Hewlett Packard Enterprise

Introduction to perftools

Comprehensive General LUMI Course

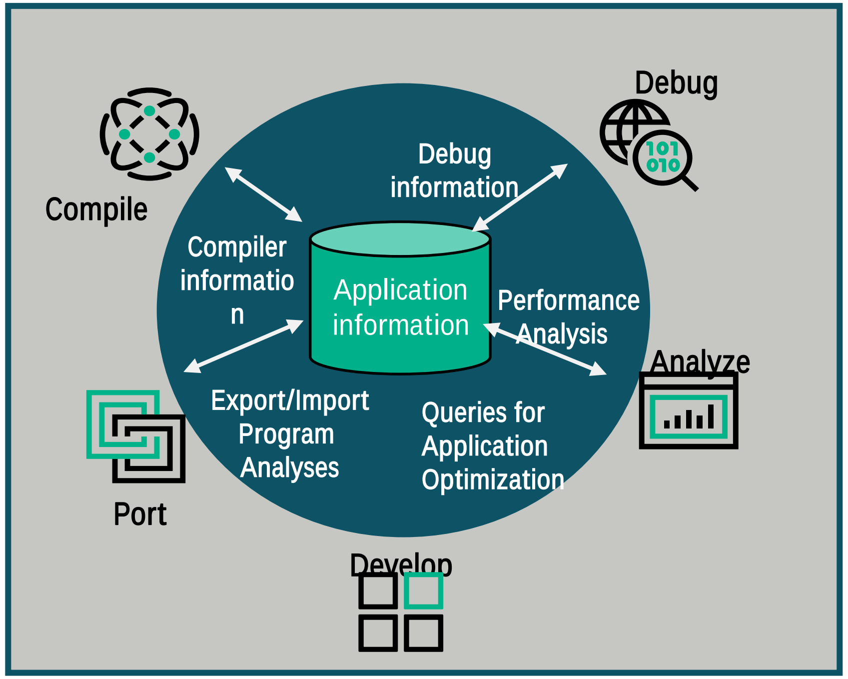
April 23–26, 2024

Agenda

- Introduction
  - General Remarks on Performance Analysis
  - Possible Bottlenecks and Remedies
- Perftools
  - Landscape and Properties
  - Sampling vs. Tracing
  - Performance analysis with `perftools-lite`
  - Apprentice2
- Demo

Motivation

- Very tempting to skip performance analysis when tests validate and time to solution is smaller after port or algorithm improvement.
- But performance does matter!
  - Might still be inefficient most of the of time.
  - Poor code performance can affect other users as well, for example because of bad I / O access patterns.
  - Want to efficiently use expensive resources and get as much information as possible for the allocated resources.
  - Simulating larger models may only be feasible after optimization.



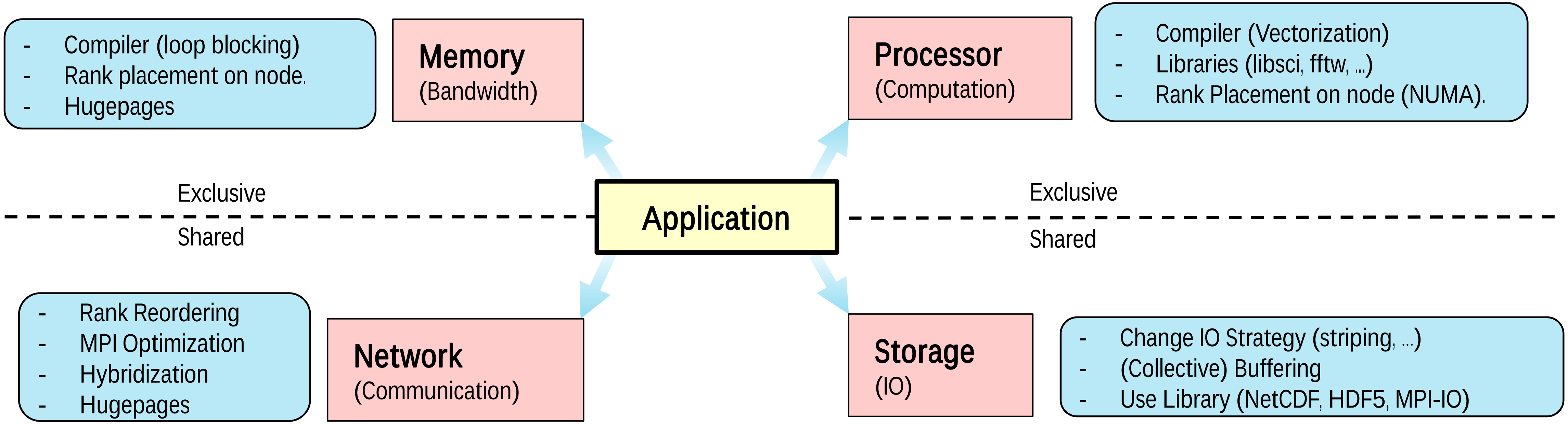
- Applies to various scenarios
  - Code has been ported to a new system or otherwise significantly changed.
  - Application is running in production since a while.
  - Makes extensive use of third-party libraries (distributed ML, dense LA, …) or even fully proprietary or it is mostly home-grown code.

General Remarks on Performance Analysis

- Performance is usually associated to FLOPs/sec.
  - But what if your code does many scattered memory references like in Graph Analytics and not many FLOPs ?
  - Chose the metric which suits your application (like time to solution or updates/sec instead of FLOPs/sec) and keep that metric throughout the optimization process.
- Use examples with different sizes for your experiments.
  - Small, medium, large, where node count differs by an order of magnitude (strong or weak scaling or both).
    - Try the same model with different grid sizes or number of particles
  - Do not use fully artificial examples but rather meaningful representatives of your target (large scale) simulation.

