Performance Analysis with CrayPat

Part 2

Aniello Esposito
(esposito@cray.com)

Outline

- **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**
- **Load balancing Analysis**
  - Using Apprentice2
- **Monitoring Power**
- **Craypat-lite**

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 | Loop Incl |      Time |    Loop |  Loop |  Loop |  Loop |Function=/.LOOP[.]
   Incl |      Time |     (Loop |     Hit | Trips | Trips | Trips | PE=HIDE
  Time% |           |      Adj.) |         |   Avg |   Min |   Max |
|-----------------------------------------------------------------------------
| 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 Feedack and Variable scoping

Reveal

- **For an OpenMP port a developper 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
```

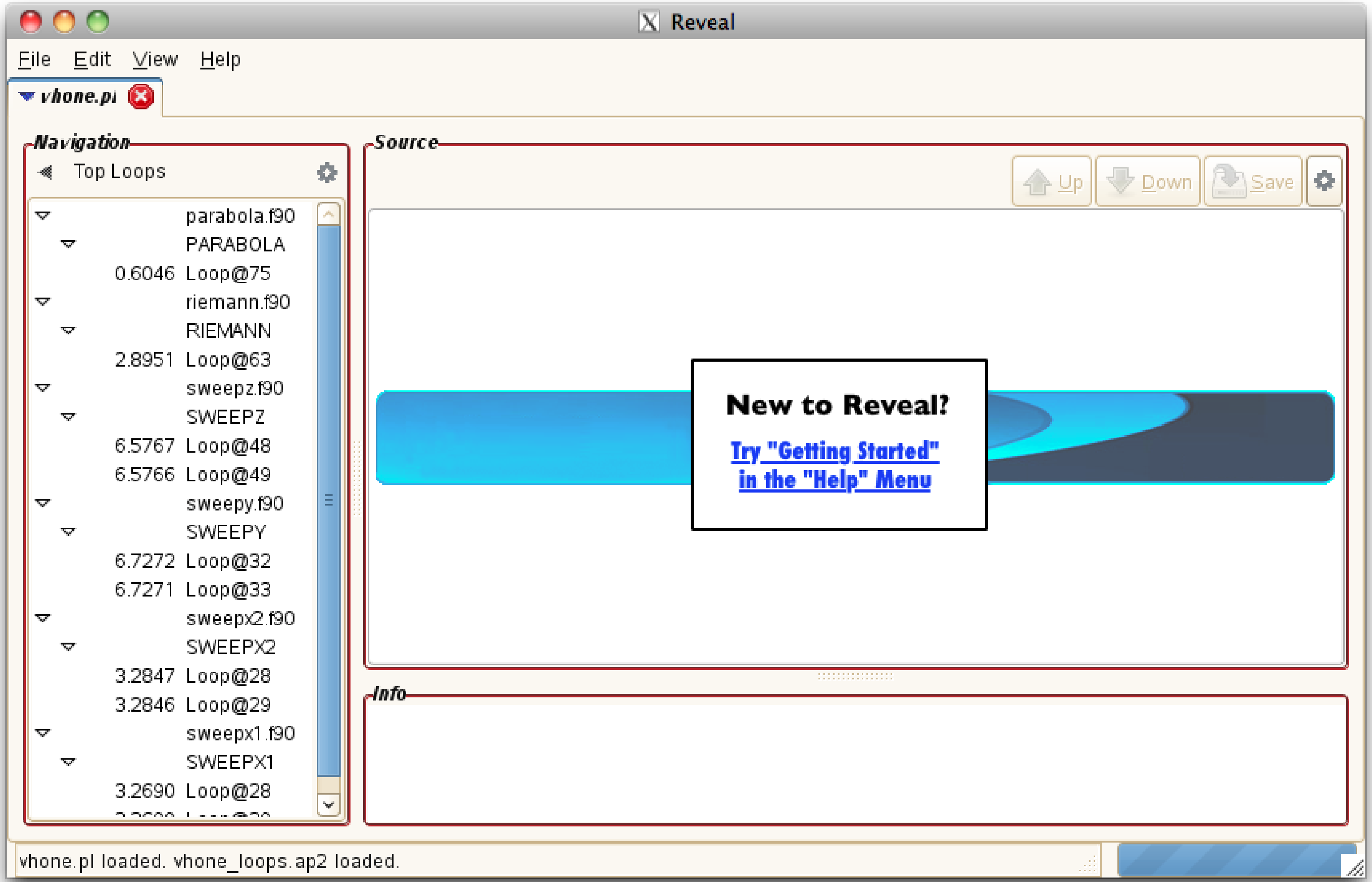
- Recompile only sources to generate program library `my_program.pl`
- The program library is most useful when generated from fully optimized code.
- Use absolute paths to specify the program library if necessary.

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

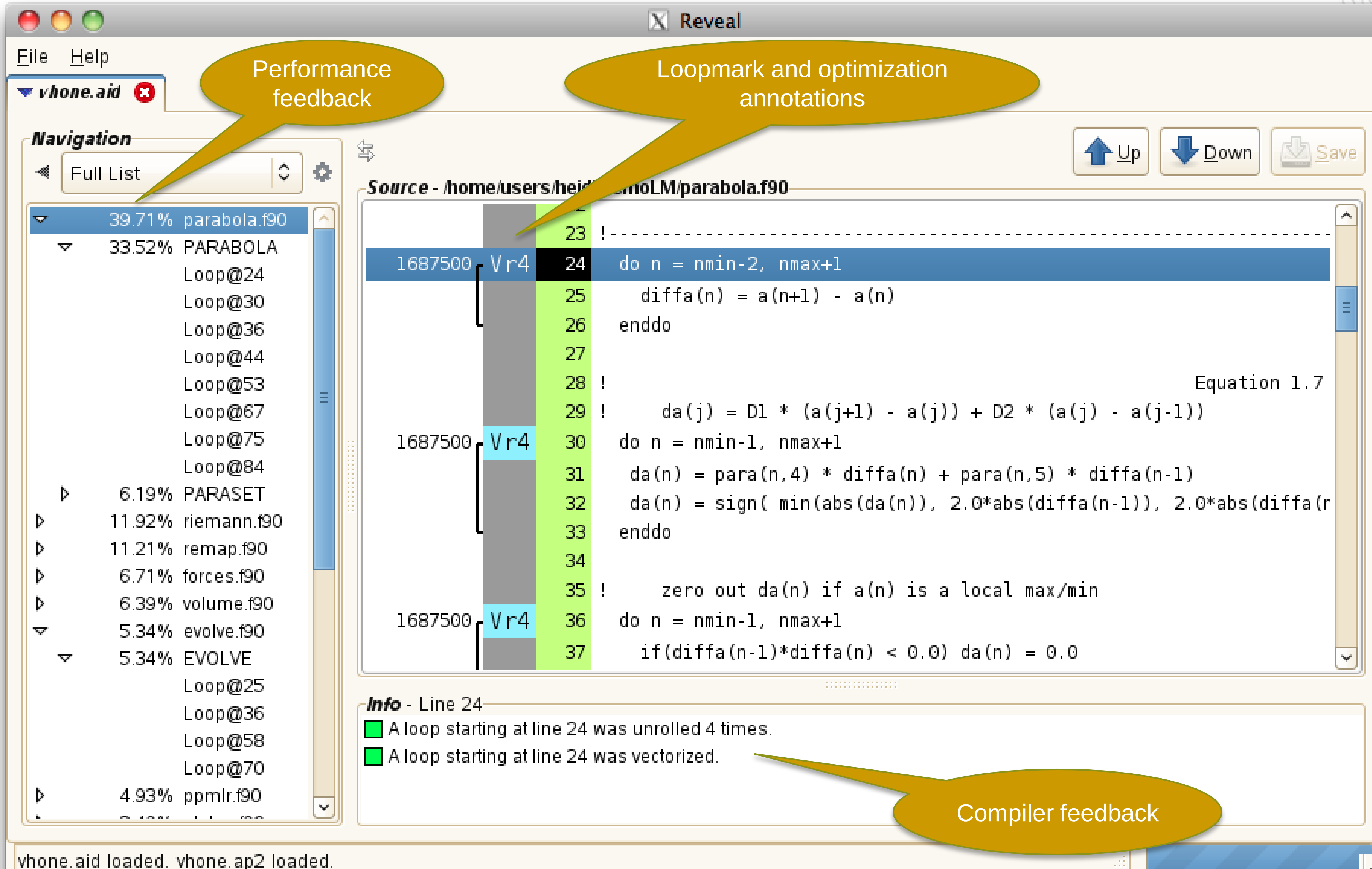
- After the collection of performance data in a separate experiment and generation of a program libary you can launch Reveal.

- **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.

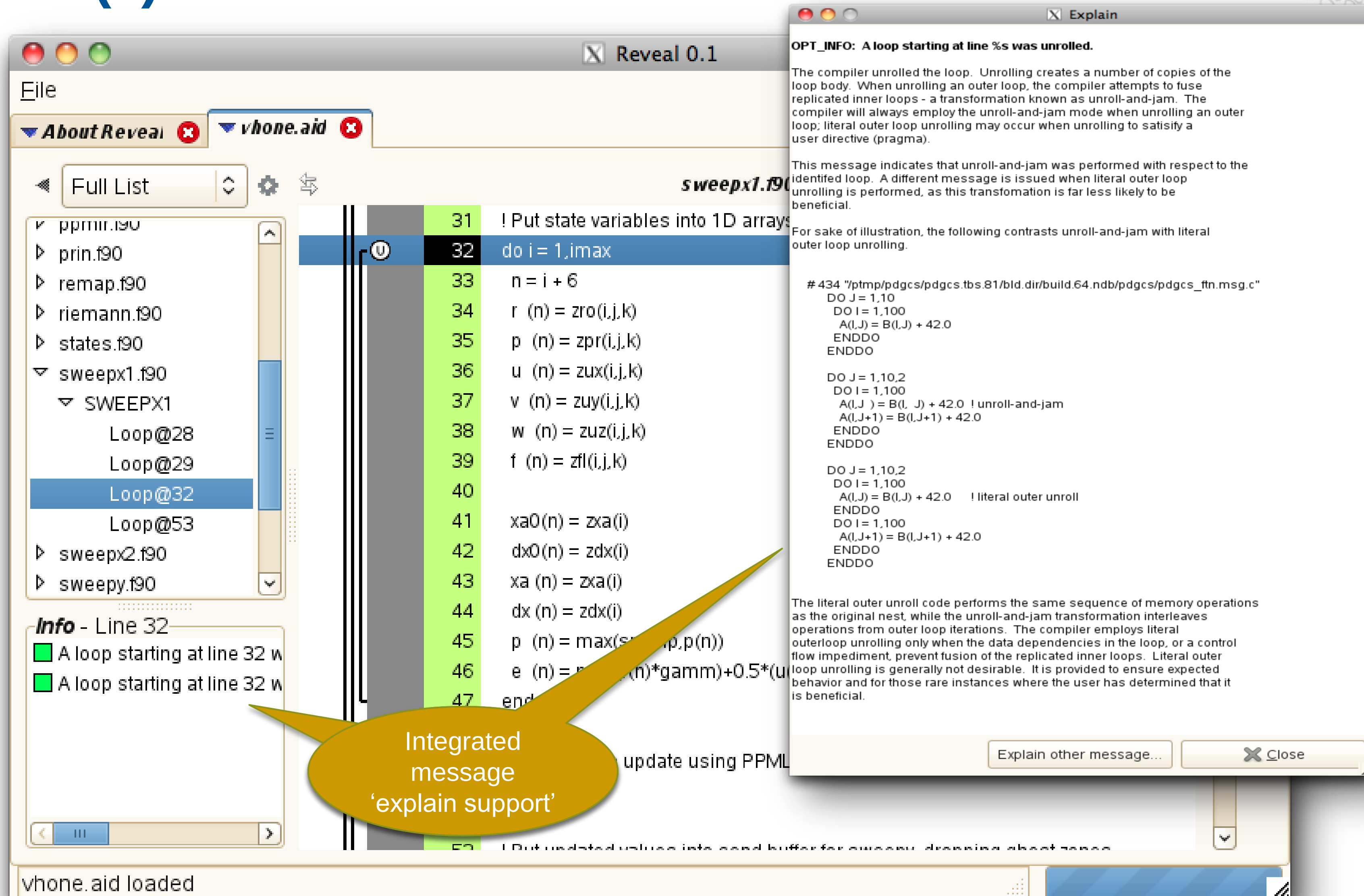
Reveal with Loop Work Estimates



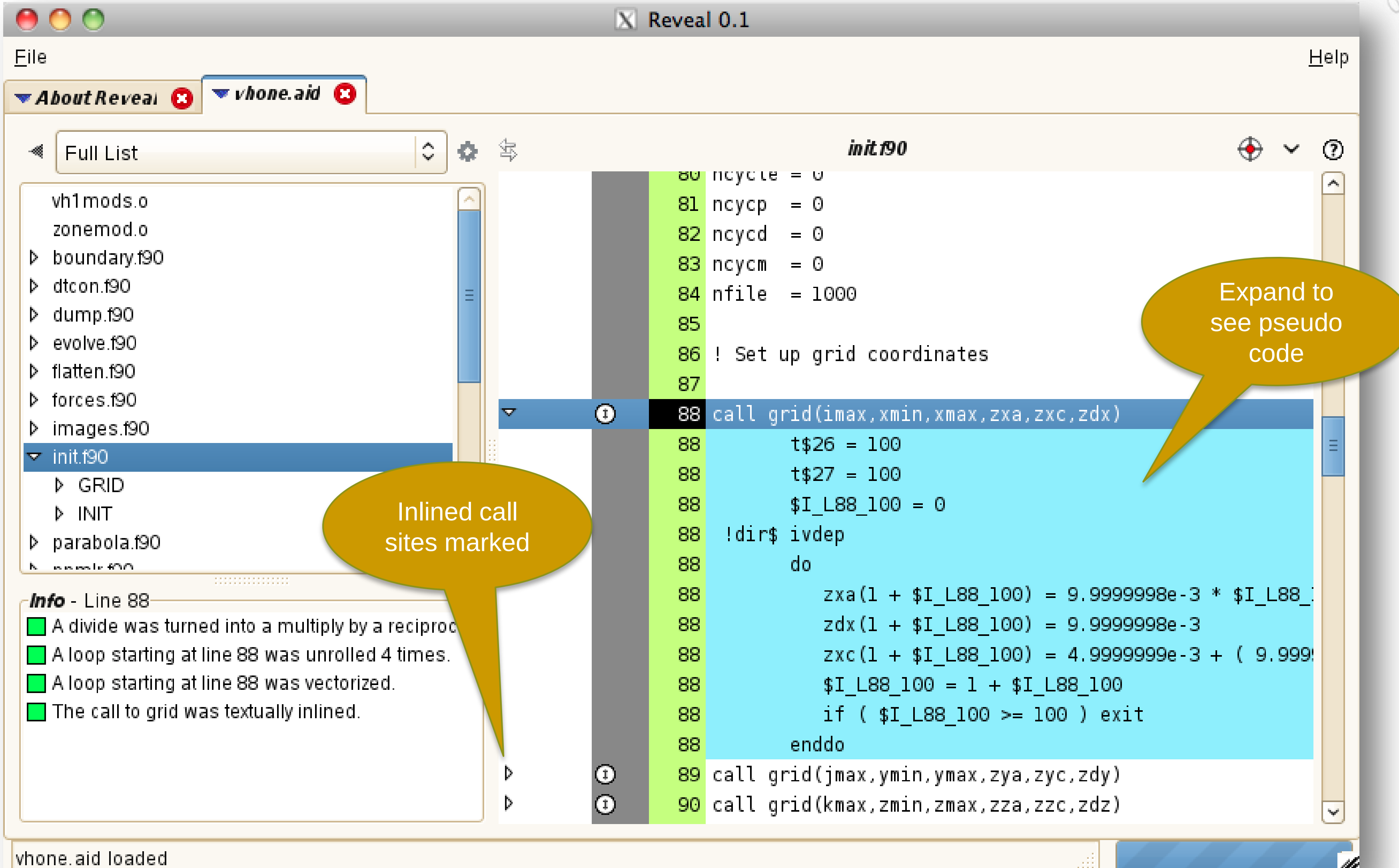
Visualize CCE's Loopmark with Performance Profile



