

Performance Bottleneck(s)

Jerome Vienne
viennej@tacc.utexas.edu

Texas Advanced Computing Center

Application Performance Snapshot

Take a quick look at your application's performance to see if it is well optimized for modern hardware.

- MPI parallelism (Linux* only)
- OpenMP* parallelism
- Memory access
- FPU Utilization
- I/O efficiency

Free !!!

It is easy to install, easy to run and provides results in a text or HTML report.

Website: <http://tinyurl.com/aps-knl>

Simple to use

- Part of Vtune or can be installed from the web
- You just have to call `aps` or `aps.sh`.
`aps.sh ./a.out`
`mpirun -np 6 aps ./a.out`
- It will generate a text file
- To generate the html you will have to recall it with specific flags (provided at the end of the initial run)
- There will be one exercise that will be provided during the lab

An Overview of Profiling

Outline

Introduction

Basic Tools

Command line timers

Code timers

gprof

Other Tools ..

Intermediate tools

Perfexpert

IPM

Advanced Tools

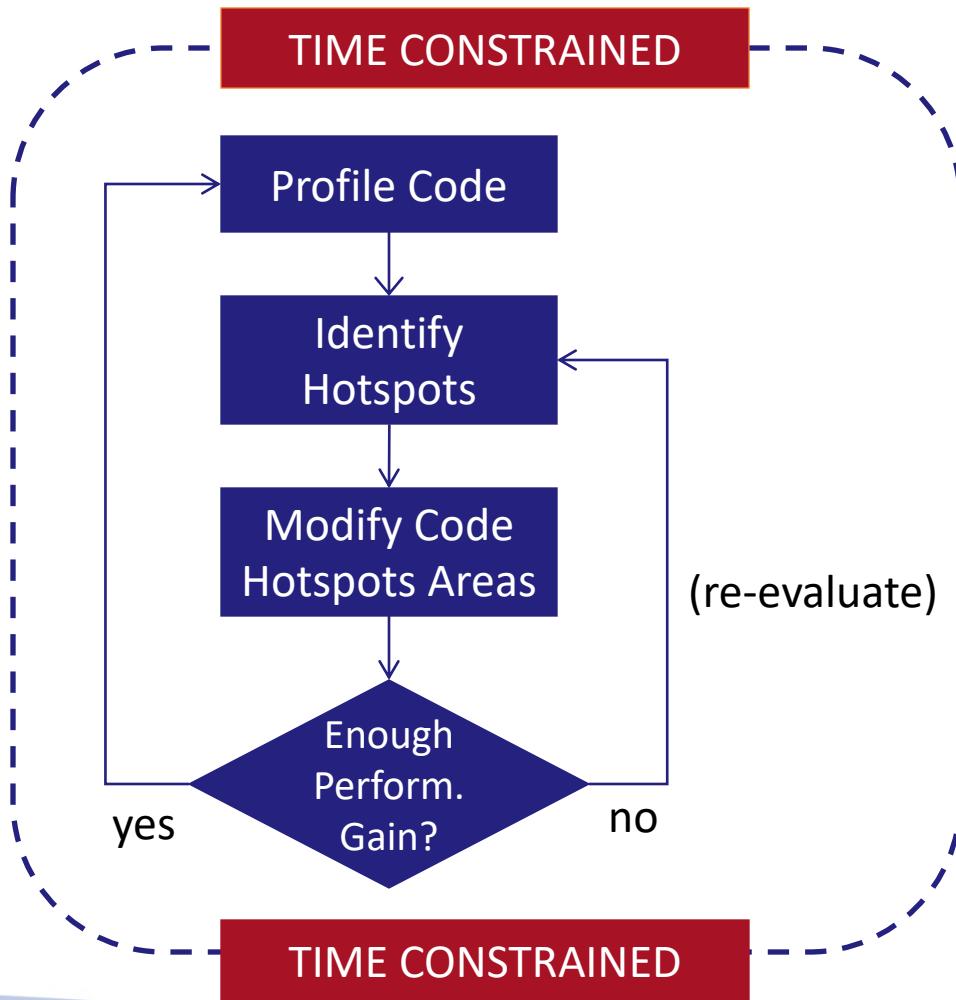
Vtune

Intel Advisor

HPCToolkit

Tau

Optimization Process



- Iterative process
- Application dependent
- Different levels
 - Compiler Options
 - Performance Libraries
 - Code Optimizations

Timers: Command Line

The command **time** is available in most Unix systems.

It is simple to use (no code instrumentation required).

Gives total execution time of a process and all its children in seconds.

```
% /usr/bin/time -p ./exeFile
```

```
real 9.95
```

```
user 9.86
```

```
sys 0.06
```

Leave out the **-p** option to get additional information:

```
% time ./exeFile
```

```
% 9.860u 0.060s 0:09.95 99.9%
```

Timers: Code Section

```
INTEGER :: rate, start, stop
REAL    :: time

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
CALL SYSTEM_CLOCK(COUNT = start)

! Code to time here

CALL SYSTEM_CLOCK(COUNT = stop)
time = REAL( ( stop - start )/ rate )
```

```
#include <time.h>

double start, stop, time;
start = (double)clock()/CLOCKS_PER_SEC;

/* Code to time here */

stop = (double)clock()/CLOCKS_PER_SEC;
time = stop - start;
```

About GPROF

GPROF is the **GNU** Project **PROFiler**.
gnu.org/software/binutils/

Requires recompilation of the code.

Compiler options and libraries provide wrappers for each routine call and periodic sampling of the program.

Provides three types of profiles

Flat profile

Call graph

Annotated source

Types of Profiles

Flat Profile

CPU time spent in each function (self and cumulative)

Number of times a function is called

Useful to identify most expensive routines

Call Graph

Number of times a function was called by other functions

Number of times a function called other functions

Useful to identify function relations

Suggests places where function calls could be eliminated

Annotated Source

Indicates number of times a line was executed

Profiling with gprof

Use the **-pg** flag during compilation:

```
% gcc -g -pg ./srcFile.c
```

```
% icc -g -pg ./srcFile.c
```

Run the executable. An output file **gmon.out** will be generated with the profiling information.

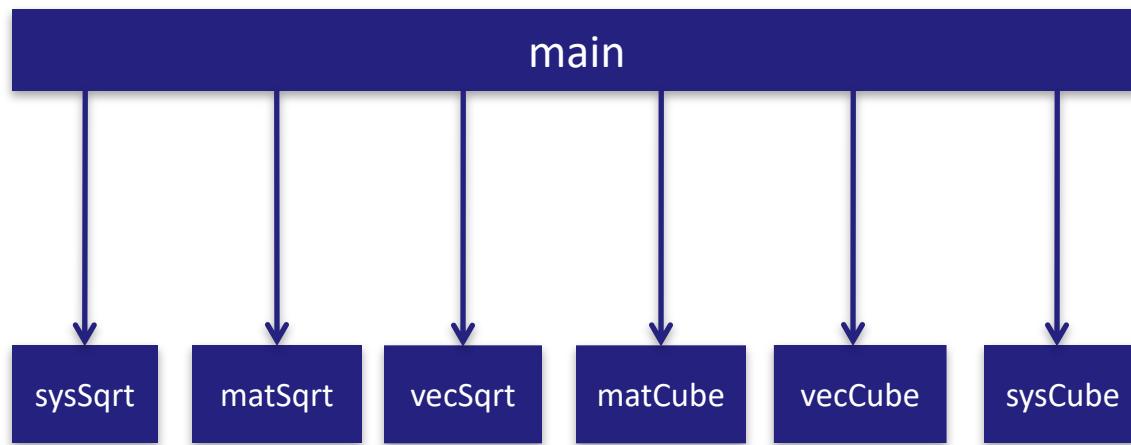
Execute **gprof** and redirect the output to a file:

```
% gprof ./exeFile gmon.out > profile.txt
```

```
% gprof -l ./exeFile gmon.out > profile_line.txt
```

```
% gprof -A ./exeFile gmon.out > profile_anotated.txt
```

Visual Call Graph



Flat profile

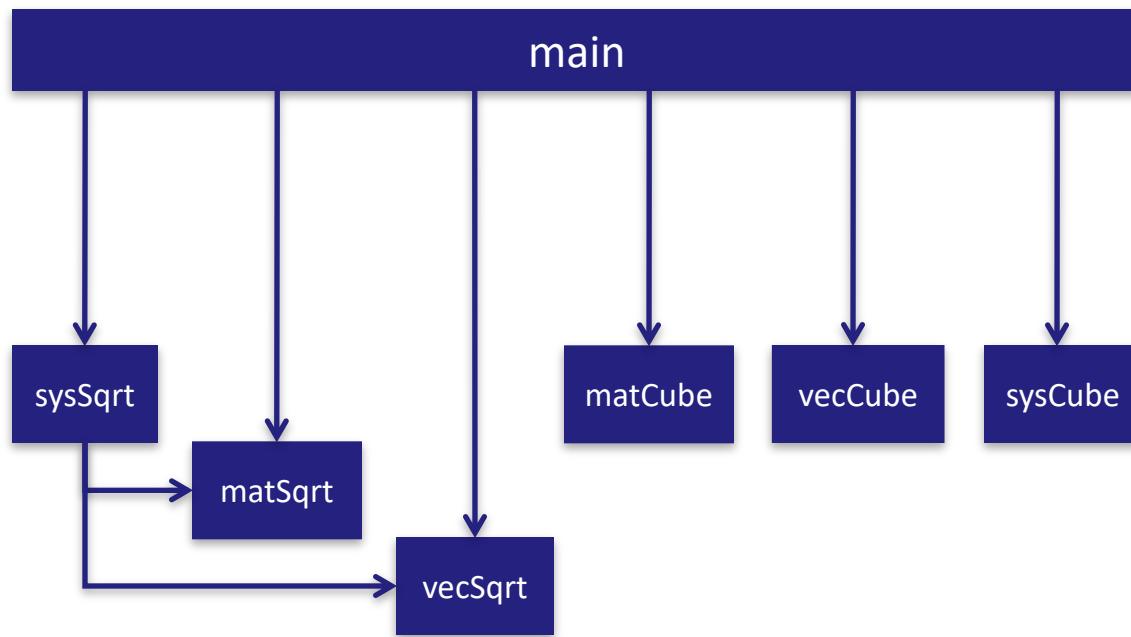
In the flat profile we can identify the most expensive parts of the code (in this case, the calls to matSqrt, matCube, and sysCube).