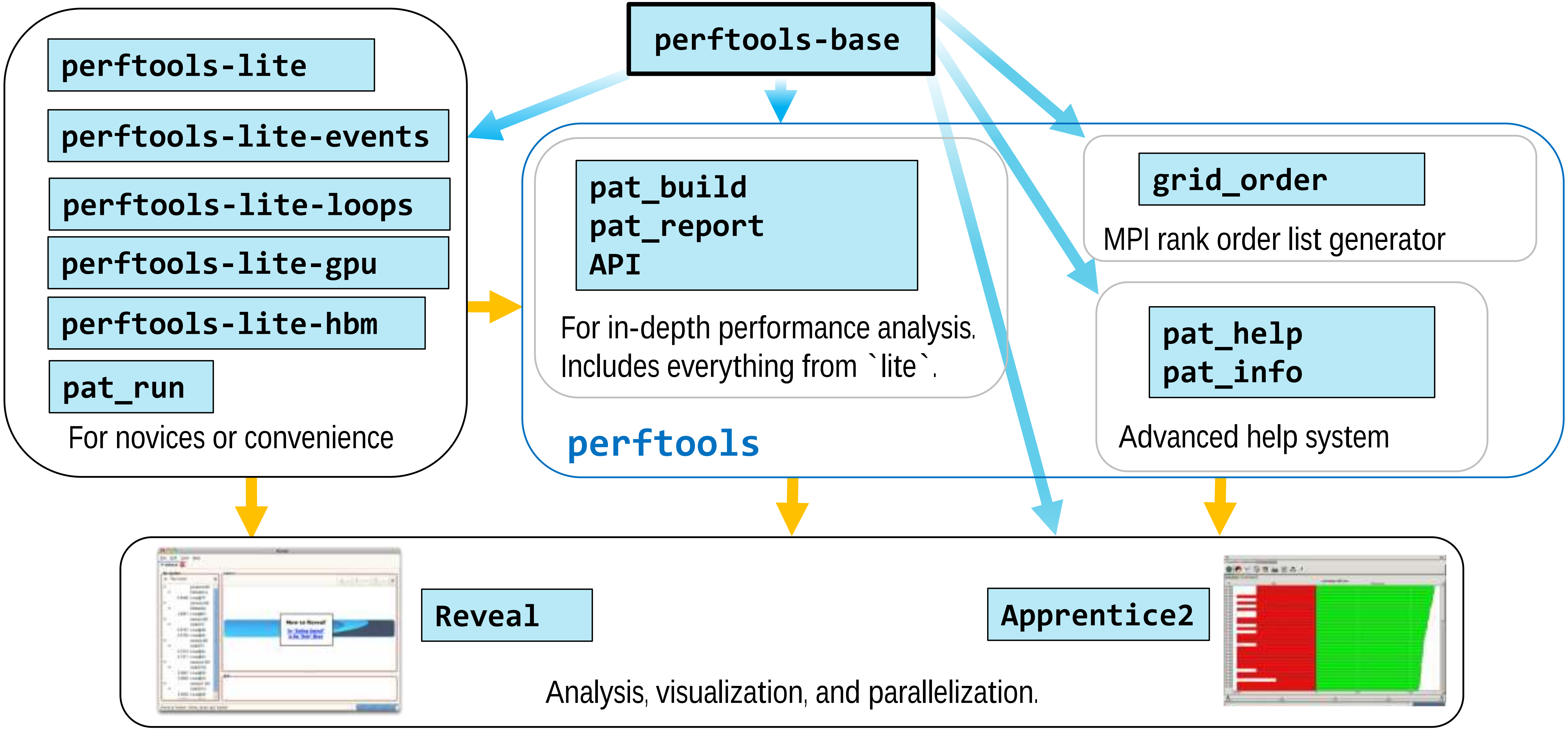
1. Start with low hanging fruits, i.e. avoid code modifications first.
   - Compiler flags, manual rank reordering, optimized libraries, huge pages, use hyperthreads, …
2. Use performance analysis tools
   - Identify critical code regions, guided rank reordering, Automatic parallelization (OpenMP), …
   - A good understanding of the workflow of your application (Communication, Computation, IO, …) helps to better interpret the profiles.

Bottlenecks and Remedies



- Good: One bottleneck which can be easily resolved without creating a new one.
- Bad: Several bottlenecks interacting with each other and changing over time.
- Need a profiler to identify bottleneck(s) and a model to estimate optimization potential.

Perftools Landscape



Two fundamental ways of profiling

1. Sampling
   - By taking regular snapshots of the applications call stack we can create a statistical profile of where the application spends most time.
   - Snapshots can be taken at regular intervals in time or when some other external event occurs, like a hardware counter overflowing

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

2. Event Tracing
   - Alternatively, we can record performance information every time a specific program event occurs, e.g. entering or exiting a function.
   - We can get accurate information about specific areas of the code every time the event occurs
   - Event tracing code can be added automatically or included manually through API calls.

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

Other Experiments

- Guided Tracing: Combining Sampling and Tracing
  - Trace only functions that are not small (i.e. very few lines of code) and contribute a lot to application's run time.
  - Automatic performance Analysis (APA) is an automated way to do this. (Not intended for `perftools-lite`)
    1. In a first round identify important routines with a sampling experiment.
    2. Then do a tracing of only these important routines to reduce overhead.
- Loop Work Estimates
  - Special flavor of event tracing targeting loops.
  - Only for Cray programming environment.
  - Useful starting point for porting to multicore or GPUs.
  - Can be used as input to Reveal.
  - Also available with `perftools-lite-loops`

Perftools-lite

An easy-to-use version of the Perftools Performance Measurement and Analysis Tool

Test 1: Generate a Sampling Profile

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

- Subsequent compiler invocations (`cc,CC,ftn`) will automatically insert necessary hooks for profiling (not always up-to-date with latest third-party compilers)

```
$> make clean; make
```

- Resulting executable `app.exe` is automatically instrumented. Object files needed!
- Not instrumented application stored as `app.exe+orig`

```
$> srun –n 8 app.exe >& job.out
```

- The report is printed to stdout.
- Successful execution creates a `app.exe+*/` directory for further analysis. Change this directory name with the `PAT_RT_EXPDIR_NAME` environment variable.

Output: Sampling

Regular program output

```
CrayPat/X:  Version 22.06.0 Revision 4b5ab6256  05/21/22 02:03:49
Sequential version array size
 mimax = 513 mjmax = 513 mkmax = 1025
Parallel version array size
 mimax = 259 mjmax = 259 mkmax = 515
imax = 257 jmax = 257 kmax =513
I-decomp = 2 J-decomp = 2 K-decomp =2
 Start rehearsal measurement process.
 Measure the performance in 100 times.

 MFLOPS: 15638.324504 time(s): 5.779363 2.794488e-04

 Now, start the actual measurement process.
 The loop will be excuted in 50 times
 This will take about one minute.
 Wait for a while

cpu : 28.858915 sec.
Loop executed for 50 times
Gosa : 2.763023e-04
MFLOPS measured : 15658.861128
Score based on Pentium III 600MHz : 189.025364
```

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

CrayPat/X:  Version 22.06.0 Revision 4b5ab6256  05/21/22 02:03:49
Experiment:                  lite  lite/sample_profile
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:  Sat Aug 20 23:43:46 2022
System name and speed:  nid005017  2.626 GHz (nominal)
AMD   Trento             CPU  Family: 25  Model: 48  Stepping:  1
Core Performance Boost:  All 8 PEs have CPB capability

Avg Process Time:     36.92 secs
High Memory:       15,544.6 MiBytes     1,943.1 MiBytes per PE
I/O Write Rate:   10.359011 MiBytes/sec
```

Regular program output

Sampling summary

```
Notes for table 1:

  This table shows functions that have significant exclusive sample
   hits, averaged across ranks.
  For further explanation, use:  pat_report -v -O samp_profile ...

Table 1:  Profile by Function

  Samp% |    Samp | Imb.  |  Imb. | Group
        |         | Samp  | Samp% |  Function=[MAX10]
        |         |       |       |   PE=HIDE

 100.0% | 3,693.2 |    -- |    -- | Total
|-----------------------------------------------------
|  84.1% | 3,107.8 |    -- |    -- | USER
||----------------------------------------------------
||  79.6% | 2,938.8 | 65.2 |  2.5% | jacobi
||   4.6% |   169.0 |  2.0 |  1.3% | initmt
||====================================================
|  14.7% |   541.5 |    -- |    -- | ETC
||----------------------------------------------------
||  13.3% |   490.2 | 31.8 |  7.0% | __cray_memcpy_ROME
||   1.4% |    50.9 |  4.1 |  8.6% | __cray_memset_ROME
||====================================================
|   1.1% |    41.8 |    -- |    -- | MPI
|=====================================================
...
```

Further analysis

```
Notes for table 3:

  This table shows energy and power usage for the nodes with the
    maximum, mean, and minimum usage, as well as the sum of usage over
    all nodes.
    Energy and power for accelerators is also shown, if applicable.
  For further explanation, use:  pat_report -v -O program_energy ...