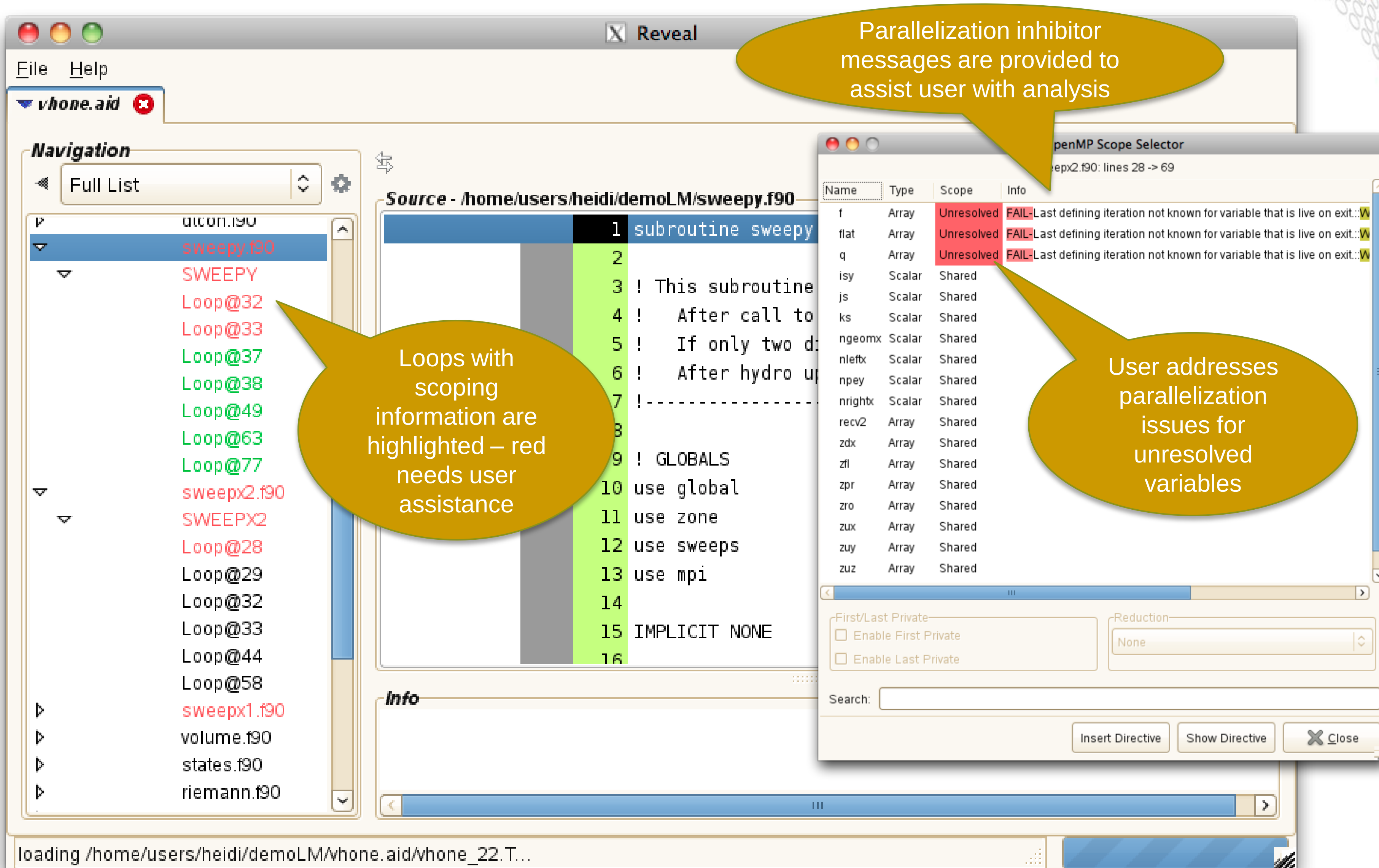
Visualize CCE's Loopmark with Performance Profile (2)



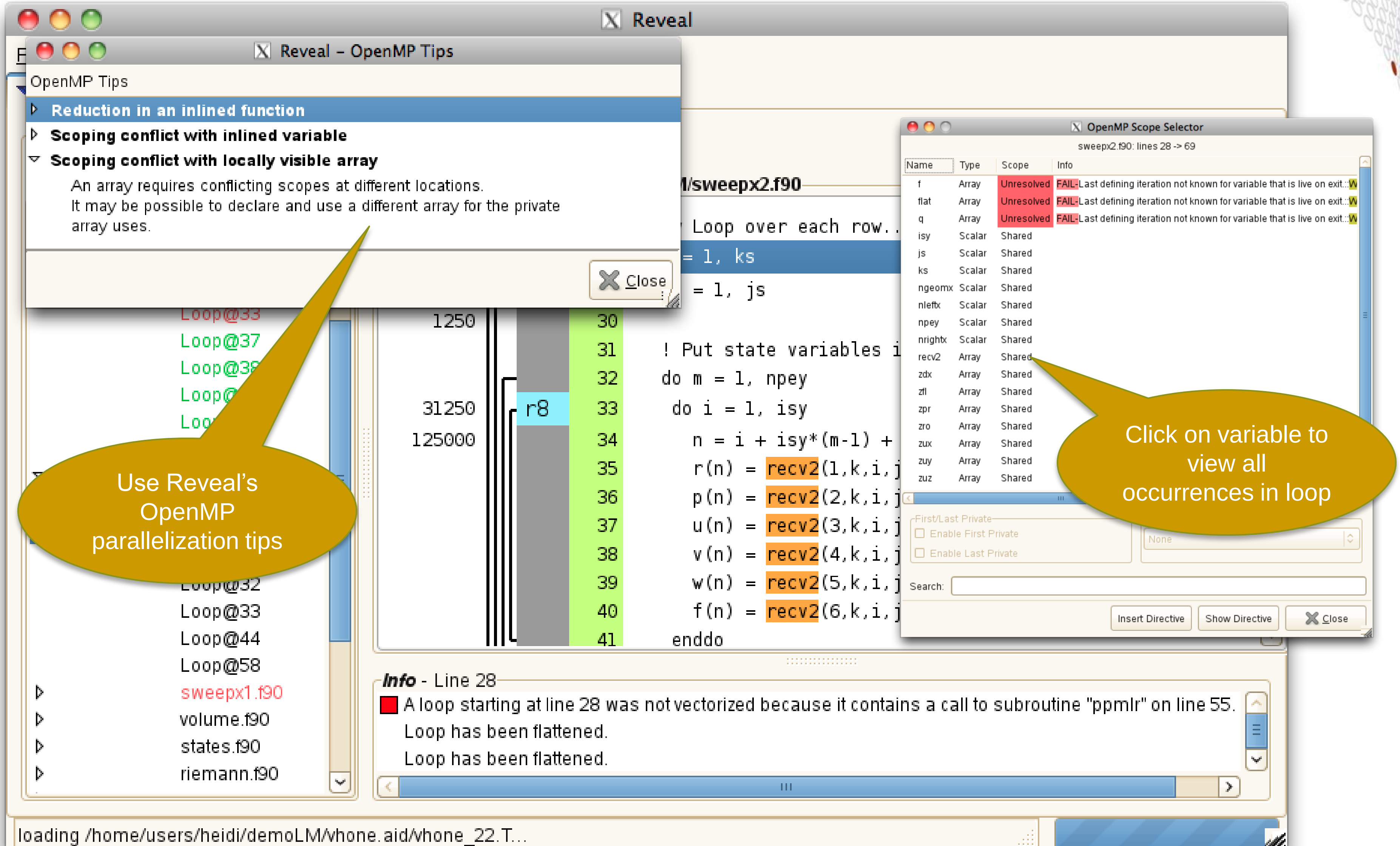
View Pseudo Code for Inlined Functions



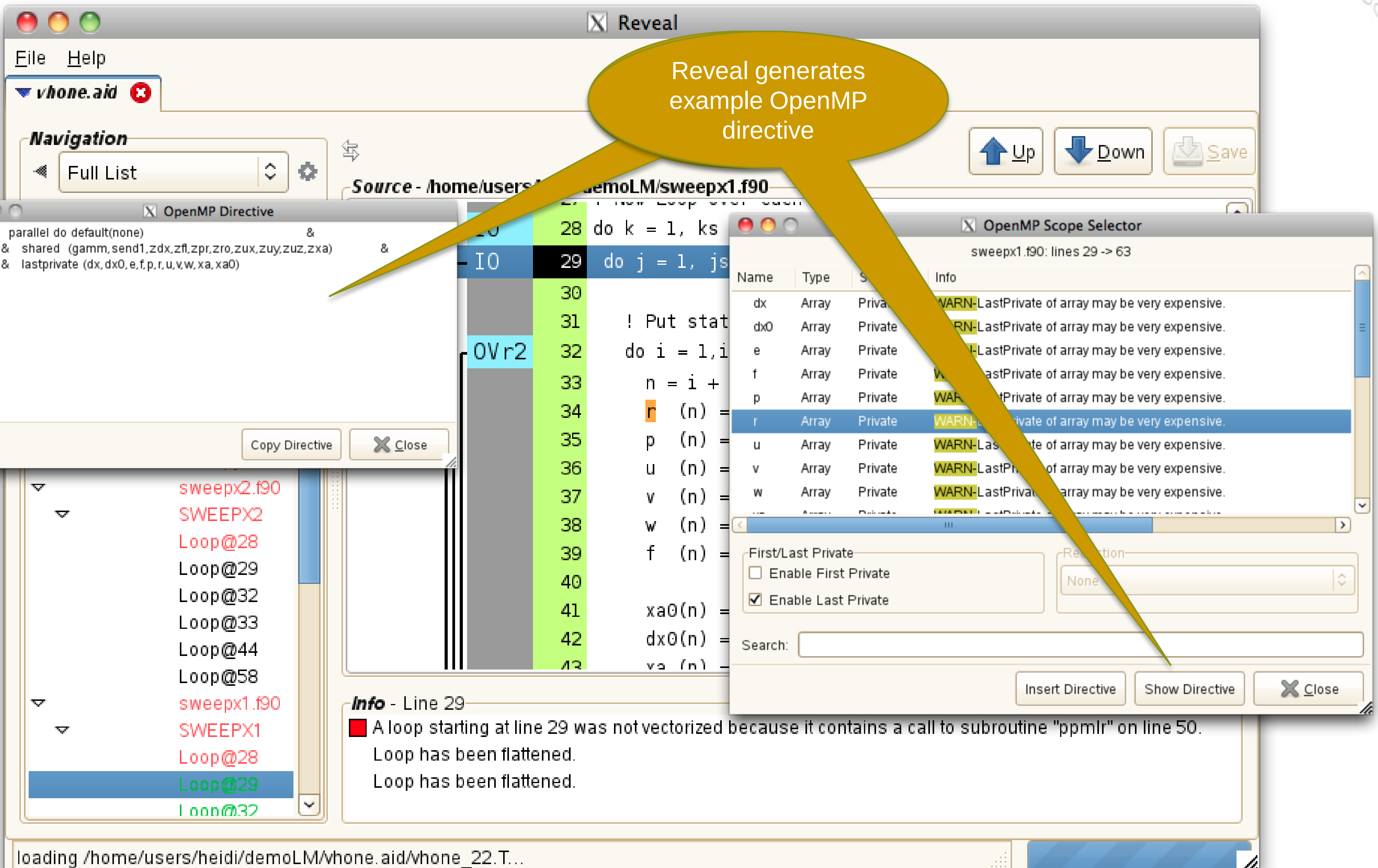
Scoping Assistance – Review Scoping Results



Scoping Assistance – User Resolves Issues



Scoping Assistance – Generate Directive



OpenMP data collection and reporting

- **For programs that use the 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. |  Imb. |  Calls |Group
        |           |     Time | Time% |        | 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. |     Imb. |  Imb.  |Thread
    Time |     Time | Time%  | 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
|=====================================
```