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
50.00	2.47	2.47	2	1.24	1.24	matSqrt
24.70	3.69	1.22	1	1.22	1.22	matCube
24.70	4.91	1.22	1	1.22	1.22	sysCube
0.61	4.94	0.03	1	0.03	4.94	main
0.00	4.94	0.00	2	0.00	0.00	vecSqrt
0.00	4.94	0.00	1	0.00	1.24	sysSqrt
0.00	4.94	0.00	1	0.00	0.00	vecCube

Call Graph Profile

	index	% time	self	children	called		name
			0.00	0.00	1/1		<hicore> (8)
[1]		100.0	0.03	4.91		1	main [1]
			0.00	1.24	1/1		sysSqrt [3]
			1.24	0.00	1/2		matSqrt [2]
			1.22	0.00	1/1		sysCube [5]
			1.22	0.00	1/1		matCube [4]
			0.00	0.00	1/2		vecSqrt [6]
			0.00	0.00	1/1		vecCube [7]
<hr/>							
[2]			1.24	0.00	1/2		main [1]
			1.24	0.00	1/2		sysSqrt [3]
		50.0	2.47	0.00		2	matSqrt [2]
<hr/>							
[3]			0.00	1.24	1/1		main [1]
		25.0	0.00	1.24		1	sysSqrt [3]
			1.24	0.00	1/2		matSqrt [2]
<hr/>							
			0.00	0.00	1/2	2	vecSqrt [6]

Visual Call Graph



PERFEXPERT



Perfexpert

A new tool, locally developed at UT

Easy to use and understand

Great for quick profiling and for beginners

Provides recommendation on “what to fix” in a subroutine

Collects information from PAPI using HPCToolkit

Combines ease of use with useful interpretation of gathered performance data

Optimization suggestions!!!

Profiling with Perfexpert: Compilation

Compile the code with full optimization and with the -g flag:

```
mpicc -g -O3 source.c
```

```
mpif90 -g -O3 source.f90
```

In your job submission script:

```
perfexpert 0.1 ./<executable> <executable args>
```

Perfexpert Output

```
Loop in function collision in collision.F90:68 (37.66% of the total runtime)
=====
ratio to total instrns      %  0.....25.....50.....75.....100
  - floating point          100.0 *****
  - data accesses           31.7 *****
* GFLOPS (% max)          19.8 *****
  - packed                  7.3 ****
  - scalar                  12.5 *****

-----
performance assessment   LCPI good.....okay.....fair.....poor.....bad
* overall                  4.51 >>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* data accesses             12.28 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
  - L1d hits                1.11 >>>>>>>>>>>>>>>>
  - L2d hits                1.05 >>>>>>>>>>>>>>
  - L3d hits                0.01
  - LLC misses              10.11 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses    0.13 >>
  - L1i hits                0.00
  - L2i hits                0.01
  - L2i misses               0.12 >>
* data TLB                 0.02
* instruction TLB          0.00
* branch instructions       0.00
  - correctly predicted     0.00
  - mispredicted             0.00
* floating-point instr     8.01 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
  - slow FP instr            0.16 >>
  - fast FP instr            7.85 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
=====
```

Perfexpert Suggestions

```
#-----  
# Recommendations for collision.F90:68  
#-----  
#  
# Here is a possible recommendation for this code segment  
#  
Description: compute values rather than loading them if doable with few operations  
Reason: this optimization replaces (slow) memory accesses with equivalent but faster computations  
Code example:
```

```
loop i {  
    t[i] = a[i] * 0.5;  
}  
loop j {  
    a[j] = c[j] - t[j];  
}  
=====>  
loop i {  
    a[i] = c[i] - (a[i] * 0.5);  
}
```

Advanced Perfexpert Use

If you select the -m or -s options, PerfExpert will try to

Automatically optimize your code

Show the performance analysis report

Show the list of suggestion for bottleneck remediation when no automatic optimization is possible.

Get the code and documentation from:

www.tacc.utexas.edu/research-development/tacc-projects/perfexpert

github.com/TACC/perfexpert

IPM: INTEGRATED PERFORMANCE MONITORING

IPM: Integrated Performance Monitoring

“IPM is a portable profiling infrastructure for parallel codes. It provides a low-overhead performance summary of the computation and communication in a parallel program”

IPM is a **quick, easy and concise** profiling tool

The level of detail it reports is smaller than TAU, PAPI or HPCToolkit

IPM: Integrated Performance Monitoring

IPM features:

- easy to use
- has low overhead
- is scalable

Requires **no source code modification**, just adding the “-g” option to the compilation

Produces XML output that is parsed by scripts to generate browser-readable html pages

IPM: Integrated Performance Monitoring

Define the type of collection:

```
export IPM_REPORT=full (full, terse, none)
```

Define collection threshold (optional)

```
export IPM_MPI_THRESHOLD=0.3
```

(Reports only routines using this percentage or more of MPI time)

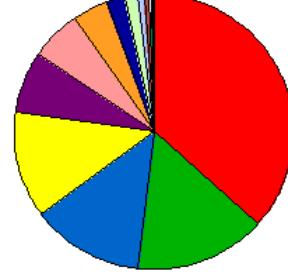
Generating HTML output:

```
ipm_parse -html <collected data dir>
```

IPM: Text Output

```
##IPMv0.983#####
#
# command : /home1/01157/carlos/TEST/profiling/mm (completed)
# host    : c404-504/x86_64_Linux          mpi_tasks : 16 on 1 nodes
# start   : 11/11/14/16:25:04           wallclock : 0.836220 sec
# stop    : 11/11/14/16:25:05           %comm      : 20.49
# gbytes  : 5.25372e+00 total          gflop/sec : NA
#
#####
# region  : *      [ntasks] =      16
#
#
#          [total]      <avg>       min       max
# entries          16            1           1           1
# wallclock        13.3793      0.836207    0.836197    0.83622
# user             18.5732      1.16082     1.05884     1.18082
# system           1.48777     0.0929854    0.074988    0.197969
# mpi              2.74098     0.171311     0.133593    0.650309
# %comm            20.4864      0.328358    0.328205    0.330112
# gbytes           5.25372     0.328358
#
#          [time]      [calls]    <%mpi>    <%wall>
# MPI_Bcast         1.6727      32000      61.03      12.50
# MPI_Recv          0.725848     4015       26.48       5.43
# MPI_Send          0.320127     4015       11.68       2.39
# MPI_Barrier       0.0223054     16         0.81       0.17
# MPI_Comm_rank     2.97837e-06     16         0.00       0.00
# MPI_Comm_size     1.43796e-06     16         0.00       0.00
#####
#
```

