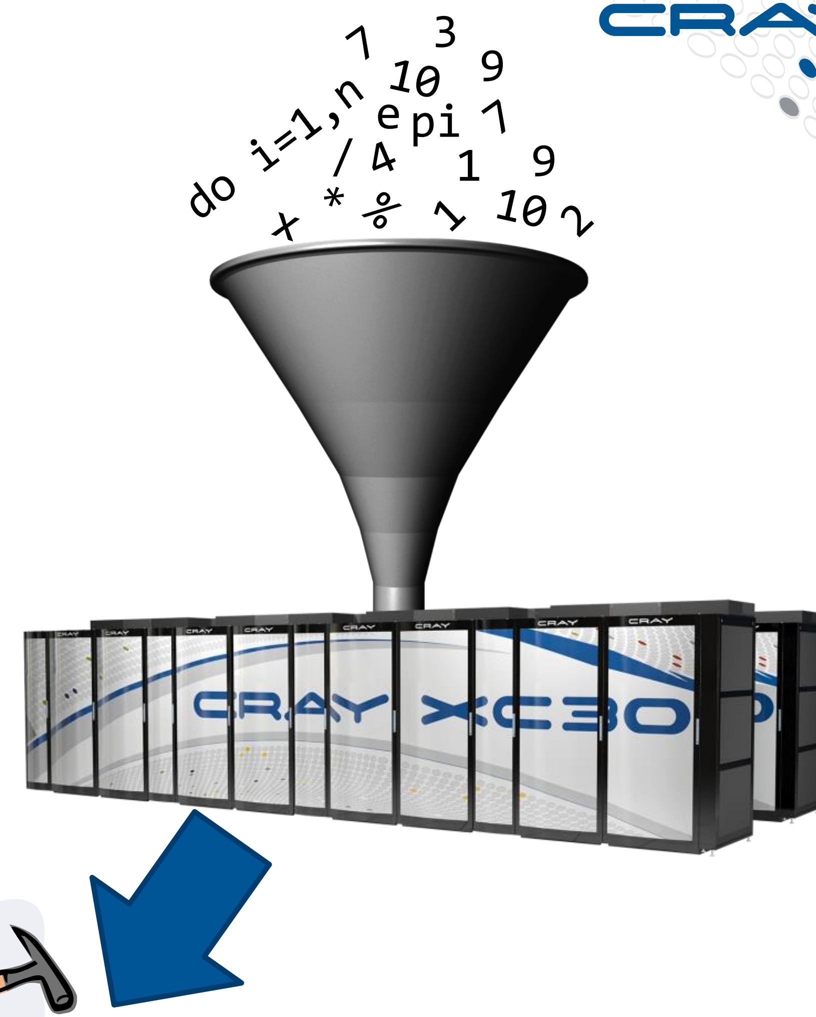
Introduction to Application Performance Analysis with CrayPAT

Performance Optimization

We want to get the most science through a supercomputing system as possible

The more efficient codes are the more productive scientists and engineers can be

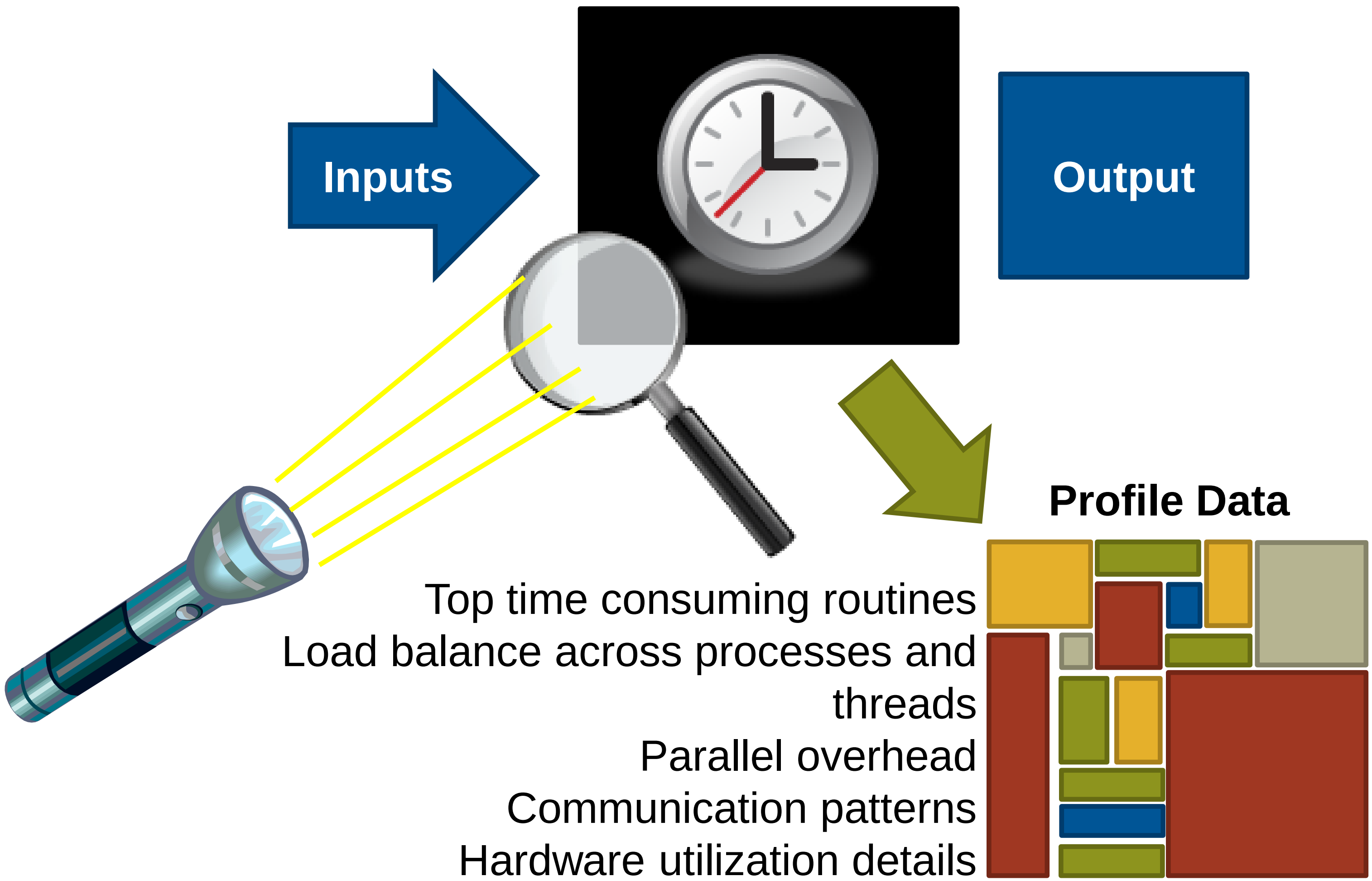


Performance Optimization

- **Adapting the problem to the underlying hardware**
- **Combination of many aspects**
  - Effective algorithms
  - Implementation: Processor utilization & efficient memory use
  - Parallel scalability
- **Important to understand interactions**
  - Algorithm – code – compiler – libraries – hardware
- **Performance is not portable!**

