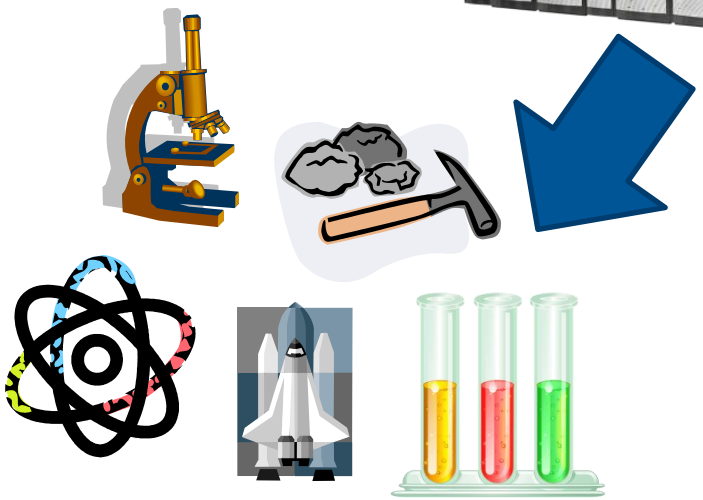
Performance Engineering

Pekka Manninen, Ph.D.
Cray Inc.
manninen@cray.com

Performance engineering

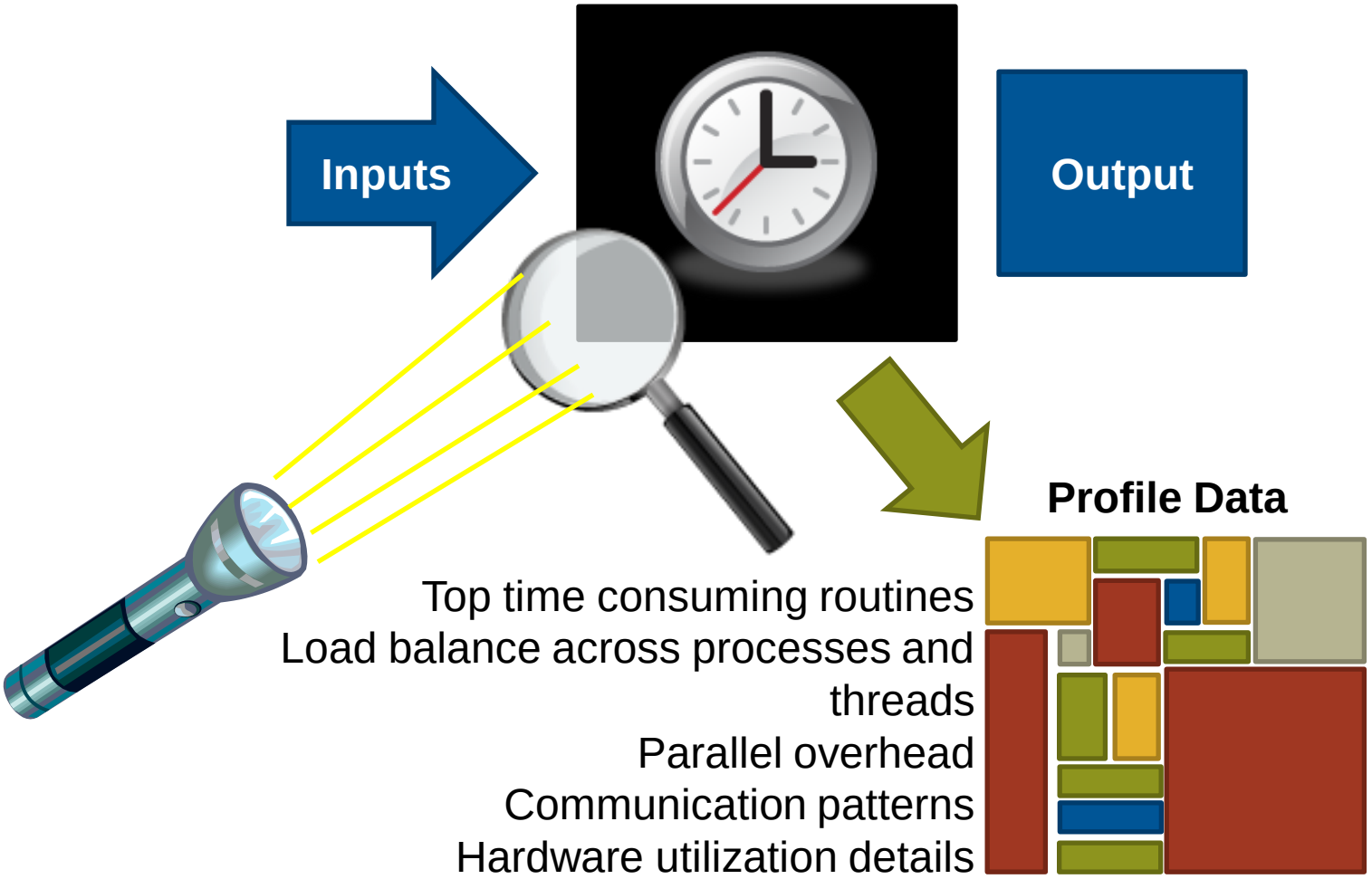
We want to get the most science and engineering through a supercomputing system as possible

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



Performance analysis

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



Performance Engineering module overview

	Tuesday
11.15-12.00	Introduction to performance engineering
12.00-13.15	Lunch
13.15-14.00	Application performance analysis
14.00-14.15	Break
14.15-15.00	Lab session
15.00-15.15	Coffee break
15.15-17.00	Lab session

	Wednesday
8.30-9.00	Interim summary, Q&A
11.15-12.00	Improving parallel scalability
12.00-13.15	Lunch
13.15-15.00	Lab session

Part I: Introduction to performance engineering

- About code optimization in general
- Not going to touch the source code?
- Data locality
- Why does scaling end?
- Application optimization flow chart

Code optimization

- **Obvious benefits**
  - Better throughput => more science
  - Cheaper than new hardware
  - Save energy, compute quota etc.
- **..and some non-obvious ones**
  - Collaboration opportunities
  - Potential for cross-disciplinary research
  - Deeper understanding of application
- **Several trends making code optimization even more important**
  - More and more cores
  - CPU’s vector units getting wider
  - The gap between CPU and memory speed ever increasing
  - Datasets growing rapidly but disk I/O performance lags behind