IPM: Integrated Performance Monitoring

<p>1469389</p> <ul style="list-style-type: none"> • Load Balance • Communication Balance • Message Buffer Sizes • Communication Topology • Switch Traffic • Memory Usage • Executable Info • Host List • Environment • Developer Info <p>Powered by </p>	<p>command: /work/01125/yye00/IPM/Benchmark/Defiant.exe perturb -runperturbed -da_grid_x 50 -da_grid_y 50 -da_grid_z 50 -seed_phi 3454345 -seed_k11 56756756 -seed_k22 235759 -seed_k33 234656 -seed_flowmask 3222111 -percentage_phi 0.1 -percentage_k11 0.1 -percentage_k22 0.1 -percentage_k33 0.1 -percentage_flowmask 0.15 -endtime 1.0 -ksp_type bicg -pc_type bjacobi</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">codename:</td> <td>unknown</td> <td>state:</td> <td>running</td> </tr> <tr> <td>username:</td> <td>yye00</td> <td>group:</td> <td>G-801077</td> </tr> <tr> <td>host:</td> <td>i115-108 (x86_64_Linux)</td> <td>mpi_tasks:</td> <td>64 on 4 hosts</td> </tr> <tr> <td>start:</td> <td>07/13/10/00:28:10</td> <td>wallclock:</td> <td>3.80580e+00 sec</td> </tr> <tr> <td>stop:</td> <td>07/13/10/00:28:13</td> <td>%comm:</td> <td>11.5752798360397</td> </tr> <tr> <td>total memory:</td> <td>10.85705 gbytes</td> <td>total gflop/sec:</td> <td>1.02192900310053</td> </tr> <tr> <td>switch(send):</td> <td>0 gbytes</td> <td>switch(recv):</td> <td>0 gbytes</td> </tr> </table>	codename:	unknown	state:	running	username:	yye00	group:	G-801077	host:	i115-108 (x86_64_Linux)	mpi_tasks:	64 on 4 hosts	start:	07/13/10/00:28:10	wallclock:	3.80580e+00 sec	stop:	07/13/10/00:28:13	%comm:	11.5752798360397	total memory:	10.85705 gbytes	total gflop/sec:	1.02192900310053	switch(send):	0 gbytes	switch(recv):	0 gbytes																	
codename:	unknown	state:	running																																											
username:	yye00	group:	G-801077																																											
host:	i115-108 (x86_64_Linux)	mpi_tasks:	64 on 4 hosts																																											
start:	07/13/10/00:28:10	wallclock:	3.80580e+00 sec																																											
stop:	07/13/10/00:28:13	%comm:	11.5752798360397																																											
total memory:	10.85705 gbytes	total gflop/sec:	1.02192900310053																																											
switch(send):	0 gbytes	switch(recv):	0 gbytes																																											
Computation																																														
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 30%;">Event</th> <th style="width: 30%;">Count</th> <th style="width: 40%;">Pop</th> </tr> </thead> <tbody> <tr> <td>PAPI_FP_OPS</td> <td>3889256866</td> <td>*</td> </tr> <tr> <td>PAPI_TOT_CYC</td> <td>93055641837</td> <td>*</td> </tr> <tr> <td>PAPI_TOT_INS</td> <td>82058705179</td> <td>*</td> </tr> <tr> <td>PAPI_VEC_INS</td> <td>8293137711</td> <td>*</td> </tr> </tbody> </table>	Event	Count	Pop	PAPI_FP_OPS	3889256866	*	PAPI_TOT_CYC	93055641837	*	PAPI_TOT_INS	82058705179	*	PAPI_VEC_INS	8293137711	*	<p>Communication</p>  <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2">% of MPI Time</th> </tr> </thead> <tbody> <tr> <td style="color: red;">MPI_Allreduce</td> <td>~45%</td> </tr> <tr> <td style="color: green;">MPI_Bcast</td> <td>~15%</td> </tr> <tr> <td style="color: blue;">MPI_Waitany</td> <td>~10%</td> </tr> <tr> <td style="color: yellow;">MPI_Waitall</td> <td>~10%</td> </tr> <tr> <td style="color: purple;">MPI_Isend</td> <td>~5%</td> </tr> <tr> <td style="color: pink;">MPI_Send</td> <td>~5%</td> </tr> <tr> <td style="color: orange;">MPI_Barrier</td> <td>~3%</td> </tr> <tr> <td style="color: darkblue;">MPI_Comm_rank</td> <td>~2%</td> </tr> <tr> <td style="color: lightblue;">MPI_Start</td> <td>~2%</td> </tr> <tr> <td style="color: lightgray;">MPI_Allgather</td> <td>~2%</td> </tr> <tr> <td style="color: maroon;">MPI_Scan</td> <td>~1%</td> </tr> <tr> <td style="color: darkgreen;">MPI_Recv</td> <td>~1%</td> </tr> <tr> <td style="color: magenta;">MPI_Startall</td> <td>~1%</td> </tr> <tr> <td style="color: darkblue;">MPI_Comm_size</td> <td>~1%</td> </tr> </tbody> </table>	% of MPI Time		MPI_Allreduce	~45%	MPI_Bcast	~15%	MPI_Waitany	~10%	MPI_Waitall	~10%	MPI_Isend	~5%	MPI_Send	~5%	MPI_Barrier	~3%	MPI_Comm_rank	~2%	MPI_Start	~2%	MPI_Allgather	~2%	MPI_Scan	~1%	MPI_Recv	~1%	MPI_Startall	~1%	MPI_Comm_size	~1%
Event	Count	Pop																																												
PAPI_FP_OPS	3889256866	*																																												
PAPI_TOT_CYC	93055641837	*																																												
PAPI_TOT_INS	82058705179	*																																												
PAPI_VEC_INS	8293137711	*																																												
% of MPI Time																																														
MPI_Allreduce	~45%																																													
MPI_Bcast	~15%																																													
MPI_Waitany	~10%																																													
MPI_Waitall	~10%																																													
MPI_Isend	~5%																																													
MPI_Send	~5%																																													
MPI_Barrier	~3%																																													
MPI_Comm_rank	~2%																																													
MPI_Start	~2%																																													
MPI_Allgather	~2%																																													
MPI_Scan	~1%																																													
MPI_Recv	~1%																																													
MPI_Startall	~1%																																													
MPI_Comm_size	~1%																																													
HPM Counter Statistics																																														
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 30%;">Event</th> <th style="width: 30%;">Ntasks</th> <th style="width: 30%;">Avg</th> </tr> </thead> <tbody> <tr> <td>PAPI_FP_OPS</td> <td>*</td> <td>60769638.53</td> </tr> <tr> <td>PAPI_TOT_CYC</td> <td>*</td> <td>1453994403.70</td> </tr> <tr> <td>PAPI_TOT_INS</td> <td>*</td> <td>1282167268.42</td> </tr> <tr> <td>PAPI_VEC_INS</td> <td>*</td> <td>129580276.73</td> </tr> </tbody> </table>	Event	Ntasks	Avg	PAPI_FP_OPS	*	60769638.53	PAPI_TOT_CYC	*	1453994403.70	PAPI_TOT_INS	*	1282167268.42	PAPI_VEC_INS	*	129580276.73	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 30%;">Min(rank)</th> <th style="width: 30%;">Max(rank)</th> </tr> </thead> <tbody> <tr> <td>53254674 (63)</td> <td>68822066 (21)</td> </tr> <tr> <td>1346848050 (23)</td> <td>1646491906 (12)</td> </tr> <tr> <td>1134309464 (0)</td> <td>1477795580 (12)</td> </tr> <tr> <td>113002002 (63)</td> <td>146915653 (21)</td> </tr> </tbody> </table>	Min(rank)	Max(rank)	53254674 (63)	68822066 (21)	1346848050 (23)	1646491906 (12)	1134309464 (0)	1477795580 (12)	113002002 (63)	146915653 (21)																				
Event	Ntasks	Avg																																												
PAPI_FP_OPS	*	60769638.53																																												
PAPI_TOT_CYC	*	1453994403.70																																												
PAPI_TOT_INS	*	1282167268.42																																												
PAPI_VEC_INS	*	129580276.73																																												
Min(rank)	Max(rank)																																													
53254674 (63)	68822066 (21)																																													
1346848050 (23)	1646491906 (12)																																													
1134309464 (0)	1477795580 (12)																																													
113002002 (63)	146915653 (21)																																													

IPM: Event Statistics

Communication Event Statistics (100.00% detail, 9.9012e-06 error)							
	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Allreduce		8	79680	4.178	8.225e-06	8.882e-04	14.82
MPI_Bcast		4	1024	4.047	5.914e-08	6.413e-02	14.35
MPI_Allreduce	512	39936	3.803	1.660e-05	1.170e-01	13.49	1.56
MPI_Allreduce	4	25472	2.250	6.012e-07	1.552e-02	7.98	0.92
MPI_Barrier	0	64	1.176	1.814e-02	1.865e-02	4.17	0.48
MPI_Isend	8	630	1.028	3.427e-07	1.647e-02	3.65	0.42
MPI_Isend	4	4556	0.943	2.738e-07	1.833e-02	3.34	0.39
MPI_Send	14976	144	0.722	1.030e-03	7.308e-03	2.56	0.30
MPI_Comm_rank	0	106948	0.620	3.725e-08	9.872e-03	2.20	0.25
MPI_Waitany	0	6093	0.542	4.615e-07	1.358e-02	1.92	0.22
MPI_Waitany	1248	27462	0.519	5.183e-07	2.723e-04	1.84	0.21
MPI_Send	16224	144	0.517	4.283e-04	7.129e-03	1.83	0.21
MPI_Waitany	1352	20370	0.496	5.197e-07	5.783e-03	1.76	0.20
MPI_Start	0	269196	0.396	3.623e-07	3.685e-05	1.40	0.16
MPI_Send	13824	48	0.298	4.035e-03	7.227e-03	1.06	0.12
MPI_Waitany	1152	10980	0.243	5.383e-07	2.310e-04	0.86	0.10
MPI_Bcast	216	576	0.231	2.302e-06	5.843e-03	0.82	0.09
MPI_Allgather	4	9088	0.215	5.118e-07	1.793e-03	0.76	0.09
MPI_Waitall	184	11	0.210	1.633e-02	2.135e-02	0.74	0.09
MPI_Scan	4	384	0.144	2.259e-05	1.600e-03	0.51	0.06
MPI_Waitany	147	453	0.141	4.866e-07	1.406e-02	0.50	0.06
MPI_Waitany	4	448	0.132	4.345e-07	5.805e-03	0.47	0.05
MPI_Waitall	320	18	0.120	3.002e-06	1.284e-02	0.42	0.05
MPI_Send	17576	42	0.108	6.682e-05	6.406e-03	0.38	0.04
MPI_Waitall	72	6	0.103	1.547e-02	2.038e-02	0.36	0.04
MPI_Waitall	96	38	0.091	2.882e-06	1.563e-02	0.32	0.04
MPI_Waitany	624	140	0.088	1.126e-06	7.880e-03	0.31	0.04
MPI_Recv	8	9	0.085	8.373e-07	7.696e-02	0.30	0.03

IPM: Integrated Performance Monitoring

When to use IPM?

To quickly find out **where your code is spending most of its time** (in both computation and communication)

For performing **scaling studies** (both strong and weak)

When you suspect you have **load imbalance** and want to verify it quickly

For a quick look at the **communication pattern**

To find out how much **memory** you are using **per task**

To find the **relative communication & compute time**

IPM: Integrated Performance Monitoring

When IPM is **NOT** the answer

When you already know where the performance issues are

When you need detailed performance information on exact lines of code

When want to find specific information such as cache misses

Advanced Profiling Tools

: the next level

Advanced Profiling Tools

Can be intimidating:

- Difficult to install
- Many dependences
- Require kernel patches

} Not your problem

Useful for serial and parallel programs

Extensive profiling and scalability information

Analyze code using:

- Timers
- Hardware registers (PAPI)
- Function wrappers

PAPI

PAPI is a **P**erformance **A**pplication **P**rogramming **I**nterface

icl.cs.utk.edu/papi

API to use hardware counters

Behind Tau, HPCToolkit

Multiplatform:

Most Intel & AMD chips

IBM POWER 4/5/6

Cray X/XD/XT

Sun UltraSparc I/II/III

MIPS

SiCortex

Cell

Available as a module on Lonestar and Stampede

HPCTOOLKIT

HPCToolkit

hpctoolkit.org

Binary-level measurement and analysis – no recompilation!

