

# Profiling and debugging

Mandes Schönherr



# Compiler feedback

- Compiler can print annotated intermediate source files

source with line numbers

```
270. + 1 b-----< DO k = 2,kmax-1
271. + 1 b b-----< DO j = 2,jmax-1
272. 1 b b Vr3-----< DO i = 2,imax-1
273. 1 b b Vr3      S0 = a(i,j,k,1)* ...
```

additional messages below

loop handling  
annotations

- Generate listing files

- CCE:

```
FLAGS+=-hlist=a
```

Produces commentary files <stem>.lst

- PGI:

```
FLAGS+=-Minfo
```

Feedback to STDERR

- Compiler teams work very hard to make feedback useful
- advice: use it, it's free!  
(i.e. no impact on performance)



# Example Loop mark

**G** = accelerated

**g** = partioned

**b** = blocked loops

**+** = additional message below

```
269.  + 1 G-----< !$omp target teams distribute reduc...
270.    1 G g-----<      DO k = 2,kmax-1
271.  + 1 G g b-----<      DO j = 2,jmax-1
272.    1 G g b gb---<      DO i = 2,imax-1
273.    1 G g b gb      S0 = a(i,j,k,1)*p(i+1,j,k) &
... 
```

ftn-6405 ftn: ACCEL File = himeno\_F\_v01.F90, Line = 269

A region starting at line 269 and ending at line 293 was placed on the accelerator.

# Runtime commentary

- Cray runtime commentary

- Compile with CCE (no special options)
- Set environment variable:  
**CRAY\_ACC\_DEBUG=2**
- Run the executable
- The commentary is written to STDERR

- **Advice:**

- Don't set both:  
**CRAY\_ACC\_DEBUG**  
with  
**COMPUTE\_PROFILE**

- NVIDIA Compute Profiler

- Event-by-event timing information
- Compile with CCE (no special options)
- Set environment variable:  
**COMPUTE\_PROFILE=1**
- Run the executable
- The commentary is written to file **cuda\_profile\_0.log**

```
ACC: Start transfer 16 items from himeno_F_v01a.F90:153
ACC:      allocate, copy to acc 'a' (136855584 bytes)
...
ACC:      allocate reusable <internal> (1008 bytes)
ACC: End transfer (to acc 444780680 bytes, to host 0 bytes)
ACC: Execute kernel jacobi_$ck_L153_1 blocks:126 threads:
      128 async(auto) from himeno_F_v01a.F90:153
ACC: Wait async(auto) from himeno_F_v01a.F90:153
```

```
$> cat cuda_profile_0.log
...
method=[ memcpyHtoD ] gputime=[ 22636.031 ] cputime=[ 22714.756 ]
method=[ memcpyHtoD ] gputime=[ 16801.504 ] cputime=[ 16814.359 ]
method=[ memcpyHtoD ] gputime=[ 1.088 ] cputime=[ 7.279 ]
...
method=[ jacobi_$ck_L153_1 ] gputime=[ 3737.728 ]
      cputime=[ 15.326 ] occupancy=[ 0.312 ]
method=[ memcpyDtoH ] gputime=[ 2.400 ] cputime=[ 19.159 ]
method=[ memcpyDtoH ] gputime=[ 15007.552 ] cputime=[ 15675.952 ]
```

COMPUTE

STORE

ANALYZE

# CrayPAT

Evaluate program behavior on Cray  
supercomputer systems ...



## Sampling

### Advantages

- Only need to instrument main routine
- Low Overhead – depends only on sampling frequency
- Smaller volumes of data produced

### Disadvantages

- Only statistical averages available
- Limited information from performance counters

## Tracing/Instrumentation

### Advantages

- More accurate and more detailed information
- Data collected from every traced function call not statistical averages

### Disadvantages

- Increased overheads as number of function calls increases
- Huge volumes of data generated

**The best approach is *guided tracing*.**  
**e.g. Only tracing functions that are not small (i.e. very few lines of code) and contribute a lot to application's run time.**

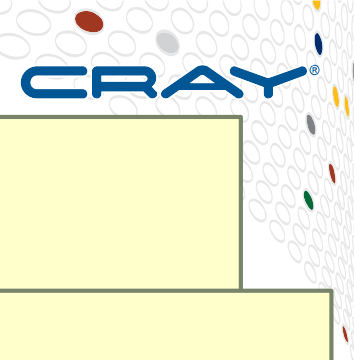
**APA is an automated way to do this.**

COMPUTE

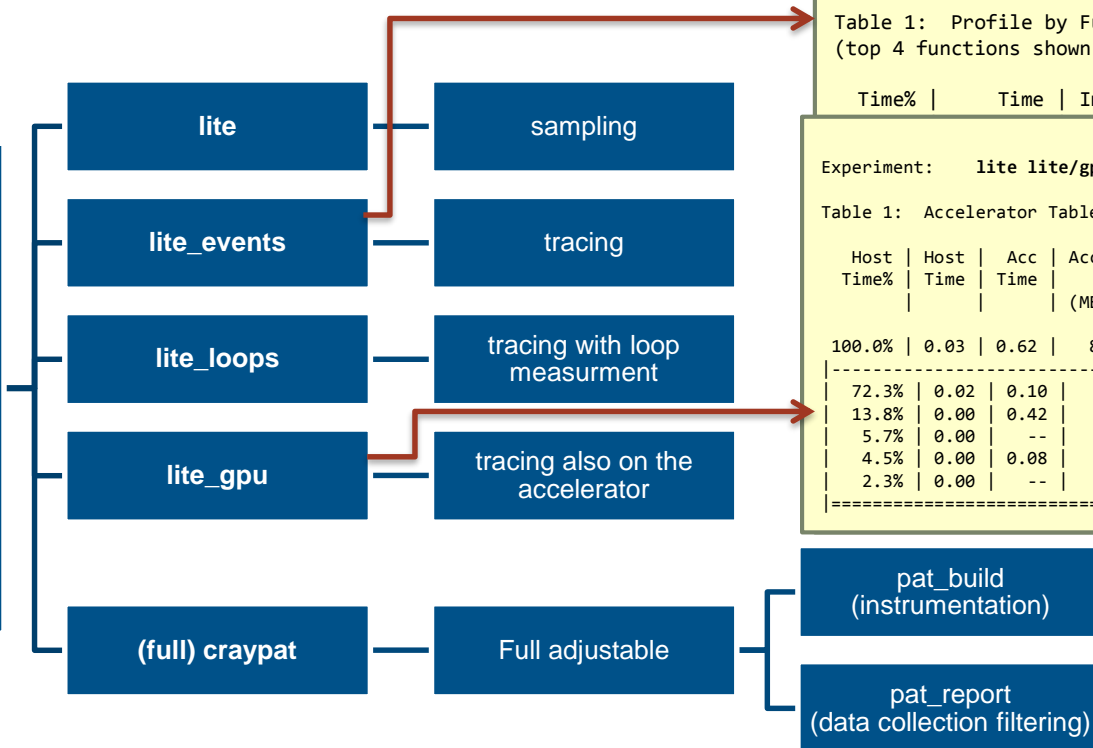
| STORE

| ANALYZE

# Profiling the application



## Perftools



Experiment: `lite lite/event_profile`

Table 1: Profile by Function Group and Function  
(top 4 functions shown)

Time%	Time	Imb.	Imb.	Calls	Group
-------	------	------	------	-------	-------

Experiment: `lite lite/gpu`

Table 1: Accelerator Table by Function (top 10 functions shown)

