More on Application profiling and optimization

Agenda

- Optimizing source code and controlling the compiler
- A Tour of the Apprentice2 GUI
- Optimizations for MPI – Rank Reordering

Doesn't the compiler do everything?

- **Not yet...**
  - Standard answer, unchanged for last 50 or so years
- **What does it do**
  - It tries to compile the loops in your application to be as fast as possible
  - Performance depends on reducing memory use and using the best machine instructions (vectorization)
  - This means your code may be significantly transformed
- **What can you do**
  - Work out what you care about (profile)
  - Experiment with alternative source implementations but a lot of expertise is needed here
  - Give the compiler additional information
  - Use compiler output to determine what it is doing and influence it via directives

Loop optimisation techniques

- **Most HPC codes are loop-based**
  - Repeatedly process all the elements of an array
- **There are various optimization techniques for loops**
  - unrolling/unwinding
  - stripmining
  - blocking/tiling
- **We are not going to explain HOW to do this manually but it is useful to be aware of these even if you are not going to optimise source**
- **In many cases, the compiler does these automatically**
  - the material here will help you understand what the compiler did
  - if necessary, you can then step in to assist the compiler

EXAMPLE 1: Loop unrolling/unwinding

- **Unrolling and unwinding are equivalent terms**
- **Replaces a loop by an equivalent set of statements**
  - Removes the overhead of loop control logic
    - incrementing the loop index counter
    - checking if the counter has exceeded the loop bounds

- **Most important for small tripcount/low work loops**
  - Especially when nested inside other loops
  - Full unwinding requires tripcount to be known at compile time

Original code	After unwinding

`do i=1,N`
` a(i)=a(i) + b(i)`
`enddo`
	`a(i)  =a(i)   + b(i)`
	`a(i+1)=a(i+1) + b(i+1)`
	`a(i+2)=a(i+2) + b(i+2)`
	`:`
	`a(N)  =a(N) + b(N)`

Example 2: Loop blocking/tiling

- **Applied to multi-dimensional loopnests**
  - Two or more loops are stripmined
  - Loop interchange moves the strip loops innermost
- **Most often used to preserve memory locality**

Original loopnest	Equivalent explicit code

`do j = 1,Nj`
` do i = 1,Ni`
`  !stencil`
` enddo`
`enddo`
	`do jb = 1,Nj,16`
	` do ib = 1,Ni,16`
	`  do j = jb,jb+16-1`
	`   do i = ib,ib+16-1`
	`    !stencil`
	`   enddo`
	`  enddo`
	` enddo`
	`enddo`

- (strictly, upper strip loop limits should be MIN(Nj,jb+16-1) and similar)

Control: Example blocking with Cray Directives

- **CCE blocks well, but it sometimes blocks better with help**

Original loopnest	Loopnest with help	Equivalent explicit code

`do k = 1,Nk`
` do j = 1,Nj`
`  do i = 1,Ni`
`   ! stencil`
`  enddo`
` enddo`
`enddo`
	`!DIR$ BLOCKABLE(j,k)`
	`!dir$ BLOCKINGSIZE(16)`
	`do k = 1,Nk`
	`!dir$ BLOCKINGSIZE(20)`
	` do j = 1,Nj`
	`  do i = 1,Ni`
	`   ! stencil`
	`  enddo`
	` enddo`
	`enddo`
		`do kb = 1,Nk,16`
		` do jb = 1,Nj,20`
		`  do k = kb,kb+16-1`
		`   do j = jb,jb+20-1`
		`    do i = 6, nx-5`
		`     ! stencil`
		`    enddo`
		`   enddo`
		`  enddo`
		` enddo`
		`enddo`

  - (again, upper limits should be MIN(Nk,kb+16-1) and similar)

- **Get the loopmark listing**
  - Identifies which loops were blocked
  - Gives the block size the compiler chose

Example 3: Loop interchange

- **One of the simplest cache optimisations**
  - aim to access consecutive elements of arrays in order
- **If multi-dimensional arrays addressed in wrong order**
  - causes a lot of cache misses = bad performance
- **Order loops in loopnest with fastest innermost**
  - Fortran is column-major (LH array index moves fastest)
  - C/C++ is row-major (RH array index moves fastest)

- **Compiler may re-order loops automatically (see loopmark)**

Original loopnest	interchanged code

`do i = 1,N`
`  do j = 1,N`
`    tot = tot + a(i,j)`
`  enddo`
`enddo`
	`do j = 1,N`
	`  do i = 1,N`
	`    tot = tot + a(i,j)`
	`  enddo`
	`enddo`

Optimization for memory access, huge pages

- **Various loop transformations we have seen**
  - Help with memory access order
  - This makes more efficient use of cache
    - Use as much cache as possible
    - Reuse data when it is in cache

- **There is a level beyond cache size to consider**
- **We have virtual memory pages which map to physical pages**
- **The OS keeps track of this in hardware (TLB) and software**
- **As a result we should try to reuse memory within a page**

Using hugepages

- **Load chosen `craype-hugepages*` module**
  - See `module avail craype-hugepages` for list of available options
- **Compile as before**
- **Execute as before, but**
  - Make sure this module is also loaded in PBS jobscript
    - It sets various environment variables

- **Which pagesize is best?**
  - You should try different settings
  - 2M or 8M are usually most successful on Cray XC systems

- **Quick cheat:**
  - no need to rebuild to try a different pagesize
  - can load different hugepages module at runtime
    - compared to that used at compile-time
  - compile-time module enables hugepages in the application
  - runtime module determines the actual size that is used

- **See `man intro_hugepages` for more details**

Vectorisation

- **The most important optimization is for memory access**
- **Then we can think of optimising computation**
- **This will be in loops**
- **Usually only one loop is vectorisable in loopnest**
  - And most compilers (not CCE) only consider inner loop
- **Optimising compilers will use vector instructions**
  - Relies on code being vectorisable
  - Or in a form that the compiler can convert to be vectorisable
    - Some compilers are better at this than others
- **Check the compiler output listing and/or assembler listing**
  - Look for packed SSE/AVX instructions

Helping vectorisation

- **Is there a good reason for this?**
  - There is an overhead in setting up vectorisation; maybe it's not worth it
    - Could you unroll inner (or outer) loop to provide more work?

- **Does the loop have dependencies?**
  - information carried between iterations
    - e.g. counter: `total = total + a(i)`

- **If there are no loop dependencies:**
  - Tell the compiler that it is safe to vectorise
    - `IVDEP` directive above loop (CCE, but works with most compilers)
    - C99: `restrict` keyword (or compile with `-hrestrict=a` with CCE)

  - Perhaps the dependencies are between iterations i and i+8
    - Then it is safe to vectorise with vectors of length 8 or less
    - Use directive: `IVDEP SAFEVL=8`
  - see `man ivdep` for more details

Inhibitors to vectorisation

- **loop dependencies:**
  - The loop cannot be executed in any order
  - Might be hard to rewrite code to fix this
- **Code is not a loop (do while)**
- **Indirect addressing**
- **Non-vectorisable functions**
- **Unknown loop trip count**
- **Function calls in loop need to be inlined**

- **Check the compiler output to see what it did**
  - CCE: `-hlist=a`
  - Intel: `-vec-report[0..5]`
  - GNU: `-ftree-vectorizer-verbose=5`

CCE directives

Some useful CCE directives

- **Compiler directives avoid the need for explicit coding**
  - They are compiler-specific but should be ignored as comments by:
    - other compilers
    - the same compiler, if overridden by compiler options
- **CCE has a large set of optimisation directives**
  - Fortran: `!DIR$ <directive>`
  - C/C++: `#pragma _CRI <directive>`
    - `_CRI` optional; include it so compiler warns about unrecognised directives
- **Some useful ones are listed on the next few slides**
- **For more information:**
  - `man directives`
  - `man <directive name>`
  - Fortran, C/C++ Reference Manuals on docs.cray.com

Selected CCE scalar optimisation directives

- **INTERCHANGE (i,j...), NOINTERCHANGE**
  - Specified loops should be interchanged, e.g. (i,j,k) -> (k,j,i)
  - NOINTERCHANGE directive suppresses loop interchange
- **UNROLL [n], NOUNROLL**
  - Specify unrolling of next loop, with optional unroll factor
- **BLOCKABLE (i,j...)**
  - Specified loops can be blocked
  - NOBLOCKING directive prevents blocking
- **BLOCKINGSIZE (n)**
  - Apply blocking factor n to next loop
  - Use separate BLOCKINGSIZE directives for each loop to be blocked
- **FUSION, NOFUSION, NOFISSION**
  - Control loop fusion and fission of specified loop

Selected CCE vectorisation directives (1)

- **IVDEP**
  - Ignore dependencies in the next loop that might inhibit vectorisation
- **NEXTSCALAR**
  - Do not vectorise the next loop
- **PREFERVECTOR**
  - If more than one loop in nest can be vectorised, indicates preference
  - Has the same effect as VECTOR ALWAYS directive
- **NOVECTOR**
  - Disable vectorisation for rest of program unit;
  - reset behaviour with VECTOR directive

Selected CCE vectorisation directives (2)

- **LOOP_INFO [min_trips(c)] [est_trips(c)] [max_trips(c)]**
  - Provide information on min/mean/max tripcounts for loop
- **PROBABILITY**
  - Indicate probability of a conditional being true
  - May suggest compiler uses gather/scatter methods to vectorise loop
- **PERMUTATION**
  - The specified integer array does not have repeated values
  - Useful for index array used in indirect addressing
- **CONCURRENT**
  - Stronger than IVDEP
    - IVDEP says loop iterations independent in current order
    - CONCURRENT says independent in any order
  - Both CONCURRENT and IVDEP should allow (possible) vectorisation

Concluding remarks

- **Compilers are good at optimising code, but not perfect**
- **If you do nothing else with your code**
  - Make sure you address arrays in the "right" order
  - Check the compiler feedback to see its not doing anything foolish

- **To go further:**
  - Understand what the compiler does
    - Look at the compiler feedback in more detail
    - Use profiling and hardware counters to see if these optimisations work
  - Help the compiler to understand your code
    - Simpler code is usually a good place to start
    - Use directives to give the compiler more information about your code
  - Only start hand-coding optimisations as a last resort

- **And remember to keep profiling your code**
  - optimise the things that take most time

