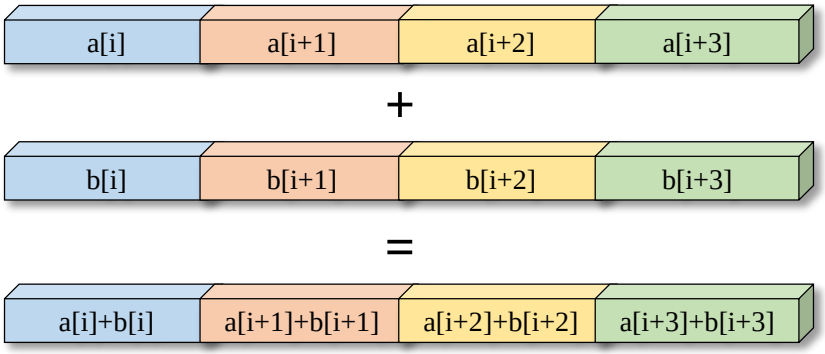
Introduction to Performance Tuning & Optimization Tools



Ian A. Cosden, Ph.D.

Manager, HPC Software Engineering and Performance Tuning

Research Computing, OIT, Princeton University

icosden@princeton.edu

CoDaS-HEP Summer School

July 13, 2017

OFFICE of INFORMATION TECHNOLOGY

PICSciE

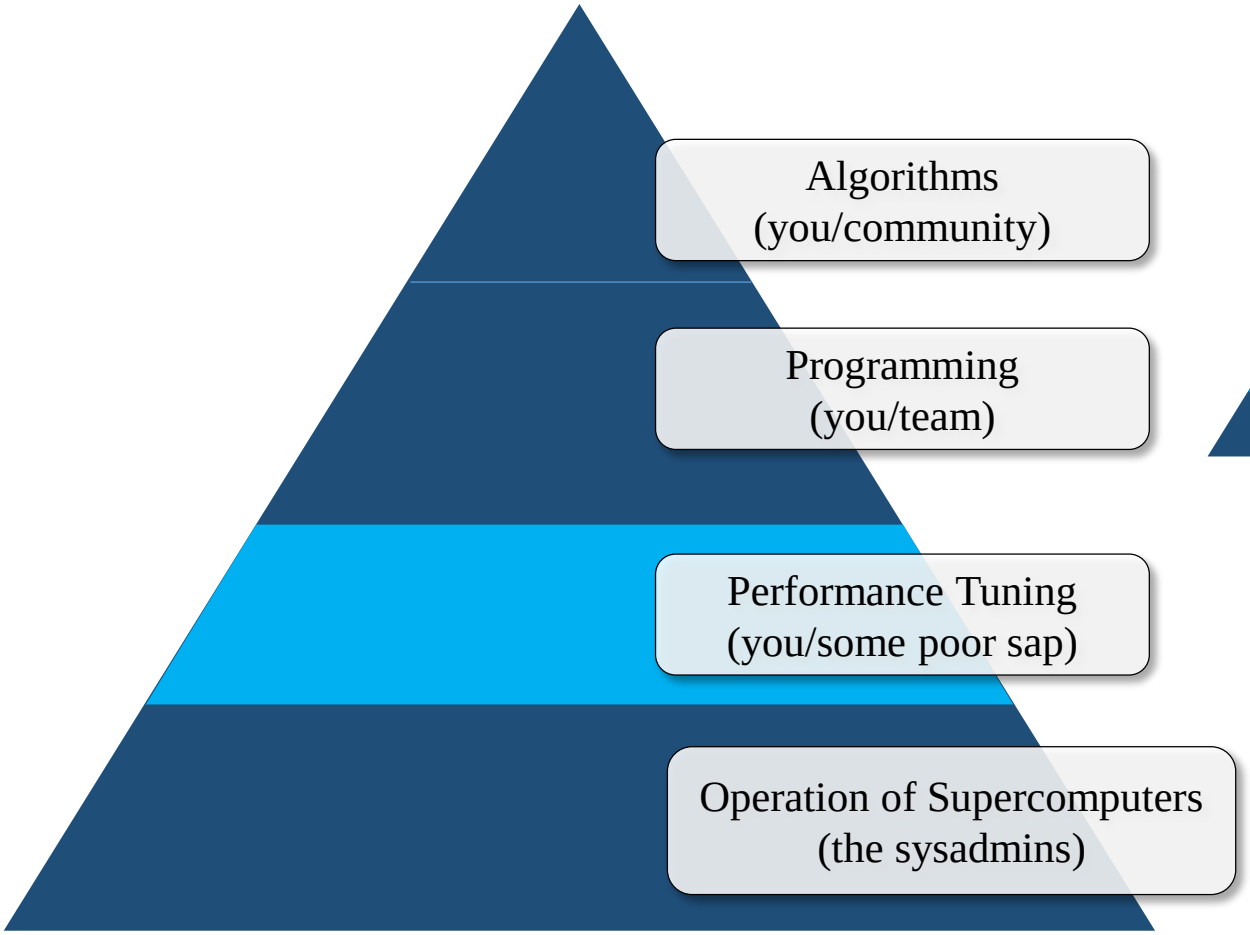
Outline

- What is Performance Tuning?
- Performance Tuning Workflow
- Software Tools
  - Hotspots: Intel VTune
  - OpenMP Scaling: Intel VTune
  - Vectorization: Intel Advisor
- Hands-on VTune Exercise
- Hardware Counters: VTune

What is Performance?

- "Performance is the degree to which a computing system meets expectations of the persons involved in it." Doherty (1970)
  - Is it doing what I want?
- "Performance…is the effectiveness with which the resources of the host computer system are utilized toward meeting the objectives of the software system." Graham (1973)
  - Is it utilizing the resources well?

What is Tuning?



Applicable to all domains

Where does Tuning fit?

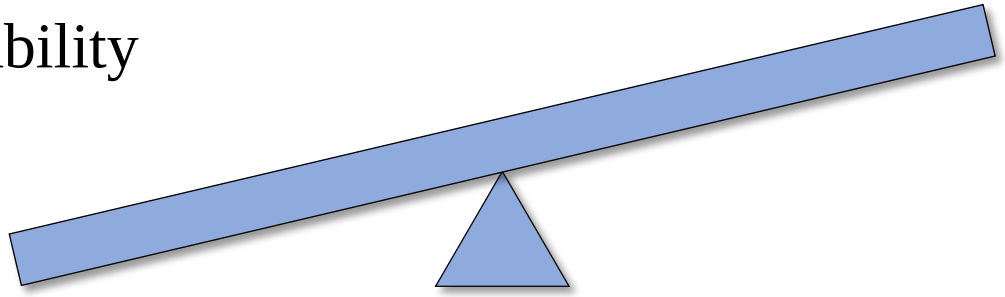
- Performance should <u>not</u> be ignored during development
  - Algorithm selection
    - Vectorization and Parallelization
  - Data structures
  - Hardware awareness
- *Tuning* typically happens near the end
  - What if you invest time tuning code you never use?