Uses sample-based measurements

Controlled overhead

Good scalability for large scale parallelism analysis

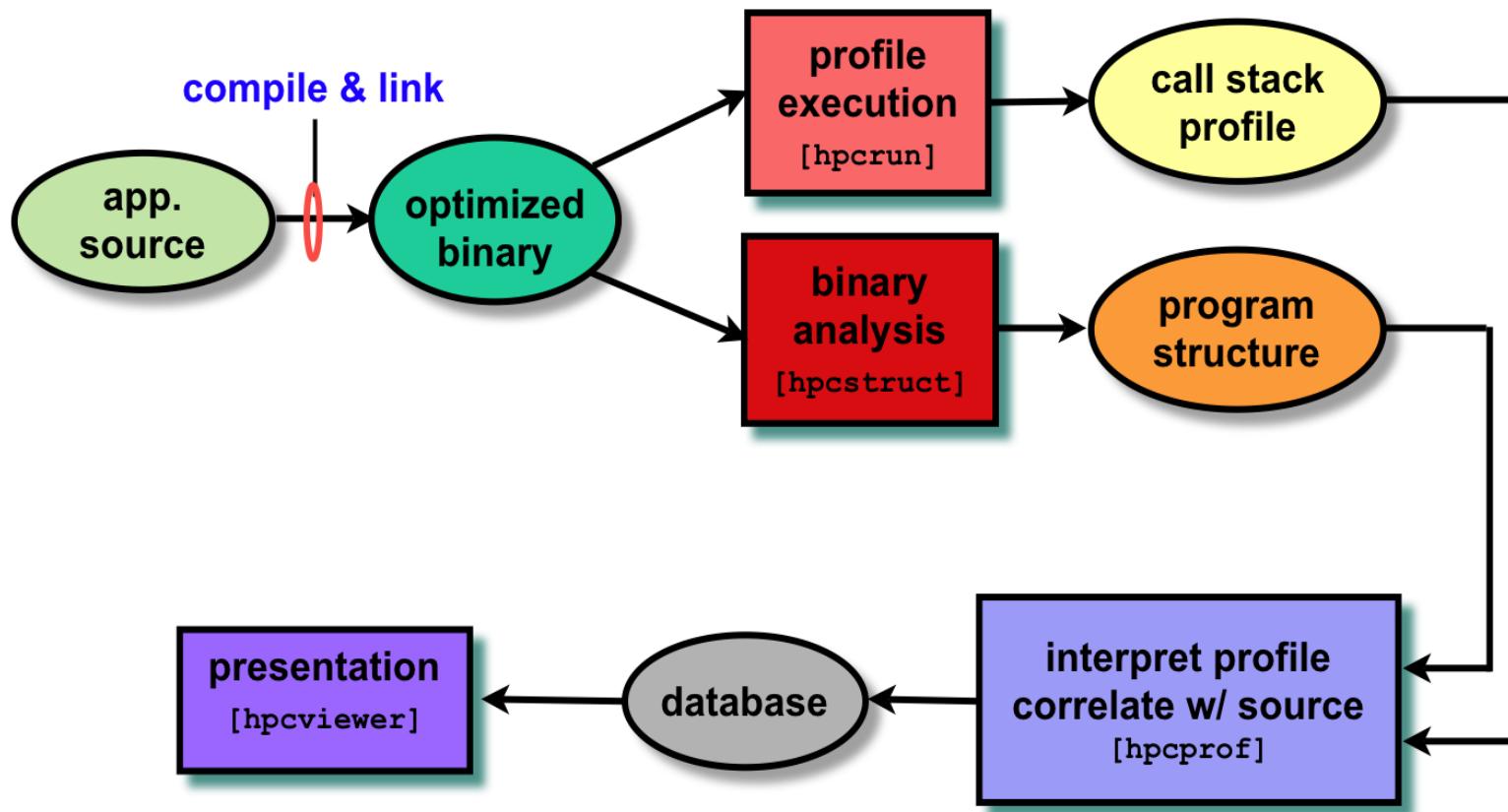
Good analysis tools

Ability to compare multiple runs at different scales

Scalable interface for trace viewer

Good derived metrics support

HPCToolkit: Overview



HPCToolkit: Overview

Compile with symbols and optimization:

```
icc -g -O3 mysource -o myexe
```

Collect profiling information:

```
(Serial / OMP) hpcrun myexe
```

```
(MPI parallel) ibrun hpcrun myexe
```

Perform static binary analysis of the executable:

```
hpcstruct myexe
```

Interpret profile and correlate it with source code:

```
hpcprof -S myexe.hpcstruct hpctoolkit-myexe-measurement-dir
```

View the results

```
hpcviewer hpctoolkit-database
```

HPCToolkit: Looking at Measurements

The screenshot shows the HPCToolkit interface. At the top is a code editor window titled "matmult.c" with the following code:

```
110 //! print out one element of the answer
111 printf("%d,%d) = %f\n", matsize,matsize,c(matsize,matsize) );
112 } // end if ( myid == master )
113 else{
114 //! workers receive B, then compute rows of C until done message
115 for(i=1;i<=matsize;i++){
116 ierr=MPI_Bcast(&b(1,i), matsize, MPI_DOUBLE, master, MPI_COMM_WORLD);
117 }
118 flag = 1;
119 while (flag != 0){
120 ierr = MPI_Recv(buffer, matsize, MPI_DOUBLE, master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
121 row = status.MPI_TAG;
122 flag = row;
123 if (flag != 0){
124 // multiply the matrices here using C(i,j) += sum (A(i,k)* B(k,j))
125 multiply_matrices(answer, buffer, b, matsize);
126 ierr = MPI_Send(answer, matsize, MPI_DOUBLE, master, row, MPI_COMM_WORLD);
127 }
128 }
```

Below the code editor is a performance analysis table titled "Calling Context View". The table has columns for Scope, PAPI_TOT_CYC:Sum (I), PAPI_TOT_CYC:Sum (E), PAPI_L2_TCM:Sum (I), PAPI_L2_TCM:Sum (E), PAPI_L2_DCA:Sum (I), and PAPI_L2_DCA:Sum (E). The table shows data for various function calls, with the first few rows being:

Scope	PAPI_TOT_CYC:Sum (I)	PAPI_TOT_CYC:Sum (E)	PAPI_L2_TCM:Sum (I)	PAPI_L2_TCM:Sum (E)	PAPI_L2_DCA:Sum (I)	PAPI_L2_DCA:Sum (E)
Experiment Aggregate	4.05e+10 100 %	4.05e+10 100 %	7.56e+08 100 %	7.56e+08 100 %	1.17e+09 100 %	1.17e+09 100 %
main	4.05e+10 100 %	3.20e+10 79.0%	7.56e+08 100 %	7.52e+08 99.5%	1.17e+09 100 %	1.05e+09 89.1%
loop at matmult.c: 1	3.27e+10 80.5%		7.18e+08 95.0%		1.04e+09 89.4%	

A red arrow points from the toolbar in the bottom right of the main window to the toolbar in the bottom right of the Calling Context View window.



Toolbar with useful
shortcuts:

- Hot Path
- Derived Metric
- Hide/Show columns
- Export data to CSV
- Change font size

TAU

Tau

TAU is a suite of **T**uning and **A**nalysis **U**tilities

www.cs.uoregon.edu/research/tau

10+ year project involving

University of Oregon Performance Research Lab

LANL Advanced Computing Laboratory

Research Centre Julich at ZAM, Germany

Integrated toolkit

Performance instrumentation

Measurement

Analysis

Visualization

Tau: Measurements

- Parallel profiling
 - Function-level, block (loop)-level, statement-level
 - Supports user-defined events
 - TAU parallel profile data stored during execution
 - Hardware counter values (multiple counters)
 - Support for callgraph and callpath profiling
- Tracing
 - All profile-level events
 - Inter-process communication events
 - Trace merging and format conversion

Tau: Results

You can also enable tracing by using:

```
export TAU_TRACE=1
```

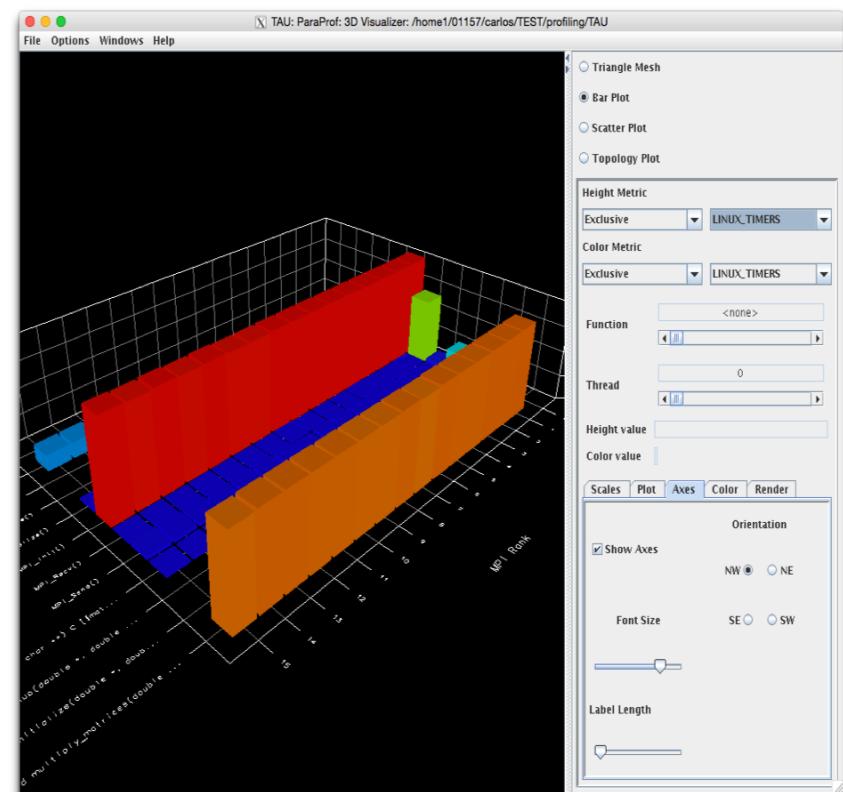
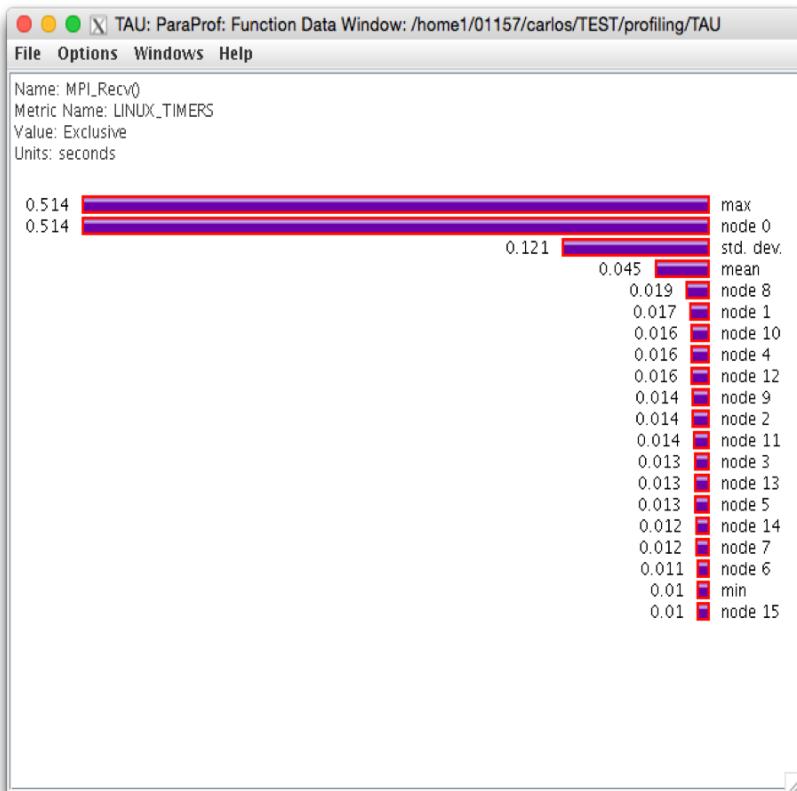
Run code normally:

```
ibrun myexe
```

