Load Imbalance Analysis

With CrayPat

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. Time	Imb. Time%	Calls	Group 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".**

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

- 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. Time	Imb. Time%	Calls	Group / 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)
|===================================================================
```

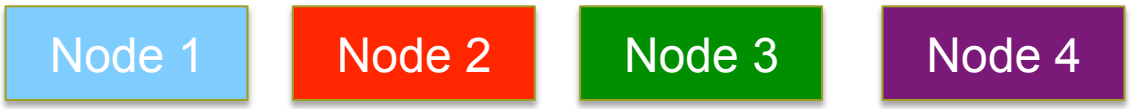
Causes and hints

- **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 even with just a few MPI tasks (more on I/O later).

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, 4] [1, 5] [2, 6] [3, 7] (8 tasks on 4 nodes, 2 tasks per node)
  - N=1: (DEFAULT) **SMP-style-** (block-) placement
    [0, 1] [2, 3] [4, 5] [6, 7] (8 tasks on 4 nodes, 2 tasks per node)
  - N=2: **Folded** rank placement
    [0, 7] [1, 6] [2, 5] [3, 4] (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 sub-communicators
  - 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
…
```

Rank placement : grid_order

- The **`grid_order`** utility is used to generate a rank order list for use by an MPI application that uses communication between nearest neighbors in a grid
- When executed with the desired arguments, grid_order generates rank order information in the appropriate format and writes it to stdout
- This output can then be copied or written into a file named **MPICH_RANK_ORDER** and used with **MPICH_RANK_REORDER_METHOD=3** environment variable to override the default MPI rank placement scheme and specify a custom rank placement

Example of use of rank reordering

- **WRF – weather forecasting application**
- **Use of rank reordering can improve performance by 5%**
  - Use grid_order utility
    - *module load perftools*
    - *$CRAYPAT_ROOT/bin/grid_order*
    - *grid_order -C -c 2,12 –g 48,128 > MPICH_RANK_ORDER*
    - For this WRF case I used '*grid_order –C –c 2,12 –g 48,128 > MPICH_RANK_ORDER*'
      - -c 2,12 is core topology on an XC node
      - -g 48,128 is WRF domain decomposition
  - *setenv MPICH_RANK_REORDER_METHOD 3*
    - Uses file called MPICH_RANK_ORDER written to working directory
- **Method improves halo exchange communications by putting more neighbors on same node**
- **Was more important in days when there were fewer cores per node**
  - Less chance of nearest neighbours ending up on same node
  - In days of 12 cores per node, rank reordering boosted perf by over 20%

WRF Rank Ordering

- Partial WRF grid for 6144 core case with a 48 by 128 decomposition on XC with 24 cores per node.

WRF grid	48x128			6144 cores													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208

Halo exchange neighbors

Default sequential rank ordering												
N1	0	1	2	3	4	5	6	7	8	9	10	11
N2	24	25	26	27	28	29	30	31	32	33	34	35
N3	48	49	50	51	52	53	54	55	56	57	58	59
N4	72	73	74	75	76	77	78	79	80	81	82	83
N4	96	97	98	99	100	101	102	103	104	105	106	107
	120	121	122	123	124	125	126	127	128	129	130	131

24 MPI ranks per node

MPI ranks placed sequentially on cores

Optimal rank ordering												
N1	0	1	48	49	96	97	144	145	192	193	240	241
N2	2	3	50	51	98	99	146	147	194	195	242	243
N3	4	5	52	53	100	101	148	149	196	197	244	245
N4	6	7	54	55	102	103	150	151	198	199	246	247

MPI ranks placed optimally on cores

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
  - 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