- Principles and techniques applicable at any stage
- We'll assume we are tuning an underperforming correct code

Tuning Tradeoff

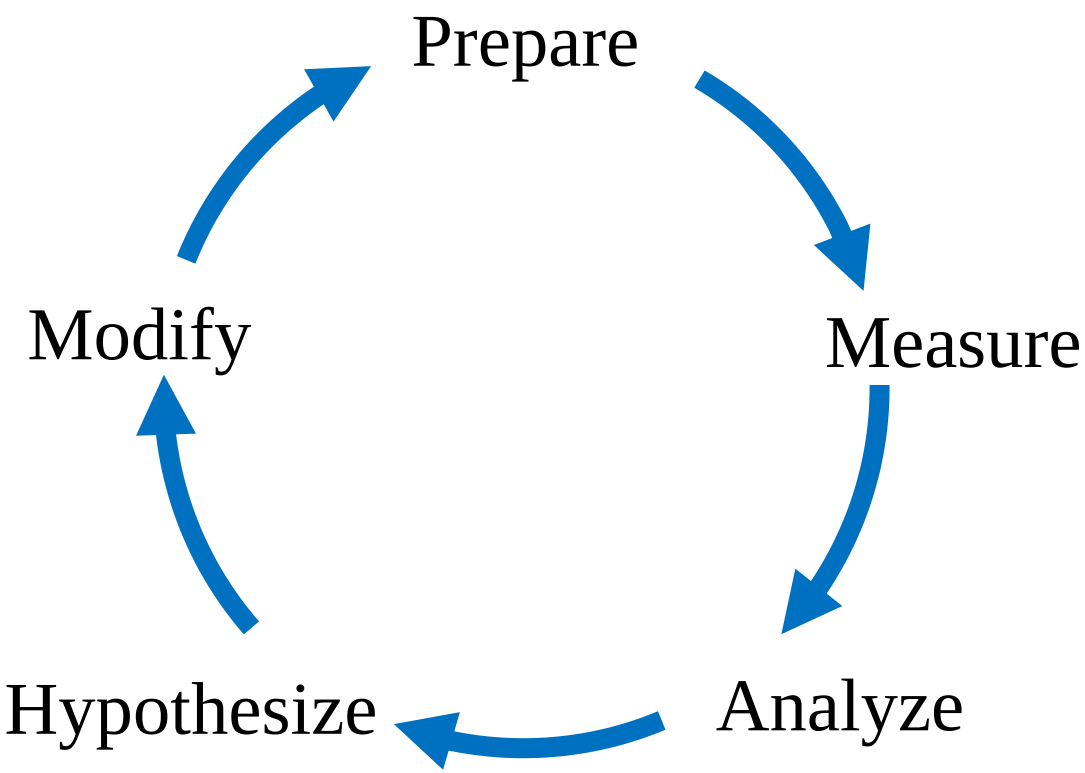
- Readability
- Simplicity
- Maintainability
- Time

- Performance



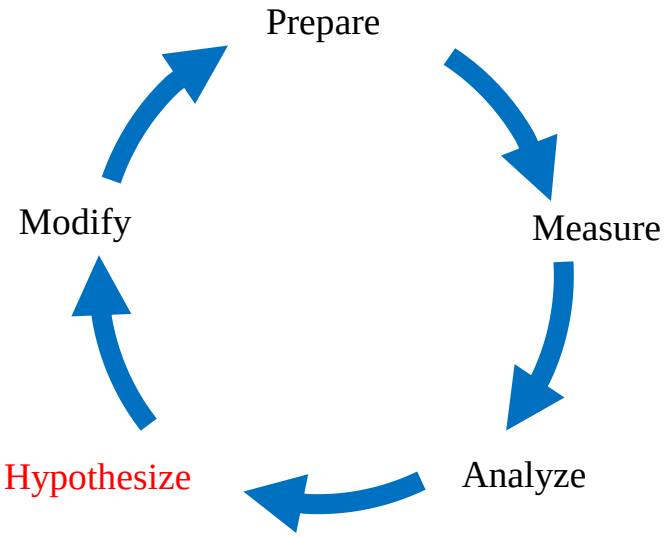
- Set goals - *Faster*
  - It could always be faster.
  - How much faster? Be specific.
  - Set an upper limit on your time investment

Performance Tuning Workflow



Hypothesis

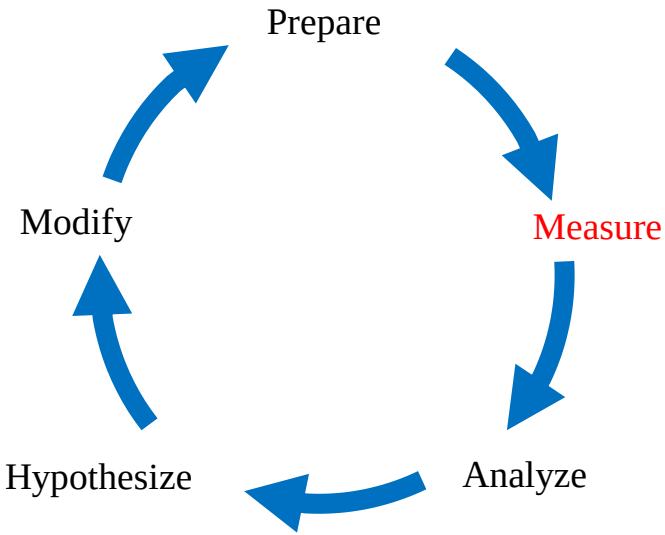
- Why is my code slow?
  - CPU bound
  - Memory bound
  - I/O bound
  - Network bound
  - Unbalanced Workload (Parallel)



- What is the best I can expect?
  - CPU
  - Memory/Cache
  - I/O
  - Network
  - Parallel Scaling

Measure

- Need some kind of monitor
- Direct Measurement a.k.a. *Instrumentation*
  - Event is what we want to measure
  - Accurate
  - High overhead for frequent events
  - Ex. Tracing
- Indirect Measurement a.k.a. *Sampling*
  - Record system state at regular interval
  - Typically low overhead
  - Not every event recorded
  - Ex. Profiling
- Measurement can influence result



Measuring

- Measuring the performance of your code
  - You can't fix what you can't see
  - Find the "hotspots"
    - How much time is spent in each function
    - Not always where you think it is
    - Identify regions to optimize/parallelize
  - Hardware Performance
    - Vectorization, cache misses, branch misprediction, etc.

- Do it yourself
  - Put time calls around loops/functions
  - Only works if:
    - Done carefully during development
    - On a small code base
- Use a tool

Easy Measures - Linux

- Total Runtime ("time")
  - Likely not enough
  - Critically important number
- CPU usage ("top")
  - N*100% cpu = N threads running full-tilt
  - 100% ≠ 100% peak theoretical performance
  - Useful for threading efficiency only

```
top - 08:32:06 up 8 days, 19:47,  7 users,  load average: 0.72, 0.16, 0.09
Tasks: 771 total,   2 running, 769 sleeping,   0 stopped,   0 zombie
%Cpu(s): 42.8 us,  0.1 sy,  0.0 ni, 57.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 26372923+total, 25595838+free,  2061328 used,  5709524 buff/cache
KiB Swap:  8191996 total,  8191996 free,        0 used. 26107923+avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  3863 icosden   20   0  857096   5788   1620 R  1201  0.0   1:04.35 app-ICC
```

- Slurm feedback ("seff")
  - Essentially same as top for a cluster
  - Only useful for threading/MPI efficiency and I/O

```
[icosden@tiger1 iter1-omp]$ seff 1878021
Job ID: 1878021
Cluster: tiger
User/Group: icosden/cses
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 16
CPU Utilized: 00:10:06
CPU Efficiency: 99.67% of 00:10:08 core-walltime
Memory Utilized: 30.72 MB
Memory Efficiency: 0.06% of 46.88 GB
```

Untuned

```
[icosden@tiger1 iter5-tiled]$ seff 1878019
Job ID: 1878019
Cluster: tiger
User/Group: icosden/cses
State: COMPLETED (exit code 0)
Nodes: 1
Cores per node: 16
CPU Utilized: 00:02:14
CPU Efficiency: 104.69% of 00:02:08 core-walltime
Memory Utilized: 1.56 MB
Memory Efficiency: 0.00% of 46.88 GB
```