Host Time%	Host Time	Acc Time	Acc Copy In (MBytes)	Acc Copy Out (MBytes)	Events	Function=[max10] PE=HIDE Thread=HIDE
100.0%	0.03	0.62	848.36	65.26	1,038	Total
72.3%	0.02	0.10	424.18	32.63	402	main_.ACC_COPY@li.174
13.8%	0.00	0.42	--	--	200	main_.ACC_ASYNC_KERNEL@li.174
5.7%	0.00	--	--	--	201	main_.ACC_SYNC_WAIT@li.174
4.5%	0.00	0.08	424.18	32.63	14	main_.ACC_COPY@li.153
2.3%	0.00	--	--	--	200	main_.ACC_REGION@li.174

COMPUTE

STORE

ANALYZE

# Profiling the application



- More details of CrayPAT lite output

- Reanalyze the data using pat\_report file and e.g. -T option

```
$> pat_report -T *.ap2
```

```
...
Experiment:  lite lite/gpu
```

**Table 1: Profile by Function Group and Function**

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function Thread=HIDE
100.0%	1.273765	--	--	9,763.0	Total
86.2%	1.098388	--	--	8,714.0	ETC
33.5%	0.426703	--	--	208.0	__cray_acc_hw_wait
20.2%	0.257243	--	--	1.0	_END
17.3%	0.220493	--	--	11.0	__cray_acc_hw_init
12.0%	0.153406	--	--	837.0	__cray_acc_hw_copy_host_to_acc
13.8%	0.175323	--	--	1,045.0	USER
9.9%	0.126006	--	--	1.0	initmtm.LOOP@li.212

```
...
Experiment:  lite lite/gpu
```

**Table 3: Time and Bytes Transferred for Accelerator Regions**

Host Time%	Host Time	Acc Time	Acc Copy In (MBytes)	Acc Copy Out (MBytes)	Events	Calltree Thread=HIDE
100.0%	0.01	0.60	848.36	65.26	1,038	Total
100.0%	0.01	0.60	848.36	65.26	1,038	main_
						jacobi_
3	90.7%	0.00	0.44	0.00	1,030	jacobi_.REGION@li.279
4						jacobi_
3	9.3%	0.00	0.16	848.35	65.26	8  jacobi_.ACC_DATA_REGION@li.276
4	8.6%	0.00	0.15	848.35	--	2  jacobi_.ACC_COPY@li.276
4	0.5%	0.00	0.01	--	65.26	2  jacobi_.ACC_COPY@li.329
4	0.2%	0.00	--	--	--	2  jacobi_.ACC_DATA_REGION@li.276..
4	0.1%	0.00	--	--	--	2  jacobi_.ACC_SYNC_WAIT@li.329

COMPUTE

STORE

ANALYZE



# Profiling using (fully) CrayPAT

```
$> module load perftools-base
$> module load perftools
$> pat_build <exe>
```

- Instrumentation
- set what/how is measured
- Create new executable <exe>+pat

## Run job with new executable

```
$> pat_report *.xf
```

- Measurement data analysis
- Generates ap2 (compressed) file and text report
- ap2 is input for pat\_report, Apprentice<sup>2</sup>, Reveal

Option	Description
	Sampling profile
-w	Tracing is default experiment
-u	tracing of functions in user source files
-T <func>	Specifies a function which will be traced
-t <file>	Tracing all functions in the specified file
-g <group>	Instrument functions in group, e.g. omp, mpi, cuda, blas, io, netcdf, syscall

Option	Description
-O <table opt>	Variety of table options
-sfilter-input=...	Defines filter, e.g. 'pe%2==0'
-o	Defines report output file name
-f <format>	Output format, e.g. plot, rpt, ap2, ap2-xml, ap2-txt, xf-xml, xf-txt, html

# Interlude: A synchronous profile



- GPU kernels launch asynchronously

- So the compute time shows up in the SYNC\_WAIT events

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function
100.0%	0.755941	--	--	1131.0	Total
100.0%	0.755749	--	--	730.0	USER
40.9%	0.308847	--	--	1.0	main_
28.4%	0.214851	--	--	103.0	jacobi_.ACC_SYNC_WAIT@li.320
24.1%	0.182321	--	--	2.0	jacobi_.ACC_COPY@li.280
4.3%	0.032620	--	--	103.0	jacobi_.ACC_COPY@li.293
1.0%	0.007862	--	--	2.0	jacobi_.ACC_COPY@li.340

- We can switch off the automatic asynchronicity

- This can give a clearer profile, but it may be skewed
- Recompile with CCE flag **-hacc\_model=auto\_async\_none**
- And do a new CrayPAT profile

avoid synchronization in profiling

# Combining Sampling and Tracing: APA



- **Automatic Profiling Analysis:**

- **Target:** large, long-running program (general a trace will inject considerable overhead)
- **Goal:** limit tracing to those functions that consume the most time.
- **Procedure:** use a preliminary sampling experiment to determine and instrument functions consuming the most time
- Note: Accelerator performance data for sampling experiments is not supported

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

```
$> pat_build <exe>
```

- The APA is the default experiment. No option needed.
- A new executable is created with a name like <exe>+pat

```
$> qsub job.pbs #aprun ... <exe>+pat  
$> pat_report -o myrep.txt <exe>+pat+*
```

- Applying pat\_report to the \*.xf generates an \*.apa file in addition to the \*.ap2 file.

```
$> vi *.apa
```

- The \*.apa file contains instructions for the next instrumentation step.
- Modify it according to your needs.

```
$> pat_build -O *.apa
```

- Generates an instrumented binary <exe>+apa for tracing

```
$> qsub job.pbs #aprun ... <exe>+apa  
$> pat_report -o myrep.txt <exe>+apa+*
```

- Applying pat\_report to the \*.xf generates a new \*.ap2 file.

COMPUTE

STORE

ANALYZE

# Loop work estimates



- **Assess suitability of loop nests for porting to the GPU**
  - Requires information on inclusive time spent in the loop nests and typical trip count of the loops.
  - CrayPAT can generate this information via a special kind of tracing experiment.
- **Load the performance tools**

```
$> module load perftools-base  
$> module load perftools
```
- **Recompile your program for gathering loop statistics**

```
$> ftn -c -h profile_generate my_program.f  
$> ftn -o my_program -h profile_generate my_program.o
```
- **Instrument the application for tracing (APA also possible)**

```
$> pat_build -w[-u] my_program
```
- **This generates a new binary named `my_program+pat`.**
- **Execute the instrumented program.**

```
> aprun -n #PE ./my_program+pat
```
- **This generates one or more raw data files(s) in .xf format. Process the raw data files(s) for use by Reveal.**

```
> pat_report -o report.txt my_program*.xf
```

  - This generates a performance data file `my_program.ap2` and text report `report.txt`.
- **Reveal can use the `my_program.ap2` to visualize time expensive loops.**
- **It is recommended to turn off OpenMP and OpenACC for the loop work estimates via `-h noomp -h noacc`**



# Profiling accelerated codes

- CrayPAT tracing offers a powerful profiling for accelerated codes. (Sampling does not collect GPU performance data)
- Load the GPU module and the performance tools
  - > `module load craype-accel-nvidia60`
  - > `module load perftools-base perftools`
- Recompile your program
  - > `ftn -c my_program.f`
  - > `ftn -o my_program my_program.o`
- Instrument the application for tracing and execute
  - > `pat_build -w my_program`
  - > `aprun -n pes my_program`
- Generate a report out of the raw data file(s)
  - > `pat_report -o report.txt my_porgram*.xf`