Table 3:  Program energy and power usage (from Cray PM)

   Node |    Node |   Process | PE=HIDE
 Energy |   Power |      Time |
    (J) |     (W) |           |
----------------------------------------
 25,504 | 687.169 | 37.114590 | Total
========================================

Notes for table 4:

  This table show the average time and number of bytes written to each
    output file, taking the average over the number of ranks that
    wrote to the file.  It also shows the number of write operations,
    and average rates.
  For further explanation, use:  pat_report -v -O write_stats ...

Table 4:  File Output Stats by Filename

      Avg |      Avg |  Write Rate | Number |    Avg | Bytes/ | File
Name=!x/^/(proc|sys)/
    Write |    Write | MiBytes/sec |     of | Writes |   Call |  PE=HIDE
 Time per |  MiBytes |             | Writer |    per |        |
   Writer |      per |             |  Ranks | Writer |        |
     Rank |   Writer |             |        |   Rank |        |
          |     Rank |             |        |        |        |
|------------------------------------------------------------------------
| 0.000052 | 0.000582 |   11.224027 |      1 |   18.0 |  33.89 | stdout
|========================================================================

...
===============  End of CrayPat-lite output  ========================
```

Test 2: Generate an Event Profile

```
$> module sw perftools-lite perftools-lite-events
```

- If `perftools-lite` module not loaded, load subsequently `perftools-base` and `perftools-lite-events`.

```
$> rm app.exe; make
```

- Only relink of `app.exe` necessary if object files and user libraries have been generated with another `perftools-lite*` module.
- Otherwise do a `make clean; make`

```
$> srun –n 8 app.exe >& job.out
```

- The report is printed to stdout.
- Successful execution creates a `app.exe+*/` directory for further analysis.

Output: Event Tracing

Regular program output

```
CrayPat/X:  Version 22.06.0 Revision 4b5ab6256
05/21/22 02:03:49
 Sequential version array size
  mimax= 1025  mjmax= 513  mkmax= 513
 Parallel version  array size
  mimax= 515  mjmax= 259  mkmax= 259
  imax= 513  jmax= 257  kmax= 257
  I-decomp=  2  J-decomp=  2  K-decomp=  2

  Start rehearsal measurement process.
  Measure the performance in 3 times.
   MFLOPS: 15604.522995096944    time(s):
5.7918816709999987,  4.324619949E-4
 Now, start the actual measurement process.
 The loop will be excuted in 50  times.
 This will take about one minute.
 Wait for a while.
  Loop executed for  50  times
  Gosa : 4.21404955E-4
  MFLOPS: 15611.16692542893   time(s): 28.947083569
  Score based on Pentium III 600MHz : 188.449631
```

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

CrayPat/X:  Version 22.06.0 Revision 4b5ab6256  05/21/22
02:03:49
Experiment:                     lite  lite-events
Number of PEs (MPI ranks):         8
Numbers of PEs per Node:           8
Numbers of Threads per PE:         1
Number of Cores per Socket:       64
Execution start time:  Fri Nov 18 16:02:17 2022
System name and speed:  nid001050  2.078 GHz (nominal)
AMD   Milan              CPU  Family: 25  Model:  1
Stepping:  1
Core Performance Boost:  All 8 PEs have CPB capability


Avg Process Time:     37.05 secs
High Memory:       15,667.4 MiBytes     1,958.4 MiBytes per PE
I/O Write Rate:    3.182425 MiBytes/sec
```

General job information

Event tracing summary. Note difference to sampling

```
Notes for table 1:

  This table shows functions that have significant exclusive time,
    averaged across ranks.
  For further explanation, use:  pat_report -v -O profile ...

Table 1:  Profile by Function Group and Function

  Time% |      Time |    Imb. |  Imb. |    Calls | Group
        |           |    Time | Time% |          |  Function=[MAX10]
        |           |         |       |          |   PE=HIDE

 100.0% | 37.007250 |      -- |    -- | 1,461.4 | Total
|-----------------------------------------------------------------
|  98.4% | 36.410057 |      -- |    -- |     3.0 | USER
||----------------------------------------------------------------
||  76.9% | 28.457104 | 0.371909 |  1.5% |     1.0 | jacobi_
||  15.4% |  5.690022 | 0.078645 |  1.6% |     1.0 | himenobmtxp_
||   6.1% |  2.262931 | 0.004594 |  0.2% |     1.0 | initmt_
||================================================================
|   1.6% |  0.585681 |      -- |    -- |   983.0 | MPI
||----------------------------------------------------------------
||   1.3% |  0.482364 | 0.614413 | 64.0% |   180.0 | MPI_WAITALL
|=================================================================

Observation:  MPI utilization

    No suggestions were made because all ranks are on one node.
```

```
Notes for table 2:

  This table shows energy and power usage for the nodes with the
    maximum, mean, and minimum usage, as well as the sum of usage over
    all nodes.
    Energy and power for accelerators is also shown, if applicable.
  For further explanation, use:  pat_report -v -O program_energy ...

Table 2:  Program energy and power usage (from Cray PM)

   Node |    Node |   Process | PE=HIDE
 Energy |   Power |      Time |
    (J) |     (W) |           |
-----------------------------------------
  9,055 | 244.407 | 37.048880 | Total
=========================================
```

```
Notes for table 3:

  This table show the average time and number of bytes written to each
    output file, taking the average over the number of ranks that
    wrote to the file.  It also shows the number of write operations,
    and average rates.
  For further explanation, use:  pat_report -v -O write_stats ...

Table 3:  File Output Stats by Filename

      Avg |      Avg |  Write Rate | Number |     Avg | Bytes/ | File
Name=!x/^/(proc|sys)/
    Write |    Write | MiBytes/sec |     of |  Writes |   Call |  PE=HIDE
 Time per |  MiBytes |             | Writer |     per |        |
   Writer |      per |             |  Ranks |  Writer |        |
     Rank |   Writer |             |        |    Rank |        |
          |     Rank |             |        |         |        |
|-----------------------------------------------------------------------
| 0.000024 | 0.000008 |    0.321773 |      8 |     1.0 |   8.00 | stderr
| 0.000022 | 0.000613 |   27.644603 |      1 |    18.0 |  35.72 | stdout
|=======================================================================
```

```
Program invocation:  himeno.exe

For a complete report with expanded tables and notes, run:
  pat_report /pfs/lustrep2/projappl/project_465000297/alfiolaz/work/perftools-
lite/test/expfile.lite-events.2046565

For help identifying callers of particular functions:
  pat_report -O callers+src
/pfs/lustrep2/projappl/project_465000297/alfiolaz/work/perftools-
lite/test/expfile.lite-events.2046565
To see the entire call tree:
  pat_report -O calltree+src
/pfs/lustrep2/projappl/project_465000297/alfiolaz/work/perftools-
lite/test/expfile.lite-events.2046565

For interactive, graphical performance analysis, run:
  app2 /pfs/lustrep2/projappl/project_465000297/alfiolaz/work/perftools-
lite/test/expfile.lite-events.2046565

================  End of CrayPat-lite output  ==========================
```

Further analysis

Test 3: Generate a Loop Profile (CCE only)

```
$> module sw perftools-lite perftools-lite-loops
```

- If perftools-lite module not loaded, load subsequently perftools-base and perftools-lite-loops. Only for `PrgEnv-cray`.

```
$> make clean; make
```

- Need to clean everything and rebuild.
- Compiler drivers will use `-h profile_generate` for Fortran or `-finstrument-loops` for C implicitly. This flag turns off OpenMP and significant compiler loop restructuring optimizations except for vectorization.

```
$> srun –n 8 app.exe >& job.out
```

- Successful execution creates a `app.exe+*/` directory for further analysis.
- The report is printed to stdout.