Tuned

Performance Tools

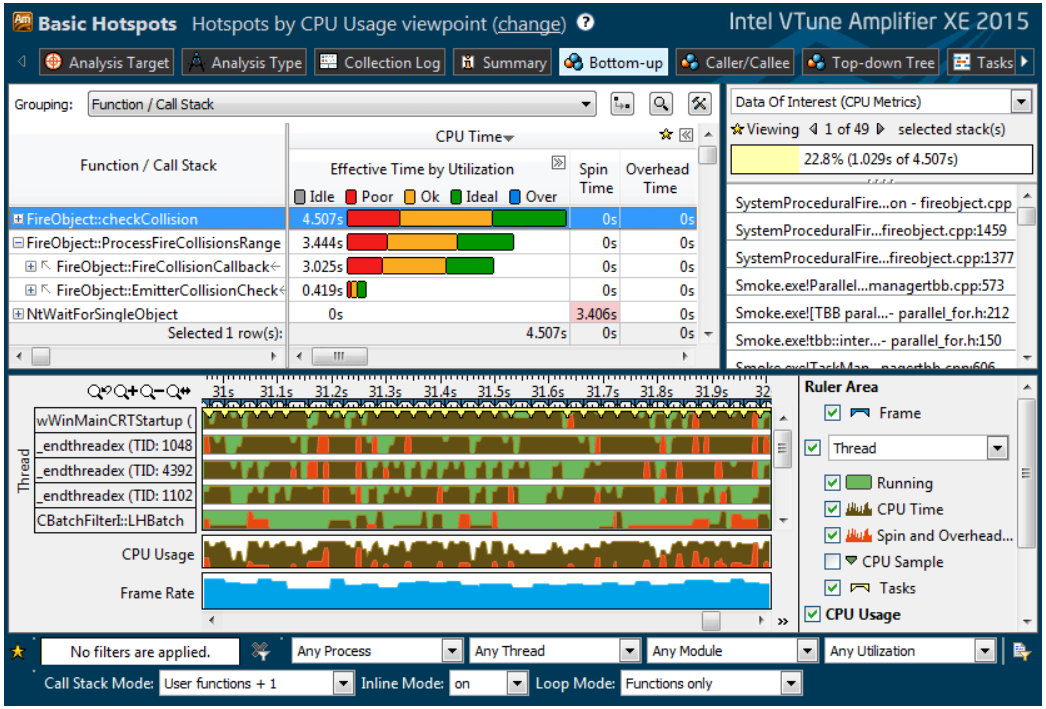
- Many free and commercial products
- Site specific – but most places have the major products

- Intel Parallel Studio
  - VTune Amplifier XE
  - Advisor

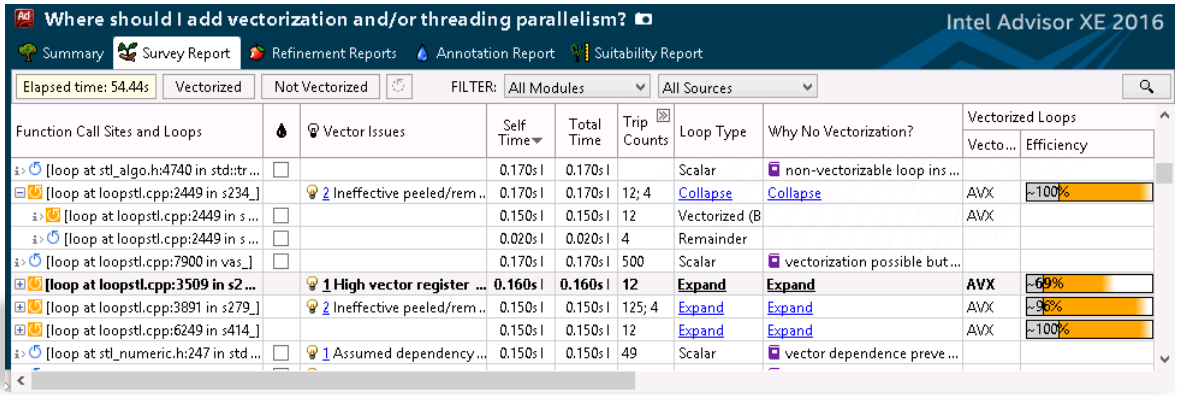
- Many others: Allinea Forge (MAP), Tau, Intel Trace Analyzer and Collector, HPCToolkit, gprof, perf, gperftools, …

Intel VTune

- Intel VTune Amplifer XE
  - Commercial Profiler
  - Extraordinarily powerful (and complicated)
  - Nice GUI
- Source code profiling
- Shared memory only
  - Serial
  - OpenMP
  - MPI on single node



Intel Advisor



- Intel Advisor
  - Vectorization (and threading) advisor
- Identifies loops to target for vectorization
  - Provides efficiency statistics and tips for improvement
- Roofline Analysis (new)

- In many ways it's a simple profiler + GUI vec-report

Using Profilers

- Strength of all profiling tools revolves around ability to trace performance back to source code
  - Need to include debug symbols in executable
    - -g flag
- Use release-build optimization flags
  - Ex. -O3, -xhost (Intel)
  - Don’t waste time optimizing code the compiler can do automatically!
- Sometimes the compiler will optimize out useful regions
  - Recommendations:
    - -debug inline-debug-info, -debug full (Intel)
- For difficult problems use more than one profiler

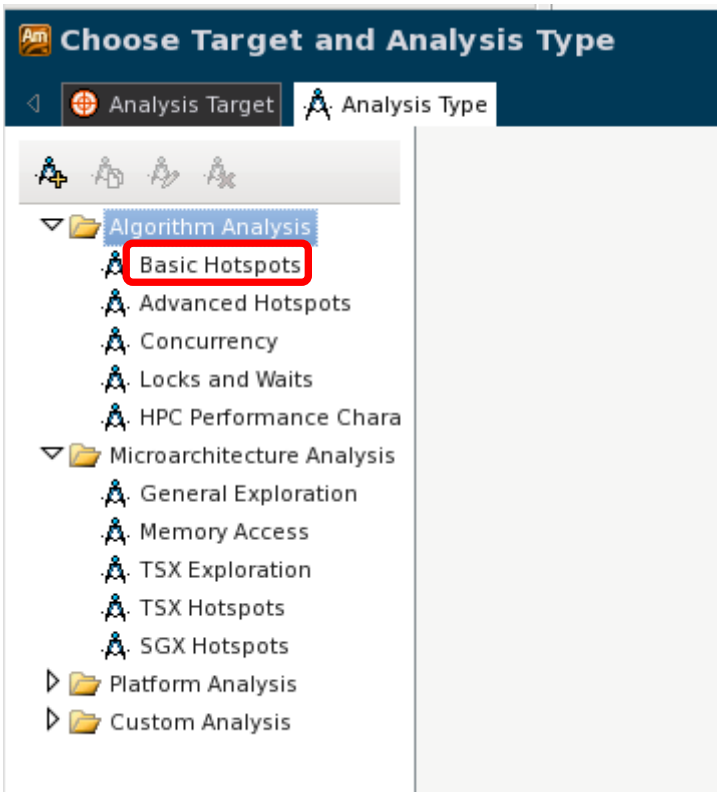
Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune

VTune Example

- Code compiled with -g to enable source code profiling
- Example serial C++ code with:
  - No multithreading
  - No MPI
  - Some I/O
- Use "Basic Hotspots" analysis
  - This will be subject of hands-on exercise later



