

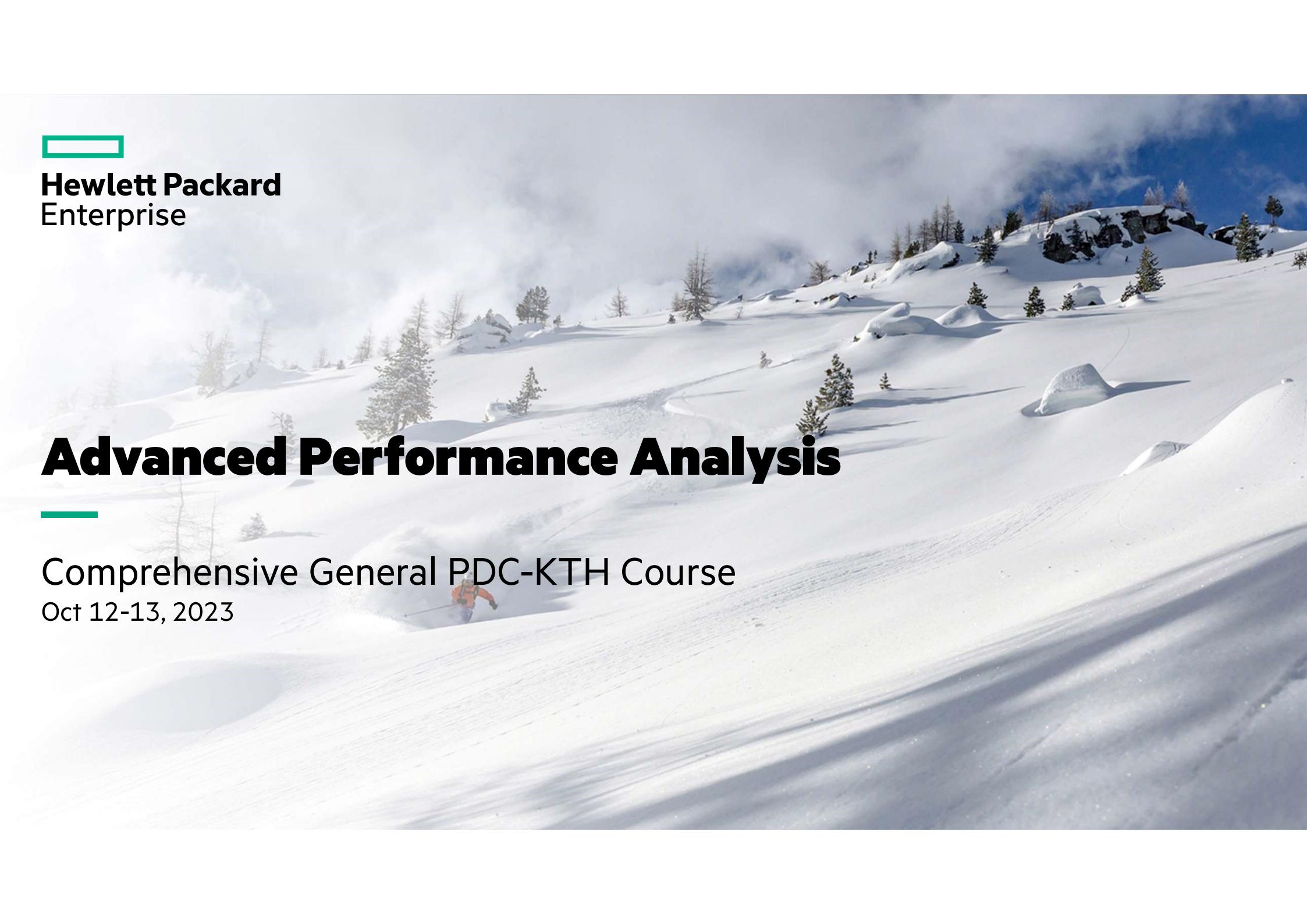


Hewlett Packard
Enterprise

Advanced Performance Analysis

Comprehensive General PDC-KTH Course

Oct 12-13, 2023



Agenda

- In-depth performance analysis with **perftools**
 - Sampling, tracing, and loop work estimates
 - Automatic Performance Analysis
- Reveal
 - Compiler Feedback and Variable Scoping
- OpenMP profiling
- Perftools API
 - Customized performance analysis
- Hardware performance counters
- Load imbalance analysis
- pat_run
 - Profile existing dynamically linked executables
- Concluding remarks



Agenda

- In-depth performance analysis with **perf-tools**
 - Sampling, tracing, and loop work estimates
 - Automatic Performance Analysis
- Reveal
 - Compiler Feedback and Variable Scoping
- OpenMP profiling
- Perf-tools API
 - Customized performance analysis
- Load imbalance analysis
- pat_run
 - Profile existing dynamically linked executables
- Concluding remarks



Perftools

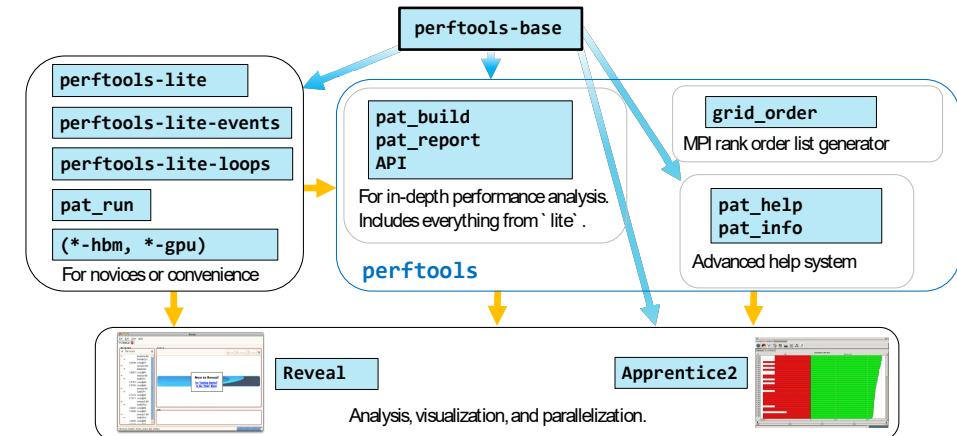
In-depth Performance Analysis



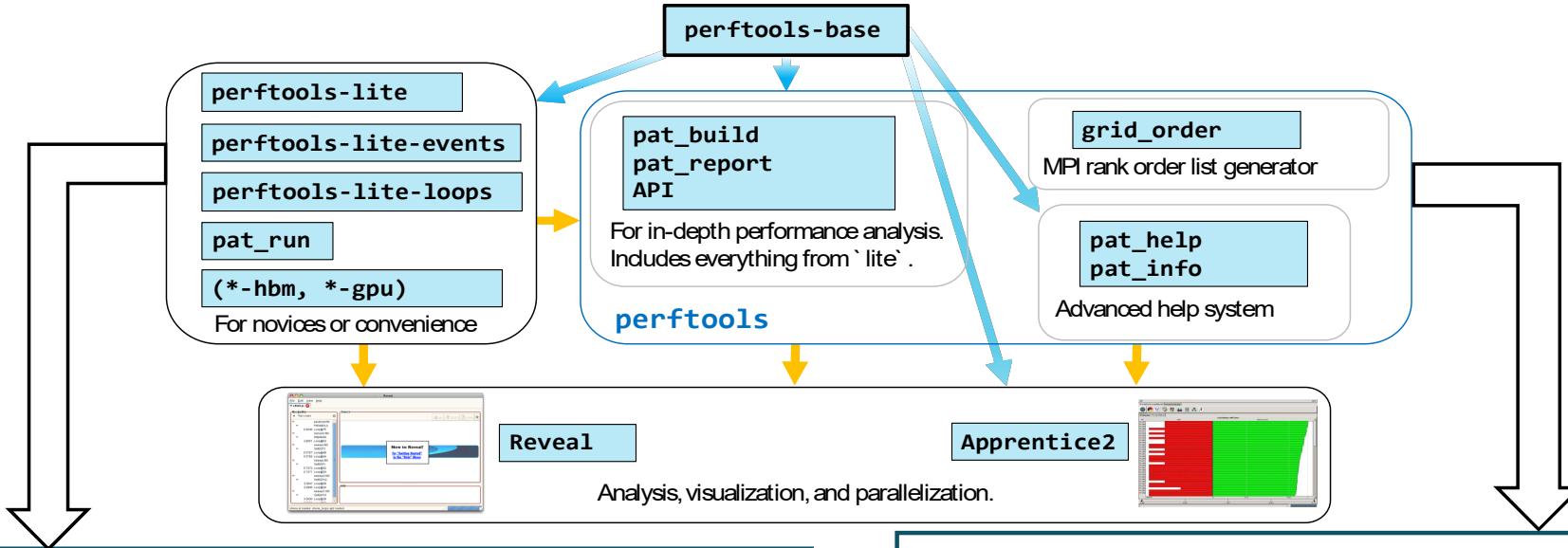
Overview

The `perftools` components:

- `pat_build` — instruments the program to be analyzed.
- `pat_report` — generates text reports from the performance data captured during program execution and exports data for use in other programs.
- These tools together with Reveal, Apprentice2, `pat_help`, and `pat_info` are in principle already available with `perftools-base` but the `perftools` module prevents one of the `perftools-lite`* to automatically instrument a binary.
- Instrumentation and report generation must be done manually.



Overview



Previous talk:

1. Load the **perftools-lite*** module wrt desired profiling
2. Generate the binary
3. Run the binary
4. Look at the **automatically** generated report

This talk:

1. Load the **perftools** module
2. Generate the binary
3. **Manually** generate an **instrumented** binary wrt desired profiling
4. Run the **instrumented** binary
5. **Manually** generate the report
6. Look at the **manually** generated report

Sampling with Perftools

```
> module load perftools-base  
> module load perftools
```

```
> make clean; make  
> pat_build -S app.exe
```

- This generates a new executable **app.exe+pat** and preserves **app.exe**.
- Object files must be present during this stage.

```
> srun -n 8 ./app.exe+pat  
> pat_report -o myrep.trace.rpt app.exe+pat+*/
```

- Running the **app.exe+pat** creates an **app.exe+pat+*/** directory.
- **pat_report** reads that directory and prints a lot of human-readable performance data.
- If worthwhile, a MPI rank reordering file will be produced.

Event Tracing using Perftools

```
> module load perftools-base  
> module load perftools
```

```
> make clean; make  
> pat_build -u -g mpi app.exe
```

- This generates a new executable **app.exe+pat** and preserves **app.exe**.
- If object files and user libraries have already been compiled with perftools enabled, just relink the application with **rm app.exe; make** instead.
- Traces MPI functions calls and functions defined in the program source files.

```
> srun -n 8 ./app.exe+pat  
> pat_report -o myrep.trace.rpt app.exe+pat+*/
```

- Running the **app.exe+pat** creates a **app.exe+pat+*/** directory.
- **pat_report** reads that directory and prints human-readable performance data.