A tour of the Apprentice2 GUI

The Three Stages of CrayPAT

1. Instrumentation

- Use `pat_build` to apply instrumentation to program binaries

2. Data Collection

- Transparent collection via CrayPAT's run-time library

3. Analysis: Sampling / Tracing

- Interpret and visualise data using post-mortem tools:
  1. `pat_report`: a command line tool for generating text reports
  2. **Cray Apprentice2: a graphical performance analysis tool**
  3. **Reveal**: graphical performance analysis and code restructuring tool

Profile Visualization with Cray Apprentice2

Cray Apprentice2

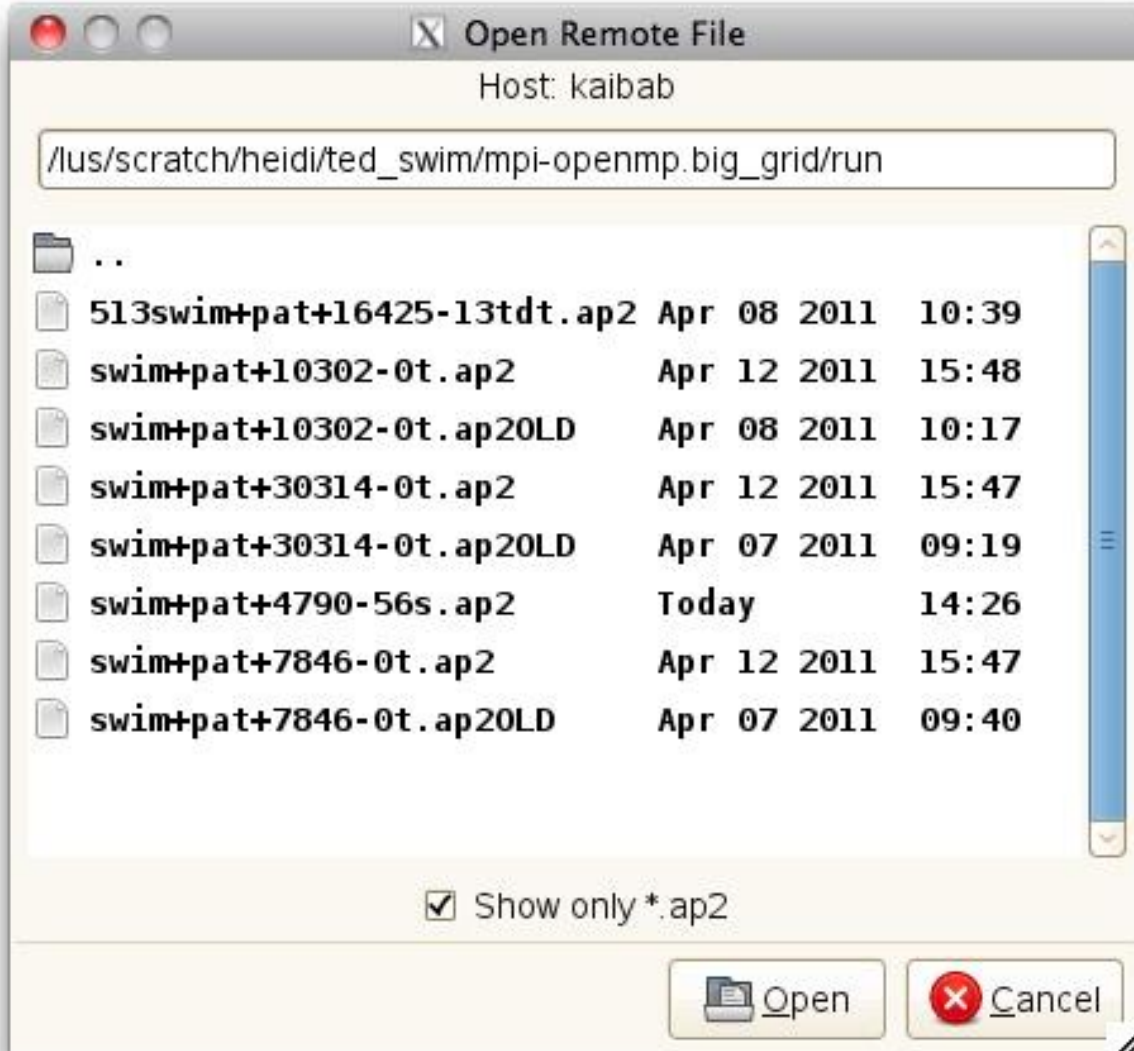
- **Features:**
  - Call graph profile
  - Communication statistics
  - Time-line view
    - Communication
    - I/O
  - Activity view
  - Pair-wise communication statistics
  - Text reports
  - Source code mapping
- **Helps identify:**
  - Load imbalance
  - Excessive communication
  - Network contention
  - Excessive serialization
  - I/O Problems

To use Cray Apprentice²

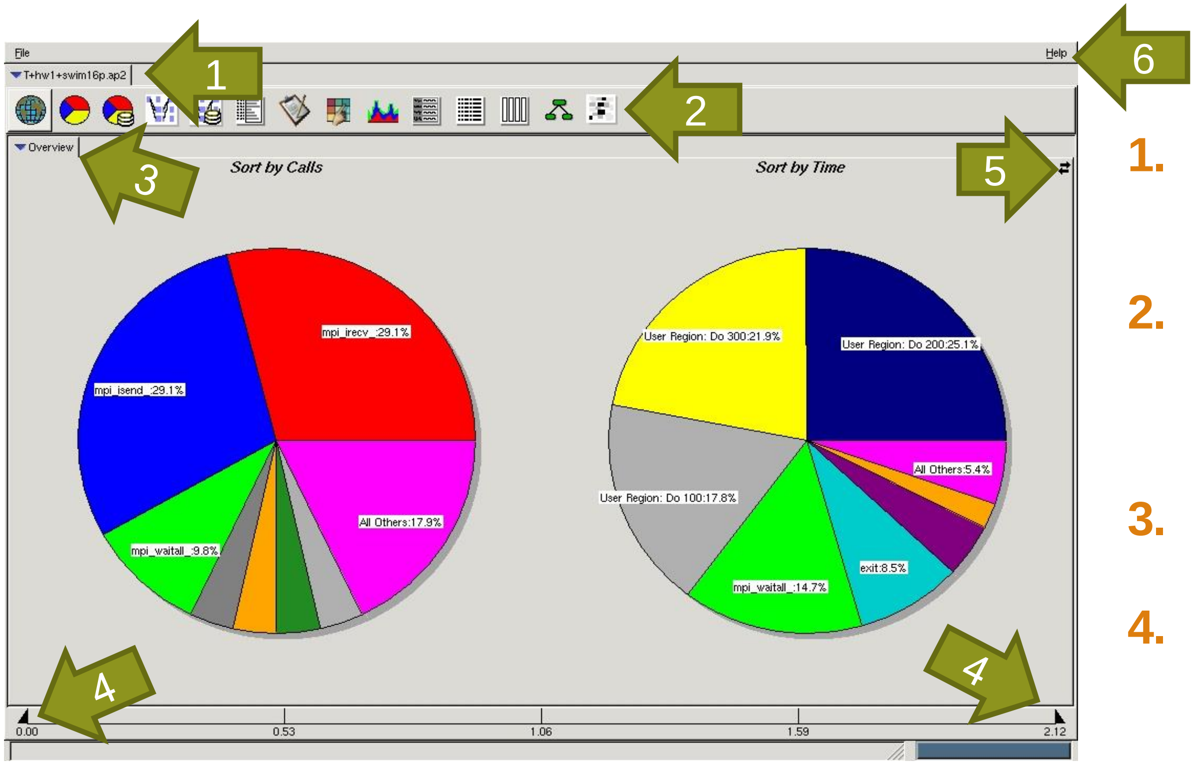
- **You can run app2 on the login nodes:**
  - You need an X session
    - `ssh -X <system name>`
    - and software to catch X windows on your local machine
  - You need `app2` in your path
    - `module load perftools`
  - The *.ap2 file contains the information (produced by pat_report)
    - `app2 data_file_name.ap2`
    - or you can load the ap2 file from the GUI

- **There is also a client version of app2**
  - You can run this on your local machine
  - Contact your site administrator for details on how to install this
  - Then just need to copy the *.ap2 file to this machine