Output: Loop Profile

Regular program output

```
CrayPat/X:  Version 22.06.0 Revision 4b5ab6256  05/21/22 02:03:49
Sequential version array size
 mimax = 513 mjmax = 513 mkmax = 1025
Parallel version array size
 mimax = 259 mjmax = 259 mkmax = 515
imax = 257 jmax = 257 kmax =513
I-decomp = 2 J-decomp = 2 K-decomp =2
 Start rehearsal measurement process.
 Measure the performance in 100 times.

 MFLOPS: 15652.801400 time(s): 5.774017 2.794488e-04

 Now, start the actual measurement process.
 The loop will be excuted in 50 times
 This will take about one minute.
 Wait for a while

cpu : 28.869171 sec.
Loop executed for 50 times
Gosa : 2.763023e-04
MFLOPS measured : 15653.298226
Score based on Pentium III 600MHz : 188.958211
```

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

CrayPat/X:  Version 22.06.0 Revision 4b5ab6256  05/21/22 02:03:49
Experiment:                  lite  lite-loops
Number of PEs (MPI ranks):      8
Numbers of PEs per Node:        8
Numbers of Threads per PE:      1
Number of Cores per Socket:    64
Accelerator Model: AMD MI200 Memory: 32.00 GB Frequency: 1.09 GHz


Execution start time:  Sun Aug 21 00:29:19 2022
System name and speed:  nid005013  2.361 GHz (nominal)
AMD   Trento               CPU  Family: 25  Model: 48  Stepping:  1
Core Performance Boost:  All 8 PEs have CPB capability



Avg Process Time:     37.12 secs
High Memory:      15,593.3 MiBytes 1,949.2 MiBytes per PE
```

General job information

Loop Statistics by function

```
Notes for table 1:

  This table shows a nested view of loops that have significant inclusive time, averaged across ranks. Intervening function calls
    are not shown (as if all functions were inlined).For each loop, the table shows its inclusive time, the number of
    times it was executed, and the average number of iterations for each execution. Times in this table include overhead from loop
    instrumentation. For an alternative view that shows min and max iterations, and exclusive times, use the option:  -O loop_times
  For further explanation, use:  pat_report -v -O loop_nest ...

Table 1:  Nested view of Loop Inclusive Time

   Incl  |  Incl  | Loop Exec |  Loop  | Calltree=/[.]LOOP[.]
  Time%  |  Time  |           | Trips  |  PE=HIDE
         |        |           |  Avg   |

 100.0% | 37.08 |        -- |     -- | Total
|------------------------------------------------------
1  93.4% | 34.64 |         2 |  30.0 | jacobi.LOOP.1.li.236
||-----------------------------------------------------
2|  79.0% | 29.31 |        60 | 255.0 | jacobi.LOOP.2.li.240
|||----------------------------------------------------
3||  79.0% | 29.31 |    15,300 | 255.0 | jacobi.LOOP.3.li.241
||||---------------------------------------------------
4|||  78.2% | 29.00 | 3,901,500 | 511.0 | jacobi.LOOP.4.li.242
|||===================================================
2|  13.5% |  5.02 |        60 | 255.0 | jacobi.LOOP.5.li.263
|||----------------------------------------------------
3||  13.5% |  5.02 |    15,300 | 255.0 | jacobi.LOOP.6.li.264
||||---------------------------------------------------
4|||   9.6% |  3.54 | 3,901,500 | 511.0 | jacobi.LOOP.7.li.265
||=====================================================
1  4.3% |  1.61 |         1 | 259.0 | initmt.LOOP.4.li.191
||-----------------------------------------------------
2|  4.3% |  1.61 |       259 | 259.0 | initmt.LOOP.5.li.192
|||----------------------------------------------------
3||  4.3% |  1.59 |    67,081 | 515.0 | initmt.LOOP.6.li.193
||=====================================================
1  2.2% |  0.82 |         1 | 257.0 | initmt.LOOP.1.li.210
||-----------------------------------------------------
2|  2.2% |  0.82 |       257 | 257.0 | initmt.LOOP.2.li.211
|||----------------------------------------------------
3||  2.1% |  0.79 |    66,049 | 513.0 | initmt.LOOP.3.li.212
|======================================================
Program invocation:  himeno.exe

For a complete report with expanded tables and notes, run:
  pat_report /pfs/lustrep3/users/lazzaroa/exercises/perftools-lite/C/himeno.exe+44287-8745533t

For help identifying callers of particular functions:
  pat_report -O callers+src /pfs/lustrep3/users/lazzaroa/exercises/perftools-lite/C/himeno.exe+44287-8745533t
To see the entire call tree:
  pat_report -O calltree+src /pfs/lustrep3/users/lazzaroa/exercises/perftools-lite/C/himeno.exe+44287-8745533t

For interactive, graphical performance analysis, run:
  app2 /pfs/lustrep3/users/lazzaroa/exercises/perftools-lite/C/himeno.exe+44287-8745533t

================  End of CrayPat-lite output  ==========================
```

Subroutine

Line number

Nested Loops

Generate an Event Profile for GPU Experiments

```
$> module load perftools-lite-gpu
```

- If `perftools-lite` module not loaded, load subsequently `perftools-base` and `perftools-lite-gpu`.

```
$> rm app.exe; make
```

- Only relink of `app.exe` necessary if object files and user libraries have been generated with another `perftools-lite*` module.
- Otherwise do a `make clean; make`

```
$> srun -n 8 app.exe >& job.out
```

- The report is printed to stdout.
- Successful execution creates a `app.exe+*/` directory for further analysis.

Memory Transfer Between Host and Device

- From the Himeno benchmark (OpenMP offload)

```
Notes for table 3:

  This table shows functions that have significant exclusive host or
    accelerator time, averaged across ranks, and also data copied in
    and out, and event counts.
  For further explanation, use:  pat_report -v -O acc_fu ...