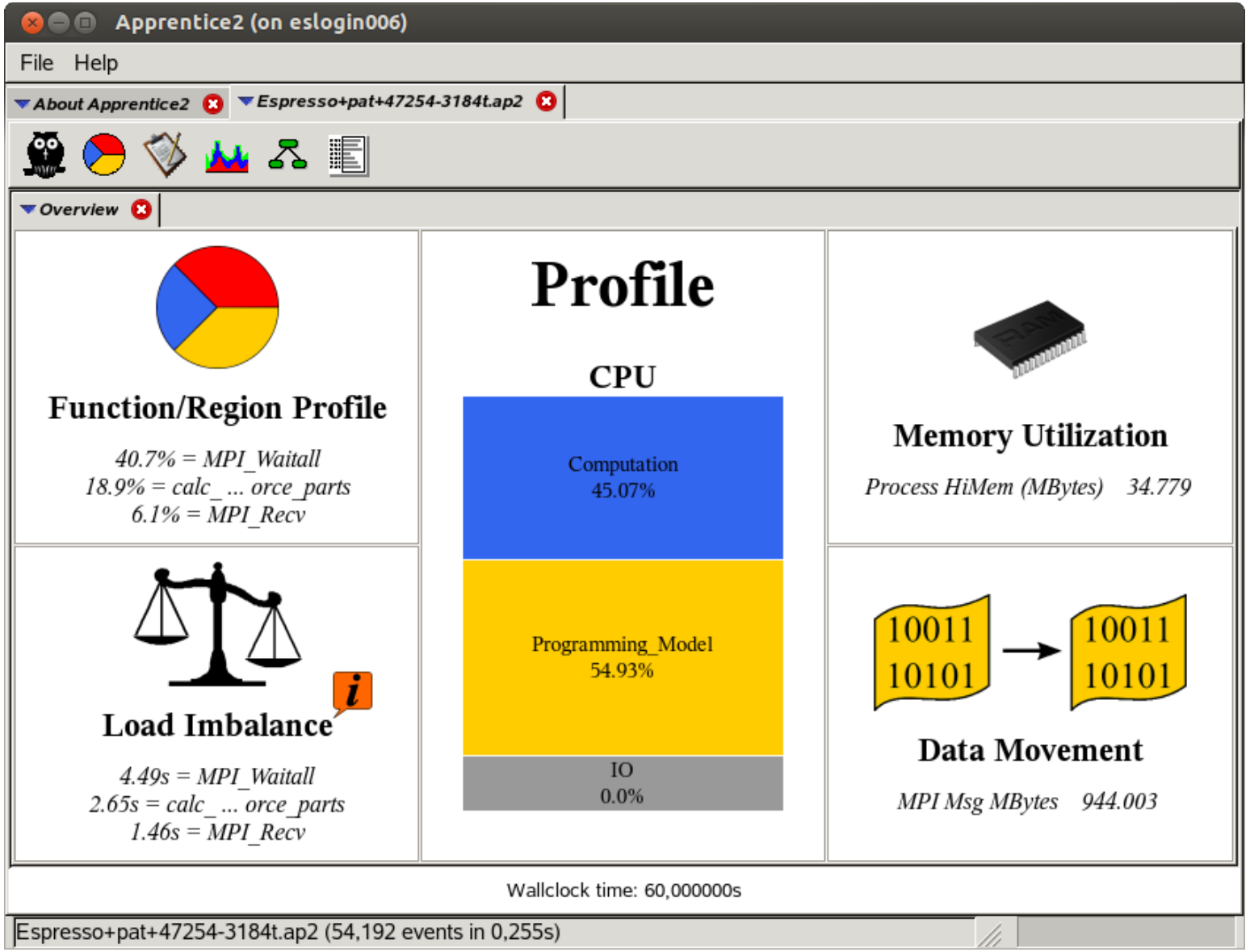
- **Cray Apprentice2 is a post-processing performance data visualization tool. Takes *.ap2 files as input.**