Then look at the results using paraprof:

```
paraprof /path/to/results
```

Tau: Function Data Window and 3D Plots



VTUNE AMPLIFIER

VTune

This is an Intel product

software.intel.com/en-us/intel-vtune-amplifier-xe

Available on stampede: `module load vtune`

Supports serial, threaded, MPI, and hybrid applications

Two part interface: command line and gui

- Sort, filter, and visualize results on the timeline and on your source

Helps to correlate timing information with original source code and assembly

Much improved support for KNL, including memory access details

- Memory bandwidth to DDR4 and MCDRAM (Flat or Cache)
- Fine grained control of memory allocation using libmemkind

Why VTune

Insight into performance, threading, bandwidth, caching and more

Analysis is simple and convenient (Good GUI)

- Sort, filter, and visualize results on the timeline and on your source

Particularly useful to look at memory access on KNL

- Memory bandwidth to DDR4 and MCDRAM (Flat or Cache)

Also useful for fine grained control of memory allocation using libmemkind

VTune Basics

VTune is easy to use by employing pre-defined **collections**

Collections have **knobs** that allow you to capture additional information

To find out which collections are available :

```
$ ampxe-cl -help collect
```

To find out which knobs are available for a collection :

```
$ ampxe-cl -help collect memory-access
```

Pre-defined Collections

Collection	Information Provided
hotspots	Identify most time-consuming code sections (user mode)
advanced-hotspots	Adds CPI, higher frequency low overhead sampling
concurrency	CPU utilization, threading synchronization overhead (user mode)
disk-io	Disk IO preview, not working in Stampede (requires root)
memory-access	Memory access details and memory bandwidth utilization. May include specific arrays that are heavy BW users (useful for fine-grained MCDRAM use)
hpc-performance	Performance characterization, including floating point unit and memory bandwidth utilization

Useful knobs & options

Option	Description
-data-limit=0	Override default maximum data collection size
-no-summary	Do not produce text summary
-no-auto-finalize	Do not finalize data analysis after collection
-finalize	Carry out data analysis after collection
-start-paused	Start application without profiling
-resume-after=X	Resume profiling after X seconds
-duration=Y	Profile only for Y seconds

Knob	Description
analyze-memory-objects=true	Determine arrays using most memory bandwidth (highest L2 miss rates)
analyze-openmp=true	Determine inefficiencies in OpenMP regions

VTune: Collecting Performance Data

Result finalization and viewing on KNL target might be slow. Use the recommended workflow:

Compile with debug symbols

```
$ mpicc -g -O3 -xMIC-AVX512 -qopenmp ./code.c
```

Run collection on KNL deferring finalization on host

```
$ amplxe-cl -c hotspots -no-auto-finalize -r <result_dir> ./app
```

Finalize the result on the host

```
$ amplxe-cl -finalize -r <result_dir> -search-dir <bin_dir>
```

Generate reports, work with GUI

```
$ amplxe-cl -report hotspots -r <result_dir>
```

```
$ amplxe-gui
```

VTune in a batch Job

```
#!/bin/bash
#SBATCH -J knl-run.%j.err      # job name
#SBATCH -o knl-run.%j.out      # out/err file name
#SBATCH -N 1                    # Number of nodes
#SBATCH -n 1                    # MPI Tasks
#SBATCH -p normal               # queue (partition)
#SBATCH -t 00:30:00              # runtime (hh:mm:ss)
#SBATCH -A <Allocation>
```

```
export OMP_NUM_THREADS=68
# Launch amplxe-cl with proper options
amplxe-cl -c hotspots -no-auto-finalize -- ./a.out
```

VTune: Sample Summary

Collection and Platform Info

```
-----  
Parameter          hotspots.0  
-----  
Application Command Line  ./mm  
Operating System      2.6.32-431.17.1.el6.x86_64 CentOS release 6.5 (Final)  
MPI Process Rank     0  
Computer Name        c401-604.stampede.tacc.utexas.edu  
Result Size          2012293  
Collection start time 04:01:56 12/11/2014 UTC  
Collection stop time  04:01:58 12/11/2014 UTC
```

CPU

```
-----  
Parameter          hotspots.0  
-----  
Name              Intel(R) Xeon(R) E5 processor  
Frequency         2699995693  
Logical CPU Count 16
```

Summary

```
-----  
Elapsed Time:      1.160  
CPU Time:          0.860  
Average CPU Usage: 0.714
```

Example hotspots summary

The screenshot shows the Intel VTune Amplifier XE 2016 interface with the title bar "`r00hs` - Intel VTune Amplifier". The main window displays a "Basic Hotspots" report for the "Hotspots by CPU Usage viewpoint (change)".

Elapsed Time: 19.160s

- CPU Time:** 1767.714s
- Effective Time:** 1495.563s
- Spin Time:** 267.933s

A significant portion of CPU time is spent waiting. Use this metric to discover which synchronizations are spinning. Consider adjusting spin wait parameters, changing the lock implementation (for example, by backing off then descheduling), or adjusting the synchronization granularity.

Imbalance or Serial Spinning (OpenMP): 251.729s

CPU time spent waiting on an OpenMP barrier inside of a parallel region can be a result of load imbalance. Where relevant, try dynamic work scheduling to reduce the imbalance. High metric value on serial execution (Serial - outside any region) may signal significant serial application time that is limiting efficient processor utilization. Explore options for parallelization, algorithm or microarchitecture tuning of the serial part of the application.

Lock Contention (OpenMP): 0.010s

Other: 16.194s

Overhead Time: 4.218s

Total Thread Count: 137

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.

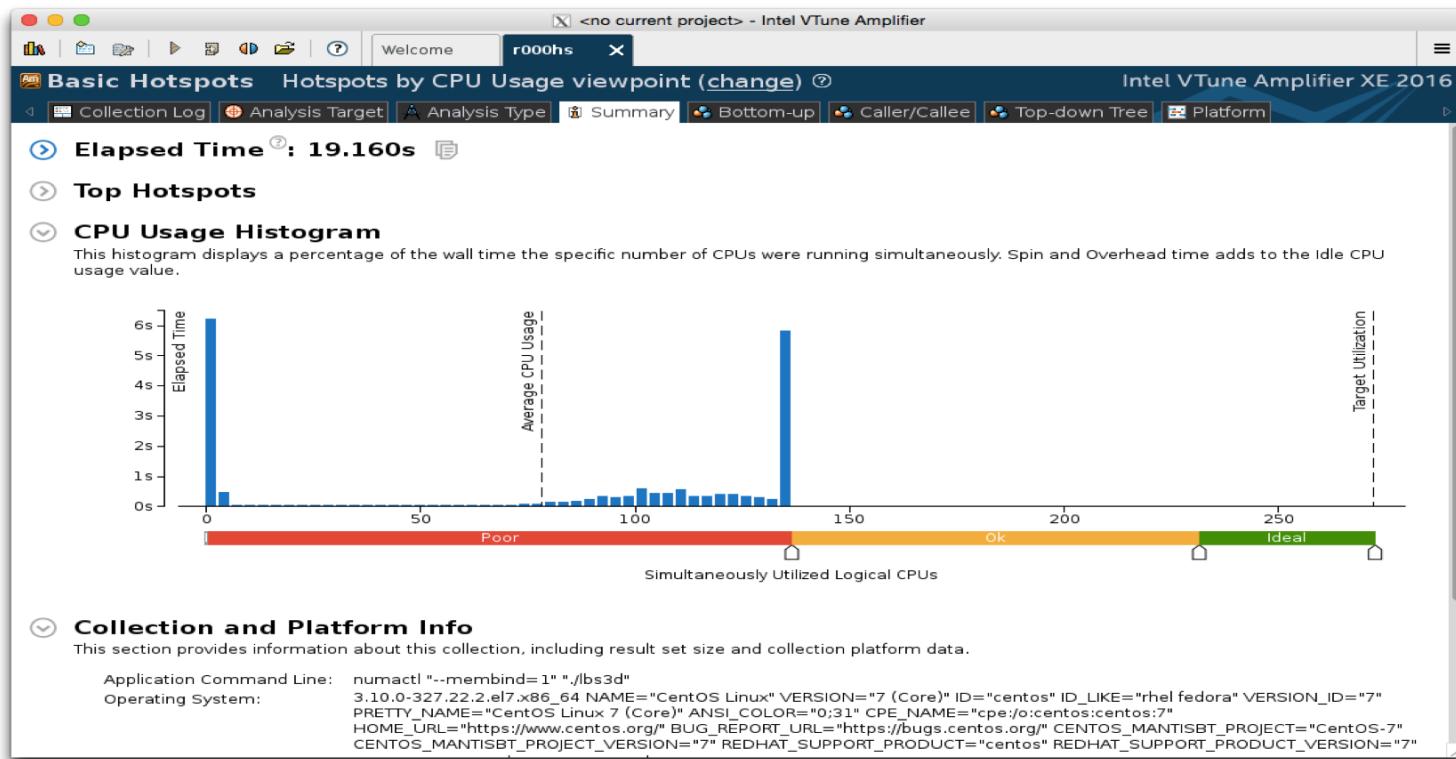
Function	Module	CPU Time
<code>collision_omp\$parallel@51</code>	lbs3d	1254.026s
<code>stream_omp\$parallel_for@39</code>	lbs3d	184.591s
<code>_kmp_fork_barrier</code>	libiomp5.so	150.065s
<code>_kmpc_barrier</code>	libiomp5.so	115.491s
<code>poststream_omp\$parallel@37</code>	lbs3d	36.135s
[Others]	N/A*	27.407s