Performance analysis

To optimise code we must know *what* is taking the time

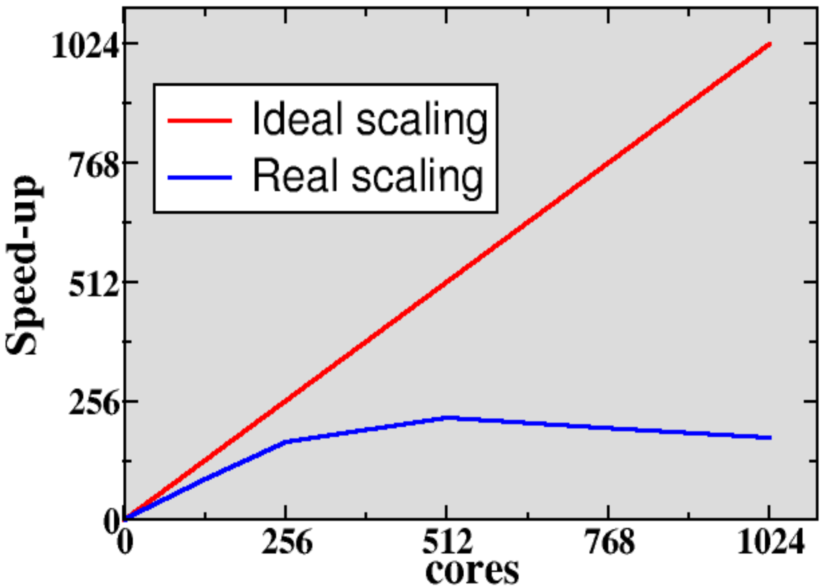


Not going to touch the source code?

- **Find the *compiler* and its *compiler flags* that yield the best performance**
- **Employ *tuned libraries* wherever possible**
- **Find suitable settings for *environment parameters***
- **Mind the *I/O***
  - Do not checkpoint too often
  - Do not ask for the output you do not need

Why does scaling end?

- **Amount of data per process small - computation takes little time compared to communication**
- **Amdahl's law in general**
  - E.g., single-writer or stderr I/O
- **Load imbalance**
- **Communication that scales badly with N_{proc}**
  - E.g., all-to-all collectives
- **Congestion on network – too many messages or lots of data**



Application timing

- **Most basic information: total wall clock time**
  - Built-in timers in the program (e.g. MPI_Wtime)
  - System commands (e.g. time) or batch system statistics
- **Built-in timers can provide also more fine-grained information**
  - Have to be inserted by hand
  - Typically, no information about hardware related issues e.g. cache utilization
  - Information about load imbalance and communication statistics of parallel program is difficult to obtain

Performance analysis tools

- **Instrumentation of code**
  - Adding special measurement code to binary
    - Special commands, compiler/linker wrappers
    - Automatic or manual
  - Normally all routines do not need to be measured
- **Measurement: running the instrumented binary**
  - Profile: sum of events over time
  - Trace: sequence of events over time
- **Analysis**
  - Text based analysis reports
  - Visualization

Sampling

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

Event Tracing

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

Guided tracing = trace only program parts that consume a significant portion of the total time

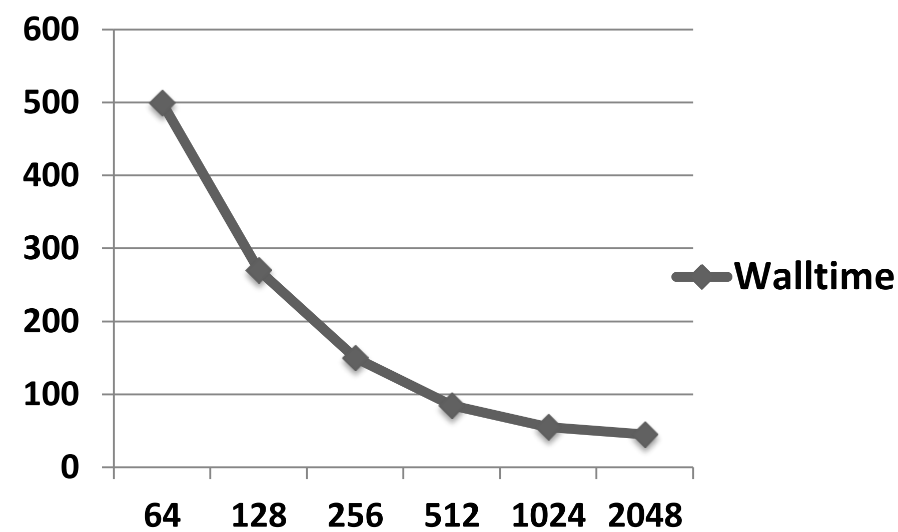
In Cray Performance Analysis Toolkit this is referred to as ”automatic profiling analysis ”(APA)

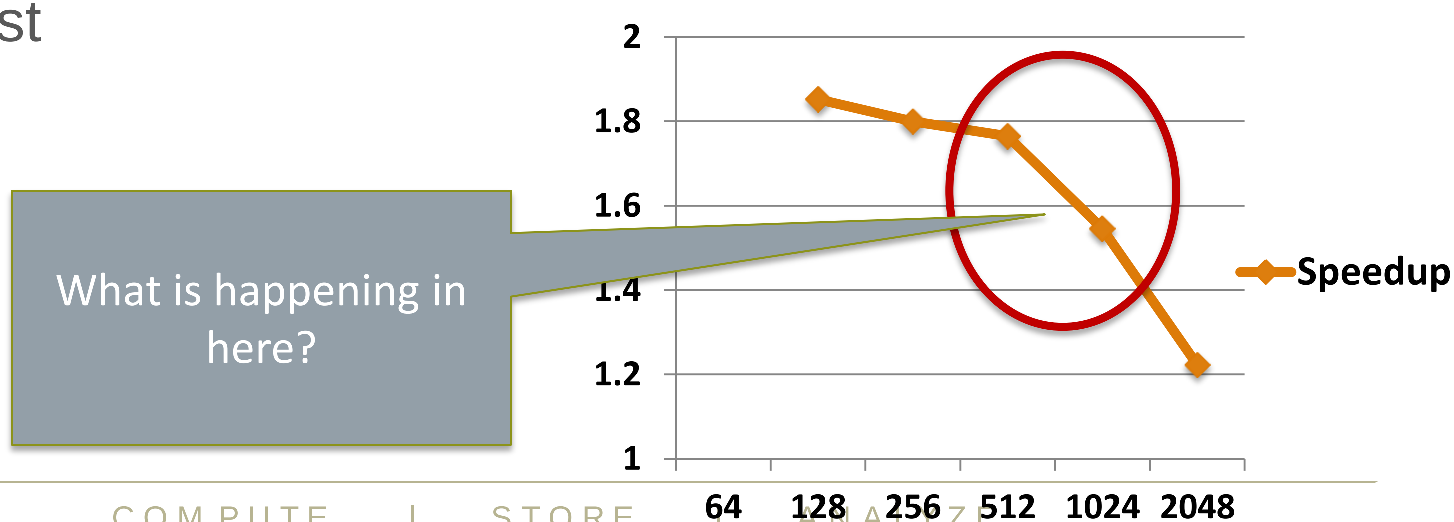
Step 1: Choose a test problem

- **The dataset used in the analysis should**
  - Make scientific sense, i.e. resemble the intended use of the code
  - Be large enough for getting a good view on scalability
  - Be runable in a reasonable time
  - For instance, with simulation codes almost a full-blown model but run only for a few time steps
- **Should be run long enough that initialization/finalization stages are not exaggerated**
  - Alternatively, we can exclude them during the analysis

Step 2: Measure Scalability

- **Run the uninstrumented code with different core counts and see where the parallel scaling stops**
- **Usually we look at strong scaling**
  - Also weak scaling is definitely of interest