VTune – Profile Result

/home/icosden/intel/amplxe/projects/simple_md_code - Intel VTune Amplifier

r000hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 2016

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 11.193s

CPU Time:	10.760s
Total Thread Count:	1
Paused Time:	0s

Top Hotspots

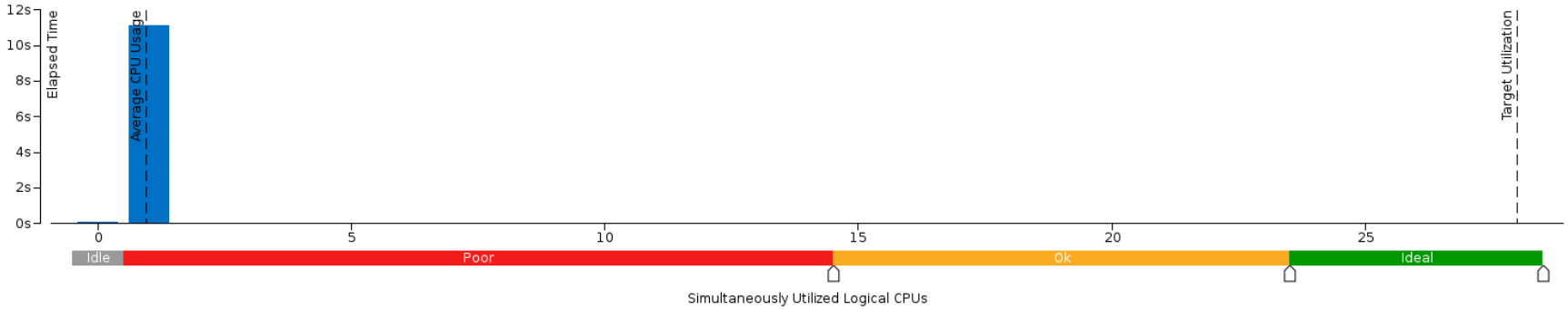
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
Atom::calc_force	md.out	9.073s
_IO_fprintf	libc.so.6	1.176s
std::pow	md.out	0.271s
_IO_fclose	libc.so.6	0.080s
std::pow	md.out	0.060s
[Others]	N/A*	0.100s

*N/A is applied to non-summable metrics.

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line: /home/icosden/svn/IntroPerfTuning/trunk/example_code/simple_md_code/md.out input.txt

Operating System: 3.10.0-327.36.3.el7.x86_64 NAME="Springdale Linux" VERSION="7.2 (Verona)" ID="rhel" ID_LIKE="fedora" VERSION_ID="7.2" PRETTY_NAME="Springdale Linux 7.2 (Verona)" ANSI_COLOR="0;32"

Bottom-up Tab (Functions)

/home/icosden/intel/amplxe/projects/simple_md_code - Intel VTune Amplifier

r000hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 2016

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Over... Time	Module	Function (Full)
Atom::calc_force	9.073s	0s	0s	md.out	Atom::calc_force(void)
_IO_fprintf	1.176s	0s	0s	libc.so.6	_IO_fprintf
std::pow	0.271s	0s	0s	md.out	std::pow(double, int)
_IO_fclose	0.080s	0s	0s	libc.so.6	_IO_fclose
std::pow	0.060s	0s	0s	md.out	std::pow(double, int)
std::vector<double, std::allocator<dc	0.040s	0s	0s	md.out	std::vector<double, std::allocator<double>>::operator[](unsigned long)
std::vector<double, std::allocator<dc	0.030s	0s	0s	md.out	std::vector<double, std::allocator<double>>::operator[](unsigned long)
std::vector<double, std::allocator<dc	0.020s	0s	0s	md.out	std::vector<double, std::allocator<double>>::operator[](unsigned long)
__fopen_internal	0.010s	0s	0s	libc.so.6	__fopen_internal
Selected 1 row(s):	9.073s	0s	0s		

CPU Time

Viewing 1 of 2 selected stack...

99.9% (9.063s of 9.073s)

md.out!Atom::calc_force - atom.cpp
md.out!Md_sim::run+0x1ab - md_s...
md.out!main+0x72 - main.cpp:12
libc.so.6!__libc_start_main+0xf4 - [...
md.out!_start+0x28 - [unknown so...

md.out (TID: 136...)

Thread

CPU Usage

Thread: Running, CPU Time, Spin and Ov..., CPU Sample

CPU Usage: CPU Time, Spin and Ov...

FILTER 100.0% | Process: Any Process | Thread: Any Thread | Module: Any Module | Any Utilization | Call Stack Mode: User functions | Inline Mode: Show inline fur | Loop Mode: Functions only

Bottom-up (Loops)

/home/icosden/intel/amplxe/projects/simple_md_code - Intel VTune Amplifier

Basic Hotspots Hotspots by CPU Usage viewpoint (change) — Intel VTune Amplifier XE 2016

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | atom.cpp

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Over... Time	Module	Function (Full)
[Loop at line 80 in Atom::calc_force]	9.474s	0s	0s	md.out	[Loop at line 80 in Atom::calc_force(void)]
[Loop at line 228 in Md_sim::write]	1.166s	0s	0s	md.out	[Loop at line 228 in Md_sim::write(void)]
[Loop at line 76 in Md_sim::run]	0.100s	0s	0s	md.out	[Loop at line 76 in Md_sim::run(void)]
[Loop at line 78 in Atom::calc_force]	0.010s	0s	0s	md.out	[Loop at line 78 in Atom::calc_force(void)]
[Loop at line 74 in Atom::calc_force]	0.010s	0s	0s	md.out	[Loop at line 74 in Atom::calc_force(void)]
Selected 1 row(s):	9.474s	0s	0s		

CPU Time — Viewing 1 of 3 selected stack... — 95.5% (9.043s of 9.474s)

md.out![Loop at line 80 in Atom::ca...
md.out![Loop at line 78 in Atom::ca...
md.out![Loop at line 76 in Md_sim::...
md.out![Outside any loop] - [unkno...

md.out (TID: 136...) — Thread — CPU Usage

Thread: Running, CPU Time, Spin and Ov..., CPU Sample
CPU Usage: CPU Time, Spin and Ov...

FILTER 100.0% | Process: Any Process | Thread: Any Thread | Module: Any Module | Any Utilization | Call Stack Mode: User functions | Inline Mode: Show inline fur | Loop Mode: Loops only

Source Code Hotspots (line-by-line)

/home/icosden/intel/amplxe/projects/simple_md_code - Intel VTune Amplifier

r000hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Intel VTune Amplifier XE 2016