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

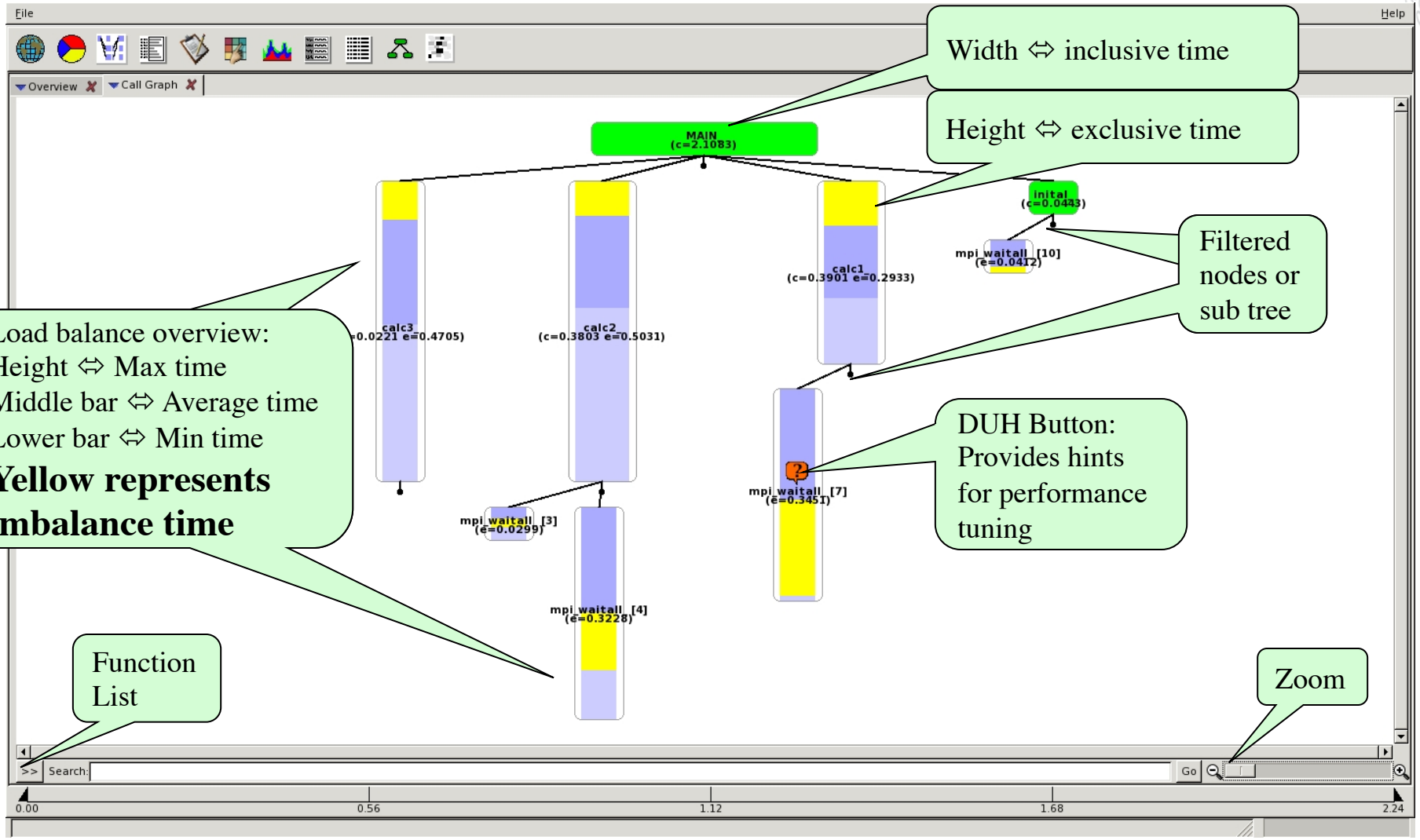
- **Cray Apprentice^2 helps identify:**
  - Load imbalance
  - Excessive communication
  - Network contention
  - Excessive serialization
  - I/O Problems

