

Performance Analysis with CrayPat

Part 2

Outline

- **Apprentice2**
 - Tool used to visualize performance data instrumented with the CrayPat tool. There are many options for viewing results
- **Loop work estimates with CrayPat**
 - How to prepare the code for collection of loop statistics.
- **Reveal**
 - Generate the program library.
 - Use the GUI.
- **Profiling OpenMP**
- **CrayPAT API**

Cray Apprentice2





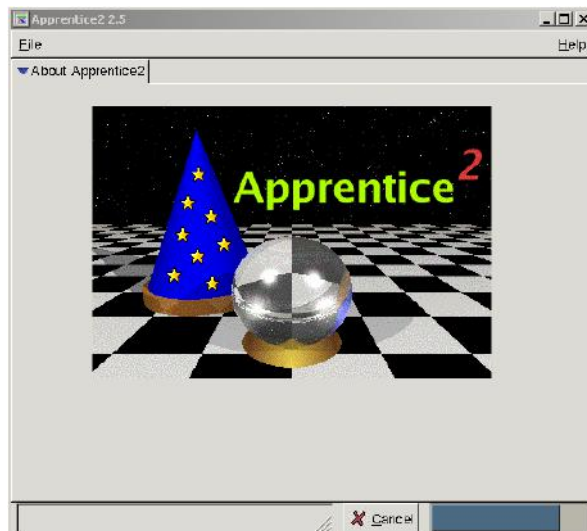
Installing Apprentice2 on Laptop

From a Cray login node

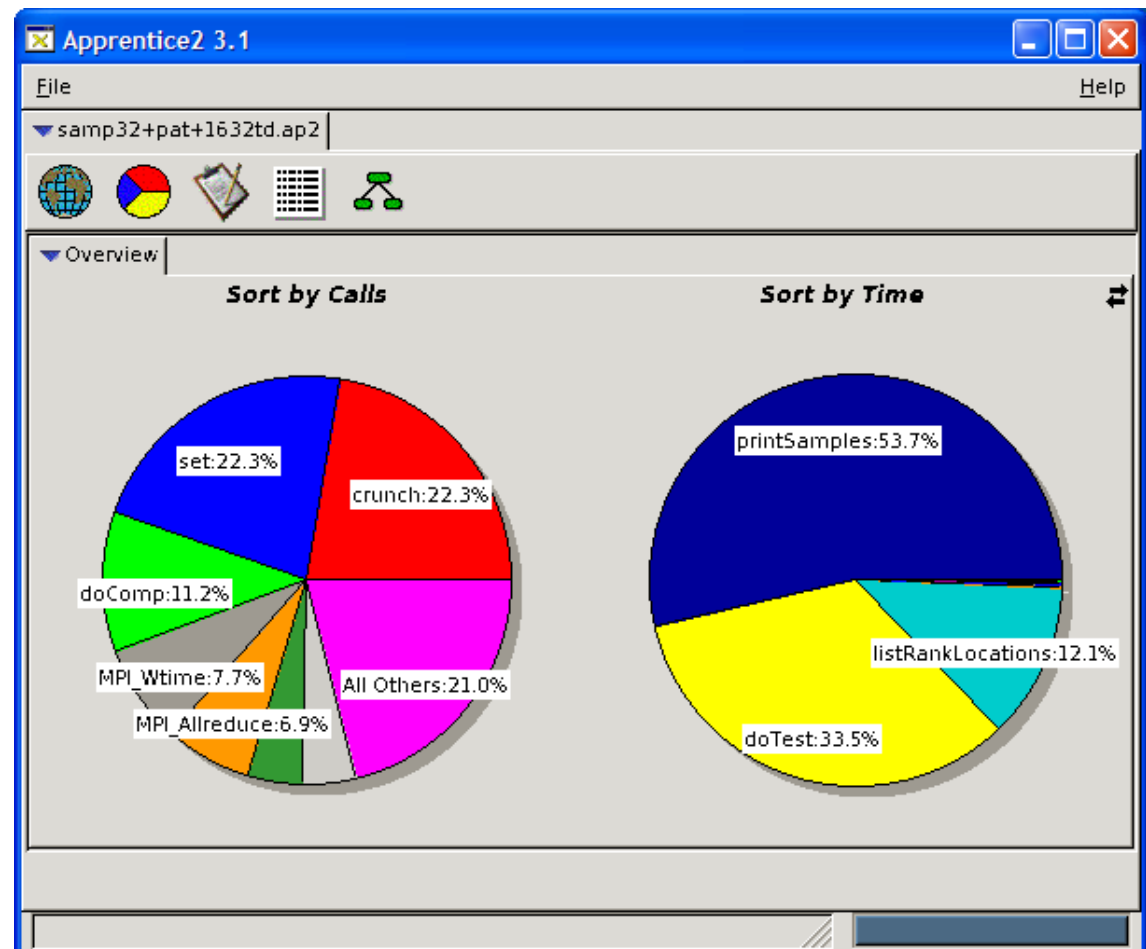
- `> module load perftools`
- **Go to:**
 - `$CRAYPAT_ROOT/share/desktop_installers/`
- **Download .dmg or .exe installer to laptop**
- **Double click on installer and follow directions to install**
- **Of course, can just run app2 from the login prompt instead**