Table 3:  Time and Bytes Transferred for Accelerator Regions

  Time% |      Time |    Acc |   Acc |  Acc Copy |  Acc Copy | Events | Function=[max10]
        |           |  Time% |  Time |        In |       Out |        |  PE=HIDE
        |           |        |       | (MiBytes) | (MiBytes) |        |   Thread=HIDE

 100.0% | 45.464262 | 100.0% | 73.31 |     7,380 |      0.00 |  1,980 | Total
|-----------------------------------------------------------------------------
|  87.9% | 39.940594 |     -- |    -- |        -- |        -- |    120 | hipStreamSynchronize
|   5.4% |  2.448416 |     -- |    -- |        -- |        -- |      0 | initmt
|   3.1% |  1.425578 |     -- |    -- |        -- |        -- |      0 | MPI_Waitall
|   2.6% |  1.160629 |   0.0% |  0.03 |     3,690 |        -- |     74 | hipMemcpyHtoD
|   0.0% |  0.002311 |   1.6% |  1.16 |     3,690 |        -- |      4 | jacobi.ACC_COPY@li.236
|   0.0% |  0.000496 |  42.6% | 31.20 |        -- |        -- |     60 | hipKernel.__omp_offloading_54bbb604_f500d049_jacobi_l242
|   0.0% |  0.000355 |   1.3% |  0.98 |        -- |        -- |     60 | hipKernel.__omp_offloading_54bbb604_f500d049_jacobi_l268
|   0.0% |  0.000058 |  43.5% | 31.90 |        -- |        -- |     60 | jacobi.ACC_KERNEL@li.242
|   0.0% |  0.000042 |  11.0% |  8.04 |        -- |        -- |     60 | jacobi.ACC_KERNEL@li.268
|=============================================================================
```

ROCM Profiler with MPI

```
$> module load rocm
```

- No need for an accelerator module for the target GPU

```
WORK_DIR=…

if [[ "$SLURM_PROCID" == 0 ]]; then

        rocprof -o ${WORK_DIR}/run/profile.${SLURM_JOBID} --trace-start \
              off ${ICON_DIR}/bin/executable

else
        ${WORK_DIR}/bin/executable
fi
```

- Put the code above in a script and launch the script with `srun`
- Single ranks can be profiled separately

Observations and Remarks

- No intervention needed for build system and batch scripts.
  - Only make sure to use the compiler driver wrappers `CC`, `cc`, and `ftn`.
- What we did not see with these simple tests:
  - `perftools-lite` can produce rank reordering files for MPI to optimize the communication. Not visible here because of small portion of time spent in communication or job size.
  - The resulting `app+exe*/` directory can be processed with `pat_report`, Apprentice2, and Reveal for further analysis. From the sample experiment one can retrieve hardware performance counter information.
- Tailored profiling, i.e. for specific routines, trace groups, or specific portions of the code is not possible.
  - Need the regular `perftools` module for this in-depth analysis.
- `CRAYPAT_LITE` environment variable can be used to distinct output files.
- Record Subset of PEs during execution: `export PAT_RT_EXPFILE_PES=0,4,5,10`
- Use `CRAYPAT_LITE_WHITELIST` for binaries you DO want instrumented (rest ignored).

Further analysis (without re-running)

- Generate full report

```
$> pat_report app.exe+pat*/ > rpt
```

- Generate report with call tree (or by callers)

```
$> pat_report -O calltree+src
$> pat_report -O callers+src
```

```
 100.0% | 9.300473 | 1,056.2 | Total
|-----------------------------------------------------
|  99.2% | 9.228240 |     5.0 | USER
||----------------------------------------------------
||  97.0% | 9.020521 |     2.0 | jacobi
|||---------------------------------------------------
3||   80.8% | 7.517532 |     1.0 | main:himeno.c:line.150
3||   16.2% | 1.502989 |     1.0 | main:himeno.c:line.119
|||===================================================
||    2.2% | 0.207618 |     1.0 | initmt
3|         |          |         |  main:himeno.c:line.102
|====================================================
```

- Show each MPI rank or each OpenMP thread in report

```
$> pat_report -s pe=ALL
$> pat_report -s th=ALL
```

- Generate a preview of data before processing the full report

```
$> pat_report -Q1
```

  - Produces report from single (lexically first) '.ap2' file
  - Useful for jobs with large number of processes

Further analysis (without re-running)

- Generate report from specific subset of ranks

```
$> pat_report –s filter_input='pe==0'
```

  - Report with only PE 0 data

```
$> pat_report –s filter_intput='pe<5'
```

  - Report with data from first 5 ranks
  - Use `pat_help report filtering` for more details

- Don't see an expected function?
  - Use the `pat_report –P` option to disable pruning.
  - You should be able to see the caller/callee relationship with `pat_report -P -O callers`
  - Use '`pat_report –T`' to see functions that didn't take much time
  - Still don't see it? Check the compiler listing to see if the function was inlined.

- Also try the GUI for analyzing performance analysis results.

```
$> app2 app.exe+*/
```

On-node analysis

- Memory bandwidth sensitivity guidance

Functions Slowed By Memory Bandwidth Utilization

The performance data for the functions shown below suggest that their performance is limited by memory bandwidth. To confirm this, try running with fewer processes placed on each node.

```
  Samp% |  Memory |   Stall | Function
        | Traffic | PerCent |  Numanode=HIDE
        |       / |         |   PE=HIDE
        | Nominal |         |
        |    Peak |         |
|------------------------------------------
| 40.9% |   54.1% |   93.8% | daxpy_kernel_8
| 36.1% |   59.4% |   93.8% | dgemv_kernel_4x4
|==========================================
```

On-node analysis

- Example traffic from an MPI+OpenMP run.

Available in default report assuming processors supports collecting the data

Table 3: Memory Bandwidth by Numanode (limited entries shown)

Memory Traffic GBytes	Local Memory Traffic GBytes	Remote Memory Traffic GBytes	Thread Time	Memory Traffic GBytes / Sec	Memory Traffic / Nominal Peak	Numanode Node Id=[max3,min3] PE=HIDE Thread=HIDE
184.47	173.59	10.89	11.578777	15.93	20.7%	numanode.0
183.50	173.59	9.91	11.569322	15.86	20.7%	nid.63
182.61	172.40	10.21	11.578777	15.77	20.5%	nid.61
178.55	167.75	10.80	11.563156	15.44	20.1%	nid.71
178.10	168.14	9.96	11.562097	15.40	20.1%	nid.62
178.08	168.07	10.01	11.564512	15.40	20.1%	nid.68
178.01	167.20	10.82	11.572032	15.38	20.0%	nid.70
60.36	14.73	45.62	9.073119	6.65	8.7%	numanode.1
60.36	14.73	45.62	9.072693	6.65	8.7%	nid.63
59.88	14.33	45.55	9.071553	6.60	8.6%	nid.62
59.48	14.19	45.29	9.068044	6.56	8.5%	nid.68
58.78	13.70	45.08	9.069259	6.48	8.4%	nid.70
58.67	13.87	44.81	9.071591	6.47	8.4%	nid.69
58.53	13.86	44.67	9.067146	6.46	8.4%	nid.71

Notice remote memory traffic by OpenMP threads

On-node analysis

- Low vectorization guidance.

Functions with Low Vectorization

The performance data for the functions shown below suggest that their performance could be improved by increased vectorization. Use compiler optimization messages to identify loops in those functions that were not vectorized and try to use directives or restructure the loops to enable them to vectorize.

```
 Samp% |    Vector |   Stall | Function
       |intensity |PerCent |   PE=HIDE
       |          |         |Thread=HIDE
|------------------------------------------------------------
|47.7% |       0.3%|   15.2% | depose_jxjyjz_esirkepov_1_1_1_
|============================================================
```

On-node analysis

- Memory latency sensitivity guidance.

Functions Slowed By Memory Latency

The performance data for the functions shown below suggest that their performance is limited by memory latency. It could be beneficial to modify prefetching in loops in those functions, by modifying compiler-generated prefetches or inserting directives into the source code.

