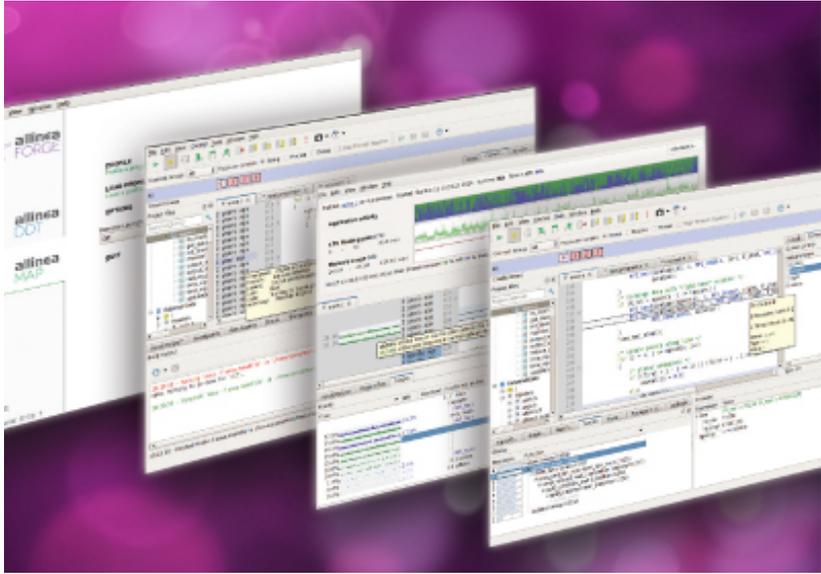
Overall Energy CPU Energy

New research: Trapped Capacity Reports on a per-application basis



Future research: energy-aware scheduling, superoptimizing compilers, ...





Active research

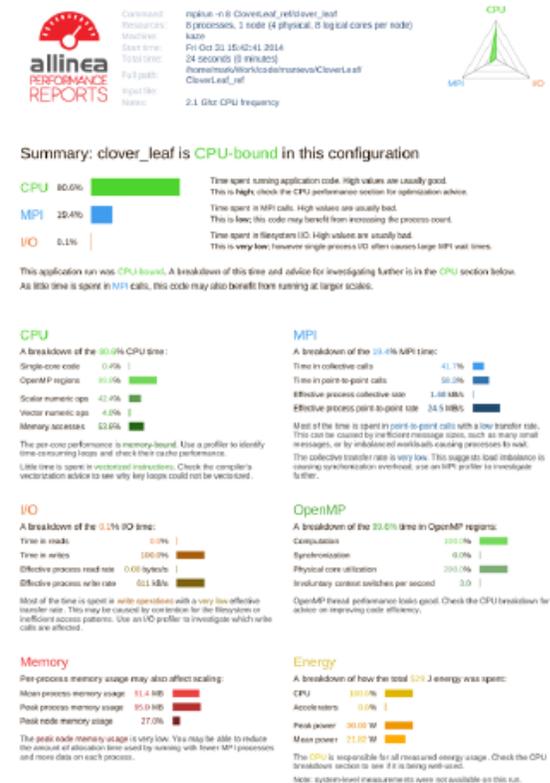
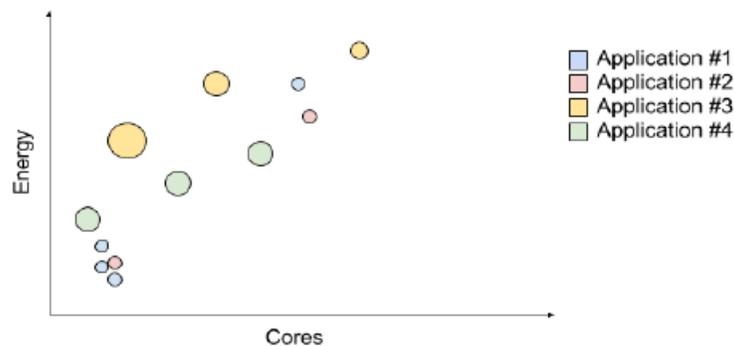
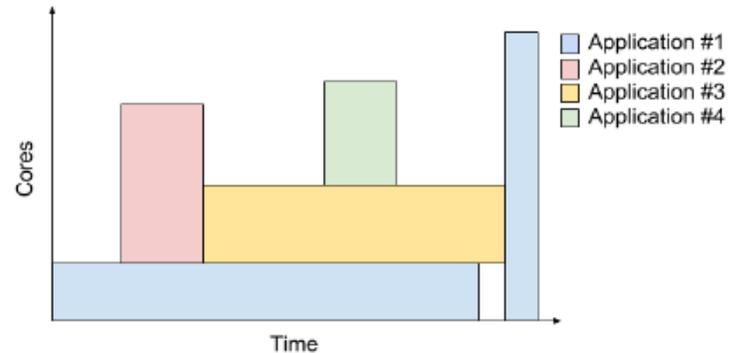
Ongoing research:

- Application Energy Dashboard
- Provide energy cost model for superoptimizing compiler
- Feed machine learning algorithm that chooses optimal compilation flags
- Data- and energy-aware scheduling via SLURM integration

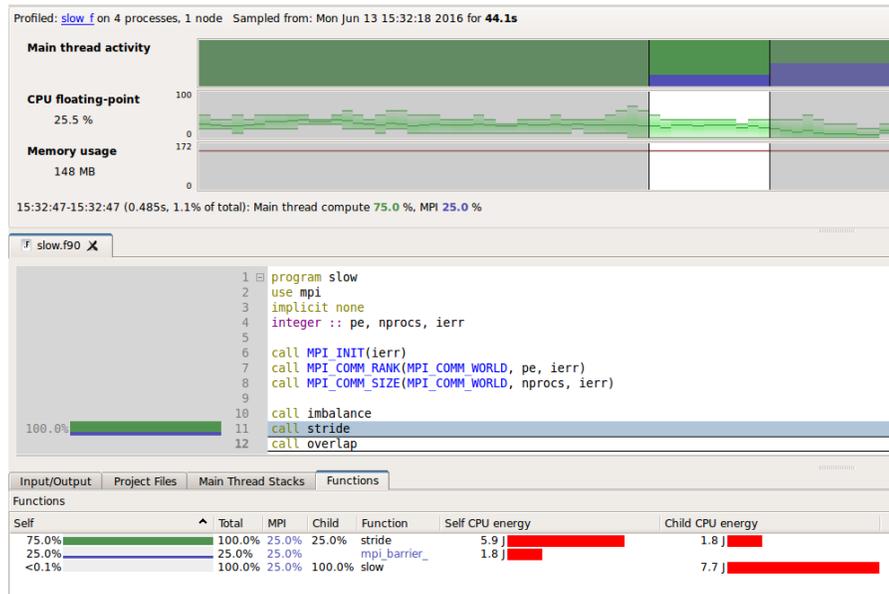
Total Software Energy Reporting and Optimization

Application Energy Dashboard

- **Collect** per-application energy metrics
- **Visualize** application runs and energy usage over time
- **Dive** into particular applications for a Performance Report



Total Software Energy Reporting and Optimization



Superoptimizing compiler

- MAP provides function-level performance data to the compiler
- EMBECOSM's stochastic superoptimizer targets most energy-costly functions
- Reruns under MAP to measure impact of changes

Machine Guided Energy Efficient Compilation

- Building on existing MAGEEC project for embedded systems
- Uses machine learning to predict compiler option combinations that boost energy efficiency
- Extending to take additional data from MAP in the input vector

Total	MPI	Child	Function	Self CPU energy
100.0%	25.0%	25.0%	stride	5.9 J
25.0%	25.0%		mpi_barrier_	1.8 J
100.0%	25.0%	100.0%	slow	