Cray Apprentice2

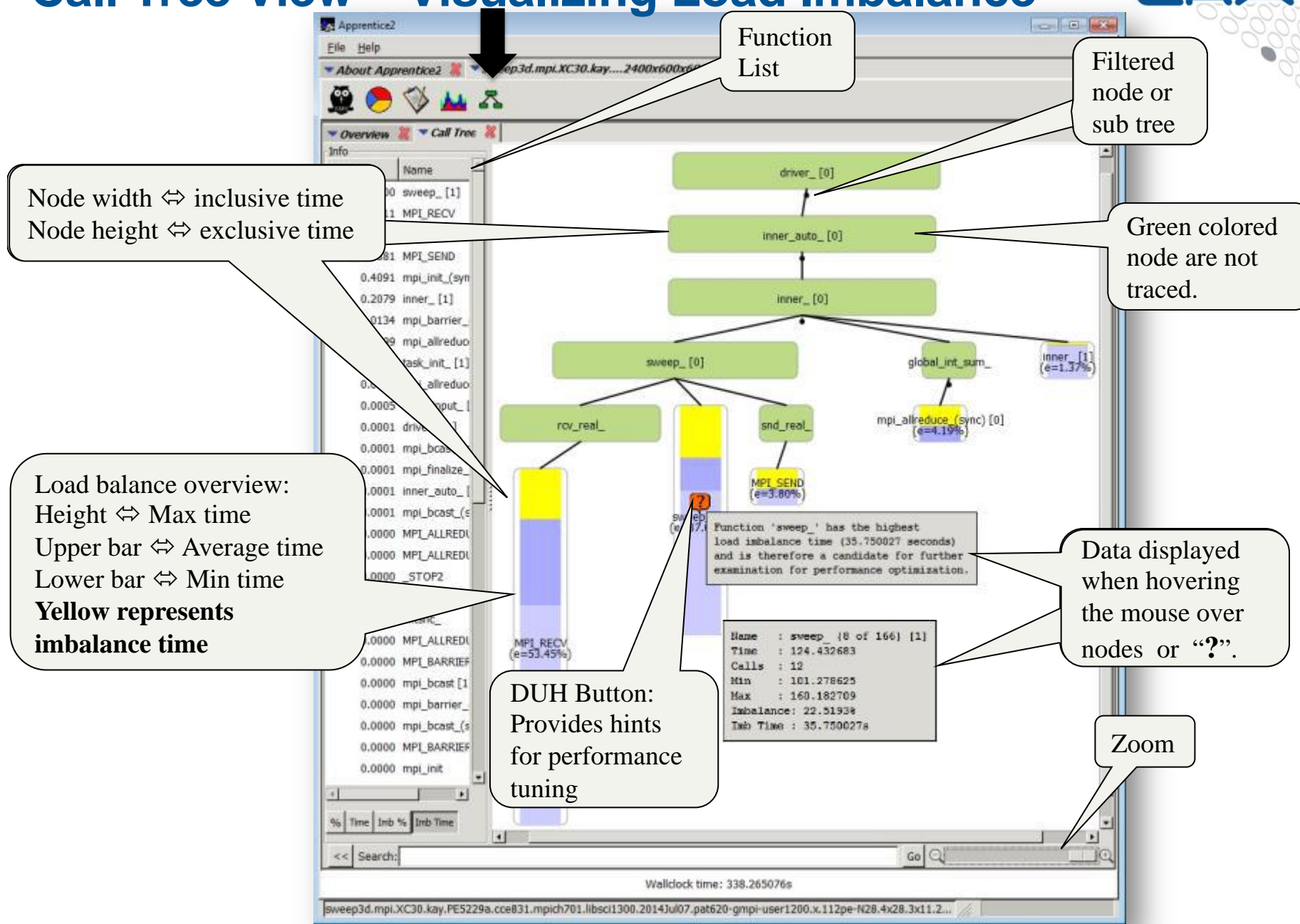
```
% module load perftools
% app2 program1+pat+180tdo-0000.ap2
```



Many options for viewing Results. See “man app2” or Cray documentation



Call Tree View – Visualizing Load Imbalance



Loop Work Estimates



Loop Work Estimates

- **Assess suitability of loop nests for optimization**

- Gives information on inclusive time spent in the loop nests and typical trip count of the loops.
- Only available with CCE. CrayPAT can generate this information via a special kind of tracing experiment. Just like adding automatic tracing at the function level, we can add tracing to individual loops

```
> module load perftools
```

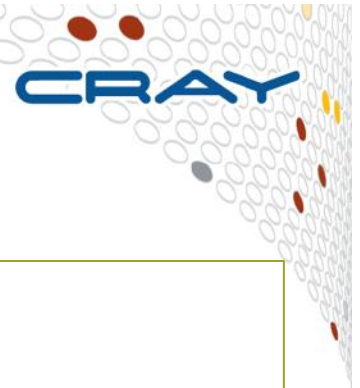
- Makes the default version of CrayPAT available

```
> ftn -c -h profile_generate himeno.f90
```

```
> ftn -o himeno.exe himeno.o
```

```
> pat_build -w[-u] himeno.exe
```

- Recompile your program for gathering loop statistics.
- It is recommended to turn off OpenMP and OpenACC for the loop work estimates via `-h noomp -h noacc`
- Instrument the application for tracing (APA also possible)



Loop Work Estimates

```
aprun -n 24 ./himeno.exe+pat
```

- Execute the instrumented program.
- This generates one or more raw data files(s) in .xf format.

```
> pat_report -o report.txt himeno.exe+pat*.xf
```

- Process the raw data files(s) for use with Reveal.
- This generates a performance data file *.ap2 and text report report.txt.
- Even without the `-u` option to `pat_build` in the previous step you will see user functions listed in the first table. These are routines containing loops.
- Consider the `-O profile_loops` option to `pat_report` to show the time spent in loops compared to other routines.
- Reveal can use the *.ap2 to visualize time expensive loops.