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | atom.cpp

Source | Assembly | Assembly grouping: Address

So. Lin.	Source	CPU Time: Total – Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Over... Time	CPU Time: Self – Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	Spin Time	Over... Time
61	void Atom::calc_force() {						
62							
63							
64	PE=0;						
65	p_PE=0;						
66							
67	double RadiusSQ=rcut*rcut;						
68	double Xr, Yr, Zr, Xr2, Yr2, Zr2;						
69	double RijSQ;						
70	double r2i, r6i;						
71	double Fij, Fxij, Fyij, Fzij;						
72	double force;						
73							
74	for (int i=0; i<natoms; i++) {	0.1%	0.0%	0.0%	0.010s	0s	0s
75	fx[i]=fy[i]=fz[i]= 0.0;						
76	}						
77							
78	for(int i=0; i<(natoms-1); i++) {						
79							
80	for(int j=i+1; j<natoms; j++) {	2.0%	0.0%	0.0%	0.220s	0s	0s
81							
82	//Calculate distance between atoms						
83	Xr = rx[i] - rx[j];	1.4%	0.0%	0.0%	0.150s	0s	0s
84	Yr = ry[i] - ry[j];	0.8%	0.0%	0.0%	0.081s	0s	0s
85	Zr = rz[i] - rz[j];	0.1%	0.0%	0.0%	0.010s	0s	0s
86	Xr = Xr - box_x*round(Xr/box_x); //periodic boundary condition	10.9%	0.0%	0.0%	1.174s	0s	0s
87	Yr = Yr - box_y*round(Yr/box_y);	13.1%	0.0%	0.0%	1.412s	0s	0s
88	Zr = Zr - box_z*round(Zr/box_z);	12.5%	0.0%	0.0%	1.340s	0s	0s
89							
90	Xr2 = Xr*Xr; //square of distance	1.3%	0.0%	0.0%	0.140s	0s	0s
91	Yr2 = Yr*Yr;	0.4%	0.0%	0.0%	0.040s	0s	0s
92	Zr2 = Zr*Zr;	3.9%	0.0%	0.0%	0.420s	0s	0s
93							
94	RijSQ = Xr2 + Yr2 + Zr2;	4.4%	0.0%	0.0%	0.471s	0s	0s
95							
96	if(RijSQ<=RadiusSQ) {	6.7%	0.0%	0.0%	0.720s	0s	0s
97							
98	r2i = 1.0/RijSQ; //Intermediate Calcs for the LJ (6-12) potential	1.3%	0.0%	0.0%	0.142s	0s	0s
99	r6i = r2i*r2i*r2i;	7.5%	0.0%	0.0%	0.811s	0s	0s
100							
101	//Fij = 48.0*r2i*r6i*(r6i-0.5); //LJ Force in reduced units						
102							
103	force = 48.0*pow(1.0/RijSQ,6)-24*pow(1.0/RijSQ,3);	3.3%	0.0%	0.0%	0.350s	0s	0s
104	Fij = force*r2i;						
105							
106	PE=PE+r6i*(r6i-1.0);	8.4%	0.0%	0.0%	0.899s	0s	0s
107							
108	//Component Forces						
109	Fxij = Xr*Fij;						
110	Fyij = Yr*Fij;	0.1%	0.0%	0.0%	0.010s	0s	0s
111	Fzij = Zr*Fij;						
112							
113	//Sum forces on atom i						
114	fx[i] += Fxij;	0.7%	0.0%	0.0%	0.080s	0s	0s
Selec...		13.1%	0.0%	0.0%	1.412s	0s	0s

CPU Time

Viewing 1 of 2 selected stack...

99.9% (9.063s of 9.073s)

- md.out!Atom::calc_force - atom.cpp
- md.out!Md_sim::run+0x1ab - md_s...
- md.out!main+0x72 - main.cpp:12
- libc.so.6!__libc_start_main+0xf4 - [...
- md.out!_start+0x28 - [unknown so...

Bottom-up – Real Code

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Ove... Time
[redacted]	11.069s	0s	0s
	5.348s	0s	0s
	3.850s	0s	0s
	3.828s	0s	0s
	3.468s	0s	0s
	3.378s	0s	0s
	1.990s	0s	0s
	1.569s	0s	0s
	1.260s	0s	0s
	0.910s	0s	0s
	0.640s	0s	0s
	0.520s	0s	0s
	0.420s	0s	0s
	0.390s	0s	0s
	0.380s	0s	0s
	0.370s	0s	0s
	0.350s	0s	0s
	0.350s	0s	0s
	0.330s	0s	0s
	0.320s	0s	0s
	0.309s	0s	0s
	0.300s	0s	0s
	0.240s	0s	0s
	0.229s	0s	0s
	0.200s	0s	0s
	0.190s	0s	0s
	0.190s	0s	0s
	0.180s	0s	0s
	0.180s	0s	0s
	0.170s	0s	0s
	0.140s	0s	0s
	0.140s	0s	0s
	0.139s	0s	0s
	0.130s	0s	0s
	0.110s	0s	0s
	0.110s	0s	0s
	0.100s	0s	0s
	0.100s	0s	0s
	0.100s	0s	0s
	0.100s	0s	0s
	0.099s	0s	0s
	0.090s	0s	0s
	0.080s	0s	0s
	0.070s	0s	0s
	0.060s	0s	0s
	0.060s	0s	0s
Selected 1 row(s):	11.069s	0s	0s

FILTER 100.0% Process Any Proce Thread Any Thread Module Any Module

Same Real Code – Top-down

Function Stack	CPU Time: Total – Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Total – Spin Time	CPU Time: Total – Overhead Time	CPU Time: Self – Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)
Total	100.0%	0.0%	0.0%	0s
_start	100.0%	0.0%	0.0%	0s
__libc_start_main	100.0%	0.0%	0.0%	0s
main	100.0%	0.0%	0.0%	0.010s
[redacted]	99.9%	0.0%	0.0%	0.010s
[redacted]	72.4%	0.0%	0.0%	0s
[redacted]	70.9%	0.0%	0.0%	0s
[redacted]	56.2%	0.0%	0.0%	0s
[redacted]	56.2%	0.0%	0.0%	0.350s
[redacted]	39.3%	0.0%	0.0%	11.069s
[redacted]	5.7%	0.0%	0.0%	2.200s
[redacted]	4.1%	0.0%	0.0%	1.910s
[redacted]	1.9%	0.0%	0.0%	0.190s
[redacted]	1.4%	0.0%	0.0%	0.390s
[redacted]	0.8%	0.0%	0.0%	0.120s
[redacted]	0.6%	0.0%	0.0%	0.100s
[redacted]	0.3%	0.0%	0.0%	0.150s
[redacted]	0.3%	0.0%	0.0%	0.120s
[redacted]	0.1%	0.0%	0.0%	0.050s
[redacted]	0.1%	0.0%	0.0%	0.030s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	8.4%	0.0%	0.0%	0.100s
[redacted]	3.9%	0.0%	0.0%	0.240s
[redacted]	3.6%	0.0%	0.0%	0.060s
[redacted]	0.1%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	8.4%	0.0%	0.0%	0s
[redacted]	5.6%	0.0%	0.0%	0.099s
[redacted]	0.7%	0.0%	0.0%	0.020s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	1.3%	0.0%	0.0%	0s
[redacted]	0.2%	0.0%	0.0%	0.010s
[redacted]	0.1%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	25.2%	0.0%	0.0%	0s
[redacted]	24.9%	0.0%	0.0%	0s
[redacted]	0.3%	0.0%	0.0%	0s
[redacted]	1.0%	0.0%	0.0%	0s
[redacted]	0.6%	0.0%	0.0%	0s
[redacted]	0.6%	0.0%	0.0%	0.060s
[redacted]	0.1%	0.0%	0.0%	0s
Highlighted 1 row(s):	39.3%	0.0%	0.0%	11.069s