Step 3: Instrument the application

- **Obtain first a sampling profile to find which user functions should be traced**
  - With a large/complex software, one should not trace them all: it causes excessive overhead
- **Make an instrumented exe with tracing time-consuming user functions plus e.g. MPI, I/O and library (BLAS, FFT,...) calls**
- **Execute and record the first analysis with**
  - The core count where the scalability is still ok
  - The core count where the scalability has ended

  and identify the largest differences between these profiles
- **CrayPAT tools have an Automatic Profile Analysis (APA) mode to handle this process:**

Steps to Collect Performance Data

- **Access performance tools software**
  - `module load perftools`
- **Build application keeping .o files (CCE: `-h keepfiles`)**
  - `make clean`
  - `make`
- **Instrument application for automatic profiling analysis**
  - `pat_build -O apa a.out`
  - You should get an instrumented program `a.out+pat`
  - This has been instrumented for sampling
- **Run application to get top time consuming routines**
  - `aprun … a.out+pat` *(or* `qsub <pat script>`*)*
  - You should get one or more *.xf performance files

Steps to Collecting Performance Data (2)

- **Run pat_report, on the .xf file or the directory**
  - `pat_report -o <report> <xf file>`
  - `pat_report -o <report> <xf directory>`
  - Generates text report and an .apa instrumentation file
    - We'll discuss pat_report in more detail later

- **At this stage the report gives us useful information and we should get sample hits in time-consuming code sections**
- **We can go further on to tracing**

- **We use the .apa file to re-instrument binary for tracing**
  - the most important functions have been identified for tracing

- **We can inspect and edit the .apa file at this point**
  - if we want to tweak the choice of routines to be traced

APA File Example

```
#  You can edit this file, if desired, and use it
#   to reinstrument the program for  tracing like this:
#
#           pat_build -O standard.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-
5.0.0.2-
Oapa.512.quad.cores.seal.090405.1154.mpi.pat_rt_exp=default.pat_rt_hwpc=no
ne.14999.xf.xf.apa
#
#  These suggested trace options are based on data from:
#
#
/home/users/malice/pat/Runs/Runs.seal.pat5001.2009Apr04/./pat.quad/homme/s
tandard.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2-
Oapa.512.quad.cores.seal.090405.1154.mpi.pat_rt_exp=default.pat_rt_hwpc=no
ne.14999.xf.xf.cdb
# ----------------------------------------------------------------------

#       HWPC group to collect by default.

  -Drtenv=PAT_RT_HWPC=1  # Summary with TLB metrics.

# ----------------------------------------------------------------------

#       Libraries to trace.

  -g mpi

# ----------------------------------------------------------------------

#       User-defined functions to trace, sorted by % of samples.

#       The way these functions are filtered can be controlled with
#       pat_report options (values used for this file are shown):
#
#       -s apa_max_count=200    No more than 200 functions are listed.
#       -s apa_min_size=800     Commented out if text size < 800 bytes.
#       -s apa_min_pct=1        Commented out if it had < 1% of samples.
#       -s apa_max_cum_pct=90   Commented out after cumulative 90%.

#       Local functions are listed for completeness, but cannot be traced.

  -w  # Enable tracing of user-defined functions.
      # Note: -u should NOT be specified as an additional option.
```

```
# 31.29%  38517 bytes
         -T prim_advance_mod_preq_advance_exp_

# 15.07%  14158 bytes
         -T prim_si_mod_prim_diffusion_

#  9.76%  5474 bytes
         -T derivative_mod_gradient_str_nonstag_

. . .

#  2.95%  3067 bytes
         -T forcing_mod_apply_forcing_

#  2.93%  118585 bytes
         -T column_model_mod_applycolumnmodel_

#  Functions below this point account for less than 10% of samples.

#  0.66%  4575 bytes
#         -T bndry_mod_bndry_exchangev_thsave_time_

#  0.10%  46797 bytes
#         -T baroclinic_inst_mod_binst_init_state_

#  0.04%  62214 bytes
#         -T prim_state_mod_prim_printstate_

. . .
#  0.00%  118 bytes
#         -T time_mod_timelevel_update_

# ----------------------------------------------------------------------

  -o preqx.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2.x+apa
# New instrumented program.

/.AUTO/cray/css.pe_tools/malice/craypat/build/pat/2009Apr03/2.1.56HD/amd64
/homme/pgi/pat-5.0.0.2/homme/2005Dec08/build.Linux/preqx.cray-xt.PE-
2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2.x  # Original program.
```

Effectively a series of command line arguments to pat_build

Generating Event Traced Profile from APA

- **Re-instrument application for further analysis**
  - `pat_build -O <apa file>`
  - creates new binary: `<exe>+apa`

- **Re-run application**
  - `aprun … a.out+apa` (or `qsub <apa script>`)
  - This generates a new set of .xf data files

- **Generate new text report and visualization file (.ap2)**
  - `pat_report -o <report> <xf file>`
  - `pat_report -o <report> <xf directory>`

- **View report in text and/or with Cray Apprentice2**
  - `app2 <ap2 file>`
  - We'll cover this in more detail later

Analysing Data with pat_report

Using pat_report

- **pat_report converts raw profiling data into a profile**
  - Combines .xf data with binary
    - Instrumented binary must still exist when data is converted!
  - Produces a text report and an .ap2 file
  - .ap2 file can be used for further pat_report calls or display in GUI

- **Generates a text report of performance results**
  - Data laid out in tables
  - Many options for sorting, slicing or dicing data in the tables.
    - `pat_report -O <table option> *.ap2`
    - `pat_report -O help` (list of available profiles)
  - Volume and type of information depends upon sampling vs tracing.

Advantages of the .ap2 file

- **.ap2 file is a self contained compressed performance file**
  - Normally it is about 5 times smaller than the .xf file
  - Contains the information needed from the application binary
  - Can be reused
- **Independent of the perftools version used to generate it**
  - The xf files are very version-dependent
- **It is the only input format accepted by Cray Apprentice²**

- **Once you have the .ap2 file, you can delete:**
  - the .xf files
  - the instrumented binary

Files Generated and the Naming Convention

File Suffix	Description
a.out+pat	Program instrumented for data collection
a.out…s.xf	Raw data from sampling experiment available after application execution
a.out…t.xf	Raw data from trace (summarized or full) experiment available after application execution
a.out….ap2	Processed data, generated by pat_report, contains application symbol information
a.out…s.apa	Automatic profiling analysis template, generated by pat_report (based on pat_build -O apa experiment)
a.out+apa	Program instrumented using .apa file
MPICH_RANK_ORDER.Custom	Rank reorder file generated by pat_report from automatic grid detection an reorder suggestions

Job Execution Information

```
CrayPat/X:  Version 5.2.3.8078 Revision 8078 (xf 8063)  08/25/11 …

Number of PEs (MPI ranks):   16

Numbers of PEs per Node:     16

Numbers of Threads per PE:    1

Number of Cores per Socket:  12

Execution start time:  Thu Aug 25 14:16:51 201

System type and speed:  x86_64 2000 MHz

Current path to data file:
  /lus/scratch/heidi/ted_swim/mpi-openmp/run/swim+pat+27472-34t.ap2

Notes for table 1:
…
```