Table 2: Loop Stats by function

Subroutine			Line number				
Loop Incl	Loop Incl Time	Time (Loop Adj.)	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE-HIDE
93.0%	19.232051	0.000849	2	26.5	3	50	jacobi.LOOP.1.li.236
77.8%	16.092021	0.001350	53	255.0	255	255	jacobi.LOOP.2.li.240
77.8%	16.090671	0.110827	13515	255.0	255	255	jacobi.LOOP.3.li.241
77.3%	15.979844	15.979844	3446325	511.0	511	511	jacobi.LOOP.4.li.242
14.1%	2.906115	0.001238	53	255.0	255	255	jacobi.LOOP.5.li.263
14.0%	2.904878	0.688611	13515	255.0	255	255	jacobi.LOOP.6.li.264
10.7%	2.216267	2.216267	3446325	511.0	511	511	jacobi.LOOP.7.li.265
4.3%	0.881573	0.000010	1	259.0	259	259	initmt.LOOP.1.li.191
4.3%	0.881563	0.000645	259	259.0	259	259	initmt.LOOP.2.li.192
4.3%	0.880918	0.880918	67081	515.0	515	515	initmt.LOOP.3.li.193
2.7%	0.560499	0.000055	1	257.0	257	257	initmt.LOOP.4.li.210
2.7%	0.560444	0.006603	257	257.0	257	257	initmt.LOOP.5.li.211
2.7%	0.553842	0.553842	66049	513.0	513	513	initmt.LOOP.6.li.212

Nested Loops

Reveal

Compiler Feedback and Variable Scoping



Reveal

- For an OpenMP port a developer 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.





Input to Reveal

```
> module load perftools
```

- Makes the default version of CrayPAT available

```
> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90
```

```
> ftn -O3 -hpl=my_program.pl -c my_program_file2.f90
```

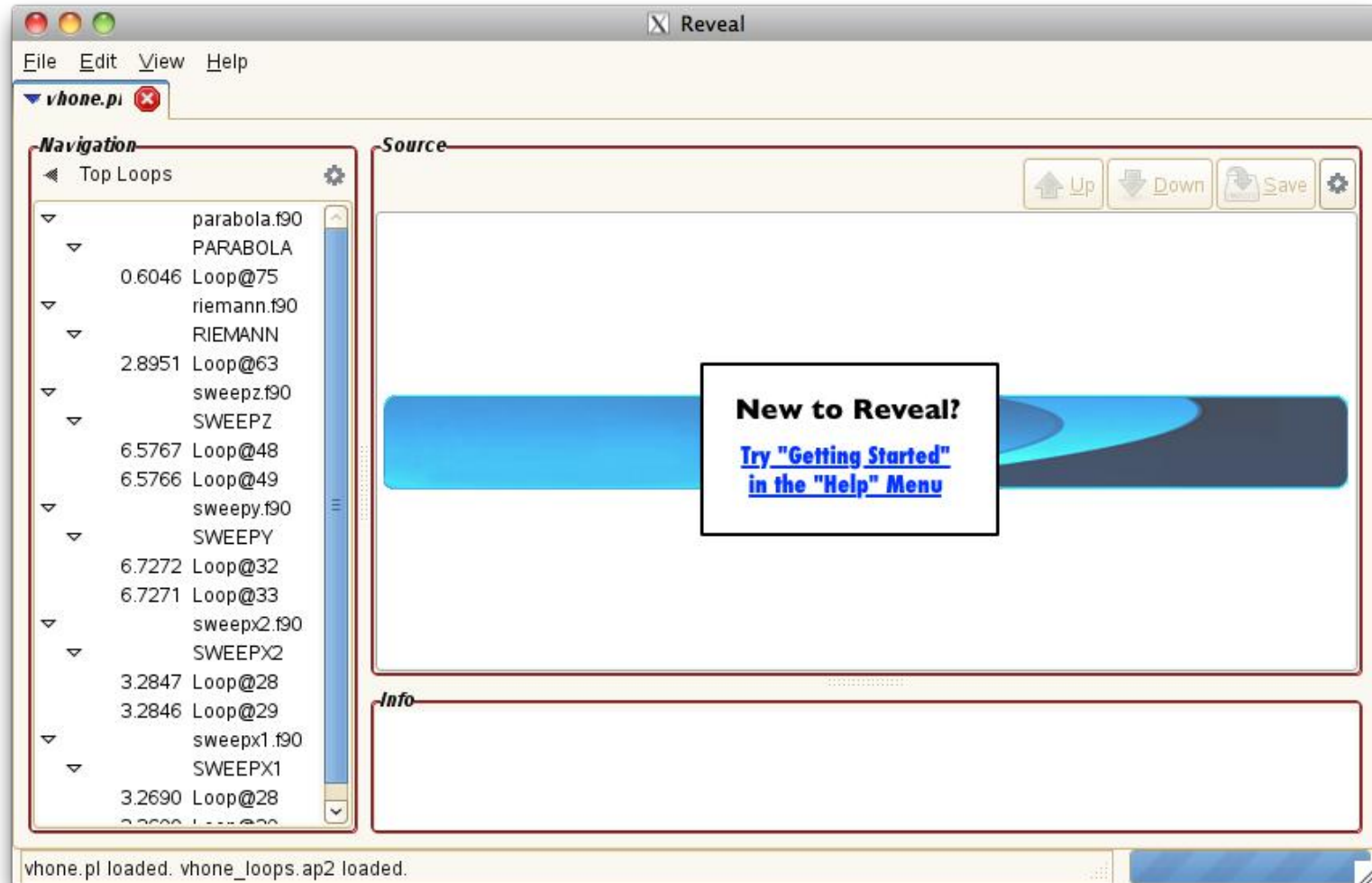
- Recompile only sources to generate program library `my_program.pl`
- The program library is most useful when generated from fully optimized code.
- Use absolute paths to specify the program library if necessary.

```
> reveal my_program.pl my_program.ap2 &
```

