Performance Measurement and Analysis Tools for Cray XE/XK Systems

Heidi Poxon
Cray Inc.

Topics

- CrayPat-lite
- Reveal

CrayPat-lite

CrayPat-lite Goals

- **Provide automatic application performance statistics at the end of a job**
  - Focus is to offer a simplified interface to basic application performance information for users not familiar with the Cray performance tools and perhaps new to application performance analysis
  - Gives sites the option to enable/disable application performance data collection for all users for a period of time

- **Keep traditional or "classic" perftools working the same as before**

- **Provide a simple way to transition from perftools-lite to perftools to encourage further tool use for performance analysis**

Steps to Using CrayPat "classic"

Access performance tools software

```
> module load perftools
```

Build program, retaining .o files

```
> make
```

→ a.out

Instrument binary

```
> pat_build –O apa a.out
```

→ a.out+pat

Modify batch script and run program

```
aprun a.out+pat
```

→ a.out+pat*.xf

Process raw performance data and create report

```
> pat_report a.out+pat*.xf
```

→ a.out+pat*.ap2
Text report to stdout
a.out+pat*.apa
MPICH_RANK_XXX

Steps to Using CrayPat-lite

Access light version of performance tools software

```
> module load perftools-lite
```

Build program

```
> make
```

→ a.out (instrumented program)

Run program (no modification to batch script)

```
aprun a.out
```

→ Condensed report to stdout
a.out*.rpt (same as stdout)
a.out*.ap2
MPICH_RANK_XXX files

Benefits of CrayPat-lite

- Program is automatically relinked to add instrumentation in a.out (pat_build step done for the user)
- .o files are automatically preserved
- No modifications are needed to a batch script to run instrumented binary, since original binary is replaced with instrumented version
- pat_report is automatically run before job exits
- Performance statistics are issued to stdout
- User can use “classic” CrayPat for more in-depth performance investigation

Performance Statistics Available

- **Set of predefined experiments, enabled with the CRAYPAT_LITE environment variable**
  - Sample_profile
  - Event_profile
  - GPU

- **Job information**
  - Number of MPI ranks, ranks per node, number of threads
  - Wallclock
  - High memory water mark
  - Aggregate MFLOPS (CPU only)

- **Profile of top time consuming routines with load balance**
- **Observations**
- **Instructions on how to get more information**

Sample Output – LAMMPS

```
#################################################################
#                                                               #
#            CrayPat-lite Performance Statistics                #
#                                                               #
#################################################################

CrayPat/X:  Version 6.1.0.10863 Revision 10863 (xf 10658)  02/13/13
15:23:08
Number of PEs (MPI ranks):   64
Numbers of PEs per Node:     32  PEs on each of  2  Nodes
Numbers of Threads per PE:    1
Number of Cores per Socket:  16
Execution start time:  Fri Feb 15 14:42:24 2013
System name and speed:  mork 2100 MHz

Wall Clock Time:  122.608994 secs
High Memory:  45.70 MBytes
MFLOPS (aggregate):  15763.16 M/sec
```

Sample Output (cont'd)

```
Table 1:  Profile by Function Group and Function (top 7 functions shown)

 Time%   |       Time |     Imb. |  Imb. |     Calls |Group
         |            |     Time | Time% |           | Function
         |            |          |       |           |  PE=HIDE

 100.0% | 101.961423 |       -- |    -- | 5315211.9 |Total
|-----------------------------------------------------------------------------
|  92.5% |  94.267451 |       -- |    -- | 5272245.9 |USER
||----------------------------------------------------------------------------
||  75.8% |  77.248585 | 2.356249 |  3.0% |    1001.0 |LAMMPS_NS::PairLJCut::compute
||   6.5% |   6.644545 | 0.105246 |  1.6% |      51.0 |LAMMPS_NS::Neighbor::half_bin_newton
||   4.1% |   4.131842 | 0.634032 | 13.5% |       1.0 |LAMMPS_NS::Verlet::run
||   3.8% |   3.841349 | 1.241434 | 24.8% | 5262868.9 |LAMMPS_NS::Pair::ev_tally
||   1.3% |   1.288463 | 0.181268 | 12.5% |    1000.0 |LAMMPS_NS::FixNVE::final_integrate
||============================================================================
|   7.0% |   7.110931 |       -- |    -- |   42637.0 |MPI
||----------------------------------------------------------------------------
||   4.8% |   4.851309 | 3.371093 | 41.6% |   12267.0 |MPI_Send
||   1.5% |   1.536106 | 2.592504 | 63.8% |   12267.0 |MPI_Wait
|=============================================================================
```