Sampling Output (Table 1)

```
Notes for table 1:

...

Table 1:  Profile by Function

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

 100.0% |  775 |     -- |      -- |Total
|-------------------------------------------
|  94.2% |  730 |     -- |      -- |USER
||------------------------------------------
||  43.4% |  336 |   8.75 |    2.6% |mlwxyz_
||  16.1% |  125 |   6.28 |    4.9% |half_
||   8.0% |   62 |   6.25 |    9.5% |full_
||   6.8% |   53 |   1.88 |    3.5% |artv_
||   4.9% |   38 |   1.34 |    3.6% |bnd_
||   3.6% |   28 |   2.00 |    6.9% |currenf_
||   2.2% |   17 |   1.50 |    8.6% |bndsf_
||   1.7% |   13 |   1.97 |   13.5% |model_
||   1.4% |   11 |   1.53 |   12.2% |cfl_
||   1.3% |   10 |   0.75 |    7.0% |currenh_
||   1.0% |    8 |   5.28 |   41.9% |bndbo_
||   1.0% |    8 |   8.28 |   53.4% |bndto_
||==========================================
|   5.4% |   42 |     -- |      -- |MPI
||------------------------------------------
||   1.9% |   15 |   4.62 |   23.9% |mpi_sendrecv_
||   1.8% |   14 |  16.53 |   55.0% |mpi_bcast_
||   1.7% |   13 |   5.66 |   30.7% |mpi_barrier_
|===========================================
```

pat_report: Flat Profile

```
Table 1:  Profile by Function Group and Function

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

 100.0% | 104.593634 |       -- |     -- | 22649 |Total
|-----------------------------------------------------------
|  71.0% |  74.230520 |       -- |     -- | 10473 |MPI
||----------------------------------------------------------
||  69.7% |  72.905208 | 0.508369 |   0.7% |   125 |mpi_allreduce_
||   1.0% |   1.050931 | 0.030042 |   2.8% |    94 |mpi_alltoall_
||==========================================================
|  25.3% |  26.514029 |       -- |     -- |    73 |USER
||----------------------------------------------------------
||  16.7% |  17.461110 | 0.329532 |   1.9% |    23 |selfgravity_
||   7.7% |   8.078474 | 0.114913 |   1.4% |    48 |ffte4_
||==========================================================
|   2.5% |   2.659429 |       -- |     -- |   435 |MPI_SYNC
||----------------------------------------------------------
||   2.1% |   2.207467 | 0.768347 |  26.2% |   172 |mpi_barrier_(sync)
||==========================================================
|   1.1% |   1.188998 |       -- |     -- | 11608 |HEAP
||----------------------------------------------------------
||   1.1% |   1.166707 | 0.142473 |  11.1% |  5235 |free
|=============================================================
```

pat_report: Message Stats by Caller

```
Table 4:  MPI Message Stats by Caller

    MPI Msg |MPI Msg |  MsgSz |  4KB<= |Function
      Bytes |  Count |   <16B |  MsgSz | Caller
            |        |  Count |  <64KB |  PE[mmm]
            |        |        |  Count |

 15138076.0 | 4099.4 |  411.6 | 3687.8 |Total
|------------------------------------------------
| 15138028.0 | 4093.4 |  405.6 | 3687.8 |MPI_ISEND
||-----------------------------------------------
||  8080500.0 | 2062.5 |   93.8 | 1968.8 |calc2_
3|            |        |        |        | MAIN_
||||---------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  8208000.0 | 2000.0 |     -- | 2000.0 |pe.9
4|||  6160000.0 | 2000.0 |  500.0 | 1500.0 |pe.15
||||=============================================
||  6285250.0 | 1656.2 |  125.0 | 1531.2 |calc1_
3|            |        |        |        | MAIN_
||||---------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.3
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.5
||||=============================================
. . .
```

Some important options to pat_report -O

callers	Profile by Function and Callers
callers+hwpc	Profile by Function and Callers
callers+src	Profile by Function and Callers, with Line Numbers
callers+src+hwpc	Profile by Function and Callers, with Line Numbers
calltree	Function Calltree View
heap_hiwater	Heap Stats during Main Program
hwpc	Program HW Performance Counter Data
load_balance_program+hwpc	Load Balance across PEs
load_balance_sm	Load Balance with MPI Sent Message Stats
loop_times	Loop Stats by Function (from -hprofile_generate)
loops	Loop Stats by Inclusive Time (from -hprofile_generate)
mpi_callers	MPI Message Stats by Caller
profile	Profile by Function Group and Function
profile+src+hwpc	Profile by Group, Function, and Line
samp_profile	Profile by Function
samp_profile+hwpc	Profile by Function
samp_profile+src	Profile by Group, Function, and Line

- For a full list see: `pat_report -O help`

Loop Statistics

- **Just like adding automatic tracing at the function level, we can add tracing to individual loops.**

- **Helps identify candidates for parallelization:**
  - Loop timings approximate how much work exists within a loop
  - Trip counts can be used to understand parallelism potential
    - useful if considering porting to manycore

- **Only available with CCE:**
  - Requires compiler add additional features into the code.
  - Should be done as separate profiling experiment
    - compiler optimizations are restricted with this feature

- **Loop statistics reported by default in `pat_report` table**

Collecting Loop Statistics

- **Load PrgEnv-cray module (default on most systems)**
- **Load perftools module**

- **Compile AND link with CCE flag: -h profile_generate**

- **Instrument binary for tracing**
  - All user functions: pat_build -u my_program
  - Or even no user functions: pat_build -w my_program
    - This is sufficient for loop-level profiling of all loops!
  - Or use an existing apa file.

- **Run the application**
- **Create report with loop statistics**
  - pat_report <xf file> > <report file>

Default Report Table 2

```
Notes for table 2:
  Table option:
    -O loops
  …
  The Function value for each data item is the avg of the PE values.
    (To specify different aggregations, see:  pat_help report options s1)

  This table shows only lines with Loop Incl Time / Total > 0.0095.
    (To set thresholds to zero, specify:  -T)

Loop instrumentation can interfere with optimizations, so time
  reported here may not reflect time in a fully optimized program.

  Loop stats can safely be used in the compiler directives:
   !PGO$       loop_info est_trips(Avg) min_trips(Min) max_trips(Max)
   #pragma pgo loop_info est_trips(Avg) min_trips(Min) max_trips(Max)

  Explanation of Loop Notes (P=1 is highest priority, P=0 is lowest):
   novec (P=0.5): Loop not vectorized (see compiler messages for reason).
   sunwind (P=1): Loop could be vectorized and unwound.
   vector (P=0.1): Already a vector loop.
```