- After the collection of performance data in a separate experiment and generation of a program library you can launch Reveal.

- **The *.ap2 is from a loop work estimate of my_program**
 - You can omit the *.ap2 and inspect only compiler feedback.
 - Note that the `profile_generate` option disables most automatic compiler optimizations, which is why Cray recommends generating this data separately from generating the `program_library` file.

Reveal with Loop Work Estimates



Visualize CCE's Loopmark with Performance Profile



The screenshot displays the CCE Loopmark visualization tool interface. On the left, a 'Navigation' pane shows a 'Full List' of code elements with their performance percentages. The main area shows the source code for 'parabola.f90' with a vertical loopmark visualization on the left side of the code lines. A status bar at the bottom provides compiler feedback.

Performance feedback (points to the Navigation pane):

- 39.71% parabola.f90
- 33.52% PARABOLA
 - Loop@24
 - Loop@30
 - Loop@36
 - Loop@44
 - Loop@53
 - Loop@67
 - Loop@75
 - Loop@84
- 6.19% PARASET
- 11.92% riemann.f90
- 11.21% remap.f90
- 6.71% forces.f90
- 6.39% volume.f90
- 5.34% evolve.f90
- 5.34% EVOLVE
 - Loop@25
 - Loop@36
 - Loop@58
 - Loop@70
- 4.93% ppmlr.f90

Loopmark and optimization annotations (points to the source code):

```
Source - /home/users/heide.../noLM/parabola.f90
23 !-----
1687500 Vr4 24 do n = nmin-2, nmax+1
25     diffa(n) = a(n+1) - a(n)
26 enddo
27
28 !
29 !      da(j) = D1 * (a(j+1) - a(j)) + D2 * (a(j) - a(j-1))      Equation 1.7
1687500 Vr4 30 do n = nmin-1, nmax+1
31     da(n) = para(n,4) * diffa(n) + para(n,5) * diffa(n-1)
32     da(n) = sign( min(abs(da(n)), 2.0*abs(diffa(n-1))), 2.0*abs(diffa(n-1)))
33 enddo
34
35 !      zero out da(n) if a(n) is a local max/min
1687500 Vr4 36 do n = nmin-1, nmax+1
37     if(diffa(n-1)*diffa(n) < 0.0) da(n) = 0.0
```

Compiler feedback (points to the Info pane):

Info - Line 24

- A loop starting at line 24 was unrolled 4 times.
- A loop starting at line 24 was vectorized.

Visualize CCE's Loopmark with Performance Profile (2)



The screenshot shows the Reveal 0.1 application interface. On the left, a file tree lists various source files, with 'sweepx1.f90' selected. Below the tree, an 'Info - Line 32' panel displays two green messages: 'A loop starting at line 32 w' and 'A loop starting at line 32 w'. The main window displays a code profile for 'sweepx1.f90', with line 32 highlighted in blue and marked with a 'U' icon. A yellow callout bubble points to this icon with the text 'Integrated message 'explain support''. On the right, an 'Explain' dialog box is open, titled 'OPT_INFO: A loop starting at line %s was unrolled.' It contains text explaining loop unrolling and provides two code snippets for comparison. The first snippet shows a nested loop with 'unroll-and-jam' transformation. The second snippet shows the same loop with 'literal outer unroll' transformation. The dialog box has 'Explain other message...' and 'Close' buttons.