Output: Event Tracing with Perftools

CrayPat/X- Version 6.4.1 Revision 6a6694f 06/27/16 17:24:11

Number of PEs (MPI ranks): 8
Numbers of PEs per Node: 8
Numbers of Threads per PE: 1
Number of Cores per Socket: 16
Execution start time: Tue Mar 7 21:28:21 2017
System name and speed: nid00036 2301 MHz (approx)
Intel haswell CPU Family: 6 Model: 63 Stepping: 2

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
					PE=HIDE
100.0%	36.703706	--	--	3,036.0	Total
97.2%	35.666406	--	--	457.0	IUSER
93.8%	34.432069	0.683771	2.2%	2.0	ljacobi
3.4%	1.232095	0.012880	1.2%	1.0	linitmt
0.0%	0.000941	0.001077	61.0%	1.0	main
0.0%	0.000486	0.000232	36.9%	150.0	sendp3
0.0%	0.000354	0.000445	63.5%	150.0	sendp3
0.0%	0.000266	0.000024	9.3%	1.0	linitcomm
0.0%	0.000172	0.000192	60.3%	150.0	sendp2
0.0%	0.000020	0.000008	31.8%	1.0	lexit
0.0%	0.000002	0.000001	29.9%	1.0	linitmaxclone_12853_1
2.8%	1.019139	--	--	2,423.0	MPI
2.3%	0.833848	0.688832	51.7%	450.0	MPI_Waitall
0.0%	0.169196	0.025124	14.8%	900.0	MPI_Send
0.0%	0.012631	0.016372	64.5%	900.0	MPI_Recv
0.0%	0.001162	0.000014	1.3%	152.0	MPI_Allreduce
0.0%	0.001077	0.000043	4.4%	1.0	MPI_Cart_create
0.0%	0.000703	0.000038	5.8%	3.0	MPI_Type_commit
0.0%	0.018161	--	--	156.0	MPI_SYNC
0.0%	0.014154	0.014143	99.9%	2.0	MPI_Barrier(sync)
0.0%	0.003969	0.003266	82.3%	152.0	MPI_Allreduce(sync)
0.0%	0.000029	0.000027	92.5%	1.0	MPI_InitSync
0.0%	0.000009	0.000009	99.6%	1.0	MPI_FinalizeSync

General job information

Load balance with MPI message statistics

More on MPI with message sizes

Table 2: Load Balance with MPI Message Stats (limited entries shown)

Time%	Time	MPI Msg Count	MPI Msg	Bytes	Avg MPI	Group
						PE=[mmm]
100.0%	36.703700	602.0	197,839,216.0	328,636.57	Total	
97.2%	35.666406	0.0	0.0	0.0	-	USER
99.1%	36.361896	0.0	0.0	0.0	-	pe.3
96.9%	35.581600	0.0	0.0	0.0	-	pe.7
95.3%	34.961048	0.0	0.0	0.0	-	pe.5
2.8%	1.019132	602.0	197,839,216.0	328,636.57	MPI	
4.7%	1.722054	602.0	197,839,216.0	328,636.57	pe.5	
2.4%	0.881184	602.0	197,839,216.0	328,636.57	pe.1	
0.9%	0.332689	602.0	197,839,216.0	328,636.57	pe.3	
0.0%	0.018161	0.0	0.0	0.0	-	MPI_SYNC
0.1%	0.029523	0.0	0.0	0.0	-	pe.6
0.1%	0.019500	0.0	0.0	0.0	-	pe.1
0.0%	0.001622	0.0	0.0	0.0	-	pe.0

Table 3: MPI Message Stats by Caller (limited entries shown)

MPI Bytes%	MPI Msg Count	MPI Msg	MPI MsgSz	64KiB<=	Function Caller
					PE=[mmm]
100.0%	197,839,216.0	602.0	152.0	450.0	Total
100.0%	197,838,600.0	450.0	0.0	450.0	MPI_Isend
40.0%	79,104,600.0	150.0	0.0	150.0	sendp2
3	79,104,600.0	150.0	0.0	150.0	jacobi
4	79,104,600.0	150.0	0.0	150.0	main
6	40.0%	79,104,600.0	150.0	0.0	150.0 pe.0
6	40.0%	79,104,600.0	150.0	0.0	150.0 pe.4
6	40.0%	79,104,600.0	150.0	0.0	150.0 pe.7
40.0%	79,104,600.0	150.0	0.0	150.0 sendp1	
3	79,104,600.0	150.0	0.0	150.0 jacobi	
4	79,104,600.0	150.0	0.0	150.0 main	
5	40.0%	79,104,600.0	150.0	0.0	150.0 pe.0
5	40.0%	79,104,600.0	150.0	0.0	150.0 pe.4
5	40.0%	79,104,600.0	150.0	0.0	150.0 pe.7
20.0%	39,629,400.0	150.0	0.0	150.0 sendp3	

All user defined routines traced and shown (-T option).

Remarks for Tracing

- More information is given in the [pat_build](#) man page
- Only true function calls can be traced. Functions that are inlined by the compiler or that have local scope in a compilation unit cannot be traced
- **-g tracegroup** Instrument the program to trace all function references belonging to the trace function group tracegroup. Only those functions actually executed by the program at runtime are traced. A selection of tracegroup values is:

– blas	Basic Linear Algebra subprograms
– netcdf	Network Common Data Form
– hdf5	HDF5 I/O library
– heap	dynamic heap
– io	includes stdio and sysio groups
– lapack	Linear Algebra Package
– mpi	MPI
– omp	OpenMP API
– sysio	I/O system calls
– syscall	system calls
– hip	hip calls

- More information on the various tracegroup values is given in [\\$CRYPAT_ROOT/share/traces](#) after loading the [perf-tools-base](#) module

Event Tracing using Perftools for GPU (HIP)

```
> module load PrgEnv-cray  
> module load craype-x86-trento craype-accel-amd-gfx90a rocm  
> module load perftools-base  
> module load perftools
```

```
> make clean; make  
> pat_build -u -g hip app.exe
```

- This generates a new executable `app.exe+pat` and preserves `app.exe`.
- If object files and user libraries have already been compiled with perftools enabled, just relink the application with `rm app.exe; make` instead.
- Traces MPI functions calls, openMP offload directives and functions defined in the program source files.

```
> srun -n 8 ./app.exe+pat  
> pat_report -T -o myrep.trace.rpt app.exe+pat*/
```

- Running the `app.exe+pat` creates a `app.exe+pat*/` directory.
- `pat_report` reads that directory and prints human-readable performance data.

Output: Event Tracing with Perftools for GPU (HIP)

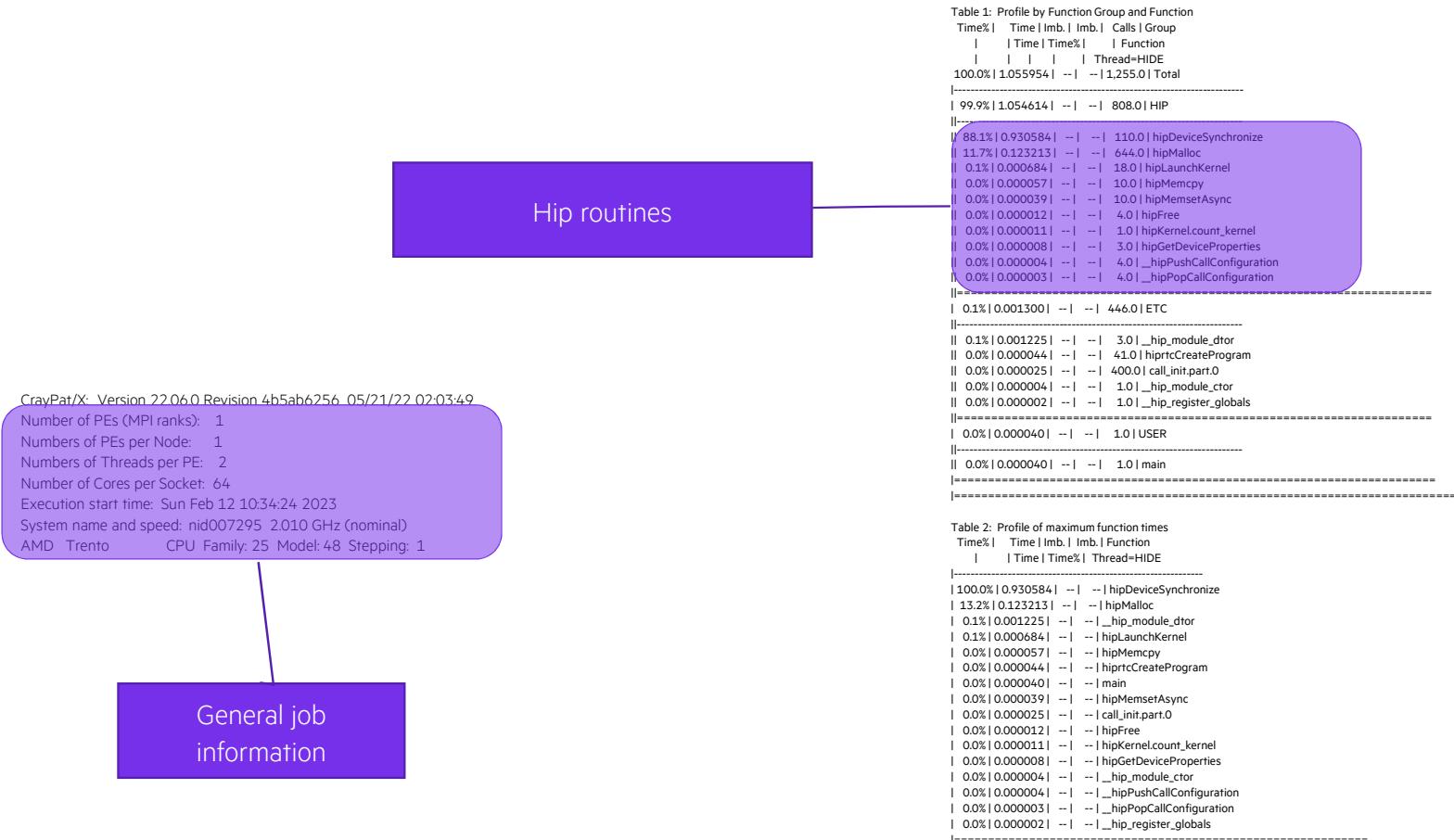


Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Function
100.0%	1.055954	-	-	1,255.0	Total
99.9%	1.054614	-	-	808.0	HIP
88.1%	0.930584	-	-	110.0	hipDeviceSynchronize
11.7%	0.123213	-	-	644.0	hipMalloc
0.1%	0.000684	-	-	18.0	hipLaunchKernel
0.0%	0.000057	-	-	10.0	hipMemcpy
0.0%	0.000039	-	-	10.0	hipMemsetAsync
0.0%	0.000012	-	-	4.0	hipFree
0.0%	0.000011	-	-	1.0	hipKernel.count_kernel
0.0%	0.000008	-	-	3.0	hipGetDeviceProperties
0.0%	0.000004	-	-	4.0	_hipPushCallConfiguration
0.0%	0.000003	-	-	4.0	_hipPopCallConfiguration
0.1%	0.001300	-	-	446.0	ETC
0.1%	0.001225	-	-	3.0	_hip_module_dtor
0.0%	0.000044	-	-	41.0	hipRtcCreateProgram
0.0%	0.000025	-	-	400.0	call_init.part.0
0.0%	0.000004	-	-	1.0	_hip_module_ctor
0.0%	0.000002	-	-	1.0	_hip_register_globals
0.0%	0.000040	-	-	1.0	USER
0.0%	0.000040	-	-	1.0	main

Table 2: Profile of maximum function times

Time%	Time	Imb.	Imb.	Function
100.0%	0.930584	-	-	hipDeviceSynchronize
13.2%	0.123213	-	-	hipMalloc
0.3%	0.001225	-	-	_hip_module_dtor
0.1%	0.000684	-	-	hipLaunchKernel
0.0%	0.000057	-	-	hipMemcpy
0.0%	0.000044	-	-	hipRtcCreateProgram
0.0%	0.000040	-	-	main
0.0%	0.000039	-	-	hipMemsetAsync
0.0%	0.000025	-	-	call_init.part.0
0.0%	0.000012	-	-	hipFree
0.0%	0.000011	-	-	hipKernel.count_kernel
0.0%	0.000008	-	-	hipGetDeviceProperties
0.0%	0.000004	-	-	_hip_module_ctor
0.0%	0.000004	-	-	_hipPushCallConfiguration
0.0%	0.000003	-	-	_hipPopCallConfiguration
0.0%	0.000002	-	-	_hip_register_globals

Event Tracing using Perftools for GPU (MPI+OpenMP offload)

```
> module load PrgEnv-cray  
> module load craype-x86-trento craype-accel-amd-gfx90a rocm  
> module load perftools-base  
> module load perftools
```

```
> make clean; make  
> pat_build -u -g mpi,omp app.exe
```

- This generates a new executable `app.exe+pat` and preserves `app.exe`.
- If object files and user libraries have already been compiled with perftools enabled, just relink the application with `rm app.exe; make` instead.
- Traces MPI functions calls, openMP offload directives and functions defined in the program source files.

```
> srun -n 8 ./app.exe+pat  
> pat_report -T -o myrep.trace.rpt app.exe+pat*/
```

- Running the `app.exe+pat` creates a `app.exe+pat*/` directory.
- `pat_report` reads that directory and prints human-readable performance data.