Sample Output (cont'd)

```
================  Observations and suggestions  ========================

MPI Grid Detection:

There appears to be point-to-point MPI communication in a 4 X 2 X 8 grid
    pattern. The execution time spent in MPI functions might be reduced
    with a rank order that maximizes communication between ranks on the
    same node. The effect of several rank orders is estimated below.

    A file named MPICH_RANK_ORDER.Grid was generated along with this
    report and contains usage instructions and the Hilbert rank order
    from the following table.

        Rank    On-Node     On-Node   MPICH_RANK_REORDER_METHOD
       Order   Bytes/PE   Bytes/PE%
                            of Total
                            Bytes/PE

     Hilbert  5.533e+10      90.66%   3
        Fold  4.907e+10      80.42%   2
         SMP  4.883e+10      80.02%   1
  RoundRobin  3.740e+10      61.28%   0
```

Reveal

Porting to a Hybrid or Many-core System

When to Move to a Hybrid Programming Model

- **When code is network bound**
  - Look at collective time, excluding sync time: this goes up as network becomes a problem
  - Look at point-to-point wait times: if these go up, network may be a problem
- **When MPI starts leveling off**
  - Too much memory used, even if on-node shared communication is available
  - As the number of MPI ranks increases, more off-node communication can result, creating a network injection issue
- **When contention of shared resources increases**
- **When you want to exploit heterogeneous nodes**

Cray performance tools and Reveal can help

Tools needed to Create Hybrid Codes

- **A good Programming Environment closes the gap between peak performance and possible performance**
  - A lot more than just a compiler
- **Specific tools needed for identifying the parallelism in an application**
  - Fine-grained profiling: loop level rather than routine
  - Profiling and character looping structures in a complex application
  - Scoping tools for investigating parallelisability of high-level looping structures
  - Tools for maintaining performance-portable applications
    - Application developers want to develop a single core that can run efficiently on multi-core nodes with or without an accelerator

WARNING!!!

- **Nothing comes for free, nothing is automatic**
  - Hybridization of an application is difficult
  - Efficient code requires interaction with the compiler to generate
    - High level OpenMP structures
    - Low level vectorization of major computational areas

- **Performance is also dependent upon the location of the data**
  - CPU: NUMA, first-touch
  - Accelerator: resident or data-sloshing

- **Software such as Cray's Hybrid Programming Environment provides tools to help, but cannot replace the developer's inside knowledge**

Optimizations for Multi-core Systems

- Reduce number of MPI ranks per node
- Add parallelism to MPI ranks to take advantage of cores within a node while minimizing network injection contention
- Maximize on-node communication between MPI ranks
- Relieve on-node shared resource contention by pairing threads or processes that perform different work (for example computation with off-node communication) on the same node
- Accelerate work intensive parallel loops

Approach to Adding Parallelism

1. **Identify possible accelerator kernels**
   - Determine where to add additional levels of parallelism
     - Assumes MPI application is functioning correctly on X86
     - Find top serial work-intensive loops (perftools + CCE loop work estimates)

2. **Perform parallel analysis, scoping and vectorization**
   - Split loop work among threads
     - Do parallel analysis and restructuring on targeted high level loops
     - Use CCE loopmark feedback, Reveal loopmark and source browsing

3. **Move to OpenMP and then to OpenACC**
   - Add parallel directives and acceleration extensions
     - Insert OpenMP directives (Reveal scoping assistance)
     - Run on X86 to verify application and check for performance improvements
     - Convert desired OpenMP directives to OpenACC

4. **Analyze performance from optimizations**

Step 1 - Identify possible accelerator kernels

Loop Work Estimates

- **Helps identify high-level serial loops to parallelize**
  - Based on runtime analysis, approximates how much work exists within a loop
  - Provides min, max and average trip counts that can be used to approximate work and help carve up loop on GPU

Collecting Loop Work Estimates

- Load PrgEnv-cray module
- Load perftools module
- Compile **AND** link with –h profile_generate
- Instrument binary for tracing
  - pat_build –w my_program
- Run application
- Create report with loop statistics
  - pat_report my_program.xf > loops_report

Example Report – Inclusive Loop Time

Table 2: Loop Stats by Function (from -hprofile_generate)

Loop Incl Time Total	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE=HIDE
8.995914	100	25	0	25	sweepy_.LOOP.1.li.33
8.995604	2500	25	0	25	sweepy_.LOOP.2.li.34
8.894750	50	25	0	25	sweepz_.LOOP.05.li.49
8.894637	1250	25	0	25	sweepz_.LOOP.06.li.50
4.420629	50	25	0	25	sweepx2_.LOOP.1.li.29
4.420536	1250	25	0	25	sweepx2_.LOOP.2.li.30
4.387534	50	25	0	25	sweepx1_.LOOP.1.li.29
4.387457	1250	25	0	25	sweepx1_.LOOP.2.li.30
2.523214	187500	107	0	107	riemann_.LOOP.2.li.63
1.541299	20062500	12	0	12	riemann_.LOOP.3.li.64
0.863656	1687500	104	0	108	parabola_.LOOP.6.li.67