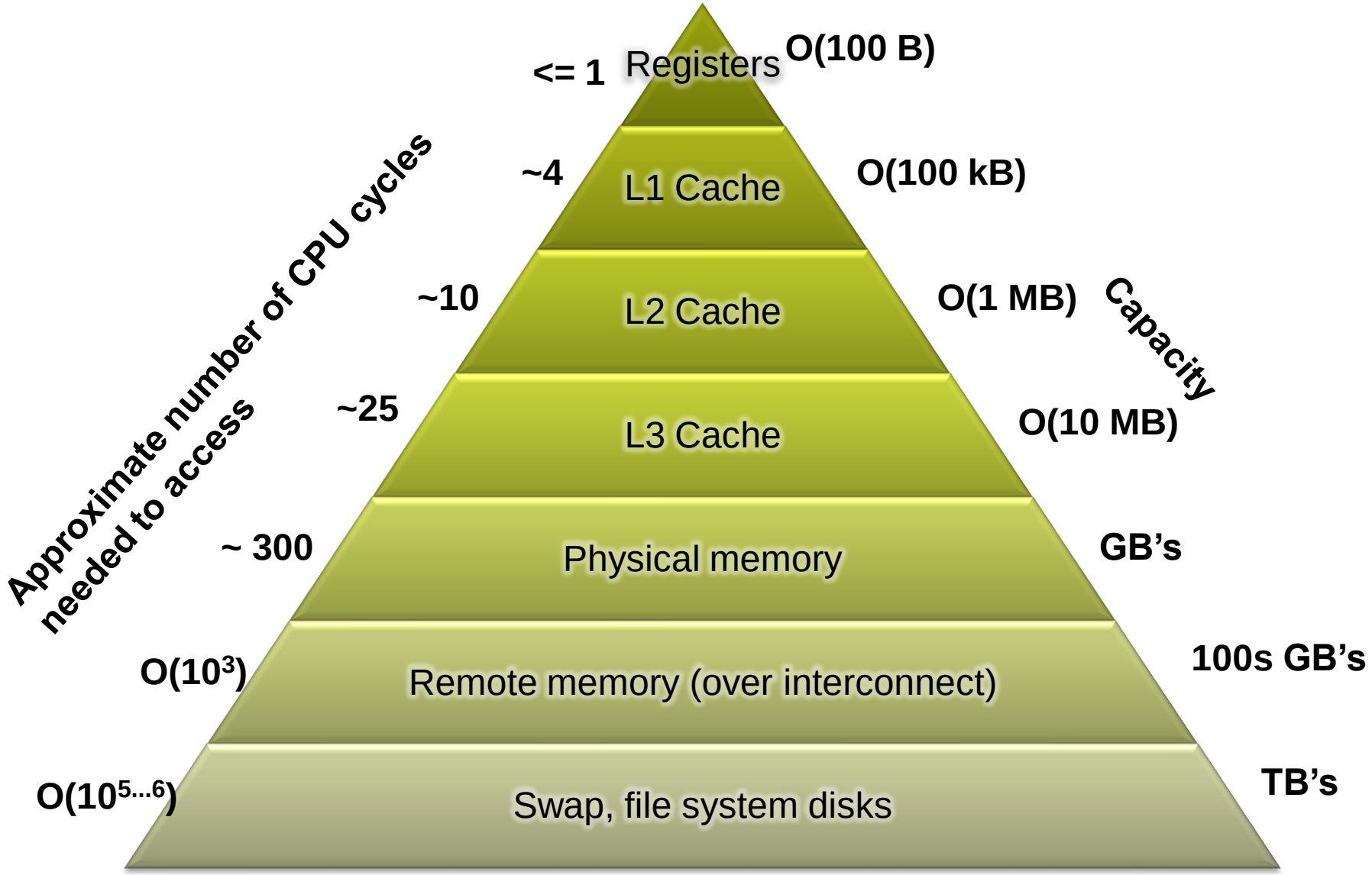
Code optimization

- **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!**

Not going to touch the source code?

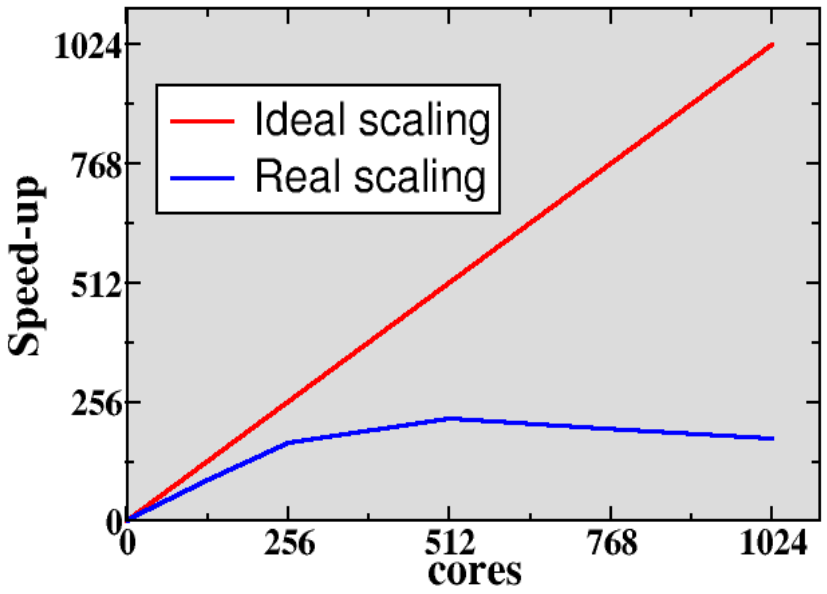
- **Find the *compiler* and its *compiler flags* that yield the best performance**
- **Employ *tuned libraries* wherever possible**
- **Find suitable settings for *environment parameters***
- **Mind the *I/O***
  - Do not checkpoint too often
  - Do not ask for the output you do not need