Output: Event Tracing with Perftools for GPU (mpi+openMP offload)



Loop Work Estimates using Perftools

```
> module load perftools-base  
> module load perftools
```

- Use the compiler flag **-h profile_generate** for Fortran or **-finstrument-loops** for C for your build (**Cray Compiler only**). This flag turns off OpenMP and significant compiler loop restructuring optimizations except for vectorization.

```
> make clean; make  
> pat_build -w app.exe
```

- This generates a new executable **app.exe+pat** and preserves **app.exe**.
- Loop profiling requires a rebuild of object files and user libraries.
- Automatically traces functions defined in the program source files despite **-w**

```
> srun -n 8 ./app.exe+pat  
> pat_report -T -o myrep.trace.rpt app.exe+pat+*/
```

- Running the **app.exe+pat** creates a directory **app.exe+pat+*/** experiment directory.
- **pat_report** reads that directory and prints a lot of human-readable performance data.

Automatic Profiling Analysis (1/2)

```
> module load perftools-base  
> module load perftools
```

```
> make clean; make  
> pat_build app.exe
```

- The APA (-O apa) is the default experiment. No option needed.
- The **pat_build** generates a binary instrumented for sampling (different from the pure sampling shown before with -S option)

```
> srun -n 8 app.exe+pat  
> pat_report -o myrep.txt app+pat+*/
```

- Running the “**+pat**” binary creates an experiment directory.
- Applying **pat_report** to the **app+pat+*/** directory generates an ***.apa** file therein.

Automatic Profiling Analysis (2/2)

```
> vi app.exe+pat+*/*.apa
```

- The ***.apa** file contains instructions for the next step, i.e. tracing. Modify it according to your needs.

```
> pat_build -O app.exe+pat+*/*.apa
```

- Generates an instrumented binary **app.exe+apa** for tracing according to the instructions in the **app.exe+pat+*/*.apa** file.

```
> srun -n 8 ./app.exe+apa  
> pat_report -o myrep.txt app.exe+apa+*/
```

- Running the **app.exe+apa** binary creates a new data file or directory.
- Applying **pat_report** to the **app+apa+*/** directory generates a new report.

Output: APA with perftools

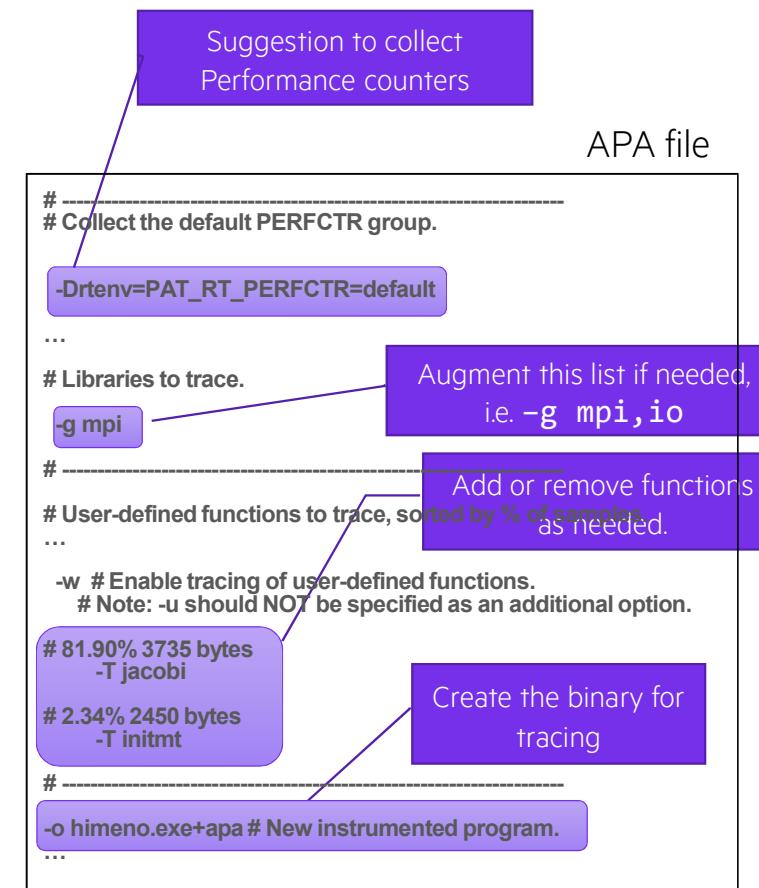
Sampling

CrayPat/X: Version 6.4.1 Revision 6a6694f 06/27/16 17:24:11

Number of PEs (MPI ranks): 8
Numbers of PEs per Node: 8
Numbers of Threads per PE: 1
Number of Cores per Socket: 16

```
Execution start time: Wed Mar 8 14:24:42 2017
System name and speed: nid00036 2301 MHz (approx)
Intel haswell CPU Family: 6 Model: 63 Stepping: 2
...
Samp% | Samp | Imb. | Imb. | Group
| | Samp | Samp% | Function
| | | | PE=HIDE
100.0% | 3,685.0 | -- | -- | Total
|-----|
| 84.2% | 3,104.5 | -- | -- | USER
|-----|
|| 81.9% | 3,018.1 | 57.9 | 2.2% | jacobi
|| 2.3% | 86.1 | 2.9 | 3.7% | initmt
|-----|
| 13.3% | 489.9 | -- | -- | IETC
|-----|
|| 12.2% | 451.2 | 6.8 | 1.7% | _cray_scopy_HSW
|| 1.0% | 38.6 | 2.4 | 6.6% | _cray_set_HSW
|-----|
| 2.5% | 90.6 | -- | -- | MPI
|-----|
|| 1.9% | 69.6 | 46.4 | 45.7% | MPI_Waitall
|-----|
...
Total
```

CPU_CLK_THREAD_UNHALTED:THREAD_P	109,408,212,031
CPU_CLK_THREAD_UNHALTED:REF_XCLK	3,588,344,079
DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK	53,188,056
PM_ENERGY:NODE	30,382 /sec
CPU_CLK	3.05GHz
TLB utilization	268.83 refs/miss
D1 cache hit,miss ratios	70.8% hits
D1 cache utilization (misses)	3.43 refs/miss
D2 cache hit,miss ratio	67.8% hits
D1+D2 cache hit,miss ratio	90.6% hits
D1+D2 cache utilization	10.65 refs/miss
D2 to D1 bandwidth	6,149.405MiB/sec
	236,055,684,984 bytes



Event Tracing

CrayPat/X: Version 6.4.1 Revision 6a6694f 06/27/16 17:24:11

Number of PEs (MPI ranks): 8
Numbers of PEs per Node: 8
Numbers of Threads per PE: 1
Number of Cores per Socket: 16

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
Time	Time%	PE=HIDE	Function		
100.0%	37.748701	--	--	2,584.0	Total
95.9%	36.184768	--	--	5.0	USER
92.5%	34.935081	1.178364	3.7%	2.0	jacobi
3.3%	1.248950	0.016487	1.5%	1.0	initmt
4.1%	1.537996	--	--	2,423.0	MPI
3.6%	1.344349	1.096798	51.3%	450.0	MPI_Waitall
...					
					USER /jacobi
Time%					92.5%
Time					34.935081 secs
Imb. Time					1.178364 secs
Imb. Time%					3.7%
Calls				0.057 /sec	2.0 calls
CPU_CLK_THREAD_UNHALTED:THREAD_P					104,712,083,433
CPU_CLK_THREAD_UNHALTED:REF_XCLK					3,465,232,627
DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK					44,033,261
DTLB_STORE_MISSES:MISS_CAUSES_A_WALK					26,116,487
L1D:REPLACEMENT					6,955,645,277
L2_RQSTS:ALL_DEMAND_DATA_RD					3,675,034,522
L2_RQSTS:DEMAND_DATA_RD_HIT					1,408,925,969
MEM_UOPS_RETIRED:ALL_LOADS					22,406,881,264
CPU_CLK					3.02GHz
TLB utilization				319.41 refs/miss	0.62 avg uses
D1 cache hit,miss ratios				69.0% hits	31.0% misses
D1 cache utilization (misses)				3.22 refs/miss	0.40 avg hits

Using pat_report

- Always need to run **pat_report** at least once to perform data conversion
 - Experiment directory contains *.xf files in the xf-files/ subdirectory
 - Perftools combines information from **xf** output and executable to produce **ap2** stored in the ap2-files/ subdirectory : **instrumented binary must still exist when data is converted!**
 - Resulting **ap2** files are the input for subsequent **pat_report** calls and Reveal or Apprentice²
 - **Always use the entire experiment directory as input and not single ap2 files**
 - **xf** files and instrumented binary files could be removed once **ap2** file is generated.
- Generates a text report of performance results
 - Data laid out in tables
 - Many options for sorting, slicing or dicing data in the tables.
 - > **pat_report -O <table option> app.exe+pat+*/**
 - > **pat_report -O help (list of available profiles)**
 - Volume and type of information depends upon sampling vs tracing

Using pat_report

- The performance numbers reported are in general an average over all tasks (also explains non-integer values)
- Not always meaningful
 - controller/worker schemes
 - MPMD
- To solve this you can filter the *.ap2 file

```
> pat_report -sfilter_input='condition' ...
```

- The 'condition' should be an expression involving 'pe' such as 'pe<1024' or 'pe%2==0'.
- This option is also useful when the size of the full data file makes a report incorporating data from all PEs take too long or exceed the available memory

More help:

- `pat_report -h` => usage
- `pat_report -O -h` => available report tables
- `pat_report -S -h` => options for content and format
- `pat_report -D -h` => options for data columns

Time%	Time	Imb.	Imb.	Calls	Group	
					Time	Time%
100.0%	20.643909	--	--	1149.0	Total	
98.8%	20.395989	--	--	219.0	USER	
91.1%	18.797060	0.115535	0.7%	2.0	jacobi	
7.7%	1.597866	0.006647	0.5%	1.0	initmt	
0.0%	0.000402	0.000167	33.5%	53.0	sendp3	

General Remarks

- Always check that the **instrumenting binary has not affected the run time significantly** compared to the original executable
- Collecting event traces on **large numbers of frequently called functions**, or setting the **sampling interval very low** can introduce a **lot of overhead** (check `trace-text-size` option to `pat_build`)
- **Highly recommended to run on Lustre !**
- The runtime analysis can be modified using environment variables of the form `PAT_RT_*`
 - Check the `PAT_LD_OBJECT_TMPDIR` variable if you cannot preserve the original build tree
- `pat_build` may recognize for instance MPI in your application and trace **`MPI_Init` and `MPI_Finalize`** when adding `-g mpi` trace **all MPI calls**
- Processing the `app+*/` directory from `perftools-lite` yields all the options of `pat_build` to reproduce the experiment with regular `perftools`.

Reveal

Performance analysis and code restructuring assistant



Motivation: Compiler Listings (CCE)

- Much information produced by compiler.
 - Can be listed together with source code for better clarity.
 - Use **-fsave-loopmark** for C and **-hlist=m** (or **a**) for Fortran.
 - It generates an ***.lst** file for every source code file.
 - A + sign indicates that more information can be found after the routine definition.
 - Can also be inspected in Reveal with a corresponding program library.

%%% Loopmark Legend %%%

Primary Loop Type	Modifiers
A - Pattern matched	a - atomic memory operation b - blocked c - conditional and/or computed
C - Collapsed	
D - Deleted	
E - Cloned	
F - Flat - No calls	f - fused
G - Accelerated	g - partitioned
I - Inlined	i - interchanged
M - Multithreaded	m - partitioned n - non-blocking remote transfer p - partial
R - Rerolling	r - unrolled s - shortloop w - unwound
V - Vectorized	w - vectorized
+ - More messages listed at end of listing	

210. +1-----< for(i=0 ; i<imax ; ++i)
211. +1 2-----< for(j=0 ; j<jmax ; ++j)
212. 1 2 Vr2-----< for(k=0 ; k<kmax ; ++k){
213. 1 2 Vr2 a[0][i][j][k]=1.0;
214. 1 2 Vr2 a[1][i][j][k]=1.0;
215. 1 2 Vr2 a[2][i][j][k]=1.0;
216. 1 2 Vr2 a[3][i][j][k]=1.0/6.0;
217. 1 2 Vr2 b[0][i][j][k]=0.0;
218. 1 2 Vr2 b[1][i][j][k]=0.0;
219. 1 2 Vr2 b[2][i][j][k]=0.0;
220. 1 2 Vr2 c[0][i][j][k]=1.0;
221. 1 2 Vr2 c[1][i][j][k]=1.0;
222. 1 2 Vr2 c[2][i][j][k]=1.0;
223. 1 2 Vr2 A-----> p[i][j][k]=(float)((i+it)*(i+it))/(float)((mx-1)*(mx-1));
224. 1 2 Vr2 A-----> wrk1[i][j][k]=0.0;
225. 1 2 Vr2 A-----> wrk2[i][j][k]=0.0;
226. 1 2 Vr2 A-----> bnd[i][j][k]=1.0;
227. 1 2 Vr2----->>> }
228. }

CC-6294 CC: VECTOR initmt, File = himeno.c, Line = 210
A loop was not vectorized because a better candidate was found at line 212.