```
  Samp% |  Memory |   Stall | Function
        | Traffic | PerCent |  Numanode=HIDE
        |       / |         |   PE=HIDE
        | Nominal |         |
        |    Peak |         |
|-----------------------------------------------------------
| 72.8% |   34.8% |   33.9% | dim3_sweep$dim3_sweep_module_
|===========================================================
```

Documentation

- Module help
  - module help perftools-base
  - module help perftools
  - module help perftools-lite
  - module help perftools-lite-*
- Man pages
  - man pat_build
  - man pat_report
- Advanced help system
  - pat_help
  - pat_info [[.ap2_file] [experiment_data_directory]…]

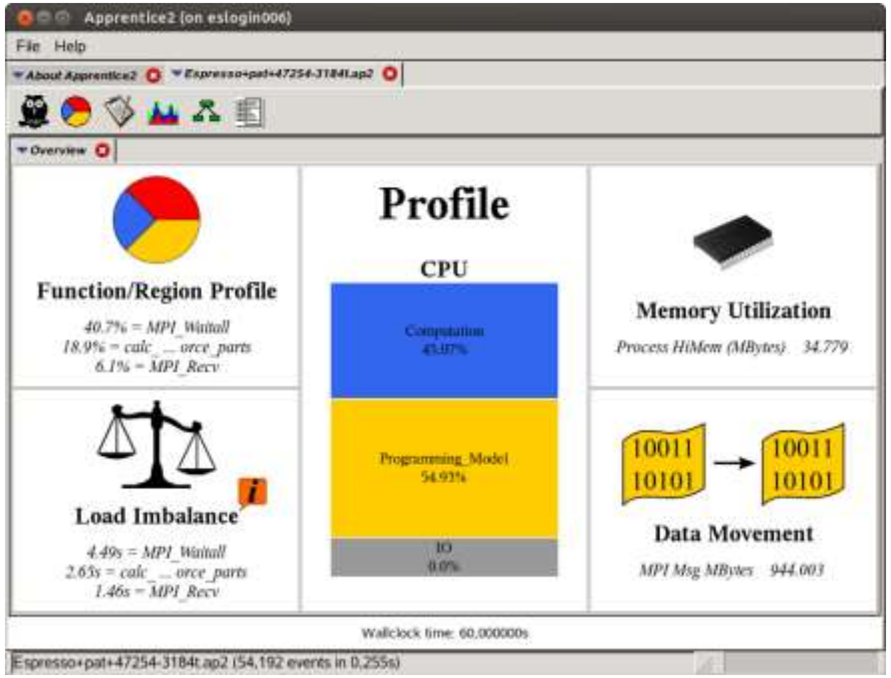
APPRENTICE2

Display your performance analysis results

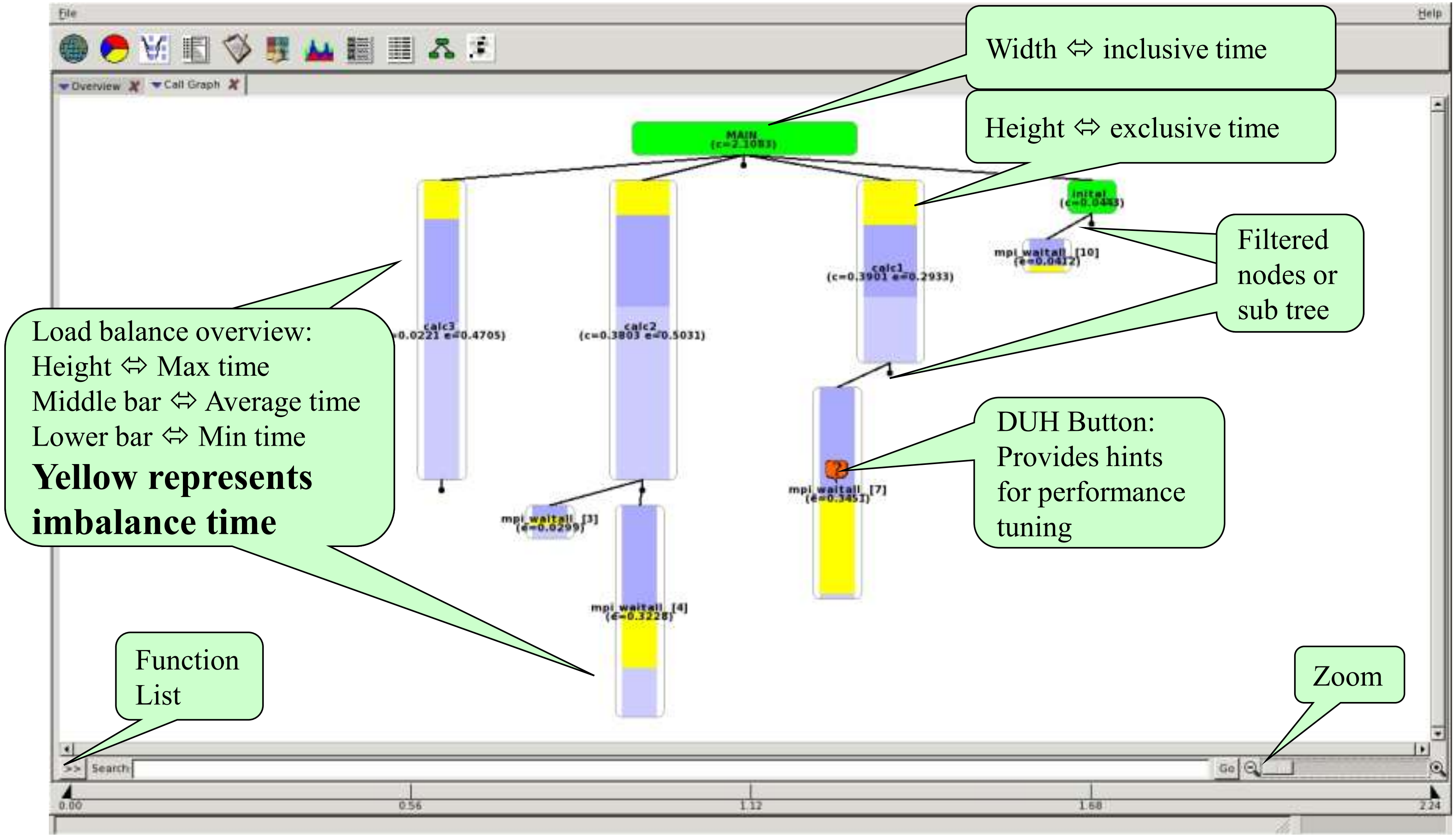
Apprentice2