Keep your friends close and data even closer

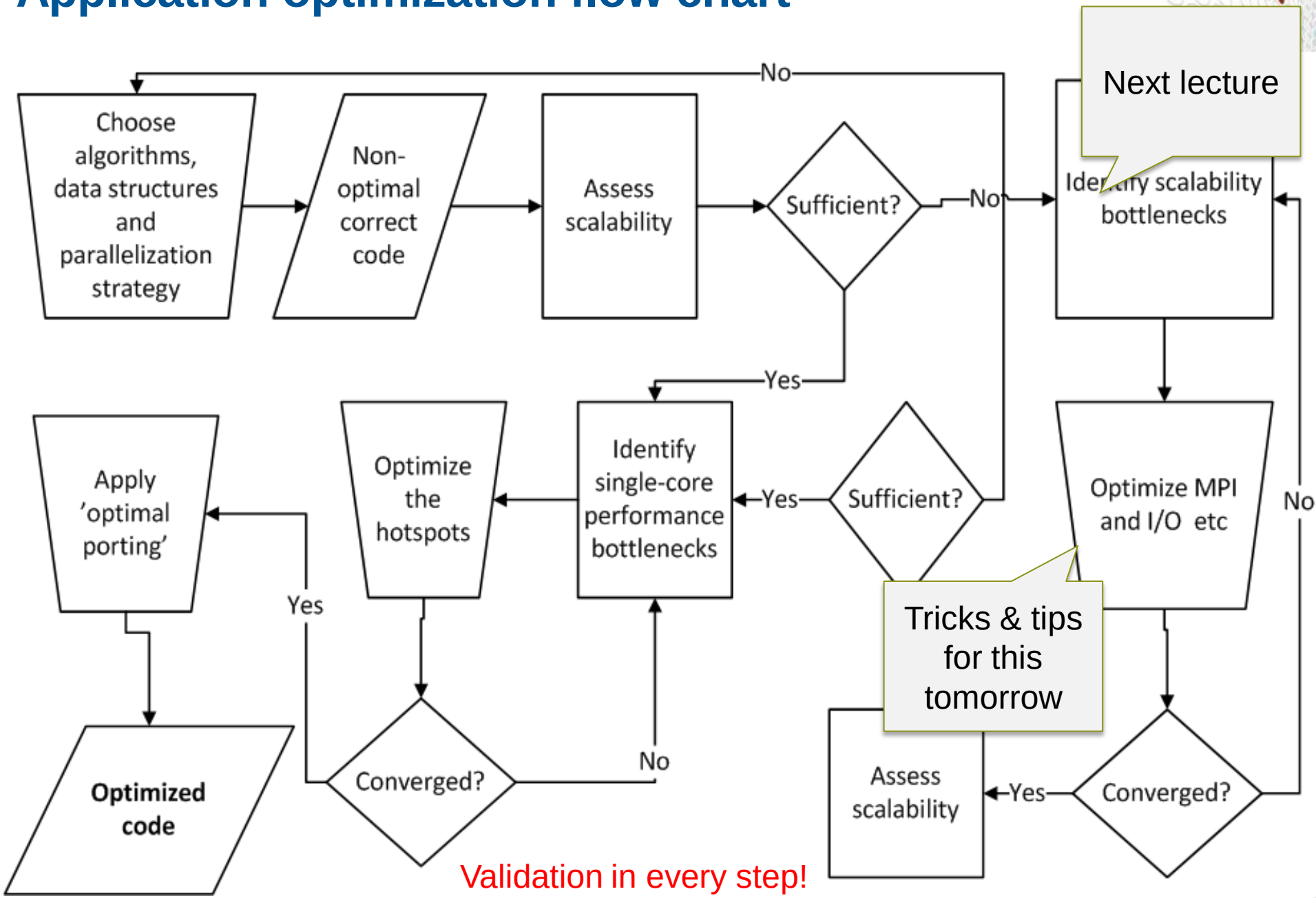


Why does scaling end?

- 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 optimization flow chart



Part II: Application performance analysis

- Tools for performance analysis
- Eight-stage procedure for identifying performance bottlenecks
- Example: Cray Performance Analysis Toolkit

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

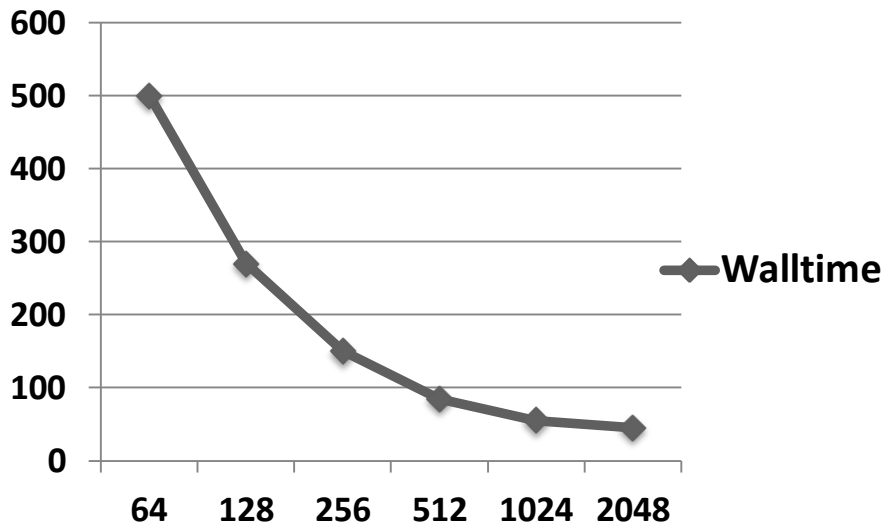
In Cray Performance Analysis Toolkit this is referred to as ”automatic profiling analysis ”(APA)

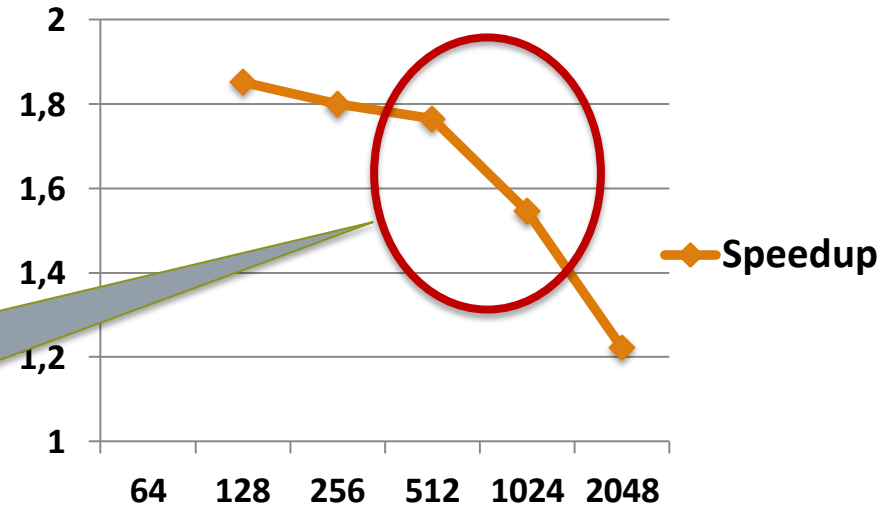
Step 1: Choose a test problem

- **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





What is happening in here?

