Performance Engineering

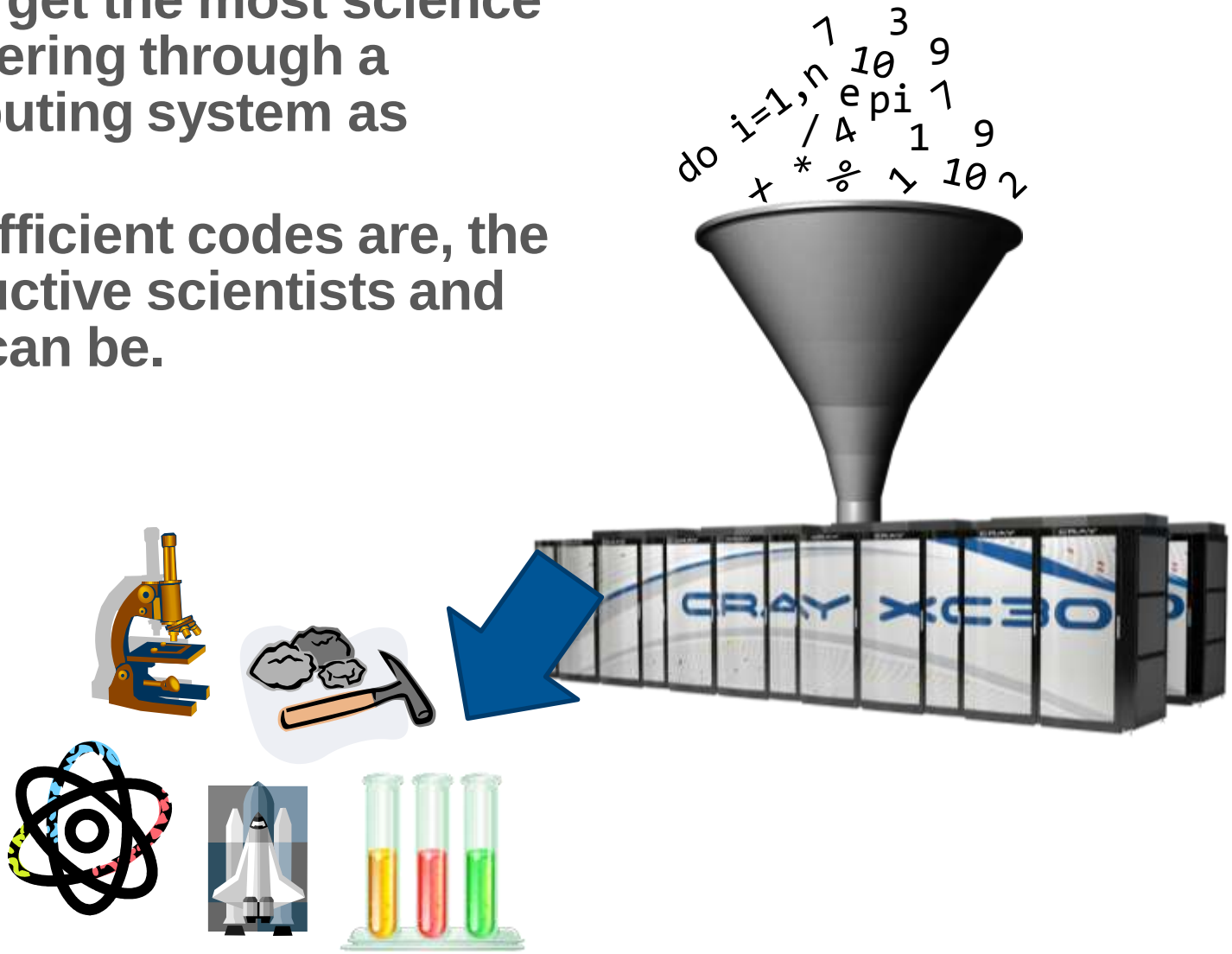
Pekka Manninen, Ph.D.