```
$> app2 app.exe+*/ &
```

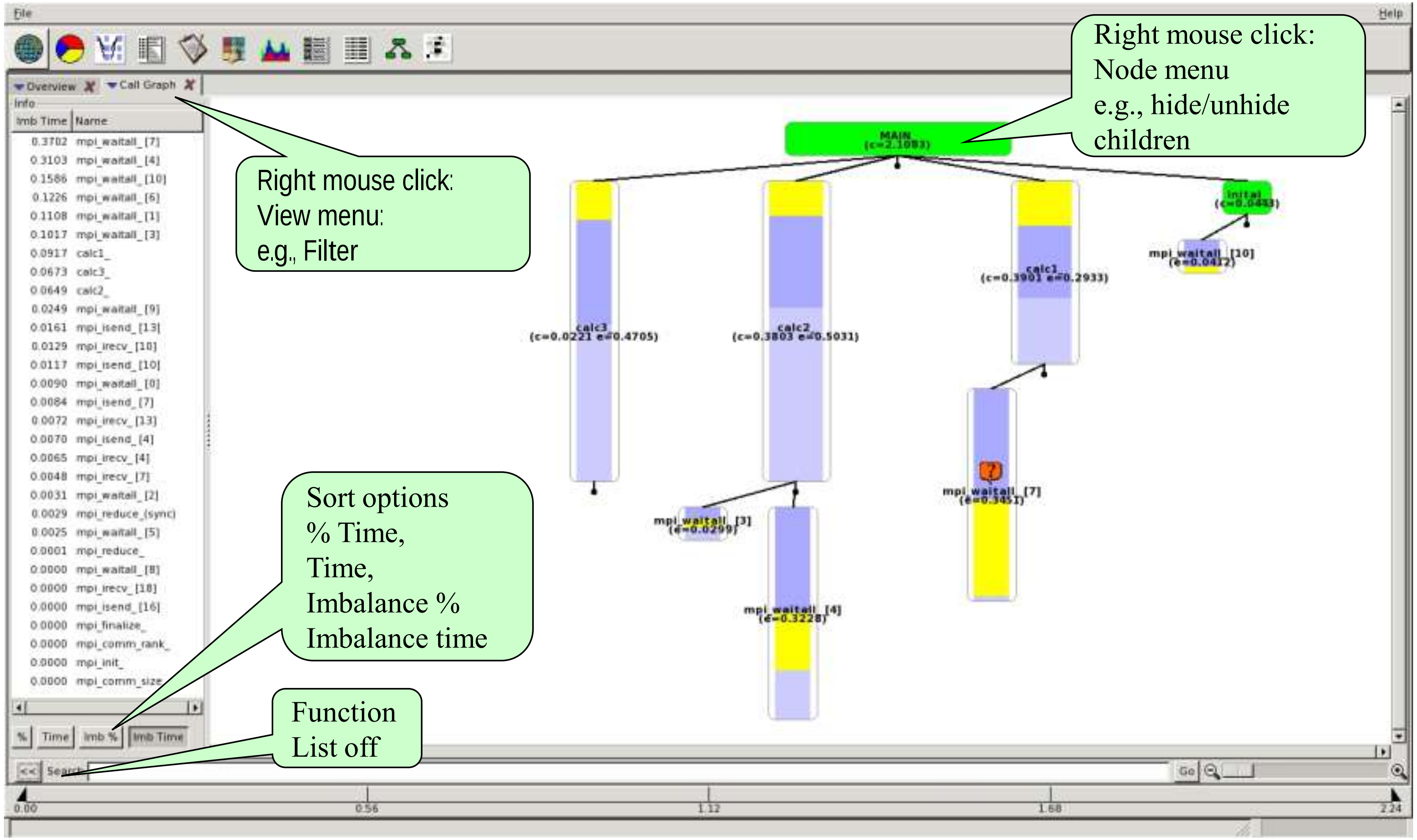
- Use an experiment directory as input. Can be from perftools-lite if pat_report has been used to generate *ap2 files.
- Desktop client installer: /opt/cray/pe/perftools/<version>/share/desktop_installers/



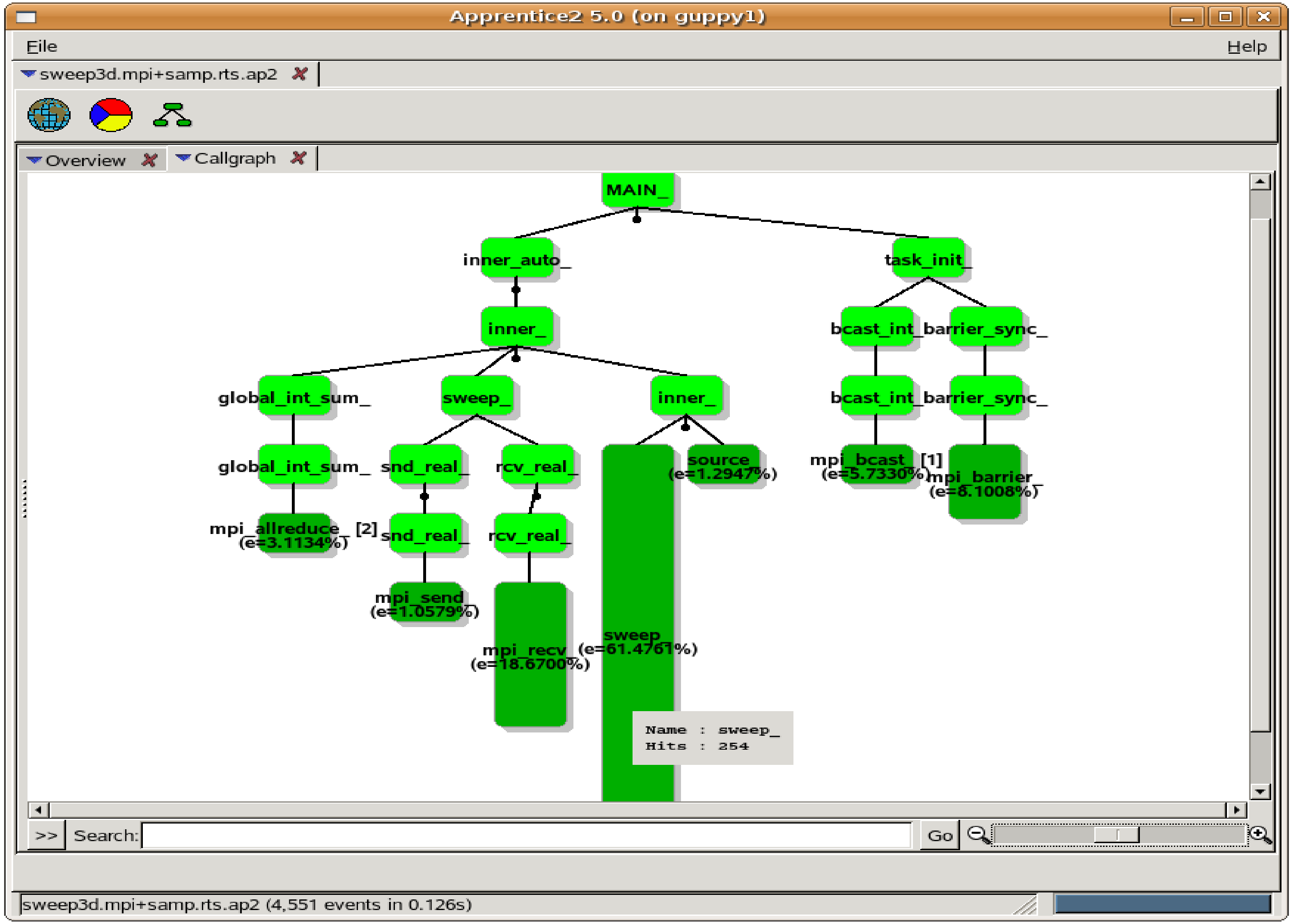