Profile guided optimization feedback for compiler: see man pgo

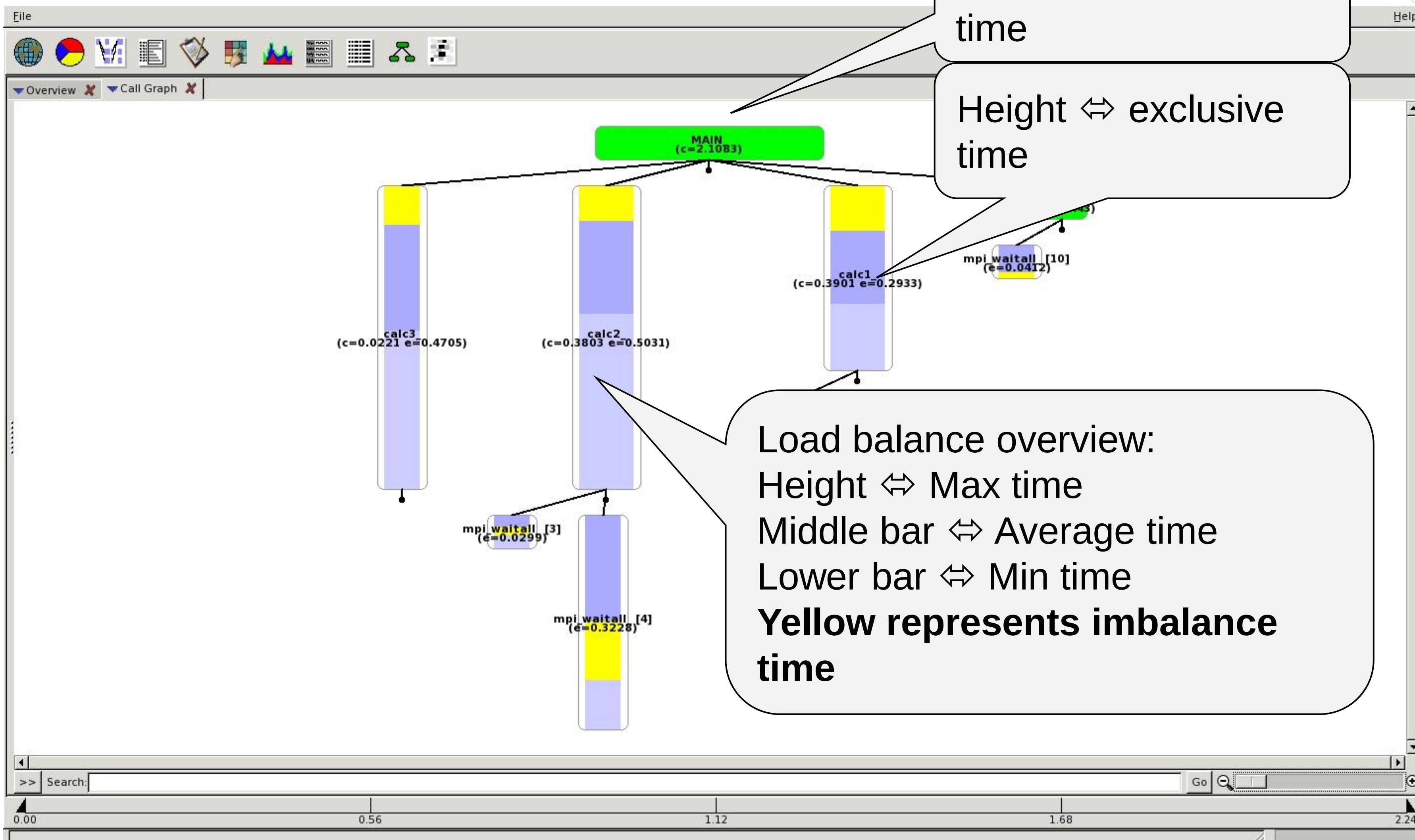
Default Report Table 2

```
Table 2:  Loop Stats from -hprofile_generate

   Loop |Loop Incl |Loop Incl |  Loop |  Loop |    Loop |Function=/.LOOP\.
   Incl |     Time |   Time / |   Hit | Trips |   Notes | PE='HIDE'
 Time / |          |      Hit |       |   Avg |         |
  Total |          |          |       |       |         |

|-------------------------------------------------------------------------
|   24.6% | 0.057045 | 0.000570 |   100 |  64.1 |   novec |calc2_.LOOP.0.li.614
|   24.0% | 0.055725 | 0.000009 |  6413 | 512.0 |  vector |calc2_.LOOP.1.li.615
|   18.9% | 0.043875 | 0.000439 |   100 |  64.1 |   novec |calc1_.LOOP.0.li.442
|   18.3% | 0.042549 | 0.000007 |  6413 | 512.0 |  vector |calc1_.LOOP.1.li.443
|   17.1% | 0.039822 | 0.000406 |    98 |  64.1 |   novec |calc3_.LOOP.0.li.787
|   16.7% | 0.038883 | 0.000006 |  6284 | 512.0 |  vector |calc3_.LOOP.1.li.788
|    9.7% | 0.022493 | 0.000230 |    98 | 512.0 |  vector |calc3_.LOOP.2.li.805
|    4.2% | 0.009837 | 0.000098 |   100 | 512.0 |  vector |calc2_.LOOP.2.li.640
|=========================================================================
```

Step 4: Assessing the big picture

- **Profile = Where the most of the time is really being spent?**
  - See also the call-tree view
  - Ignore (from the optimization point-of-view) user routines with less than 5% of the execution time
- **Why does the scaling end: the major differences in these two profiles?**
  - Has the MPI fraction ’blown up’ in the larger run?
  - Have the load imbalances increased dramatically?
  - Has something else emerged to the profile?
  - Has the time spent for user routines decreased as it should (i.e. do they scale independently)?

Example with CrayPAT



Width ⇔ inclusive time

Height ⇔ exclusive time

Load balance overview:
Height ⇔ Max time
Middle bar ⇔ Average time
Lower bar ⇔ Min time
Yellow represents imbalance time

Step 5: Analyze load imbalance

- **What is causing the imbalance?**
- **Computation**
  - Tasks call for computational kernels (user functions, BLAS routines,...) for varying times and/or the execution time varies depending on the input/caller
- **Communication**
  - Large MPI_Sync times
- **I/O**
  - One or more tasks are performing I/O and the others are just waiting for them in order to proceed

Example with CrayPAT

Step 6: Analyze communication

- **What communication pattern is dominating the true time spent for MPI (excluding the sync times)**
  - Refer to the call-tree view on Apprentice2 and the “MPI Message Stats” tables in the text reports produced by pat_report
- **Note that the analysis tools may report load imbalances as ”real” communication**
  - Put an MPI_Barrier before the suspicious routine - load imbalance will aggregate into it in when then analysis is rerun
- **How does the message-size profile look like?**
  - Are there a lot of small messages?

Example with CrayPAT report (message stats)

```
Table 4:  MPI Message Stats by Caller

    MPI Msg |MPI Msg |  MsgSz |  4KB<= |Function
      Bytes |  Count |   <16B |  MsgSz | Caller
            |        |  Count |  <64KB |  PE[mmm]
            |        |        |  Count |

 15138076.0 | 4099.4 |  411.6 | 3687.8 |Total
|-----------------------------------------------
| 15138028.0 | 4093.4 |  405.6 | 3687.8 |MPI_ISEND
||----------------------------------------------
||  8080500.0 | 2062.5 |   93.8 | 1968.8 |calc2_
3|            |        |        |        | MAIN_
||||--------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  8208000.0 | 2000.0 |     -- | 2000.0 |pe.9
4|||  6160000.0 | 2000.0 |  500.0 | 1500.0 |pe.15
||||============================================
||  6285250.0 | 1656.2 |  125.0 | 1531.2 |calc1_
3|            |        |        |        | MAIN_
||||--------------------------------------------
4|||  8216000.0 | 3000.0 | 1000.0 | 2000.0 |pe.0
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.3
4|||  6156000.0 | 1500.0 |     -- | 1500.0 |pe.5
||||============================================
. . .
```

Step 7: Analyze I/O

- **Trace POSIX I/O calls (fwrite, fread, write, read,...)**
- **How much I/O?**
  - Do the I/O operations take a significant amount of time?
- **Are some of the load imbalances or communication bottlenecks in fact due to I/O?**
  - Synchronous single writer
  - Insert MPI_Barriers to investigate this

Step 8: Find single-core hotspots

- **Remember: pay attention only to user routines that consume significant portion of the total time**
- **View the key hardware counters, for example**
  - L1 and L2 cache metrics
  - use of vector (SSE/AVX) instructions
  - Computational intensity (= ratio of floating point ops / memory accesses)
- **CrayPAT has mechanisms for finding "the" hotspot in a routine (e.g. in case the routine contains several and/or long loops)**
  - CrayPAT API
    - Possibility to give labels to "PAT regions"
  - Loop statistics (works only with Cray compiler)
    - Compile & link with CCE using -h profile_generate
    - pat_report will generate loop statistics if the flag is being enabled