Step 2 - Perform parallel analysis, scoping and vectorization
&
Step 3 - Move to OpenMP and then to OpenACC


Reveal 0.1
File
Help
About Reveal
CRAY
REVEAL

Reveal

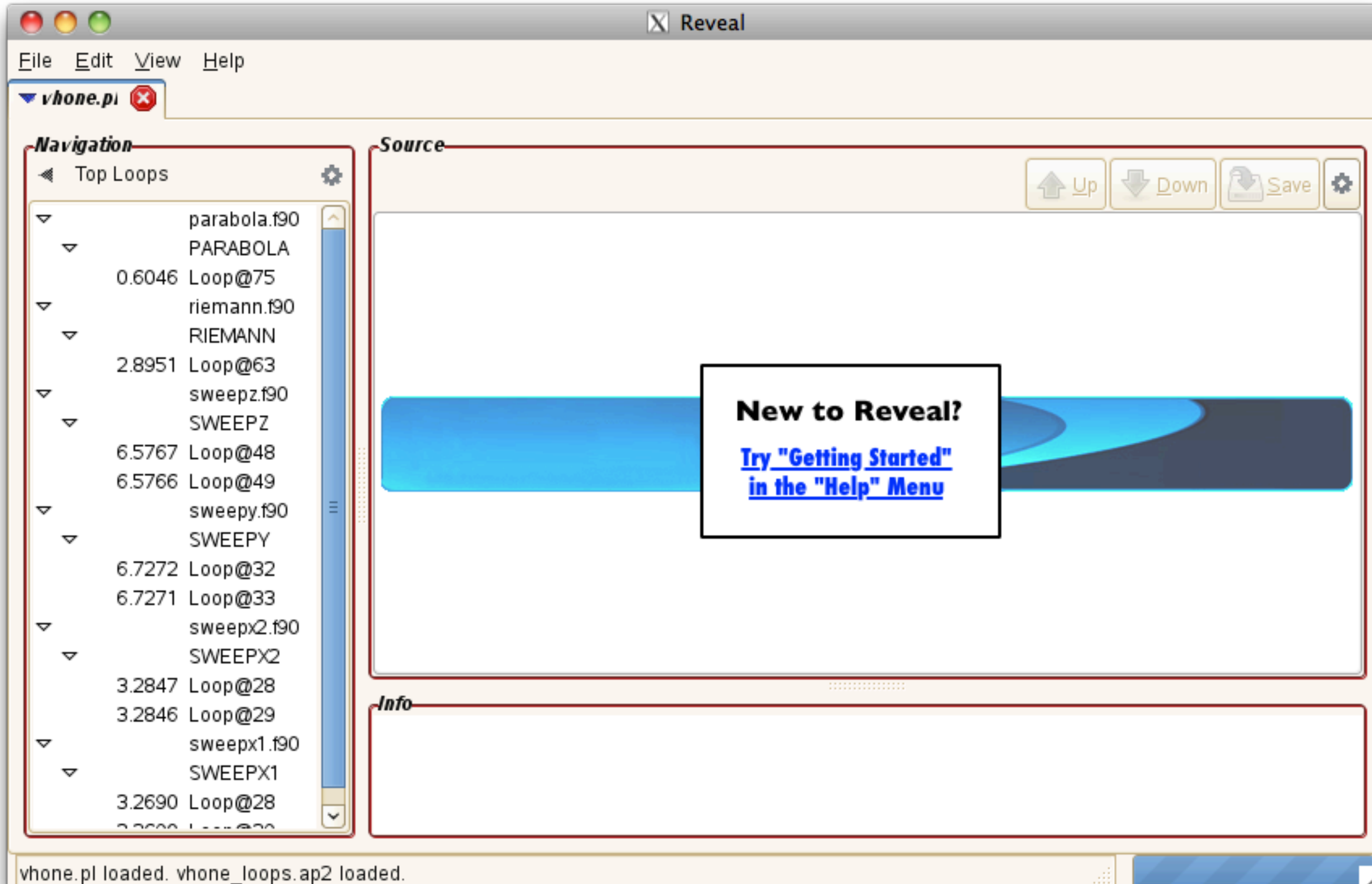
New code analysis and restructuring assistant…

- **Uses both the performance toolset and CCE's program library functionality to provide static and runtime analysis information**
- **Key Features**
  - Annotated source code with compiler optimization information
    - Feedback on critical dependencies that prevent optimizations
  - Scoping analysis
    - Identify, shared, private and ambiguous arrays
      - Allow user to privatize ambiguous arrays
      - Allow user to override dependency analysis
  - Source code navigation based on performance data collected through CrayPat