CC-6294 CC: VECTOR initmt, File = himeno.c, Line = 211
A loop was not vectorized because a better candidate was found at line 212.

CC-6005 CC: SCALAR initmt, File = himeno.c, Line = 212
A loop was unrolled 2 times.

CC-6204 CC: VECTOR initmt, File = himeno.c, Line = 212
A loop was vectorized.

Input to Reveal: Program Library

```
> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90  
> cc -O3 -fcray-program-library-path=my_program.pl -c my_program_file2.c
```

- Recompile only sources to generate program library **my_program.pl**
- The program library is most useful when generated from fully optimized code.
- Use absolute paths to specify the program library if necessary.
- Use **-hpl=my_program.pl** for C/C++ for cce/*classic modules.

```
> reveal my_program.pl &
```

- Optionally, collect loop work estimates in a separate experiment and load it with the program library or directly from the tool.
- Note that **-hprofile_generate** option disables OpenMP and significant compiler loop restructuring optimizations except for vectorization, which is why Cray recommends generating this data separately from generating the program library.
- Desktop client installer: /opt/cray/pe/perftools/<version>/share/desktop_installers/

Input to Reveal: Program Library

- Mandatory

```
> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90  
> cc -O3 -fcray-program-library-path=my_program.pl -c my_program_file2.c  
> reveal my_program.pl &
```

- Recompile only sources to generate program library `my_program.pl`

- Optional but highly recommended

```
> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90  
> cc -O3 -fcray-program-library-path=my_program.pl -c my_program_file2.c  
> reveal my_program.pl my_program_exe_*/ &
```

- Collect loop work estimates in a separate experiment and load it too with `my_program.pl`
- Note that `-hprofile_generate` option disables OpenMP and significant compiler loop restructuring optimizations except for vectorization

Reveal

- View source, performance, and optimization information at the same time.

The screenshot shows the Reveal IDE interface. On the left is a 'Navigation' pane titled 'Loop Performance' which lists various loops and their instances. The main area is a 'Source' editor showing a portion of a Fortran-like program:

```
Source - /home/users/heidi/reveal/parabola.f90
66
67 do n = nmin, nmax
68     deltaa(n) = ar(n) - al(n)
69     a6(n)      = 6. * (a(n) - .5 * (al(n) + ar(n)))
70     scrch1(n) = (ar(n) - a(n)) * (a(n)-al(n))
71     scrch2(n) = deltaa(n) * deltaa(n)
72     scrch3(n) = deltaa(n) * a6(n)
73 enddo
74
75 do n = nmin, nmax
76     if(scrch1(n) <= 0.0) then
77         ar(n) = a(n)
78         al(n) = a(n)
79     endif
```

The code is color-coded: lines 66-73 are grey, line 74 is light blue, and lines 75-79 are yellow. A callout box points from the 'Navigation' pane to line 74. The right side of the interface is a 'Loopmark Legend' window with a detailed list of optimization flags and their meanings.

Loopmark Legend

- A Pattern Matched
- C Collapsed
- D Deleted
- E Cloned
- G Accelerated
- I Inlined
- H Not inlined
- L Loop
- M Multithreaded
- R Region
- S Scoping Analysis
- V Vectorized
- a Atomic Memory Operation
- b Blocked
- c Conditional and/or Computed
- t Fused
- g Partitioned
- i Interchanged
- n Non-blocking Remote Transfer
- p Partial
- r Unrolled
- s Shortloop
- w Unwound

Info - Line 67

A loop starting at line 67 was fused with the loop starting at line 53.

vhone.pl loaded. vhone_loops.ap2 loaded.

Reveal

- View source, performance, and optimization information at the same time.

The screenshot shows the Reveal IDE interface. On the left is a navigation pane titled "Loop Performance" with a tree view of loop instances. The main area displays the source code of a Fortran file named "himeno.f". The code contains several loops, some of which are highlighted with different colors (e.g., green, blue, grey) to indicate their status or characteristics. The right side of the interface contains a "Loopmark Legend" that defines these colors and other symbols used in the code highlighting. The legend includes categories such as Pattern Matched, Collapsed, Deleted, Cloned, Flat, Accelerated, Inlined, Not Inlined, Loop, Multithreaded, Region, Scoping Analysis, and Vectorized. It also includes symbols for Atomic Memory Operation, Blocked, Conditional and/or Computed, Fused, Partitioned, Interchanged, Non-blocking Remote Transfer, Partial, Unrolled, Shortloop, and Unwound. At the bottom of the interface, there is a status bar with the message "himeno.pl loaded. himeno.exe+pat+48593-8638944t loaded."

Navigation

Source - /pfs/lustrep2/users/bracconi/Reveal/himeno.f

cce/14.0.2 Up Down Save

Loopmark Legend

Info - Line 211

A loop starting at line 211 is flat (contains no external calls).

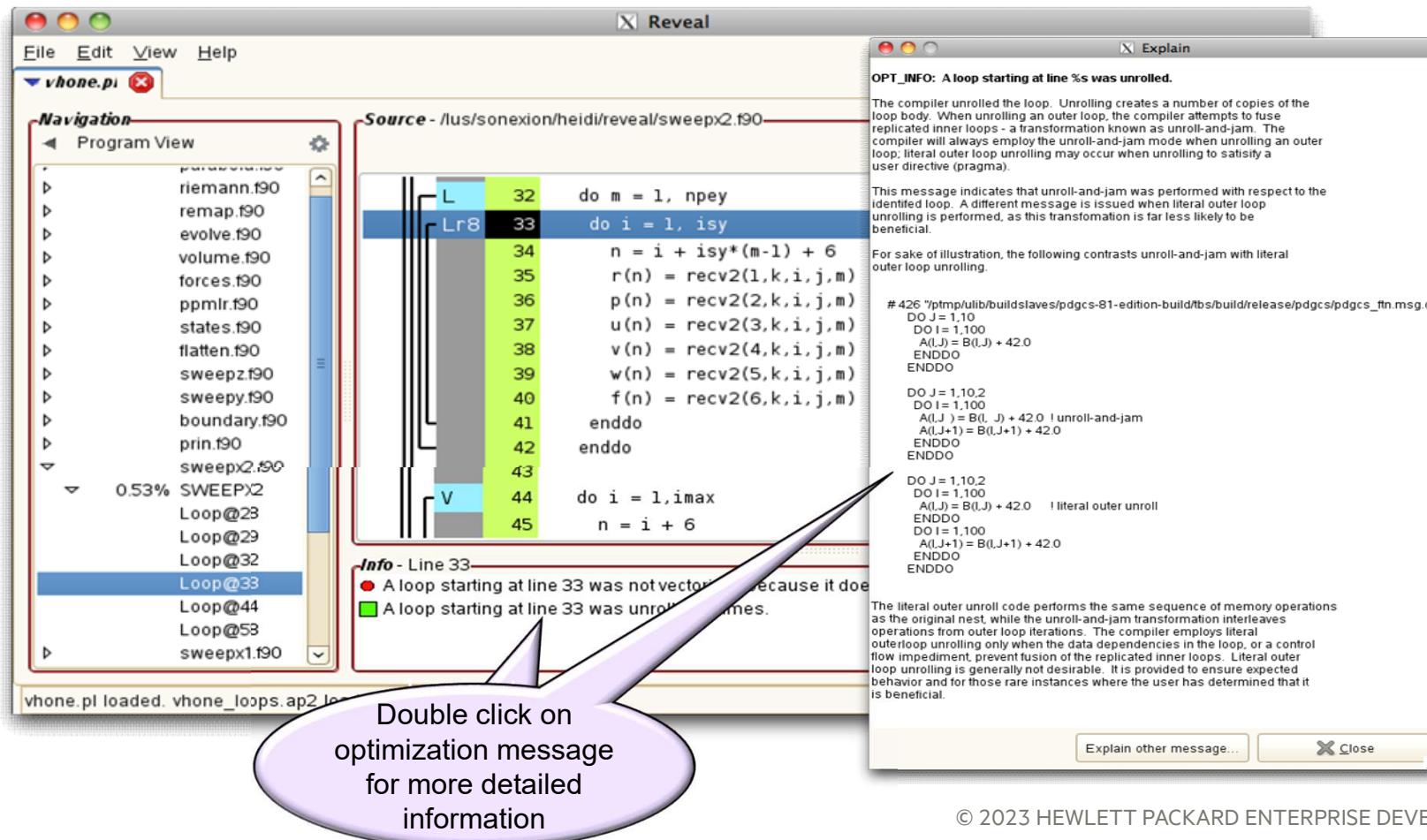
A loop starting at line 211 was not vectorized because a better candidate was found at line 213.

A loop starting at line 212 is flat (contains no external calls).

himeno.pl loaded. himeno.exe+pat+48593-8638944t loaded.

Reveal

- Access Cray compiler message information



Reveal

- Access Cray compiler message information

The screenshot shows the Reveal IDE interface. On the left, the 'Program View' navigation pane lists source files and functions, with 'himeno.f' selected. The main window displays the source code for 'himeno.f' with line numbers 211 to 236. A callout bubble points to the line number 213, which is highlighted in grey. The code at line 213 is:

```
DO K=2, kmax-1
  DO J=2, jmax-1
    DO I=2, imax-1
      S0=a(I,J,K,1)*p(I+1,J,K)+a(I,J,K,2)*p(I,J+1,K)
      +a(I,J,K,3)*p(I,J,K+1)
      +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K)
      -p(I-1,J+1,K)+p(I-1,J-1,K))
      +b(I,J,K,2)*(p(I,J+1,K+1)-p(I,J-1,K+1)
      -p(I,J+1,K-1)+p(I,J-1,K-1))
      +b(I,J,K,3)*(p(I+1,J,K+1)-p(I-1,J,K+1)
      -p(I+1,J,K-1)+p(I-1,J,K-1))
      +c(I,J,K,1)*p(I-1,J,K)+c(I,J,K,2)*p(I,J-1,K)
      +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
      SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
      WGOSA=WGOSA+SS*SS
      wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
    enddo
  enddo
enddo
```

Below the code, an info bar states: "A loop starting at line 213 is flat (contains no external calls).", "A loop starting at line 213 was unrolled 3 times.", and "A loop starting at line 213 was vectorized." A callout bubble points to this info bar with the text: "Double click on optimization message for more detailed information".

On the right, a separate window titled 'Explain ftn-6005' provides detailed information about the unrolling transformation:

OPT_INFO: A loop starting at line %s was unrolled.

The compiler unrolled the loop. Unrolling creates a number of copies of the loop body. When unrolling an outer loop, the compiler attempts to fuse replicated inner loops - a transformation known as unroll-and-jam. The compiler will always employ the unroll-and-jam mode when unrolling an outer loop; literal outer loop unrolling may occur when unrolling to satisfy a user directive (pragma).

This message indicates that unroll-and-jam was performed with respect to the identified loop. A different message is issued when literal outer loop unrolling is performed, as this transformation is far less likely to be beneficial.