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

Example with CrayPAT (1/2)

- **Load performance tools software**

```
module load perftools
```

- **Re-build application (keep .o files)**

```
make clean
make
```

- **Instrument application for automatic profiling analysis**
  - You should get an instrumented program `a.out+pat`

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

- **Run the instrumented application (...+pat) to get top time consuming routines**
  - You should get a performance file ("`<sdatafile>.xf`") or multiple files in a directory `<sdatadir>`

Example with CrayPAT (2/2)

- **Generate text report and an `.apa` instrumentation file**

```
pat_report [<sdatafile>.xf | <sdatadir>]
```

  - Inspect the `.apa` file and sampling report whether additional instrumentation is needed
    - See especially sites “Libraries to trace” and “HWPC group to collect”

- **Instrument application for further analysis (a.out+apa)**

```
pat_build -O <apafile>.apa
```

- **Run application (...+apa)**
- **Generate text report and visualization file (.ap2)**

```
pat_report -o my_text_report.txt [<datafile>.xf | <datadir>]
```

- **View report in text and/or with Cray Apprentice^2**

```
app2 <datafile>.ap2
```

Some important options to pat_report -O

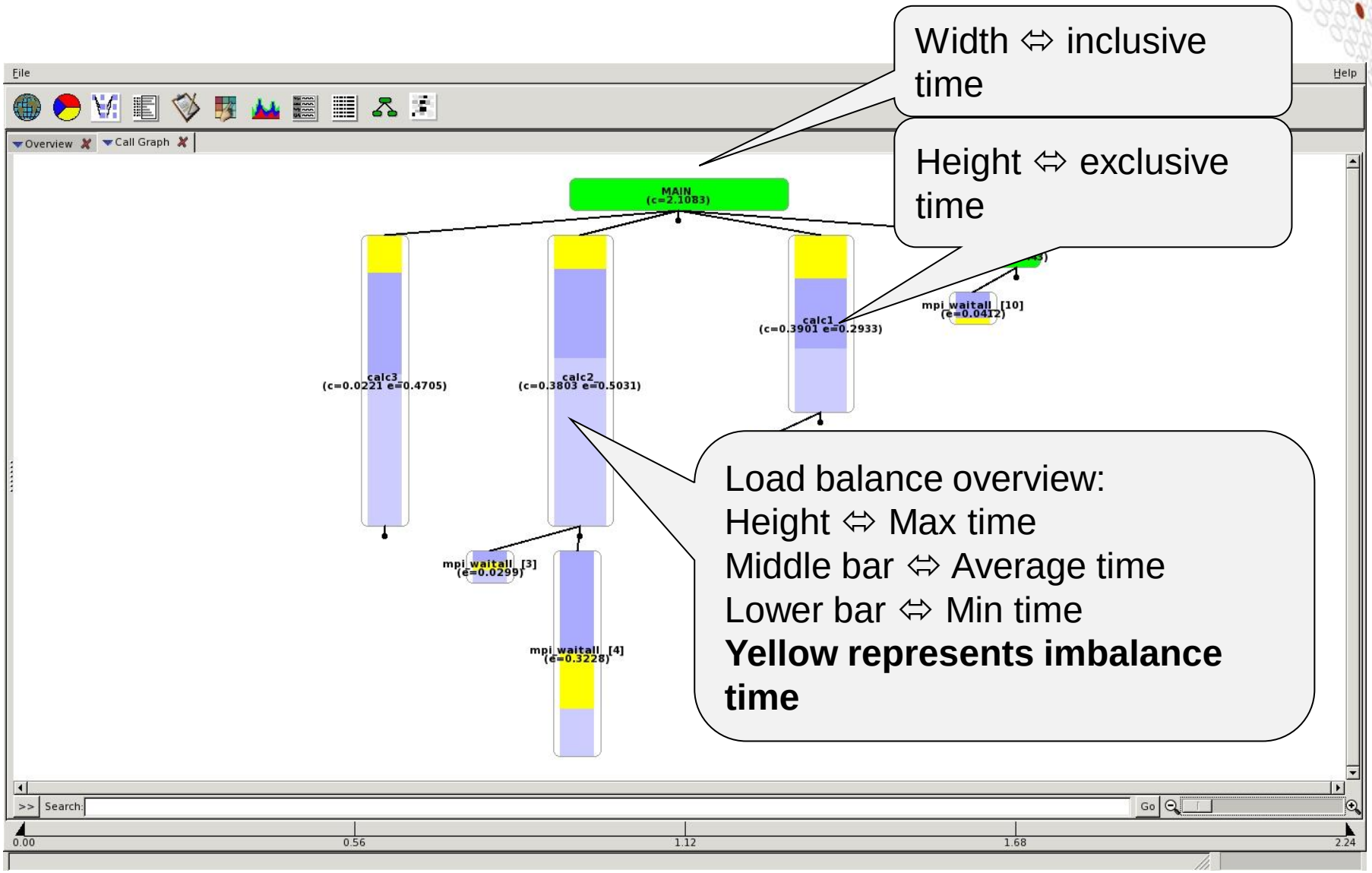
```
callers                     Profile by Function and Callers
callers+hwpc                Profile by Function and Callers
callers+src                 Profile by Function and Callers, with Line Numbers
callers+src+hwpc            Profile by Function and Callers, with Line Numbers
calltree                    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
```

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



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

```
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  1405449334558 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
|=========================================================================
```

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?