Accessing Remote File from app2 Client



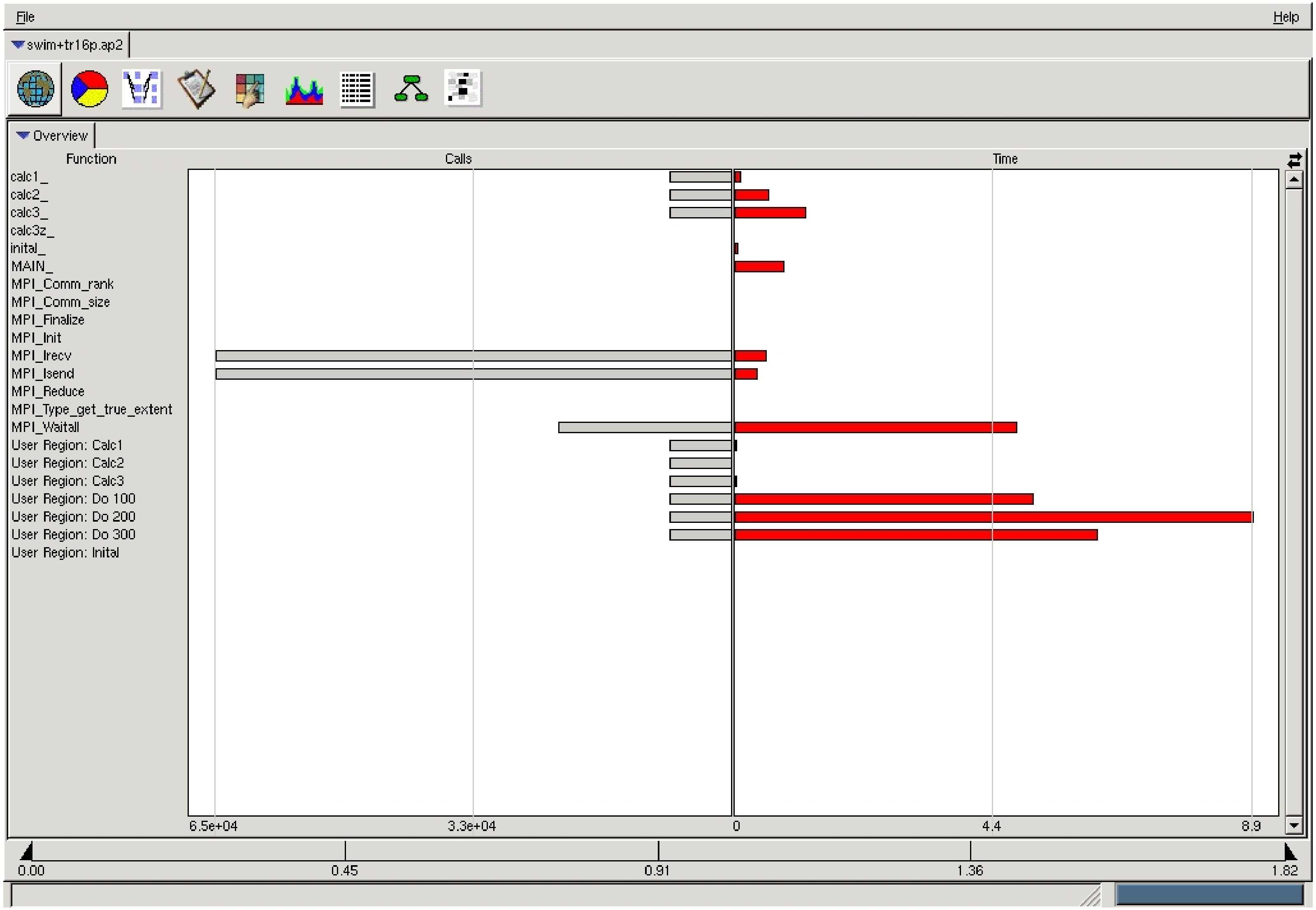
Statistics Overview: Pie Chart



Note that report toolbar ONLY what you have decide to collected with pat_build

1. **Data tab: shows the name of the data file currently displayed**
2. **Report toolbar: show the reports that can be displayed for the data currently selected**
3. **Report tabs: show the reports**
4. **On many reports, the total duration of the experiment is shown as a graduated bar at the bottom of the window**
5. **Change view from pie chart to bar graph**
6. **Help menu**

Statistics Overview: Bar Graph



Function Profile View

File Help

swim+iompi+1566td.ap2 | T+hw1+swp+io+mpi+48p.ap2

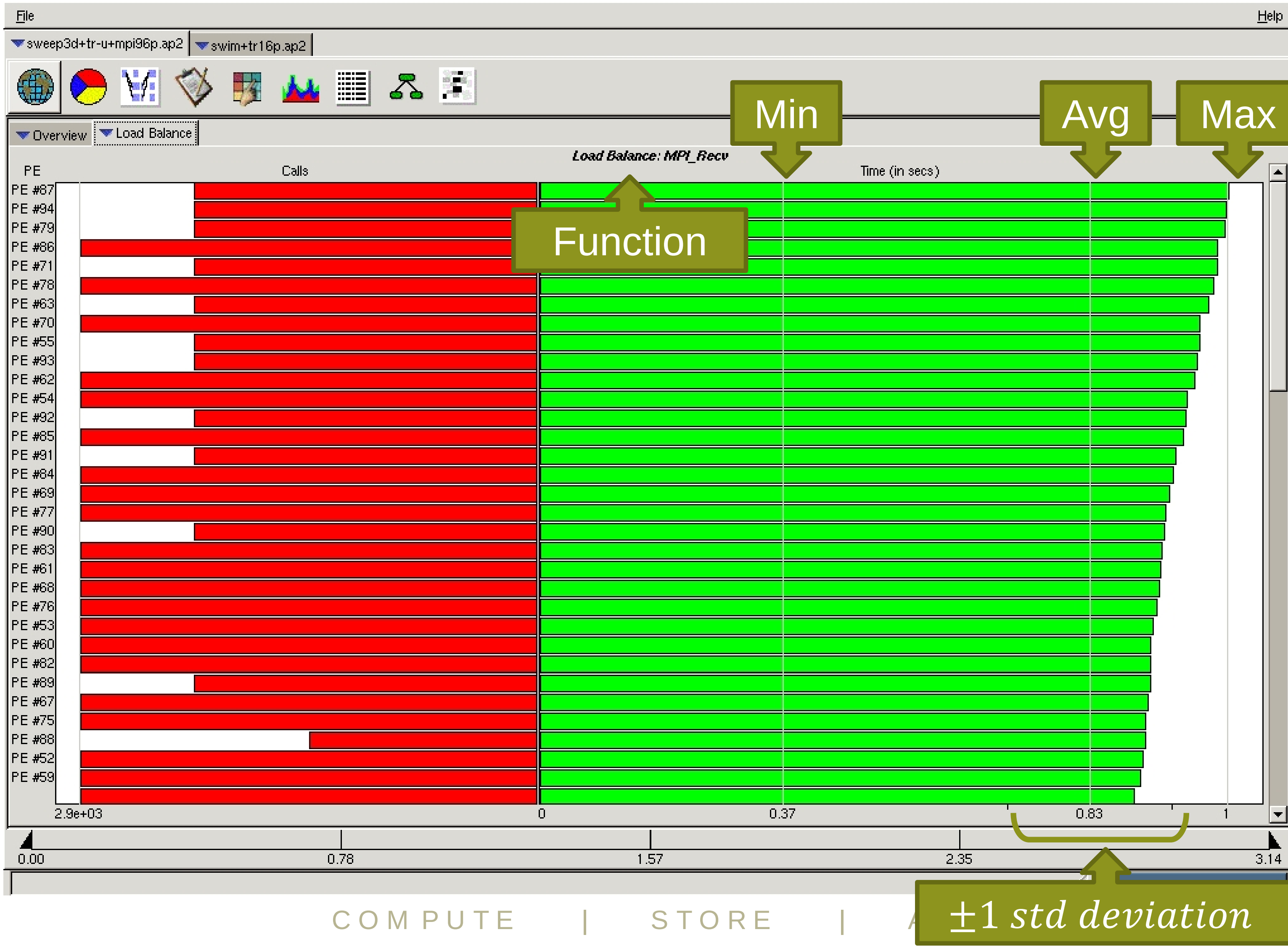
Overview | Function

Time	Percent	Hits	Callsites	Imbalance %	Potential Savings	Function	Line	File
124.175511	63.29	576	1	5.63	0.15	sweep_	116	/us/nid00036/dr/Apps/sweep3d/sweep.f
40.211774	20.50	118080	1	23.40	0.25	mpi_recv_	0	==NA==
16.319527	8.32	48	1	48.26	0.30	exit	35	/notbackedup/users/rsrel/rs64.REL_1_4_33.060914.Thu/pe/computelibs/glibc/stdlib/exit.c
6.173236	3.15	1536	3	50.00	0.12	mpi_allreduce_	0	==NA==
2.760376	1.41	118080	1	17.58	0.01	mpi_send_	0	==NA==
2.250029	1.15	576	1	2.62	0.00	source_	18	/us/nid00036/dr/Apps/sweep3d/source.f
1.984620	1.01	144	1	2.59	0.00	mpi_barrier_	0	==NA==
0.867359	0.44	192	2	2.47	0.00	mpi_bcast_	0	==NA==
0.416231	0.21	576	1	2.60	0.00	flux_err_	17	/us/nid00036/dr/Apps/sweep3d/flux_err.f
0.382130	0.19	118080	2	10.98	0.00	snd_real_	135	/us/nid00036/dr/Apps/sweep3d/mpi_stuff.f
0.237772	0.12	309	1	95.76	0.07	fwrite	36	/notbackedup/users/rsrel/rs64.REL_1_4_33.060914.Thu/pe/computelibs/glibc/libio/iofwrite.c
0.185067	0.09	118080	2	17.30	0.00	rcv_real_	164	/us/nid00036/dr/Apps/sweep3d/mpi_stuff.f
0.067832	0.03	48	1	4.56	0.00	initialize_	42	/us/nid00036/dr/Apps/sweep3d/initialize.f
0.059407	0.03	48	1	4.99	0.00	initxs_	77	/us/nid00036/dr/Apps/sweep3d/initialize.f
0.041371	0.02	48	1	23.84	0.00	inner_	72	/us/nid00036/dr/Apps/sweep3d/inner.f
0.023948	0.01	8	1		0.00	fputc	35	/notbackedup/users/rsrel/rs64.REL_1_4_33.060914.Thu/pe/computelibs/glibc/libio/fputc.c
0.016902	0.01	68	1		0.00	getc	36	/notbackedup/users/rsrel/rs64.REL_1_4_33.060914.Thu/pe/computelibs/glibc/libio/getc.c
0.008104	0.00	4992	2	28.14	0.00	octant_	17	/us/nid00036/dr/Apps/sweep3d/octant.f
0.002457	0.00	576	1	18.79	0.00	global_real_max_	321	/us/nid00036/dr/Apps/sweep3d/mpi_stuff.f
0.002083	0.00	48	1	69.55	0.00	MAIN_	72	/us/nid00036/dr/Apps/sweep3d/driver.f
0.001588	0.00	576	1	39.10	0.00	global_int_sum_	373	/us/nid00036/dr/Apps/sweep3d/mpi_stuff.f
0.001393	0.00	48	1	10.23	0.00	inner_auto_	69	/us/nid00036/dr/Apps/sweep3d/inner_auto.f
0.000982	0.00	48	1	97.74	0.00	task_init_	24	/us/nid00036/dr/Apps/sweep3d/mpi_stuff.f
0.000739	0.00	384	2	27.87	0.00	global_real_sum_	347	/us/nid00036/dr/Apps/sweep3d/mpi_stuff.f
0.000662	0.00	2	1		0.00	fopen	106	/notbackedup/users/rsrel/rs64.REL_1_4_33.060914.Thu/pe/computelibs/glibc/libio/iofopen.c
0.000499	0.00	48	1	7.54	0.00	initsnc_	175	/us/nid00036/dr/Apps/sweep3d/initialize.f