# Contents of report.txt (Table 1 and 2)

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
					Thread=HIDE
100.0%	1.222670	--	--	1,046.0	Total
-----					
79.0%	0.965458	--	--	1,043.0	USER
-----					
34.5%	0.421250	--	--	201.0	main_.ACC_SYNC_WAIT@li.177
24.0%	0.293347	--	--	1.0	main_.ACC_COPY@li.148
17.2%	0.209976	--	--	1.0	main_
=====					
21.0%	0.257213	--	--	3.0	ETC
-----					

Table 2: Load Imbalance by Thread

Max.	Imb.	Imb.	Thread
Time	Time	Time%	
1.222704	--	--	Total
-----			
1.222704	--	--	thread.0
=====			

- ACC\_COPY lines report data movements
- ACC\_KERNEL is essentially zero as it is launched asynchronously.
- GPU time is measured in the ACC\_SYNC\_WAIT
- ACC\_REGION measures internal ops (set up pointers for transfer.)



# Contents of report.txt (Table 3)

Table 3: Time and Bytes Transferred for Accelerator Regions

Host	Host	Acc	Acc Copy	Acc Copy	Events	Calltree
Time%	Time	Time	In	Out		Thread=HIDE
			(MBytes)	(MBytes)		
100.0%	0.76	0.52	424.18	0.00	1,042	Total
-----						
100.0%	0.76	0.52	424.18	0.00	1,042	main_
						main_.ACC_DATA_REGION@li.148
-----						
3	59.3%	0.45	0.43	0.01	0.00	1,004  main_.ACC_DATA_REGION@li.177
4	59.3%	0.45	0.43	0.01	0.00	1,000  main_.ACC_REGION@li.177
-----						
5	55.8%	0.42	--	--	--	200  main_.ACC_SYNC_WAIT@li.177
5	3.1%	0.02	0.01	0.01	0.00	400  main_.ACC_COPY@li.177

- Table shows details on data transfers and timings for CPU and GPU.
- ACC\_SYNC\_WAIT time on CPU includes kernel time on GPU.
- For MPI programs the statistics are averaged over the PE.

# Profiling with GPU Hardware Counters



- CrayPAT supports a wide range of accelerator performance counter
- A predefined set of groups has been created for ease of use (combines events that can be counted together.)
  - > `module load perftools-base perftools`
  - > `man accpc`
  - > `more $CRAYPAT_ROOT/share/CounterGroups.nvidia_k20x`
- Enable collection similarly to CPU counter collection
  - `PAT_RT_PERFCTR=group or events`
- Set the `PAT_RT_ACCPC` variable appropriately and run the instrumented (tracing) program.



# Apprentice<sup>2</sup>

Graphical representation  
of  
performance data

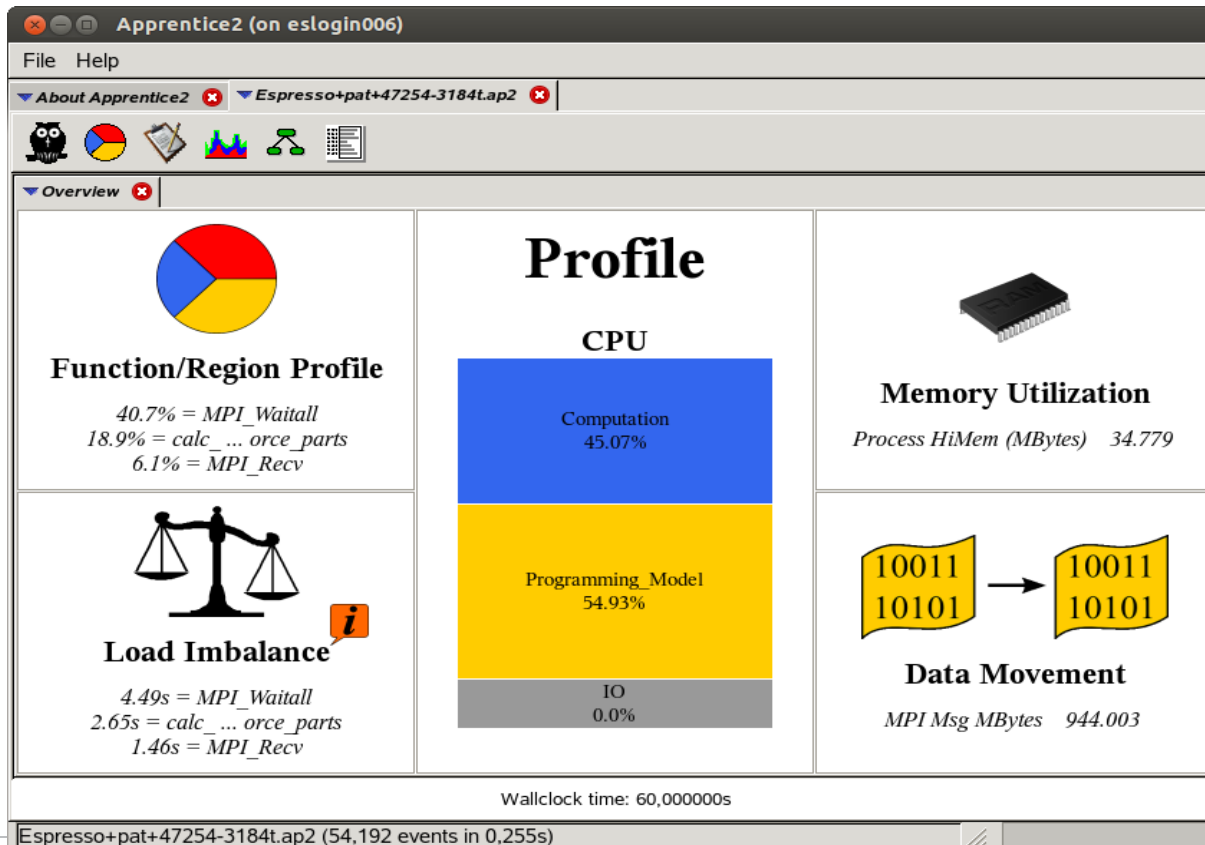
# Cray Apprentice<sup>2</sup>



- **Cray Apprentice<sup>2</sup> is a post-processing performance data visualization tool. Takes \*.ap2 files as input.**
- **Main features are**
  - Call graph profile
  - Communication statistics
  - Time-line view for Communication and IO.
  - Activity view
  - Pair-wise communication statistics
  - Text reports
- **helps identify:**
  - Load imbalance
  - Excessive communication
  - Network contention
  - Excessive serialization
  - I/O Problems

```
$> module load perftools-base  
$> app2 my_program.ap2 &
```