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.

```
# 434 "/ptmp/pdgcgs/pdgcgs.tbs.81/bld.dir/build.64.ndb/pdgcgs/pdgcgs_ftn.msg.c"
DO J = 1,10
DO I = 1,100
  A(I,J) = B(I,J) + 42.0
ENDDO
ENDDO

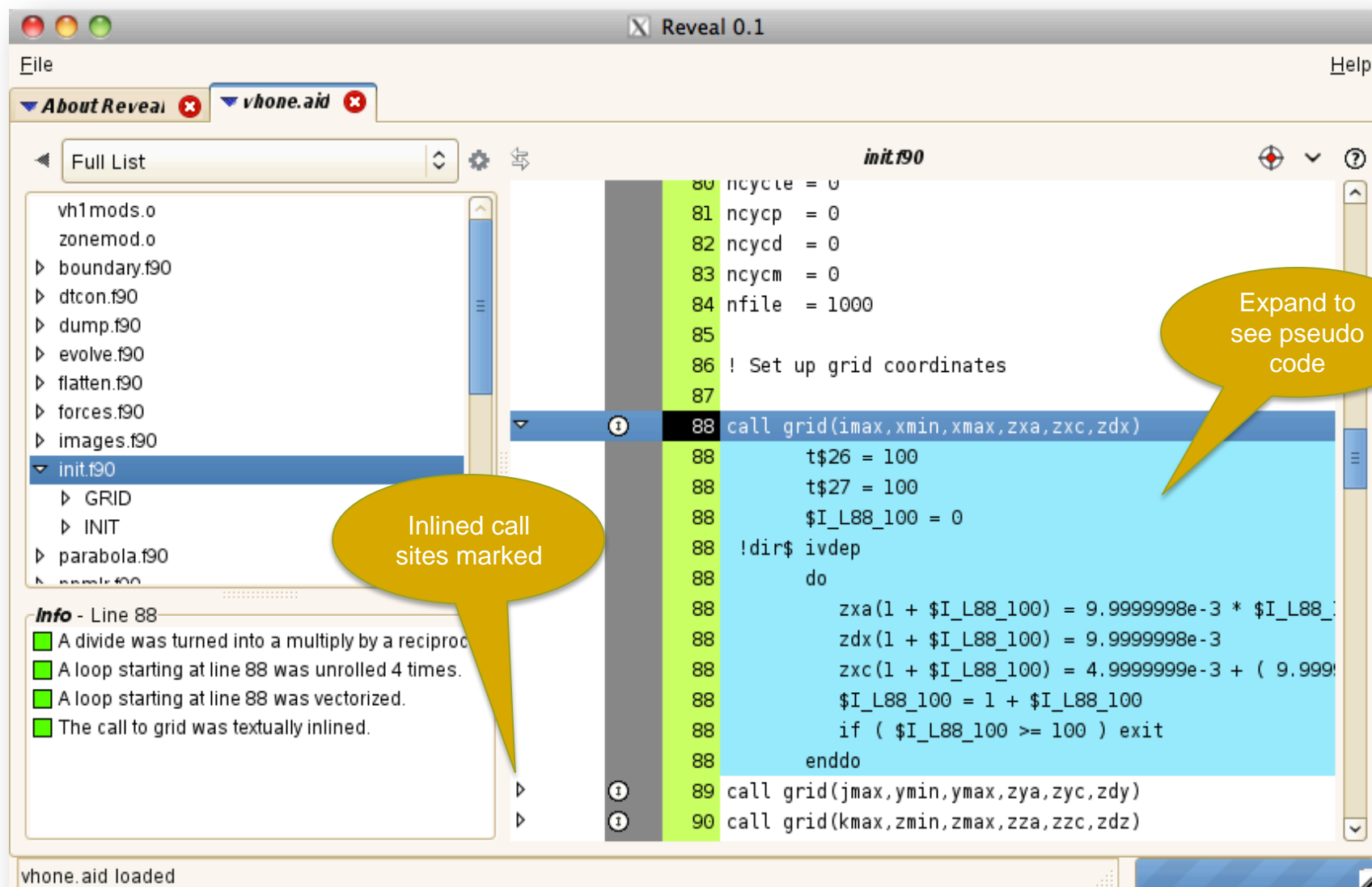
DO J = 1,10,2
DO I = 1,100
  A(I,J) = B(I,J) + 42.0 ! unroll-and-jam
  A(I,J+1) = B(I,J+1) + 42.0
ENDDO
ENDDO

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

The literal outer unroll code performs the same sequence of memory operations as the original nest, while the unroll-and-jam transformation interleaves operations from outer loop iterations. The compiler employs literal outerloop unrolling only when the data dependencies in the loop, or a control flow impediment, prevent fusion of the replicated inner loops. Literal outer loop unrolling is generally not desirable. It is provided to ensure expected behavior and for those rare instances where the user has determined that it is beneficial.

Explain other message... Close

View Pseudo Code for Inlined Functions



The screenshot shows the Reveal 0.1 application window. The left sidebar displays a file tree with the following items:

- vh1mods.o
- zonemod.o
- ▶ boundary.f90
- ▶ dtcon.f90
- ▶ dump.f90
- ▶ evolve.f90
- ▶ flatten.f90
- ▶ forces.f90
- ▶ images.f90
- ▼ **init.f90**
 - ▶ GRID
 - ▶ INIT
 - ▶ parabola.f90

The main window displays the pseudo code for the **init.f90** file. The code is as follows:

```

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_
88         zdx(1 + $I_L88_100) = 9.9999998e-3
88         zxc(1 + $I_L88_100) = 4.9999999e-3 + ( 9.999
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)