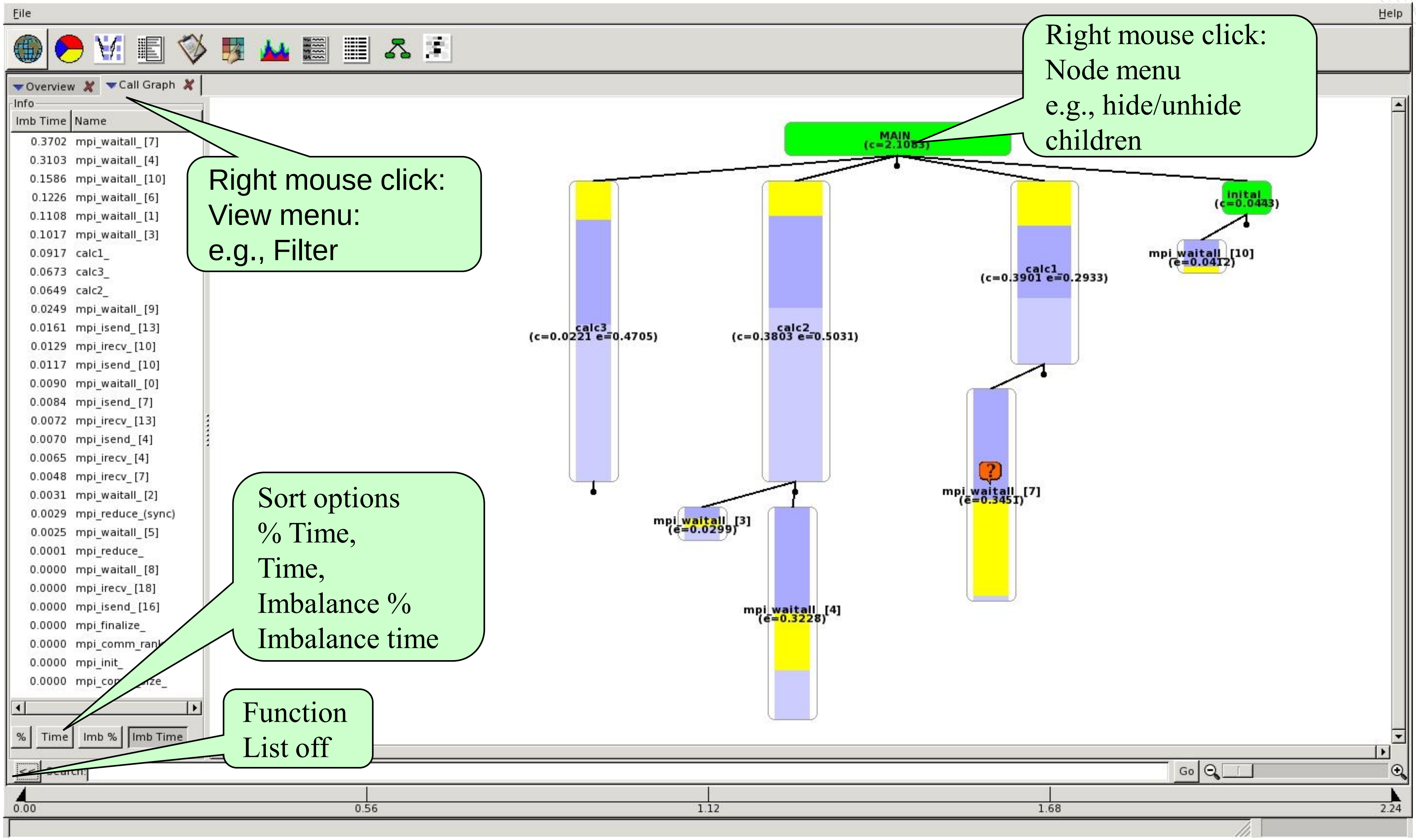
0.00 1.15 2.30 3.45 4.61

Load Balance View (Aggregated from Overview)

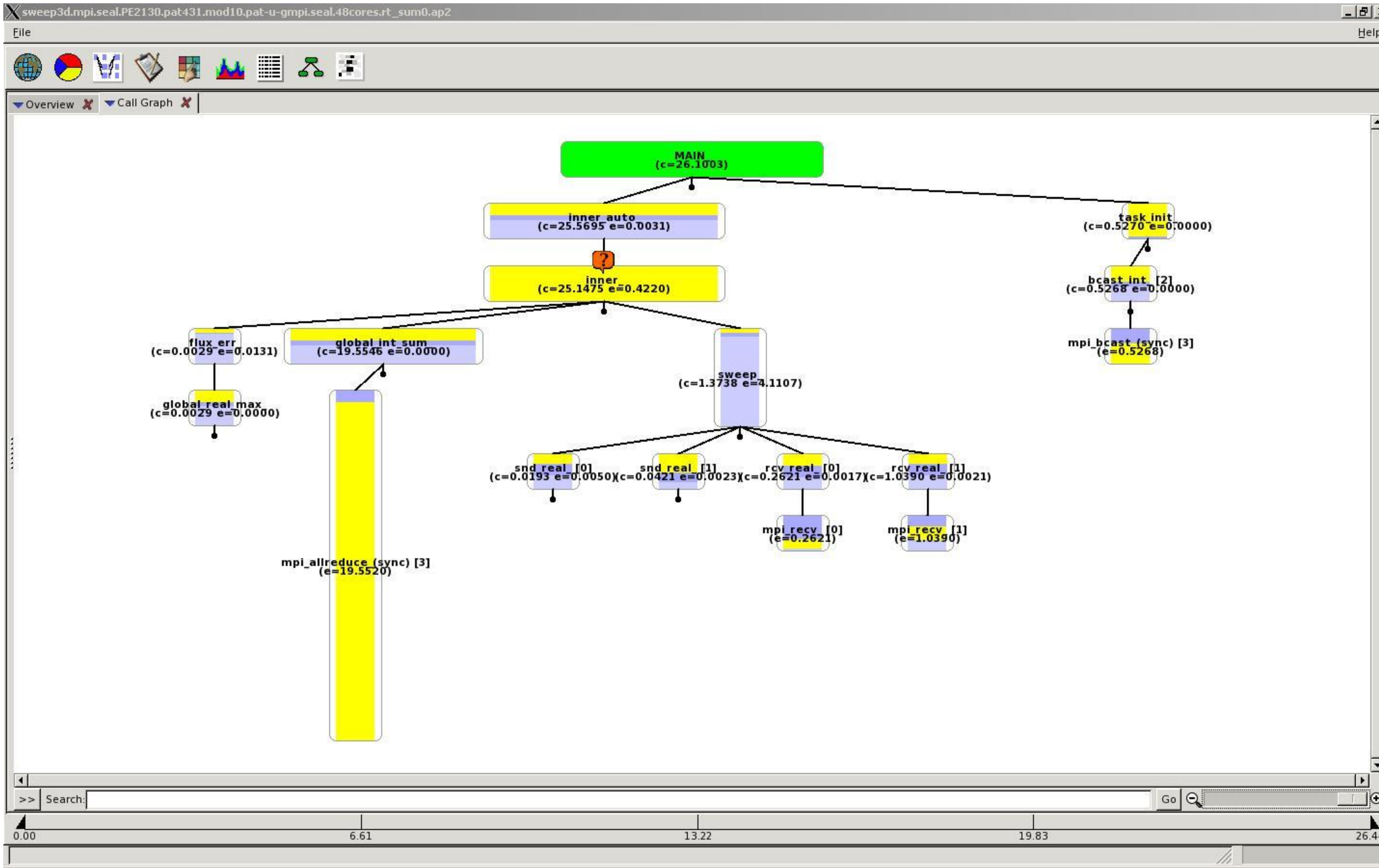
By clicking on a give function, we can show the breakdown per each PE