# Cray Apprentice<sup>2</sup>



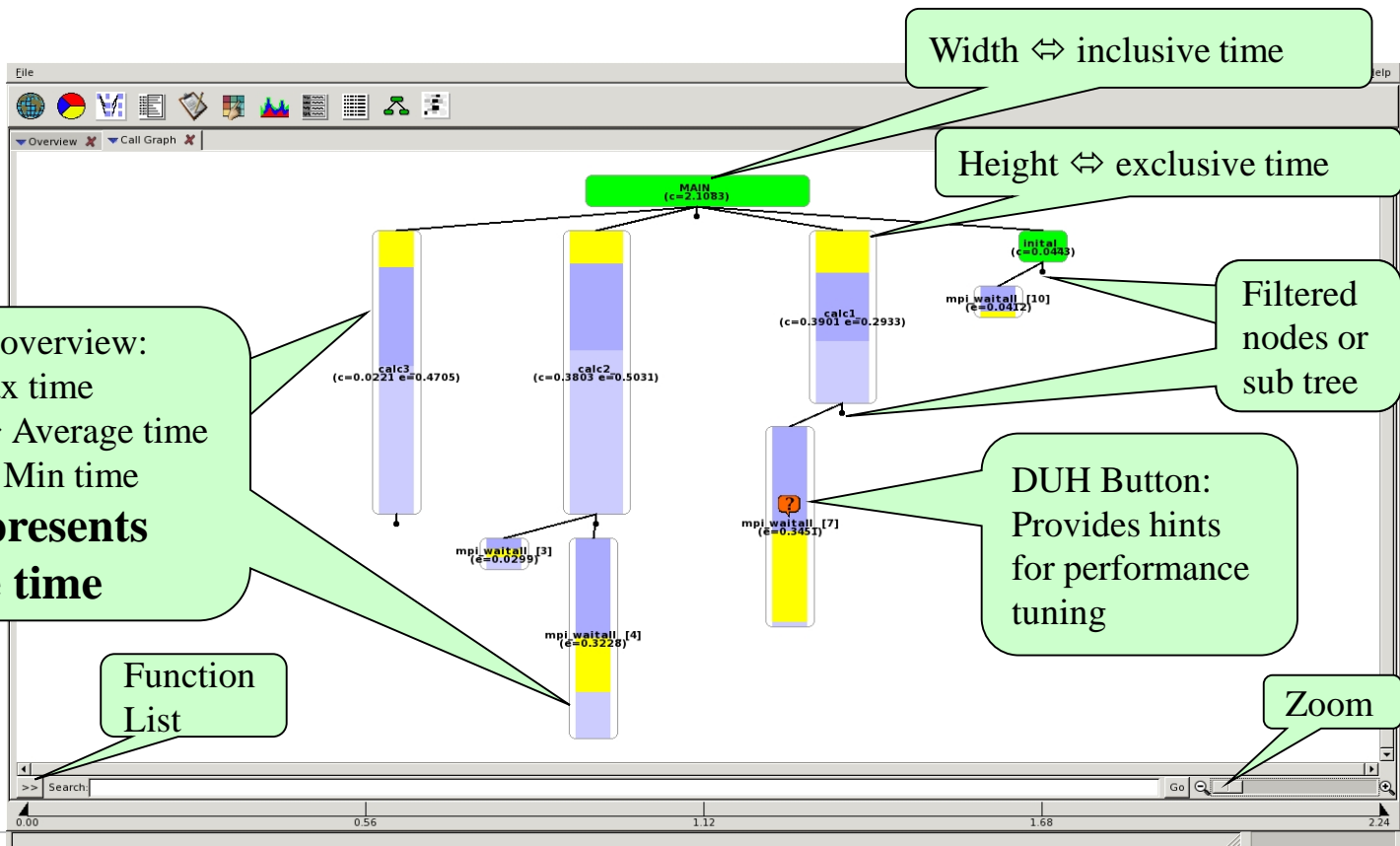
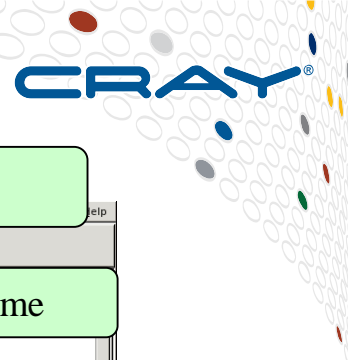
COMPUTE

STORE

ANALYZE

Copyright 2017 Cray Inc.