For sake of illustration, the following contrasts unroll-and-jam with literal outer loop unrolling.

```
DO J = 1,10
  DO I = 1,100
    A(I,J) = B(I,J) + 42.0
  ENDDO
ENDDO

DO J = 1,10,2
  DO I = 1,100
    A(I,J) = B(I,J) + 42.0 ! unroll-and-jam
    A(I,J+1) = B(I,J+1) + 42.0
  ENDDO
ENDDO

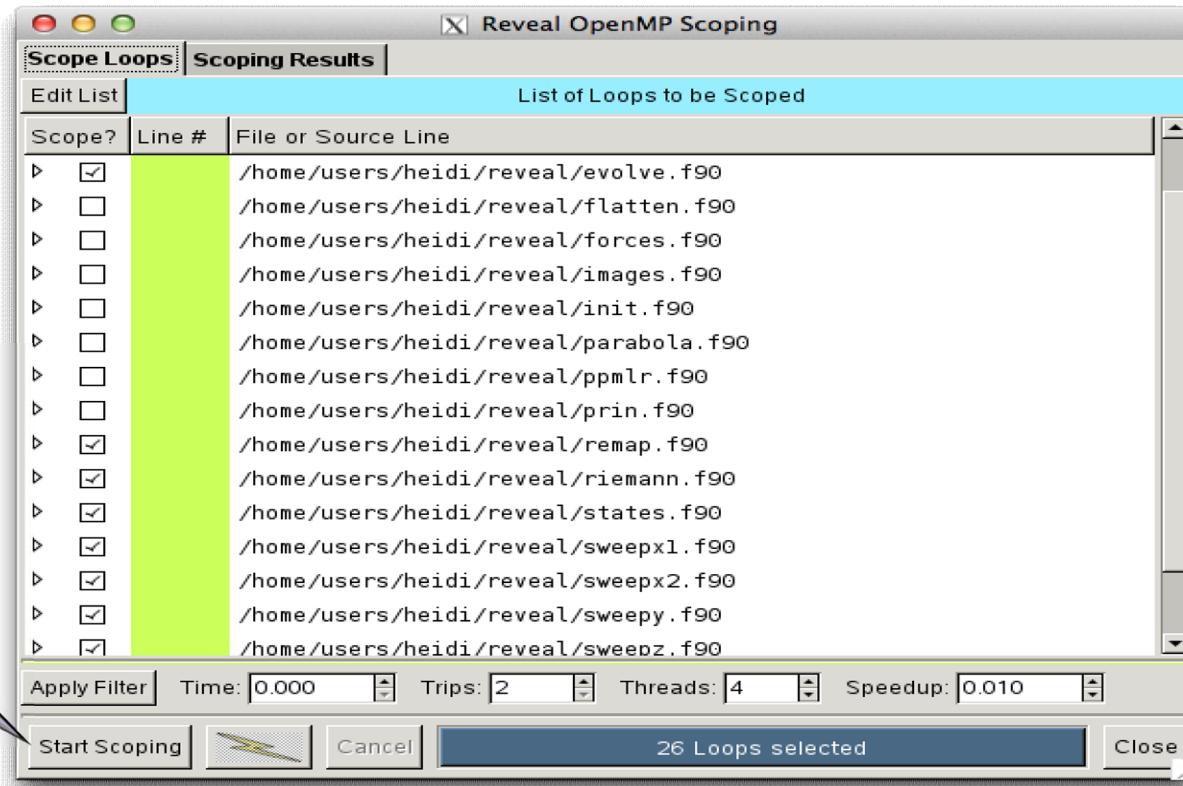
DO J = 1,10,2
  DO I = 1,100
    A(I,J) = B(I,J) + 42.0 ! literal outer unroll
    A(I,J+1) = B(I,J+1) + 42.0
  ENDDO
ENDDO
```

Explain other message... **X Close**

A callout bubble points to the 'Explain other message...' button with the text: "Double click on optimization message for more detailed information".

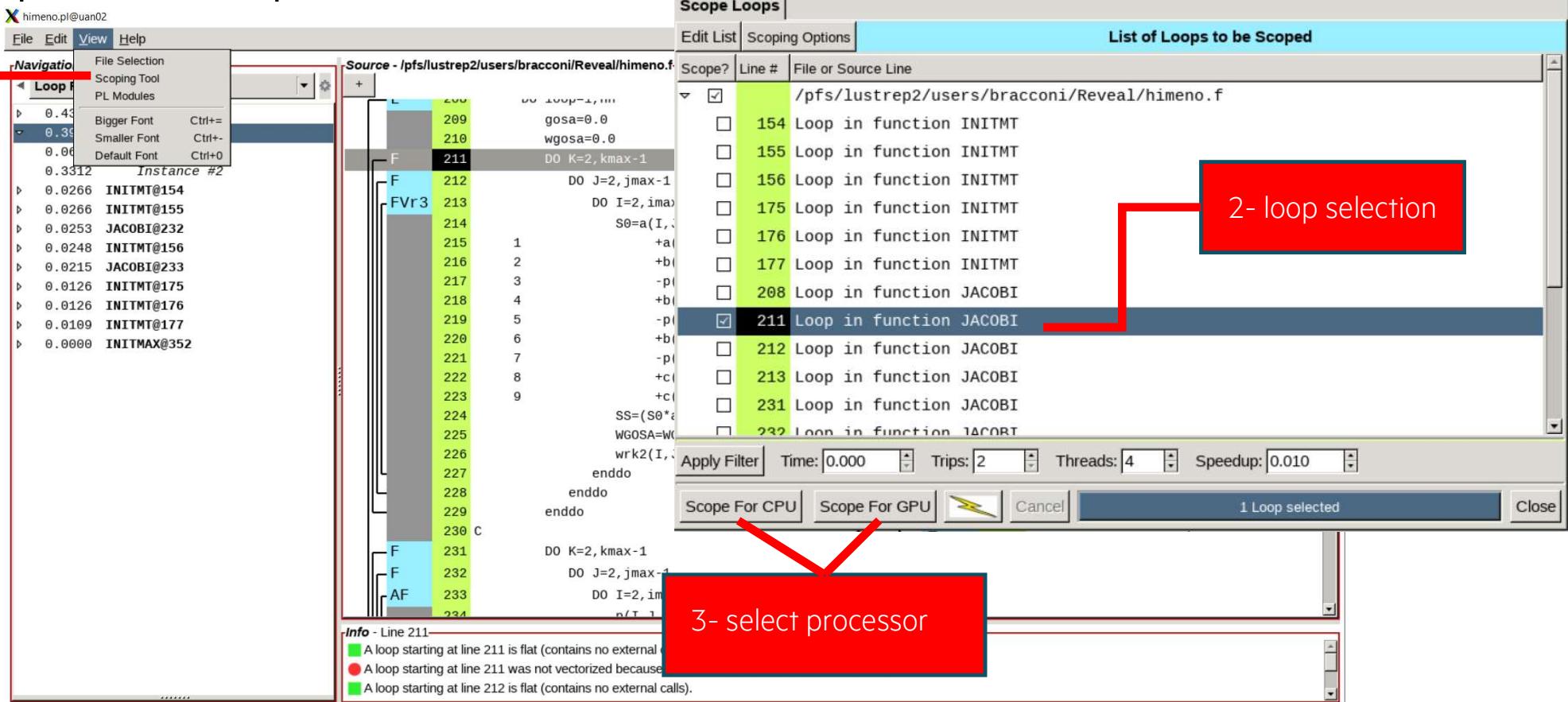
Reveal

- Scope selected loop(s)



Reveal

- Scope selected loop(s)



Reveal

- Scope selected loop(s) for CPU

himeno.f: Loop@211

Name	Type	Scope
a	Array	P S C U
b	Array	P S C U
bnd	Array	P S C U
c	Array	P S C U
i	Scalar	P S C U
imax	Scalar	P S C U
j	Scalar	P S C U
jmax	Scalar	P S C U
k	Scalar	P S C U
kmax	Scalar	P S C U

Directive inserted by Cray Reveal. May be incomplete.

```
!$OMP parallel do default(None)
!$OMP& shared(a, b, bnd, c, imax, jmax, kmax, omega, p, wgosa, wrk1,
!$OMP& wrk2)
!$OMP& private(i, j, k, s0, ss)
!$OMP& reduction(+: wgosa)
```

First/Last Private

Enable FirstPrivate

Enable LastPrivate

Find Name: []

Reduction

None

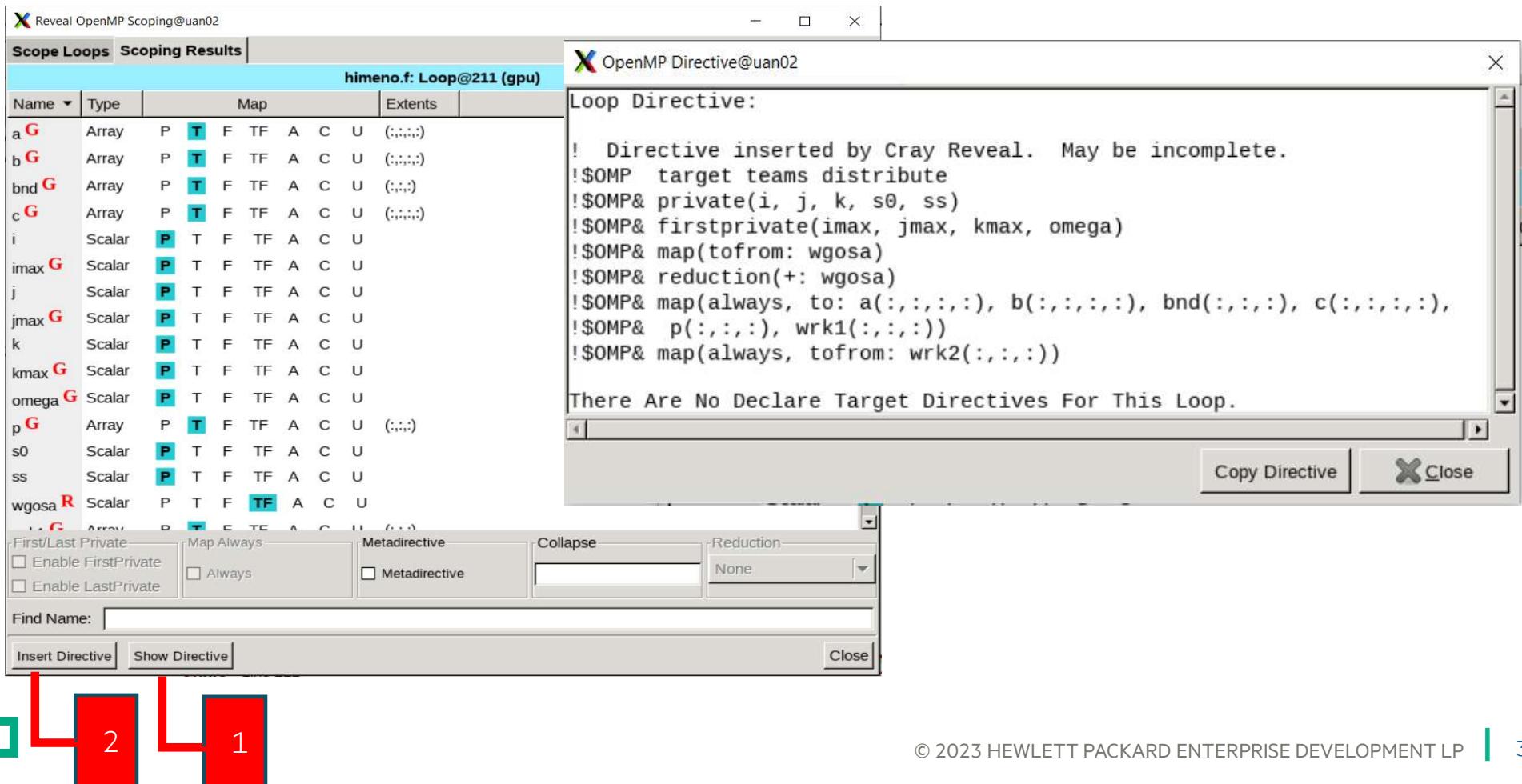
Insert Directive | Show Directive | Close