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

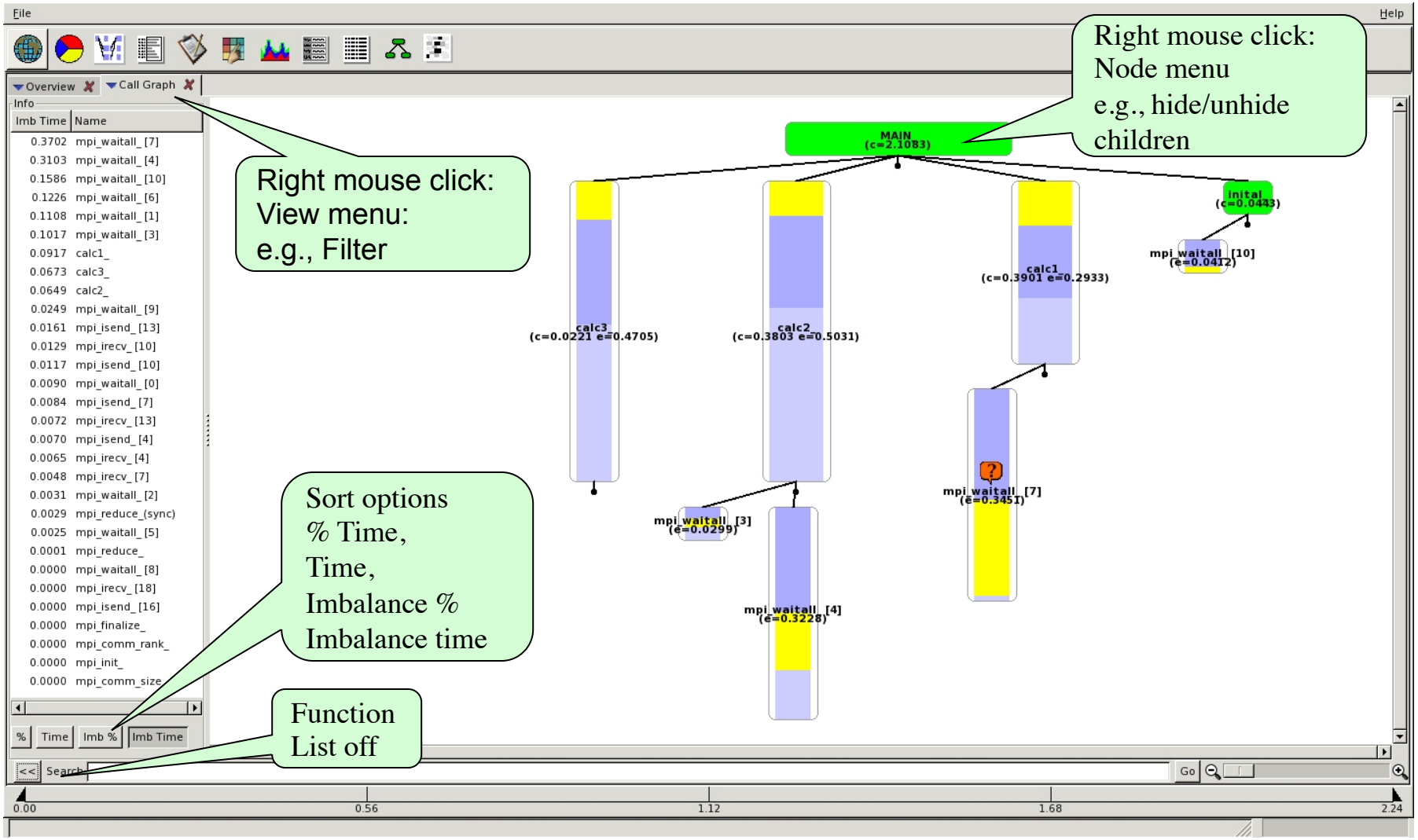
Cray Apprentice^2