Call tree view



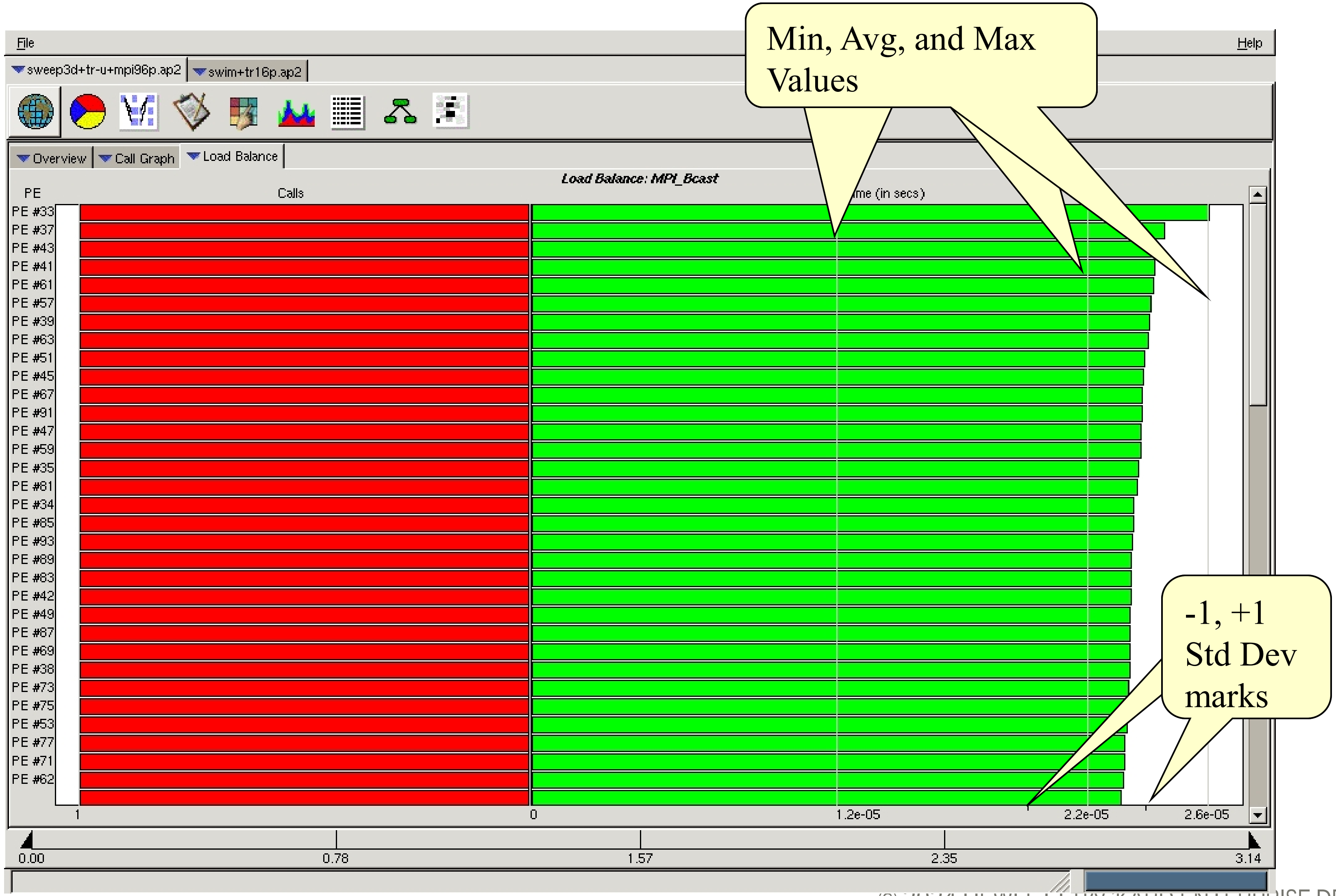
Call tree view - function list



Call tree view of sampled data

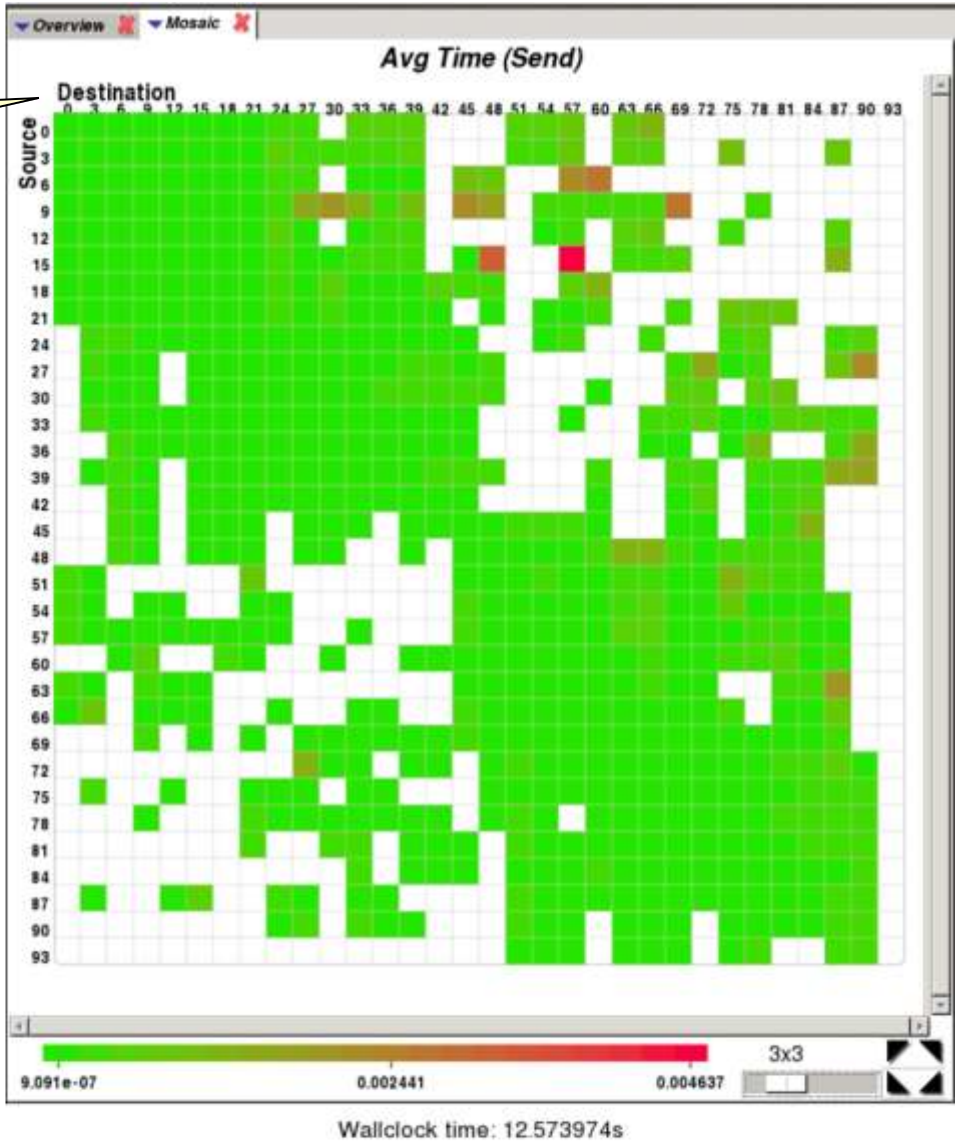


Call tree view of sampled data



Mosaic View

Send/receive of data, useful to check communication patterns



Questions?