Example with CrayPAT

```
==========================================================================
USER / conj_grad_.LOOPS
------------------------------------------------------------------------------
  Time%                                                 59.5%
  Time                                              73.010370 secs
  Imb. Time                                          3.563452 secs
  Imb. Time%                                             4.7%
  Calls                          1.383 /sec             101.0 calls
  PERF_COUNT_HW_CACHE_L1D:ACCESS                 183909710385
  PERF_COUNT_HW_CACHE_L1D:
    PREFETCH                                       7706793512
  PERF_COUNT_HW_CACHE_L1D:MISS                    21336476999
  ...
  SIMD_FP_256:PACKED_DOUBLE                        1961227352
  User time (approx)            73.042 secs    189983282830 cycles  100.0% Time
  CPU_CLK                        3.454GHz
  HW FP Ops / User time         969.844M/sec     70839736685 ops    9.3%peak(DP)
  Total DP ops                  969.844M/sec     70839736685 ops
  Computational intensity          0.37 ops/cycle       0.33 ops/ref
  MFLOPS (aggregate)          124140.04M/sec
  TLB utilization               1058.97 refs/miss      2.068 avg uses
  D1 cache hit,miss ratios         90.0% hits          10.0% misses
  D1 cache utilization (misses)    9.98 refs/miss      1.248 avg hits
  D2 cache hit,miss ratio          17.5% hits          82.5% misses
  D1+D2 cache hit,miss ratio       91.7% hits           8.3% misses
  D1+D2 cache utilization         12.10 refs/miss      1.512 avg hits
  D2 to D1 bandwidth          18350.176MB/sec  140549334558 bytes
  Average Time per Call                              0.722875 secs
```

Flat profile data

HW counter values

Derived metrics

Example with CrayPAT

```
Table 2:  Loop Stats from -hprofile_generate

   Loop |Loop Incl |Loop Incl |  Loop |  Loop |    Loop |Function=/.LOOP\.
   Incl |     Time |   Time / |   Hit | Trips |   Notes | PE='HIDE'
 Time / |          |      Hit |       |   Avg |         |
  Total |          |          |       |       |         |

|-------------------------------------------------------------------------
|  24.6% | 0.057045 | 0.000570 |   100 |  64.1 |   novec |calc2_.LOOP.0.li.614
|  24.0% | 0.055725 | 0.000009 |  6413 | 512.0 |  vector |calc2_.LOOP.1.li.615
|  18.9% | 0.043875 | 0.000439 |   100 |  64.1 |   novec |calc1_.LOOP.0.li.442
|  18.3% | 0.042549 | 0.000007 |  6413 | 512.0 |  vector |calc1_.LOOP.1.li.443
|  17.1% | 0.039822 | 0.000406 |    98 |  64.1 |   novec |calc3_.LOOP.0.li.787
|  16.7% | 0.038883 | 0.000006 |  6284 | 512.0 |  vector |calc3_.LOOP.1.li.788
|   9.7% | 0.022493 | 0.000230 |    98 | 512.0 |  vector |calc3_.LOOP.2.li.805
|   4.2% | 0.009837 | 0.000098 |   100 | 512.0 |  vector |calc2_.LOOP.2.li.640
|=========================================================================
```

The Golden Rules of profiling:

- **Profile your code**
  - The compiler/runtime will not do all the optimisation for you.
- **Profile your code yourself**
  - Don't believe what anyone tells you. They're wrong.
- **Profile on the hardware you want to run on**
  - Don't profile on your laptop if you plan to run on a Cray system
- **Profile your code running the full-sized problem**
  - The profile will almost certainly be qualitatively different for a test case.
- **Keep profiling your code as you optimize**
  - Concentrate your efforts on the thing that slows your code down.
  - This will change as you optimise.
  - So keep on profiling.

Performance Optimization: Improving Parallel Scalability

Scalability bottlenecks

- **Review the performance measurements (between the two runs)**
- **Case: user routines scaling but MPI time blowing up**
  - Issue: Not enough to compute in a domain
    - Weak scaling could still continue
  - Issue: Expensive (all-to-all) collectives
  - Issue: Communication increasing as a function of tasks
- **Case: MPI_Sync times increasing**
  - Issue: Load imbalance
    - Tasks not having a balanced role in communication?
    - Tasks not having a balanced role in computation?
    - Synchronous (single-writer) I/O or stderr I/O?

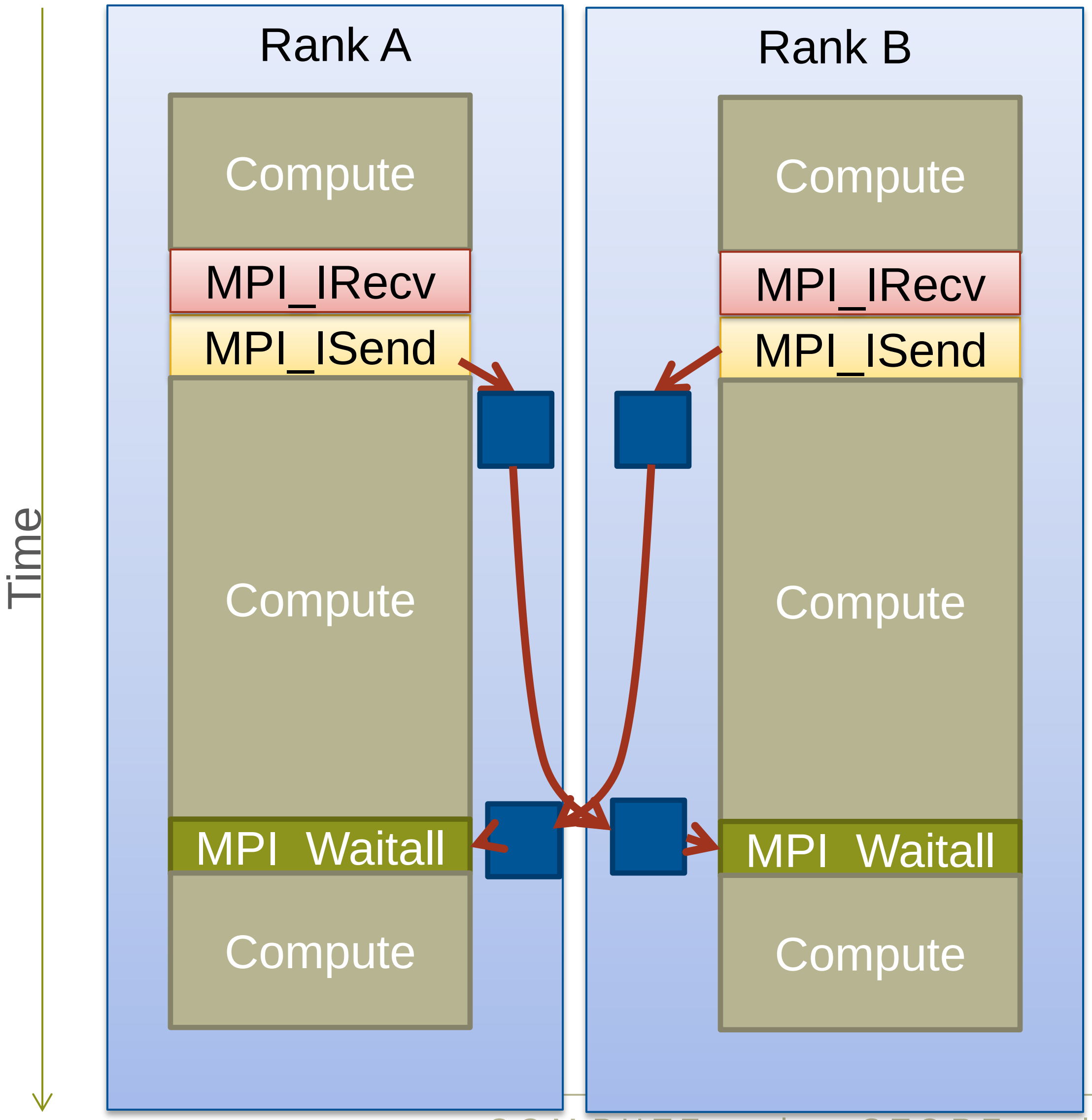
Issue: Load imbalances

- **Identify the cause**
  - How to fix I/O related imbalance will be addressed later
- **Unfortunately algorithmic, decomposition and data structure revisions are needed to fix load balance issues**
  - Dynamic load balancing schemas
  - MPMD style programming
  - There may be still something we can try without code re-design
- **Consider hybridization (mixing OpenMP with MPI)**
  - Reduces the number of MPI tasks - less pressure for load balance
  - May be doable with very little effort
    - Just plug omp parallel do’s/for’s to the most intensive loops
  - However, in many cases large portions of the code has to be hybridized to outperform flat MPI