Web resources

- CrayPAT documentation
  http://docs.cray.com
- Scalasca
  http://www.scalasca.org/
- Paraver
  http://www.bsc.es/computer-sciences/performance-tools/paraver
- Tau performance analysis utility
  http://www.cs.uoregon.edu/Research/tau

Lab session: Performance analysis

- **The *Game of Life* (GoL) is a cellular automaton devised by John Horton Conway, read**
  http://en.wikipedia.org/wiki/Conway's_Game_of_Life
- **A parallel (MPI) implementation of the GoL is provided in GoL_mpi (.f90 or .c)**
  - Compile and run the software

```
make mpi
aprun –n 4 ./gol 100 500 500
```

Develop a 500x500 board
for 100 iterations
Run through the batch
job scheduler

- **<u>To do</u>: Find out the reason(s) why the default version of the GoL code does not scale**
  - Indeed it *should* scale, it is a simple domain decomposition with thin halos
  - You will find other implementations from the makefile as follows:
    - make nonb - replaces MPI_Sendrecv by nonblocking operations
    - make pario - parallelizes the disk I/O
    - make hyb - hybrid MPI+OpenMP
- **Alternatively (even preferably) you can analyse your own application!**

Part III: Improving parallel scalability

- Load imbalance due to communication
- Many messages and/or large amount of data
- Expensive collectives
- I/O bottlenecks

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: Load imbalances

- **Changing rank placement (on Cray and other MPI libraries based on MPICH2)**

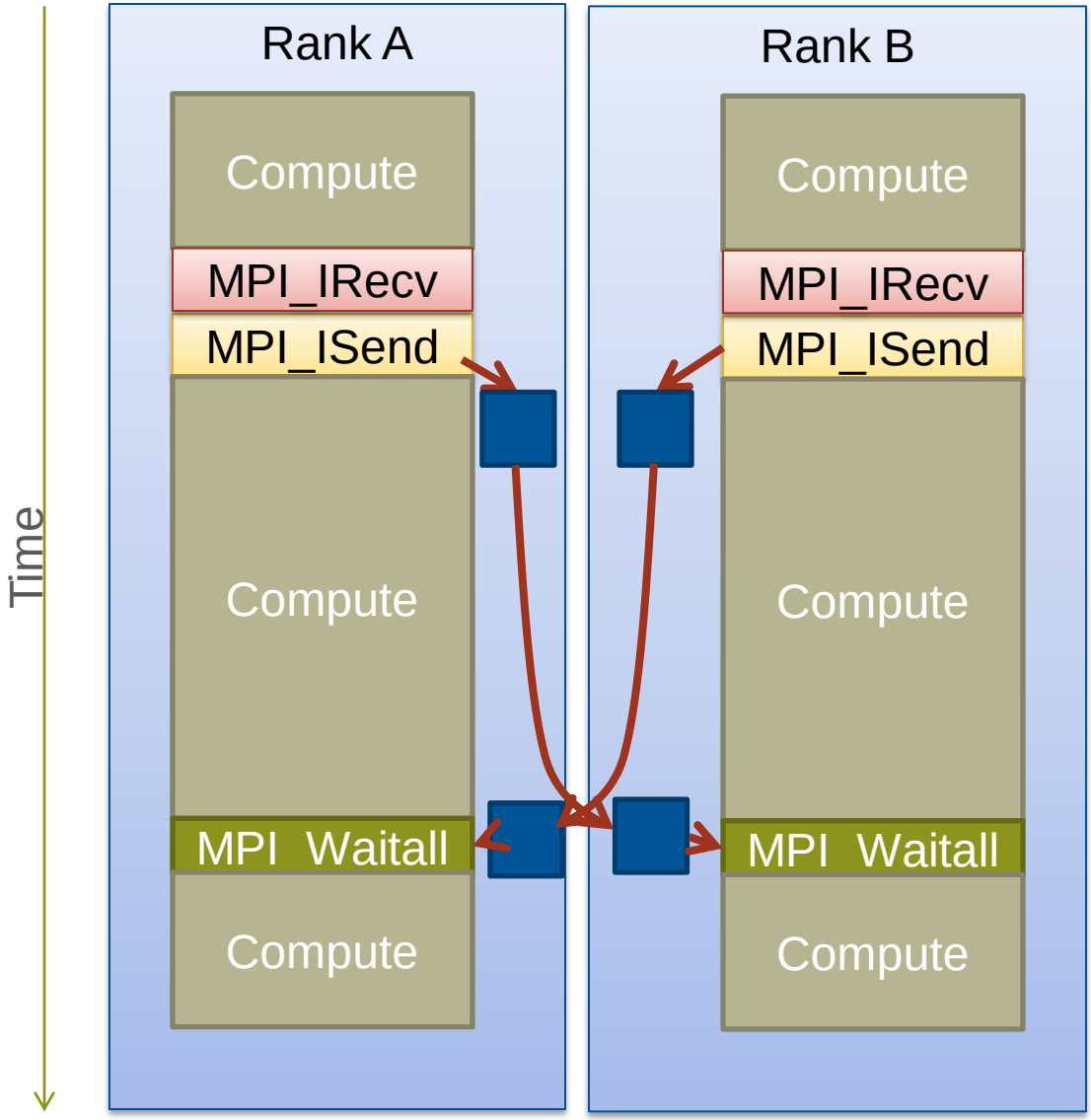
  `export MPICH_RANK_REORDER_METHOD=N`

  - These are the different values (N) that you can set it to:
    - 0: Round-robin placement
    - 1: (DEFAULT) SMP-style placement
    - 2: Folded rank placement
    - 3: Custom ordering. The ordering is specified in a file named MPICH_RANK_ORDER.
  - So easy to experiment with that it should be tested with every application!
  - 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
    - This output can then be copied or written into a file named MPICH_RANK_ORDER and used with `MPICH_RANK_REORDER_METHOD=3`
  - CrayPAT is also able to make suggestions for optimal rank placement: `pat_report -O mpi_rank_order datafile.xf`

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
- **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