# Call Tree View

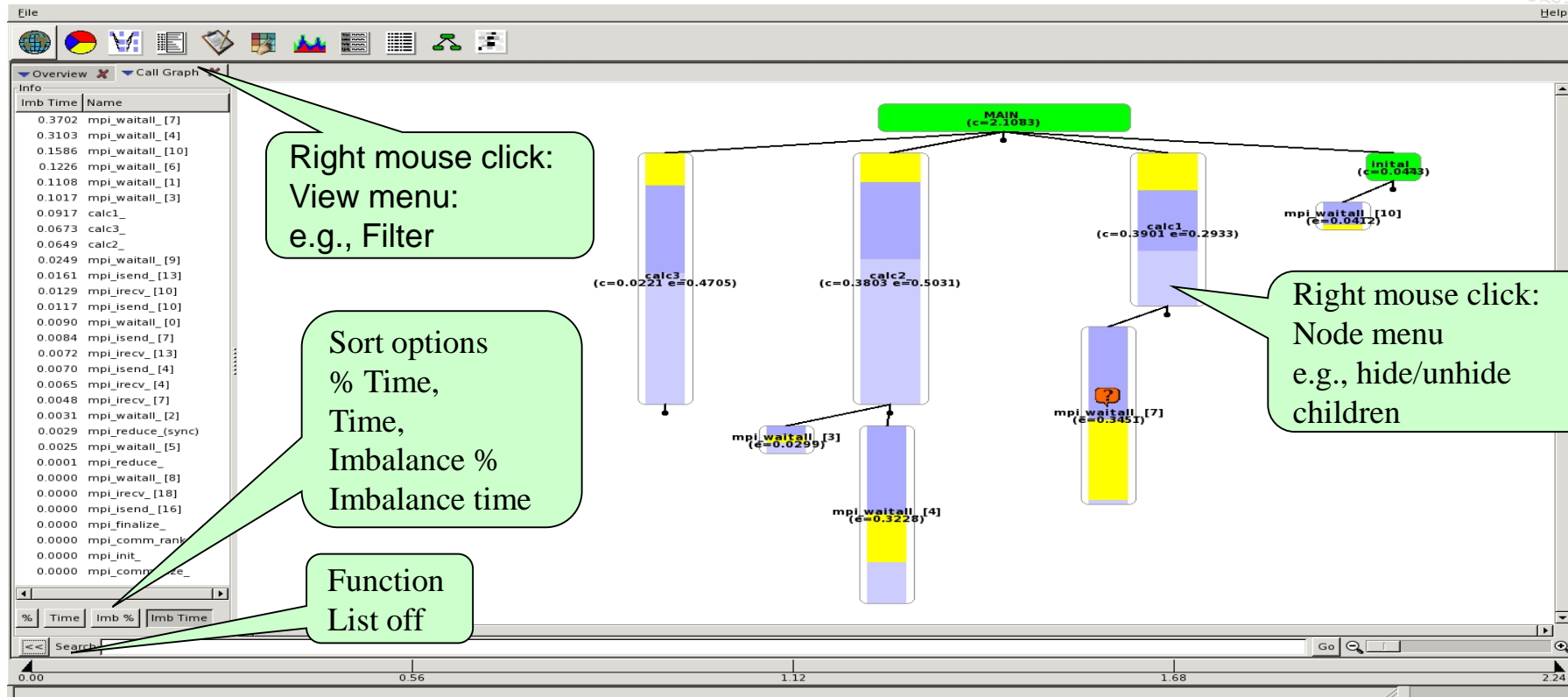


COMPUTE

STORE

ANALYZE

# Call Tree View – Function List

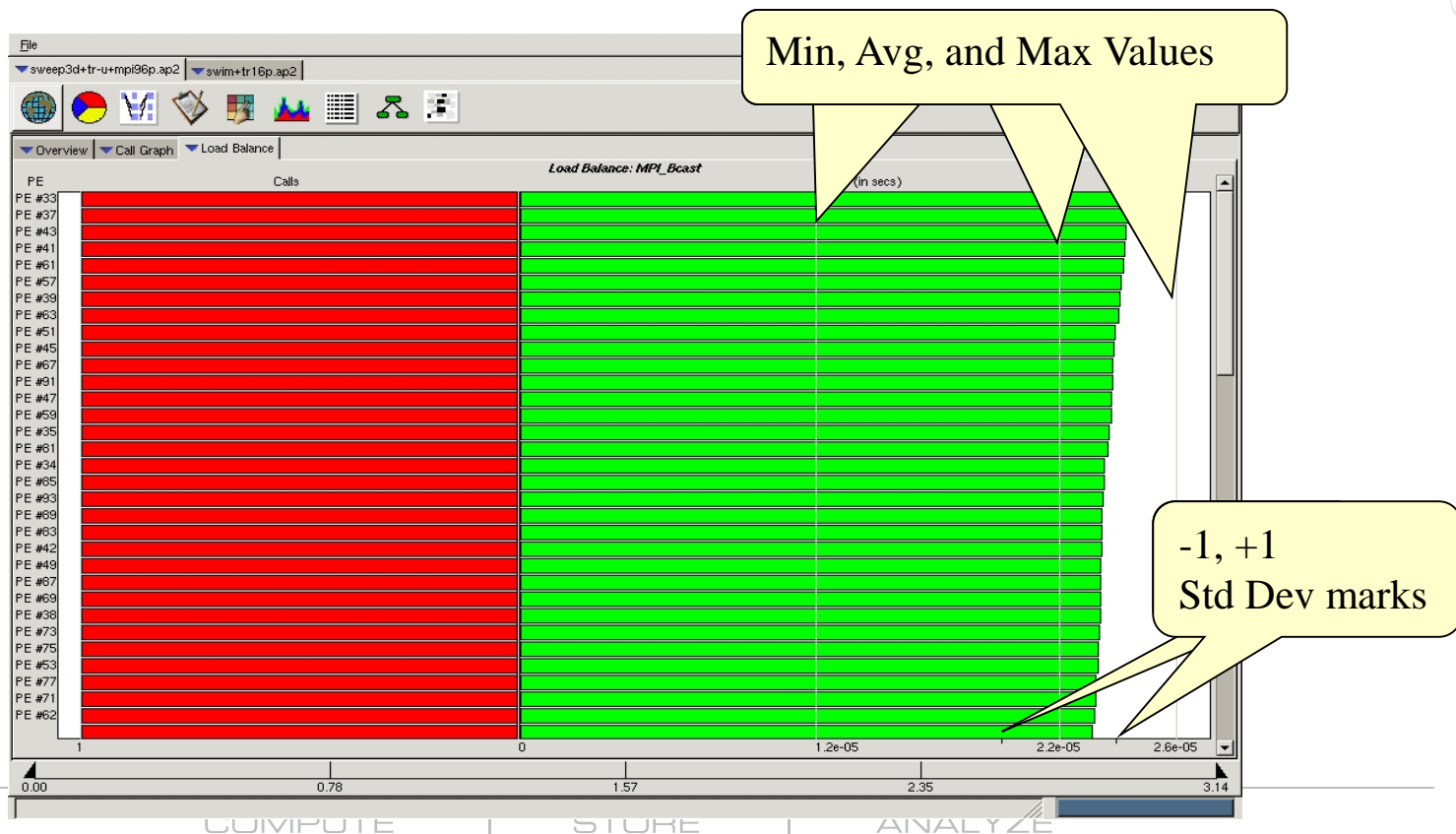
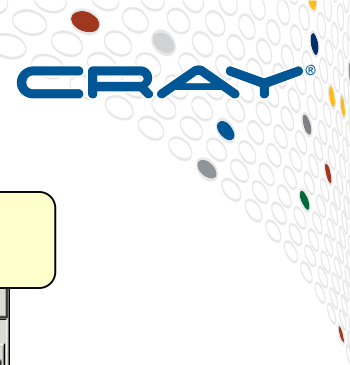


COMPUTE

STORE

ANALYZE

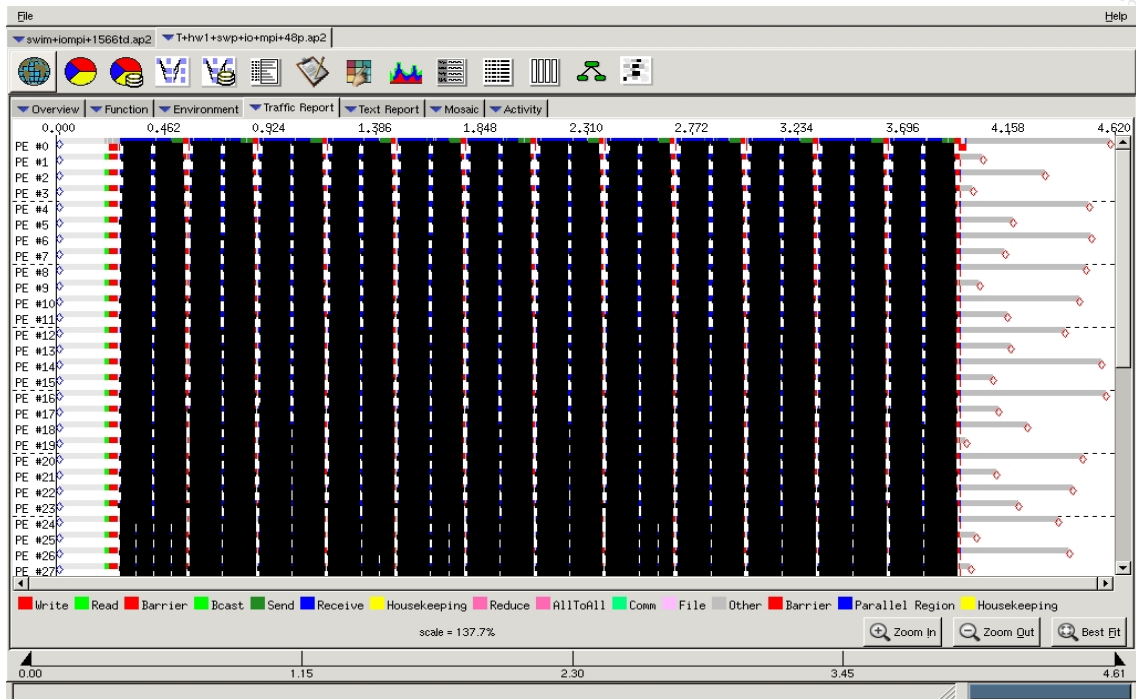
# Load Balance View (from Call Tree)