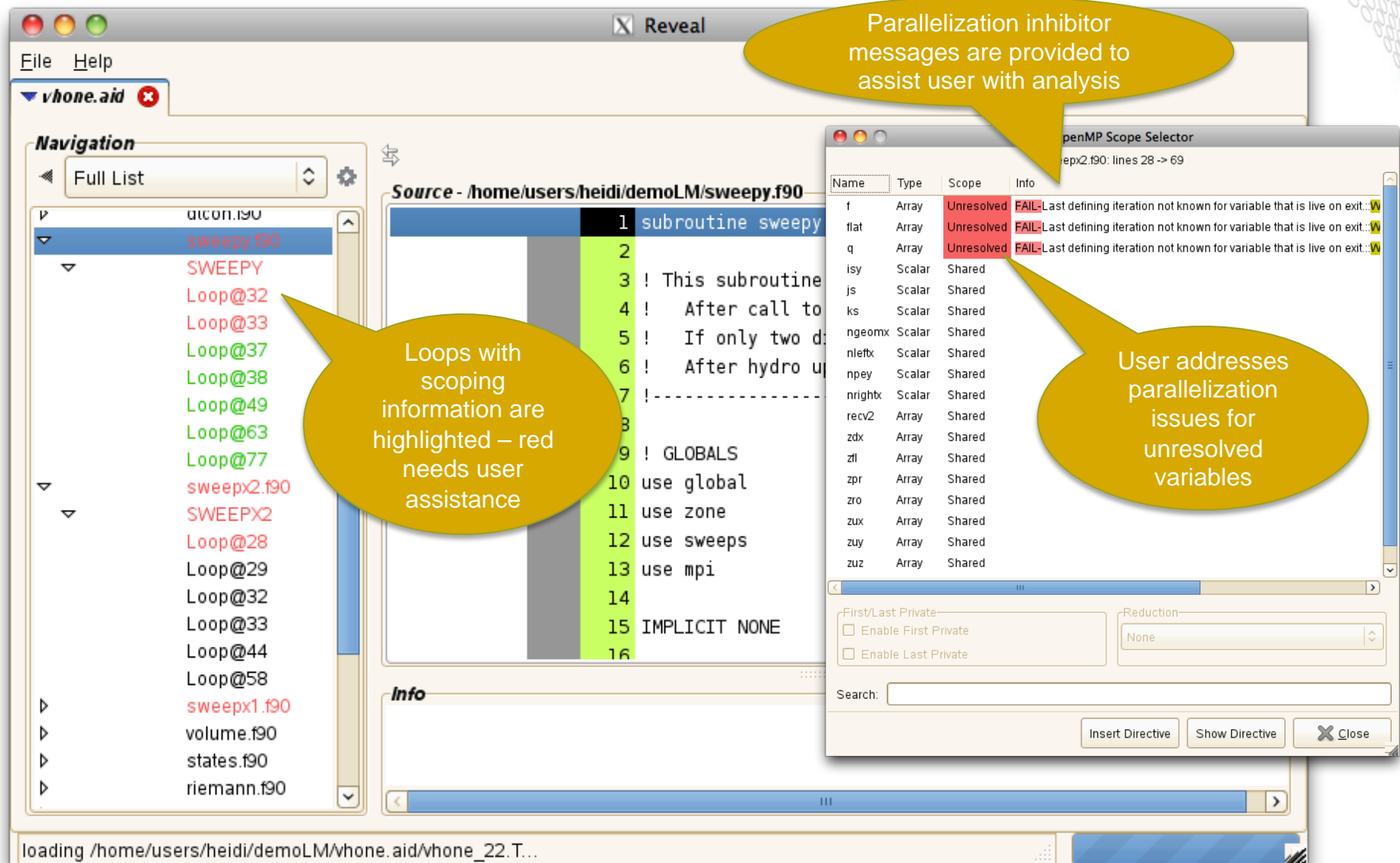
```

Two callout boxes provide additional information:

- Inlined call sites marked:** Points to the line number 88 in the left margin, indicating that the call to the `grid` function is inlined.
- Expand to see pseudo code:** Points to the expanded pseudo code block for the `call grid` function, showing the internal operations.

The bottom status bar indicates "vhone.aid loaded".

Scoping Assistance – Review Scoping Results



Navigation

- Full List
- utcon.f90
- sweep.f90**
 - SWEEPY
 - Loop@32
 - Loop@33
 - Loop@37
 - Loop@38
 - Loop@49
 - Loop@63
 - Loop@77
 - sweepx2.f90
 - SWEEPX2
 - Loop@28
 - Loop@29
 - Loop@32
 - Loop@33
 - Loop@44
 - Loop@58
 - sweepx1.f90
 - volume.f90
 - states.f90
 - riemann.f90

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