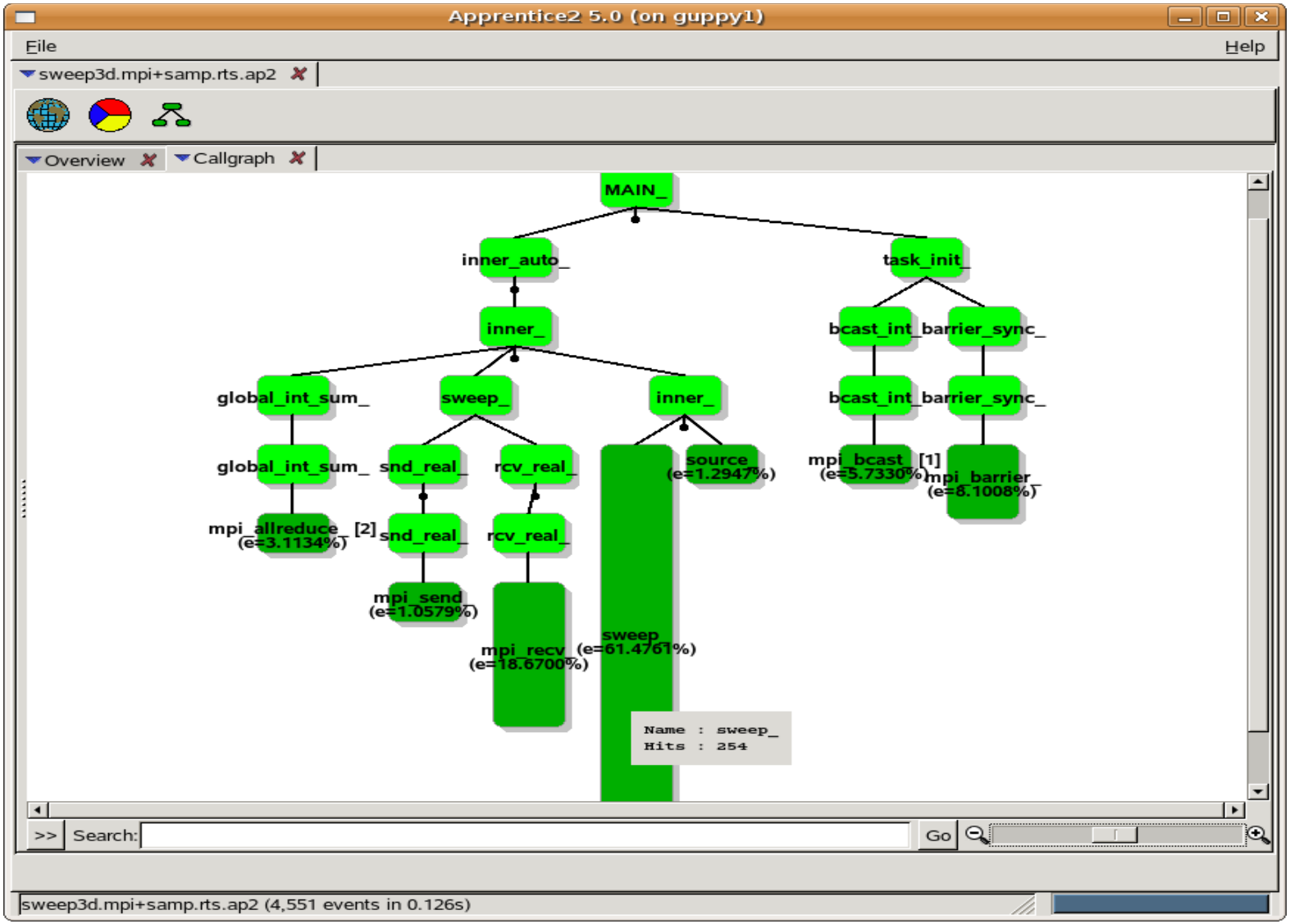
Call Tree View



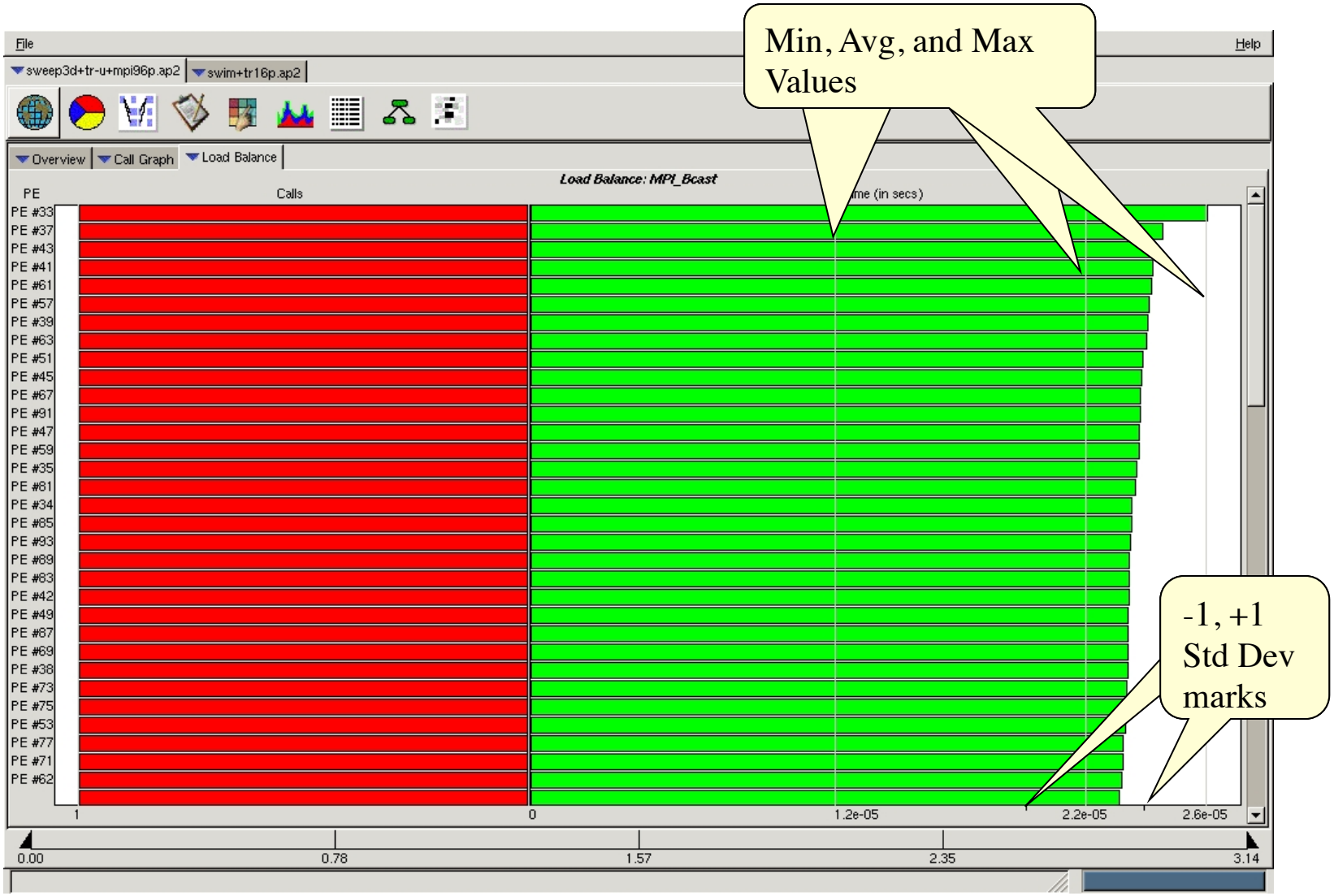
Call Tree View – Function List