Load Imbalance Analysis

- **Imbalance time is a metric based on execution time and is dependent on the type of activity:**
  - User functions
    Imbalance time = Maximum time – Average time
  - Synchronization (Collective communication and barriers)
    Imbalance time = Average time – Minimum time
- Identifies computational code regions and synchronization calls that could benefit most from load balance optimization
- Estimates how much overall program time could be saved if corresponding section of code had a perfect balance.
- Represents upper bound on "potential savings"
- Assumes other processes are waiting, not doing useful work while slowest member finishes.

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

 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
||==============================================================
|   1.2% |  0.239306 |       -- |     -- |  871.0 |MPI
||--------------------------------------------------------------
||   0.7% |  0.148981 | 0.094595 | 44.4% |  159.0 |MPI_Waitall
||   0.4% |  0.085824 | 0.023669 | 24.7% |  318.0 |MPI_Isend
```

Load Imbalance Analysis

- **Imbalance time percentage represents the percentage of resources available for parallelism that is "wasted".**

$$\text{Imbalance\%} = 100 \ X \ \frac{\text{Imbalance time}}{\text{Max Time}} \ X \ \frac{N}{N - 1}$$

- Corresponds to percentage of time that rest of team is not engaged in useful work on the given function.
- Perfectly balanced code segment has imbalance of zero percentage.
- Serial code segment has imbalance of 100 percent.

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

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
||=================================================================
|   1.2% |  0.239306 |        -- |     -- |  871.0 |MPI
||-----------------------------------------------------------------
||   0.7% |  0.148981 | 0.094595 |  44.4% |   159.0 |MPI_Waitall
||   0.4% |  0.085824 | 0.023669 |  24.7% |   318.0 |MPI_Isend
```

Load Imbalance Analysis

- **MPI Sync time measures load imbalance in programs instrumented to trace MPI functions to determine if MPI ranks arrive at collectives together**
  - Separates potential load imbalance from data transfer
  - Sync times reported by default if MPI functions traced
  - If desired, PAT_RT_MPI_SYNC=0 deactivates this feature
  - Only reported for tracing experiments.

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

 100.0% | 20.643909 |       -- |    -- | 1149.0 |Total
|------------------------------------------------------------------
...
| 0.0% | 0.008614 |       -- |    -- |   59.0 |MPI_SYNC
||-----------------------------------------------------------------
||  0.0% |  0.006696 | 0.006627 | 99.0% |    2.0 |MPI_Barrier(sync)
||  0.0% |  0.001802 | 0.001399 | 77.6% |   55.0 |MPI_Allreduce(sync)
||  0.0% |  0.000061 | 0.000052 | 86.3% |    1.0 |MPI_Init(sync)
||  0.0% |  0.000056 | 0.000051 | 91.7% |    1.0 |MPI_Finalize(sync)
|==================================================================
```

Causes and hints

- **What is causing the load imbalance?**
  - Need profiler reports like CrayPAT gives for the ‘where’
  - Need application expertise for the ‘why’
- **Computation**
  - Is decomposition appropriate?
  - Would reordering ranks help?
- **Communication**
  - Is decomposition appropriate?
  - Would reordering ranks help?
  - Are receives pre-posted?
  - Any All-to-1 communication?
- **I/O**
  - synchronous single-writer I/O will cause significant load imbalance already with a couple of MPI tasks (More on IO tomorrow)

Rank placement

- **The default ordering can be changed using the following environment variable:**

```
export MPICH_RANK_REORDER_METHOD=N
```

- **These are the different values (N) that you can set it to:**
  - N=0: **Round-robin** placement – Sequential ranks are placed on the next node in the list.
    0, 1, 2, 3, 0, 1, 2, 3 (8 tasks on 4 nodes, 2 tasks per node)
  - N=1: (DEFAULT) **SMP-style-** (block-) placement
    0, 0, 1, 1, 2, 2, 3, 3 (8 tasks on 4 nodes, 2 tasks per node)
  - N=2: **Folded** rank placement
    0, 1, 2, 3, 3, 2, 1, 0 (8 tasks on 4 nodes, 2 tasks per node)
  - N=3: **Custom** ordering. The ordering is specified in a file named MPICH_RANK_ORDER.

Rank placement with CrayPat

- **When is rank placement a priori useful?**
  - Point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  - Also shown to help for collectives (alltoall) on subcommunicators
  - Spread out I/O servers across nodes
- **CrayPat can provide the following feedback**