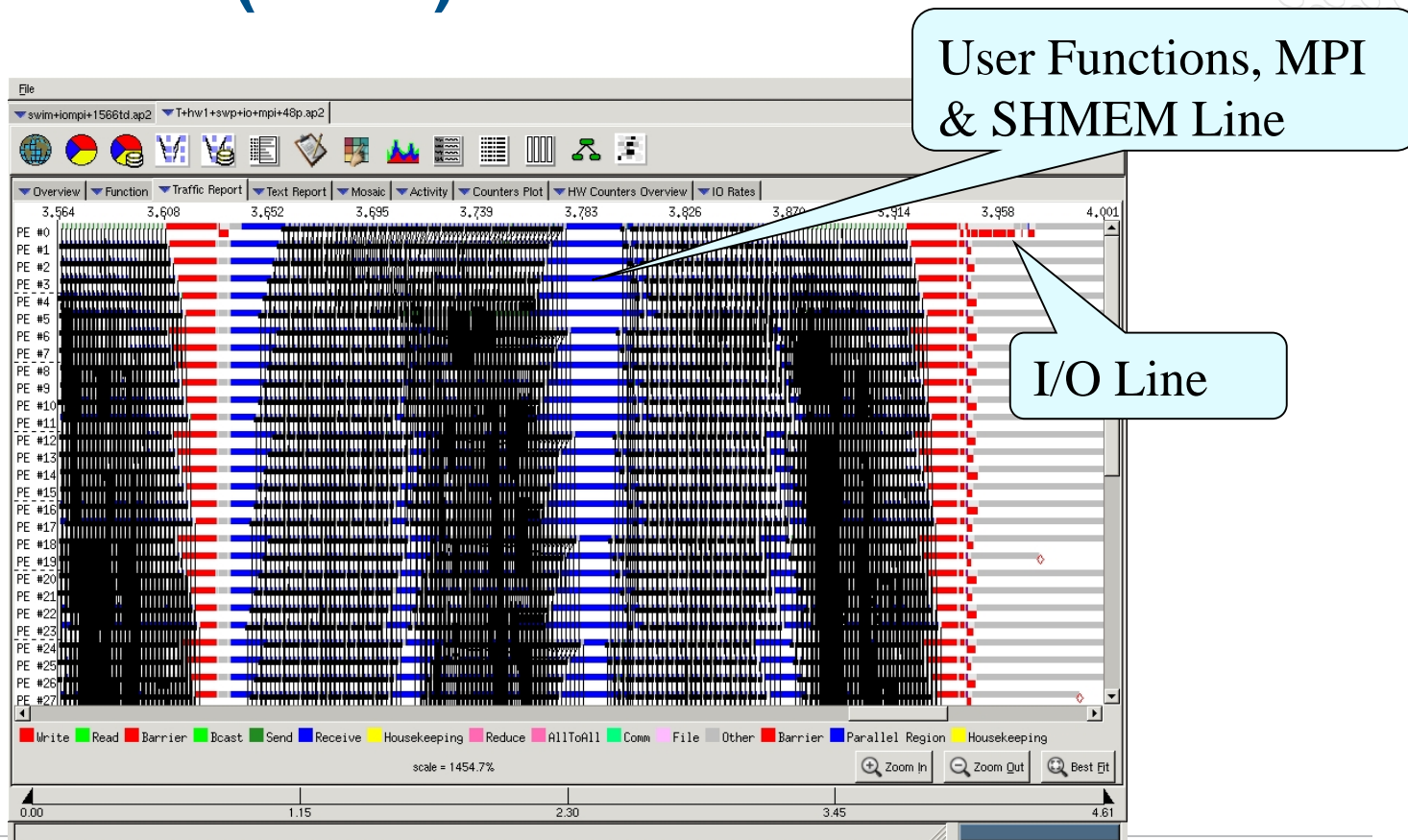
# Time Line View



- Full trace (sequence of events) enabled by setting **PAT\_RT\_SUMMARY=0**
- Helpful to see communication bottlenecks.
- Use it only for small experiments !



# Time Line View (Zoom)



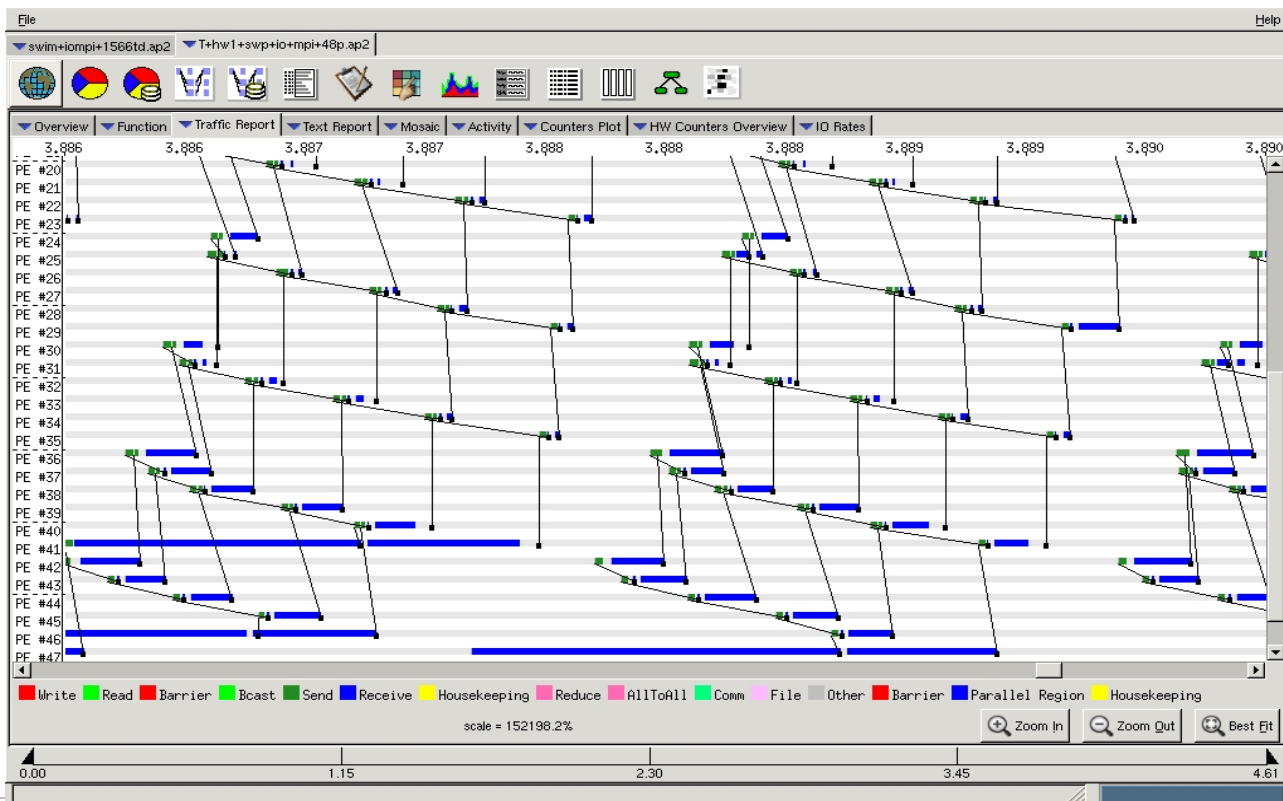
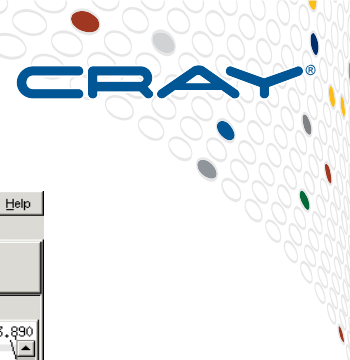
COMPUTE

STORE

ANALYZE



# Time Line View (Fine Grain Zoom)



COMPUTE

STORE

ANALYZE



# Reveal

- **Variable scoping and loopmark listing**

# Overview



- As with OpenMP , in order to use the OpenACC directives one has to understand the scoping of the variables, i.e. whether variables are shared or private.
- **Reveal is Cray's next-generation integrated performance analysis and code optimization tool.**
  - Source code navigation using whole program analysis (data provided by the Cray compilation environment.)
  - Coupling with performance data collected during execution by CrayPAT. Understand which high level serial loops could benefit from parallelism.
  - Enhanced loop mark listing functionality.
  - Dependency information for targeted loops
  - Assist users optimize code by providing variable scoping feedback and suggested compile directives.