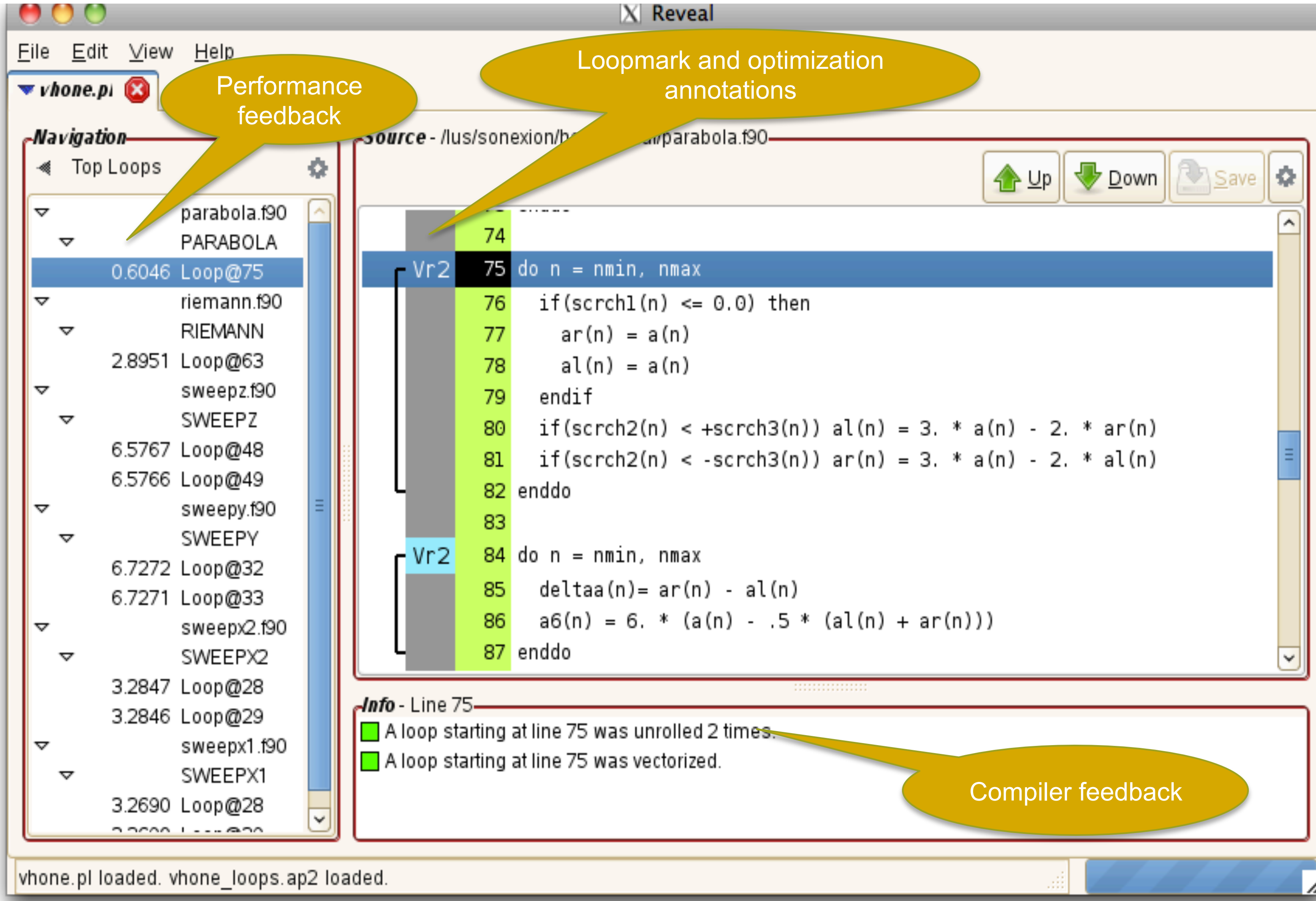
How to Use

- **Optionally create loop statistics using the Cray performance tools to determine which loops have the most work**
- **Compile your application with Cray CCE to generate a program library**
  - > ftn –h pl=vhone.pl –c file1.f90
- **Run reveal**
  - Compiler information only:
    - > reveal vhone.pl
  - Compiler + loop work estimates
    - > reveal vhone.pl vhone_loops.ap2