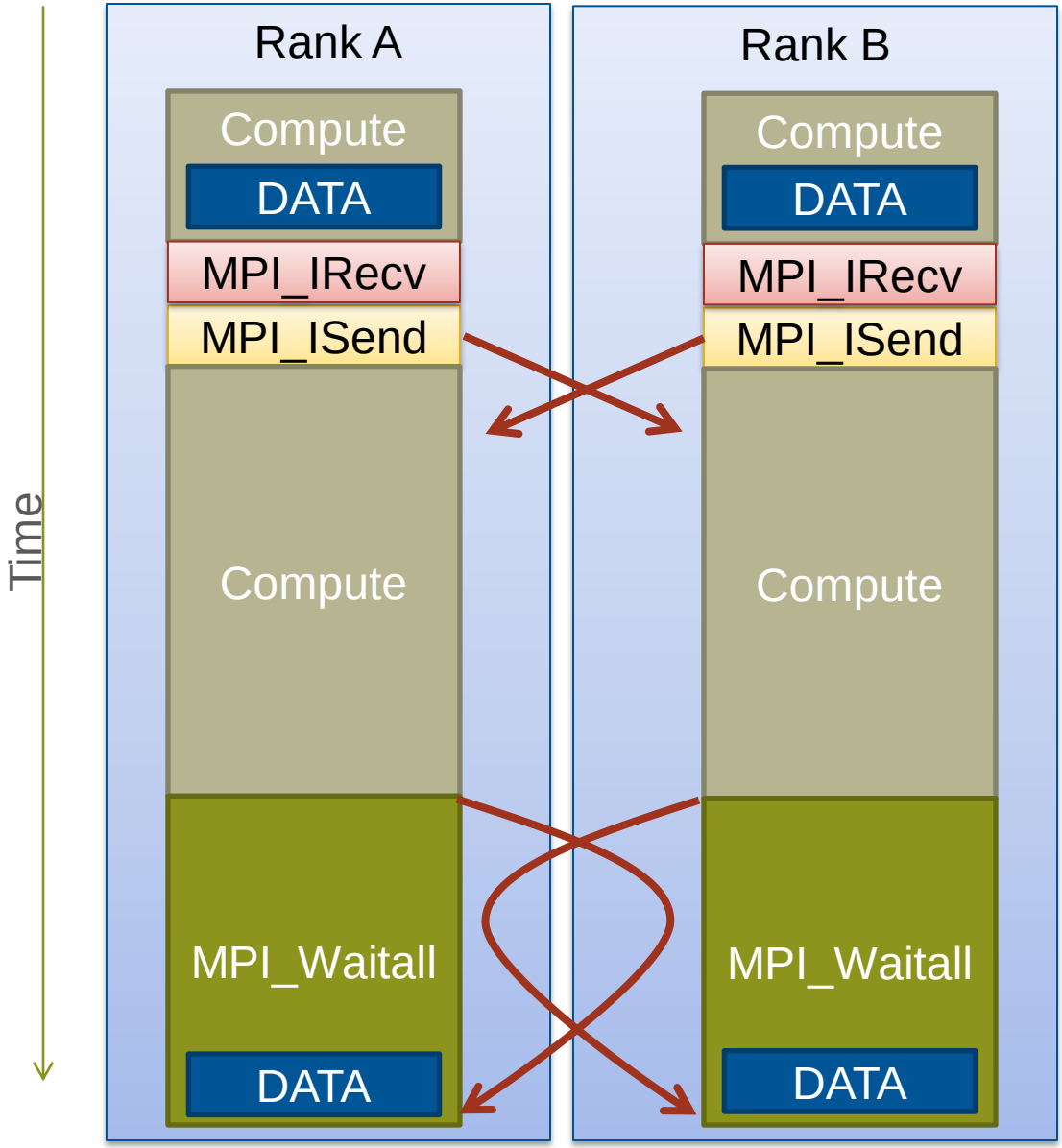
Data is pushed into an empty buffer(s) on the remote processor.

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.

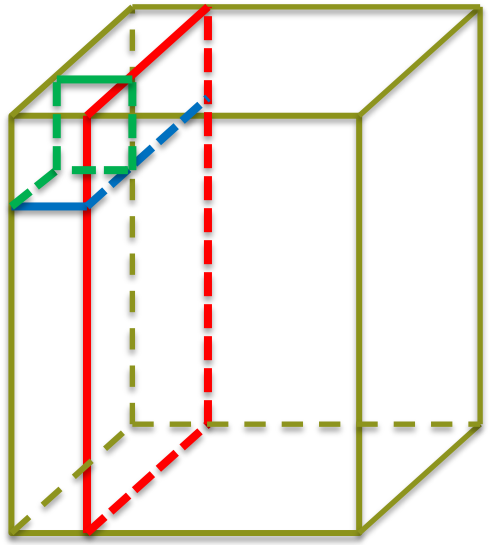
Further info

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

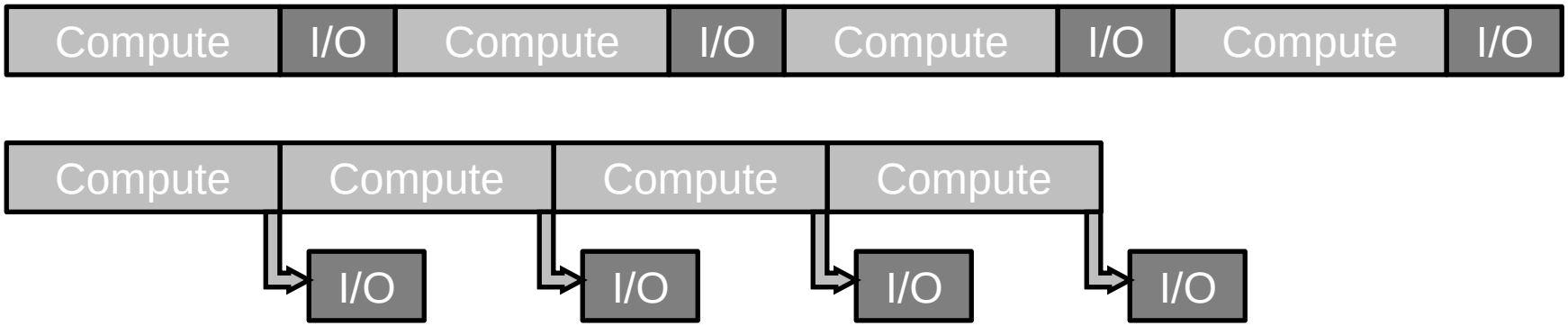
- **Reducing MPI tasks by mixing OpenMP is likely to help**
- **See if every all-to-all collective operation needs to be all-to-all rather than one-to-all or all-to-one**
  - Often encountered case: convergence checking
- **See if you can live with the basic version of a routine instead of a vector version (`MPI_Alltoallv` etc)**
  - May be faster even if some tasks would be receiving dummy data
- **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 sc. 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)**