OFFICE of INFORMATION

CSciE

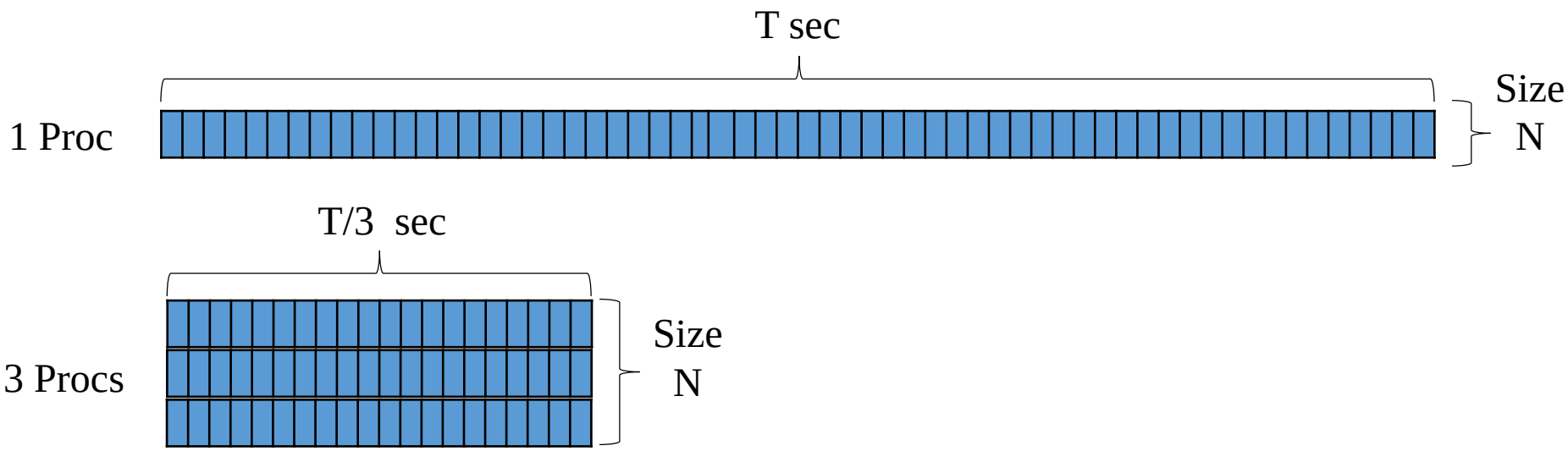
Examples

A handful of real(ish) codes to demonstrate profiling:

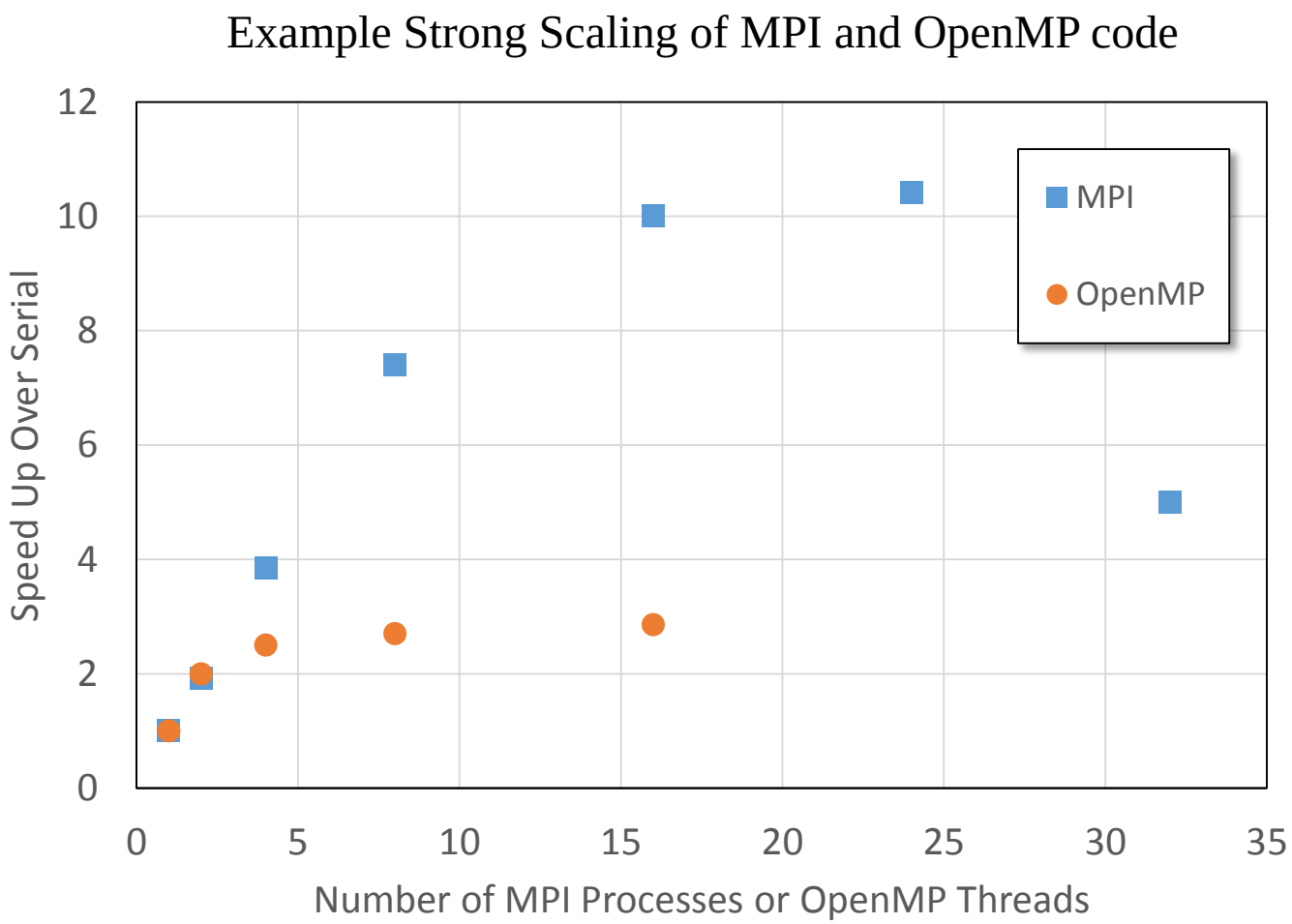
1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune

Strong Scaling

- Strong Scaling
  - Fixed problem size
  - Measure how solution time decreases with more processors

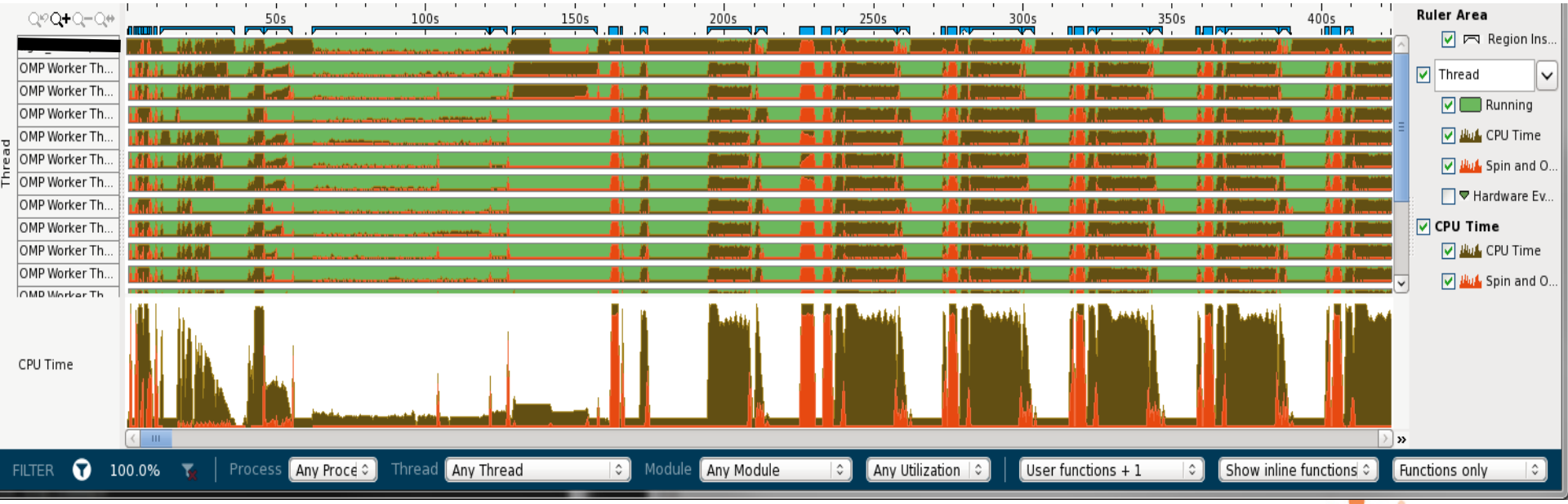


Scaling and Performance



OpenMP Threading Efficiency

- Example code was seeing ~3x speed up with 16 threads
  - Why? How to fix?
- Zoom in on threading timeline in VTune:
  - Brown = good
  - Green (idle) and Orange (overhead) = bad



Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune

Intel Advisor Example

- Example code compiled with intel compiler with flags: –no-vec and –no-simd
  - Purposefully turns *off* auto-vectorization
  - Useful with debugging or performance benchmarking only
    - Or faking poorly vectorized code
- Then built normally
  - –g (debugging symbols) and all release flags
- Ran on Broadwell node
  - AVX2 instruction set (FMA)
  - Two, 256-bit vector registers
    - Single precision (32 bit) floating point numbers: 8 per register possible

Poorly Vectorized - Advisor

Where should I add vectorization and/or threading parallelism?