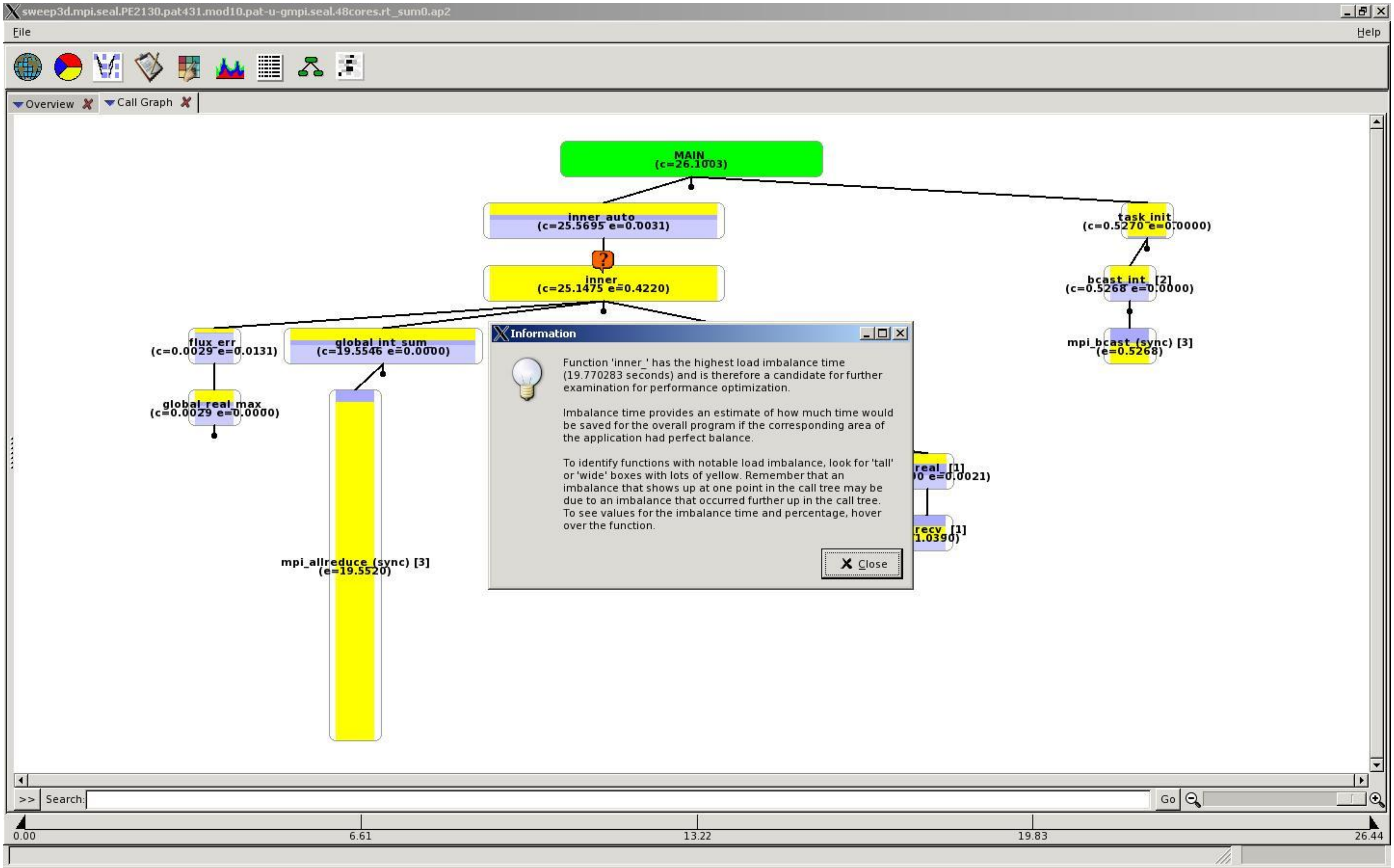
Call Tree View – Function List



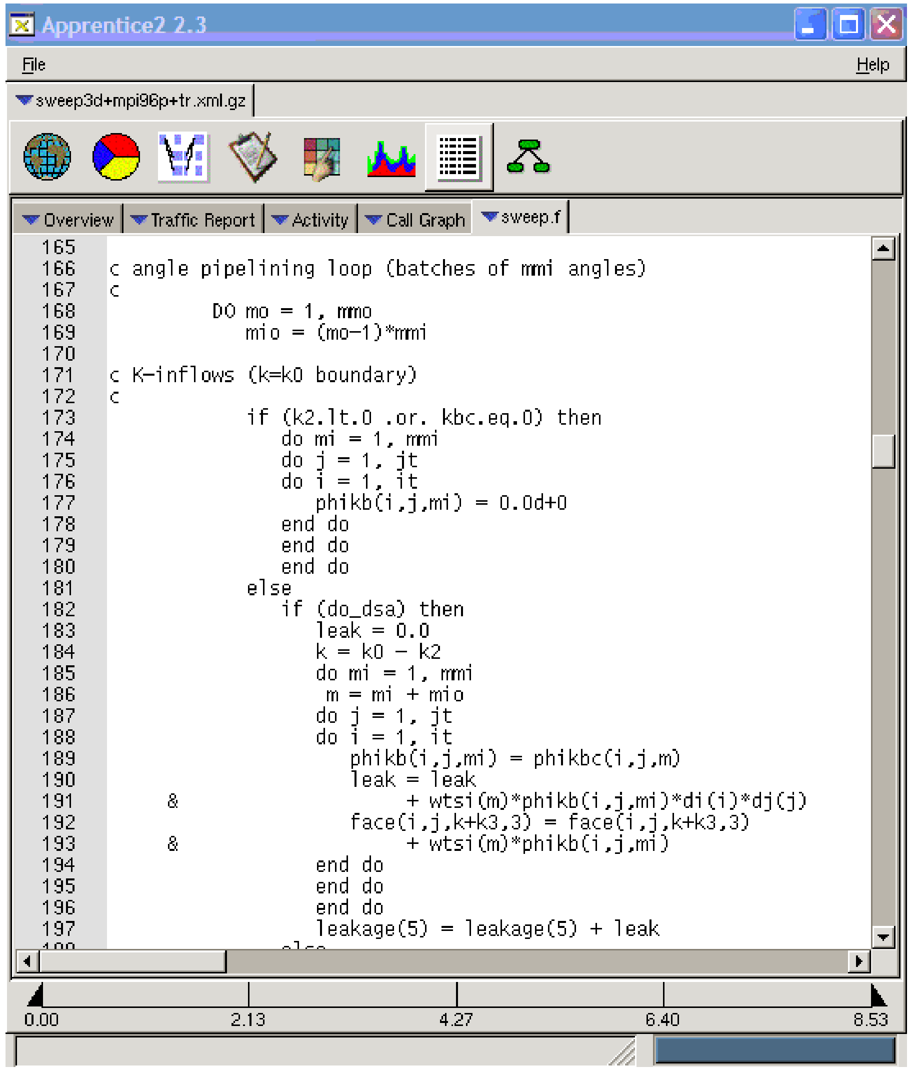
Call Tree Visualization



Discrete Unit of Help (DUH Button)



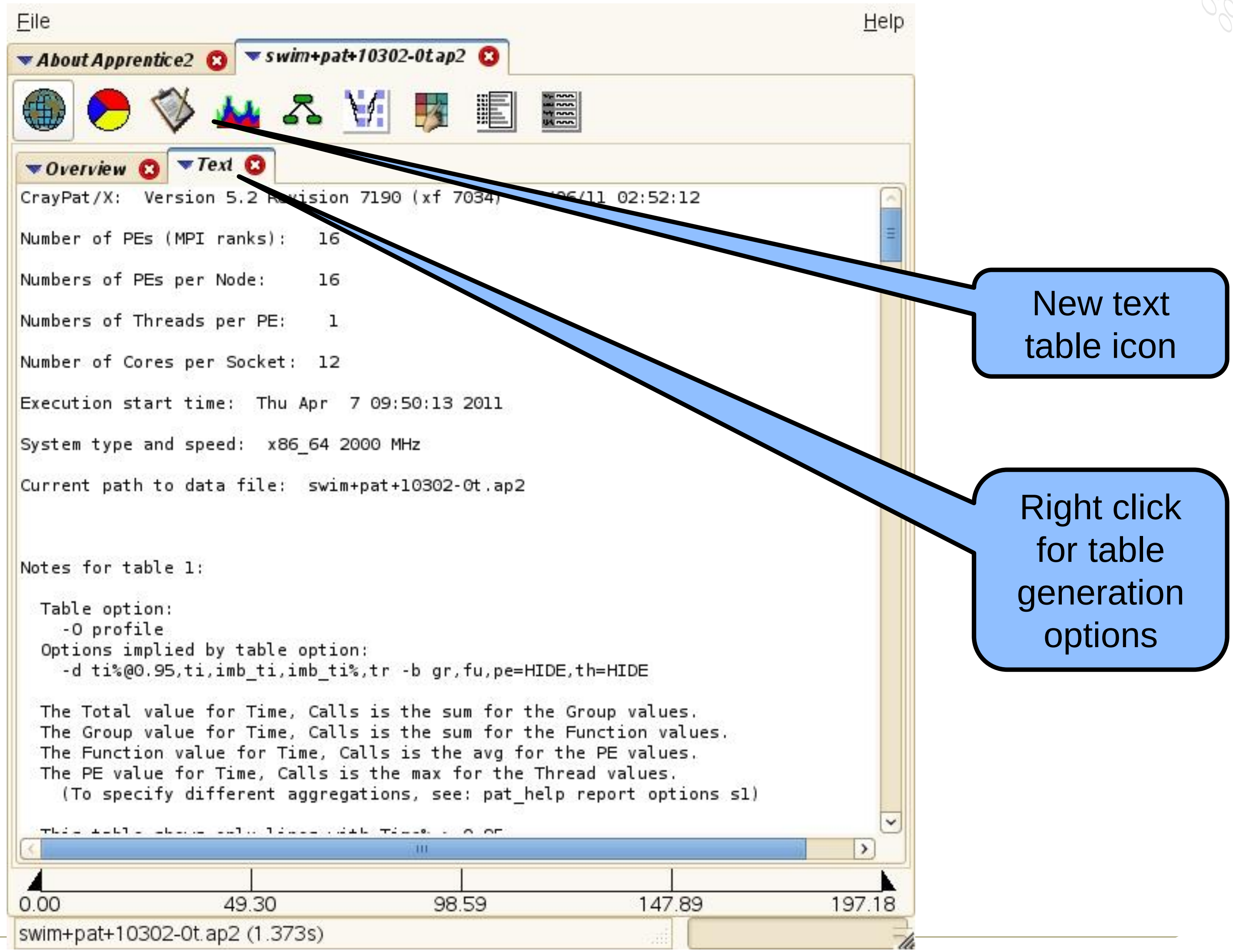
Source Mapping from Call Graph view



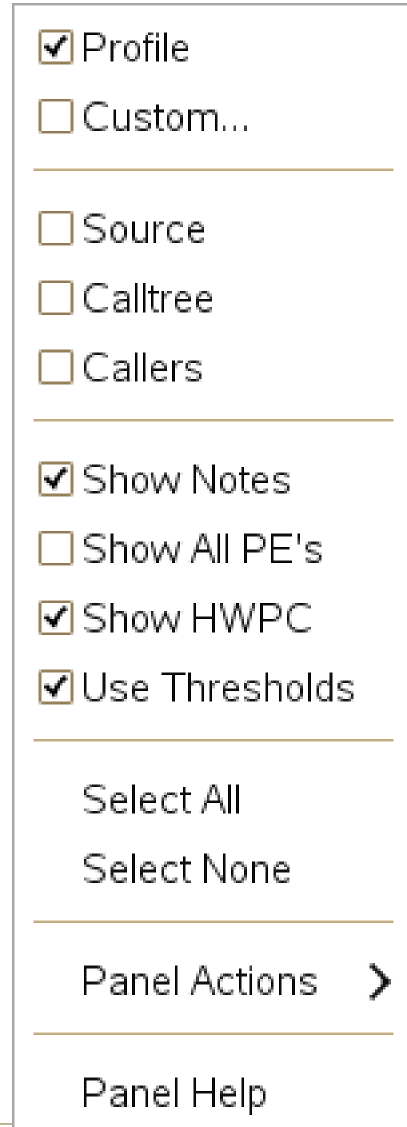
pat_report Tables in Cray Apprentice2

- **Complementary performance data available in one place**
- **Most reports easily accessible**
  - using drop-down menu for easy navigation
- **Can easily generate new views of performance data**
- **Provides mechanism for more in depth explanation of data presented**