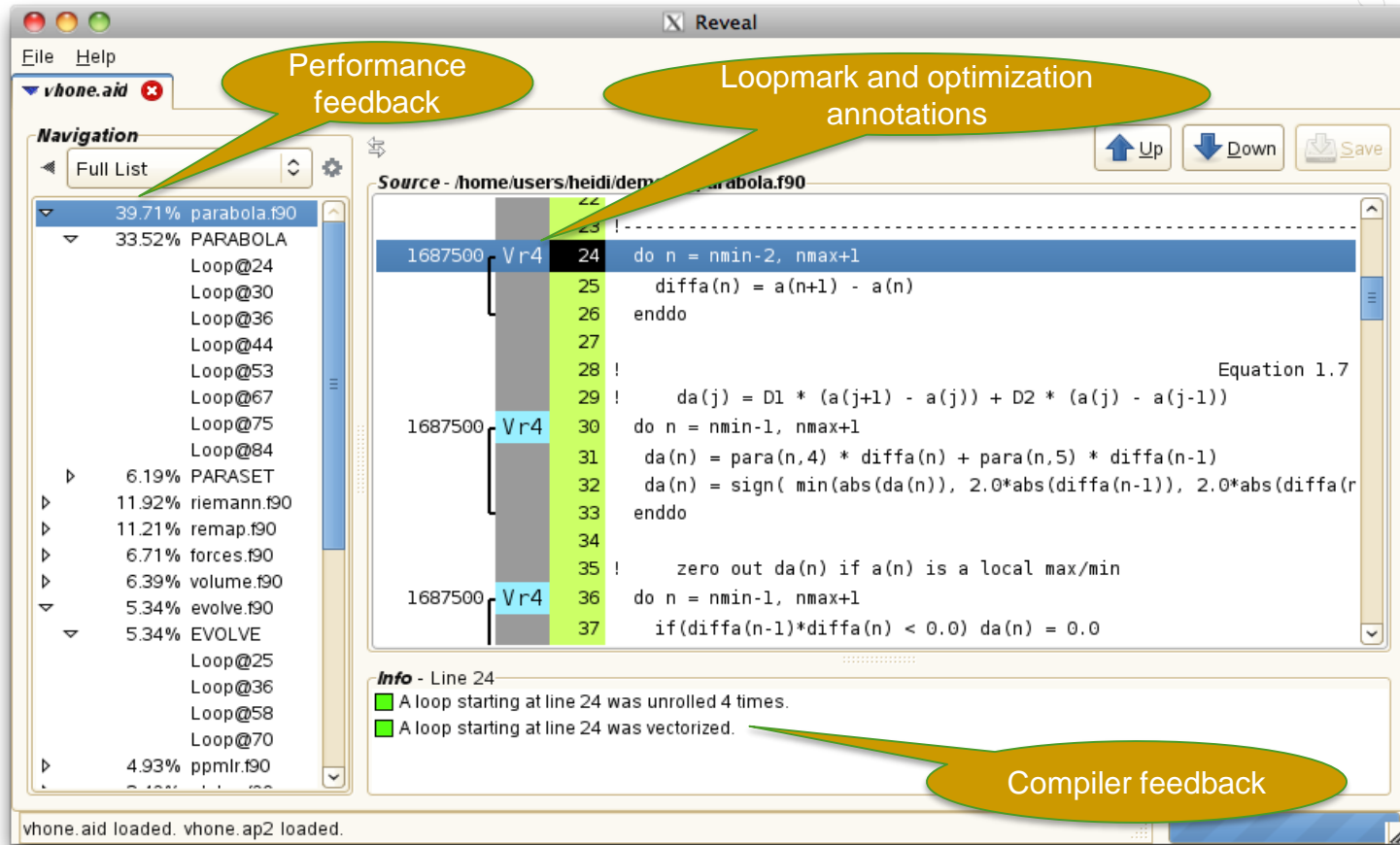


# How to use Reveal (generate program library)



- **Load the performance tools**
  - > `module load perftools-base`
- **Recompile only sources to generate program library**
  - > `ftn -O3 -hpl=my_program.pl -c my_program_file1.f90`
  - > `ftn -O3 -hpl=my_program.pl -c my_program_file2.f90`
- **The `program_library` is most useful when generated from fully optimized code. Instrument the program for tracing.**
- **After the collection of performance data and generation of a program library you can launch Reveal**
  - > `reveal my_program.pl my_program.ap2`
- **The `*.ap2` is from a loop work estimate of `my_program`**
  - You can omit the `*.ap2` and inspect only code listings.

# Visualize CCE's Loopmark with Performance Profile



COMPUTE

STORE

ANALYZE

# View Pseudo Code for Inlined Functions

The screenshot shows the Reveal 0.1 IDE interface. On the left, a file explorer lists various files, with 'init.f90' selected. A yellow callout bubble points to the 'init.f90' file in the list, stating 'Inlined call sites marked'. The main editor window displays the source code of 'init.f90'. Line 88, 'call grid(imax,xmin,xmax,zxa,zxc,zdx)', is highlighted in blue. A yellow callout bubble points to this line, stating 'Expand to see pseudo code'. Below the code, an 'Info' panel provides details about the inlining process:

- Info - Line 88
- A divide was turned into a multiply by a reciprocal.
- A loop starting at line 88 was unrolled 4 times.
- A loop starting at line 88 was vectorized.
- The call to grid was textually inlined.

```
80 ncycle = 0
81 ncycp = 0
82 ncycd = 0
83 ncycm = 0
84 nfile = 1000
85
86 ! Set up grid coordinates
87
88 call grid(imax,xmin,xmax,zxa,zxc,zdx)
88     t$26 = 100
88     t$27 = 100
88     $I_L88_100 = 0
88     !dir$ ivdep
88     do
88         zxa(1 + $I_L88_100) = 9.9999998e-3 * $I_L88_100
88         zdx(1 + $I_L88_100) = 9.9999998e-3
88         zxc(1 + $I_L88_100) = 4.9999999e-3 + ( 9.9999999e-3 * $I_L88_100 )
88         $I_L88_100 = 1 + $I_L88_100
88         if ( $I_L88_100 >= 100 ) exit
88     enddo
89 call grid(jmax,ymin,ymax,zya,zyc,zdy)
90 call grid(kmax,zmin,zmax,zza,zzc,zdz)
```

# Scoping Assistance – Review Scoping Results



Parallelization inhibitor messages are provided to assist user with analysis

Loops with scoping information are highlighted – red needs user assistance

User addresses parallelization issues for unresolved variables

Click on variable to view all occurrences in loop

Navigation

Full List

Source - /home/users/heidi/demoLM/sweepy.f90

```
1 subroutine sweepy
2
3 ! This subroutine performs sweeps in
4 ! After call to MPI_ALLTOALL, data
5 ! only two dimensions, perform
6 ! r hydro update, data is put
```

Info

loading /home/users/heidi/demoLM/vhone.aid/vhone\_22.T...