Elapsed time: 10.19s | Vectorized | Not Vectorized | FILTER: All Module | All Sources | Loop | All Threads

Summary | Survey Report | Refinement Reports | Annotation Report

Higher instruction set architecture (ISA) available

Consider recompiling your application using a higher ISA.

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops			Instruction Set
						Vec...	Gai...	VL ...	Traits
[loop in MoveParticles at nbody.cc:27]	1 Potential...	231.026s	231.026s	Scalar	auto-vectorization is disabled with -no-vec flag				Square Roots
[loop in MoveParticles at nbody.cc:20]	2 Assume ...	0.010s	231.036s	Threaded (Op...	vector dependence prevents vectorization				
[loop in start_thread]		0.000s	222.980s	Scalar					
[loop in main at nbody.cc:101]	1 Assume ...	0.000s	8.056s	Scalar	vector dependence prevents vectorization				Divisions
[loop in [OpenMP worker at z_Linux_util.c:769]		0.000s	222.980s	Scalar					
[loop in __kmp_launch_thread at kmp_runtime ...		0.000s	222.980s	Scalar					

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

File: nbody.cc:27 MoveParticles

Line	Source
21	
22	`// Components of the gravity force on particle i`
23	`float Fx = 0, Fy = 0, Fz = 0;`
24	
25	`// Loop over particles that exert force: vectorization expected here`
26	`#pragma vector aligned`
27	`for (int i = 0; i < nParticles; i++) {`
28	
29	`// Avoid singularity and interaction with self`
30	`const float softening = 1e-20;`
31	
32	`// Newton's law of universal gravity`
33	`const float dx = particle.x[j] - particle.x[i];`
34	`const float dy = particle.y[j] - particle.y[i];`
35	`const float dz = particle.z[j] - particle.z[i];`
36	
37	`const float drSquared = dx*dx + dy*dy + dz*dz + softening;`
38	`const float drSqrt = sqrtf(drSquared);`
39	`const float drSqrtRecip = 1.0f/(drSqrt);`

Selected (Total Time):

Vectorized - Advisor

Where should I add vectorization and/or threading parallelism?

Elapsed time: 1.47s | Vectorized | Not Vectorized | FILTER: All Module | All Sources | Loop | All Threads

Summary | Survey Report | Refinement Reports | Annotation Report

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops: Vec...	Efficiency	Gai...	VL ...	Instruction Set Analysis: Traits
[loop in MoveParticles at nbody.cc:27]		32.896s	32.896s	Vectorized (B...		AVX2	~96%	7.71x	8	FMA; Square Roots
[loop in MoveParticles at nbody.cc:20]	1 Potential...	0.060s	32.956s	Threaded (Op...	inner loop was already vectorized					Extracts; FMA; Shu...
[loop in MoveParticles at nbody.cc:57]	2 Assume...	0.020s	0.020s	Threaded (Op...	vector dependence prevents vectorization					FMA
[loop in start_thread]		0.000s	31.616s	Scalar						
[loop in main at nbody.cc:101]	1 Assume...	0.000s	1.360s	Scalar	vector dependence prevents vectorization					Divisions
[loop in [OpenMP worker at z_Linux_u...		0.000s	31.616s	Scalar						
[loop in __kmp_launch_thread at kmp_...		0.000s	31.616s	Scalar						

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

File: nbody.cc:27 MoveParticles

```
17
18        // Loop over particles that experience force
19      #pragma omp parallel for
20      for (int i = 0; i < nParticles; i++) {
21
22          // Components of the gravity force on particle i
23          float Fx = 0, Fy = 0, Fz = 0;
24
25          // Loop over particles that exert force: vectorization expected here
26          #pragma vector aligned
27          for (int j = 0; j < nParticles; j++) {
28
29            // Avoid singularity and interaction with self
30            const float softening = 1e-20;
31
32            // Newton's law of universal gravity
33            const float dx = particle.x[j] - particle.x[i];
34            const float dy = particle.y[j] - particle.y[i];
35            const float dz = particle.z[j] - particle.z[i];
36
```

Selected (Total Time):

Vectorized - Advisor

Where should I add vectorization and/or threading parallelism?

INTEL ADVISOR XE 2016

Elapsed time: 1.47s | Vectorized | Not Vectorized | FILTER: All Module | All Sources | Loop | All Threads

Summary | Survey Report | Refinement Reports | Annotation Report

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Instruction Set Analysis			Advanced	Location
						Vec...	Efficiency	Gai...	VL ...	Traits	Data ...	Num.		
[loop in MoveParticles at nbody. ...		**32.896s**	**32.896s**	**Vectorized ...**		**AVX.**	**~96%**	**7.7..**	**8**	**FMA; Square Roots**	**Float...**	**14**		**nbody.cc:27**
[loop in MoveParticles at nbody.cc:20]	1 Potential..	0.060s	32.956s	Threaded (Op...	inner loop was already vectorized					Extracts; FMA; Shu...	Float3...	16		nbody.cc:20
[loop in MoveParticles at nbody.cc:57]	2 Assume ...	0.020s	0.020s (	Threaded (Op...	vector dependence prevents vectorization					FMA	Float32	4		nbody.cc:57
[loop in start_thread]		0.000s	31.616s	Scalar								0		
[loop in main at nbody.cc:101]	1 Assume ...	0.000s	1.360s (	Scalar	vector dependence prevents vectorization					Divisions	Float64	8		nbody.cc:101
[loop in [OpenMP worker at z_Linux_u ...		0.000s	31.616s	Scalar								0		z_Linux_util.c:769
[loop in __kmp_launch_thread at kmp_ ...		0.000s	31.616s	Scalar								0		kmp_runtime.c:5606

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

32.896s

Vectorized (Body) *Total time*

AVX; FMA 32.896s

Instruction Set *Self time*

- **Memory** 17% (4)
- **Compute** 74% (17)
- **Other** 9% (2)

Insruction Mix Summary

7.71x

Vectorization Gain

~96%

Vectorization Efficiency

Traits

FMA, Square Roots

Instruction Mix

Memory: **4** *Compute:* **17** *Other:* **2** *Number of Vector Registers:* **14**

Hands-on Exercise

- Goal: Identify hotspots in sample code
  - Targets for parallelization and/or optimization

- Test code has 4 functions: mm[1-4]
  - Each does a different version of matrix-matrix multiplication C=AxB
- Each function is called a different number of times
  - Approximate call frequency:
    - mm1: 10%
    - mm2: 20%
    - mm3: 30%
    - mm4: 40%
  - Where should we optimize?