```
================  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
```

```
# The 'Custom' rank order in this file targets nodes with
multi-core
# processors, based on Sent Msg Total Bytes collected for:
#
# Program:      /lus/nid00030/heidi/sweep3d/mod/sweep3d.mpi
# Ap2 File:     sweep3d.mpi+pat+27054-89t.ap2
# Number PEs:   48
# Max PEs/Node: 4
#
# To use this file, make a copy named MPICH_RANK_ORDER, and
set the
# environment variable MPICH_RANK_REORDER_METHOD to 3 prior
to
# executing the program.
#
# The following table lists rank order alternatives and the
grid_order
# command-line options that can be used to generate a new
order.
…
0,532,64,564,32,572,96,540,8,596,72,524,40,604,24,588
104,556,16,628,80,636,56,620,48,516,112,580,88,548,120,612
1,403,65,435,33,411,97,443,9,467,25,499,105,507,41,475
73,395,81,427,57,459,17,419,113,491,49,387,89,451,121,483
…
```

Hybrid MPI + OpenMP?

- **OpenMP may help**
  - Able to spread workload with less overhead
  - Large amount of work to go from all-MPI to (better performing) hybrid - must accept challenge to hybridize large amount of code
- **When does it pay to add OpenMP to my MPI code?**
  - Add OpenMP when code is network bound
  - Adding OpenMP to memory bound codes may aggravate memory bandwidth issues, but you have more control when optimizing for cache
  - 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
  - If an all-to-all communication pattern becomes a bottleneck, hybridization often overcomes this
  - Hybridization can be used to avoid replicated data

Cray Apprentice^2

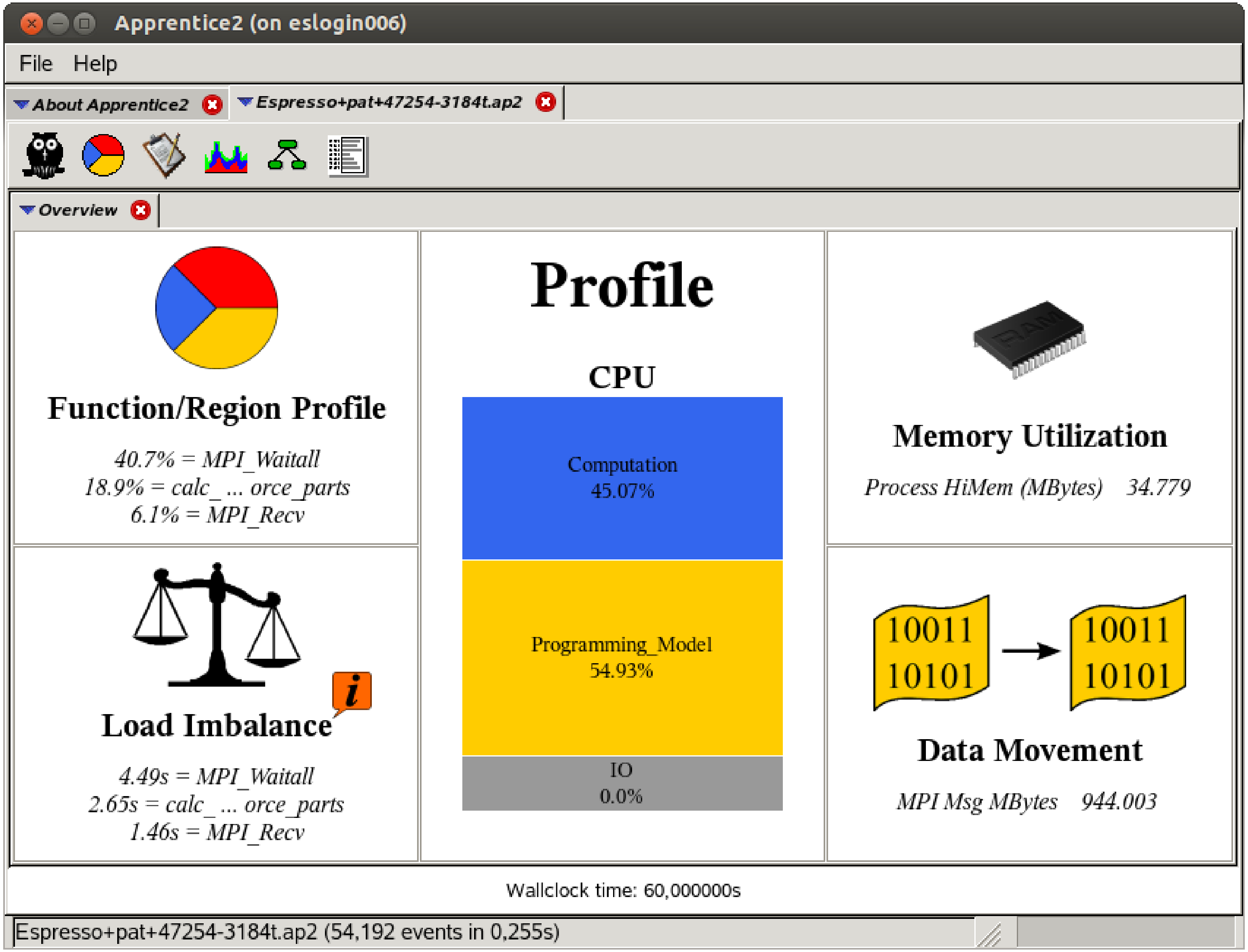
- **Cray Apprentice2 is a post-processing performance data visualization tool. Takes *.ap2 files as input.**

- **Main features are**
  - Call graph profile
  - Communication statistics
  - Time-line view for Communication and IO.
  - Activity view
  - Pair-wise communication statistics
  - Text reports
  - Source code mapping

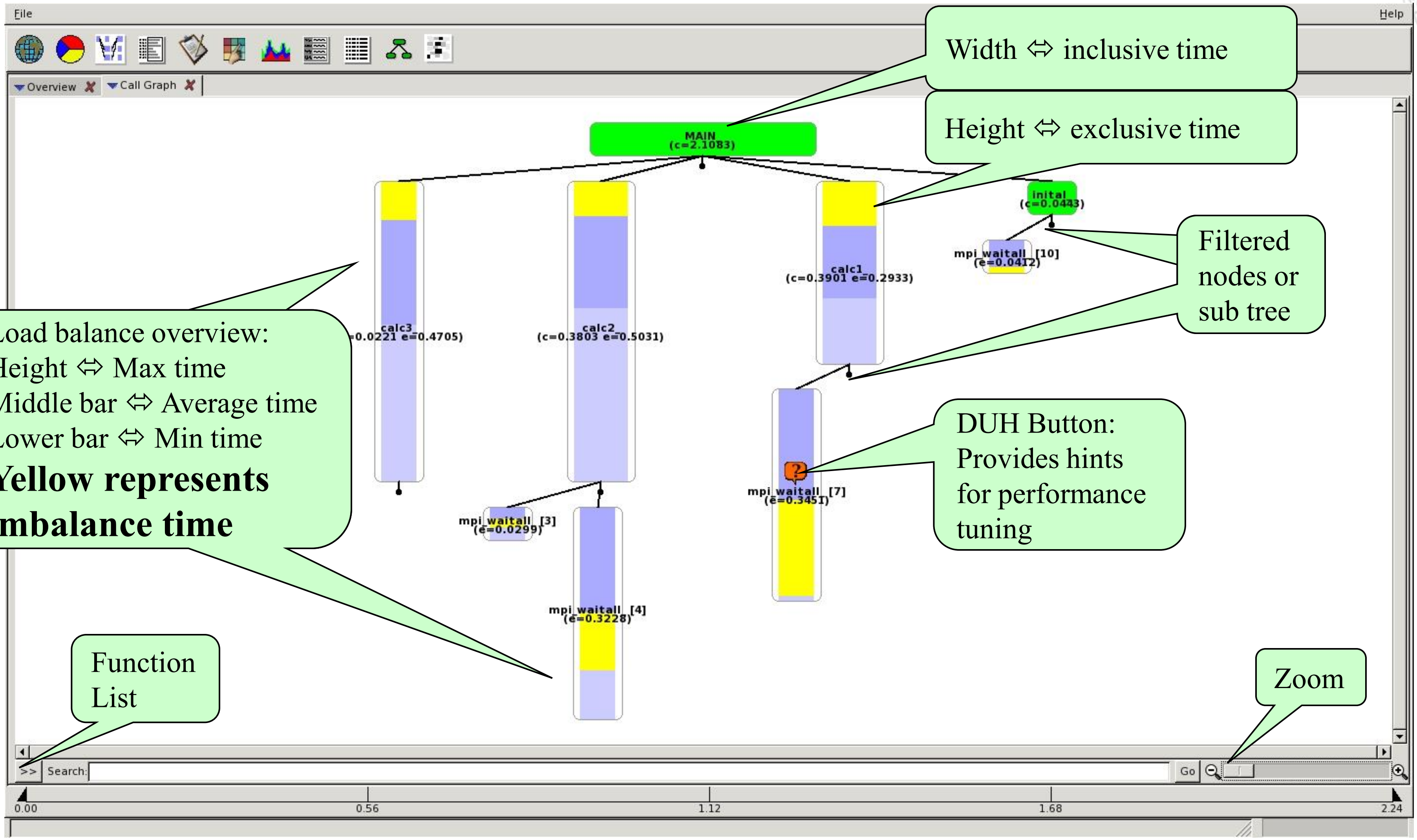
- **Cray Apprentice^2 helps identify:**
  - Load imbalance
  - Excessive communication
  - Network contention
  - Excessive serialization
  - I/O Problems