Apprentice^2 Call Tree View of Sampled Data

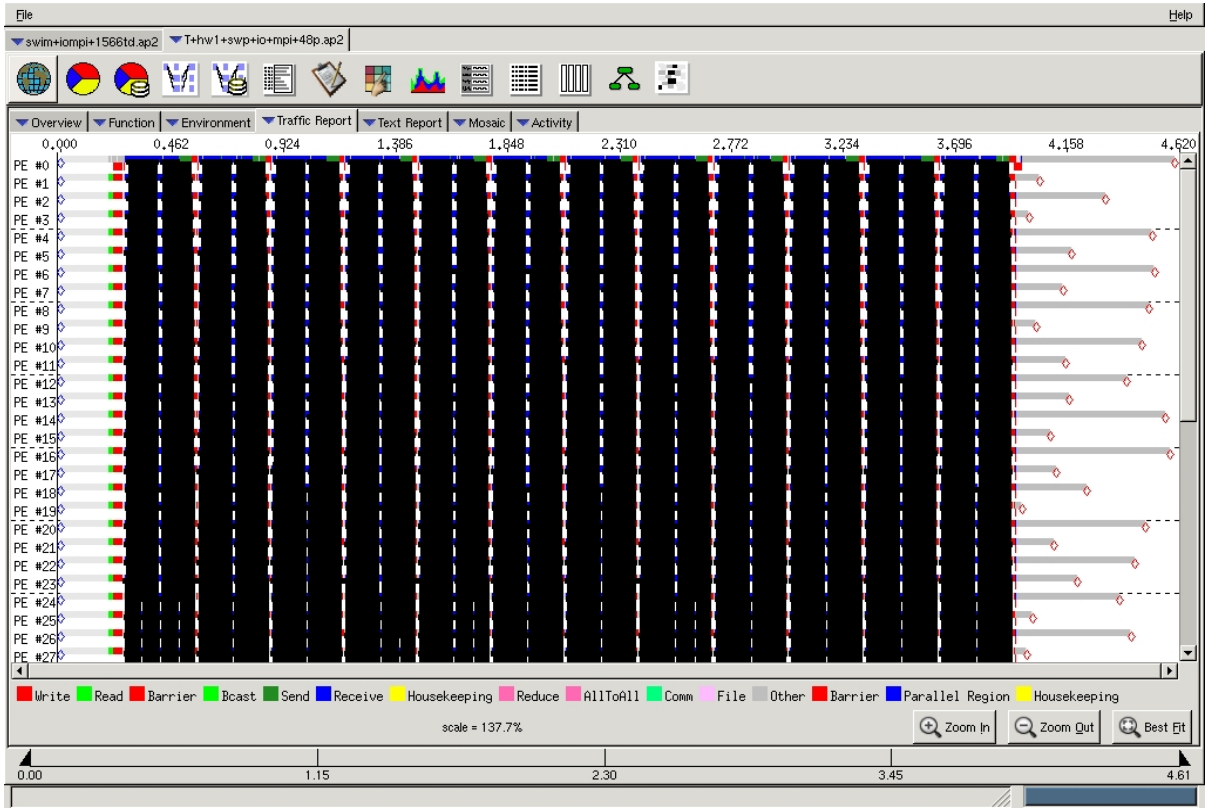


Load Balance View (from Call Tree)