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

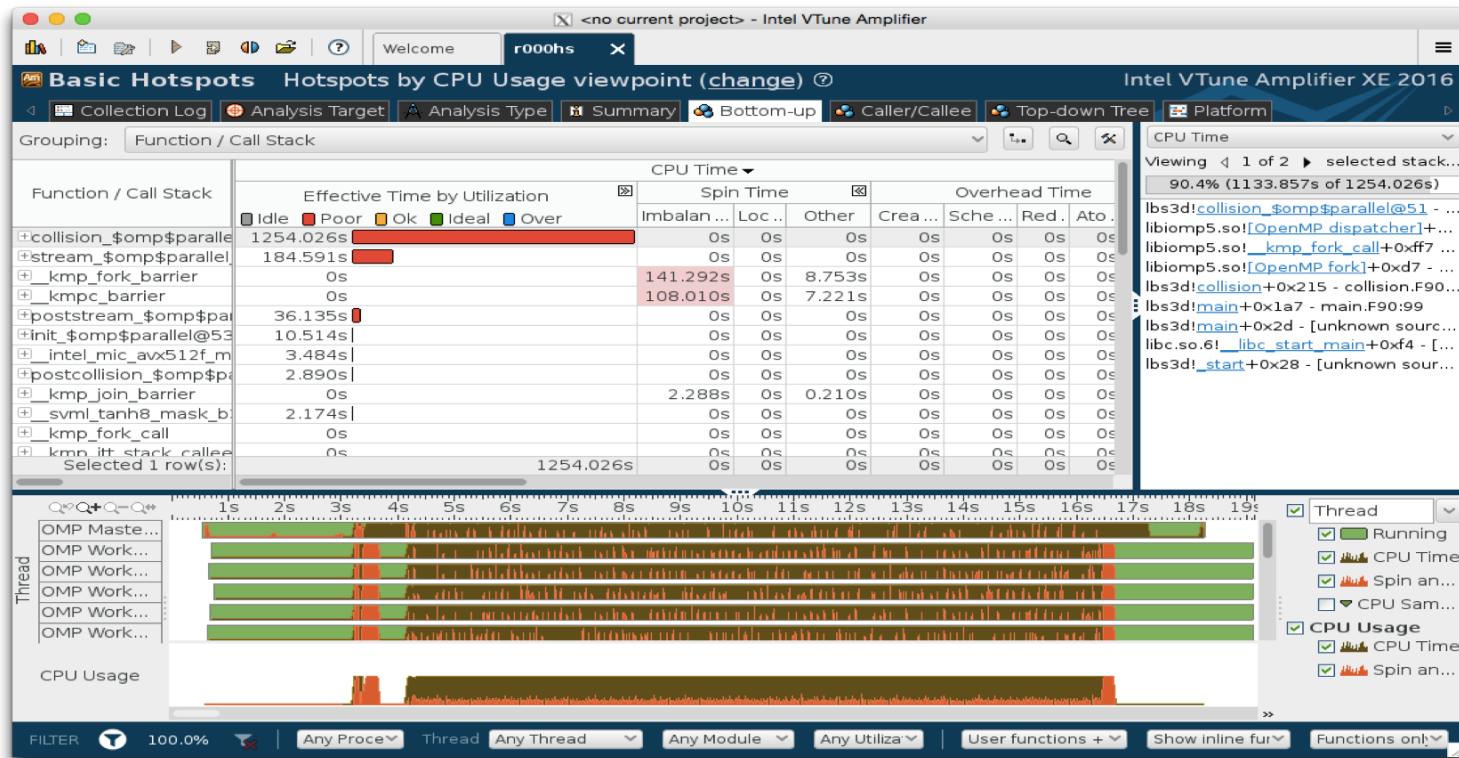
CPU Usage Histogram

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

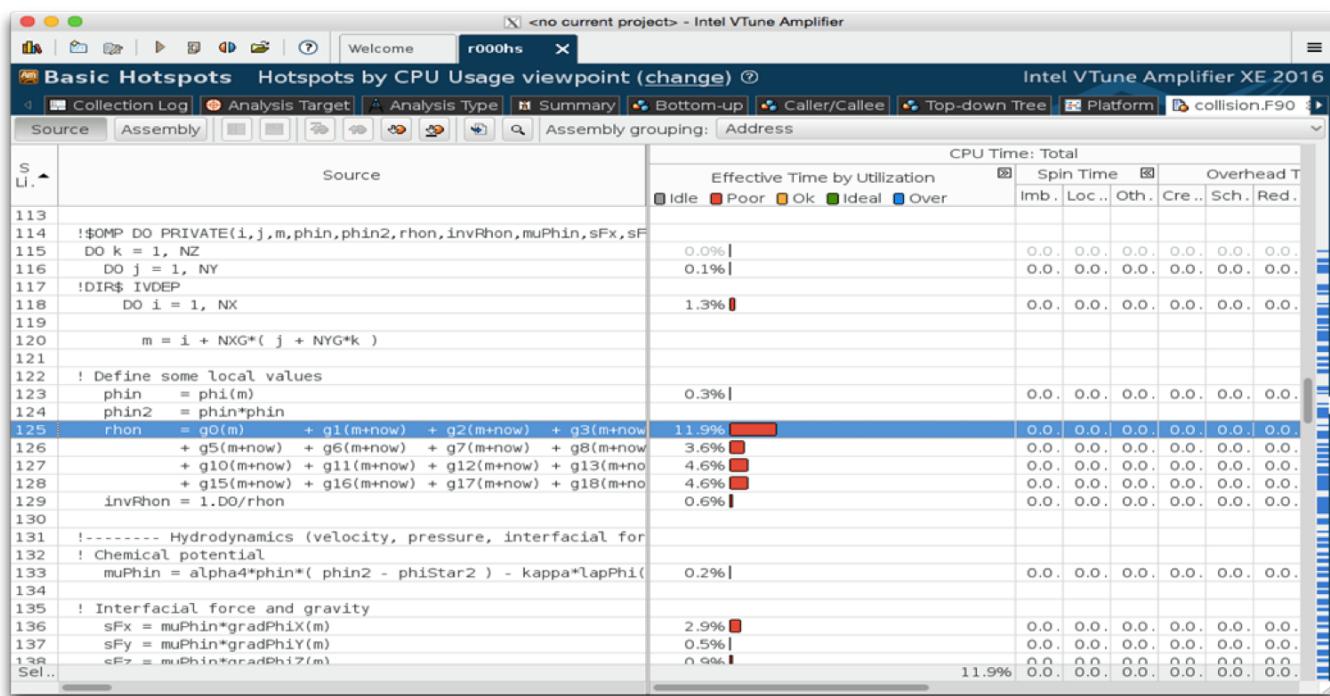
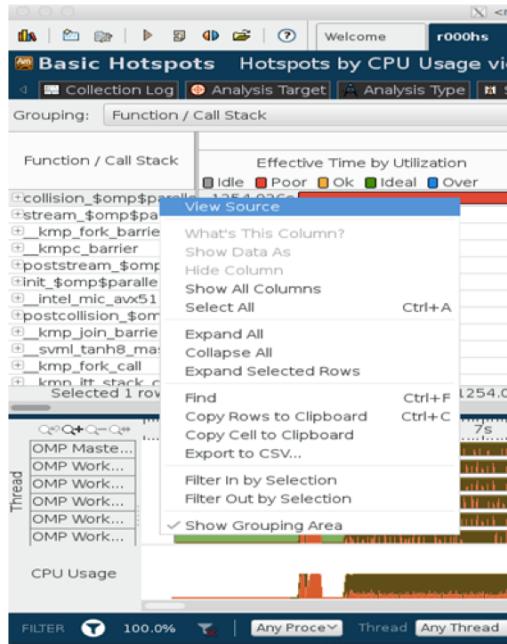
CPU histogram