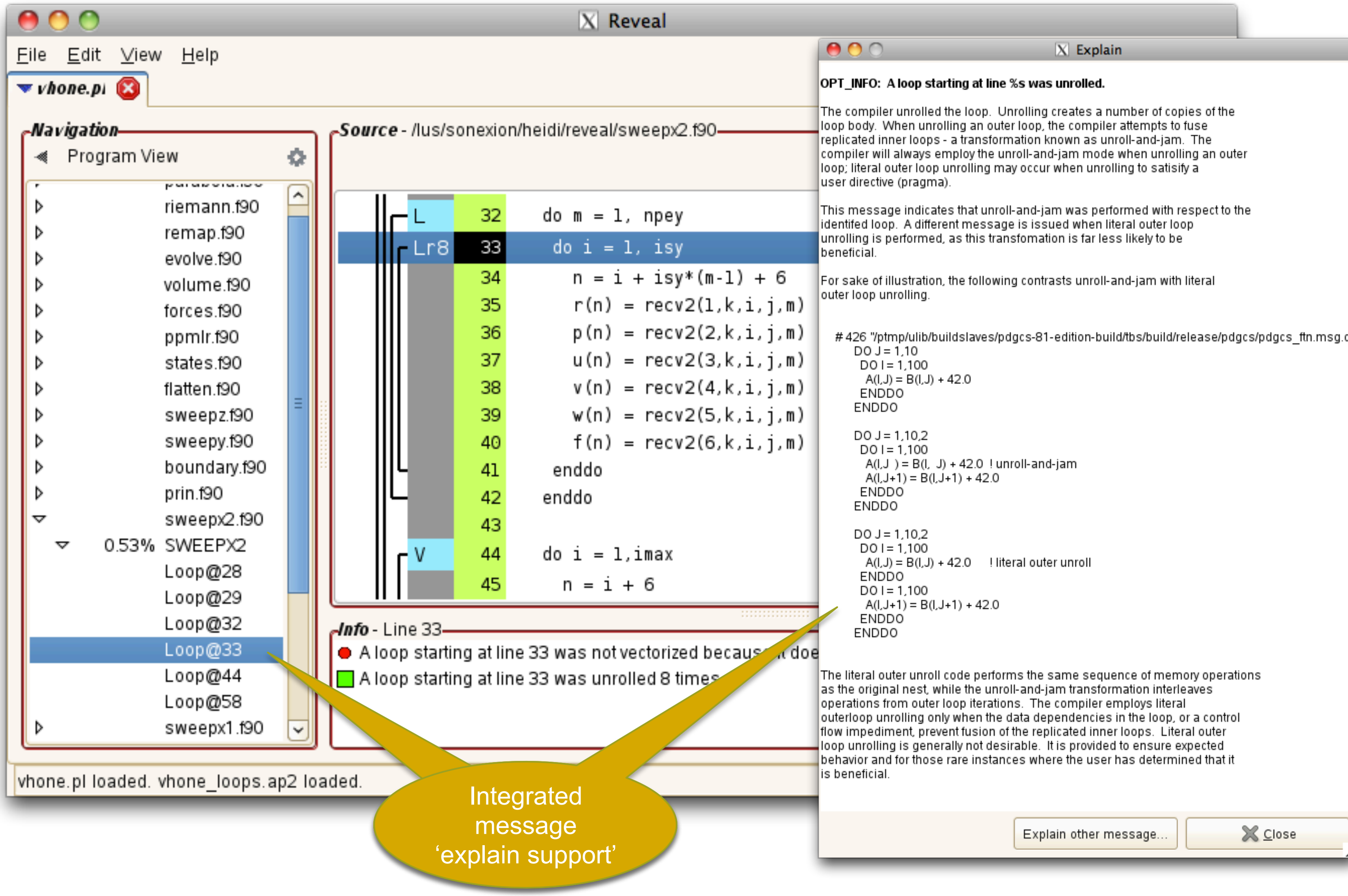
Reveal with Loop Work Estimates



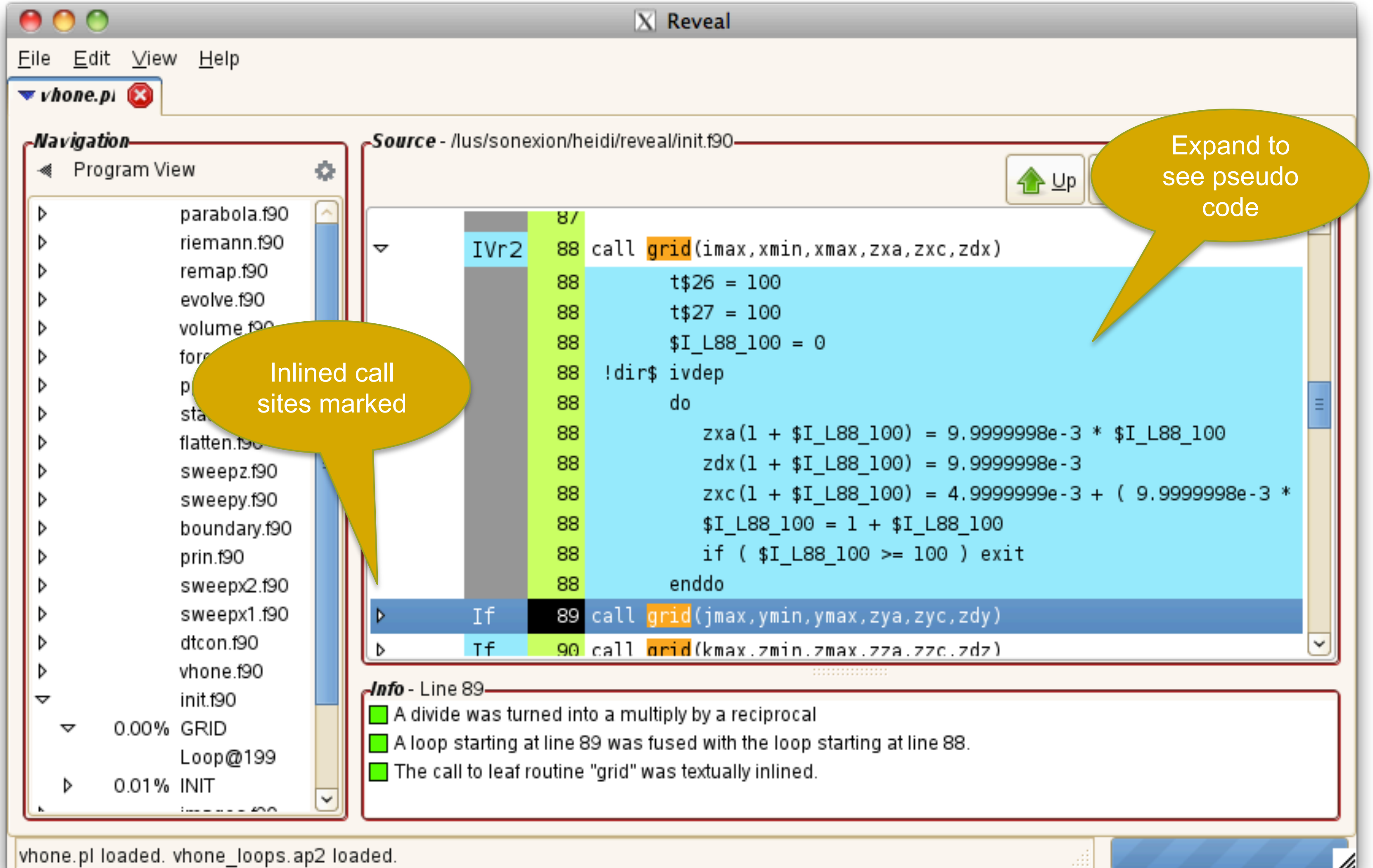