2

1

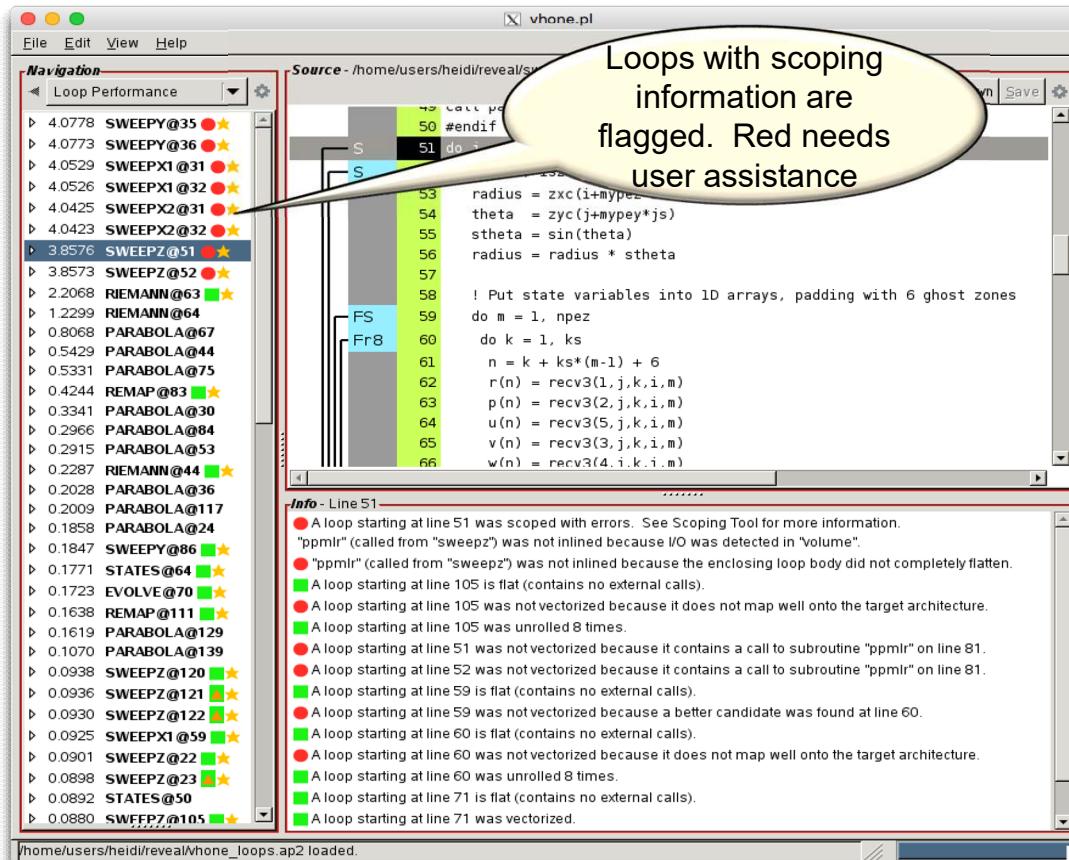
Reveal

- Scope selected loop(s) for GPU



Reveal

- Review scoping results.

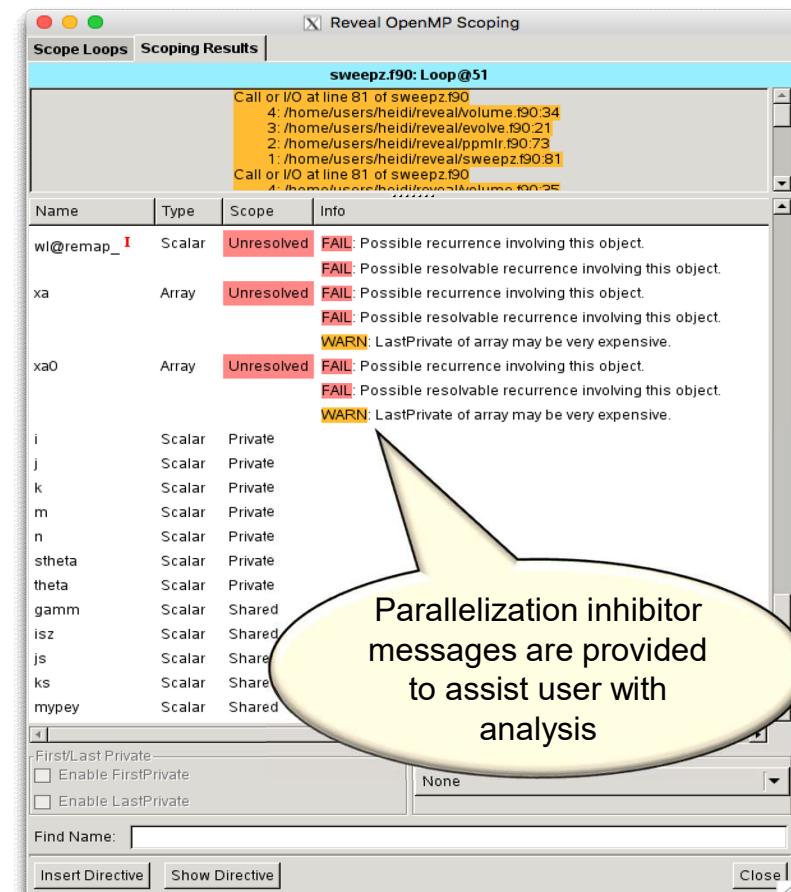


The screenshot shows the Reveal IDE interface. On the left is a navigation pane titled "Loop Performance" listing various loops with their execution times and error counts. In the center is the source code for "vhone.pl". A callout bubble points to a red flagged entry point for a loop labeled "3.8576 SWEEPZ@51". The source code snippet for this loop is:

```
radius = zxc(i+mpey)
theta = zyc(j+mpey*js)
stheta = sin(theta)
radius = radius * stheta

! Put state variables into 1D arrays, padding with 6 ghost zones
do m = 1, npez
    do k = 1, ks
        n = k + ks*(m-1) + 6
        r(n) = recv3(1,j,k,i,m)
        p(n) = recv3(2,j,k,i,m)
        u(n) = recv3(5,j,k,i,m)
        v(n) = recv3(3,j,k,i,m)
        w(n) = recv3(4,i,k,i,m)
```

At the bottom, an "Info - Line 51" section provides detailed analysis of the flagged loop, including reasons for errors and optimization attempts.



The screenshot shows the "Reveal OpenMP Scoping" tool. It displays the "Scoping Results" for the "sweepz.f90" file. A callout bubble points to a message indicating parallelization inhibitor messages:

sweepz.f90: Loop@51

Name	Type	Scope	Info
wl@remap_	I	Unresolved	FAIL: Possible recurrence involving this object. FAIL: Possible resolvable recurrence involving this object.
xa	Array	Unresolved	FAIL: Possible recurrence involving this object. FAIL: Possible resolvable recurrence involving this object. WARN: LastPrivate of array may be very expensive.
xa0	Array	Unresolved	FAIL: Possible recurrence involving this object. FAIL: Possible resolvable recurrence involving this object. WARN: LastPrivate of array may be very expensive.
i	Scalar	Private	
j	Scalar	Private	
k	Scalar	Private	
m	Scalar	Private	
n	Scalar	Private	
stheta	Scalar	Private	
theta	Scalar	Private	
gamm	Scalar	Shared	
isz	Scalar	Shared	
js	Scalar	Shared	
ks	Scalar	Shared	
mpey	Scalar	Shared	

First/Last Private
 Enable FirstPrivate
 Enable LastPrivate

None

Find Name: []

Insert Directive | Show Directive | Close |

Parallelization inhibitor messages are provided to assist user with analysis

Reveal

- Generate OpenMP directives.

```
! Directive inserted by Cray Reveal. May be incomplete.
!$OMP parallel do default(none)
!$OMP& unresolved (dvol,dx,dx0,e,f,flat,p,para,q,r,radius,svel,u,v,w,
!$OMP& xa,xa0)
!$OMP& private (i,j,k,m,n,$$_n,delp2,delp1,shock,temp2,old_flat,
!$OMP& onemfl,hdt,sinxfo,gamfac1,gamfac2,dtheta,deltx,fractn,
!$OMP& ekin)
!$OMP& shared (gamm,isy,js,ks,mypey,ndim,ngeom,y,nlefty,npay,nrighty,
!$OMP& recv1,send2,zdy,zxc,zya)
do k = 1, ks
  do i = 1, isy
    radius = zxc(i+mypey*isy)

    ! Put state variables into 1D arrays, padding with 6 ghost zones
    do m = 1, npay
      do j = 1, js
        n = j + js*(m-1) + 6
        r(n) = recv1(1,k,j,i,m)
        p(n) = recv1(2,k,j,i,m)
        u(n) = recv1(4,k,j,i,m)
        v(n) = recv1(5,k,j,i,m)
        w(n) = recv1(3,k,j,i,m)
        f(n) = recv1(6,k,j,i,m)
      enddo
    enddo

    do j = 1, jmax
      n = j + 6
    enddo
```

Reveal generates OpenMP directive with illegal clause marking variables that need addressing

Reveal

- Validate user inserted directive.

The screenshot shows the Reveal tool interface with the following details:

- Title Bar:** Reveal
- File Menu:** File, Edit, View, Help
- Project:** vphone.pl
- Navigation:** Program View (selected), showing a tree view of files: images.f90, init.f90, parabola.f90, ppmir.f90, prin.f90, remap.f90, riemann.f90 (selected), RIEMANN (selected), Loop@44, Loop@69 (selected), Loop@70, Loop@89, states.f90, sweepx1.f90, sweepx2.f90, sweepy.f90, sweepz.f90, vh1mod.s90, vphone.f90, volume.f90, zonemod.f90. A message "vphone.pl loaded" is displayed at the bottom.
- Source Editor:** Source - /ufs/home/users/heidi/reveal/riemann.f90. The code is color-coded by scope:
 - Line 63: Directive inserted by Cray Reveal. May be incomplete.
 - Line 64: !\$OMP parallel do default(private)
 - Line 65: !\$OMP& private (l)
 - Line 66: !\$OMP& shared (lmin,lmax,prgh,urgh,vrgh,plft,ulft,vlft,pmid,clft,
 - Line 67: !\$OMP& crgh,gamfac1,gamfac2,plfti,pmold,prghi,umidl,umidr
 - Line 68: !\$OMP& wlft,wrgh,zlft,zrgh,n)
 - Line 69: do l = lmin, lmax (highlighted with a blue arrow)
 - Line 70: do n = 1, 12
 - Line 71: pmold(l) = pmid(l)
 - Line 72: wlft (l) = 1.0 + gamfac1*(pmid(l) - plft(l))
 - Line 73: wrgh (l) = 1.0 + gamfac1*(pmid(l) - prgh(l))
 - Line 74: wlft (l) = clft(l) * sqrt(wlft(l))
 - Line 75: wrgh (l) = crgh(l) * sqrt(wrgh(l))
 - Line 76: zlft (l) = 4.0 * vlft(l) * wlft(l) * wlft(l)
- Info Panel:** Info - Line 69
 - A loop starting at line 69 was not vectorized for an unspecified reason.
 - A loop starting at line 69 was partitioned.
- Scope Loops Dialog:** riemann.f90: Loop@69
 - Scope Loops tab selected.
 - Scoping Results table:

Name	Type	Scope	Info
I	Scalar	Private	
n	Scalar	Private	WARN: Scope does not agree with user OMP directive.
clft	Array	Shared	
crgh	Array	Shared	
gamfac1	Scalar	Shared	
gamfac2	Scalar	Shared	
 - First/Last Private section:
 - Enable FirstPrivate
 - Enable LastPrivate
 - Reduction section: None
 - Find Name:
 - Buttons: Insert Directive, Show Directive, Close
- Annotations:** A callout bubble points to the line "do n = 1, 12" with the text "User inserted directive with mis-scoped variable 'n'".

Reveal

- Look for vectorization opportunities.

Choose “Compiler Messages” view to access message filtering, then select desired type of message

The screenshot shows the Reveal IDE interface. On the left, there is a large orange callout bubble pointing to the 'Compiler Messages' view. The 'Compiler Messages' view has a red border and contains a dropdown menu with 'Not Vectorized' selected. Below the dropdown are several file entries with their execution times: 'line 123' (0.000 sec), 'images.f90' (line 149), 'init.f90' (lines 113-139), 'prin.f90' (lines 125-129), 'riemann.f90' (line 63: 0.387 sec, line 64: 0.224 sec), and 'sweepx1.f90' (lines 31-59). On the right, the 'Source' editor window displays the code for 'vhone.f90'. The code includes loops at lines 63 and 64. The 'Info - Line 64' panel at the bottom right provides analysis: 'A loop starting at line 64 is flat (contains no external calls)' (green icon) and 'A loop starting at line 64 was not vectorized because a recurrence was found on "pmid" at line 77.' (red icon). The status bar at the bottom of the IDE indicates 'vhone.pl loaded. vhone_loops.ap2 loaded.'