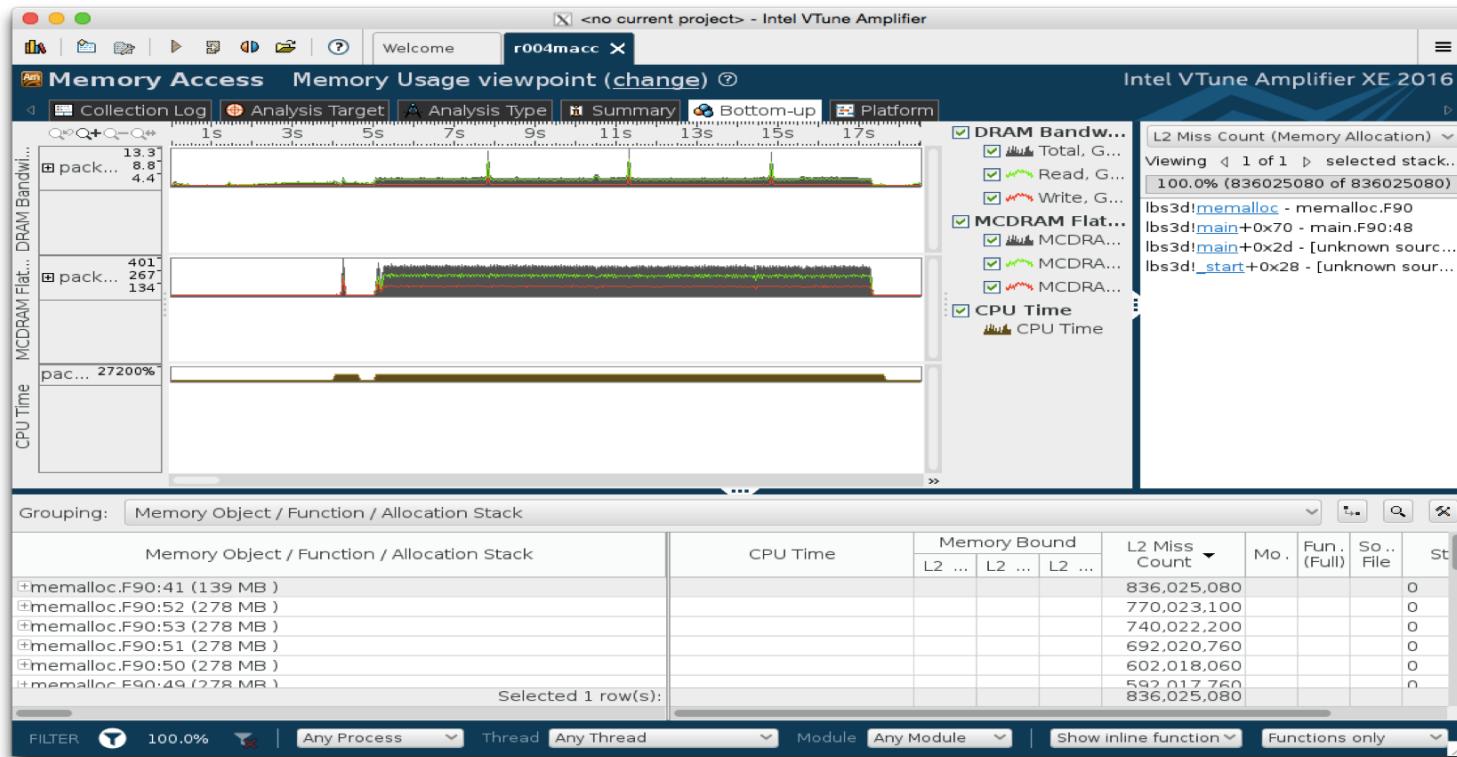
Hotspots Bottom-up View



Getting to the Source



Memory-access Objects Details



hpc-performance Summary

Elapsed Time: 14.854s
GFLOPS Upper Bound: 92.504

CPU Utilization: 40.2%

- Average CPU Usage: 109,260 Out of 272 logical CPUs
- Serial Time: 2.085s (14.0%)
- Parallel Region Time**: 12.769s (86.0%)
- Top OpenMP Regions by Potential Gain**
- CPU Usage Histogram**

Memory Bound

- L2 Hit Bound: 0.143
- L2 Miss Bound: 1.000
- A high number of CPU cycles is being spent waiting for L2 load misses to be serviced. Possible optimizations are to reduce data working set size, improve data access locality, blocking and consuming data in chunks that fit in the L2, or better exploit hardware prefetchers. Consider using software prefetchers but they can increase latency by interfering with normal loads, and can increase pressure on the memory system.
- MCDRAM Flat Bandwidth Bound: 84.0%
- The system spent a significant percentage of elapsed time with high MCDRAM Flat bandwidth utilization. Review the Bandwidth Utilization Histogram to estimate the scale of the issue. Consider improving data locality and/or merging compute-intensive code with bandwidth-intensive code.
- DRAM Bandwidth Bound: 0.0%

FPU Utilization Upper Bound: 1.6%

- GFLOPS Upper Bound**: 92.504
- Scalar GFLOPS Upper Bound: 0.000
- Packed GFLOPS Upper Bound: 92.504
- Top 5 hotspot loops (functions) by FPU usage**

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	FPU Utilization Upper Bound	Loop Characterization
fLoop at line 118 in collision_omp\$parallel@51	1101.094s	2.0%	Vectorized (Body)
fLoop at line 43 in stream_omp\$parallel_for@39	180.998s	0.7%	Vectorized (Body)

Intel Advisor

About Advisor

- Intel Advisor is a vectorization optimization and shared memory threading assistance tool for C, C++, C# and Fortran code
- Advisor supports both serial, threaded, and MPI applications

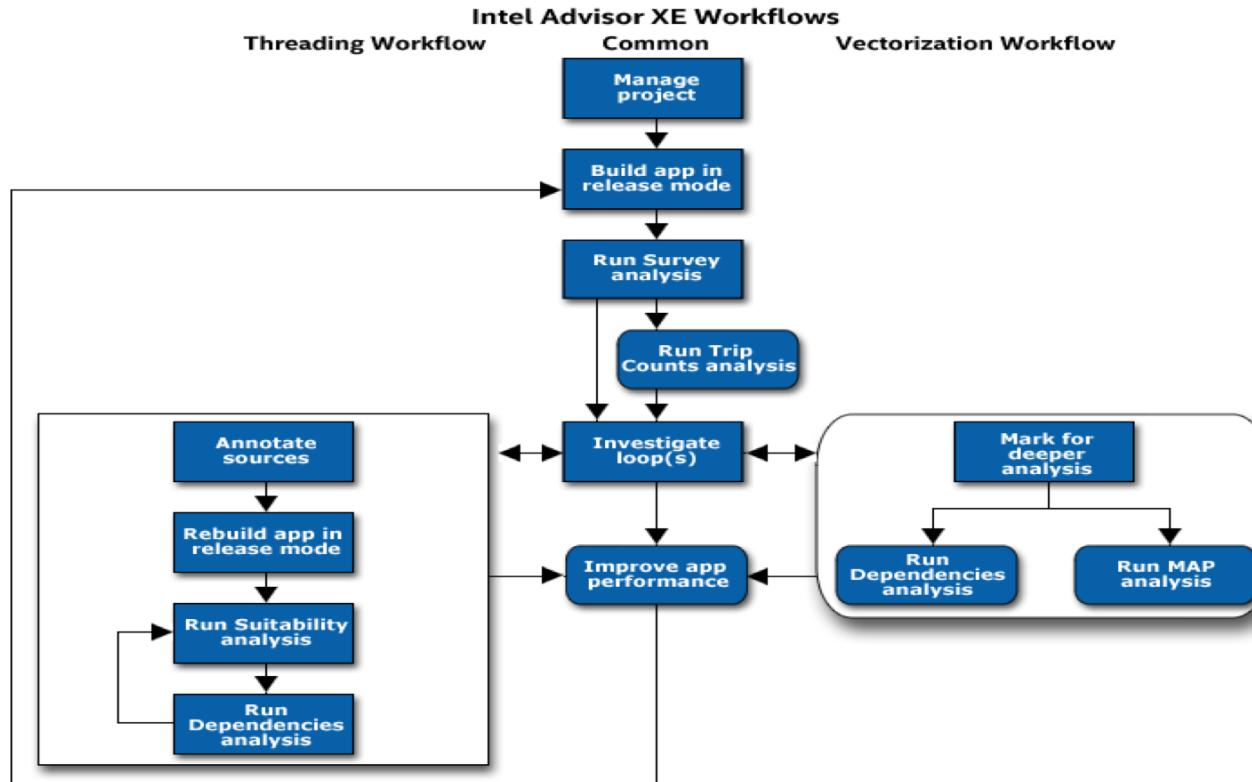
Advisor Capabilities

- Evaluate the efficiency of vectorized code
- Discover where to add parallelism to your code by identifying where your code spends its time
 - You propose parallel code regions when you annotate the parallel sites and tasks.
- Predict the performance you might achieve with the proposed parallel code regions.
- Predict the data sharing problems that could occur in the proposed parallel code regions

Tool	Description	Target Program Requirements
Survey	Helps you discover and select the best places to add parallelism in your program.	Moderate optimization. Limited function inlining.
Trip Counts	Helps you to collect loop iteration statistics.	Moderate optimization. Run survey analysis first
Suitability	Helps you predict the likely performance impact of adding parallelism to the selected places.	Moderate optimization. Limited function inlining.
Dependencies	Helps you predict and eliminate data sharing problems before you add parallelism. 50 to several hundred times slower.	Optimization disabled. Minimize execution time.
Memory Access Patterns (MAP)	Helps you to collect data on memory access strides. 50 to several hundred times slower.	Optimization disabled. Minimize execution time.

Intel Advisor Reference: <https://software.intel.com/en-us/node/527404>

How to use Advisor: workflows



Intel Advisor Reference:

<https://software.intel.com/en-us/node/608339>

Using Advisor

Setup environment

```
$source /opt/intel/advisor_2018.1.0.523188/advixe-vars.sh
```

Compile with optimization and debug symbols

```
$icc -g -xMIC-AVX512 -O2 -qopt-report=5 demo.c
```

Collect data

```
$ advixe-cl -c survey -- ./a.out
```

Summary

0410/huang/training/lab_advisor/advi - Intel Advisor

File View Help

Welcome e000 X

Vectorization Workflow Threading Workflow

Elapsed time: 5.41s Vectorized Not Vectorized FILTER: All Module All Source Loop All Threads

INTEL ADVISOR XE 2016

Summary of predicted parallel behavior

Summary Survey Report Refinement Reports Annotation Report

Vectorization Advisor

Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.