MP Scope Selector

2.f90: lines 28 -> 69

Name	Type	Scope	Info
f	Array	Unresolved	FAIL: Last defining iteration not known for variable that is live on exit...
flat	Array	Unresolved	FAIL: Last defining iteration not known for variable that is live on exit...
q	Array	Unresolved	FAIL: Last defining iteration not known for variable that is live on exit...
isy	Scalar	Shared	
js	Scalar	Shared	
ks	Scalar	Shared	
ngeomx	Scalar	Shared	
nleftx	Scalar	Shared	
npey	Scalar	Shared	
nrightx	Scalar	Shared	
recv2	Array	Shared	
zdx	Array	Shared	
zfi	Array	Shared	
zpr	Array	Shared	
zro	Array	Shared	
zux	Array	Shared	
zuy	Array	Shared	
zuz	Array	Shared	

First/Last Private

☐ Enable First Private

☐ Enable Last Private

Search:

Insert Directive Show Directive Close

# Scoping Assistance – Generate Directive



Reveal generates  
example OpenMP  
directive

The screenshot shows the Reveal IDE interface. The main window displays source code for a file named `vhone.f90`. The code includes a loop starting at line 29, which is highlighted in green. A yellow callout bubble points to this loop, stating "Reveal generates example OpenMP directive".

On the left, there is a "Navigation" pane showing a "Full List" of code elements. Below it, an "OpenMP Directive" window is open, displaying the following code:

```
$OMP parallel do default(none) &  
$OMP shared (gamm,send1,zdx,zf,zpr,zro,zux,zuy,zuz,zxa) &  
$OMP lastprivate (dx,dx0,e,f,p,r,u,v,w,x,a,xao)
```

Below the directive window, a list of code elements is shown, including "Loop@28", "Loop@29", "Loop@32", "Loop@33", "Loop@44", "Loop@58", "sweepx1.f90", "SWEEPX1", "Loop@28", "Loop@29", and "Loop@32".

On the right, an "OpenMP Scope Selector" window is open, showing a table of variables and their types. The table has columns for "Name", "Type", and "Info". The "Info" column contains warnings such as "WARN: LastPrivate of array may be very expensive." for several variables.

At the bottom, an "Info" window is open, displaying the following message:

```
Info - Line 29  
A loop starting at line 29 was not vectorized because  
Loop has been flattened.  
Loop has been flattened.
```

COMPUTE

STORE

ANALYZE





# Allinea DDT / Forge

## Visual Debugging

---

COMPUTE

| STORE |

ANALYZE

# Overview



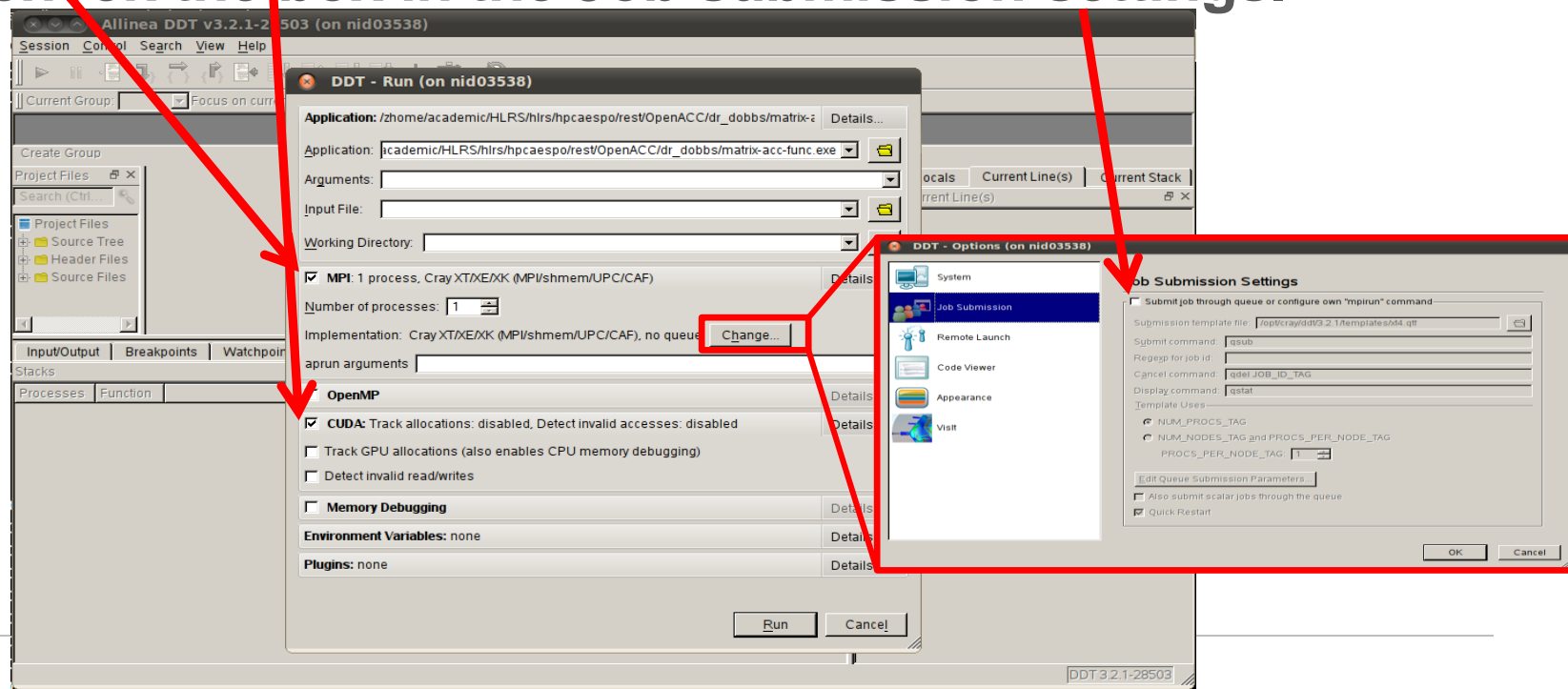
- **Graphical debugger designed for**
  - Distributed or shared memory parallelism or a combination.
  - Memory debugging, data analysis
  - C, C++, Fortran, UPC, CUDA, OpenACC
- **Recompile your application for DWARF support**
  - `module load craype-accel-nvidia60`
  - `{CC,cc,ftn} -g <your_application>`
- **Get an interactive session and load the debugger**
  - `module load craype-accel-nvidia60`
  - `ddt [--connect] srun -C gpu -n1 -t15 ./exe`
- **For sequential codes (no MPI) you need to add a dummy MPI\_Init and MPI\_Finalize.**



# Using DDT for debugging OpenACC codes



- MPI and CUDA boxes should be already checked.
- Uncheck the box in the Job submission settings.



# Summary and general remarks



- **Use CrayPAT to understand where your application is spending time.**
  - Automatic performance analysis based on tracing and sampling for large applications. Only tracing more efficient for smaller programs.
  - Loop work estimate to identify interesting loops to port to the GPU. Can also be done in the framework of the APA.
- **Use Reveal to better understand loop mark listings and do variable scoping for the interesting loops. Use the loop work estimates from the CrayPAT runs.**
- **Comparative debugging, e.g. comparing messages from different compiler (Cray, PGI, Nvidia, ...) can be very helpful.**