```

1 subroutine sweepy
2
3 ! This subroutine
4 ! After call to
5 ! If only two d
6 ! After hydro u
7 ! -----
8
9 ! GLOBALS
10 use global
11 use zone
12 use sweeps
13 use mpi
14
15 IMPLICIT NONE
16

```

OpenMP Scope Selector

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	
zfl	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

Reduction

None

Search:

Scoping Assistance – User Resolves Issues



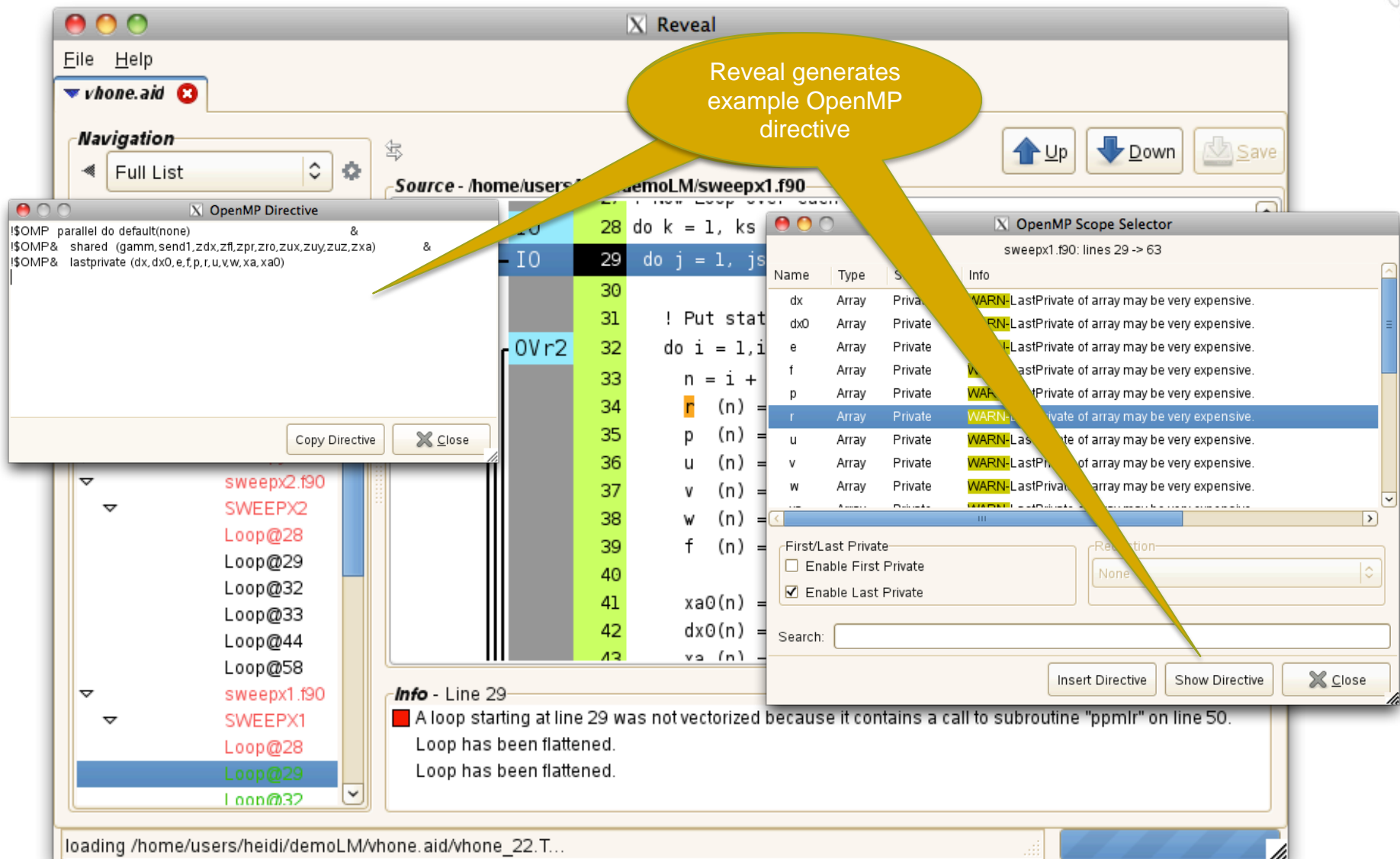
The screenshot displays the Reveal IDE interface. On the left, a sidebar titled "OpenMP Tips" lists several categories: "Reduction in an inlined function", "Scoping conflict with inlined variable", and "Scoping conflict with locally visible array". The third category is expanded, showing a tip: "An array requires conflicting scopes at different locations. It may be possible to declare and use a different array for the private array uses." A yellow callout bubble points to this tip with the text: "Use Reveal's OpenMP parallelization tips".

The main editor shows a Fortran code snippet with a loop over each row. The code includes a loop over `m` from 1 to `npey`, and an inner loop over `i` from 1 to `isy`. The code uses `recv2` subroutines. A yellow callout bubble points to the `recv2` calls with the text: "Click on variable to view all occurrences in loop".

On the right, the "OpenMP Scope Selector" window is open, showing a table of variables and their scopes. The table has columns: Name, Type, Scope, and Info. The variables listed are `f`, `flat`, `q`, `isy`, `js`, `ks`, `ngeomx`, `nletx`, `npey`, `nrightx`, `recv2`, `zdx`, `zfi`, `zpr`, `zro`, `zux`, `zuy`, and `zuz`. The `recv2` variable is highlighted in orange. The "Info" column for `recv2` contains the text: "Unresolved FAIL: Last defining iteration not known for variable that is live on exit...".

At the bottom, an "Info" window shows a message: "Info - Line 28: A loop starting at line 28 was not vectorized because it contains a call to subroutine 'ppmlr' on line 55. Loop has been flattened. Loop has been flattened." The status bar at the bottom indicates the file path: "loading /home/users/heidi/demoLM/vhone.aid/vhone_22.T..."

Scoping Assistance – Generate Directive



Reveal generates example OpenMP directive

OpenMP Directive