Program metrics

Elapsed Time: 5.41s
Vector Instruction Set: AVX512
Number of CPU Threads: 1

Loop metrics

Total CPU time	5.40s	100.0%
Time in 1 vectorized loop	4.24s	78.5%
Time in scalar code	1.16s	21.5%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency 14.67x
Program Theoretical Gain 11.73x

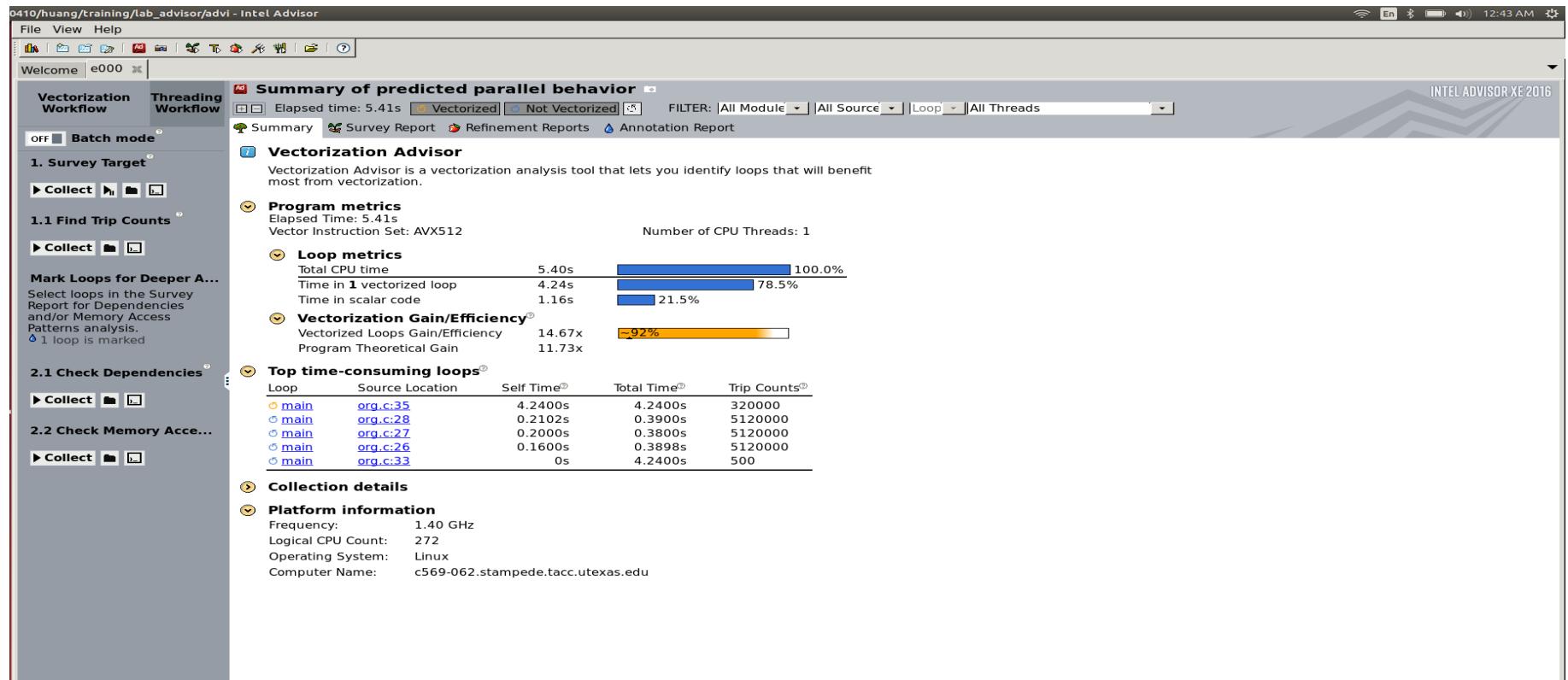
Top time-consuming loops

Loop	Source Location	Self Time ^③	Total Time ^③	Trip Counts ^③
main	org.c.35	4.2400s	4.2400s	320000
main	org.c.28	0.2102s	0.3900s	5120000
main	org.c.27	0.2000s	0.3800s	5120000
main	org.c.26	0.1600s	0.3898s	5120000
main	org.c.33	0s	4.2400s	500

Collection details

Platform information

Frequency: 1.40 GHz
Logical CPU Count: 272
Operating System: Linux
Computer Name: c569-062.stampede.tacc.utexas.edu



Survey, loop analytics

0410/huang/training/lab_advisor/advi - Intel Advisor

File View Help

Welcome e000

Vectorization Workflow Threading Workflow

Where should I add vectorization and/or threading parallelism?

Elapsed time: 5.41s Vectorized Not Vectorized FILTER: All Modules All Sources Loops All Threads

Summary Survey Report Refinement Reports Annotation Report

INTEL ADVISOR XE 2016

Function Call Sites and Loops

	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	Tri...	Instruction Set Analysis		
						Vecto...	Efficiency	Gain ... VL (V... Medi...	Traits	Data Ty...
+ [loop in main at org.c:35]	□	4.240s	4.240s	Vectorized (Body)	function call cannot ...	AVX5...	~92%	14.67x 16	NT-stores	Float32...
+ [loop in main at org.c:28]	□	0.210s	0.210s	Scalar	function call cannot ...				Exponent extractions; ...	Float64
+ [loop in main at org.c:27]	□	0.200s	0.200s	Scalar	function call cannot ...				Exponent extractions; ...	Float64
+ [loop in main at org.c:26]	□	0.160s	0.160s	Scalar	function call cannot ...				Exponent extractions; ...	Float64
+ [loop in main at org.c:33]	□	0.000s	4.240s	Scalar	inner loop was already ...				Mask Manipulations	Int32; U...

1. Survey Target

Collect

1.1 Find Trip Counts

Collect

Mark Loops for Deeper Analysis

Select loops in the Survey Report for Dependencies and/or Memory Access Patterns analysis.

-- There are no marked loops...

2.1 Check Dependencies

Collect

-- Nothing to analyze --

2.2 Check Memory Access

Collect

-- Nothing to analyze --

4.24s

Vectorized (Body) Total time

AVX512F_512 4.24s

Instruction Set Self time

Memory 44% (4) Compute 33% (3) Other 22% (2)

Instruction Mix Summary

14.67x

~92%

Vectorization Gain

~92% Achieved Vectorization Efficiency
Achieved Vectorization Efficiency = (Estimated Gain/Vector Length) * 100%
Estimated Gain = 14.67x
Vector Length = 16

Orange color = Achieved vectorization efficiency is higher than reference efficiency for original scalar loop

⚠ Efficiency is approximately ~92%, which means actual efficiency may be lower

Traits NT-stores

Instruction Mix

Memory: 4 Compute: 3 Other: 2 Number of Vector Registers: 4

Memory: 44.44%	Compute: 33.33%	Other: 22.22%
Vector: 44.44%	Vector: 22.22%	Scalar...
AVX-512: 44.44%	AVX-512: 22.22%	x86: 11...

Other: 22.22%

Survey, loop Assembly

0410/huang/training/lab_advisor/advi - Intel Advisor

File View Help

Welcome e000

Vectorization Workflow Threading Workflow

Where should I add vectorization and/or threading parallelism?

Elapsed time: 5.41s Vectorized Not Vectorized FILTER: All Modules All Sources Loops All Threads INTEL ADVISOR XE 2016

Summary Survey Report Refinement Reports Annotation Report

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	Tri...	Instruction Set Analysis
						Vector... Efficiency Gain ... VL (V... Medi...	Traits	Data Ty...
[loop in main at org.c:35]		4.240s	4.240s	Vectorized (Body)		AVX5 ... -92%	14.67x 16	NT-stores
[loop in main at org.c:28]	3 Data type c...	0.210s	0.210s	Scalar	function call cannot ...			Float32...
[loop in main at org.c:27]	4 Data type c...	0.200s	0.200s	Scalar	function call cannot ...			Exponent extractions; ...
[loop in main at org.c:26]	3 Data type c...	0.160s	0.160s	Scalar	function call cannot ...			Exponent extractions; ...
[loop in main at org.c:33]		0.000s	4.240s	Scalar	inner loop was already...			Mask Manipulations

Mark Loops for Deeper Analysis

Select loops in the Survey Report for Dependencies and/or Memory Access Patterns analysis.

-- There are no marked loops.

2.1 Check Dependencies

Collect -- Nothing to analyze --

2.2 Check Memory Access

Collect -- Nothing to analyze --

Module: org!0x400e4b

Address	Line	Assembly	Total Time	%	Self Time	%	Traits
0x400e80	35	cmp %r11, %r10					
0x400e83	35	jb 0x400e4b <Block 1>					
body	0x400e99	Block 2:					
0x400e99	37	vmovupsz (%r14,%r11,8), %k0, %zmm4	0.050s		0.050s		
0x400ea0	37	vmovupsz 0x40(%r14,%r11,8), %k0, %zmm5	0.560s		0.560s		
0x400ea8	37	vaddpdz (%rbx,%r11,8), %zmm4, %k0, %zmm6	1.100s		1.100s		
0x400ef4	37	vaddpdz 0x40(%rbx,%r11,8), %zmm5, %k0, %zmm7	1.040s		1.040s		
0x400eb7	37	vmovntpdz %zmm6, (%r15,%r11,8)	1.050s		1.050s		
0x400ebe	37	vmovntpdz %zmm7, 0x40(%r15,%r11,8)	0.120s		0.120s		NT-stor...
0x400ec6	35	add \$0x10, %r11	0.190s		0.190s		NT-stor...
0x400eca	35	cmp %r9, %r11	0.060s		0.060s		
0x400ecd	35	jb 0x400e99 <Block 2>	0.070s		0.070s		
remainder	0x400f0a	Block 3:					
0x400f0a	35	vpsubd %ymm0, %ymm5, %ymm6					
0x400f0e	37	lea (%rax,%r10,1), %r11d					
0x400f12	35	vpaddd %ymm2, %ymm5, %ymm5					
0x400f16	37	movsxd %r11d, %r11					
0x400f19	35	add \$0x8, %r10d					
0x400f1d	35	vpcmpqd %zmm4, %zmm6, %k1, %k0					
0x400f23	35	cmp %r9d, %r10d					
0x400f26	35	knotw %k0, %k2					
0x400f2a	37	vmovupdz (%r14,%r11,8), %k2{z}, %zmm7					
0x400f31	37	vmovupdz (%rbx,%r11,8), %k2{z}, %zmm8					

Selected (Total Time): 0s

Profiling do and don't

DO

- Test every change you make
- Profile typical cases
- Compile with optimization flags
- Test for scalability

DO NOT

- Assume a change will be an improvement
- Profile atypical cases
- Profile *ad infinitum*
- Set yourself a goal or
- Set yourself a time limit

Other tools

Valgrind*

valgrind.org

Powerful instrumentation framework, often used for debugging memory problems

MPIP

mpip.sourceforge.net

Lightweight, scalable MPI profiling tool

Scalasca

www.fz-juelich.de/jsc/scalasca

Similar to Tau, complete suit of tuning and analysis tools.

MPIP

<https://github.com/TACC/remora>

REsource MOnitoring for Remote Applications