Time Line View

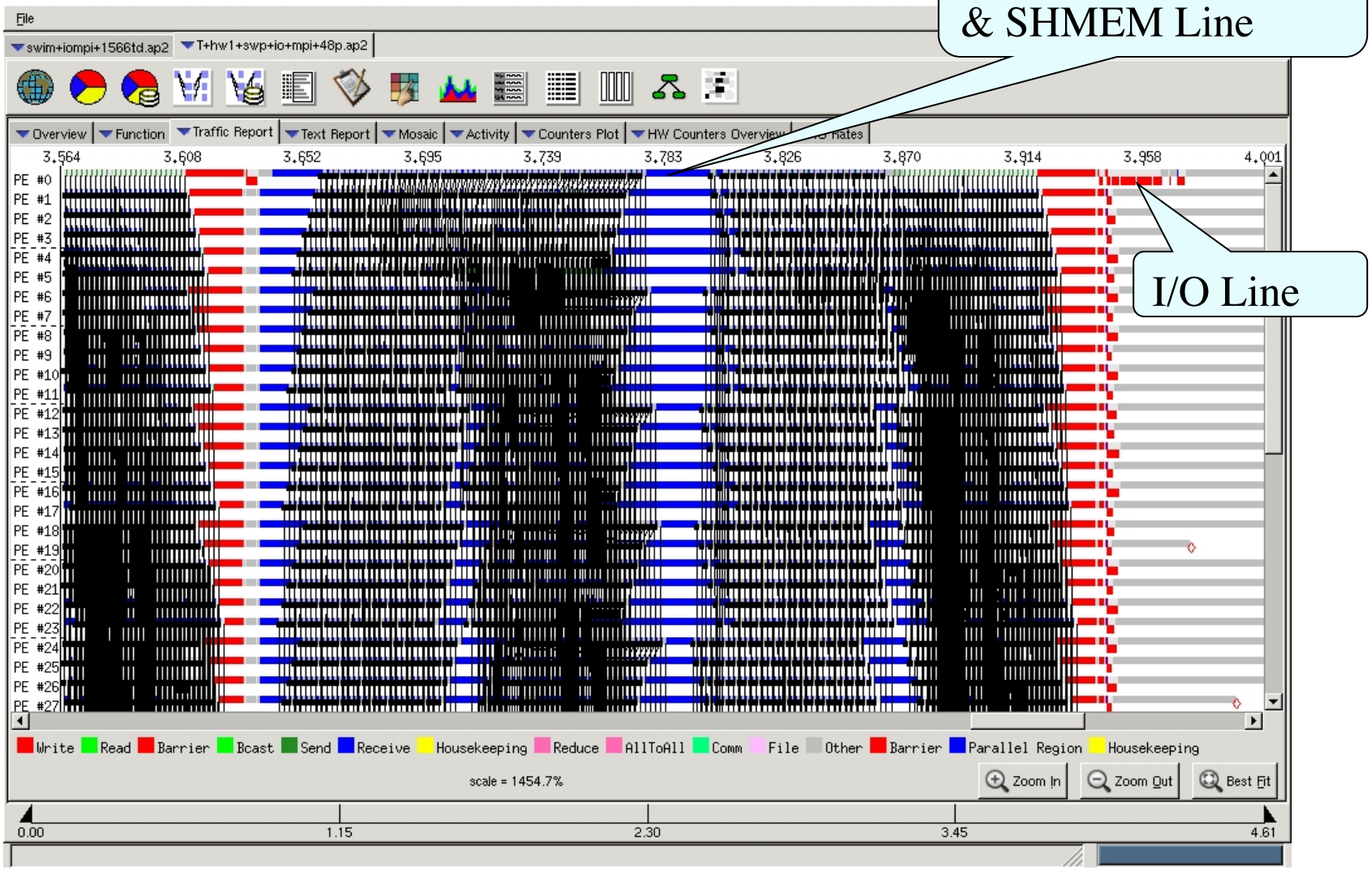
- Full trace (sequence of events) enabled by setting `PAT_RT_SUMMARY=0`
- Helpful to see communication bottlenecks.
- Use it only for small experiments !