Example of pat_report Tables in Cray Apprentice2



Generating New pat_report Tables



Reduce ap2 file information

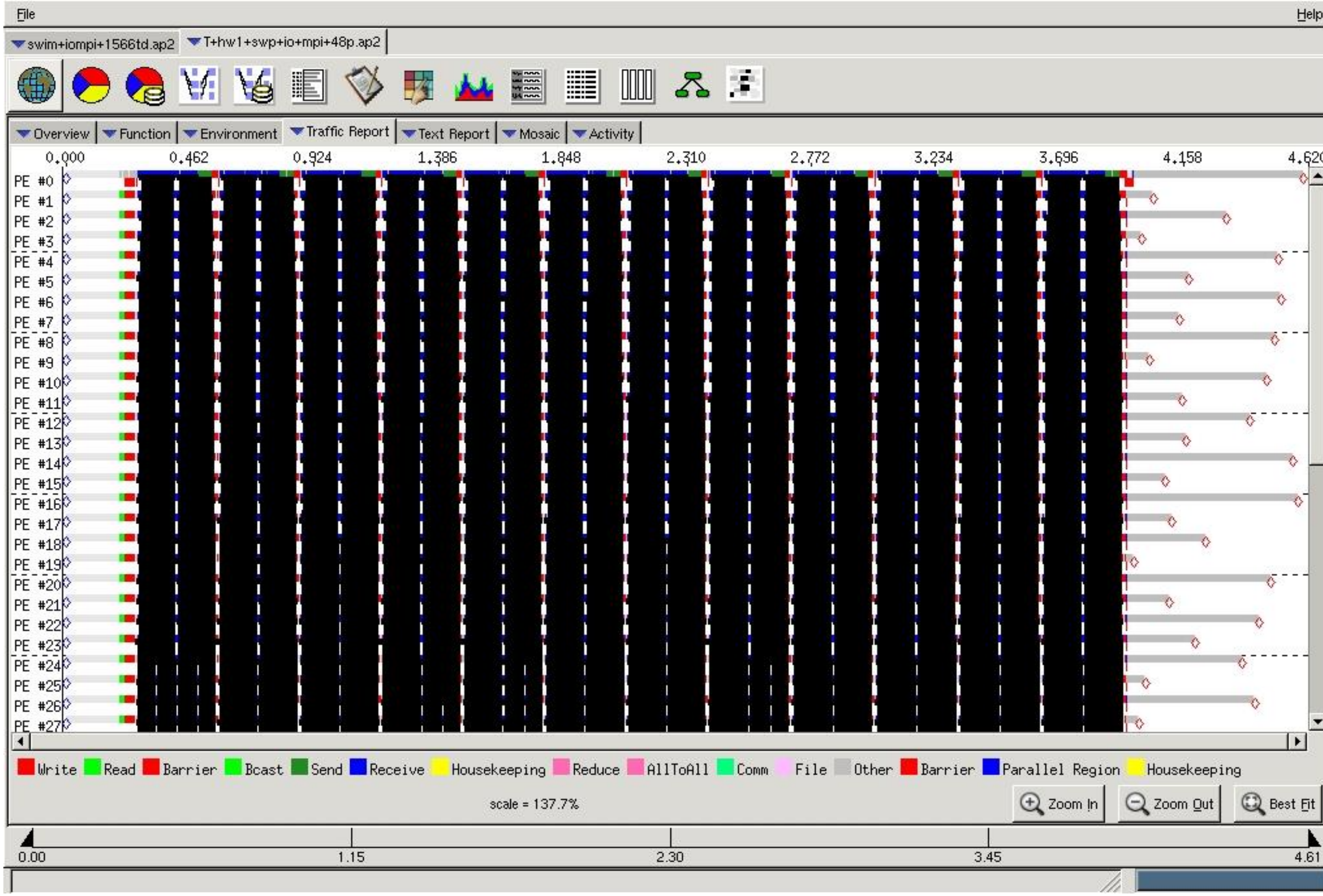
- **Sometimes the amount of data in ap2 file can be large**
  - Very long-running applications
  - Applications running on a large number of PEs
- **The app2 command supports two options to help**
  - `--limit` and `--limit_per_pe`
  - Restrict the amount of data being read in from the ap2 file
  - use K, M, and G abbreviations for kilo, mega, and giga
- **`--limit` sets a global limit on data size.**
- **`--limit_per_pe` sets limit per PE**
  - `--limit_per_pe` generally preferable (not always, but generally)
    - preserves full parallism in analysis
- **Example: first 3M data items**
  - `app2 --limit 3M data_file_name.ap2 &`

Timeline views with Cray Apprentice^2

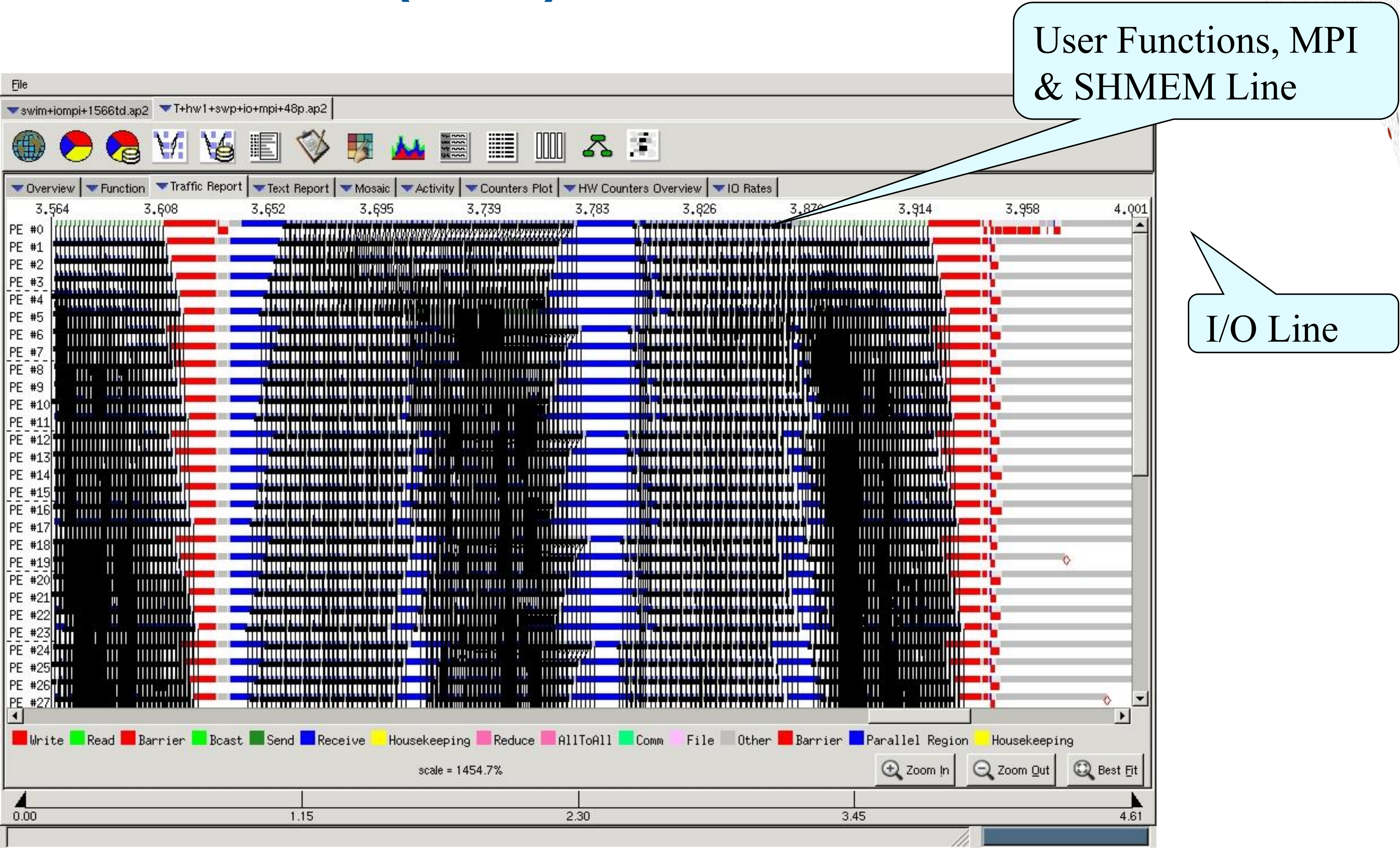
Tracing

- **Show tracing results (Time Live View)**
  - Information broken out by communication type (read, write, barrier, and so on)
- **Only true function calls can be traced**
  - Functions that are inlined by the compiler or that have local scope in a compilation unit cannot be traced
- **Enabled with `pat_build -g, -u, -T` or `-w` options**
- **Full trace (sequence of events) enabled by setting `PAT_RT_SUMMARY=0`**

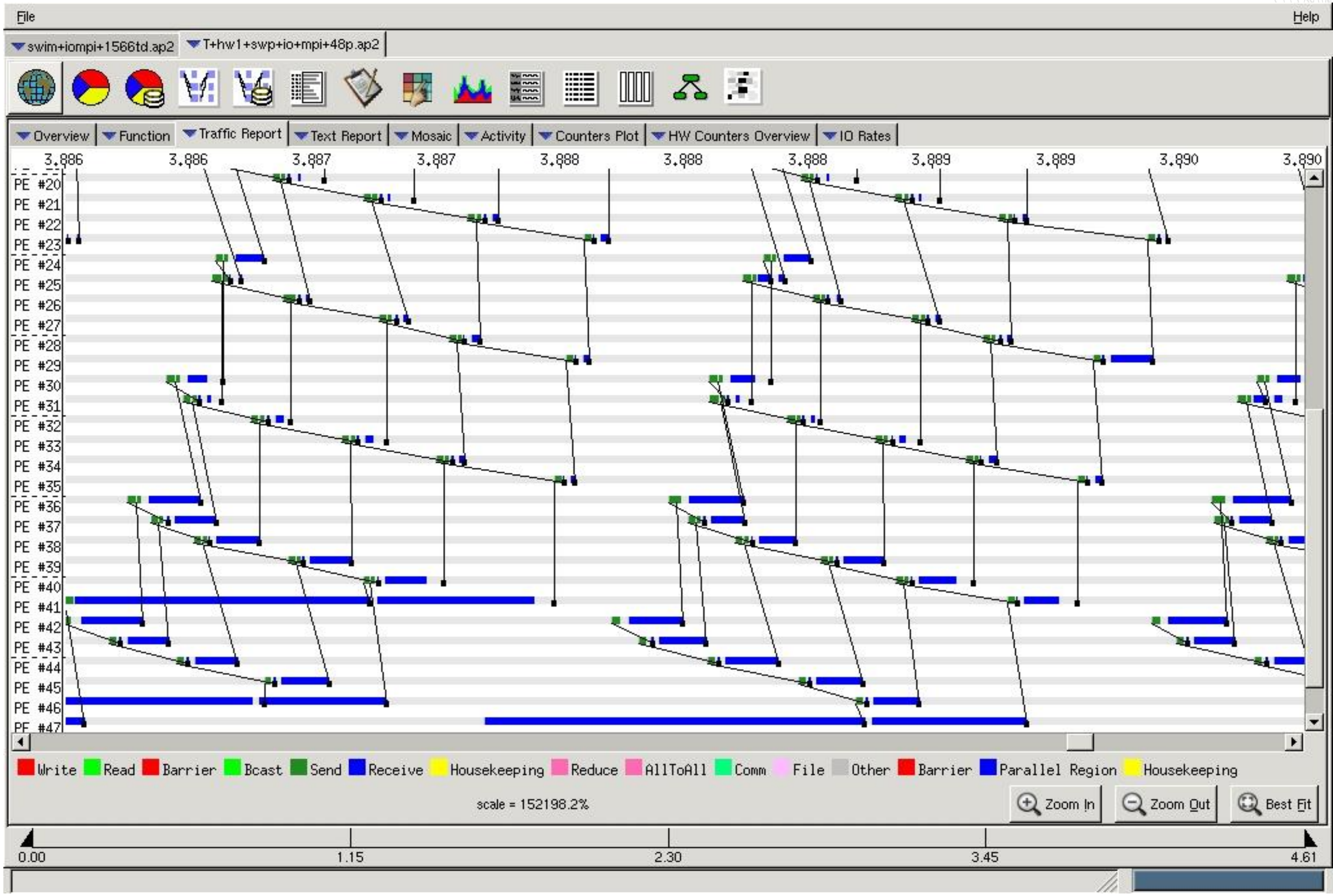
Time Line View (Sweep3D)