Issue: Point-to-point communication consuming time

- **Message transfer time ∝ latency + message size / bandwidth**
  - Latency: Startup for message handling
  - Bandwidth: Network BW / number of messages using the same link
- **Reduce latency by aggregating multiple small messages if possible**
  - Do not pack manually but use MPI's user-defined datatypes
    - Always use the least general datatype constructor possible
- **Bandwidth and latency depend on the used protocol**
  - *Eager* or *rendezvous*
    - Latency *and* bandwidth higher in rendezvous
  - Rendezvous messages usually do not allow for overlap of computation and communication (see the extra slides for explanation), even when using non-blocking communication routines
  - The platform will select the protocol basing on the message size, these limits can be adjusted

EAGER potentially allows overlapping



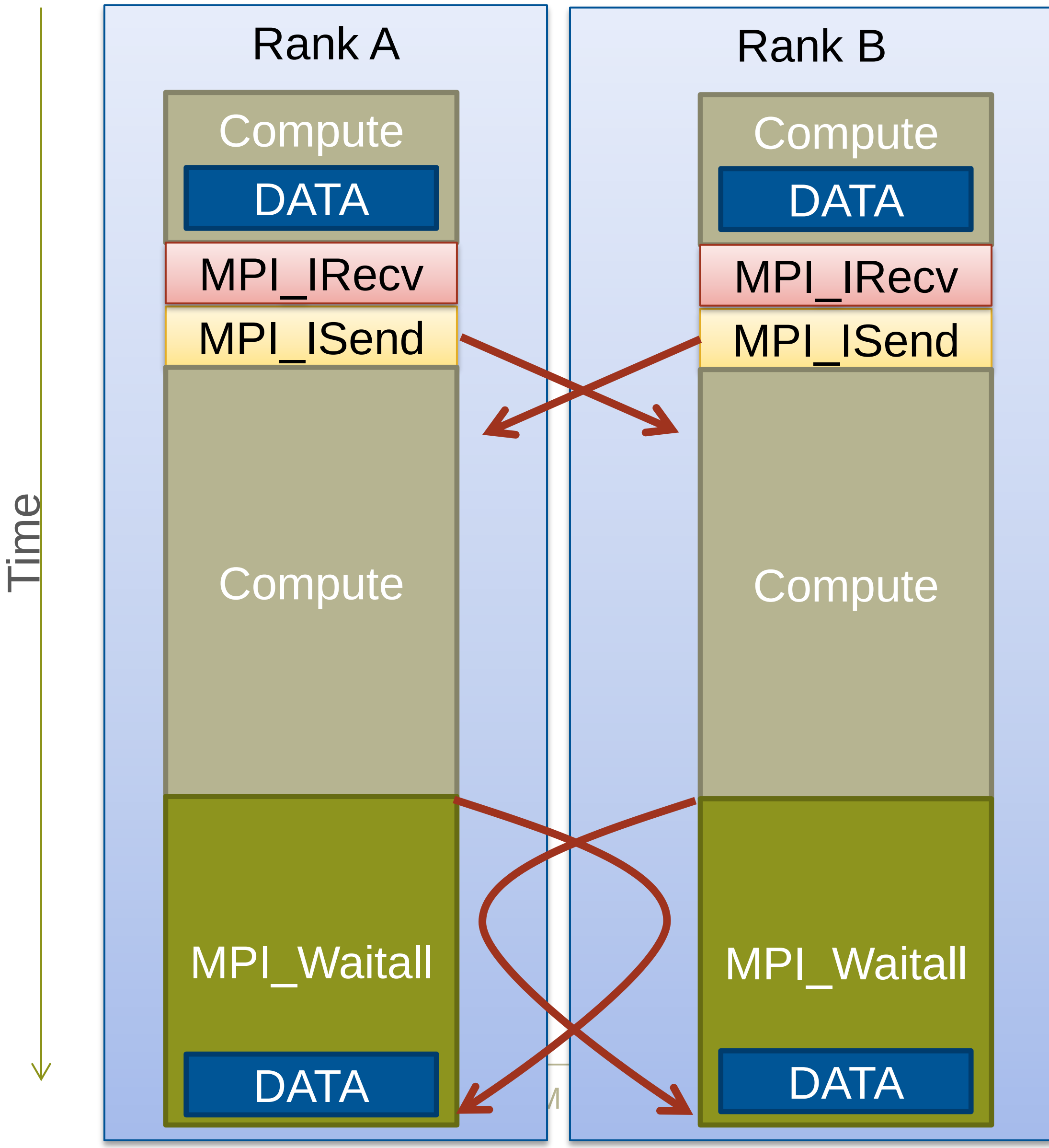
Data is pushed into an empty buffer(s) on the remote processor.

Data is copied from the buffer into the real receive destination when the wait or waitall is called.

Involves an extra memcopy, but much greater opportunity for overlap of computation and communication.

Further info

RENDEZVOUS does not usually overlap



With rendezvous data transfer is often only occurs during the Wait or Waitall statement.

When the message arrives at the destination, the host CPU is busy doing computation, so is unable to do any message matching.

Control only returns to the library when MPI_Waitall occurs and does not return until all data is transferred.

There has been no overlap of computation and communication.

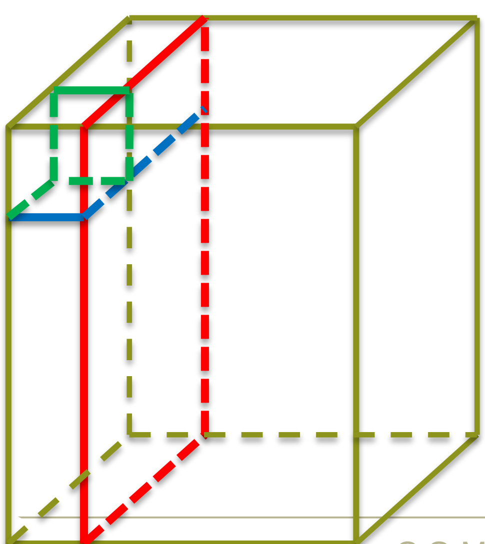
Further info

Issue: Point-to-point communication consuming time

- **One way to improve performance is to send more messages using the eager protocol**
  - This can be done by raising the value of the eager threshold, by setting environment variable:
    `export MPICH_GNI_MAX_EAGER_MSG_SIZE=X`
  - Values are in bytes, the default is 8192 bytes. Maximum size is 131072 bytes (128KB)
- **Try to post MPI_Irecv calls before the MPI_Isend calls to avoid unnecessary buffer copies**
- **On Cray XE & XC: Asynchronous Progress Engine**
  - Progresses also rendezvous messages on the background by launching an extra helper thread to each MPI task
  - Consult ‘man mpi’ and there the variable
    `MPICH_NEMESIS_ASYNC_PROGRESS`