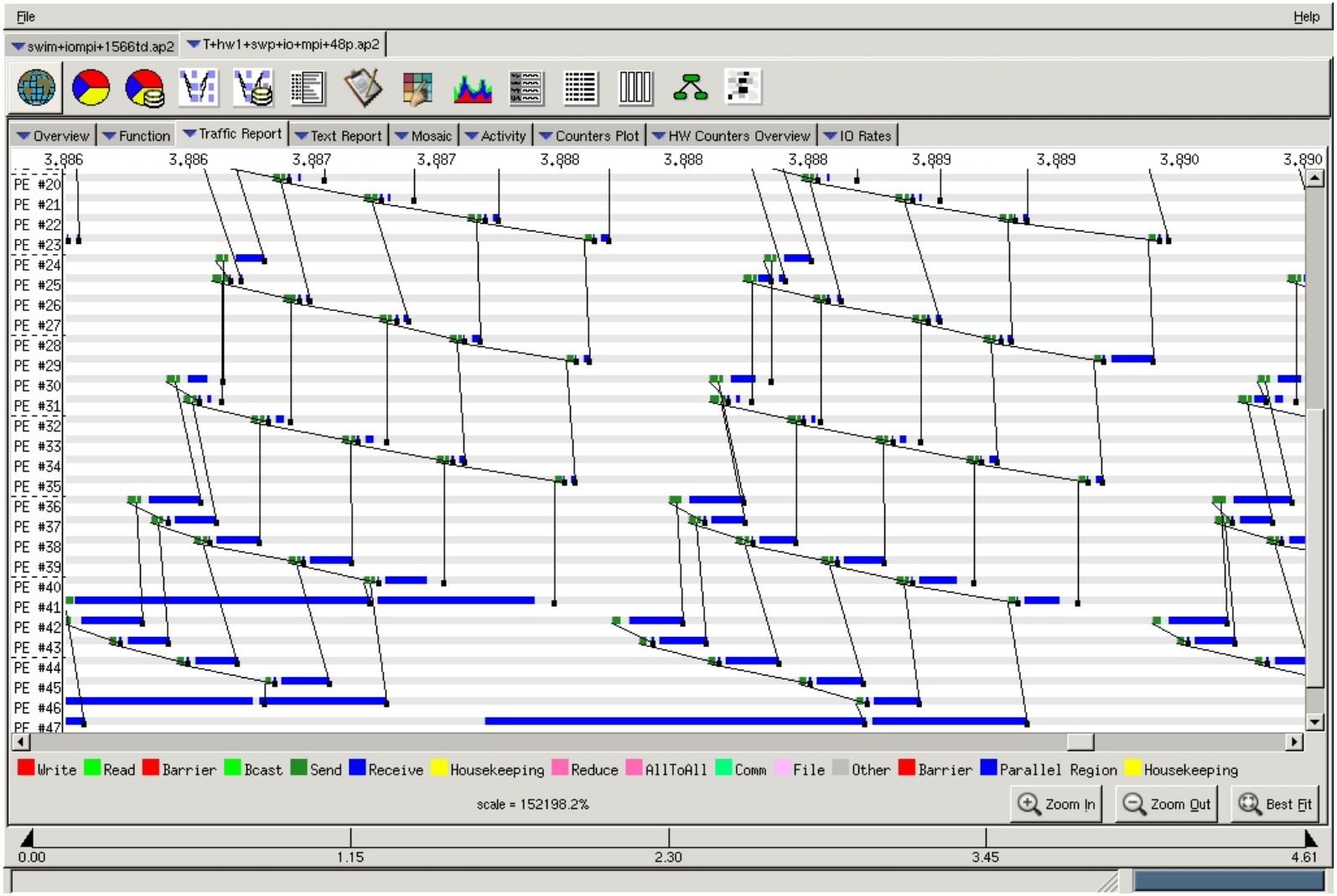
Traffic Report

- **Shows internal PE-to-PE traffic over time.**
  - Broken out by communication type (Send, Recv, Sync, AllToAll, Bcast, Housekeeping and User/other).
- **While this report is displayed, you can:**
  - Hover over an item to display quantitative information.
  - Zoom in and out
  - Right-click an area of interest to open a popup menu, which enables you to hide the origin or destination of the call or go to the call site in the source code, if the source file is available.
  - Access alternate zoom in and out controls, or filter the communications shown on the report by the duration of the messages. Filtering messages by duration is useful if you are only interested in a particular group of messages.
- **Often quite dense, typically requires zooming in to reveal meaningful data.**
- **Look for large blocks of barriers that are being held up by a single PE.**
  - May indicate that the single PE is waiting for a transfer, or it can also indicate that the rest of the PEs are waiting for that PE to finish a computational piece before continuing.

Time Line View (Zoom) – Traffic Report



Time Line View (Fine Grain Zoom)



Other Reports Available

- **Mosaic Report**
  - Matrix of communications between source and destination PEs, using colored blocks to represent the relative point-to-point send times between PEs
    - Based on average communication times by default
    - Color-coded: Light green blocks indicates good values, while dark red blocks may indicate problem areas. Hover the cursor over any block to show the actual values associated with that block.
    - Scroll through the report and look for red "hot spots." These are generally an indication of bad data locality and may represent an opportunity to improve performance by better memory or cache management.
- **Activity Report**
  - Shows communication activity over time, bucketed by logical function such as synchronization. Compute time is not shown.
  - Look for high levels of usage from one of the function groups, either over the entire duration of the program or during a short span of time that affects other parts of the code. You can use the calipers to filter out the startup and closeout time, or to narrow the data being studied down to a single iteration.
- **I/O Rates**
  - Table listing quantitative information about the program's I/O usage (only avaialble if I/O data was collected during program execution). Can be sorted by any column, in either ascending or descending order
  - Look for I/O activities that have low average rates and high data volumes.