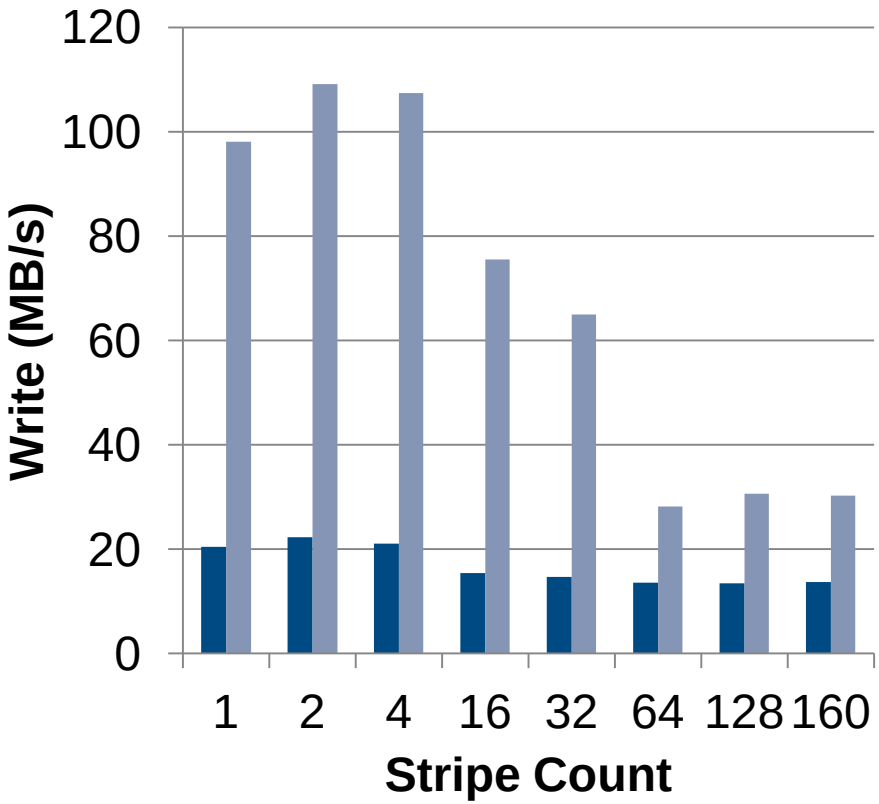
  - Available on MPI I/O (MPI_File_iwrite/read(_at))
  - One can also add dedicated ”I/O servers” into code: separate MPI tasks or dedicating one I/O core per node on a hybrid MPI+OpenMP application

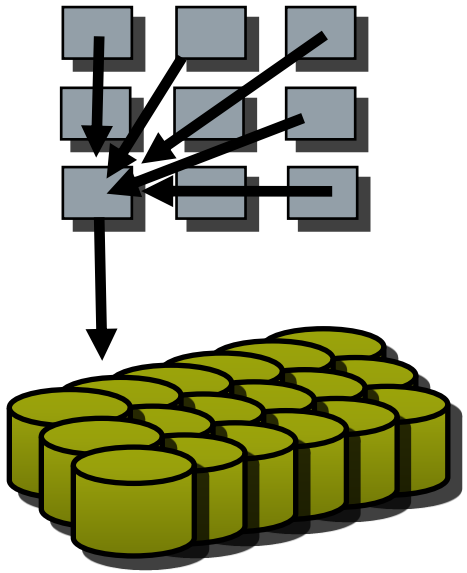
Issue: Performance bottlenecks due to I/O

- **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

Case study: Single-writer I/O

- **32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size**
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance

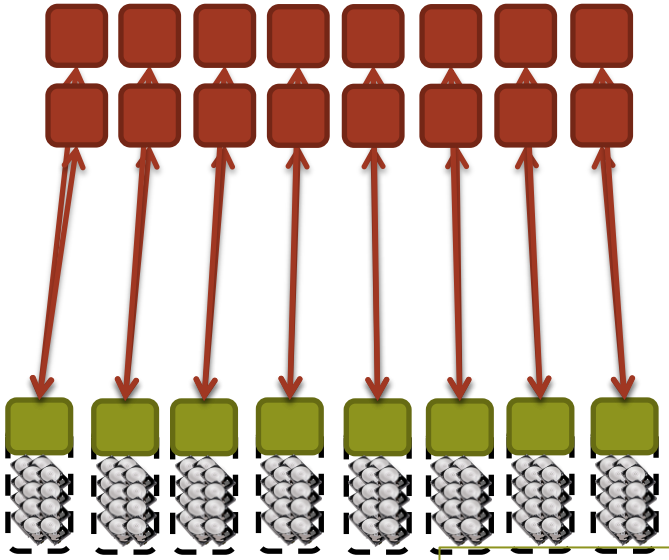




Further info

Case study: Parallel I/O into a single file

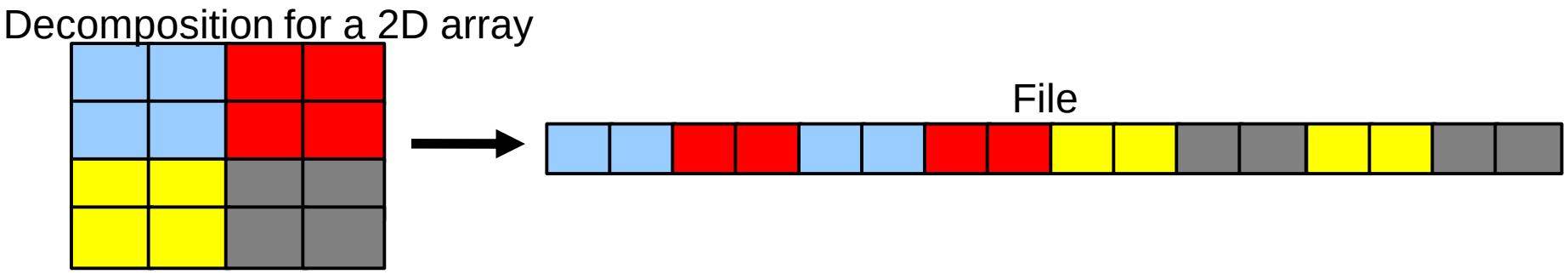
- **A particular code both reads and writes a 377 GB file, runs on 6000 cores**
  - Total I/O volume (reads and writes) is 850 GB
  - Utilizes parallel HDF5 I/O library
- **Default stripe settings: count =4, size=1M**
  - 1800 s run time (~ 30 minutes)
- **New stripe settings: count=-1, size=1M**
  - 625 s run time (~ 10 minutes)