Adroit Test Set Up

- Enable X11 forwarding
  - “ssh –Y -C <user>@adroit.princeton.edu
  - Will need local xserver (XQuartz for OSX, Xming for Windows)
- Clone the repo

https://github.com/cosden/CoDaS-HEP-Perf-Tuning

- Follow instructions in repo Readme.md

- What functions are most/least expensive?
- Change threshold values to select only the fastest function

Hands-on Discussion

- During a break feel free to try
  - GUI on head node
  - Other analyses on compute node
    - advanced-hotspots
    - general-exploration
  - Your code?

Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune

Hardware Counters

- Basic profilers "sample" the current program counter to see what instruction the cpu is currently executing.
- Then, after collecting enough samples, can give a good approximation as to what the program was doing.

- Performance Hardware Counters
  - Special purpose registers to track hardware events
- VTune can access hardware counters
  - Other profiling tools can access too, many build on top of open-source PAPI
- Literally hundreds of possible events to measure
  - Different in name and meaning for different chips

Hardware Counter Code Example

- 2D Matrix multiplication from before – mm1
  - 3 nested for loops
  - Let's assume we don't know why it is slow or even if it is slow

VTune Hotspots – Bottom-up View

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Ove... Time	Module	
[Loop at line 58 in main]	137.814s	0s	0s	mm1.icc.O0. ...	[Loop at line 58 in main]
[Loop at line 55 in main] ← [Loop at line	137.814s	0s	0s	mm1.icc.O0. ...	[Loop at line 55 in main]
[Loop at line 39 in ma	0.100s	0s	0s	mm1.icc.O0. ...	[Loop at line 39 in main]
[Loop at line 55 in main]	0.030s	0s	0s	mm1.icc.O0. ...	[Loop at line 55 in main]

Double click here

opens source below:

58	for (int k = 0; k < matrix_size; k++)	2.6%	0.0%	0.0%
59	{			
60	C[i][j] += A[i][k] * B[k][j];	97.3%	0.0%	0.0%
61	}			
62	}			
63	}			

Clearly there is one very expensive line inside nested for loops. *But why?*

General Exploration Analysis - VTune

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Event Count | Platform | matrix_multip...

Elapsed Time: 113.478s

Clockticks:	325,550,488,325
Instructions Retired:	369,072,553,608
CPI Rate:	0.882
MUX Reliability:	0.987
Front-End Bound:	**2.1%**
Bad Speculation:	**0.2%**
Back-End Bound:	**69.5%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more u
describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their r
uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the ov
issues.

Memory Bound:	**49.8%**

The metric value is high. This can indicate that the significant fraction of execution pipeline slots
Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory band

L1 Bound:	**0.075**

This metric shows how often machine was stalled without missing the L1 data cache. The L1
cases like loads blocked on older stores, a load might suffer a high latency even though it is
highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.503

A significant proportion of cycles is being spent handling first-level data TLB misses. As with ordinary data caching, focus on improving data locality and reducing working-set size to reduce DTLB overhead. Additionally, consider using profile-guided optimization (PGO) to collocate frequently-used data on the same page. Try using larger page sizes for large amounts of frequently-used data.

Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.005
L2 Bound:	0.016
L3 Bound:	**0.213**

This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.

Contested Accesses:	0.000
Data Sharing:	0.000
L3 Latency:	0.327

This metric shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited). Avoiding private cache misses (i.e. L2 misses/L3 hits) will improve the latency, reduce contention with sibling physical cores and increase performance. Note the value of this node may overlap with its siblings.

SQ Full:	0.000
DRAM Bound:	**0.105**

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth:	0.089
Memory Latency:	**0.654**

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Store Bound:	**0.000**
Core Bound:	**19.7%**

All flagged in pink:

Memory Bound: 49.8%
DRAM Bound: 0.105
Memory Latency: 0.654

Hardware Counters

- Remember every generation of chips has different counters
- VTune developers have calculated *derived metrics* that they feel best represent real problems
  - For example the sandybridge/ivybridge hardware counter: MEM_LOAD_UOPS_RETIRED.LLC_MISS_PS may not truly indicate if the cpu was stalled waiting for the data.
  - Colored in pink when they represent a value that *might* warrant investigation.

- My experience with cache misses:
  - Memory Bound, DRAM Bound, & Memory Latency together are very good indicators that cache is not being used well.

- How to fix cache misses?
  - In this case: Reorder nested loops (mm2)

After Loop Reodering

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Event Count | Platform | matrix_multip...

Elapsed Time: 2.602s

Clockticks:	7,446,011,169
Instructions Retired:	13,616,020,424
CPI Rate:	0.547
MUX Reliability:	0.981
Front-End Bound:	**8.9%**
Bad Speculation:	**0.3%**
Back-End Bound:	**42.2%**

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the
describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective e
uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded di
issues.

Memory Bound:	**24.4%**

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be st
Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth inforn

L1 Bound:	**0.051**

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typi
cases like loads blocked on older stores, a load might suffer a high latency even though it is being satis
highlighted due to DTLB Overhead or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.046
Loads Blocked by Store Forwarding:	0.000
Lock Latency:	0.000
Split Loads:	0.000
4K Aliasing:	0.120

A significant proportion of cycles is spent dealing with false 4k aliasing between loads and stores. Use the source/assembly view to identify the aliasing loads and stores, and then adjust your data layout so that the loads and stores no longer alias.

L2 Bound:	0.102

This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance.

L3 Bound:	**0.109**

This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.

Contested Accesses:	0.000
Data Sharing:	0.000
L3 Latency:	0.389

This metric shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited). Avoiding private cache misses (i.e. L2 misses/L3 hits) will improve the latency, reduce contention with sibling physical cores and increase performance. Note the value of this node may overlap with its siblings.

SQ Full:	0.333

This metric measures fraction of cycles where the Super Queue (SQ) was full taking into account all request-types and both hardware SMT threads. The Super Queue is used for requests to access the L2 cache or to go out to the Uncore.

DRAM Bound:	**0.032**
Memory Bandwidth:	0.011
Memory Latency:	**0.880**
Store Bound:	**0.000**
Core Bound:	**17.8%**
Retiring:	**48.7%**
Total Thread Count:	1
Paused Time:	0s

Change in metrics:

Memory Bound: 49.8% → 24.4%
DRAM Bound: 0.105 → 0.032
Memory Latency: 0.654 → 0.880

Elapsed Time: 132s → 2.6s

Results

- Reording nested loops resulted in a better use of cache and a 50x speedup
- Clues in the VTune metrics (hardware counters)
- Note that “optimized” case still showed MANY pink highlighted metrics
  - Something will always be limiting performance
  - Always requires interpretation and consideration

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Ove... Time	Module	
[Loop at line 59 in main]	2.499s	0s	0s	mm1a.icc.O3 ...	[Loop at line 59 in main]
[Loop at line 56 in main] ← [Loop at line	2.499s	0s	0s	mm1a.icc.O3 ...	[Loop at line 56 in main]
[Loop at line 39 in main]	0.080s	0s	0s	mm1a.icc.O3 ...	[Loop at line 39 in main]
[Loop at line 56 in main]	0.050s	0s	0s	mm1a.icc.O3 ...	[Loop at line 56 in main]
[Loop at line 30 in main]	0.010s	0s	0s	mm1a.icc.O3 ...	[Loop at line 30 in main]
[Loop at line 122 in main]	0.010s	0s	0s	mm1a.icc.O3 ...	[Loop at line 122 in main]

Final Remarks

- Measurement is key
- Tools are helpful
- Fast computing of wrong results is completely irrelevant!
  - Have correctness test(s)
  - Test after each modification
- Don’t fall into the “tuning trap”
  - Remember what really matters: **Total Runtime**