Issue: Point-to-point communication consuming time

- **Minimize the data to be communicated by carefully designing the partitioning of data and computation**
  - Example: domain decomposition of a 3D grid (n x n x n) with halos to be communicated, cyclic boundaries



1D decomposition ("slabs"):
communication $\propto n^2 * w * 2$

2D decomposition ("tubes"):
communication $\propto n^2 * p^{-1/2} * w * 4$

3D decomposition ("cubes"):
communication $\propto n^2 * p^{-2/3} * w * 6$

w = halo width
p = number of MPI tasks

Issue: Expensive collectives

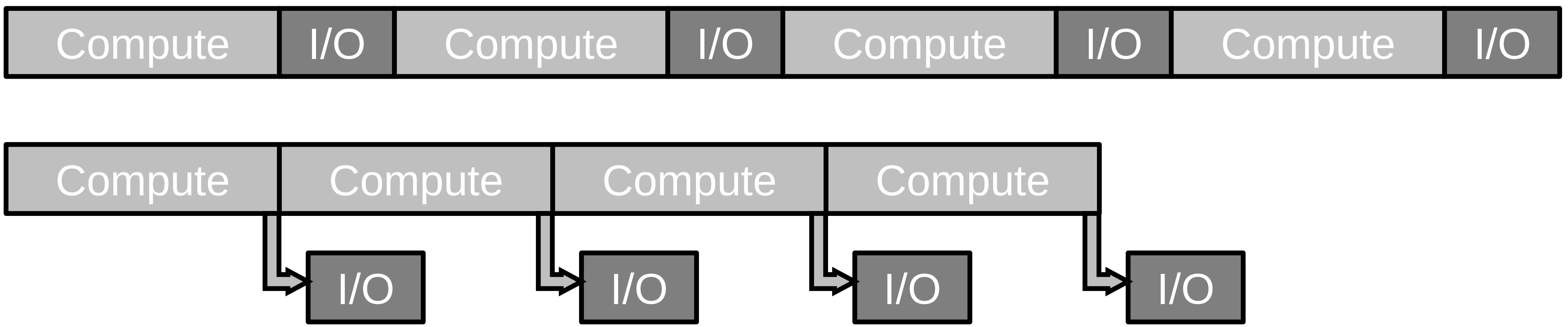
- **Reducing MPI tasks by mixing OpenMP is likely to help**
- **See if every all-to-all collective operation needs to be all-to-all rather than one-to-all or all-to-one**
  - Often encountered case: convergence checking
- **See if you can live with the basic version of a routine instead of a vector version (`MPI_Alltoallv` etc)**
  - May be faster even if some tasks would be receiving data never referenced
- **The MPI 3.0 introduces non-blocking collectives (MPI_Ialltoall,...)**
  - Allow for overlapping collectives with other operations, e.g. computation, I/O or other communication
  - Are faster (at least on Cray) than the blocking corresponds even without the overlap, and replacement is trivial

Issue: Expensive collectives

- **Hand-written RDMA collectives may outperform those of the MPI library**
  - Fortran coarrays, Unified Parallel C, MPI one-sided communication
- **On Cray XE and XC systems, the sc. DMAPP collectives will (usually significantly) improve the performance of the expensive collectives**
  - Enabled by the variable:
    `export  MPICH_USE_DMAPP_COLL=1`
  - Can be used selectively, e.g.
    `export  MPICH_USE_DMAPP_COLL=mpi_allreduce`
  - Features some restrictions and requires explicit linking with the corresponding library and using the huge pages; consult 'man mpi'

Issue: Performance bottlenecks due to I/O

- **Parallelize your I/O !**
  - MPI I/O, I/O libraries (HDF5, NetCDF), hand-written schmas,...
  - Without parallelization, I/O will be a scalability bottleneck in every application
- **Try to hide I/O (asynchronous I/O)**



  - Available on MPI I/O (MPI_File_iwrite/read(_at))
  - One can also add dedicated ”I/O servers” into code: separate MPI tasks or dedicating one I/O core per node on a hybrid MPI+OpenMP application

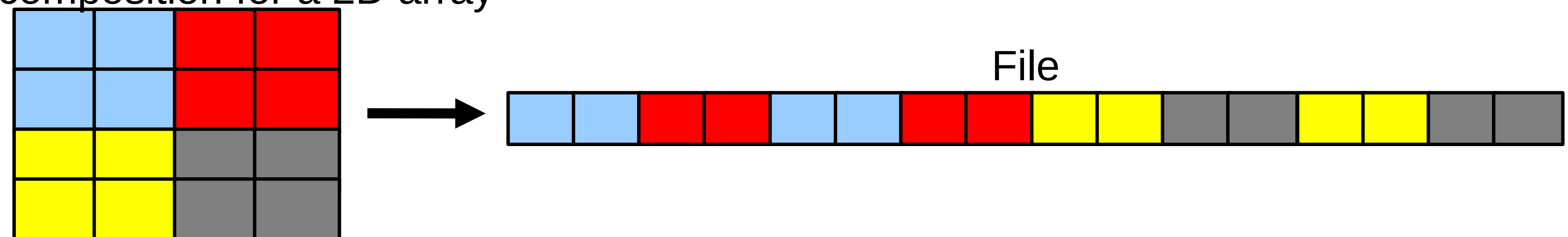
Issue: Performance bottlenecks due to I/O

- **Tune filesystem (Lustre) parameters**
  - Lustre stripe counts & sizes, see ”man lfs”
  - Rule of thumb:
    - # files > # OSTs => Set stripe_count=1
      You will reduce the lustre contention and OST file locking this way and gain performance
    - #files==1 => Set stripe_count=#OSTs
      Assuming you have more than 1 I/O client
    - #files<#OSTs => Select stripe_count so that you use all OSTs
- **Use I/O buffering for all sequential I/O**
  - IOBUF is a library that intercepts standard I/O (stdio) and enables asynchronous caching and prefetching of sequential file access
  - No need to modify the source code but just
    - Load the module iobuf
    - Rebuild your application

Issue: Performance bottlenecks due to I/O

- **When using MPI-I/O and making non-contiguous writes/reads (e.g. multi-dimensional arrays), always define file views with suitable user-defined types and use collective I/O**
  - Performance can be 100x compared to individual I/O

Decomposition for a 2D array




```
call mpi_type_create_subarray(2, sizes, subsizes, starts, mpi_integer, &
     mpi_order_c, filetype, err)
call mpi_type_commit(filetype)
disp = 0
call mpi_file_set_view(file, disp, mpi_integer, filetype, 'native', &
     mpi_info_null, err)
call mpi_file_write_all(file, buf, count, mpi_integer, status, err)
```

Concluding remarks

- **Apply the scientific method to performance engineering: make hypotheses and measurements!**
- **Scaling up is the most important consideration in HPC**
- **Possible approaches for alleviating typical scalability bottlenecks**
  - Find the optimal decomposition & rank placement
  - Overlap computation & communication - use non-blocking communication operations for p2p and collective communication both!
  - Make more messages 'eager' and/or employ the Asynchronous Progress Engine (on Cray)
  - Hybridize (=mix MPI+OpenMP) the code to improve load balance and alleviate bottleneck collectives
- **Mind your I/O!**
  - Use parallel I/O
  - Tune filesystem parameters