Further info

Issue: Performance bottlenecks due to I/O

- **When using MPI 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



Concluding remarks

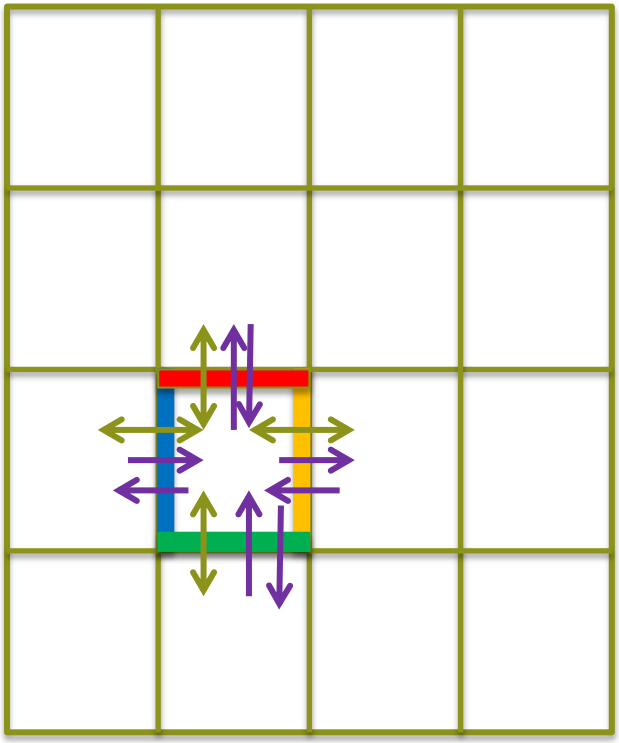
- **Apply the scientific method to performance engineering: make hypotheses and measurements!**
- **Scaling up is the most important consideration in HPC**
- **Possible approaches for alleviating typical scalability bottlenecks**
  - Find the optimal decomposition & rank placement
  - Overlap computation & 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

Lab session: Optimizing point-to-point communication

- **The file halo-exchange(.c|.f90) contains a simple benchmark that simulates the (2D) halo exchange procedure encountered in several domain-decomposition parallel algorithms**
  - There are many ways to implement it with MPI - see the following slides
- **To do:**
  - Read the provided implementations such that you understand the difference between them
  - Measure the obtained bandwidth on Lindgren (and on other platforms you may have an access to), run with e.g. 16x16=256 cores
  - There are also other possibilities - see if you can identify and implement them and measure their performance
  - See if you can improve the performance with any of the suggested Cray MPI environment parameters
  - Bonus exercise (no solution provided): Implement (and benchmark) or just sketch an equivalent scheme for a hybrid MPI+OpenMP application

Lab session: Optimizing point-to-point communication

- Halo exchange in 2D decomposition



Blocking 1

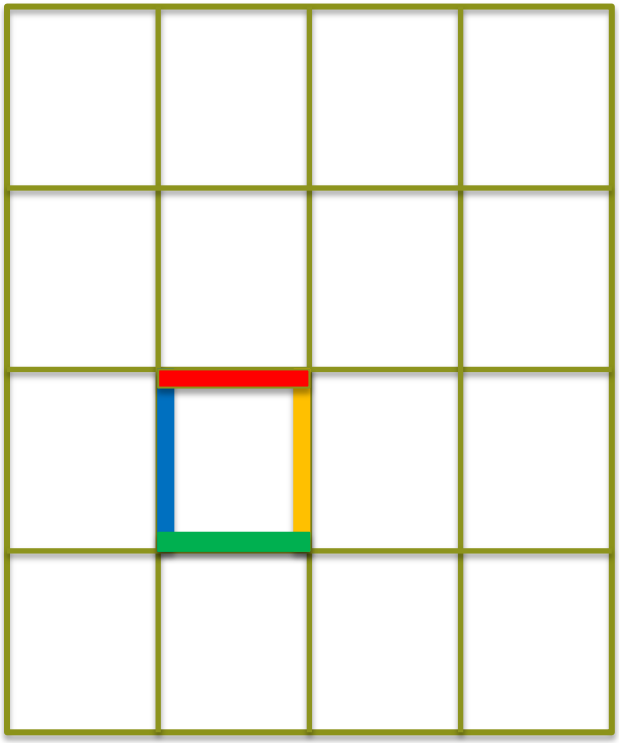
```
Send(to left)
Recv(from left)
Send(to right)
Recv(from right)
Send(to up)
Recv(from up)
Send(to down)
Recv(from down)
```

Blocking 2

```
Send(to left)
Recv(from right)
Send(to right)
Recv(from left)
Send(to up)
Recv(from down)
Send(to down)
Recv(from up)
```

Lab session: Optimizing point-to-point communication

- Halo exchange in 2D decomposition

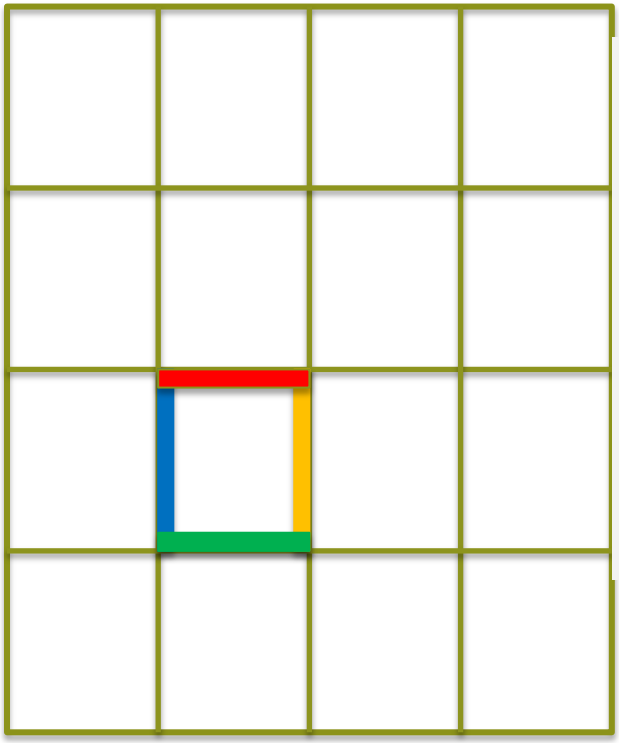


Blocking_3

```
Sendrecv(to left, from right)
Sendrecv(to right, from left)
Sendrecv(to up, from down)
Sendrecv(to down, from up)
```

Lab session: Optimizing point-to-point communication

- Halo exchange in 2D decomposition



Non-Blocking 1

```
Irecv(from left)
Irecv(from right)
Irecv(from up)
Irecv(from down)
Isend(to left)
Isend(to right)
Isend(to up)
Isend(to down)
```

Non-Blocking 2

```
Isend(to left)
Isend(to right)
Isend(to up)
Isend(to down)
Irecv(from left)
Irecv(from right)
Irecv(from up)
Irecv(from down)
```