```
> module load perftools
> app2 my_program.ap2 &
```

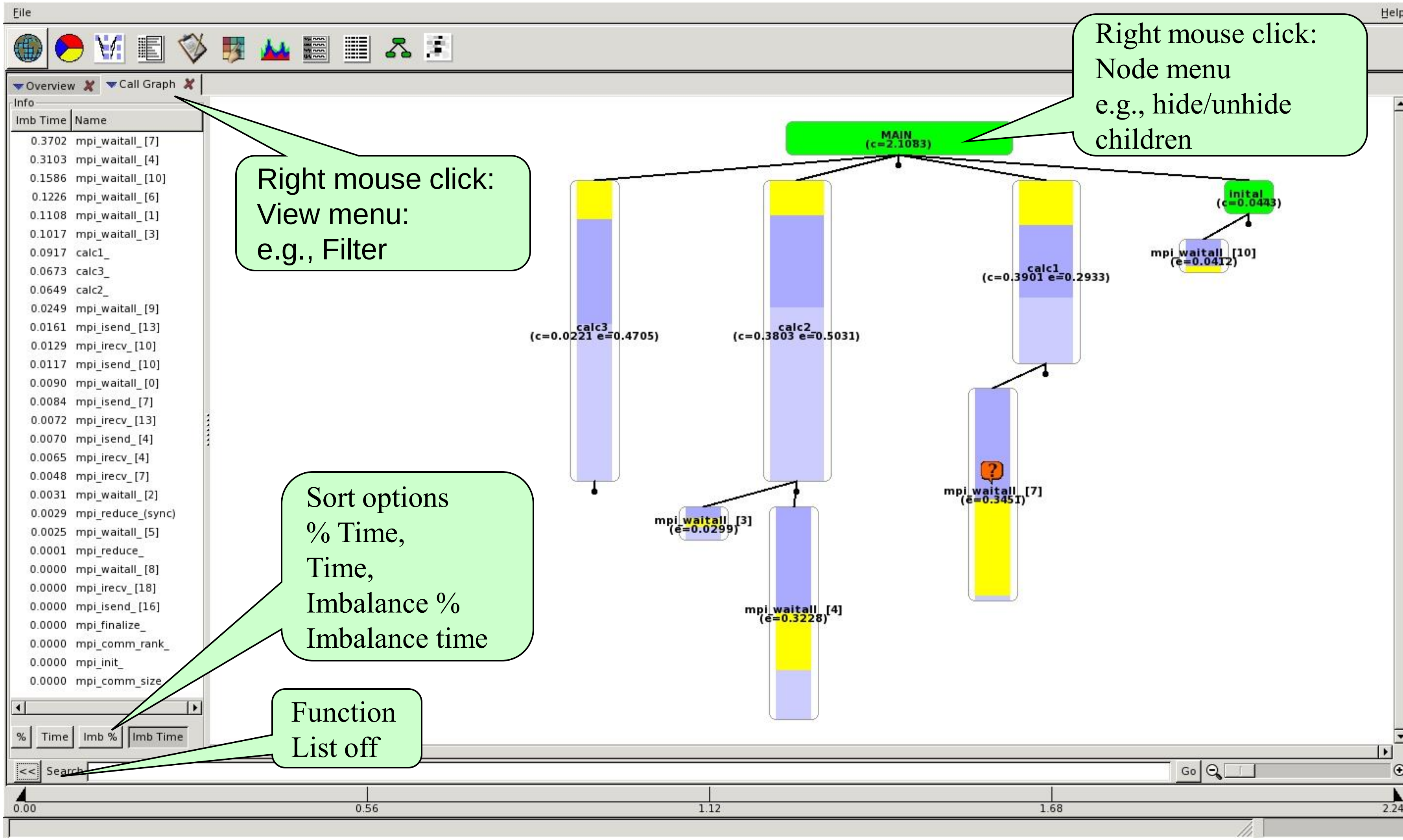
Cray Apprentice²



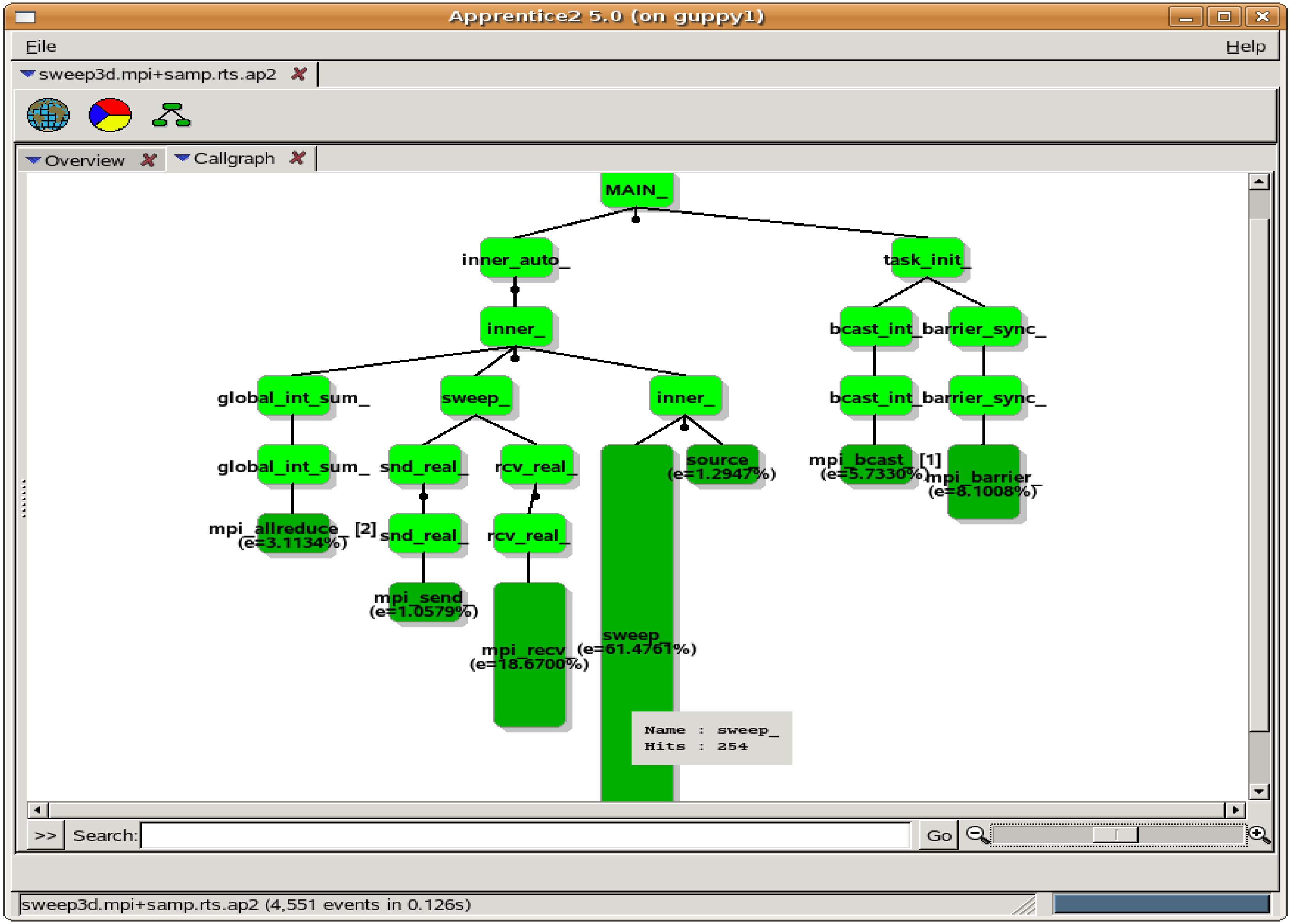
Call Tree View



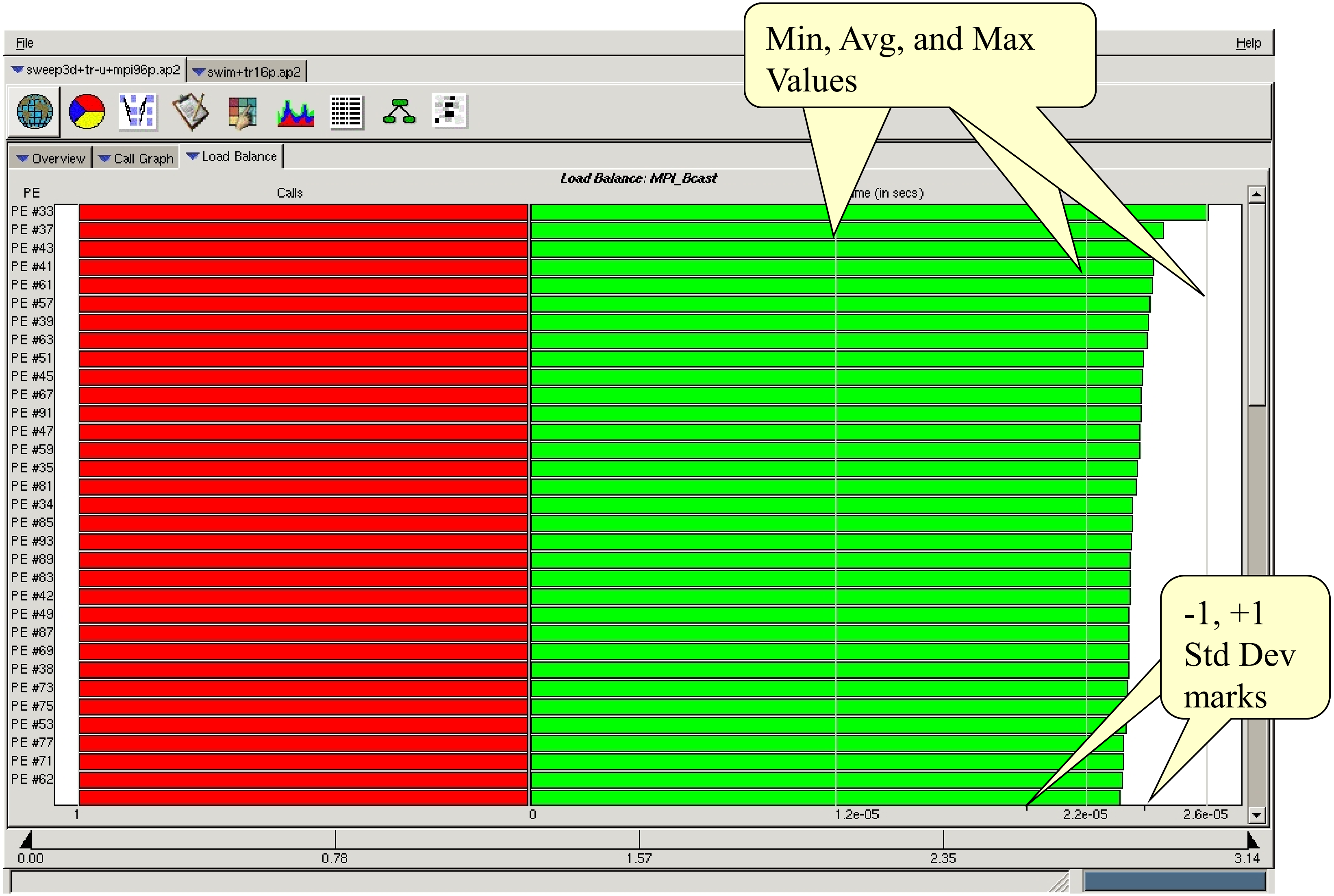
Call Tree View – Function List



Apprentice[2] Call Tree View of Sampled Data

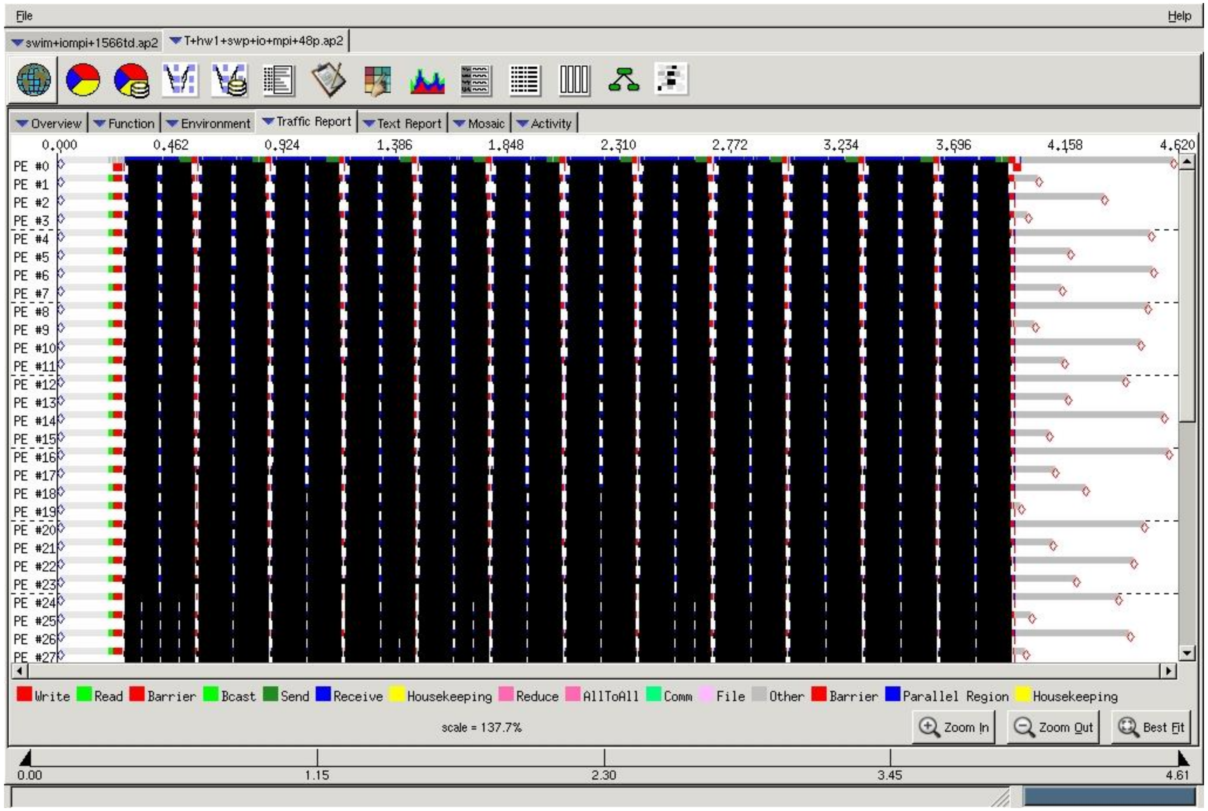


Load Balance View (from Call Tree)

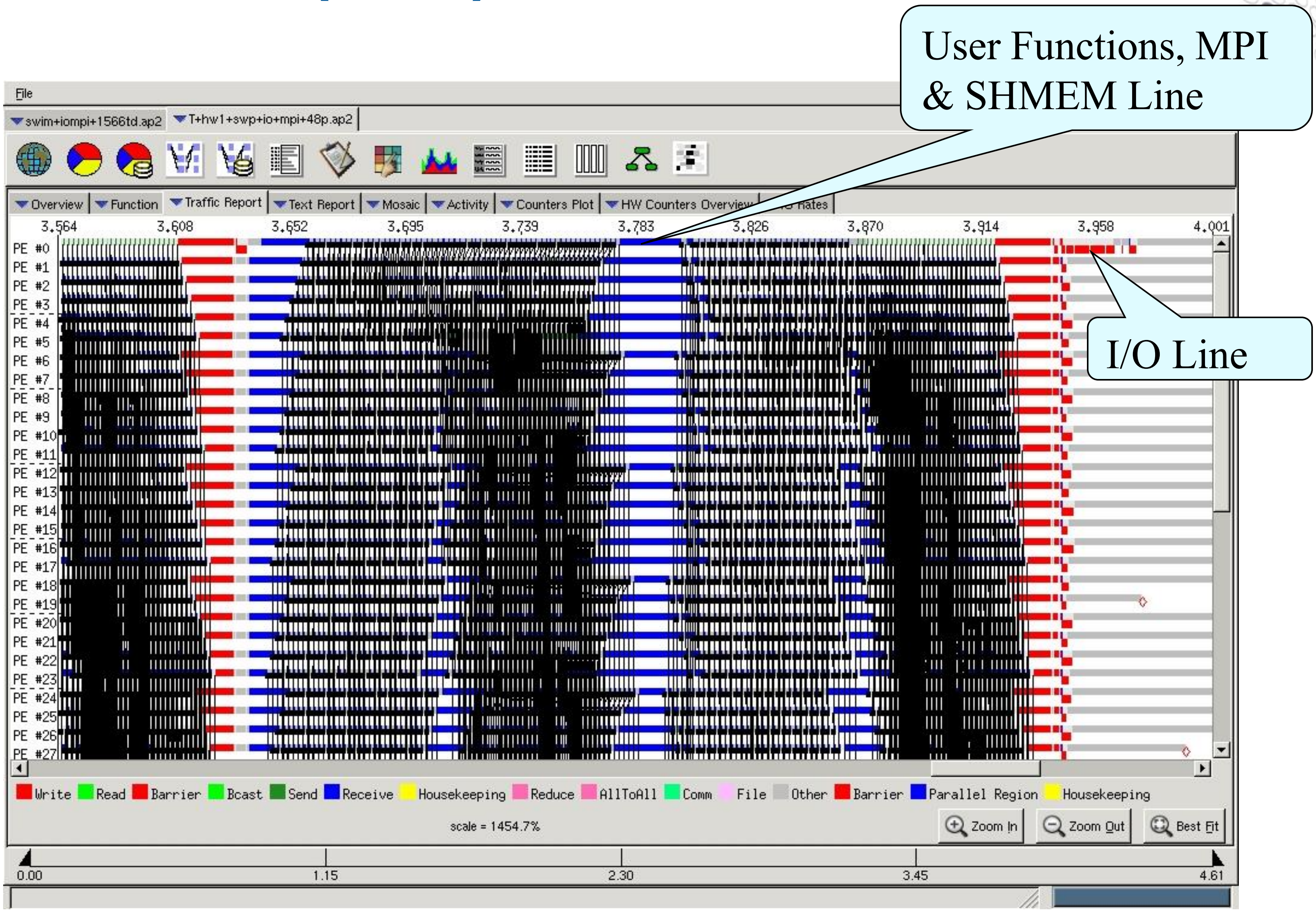


Time Line View

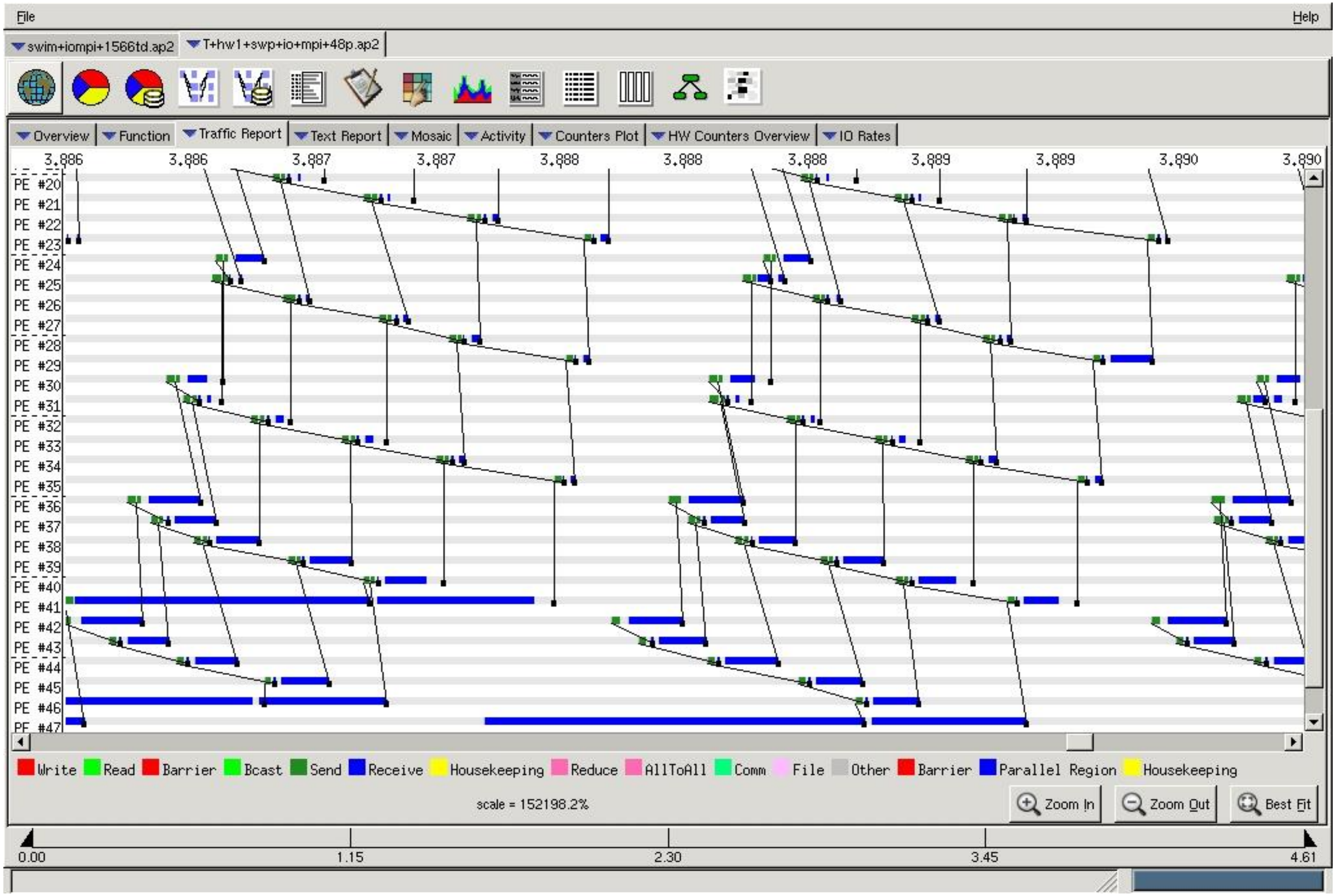
- Full trace (sequence of events) enabled by setting `PAT_RT_SUMMARY=0`
- Helpful to see communication bottlenecks.
- Use it only for small experiments !



Time Line View (Zoom)



Time Line View (Fine Grain Zoom)



Monitoring Power

(Courtesy of Heidi Poxon)

Monitoring Power on Intel

- Feedback to the user on performance and power consumption will be key to understanding the behavior of an applications on future systems.
- To see a list of events, execute the following on compute node:

```
> aprun papi_native_avail -i crayrapl

> aprun papi_native_avail -i craypm
```

- See `rapl(5)` and `pmpc(5)` man pages for more information

Intel's RAPL (Running Average Power Level)

- **Provides mechanism to enforce power consumption limit. Also facilitates the ability to measure energy consumption on SNB and IVB processors**
- **32-bit counter measurements available on a per socket basis with update frequency of approximately 1 millisecond**
- **7 monitoring counter events available**
  - Provides dynamic readings from various components of the socket
  - Constant values are available for thermal specifications, max and min power caps, and time windows

Event	Description
PACKAGE_ENERGY	Total amount of energy consumed by socket
PP0_ENERGY	Total amount of energy consumed by the cores
DRAM_ENERGY	Total amount of energy consumed by the DRAM

Cray Power and Energy Management Counters

- Support CLE Intel power and energy management performance counters
- Provides **compute node-level access** to additional power management counters at the application level
- Enables user to monitor and report energy usage during program execution for both CPU and GPU

Event	Description
PM_POWER_CAP:NODE	Compute node power cap
PM_POWER:NODE	Compute node point in time power
PM_ENERGY:NODE	Compute node accumulated energy
PM_POWER_CAP:ACC	Accelerator power cap
PM_POWER:ACC	Accelerator point in time power
PM_ENERGY:ACC	Accelerator accumulated energy

Accessing Power Information

```
> export PAT_RT_PERFCTR=PM_ENERGY:NODE
> export PAT_RT_PERFCTR=PACKAGE_ENERGY
```

- **RAPL counters**
  - Launch application with aprun -cc cpu to bind MPI ranks to sockets
  - Counters collected by processor 0 on each socket (assuming the application is running on processor 0)
  - 32-bit RAPL counters have a wraparound time of approximately 60 seconds when power consumption is high
- **Cray PM counters**
  - Collected by processor 0 on each node (assuming the application is running on processor 0)
- **Counter collection has high overhead (RAPL higher than Cray PM). It's best not to collect performance information at the same time**

PM Counters for CP2K MPI+OpenMP on IVB

```
USER / process_mm_stack$dbcsr_mm_stack_
-----------------------------------------------------
  Time%                                        2.0%
  Time                                    15.642021 secs
  Imb. Time                                7.142276 secs
  Imb. Time%                                  32.7%
  Calls                     0.005M/sec      72311.2 calls
  PM_ENERGY:NODE           51.384 /sec      803.750 J


=====================================================
  USER / build_core_hamiltonian_matrix$qs_core_hamiltonian_
-----------------------------------------------------
  Time%                                        1.5%
  Time                                    11.564295 secs
  Imb. Time                                2.392148 secs
  Imb. Time%                                  17.9%
  Calls                      1.902 /sec         22.0 calls
  PM_ENERGY:NODE           42.847 /sec      495.500 J
```

CrayPat-lite

Light-weight application profiling

CrayPat-lite Overview

- **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.**

- **The tool is enabled by loading a module and rebuild**

```
> module load perftools-lite
> make clean && make
```

- **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

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

Predefined Set of Performance Experiments

- **Set of predefined experiments, enabled with the `CRAYPAT_LITE` environment variable (before compilation)**
  - `sample_profile`
  - `event_profile`
- **The `sample_profile` is equivalent to**

```
> pat_build -O apa a.out
```

  - Includes collection of summary CPU performance counters around MAIN
  - Includes Imbalance information.
- **The `event_profile` is equivalent to**

```
> pat_build -u -gmpi a.out
```

  - Provides profile based on summarization of events.
  - Includes OpenMP if these models are used within program.
  - Collection of summary CPU performance counters
  - Filter to only trace functions above 1200 bytes
    - In most cases, omits tiny repetitive functions that can perturb results.

Performance Statistics Available

- **Job information**
  - Number of MPI ranks, …
  - Wallclock
  - Memory high water mark
  - Performance counters (CPU only)

```
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

Wall Clock Time:  122.608994  secs
High Memory:  45.70  MBytes
```

- **Profile of top time consuming routines with load balance**

```
Samp%  |  Samp  | Imb.  |  Imb.  |Group
       |        | Samp  | Samp%  | Function
       |        |       |        |  PE=HIDE

100.0% | 14272.5 |   -- |    -- |Total
|-----------------------------------------
| 46.0% | 6561.4 |   -- |    -- |USER
||----------------------------------------
||  5.9% |  847.6 | 155.4 | 15.7% |collocate_core_1_
||  4.9% |  700.3 | 125.7 | 15.5% |integrate_core_2_
||  3.8% |  544.0 | 124.0 | 18.9% |collocate_core_2_
||  3.7% |  523.1 |  73.9 | 12.6% |integrate_core_1_
||========================================
| 29.7% | 4239.6 |   -- |    -- |MPI
||----------------------------------------
||  9.3% | 1328.3 | 198.7 | 13.2% |mpi_alltoallv
||  4.2% |  598.5 |  71.5 | 10.8% |mpi_waitall
||  2.9% |  413.8 | 107.2 | 20.9% |MPI_WAITANY
||  2.9% |  409.1 |  66.9 | 14.3% |MPI_Comm_create
||========================================
```

```
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
||==========================================================
```

- **Observations and Instructions on how to get more info.**