OpenMP profiling



OpenMP Data Collection and Reporting

- For OpenMP programs

- Measure the overhead incurred by entering and leaving parallel regions and work-sharing constructs within parallel regions
- Show per-thread timings in addition to other data.
- Calculate the load balance across threads for such constructs.

- For programs that use both MPI and OpenMP

- Profiles by default show the load balance over PEs of the average time in the threads for each PE
- But you can also see load balances for each programming model separately.

- Options for `pat_report`

- `profile_pe.th`

- Imbalance based on the set of all threads in the program

- `profile_pe_th` (default view)

- Highlights imbalance across MPI ranks
 - Uses max for thread aggregation to avoid showing under-performers
 - Aggregated thread data merged into MPI rank data

- `profile_th_pe`

- For each thread, show imbalance over MPI ranks
 - Example: Load imbalance shown where thread 4 in each MPI rank didn't get much work

OpenMP Data Collection and Reporting

- OpenMP support needs to be enabled during compilation
- OpenMP tracing calls inserted by default when perftools is loaded

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function PE=HIDE Thread=HIDE
100.0%	2.452453	--	--	1426.8	Total
96.9%	2.377154	--	--	309.8	USER
82.1%	2.013394	0.027282	1.8%	100.0	work.LOOP@li.533
10.6%	0.259470	0.000282	0.1%	1.0	exit
2.4%	0.057711	0.000562	1.3%	1.0	initializeMatrix
1.0%	0.024130	0.000313	1.7%	1.0	setPEsParams.SINGLE@li.355
1.6%	0.039963	--	--	909.0	MPI
1.6%	0.039247	0.079519	89.3%	301.5	MPI_Wait
1.2%	0.029108	--	--	101.0	OMP
1.2%	0.029058	0.012000	39.0%	100.0	work.REGION@li.492(ovhd)

Work sharing construct

Region

Overhead

Table 2: Load Imbalance by Thread

Max. Time	Imb. Time	Imb. Time%	Thread PE=HIDE
2.452470	0.316486	17.2%	Total
2.453287	0.000817	0.0%	thread.0
2.078727	0.036293	2.3%	thread.2
2.074969	0.048712	3.1%	thread.1
2.066243	0.043468	2.8%	thread.3

Perftools API

Customized performance analysis.



API for Adding User Instrumentation

The Perftools API calls enable you to insert functions into your source code that write special tracing records into the experiment data file at runtime. Useful for large routines

- API calls are supported in both Fortran and C
- API works for sampling, tracing, and loop profiling as well as with the [perftools-lite*](#) modules
- When [perftools](#) module is loaded
 - `-I$CRYPAT_ROOT/include` is added to compiling flag (C header `pat_api.h`, Fortran header `pat_apif.h` and Fortran 77 header files, `pat_apif.h` and `pat_apif77.h` are available)
 - Macro `CRYPAT` and `-DCRYPAT` is added to compiling flag
- `int PAT_region_begin (int id, char *label)`
 - `id` is a unique identifier for the region
 - `label` is the description that will appear in profiling output
- `int PAT_region_end (int id)`
 - `id` must match begin call
- Fortran subroutines with extra final integer argument for return value similar to MPI.
- Data collection through API can be controlled with `PAT_RT_TRACE_API`
- For more information, see `pat_build` man page in the section APPLICATION PROGRAM INTERFACE

API for Adding User Instrumentation

- Main information is given in the [pat_build](#) man page in the section APPLICATION PROGRAM INTERFACE.
 - For further examples of Pertools API calls in source code, see the topic "API" in the [pat_help](#) system.
- When one of the [perftools](#) instrumentation modules is loaded, it defines a compiler macro called **CRAYPAT** which can be used to wrap headers and API calls.

C:

```
#ifdef CRAYPAT
PAT_region_begin ( 1, "loop" );
#endif
```

Fortran:

```
#ifdef CRAYPAT
call PAT_region_begin ( 1, "loop", istat );
#endif
```

- HWPC are collected for individual regions.
- The API works for sampling, tracing, and loop profiling as well as with the [perftools-lite*](#) modules.

PAT Regions Example in C and Fortran

C:

```
#ifdef CRYPAT
#include "pat_api.h"
#endif

...
#ifndef CRYPAT
PAT_region_begin( 1, "jacobi_part1");
#endif
// the execution of this code segment will
// appear in CrayPAT output as "jacobi_part1"
...
#ifndef CRYPAT
PAT_region_end( 1);
#endif
...
#ifndef CRYPAT
PAT_region_begin( 2, "jacobi_part2");
#endif
// the execution of this code segment will
// appear in CrayPAT output as "jacobi_part2"
...
#ifndef CRYPAT
PAT_region_end( 2);
#endif
...
```

Fortran:

```
#ifdef CRYPAT
#include "pat_apif.h"
#endif

...
#ifndef CRYPAT
Call PAT_region_begin( 1, "jacobi_part1", istat)
#endif
// the execution of this code segment will
// appear in CrayPAT output as "jacobi_part1"
...
#ifndef CRYPAT
Call PAT_region_end( 1, istat)
#endif
...
#ifndef CRYPAT
Call PAT_region_begin( 2, "jacobi_part2", istat)
#endif
// the execution of this code segment will
// appear in CrayPAT output as "jacobi_part2"
...
#ifndef CRYPAT
Call PAT_region_end( 2, istat)
#endif
```

Event Tracing w/ and w/o Perftools API (C)

Time%		Time		Imb.	Imb.	Calls	Group
		Time	Time%				Function
							PE=HIDE
100.0%	45.429778	--	--	3,336.0	Total		
82.6%	37.513233	--	--	757.0	USER		
69.9%	31.777804	7.855317	22.7%	150.0	#1.jacobi_part1		
9.9%	4.480465	0.484320	11.1%	150.0	#2.jacobi_part2		
2.8%	1.252451	0.066733	5.8%	1.0	initmt		
0.0%	0.001015	0.002053	76.5%	1.0	main		
0.0%	0.000479	0.000493	58.0%	150.0	sendp		
0.0%	0.000298	0.000075	22.9%	2.0	jacobi		
0.0%	0.000297	0.000026	9.2%	1.0	initcomm		
0.0%	0.000271	0.000211	50.0%	150.0	sendp3		
0.0%	0.000132	0.000255	75.2%	150.0	sendp2		
0.0%	0.000019	0.000000	2.8%	1.0	exit		
17.3%	7.843884	--	--	2,423.0	MPI		
16.8%	7.652373	2.858017	31.1%	450.0	MPI_Waitall		
0.4%	0.170341	0.033417	18.7%	900.0	MPI_Isend		
0.0%	0.016676	0.018946	60.8%	900.0	MPI_Irecv		
0.0%	0.001239	0.000039	3.4%	1.0	MPI_Cart_create		
0.0%	0.001214	0.000015	1.4%	152.0	MPI_Allreduce		
0.0%	0.000933	0.000022	2.6%	3.0	MPI_Type_commit		
0.0%	0.000644	0.000031	5.2%	3.0	MPI_Cart_shift		
0.0%	0.000459	0.000027	6.5%	3.0	MPI_Type_vector		
0.0%	0.000004	0.000002	34.5%	2.0	MPI_Barrier		

With API

Regions							
Time%		Time		Imb.	Imb.	Calls	Group
		Time	Time%				Function
100.0%	45.649773	--	--	3,036.0	Total		
83.4%	38.065287	--	--	457.0	USER		
80.6%	36.771025	7.132552	18.6%	2.0	jacobi		
2.8%	1.291914	0.077825	6.5%	1.0	initmt		
0.0%	0.001031	0.000933	54.3%	1.0	main		
0.0%	0.000520	0.000342	45.3%	150.0	sendp		
0.0%	0.000337	0.000028	8.7%	1.0	initcomm		
0.0%	0.000320	0.000264	51.6%	150.0	sendp3		
0.0%	0.000121	0.000322	83.1%	150.0	sendp2		
0.0%	0.000017	0.000001	3.9%	1.0	exit		
16.4%	7.500291	--	--	2,423.0	MPI		
16.0%	7.298155	4.192142	41.7%	450.0	MPI_Waitall		
0.4%	0.183874	0.048775	24.0%	900.0	MPI_Isend		
0.0%	0.014285	0.020122	66.8%	900.0	MPI_Irecv		
0.0%	0.001146	0.000013	1.3%	1.0	MPI_Cart_create		
0.0%	0.001003	0.000012	1.4%	152.0	MPI_Allreduce		
0.0%	0.000756	0.000012	1.7%	3.0	MPI_Type_commit		
0.0%	0.000566	0.000009	1.8%	3.0	MPI_Type_vector		
0.0%	0.000501	0.000015	3.3%	3.0	MPI_Cart_shift		
0.0%	0.000005	0.000002	34.1%	2.0	MPI_Barrier		
0.0%	0.000000	0.000000	15.4%	4.0	MPI_Wtime		
0.0%	0.000000	0.000000	18.7%	1.0	MPI_Cart_get		

Without API

Load Imbalance Analysis



Load Imbalance

Common cause for performance bottlenecks when running applications at scale

- Look for high imbalance time and percentage

- User functions

Imbalance time = Maximum time – Average time

- Synchronization (Collective communication and barriers)

Imbalance time = Average time – Minimum time

- Imbalance percentage of time that the rest of the team is not engaged in useful work on the given function

- Also look for MPI (SYNC) time.

- Measure for time spent waiting in collectives

- Only available with event tracing experiments

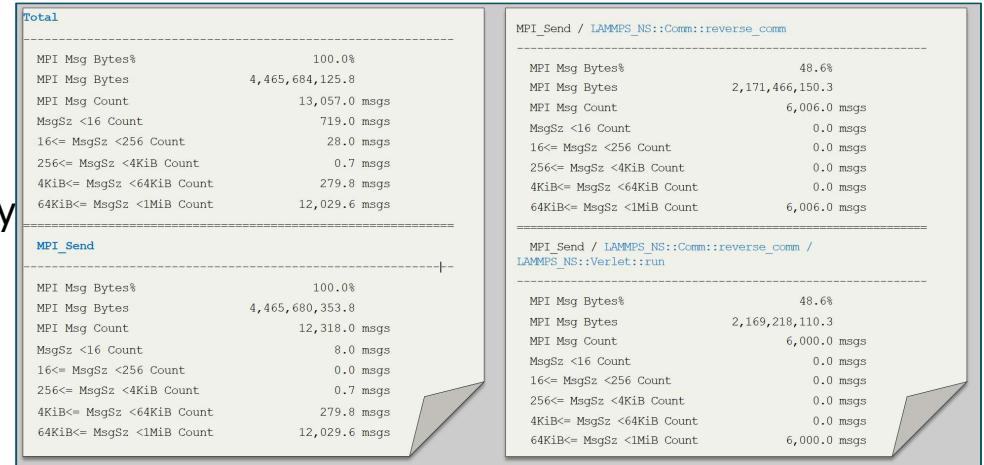
- Guidance on rank reordering for better load balance might appear in report
(MPI Utilization: ...)