Horizon 2020: Next Generation I/O for Exascale

Work package overview

Target: **50% reduction in energy to solution** when compared to today's systems

Target: 20x improved I/O performance for **real applications**

Performance Reports will feed energy and I/O measurements into the SLURM workload scheduler

Scheduler integration

Preload data onto nodes before job starts or schedule work close to existing data location

Schedule compute-intensive jobs on cooler parts of the system

Limit clock speed on jobs known to be memory-bound

Application Trapped Capacity and Energy Reporting

The screenshot shows the allinea website with navigation links for Developer Tools, Performance Reports, Trials, Purchase, Support, Resources, News, Events, and Company. Key sections include:

- allinea PERFORMANCE REPORTS**: A red heart-rate graphic.
- DEVELOPERS**: THE INDUSTRY STANDARD HPC C++ AND F90 DEVELOPMENT SUITE.
- allinea FORGE**: THE WORLD'S MOST SCALABLE DEBUGGER.
- allinea DDT**: INCREASE APPLICATION PERFORMANCE.
- allinea MAP**: APPLICATION ANALYTICS FOR HPC CLUSTERS.

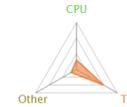
Below the main navigation, there are three columns of text:

- allinea**: The problem we solve... From climate modeling to astrophysics, from financial modeling to engine design, the power of clusters and supercomputers advances the frontiers of knowledge and delivers results for industry. Writing and deploying software that exploits that computing power is a
- allinea PERFORMANCE REPORTS**: How well do your applications exploit your hardware? Allinea Performance Reports are the most effective way to characterize and understand the performance of HPC application runs.
- allinea FORGE**: Allinea Forge is the complete tool suite for software development - with everything needed to debug, profile, optimize, edit and build applications for high performance.

Today: Application-centric performance tuning and energy metrics



Command: /home/abowen/work/trapped-examples/mmult-orig 300 4096
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GiB per node
Tasks: 8 processes
Machine: mars
Start time: Fri Jul 22 2016 12:18:14 (UTC+01)
Total time: 463 seconds (about 8 minutes)
Full path: /home/abowen/work/trapped-examples



Energy Usage

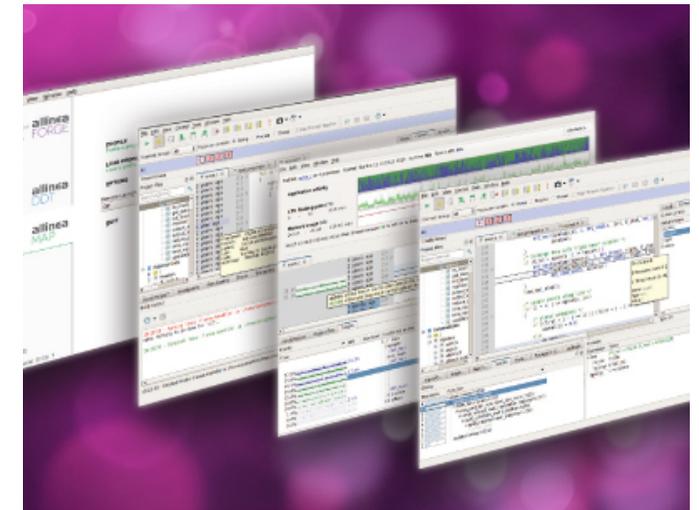


To reduce trapped capacity look at vectorization first, for more information see the CPU section below. In addition, look at memory accesses, also in the CPU section. This job is bound by `compute`, scheduling it alongside jobs bound by `MPI` and `I/O` may reduce overall trapped capacity.

Overall Energy

CPU Energy

New research: Trapped Capacity Reports on a per-application basis



Future research: energy-aware scheduling, superoptimizing compilers, ...



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

Technology wish list: open cross-vendor energy APIs (that do *not* require root access)

Mark O'Connor

VP Product Management