Time Line View (Zoom)



Time Line View (Fine Grain Zoom)



Other Cray Apprentice2 Reports

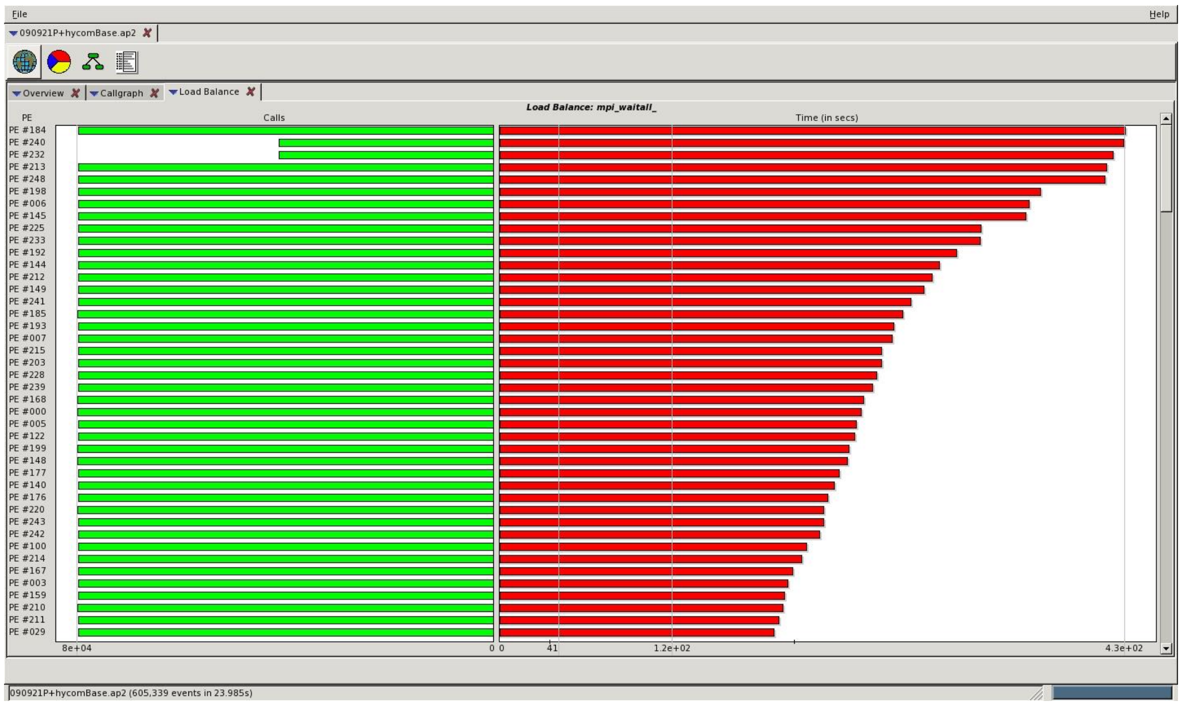
- **Environment reports**
  - Provide general information about the conditions under which the data file currently being examined was created
- **Traffic Report**
  - shows internal PE-to-PE traffic over time. T
    - information is broken down by comm. type (read, write, barrier etc.)
- **I/O Rates Report**
  - table listing quantitative information about program's I/O usage.
    - look for I/O activities that have low average rates and high data volumes;
    - this may indicate that the file should be moved to a different file system.
- **Hardware reports**
  - Available only if hardware counter information was captured

- **Full description at: http://docs.cray.com/books/S-2376-610/**

Optimisations for MPI

Rank Reordering

- Sometimes an MPI application is not well balanced



- The MPI library can reorder the ranks at runtime based on the setting of MPICH_RANK_REORDER_METHOD