Table 1: Profile by Function Group and Function						
Time%	Time	Imb. Time	Imb. Time%	Calls	Group	Function
PE=HIDE						
100.0%	1.957703	--	--	42,970.8	Total	
60.0%	1.174021	--	--	3,602.0	USER	
30.8%	0.603850	0.176924	23.0%	1,198.0	function3	
19.2%	0.375117	0.128748	26.0%	1,200.0	function2	
9.1%	0.178111	0.081880	32.0%	1,200.0	function1	
36.0%	0.704928	--	--	9,613.0	MPI_SYNC	
25.8%	0.505174	0.385130	76.2%	9,596.0	mpi_barrier_(sync)	
10.2%	0.199537	0.199518	100.0%	1.0	mpi_init_(sync)	
4.0%	0.078736	--	--	29,754.8	MPI	
2.3%	0.045351	0.003531	7.3%	9,596.0	MPI_BARRIER	
1.1%	0.021520	0.051295	71.6%	8,756.9	MPI_ISEND	

Communication Bottlenecks

- Sort messages by caller
 - Available in default report for tracing (lite-events) experiments
- Analyze message sizes
 - Useful when tuning MPI env vars (eager, rendez-vous, ...)
 - Found in default report
- Put barrier in front of collectives to filter sync times
- Use rank reordering for max. on-node communication
 - Look for guidance and instructions in report
 - [grid_order](#) utility
- Visualizing data
 - Apprentice2 ([app2 app.exe+*/](#)) imbalance or activity report
 - [pat_report -s pe=ALL](#) shows data by MPI rank

MPI	MPI Msg	MPI Msg	MsgSz	4KiB<=	Function
Msg	Bytes	Count	<16	MsgSz	Caller
Bytes%			Count	<64KiB	PE=[mmm]
100.0%	34,940,767.4	8,771.9	258.6	8,513.3	Total
-----	-----	-----	-----	-----	-----
100.0%	34,940,647.4	8,756.9	243.6	8,513.3	MPI_ISEND
-----	-----	-----	-----	-----	-----
56.2%	19,622,700.0	4,837.5	56.2	4,781.2	calc2_
3					shallow_
-----	-----	-----	-----	-----	-----
4 56.4%	4 19,718,400.0	4 7,200.0	4 2,400.0	4 4,800.0	pe.0
4 56.4%	4 19,699,200.0	4 4,800.0	4 0.0	4 4,800.0	pe.32
4 42.3%	4 14,784,000.0	4 4,800.0	4 1,200.0	4 3,600.0	pe.63
4 42.3%	4 14,784,000.0	4 4,800.0	4 1,200.0	4 3,600.0	pe.63
4 42.3%	4 14,784,000.0	4 4,800.0	4 1,200.0	4 3,600.0	pe.63



Hardware Performance Counters (HWPC)



Hardware Performance Counters

- Perftools supports the use of hardware counters to collect hardware events
 - All counters accessed through the PAPI interface.
 - Predefined sets of hardware counters are specified that can be instrumented for performance analysis experiment.
 - Number of simultaneous counters limited by hardware.
- Perftools provides information at the function call level on hardware features like caches, vectorization and memory bandwidth.
 - Very useful feature for deep understanding of application performance bottlenecks.
 - Impact of compiler options and code optimization.
- HWPC collection can slow down the execution notably.
 - Should be used within a tracing experiment only for a small set of functions or ideally through an automatic performance analysis.
- Only LUMI, only a limited number of hardware counters are available:
 - Use `papi_avail` on a compute node to get a list

When To Collect Hardware Performance Counters

- Use to understand the “why” of a bottleneck.
- Default set of CPU counters are already collected for whole program
 - Used to present memory and vector summary metrics
- To collect performance counters
 - Set `PAT_RT_PERFCTR` environment variable to list of events or group prior to execution.
(Or use `-Drtenv=PAT_RT_PERFCTR=<event list> | <group>` for `pat_build`.
Environment variable has priority.)
- Run the following utility on a compute node to get list of events for a processor:
 - `papi_native_avail`
 - `papi_avail`
- Use `pat_help` to see counter groups and derived metrics
 - `$> pat_help counters processor_type deriv`
 - Example for LUMI-C: `$> pat_help counters trento deriv`

Total			
Time%		100.0%	
Time		70.507153 secs	
Imb. Time		-- secs	
Imb. Time%		--	
Calls	5.744 /sec	405.0 calls	
PAPI_BR_TKN	0.064G/sec	4,506,524,243.125 branch	
PAPI_TOT_INS	12.154G/sec	856,922,640,377.375 instr	
PAPI_BR_INS	0.124G/sec	8,718,726,704.750 branch	
PAPI_TOT_CYC		244,793,361,310.500 cycles	
Instr per cycle		3.50 inst/cycle	
MIPS	97,229.58M/sec		
Average Time per Call		0.174092 secs	
CrayPat Overhead : Time	0.0%		

pat_run

Launch a dynamically-linked program instrumented for performance analysis



Profile Existing Dynamically Linked Binaries

```
$> srun -n 16 pat_run ./app.exe  
$> pat_report app.exe+*/ > my_report
```

- Insert `pat_run` before executable. **No instrumentation needed**
- Useful if source code is not accessible but program is dynamically linked

```
$> export PAT_RT_PERFCTR=1  
$> srun -n 8 pat_run ./app.exe
```

- Use existing perftools capability
- Optionally collect a different group of performance counters

```
$> pat_report -P -O callers+src app.exe+*/ > my_callers_report
```

- Create additional views of the data with `pat_report` options
- If at least object files and libraries are available
 - load the `perftools-preload` module before relinking
 - more information like line number hot spots are given

Final notes and Recap on Perftools



Perftools-lite modules vs Perftools module

- **Perftools-lite modules : "novice" approach**

- Little/no knowledge of the application and do not want to dive deeper into the source code or profiling tools
- **Load the -lite module, make, run, analysis of automatically generated report**
 - Sampling `perfetto-lite`, tracing `perfetto-lite-events`, loop work estimate `perfetto-lite-loops`, GPU `perfetto-lite-gpu`
- Everything is automatic
 - Compilation and run commands are the same : **instrumented binary has the same name as the original one**
 - Report automatically generated

- **Perftools module : "advanced" approach**

- Good knowledge of the application and the used libraries (MPI, openMP, gpu, io and more)
- **module load perftools, make, pat_build opt, run, pat_report, analysis of generated report**
 - Sampling `opt=-S`; tracing `opt=-u`, loop work estimate `opt=-w`, GPU `opt=-g omp/cuda/hip`
 - **AND MUCH MORE** `opt=-g mpi,io,blas,lapack,netcdf...`
- Everything is generated by the user
 - The original binary must be instrumented with `pat_build` and the options corresponding to the analysis to be performed
 - Run command must use **the name of the new instrumented binary generated by the pat_build command**
 - Report must be generated with `pat_report`

Summary

	Sampling	Event Tracing	Loop Work Estimates	APA	Step
perftools-lite	Is the default experiment.	Available through perftools-lite-events	Available through perftools-lite-loops	N/A	1
	Rebuild for all except pat_run. Rerun for all.				2
perftools			Rebuild with -h profile_generate for Fortran or -finstrument-loops for C. Only for Cray Compiler	Is the default experiment	0
	pat_build ...	pat_build [-w -u] [-g]	pat_build -w ...	pat_build ...	1
	For all: Run app.exe+pat				2
	For all: pat_report app+pat+*/				3
					pat_build -O *.apa
					Run app.exe+apa
					pat_report <apa>.xf

Summary

	Sampling	Event Tracing	Loop Work Estimates	APA
Reveal	N/A	N/A	Need a program library obtained with the <code>-hpl=<app>.pl</code> compiler option for Fortran or <code>-fcray-program-library-path=<app.pl></code> for C and optionally an <code><app>+*/</code> directory from a loop profiling experiment (Cray compiler only)	N/A
Hardware performance counters	Can be enabled via <code>-Drtenv=PAT_RT_PERFCTR=<event list> <group></code> option for <code>pat_build</code> or the environment variable <code>PAT_RT_PERFCTR</code> . For <code>perftools-lite*</code> only the environment variable is applicable. Note that for <code>perftools-lite</code> some counters are already collected.			
Apprentice2	Can be applied to all <code><app>+*/</code> directories			
pat_run	When you can't compile (only dynamically linked binary or objects and libraries are available).			
pat_info	Can be applied to all <code><app>+*/</code> directories			

What does perftools support?

Cray Performance Tools Components

Reveal
compiler optimization presentation
OpenMP
memory activity tracking
Apprentice2 (app2)
CrayPat runtime
perftools / perftools-lite experiments
pre-defined vs custom instrumentation
sampling
tracing with runtime summarization
predefined trace groups
full trace
pat_build
pat_report
pat_run
pat_region API
grid_order
pat_view
pat_help
Performance counters
PAPI
CPU, GPU, network, energy
default, predefined groups, individual events

Programming Models

MPI
OpenMP
CUDA
OpenACC / OpenMP 4.5
PGAS (upc, Fortran coarrays)
SHMEM
OpenCL
HIP

Distros

CLE
Mac
Windows
RH/Centos
SLES

Languages

Fortran
C
C++
Chapel

Platforms

Cray XC, CS, Shasta
HPE Apollo 2000 Gen10Plus, Apollo 80*
Mac (app2, reveal), Windows (app2)

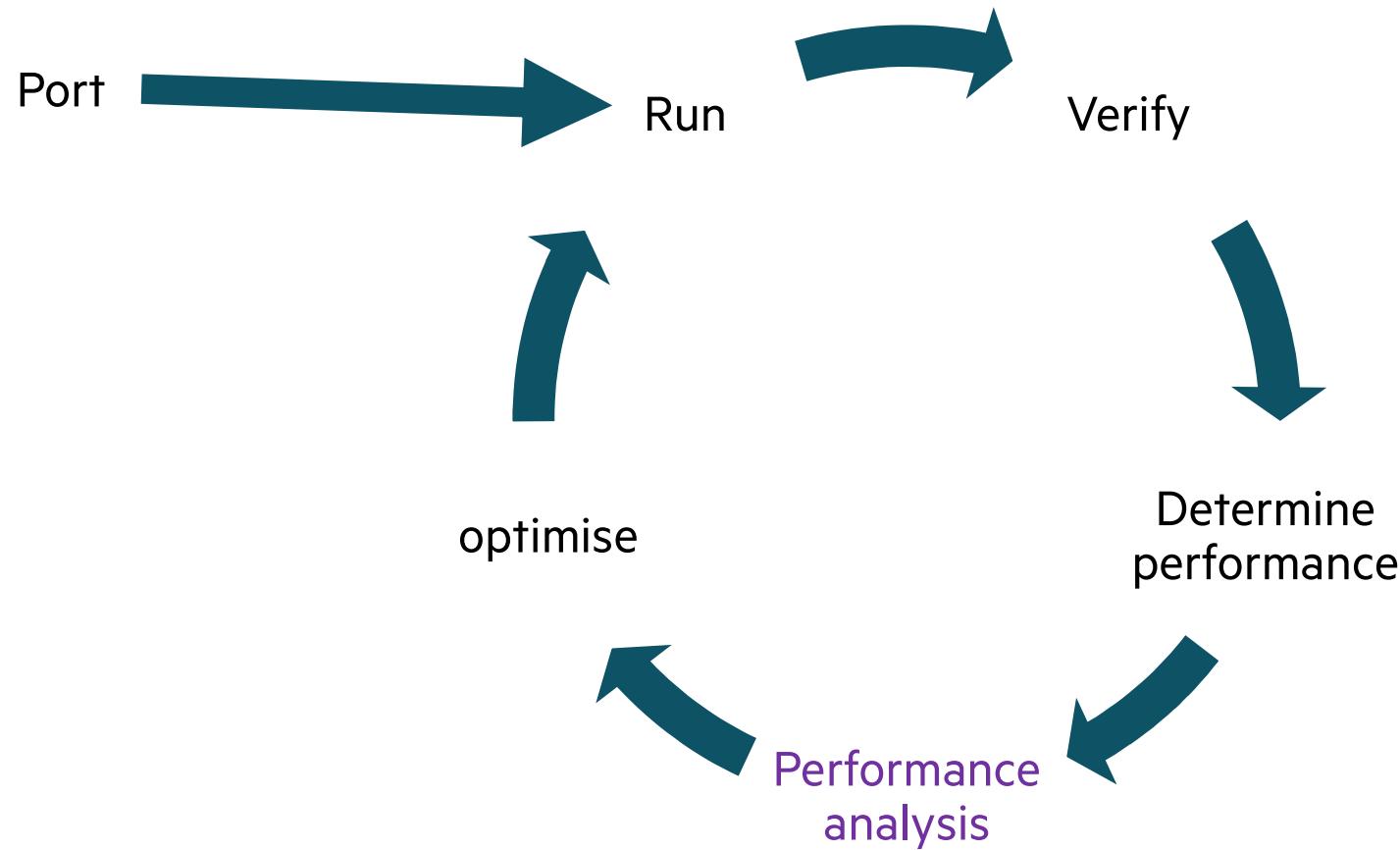
Compilers

CCE
GCC
Intel
PGI
AMD (aocc, hipclang)

Processors

Intel (SNB, IVB, HSW, KNL, BDW, [SKL](#))
[ARM \(Cavium TX2\)](#)
AMD (Rome, Milan, MI60/MI100, MI200)
NVIDIA (Pascal, Volta)

Optimisation Cycle



Questions?