Visualize Loopmark with Performance Information



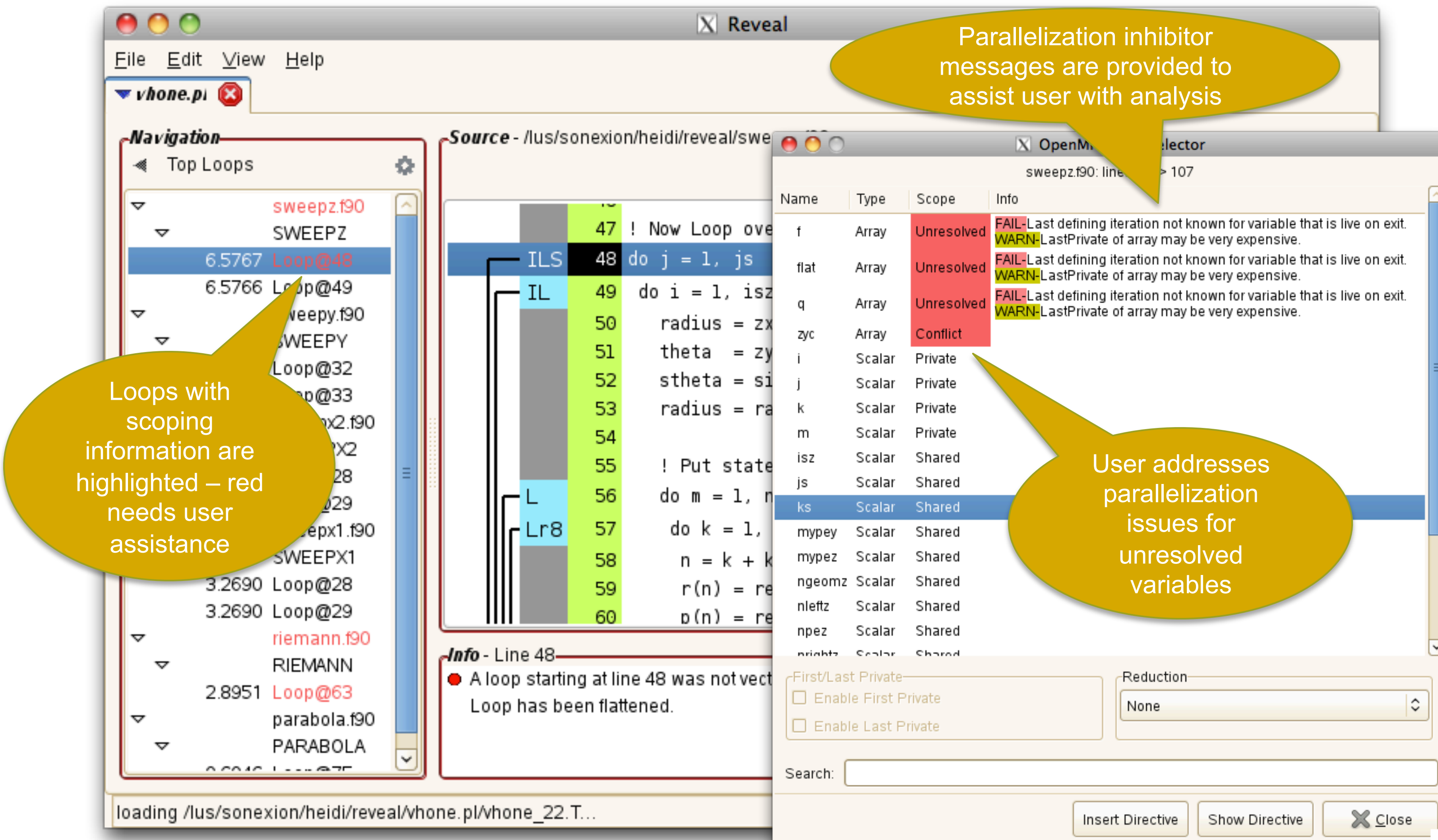
Visualize CCE's Loopmark with Performance Profile (2)