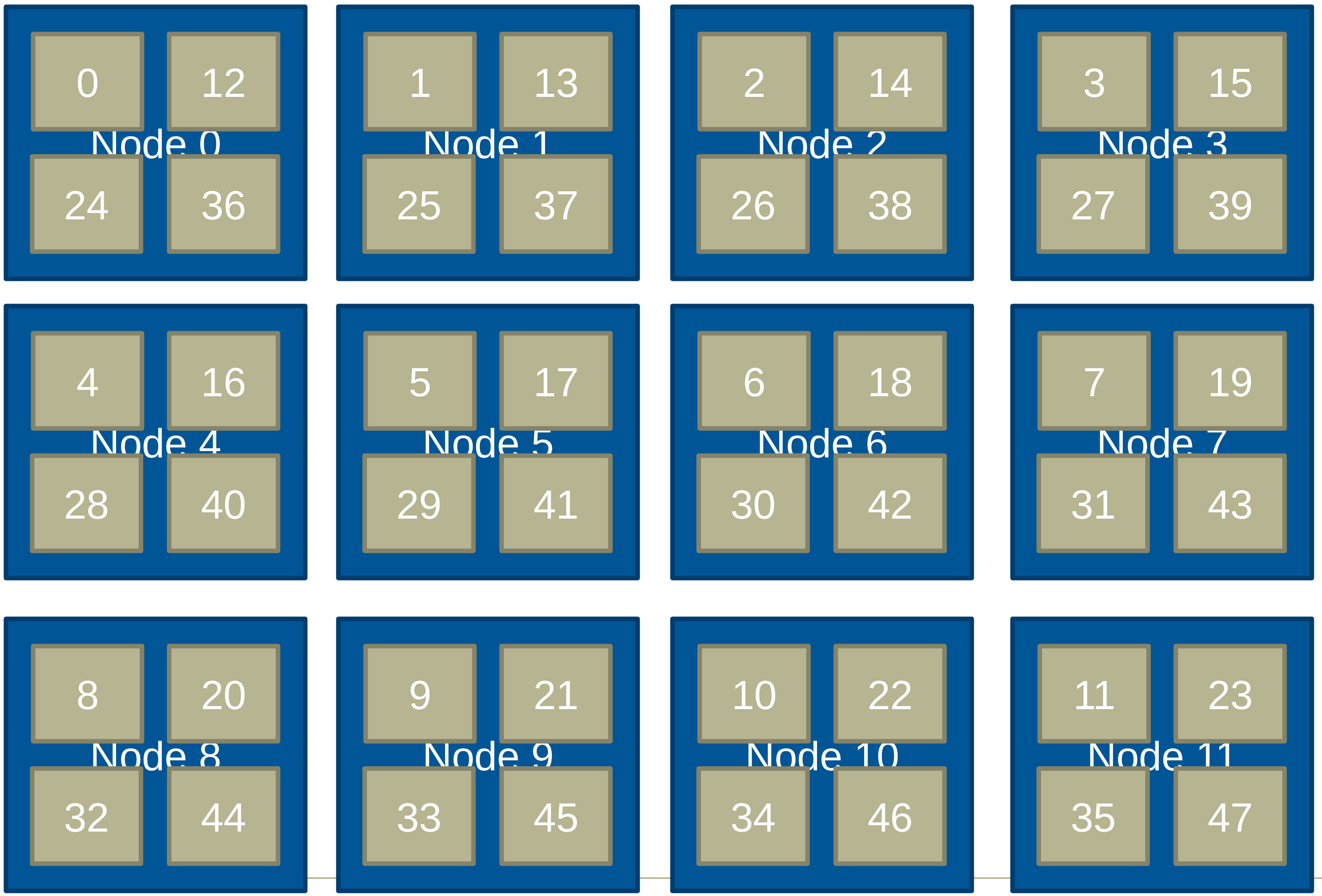
Rank Placement

- **Start with a list of nodes to run on**

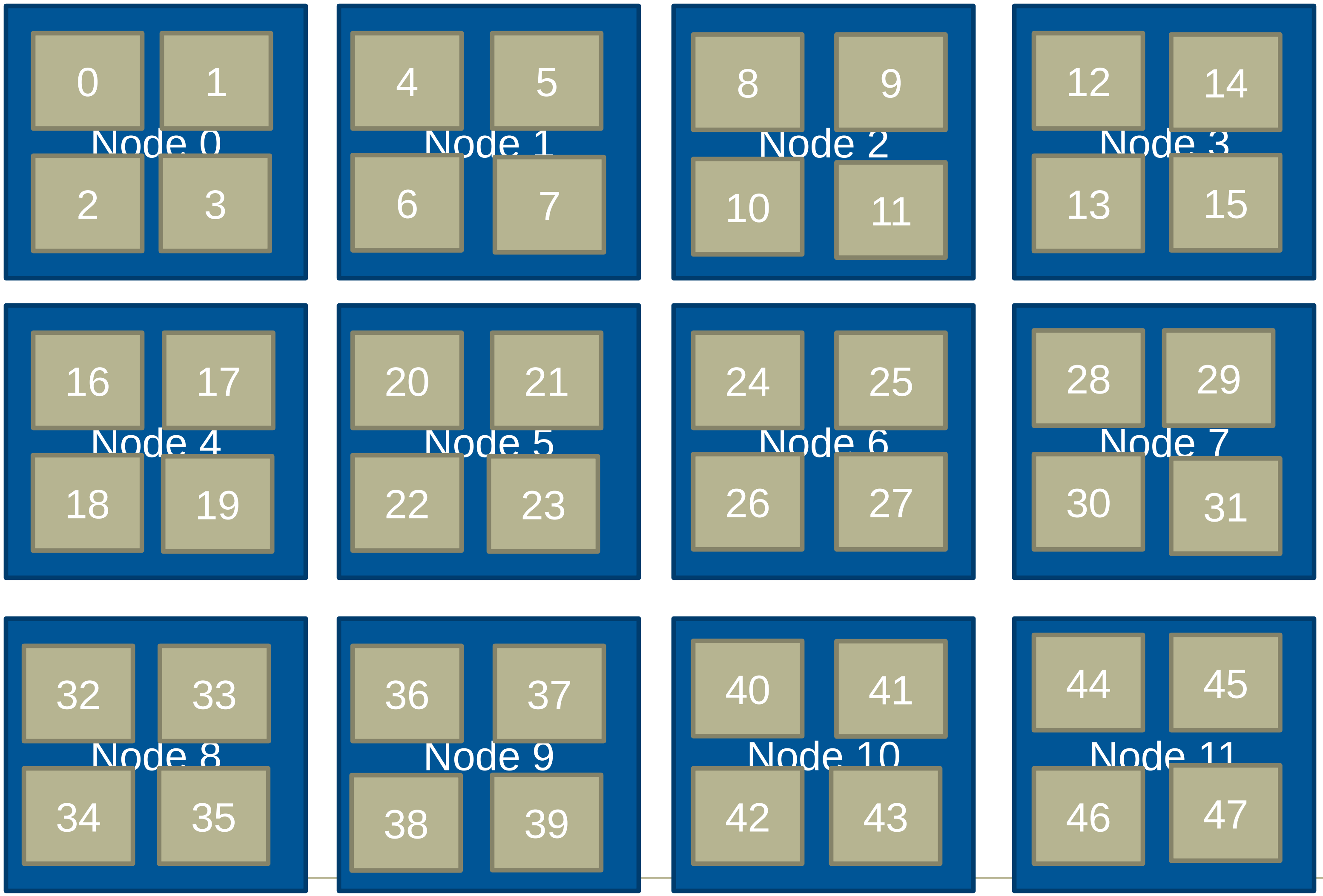
- **0: Round-robin placement**
  - Sequential ranks are allocated one per node in sequence
  - Placement starts again on first node if we reach the last node
- **1: SMP-style placement (default)**
  - Sequential ranks fill up each node in turn
  - Only then move on to the next node
- **2: Folded rank placement**
  - Similar to round-robin placement
  - except each pass over node list is in the opposite direction
- **3: Custom ordering**
  - The location of each rank in turn is specified in a list

- **Examples of these are shown on the next slide**
  - For a simplified example of four cores per node

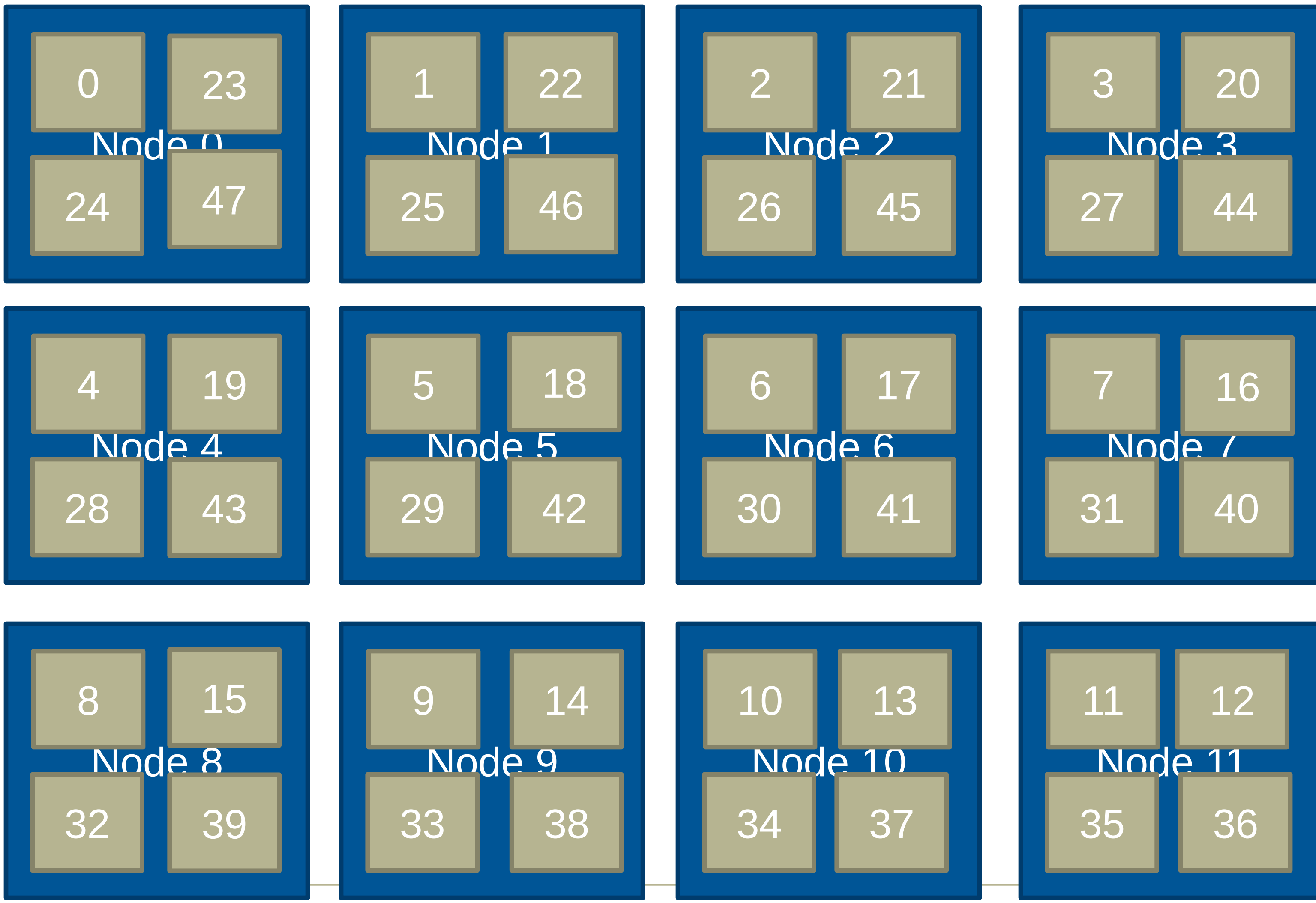
0: Round Robin Placement



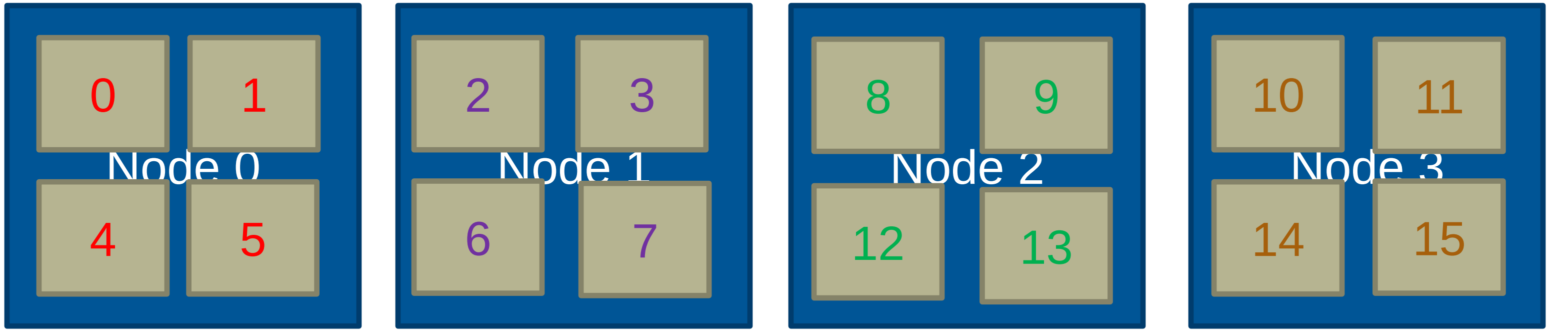
1: SMP Placement (default)



2: Folded Placement



3: Custom Example




```
MPICH_RANK_ORDER: 0,1,4,5,2,3,6,7,8,9,12,13,10,11,14,15
```

- **MPICH_RANK_REORDER=3 enables this**
- **Ordering comes from file MPICH_RANK_ORDER**
  - comma separated ordered list
    - can optionally be condensed into hyphenated ranges
  - all ranks should be included in the list once and only once
- **Nodes are filled up SMP-style**
  - but not with sequential rank numbers
  - instead, take ranks sequentially from the MPICH_RANK_ORDER list

```
MPICH_RANK_ORDER: 0,1,4,5,2,3,6-9,12,13,10,11,14,15
```

Rank placement with CrayPat

```
MPI grid detection:

    There appears to be point-to-point MPI communication in a 20 X 16
    grid pattern. The 27.5% of the total 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 Custom rank order
    from the following table.

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

       Custom   8.092e+09     75.00%  3
          SMP   4.580e+09     42.45%  1
         Fold   2.290e+08      2.12%  2
   RoundRobin   0.000e+00      0.00%  0
```

When testing this the time went only down to 348 from 360 seconds, but approach might become important when scaling higher

Further information from CrayPat

```
Metric-Based Rank Order:

    When the use of a shared resource like memory bandwidth is
    unbalanced across nodes, total execution time may be reduced
    with a rank order that improves the balance.  The metric used
    here for resource usage is: USER Time

    For each node, the metric values for the ranks on that node
    are summed. The maximum and average value of those sums are
    shown below for both the current rank order and a Custom rank
    order that seeks to reduce the maximum value.

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

      Rank    Maximum     Average    Max:Ave   Reduction in Max
     Order      Value       Value      Ratio


     Custom  3.491e+03  3.393e+03       1.03    8.77%
    Current  3.827e+03  3.393e+03       1.13
```

Rank reordering

- **Easy to experiment with**
  - defaults at least should be tested with every application…
  - CrayPAT can help generate the reorder file
- **When might rank reordering be useful?**
  - If point-to-point communication consumes a significant fraction of program time and a load imbalance detected
    - e.g. for nearest-neighbour exchanges (see next slide)
  - Also shown to help for collectives (alltoall) on subcommunicators
  - Spread out I/O servers across nodes
  - If there is a good use case for exploiting the Intel hyperthreads
- **Have used this for I/O servers (NEMO) and radiation colocation (IFS)**