```

$OMP parallel do default(none) &
$OMP shared (gamm, send1, zdx, zfl, zpr, zro, zux, zuy, zuz, zxa) &
$OMP lastprivate (dx, dx0, e, f, p, r, u, v, w, xa, xa0)

```

OpenMP Scope Selector

sweepx1.f90: lines 29 -> 63

Name	Type	Scope	Info
dx	Array	Private	WARN: LastPrivate of array may be very expensive.
dx0	Array	Private	WARN: LastPrivate of array may be very expensive.
e	Array	Private	WARN: LastPrivate of array may be very expensive.
f	Array	Private	WARN: LastPrivate of array may be very expensive.
p	Array	Private	WARN: LastPrivate of array may be very expensive.
r	Array	Private	WARN: LastPrivate of array may be very expensive.
u	Array	Private	WARN: LastPrivate of array may be very expensive.
v	Array	Private	WARN: LastPrivate of array may be very expensive.
w	Array	Private	WARN: LastPrivate of array may be very expensive.

First/Last Private

☐ Enable First Private

☒ Enable Last Private

Resolution: None

Search:

Insert Directive Show Directive Close

Info - Line 29

A loop starting at line 29 was not vectorized because it contains a call to subroutine "ppmlr" on line 50. Loop has been flattened. Loop has been flattened.



Reveal Use Cases (NERSC)

VH1 Astrophysics Code

- Written with high level loops and complex decision processes
- Ported to hybrid MPI + OpenMP using Reveal
- Reveal was able to identify
 - storage conflicts
 - private variables in modules
 - reductions down the call chain that require critical regions
- Scoping was performed in seconds where it would have taken weeks to get correct without Reveal

S3D - Structured Cartesian Mesh Flow Solver

- Pure MPI program, converted to a hybrid multi- core application suited for a multi-core node with or without an accelerator.
- When the work was started, Reveal did not exist.
- Once Reveal was available, it was instrumental in identifying bugs in the scoping of extremely large loops (3000 lines of Fortran).

Reveal Summary

- **Reveal can be used to simplify the task of adding OpenMP to MPI programs**
- **Can be used as a stepping stone for codes targeted for nodes with higher core counts (including Phi) and as the first step in adding OpenACC to applications to for execution on GPUs**
- **Requires the full CCE compiler to be available to function, so no standalone local version currently available**

CrayPat and OpenMP



OpenMP data collection and reporting

- **For programs that use OpenMP**
 - CrayPat can measure the overhead incurred by entering and leaving parallel regions and work-sharing constructs within parallel regions
 - Show per-thread timings and 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` (default view)
 - Imbalance based on the set of all threads in the program
 - `profile_pe.th`
 - 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 enabled by default with CCE
- 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



CrayPat API

API for adding User Instrumentation

- The CrayPat API calls enable you to insert functions into your source code that write special tracing records into the experiment data file at runtime
 - API calls are supported in both Fortran and C. After the perftools module is loaded, the include files that define the CrayPat API can be found in the `$CRAYPAT_ROOT/include` directory and consist of the C header file, `pat_api.h`, and the Fortran and Fortran 77 header files, `pat_apif.h` and `pat_apif77.h`, respectively.
 - `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 equivalents, like MPI, are subroutines with extra final integer argument for return value
- More information is given in the `pat_build` man page. For further examples of using CrayPat API calls in source code, see the topic "API" in the `pat_help` system.

PAT Regions example

```
include "pat_apif.h"
...
call PAT_region_begin( 1, "step 1", istat )
! the execution of this code segment will appear in
! CrayPAT output as "step 1"
...
call PAT_region_end( 1, istat )
...
call PAT_region_begin( 2, "step 2", istat )
! the execution of this code segment will appear in
! CrayPAT output as "step 2"
...
call PAT_region_end( 2, istat )
...
```

The `-DCRAYPAT` is defined by CCE compilers when `perftools` is loaded.



PAT_region example

100.0%	58225.2	--	--	Total
91.2%	53072.9	--	--	USER
43.9%	25571.3	388.7	1.5%	calc_force_
29.7%	17292.9	289.1	1.6%	calc_p_
14.3%	8305.5	75.5	0.9%	pair_table_
1.4%	844.2	74.8	8.2%	predict_
7.5%	4363.8	--	--	MPI
3.8%	2229.9	905.1	28.9%	MPI_SENDRECV
2.1%	1208.5	1050.5	46.6%	MPI_BARRIER
1.4%	829.7	487.3	37.1%	
MPI_ALLREDUCE				
1.4%	788.1	--	--	ETC

- But calc_force is 494 lines and calc_p is 334 lines long!
- Introduce 4 PAT regions to the code, to focus on 2 sig regions in each
1.force_step1, 2.force_step2, 3.p_step1, 4.p_step2

PAT_region example

100.0%	58359.5	--	--	Total

90.9%	53023.8	--	--	USER

43.1%	25131.3	510.7	2.0%	#3.force_step1
28.9%	16879.8	345.2	2.0%	#1.p_step1
14.3%	8317.4	66.6	0.8%	pair_table_
1.4%	834.7	79.3	8.7%	predict_
=====				
7.8%	4551.9	--	--	MPI

3.9%	2249.3	941.7	29.6%	MPI_SENDRECV
2.3%	1330.7	1269.3	48.9%	MPI_BARRIER
1.5%	878.0	496.0	36.2%	MPI_ALLREDUCE
=====				
1.3%	783.5	--	--	ETC
=====				

Narrowed down to
some 1/2 of the lines
of code – could
refine further



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.**
- **A working OpenMP port of your application is always a good starting point for an OpenACC port (use Reveal).**
- **Comparative debugging, e.g., comparing messages from different compiler (Cray, PGI, Nvidia, ...) can be very helpful.**