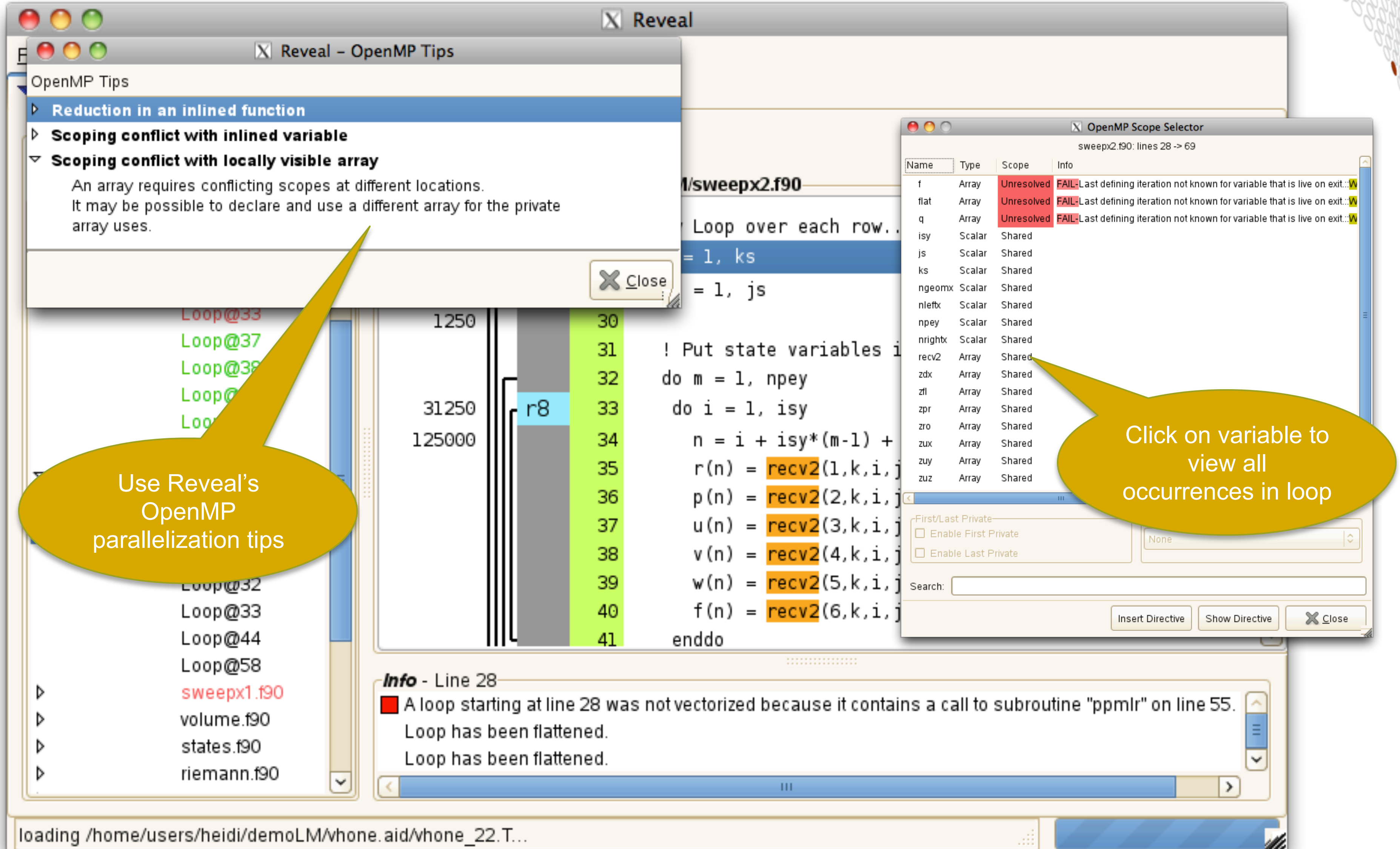
View Pseudo Code for Inlined Functions



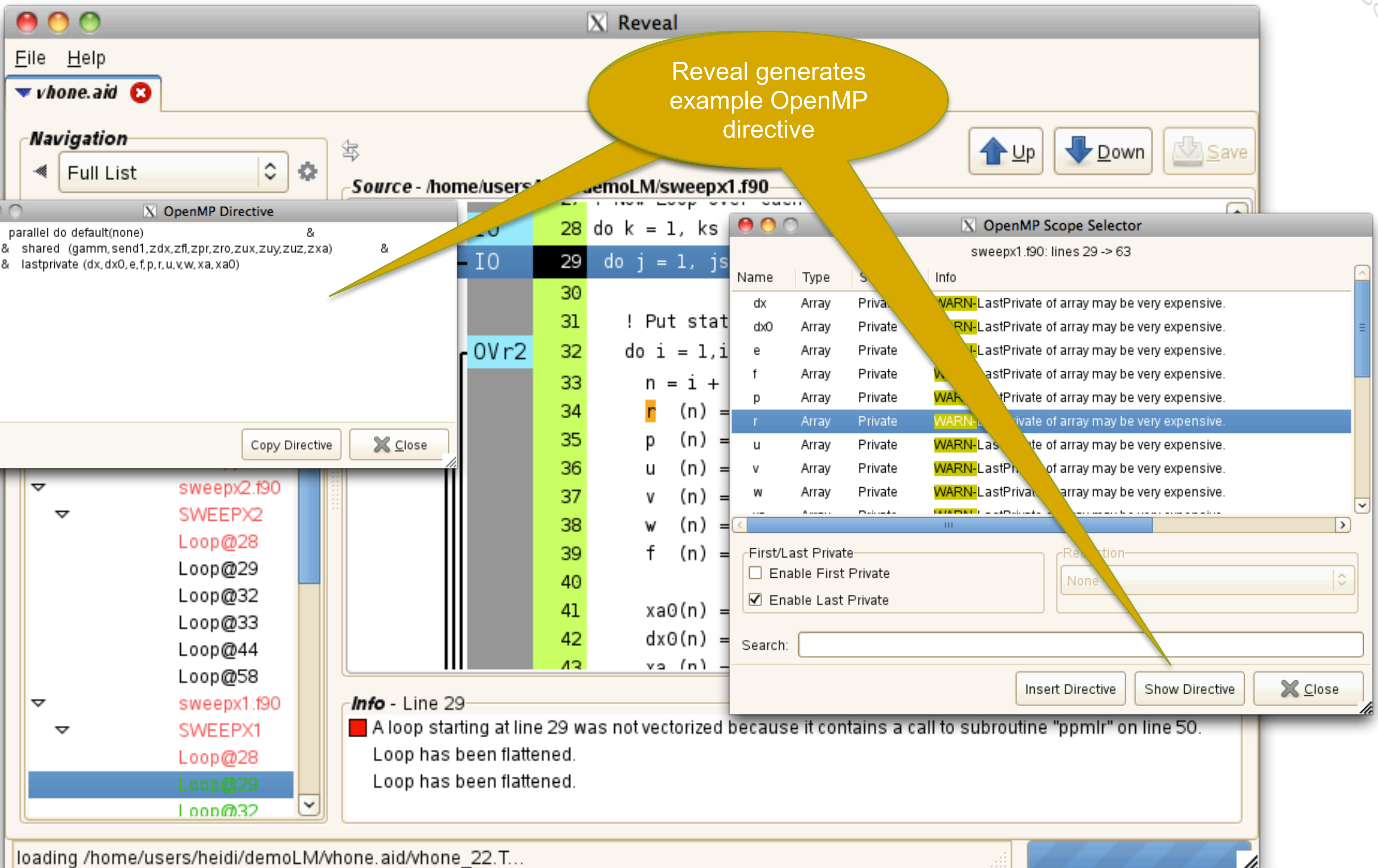
Scoping Assistance – Review Scoping Results



Scoping Assistance – User Resolves Issues



Scoping Assistance – Generate Directive



Questions
?