Cray Inc.
manninen@cray.com

Performance engineering

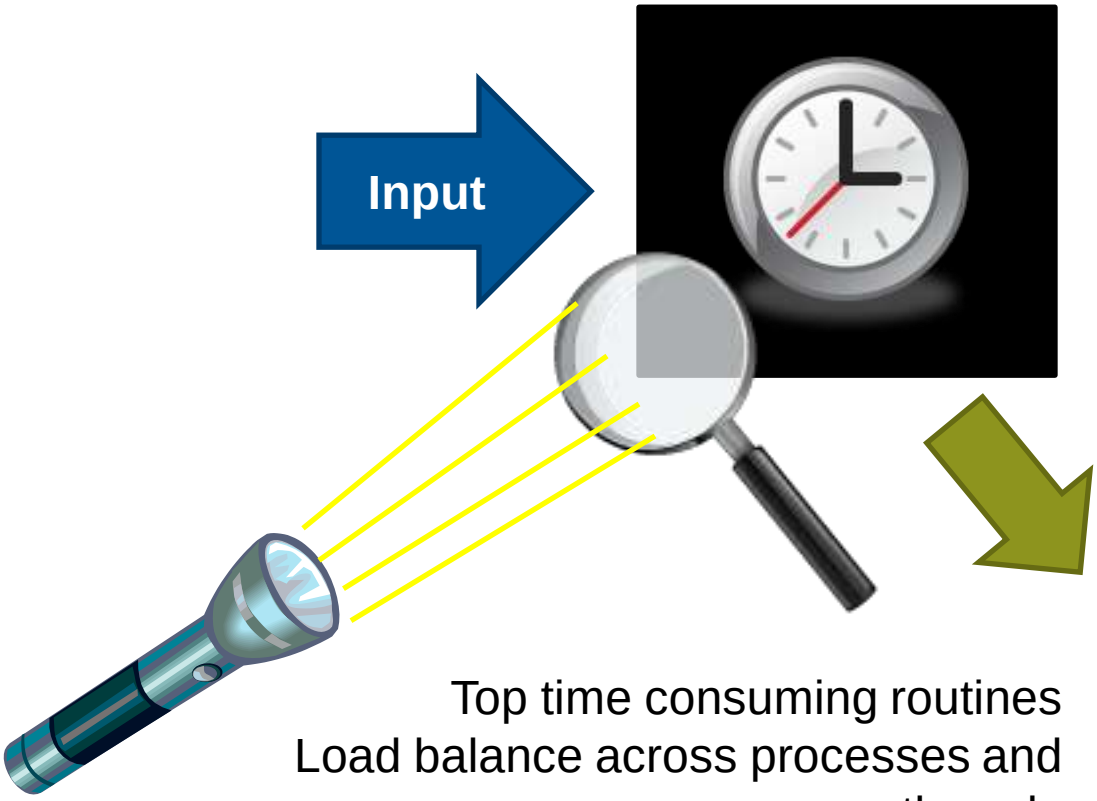
We want to get the most science and engineering through a supercomputing system as possible.

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



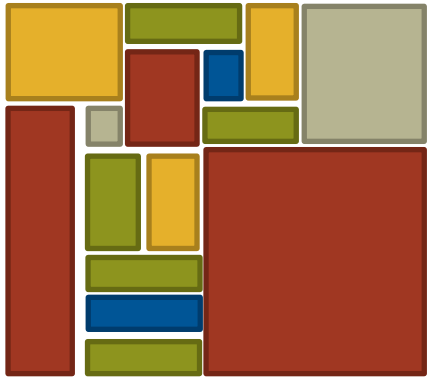
Performance analysis

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



Top time consuming routines
Load balance across processes and threads
Parallel overhead
Communication patterns
Hardware utilization details

Profile Data



Performance engineering – overview

Wednesday

9.15-9.45	**Introduction to performance engineering & briefly about ”optimal porting”**
10.00-10.15	**Coffee break**
10.15-11.00	**Application performance analysis**
11.00-11.15	**Break**
11.15-12.00	**Improving node-level efficiency**
12.00-13.00	**Lunch break**
13.00-13.45	**Improving parallel scalability**
13.45-14.00	**Break**
14.00-17.00	**Hands-on session: application profiling & optimization (coffee being served at 15:00)**

+ A wrap-up & lab session review on Friday morning at 8:30

Lecture 1:

INTRODUCTION TO PERFORMANCE ENGINEERING

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

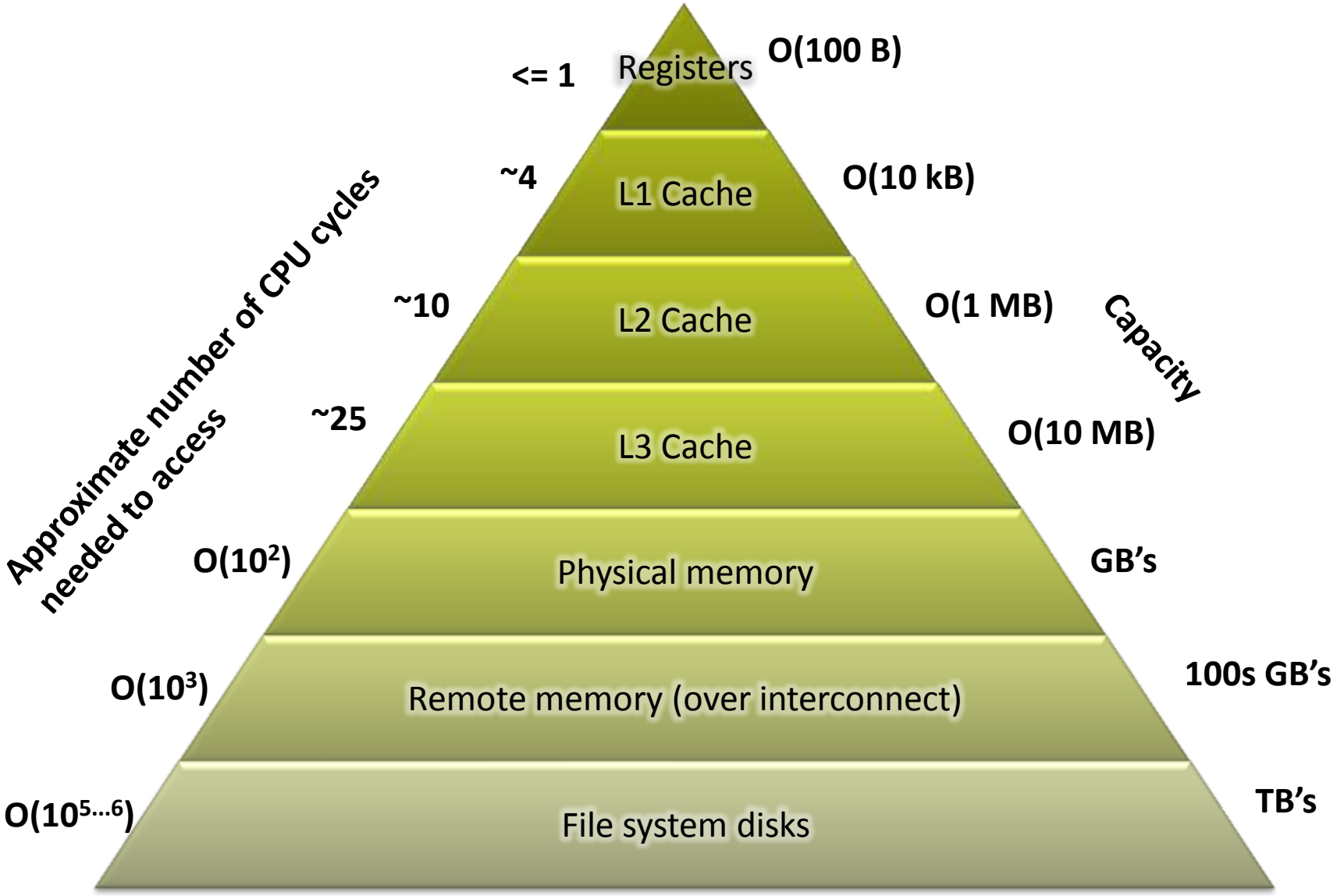
Code optimization

- **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

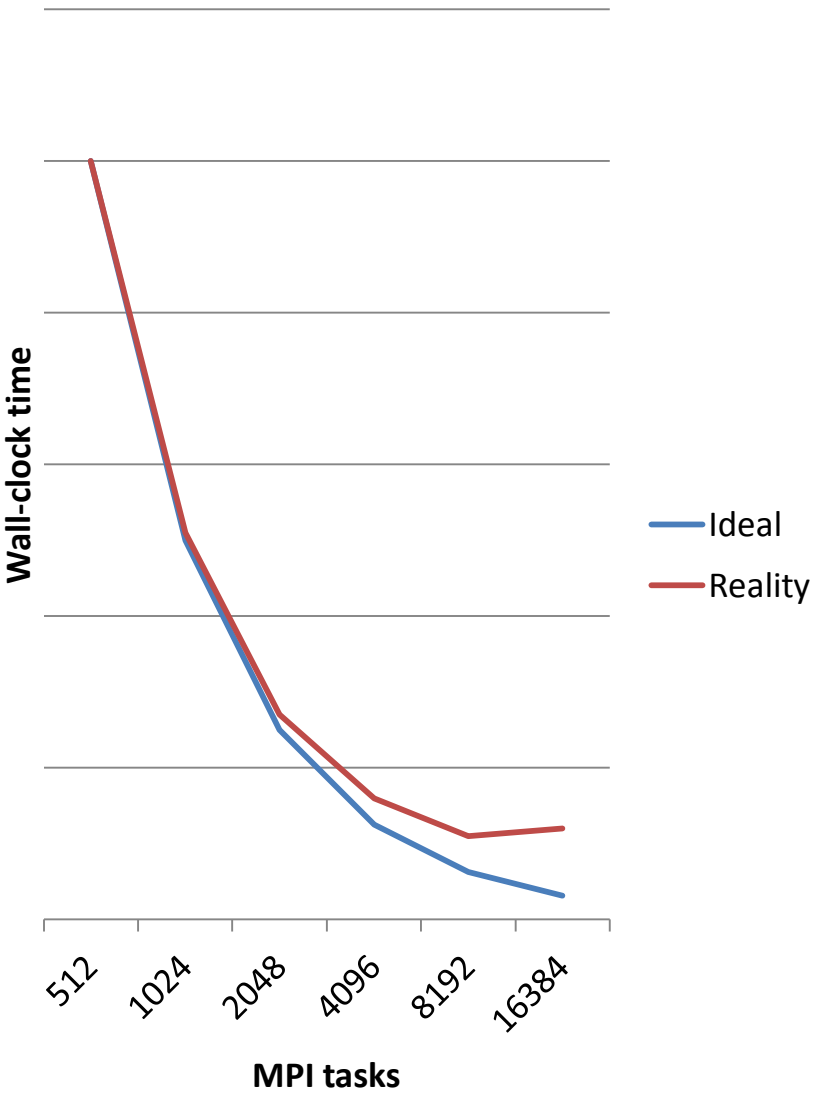
Code optimization

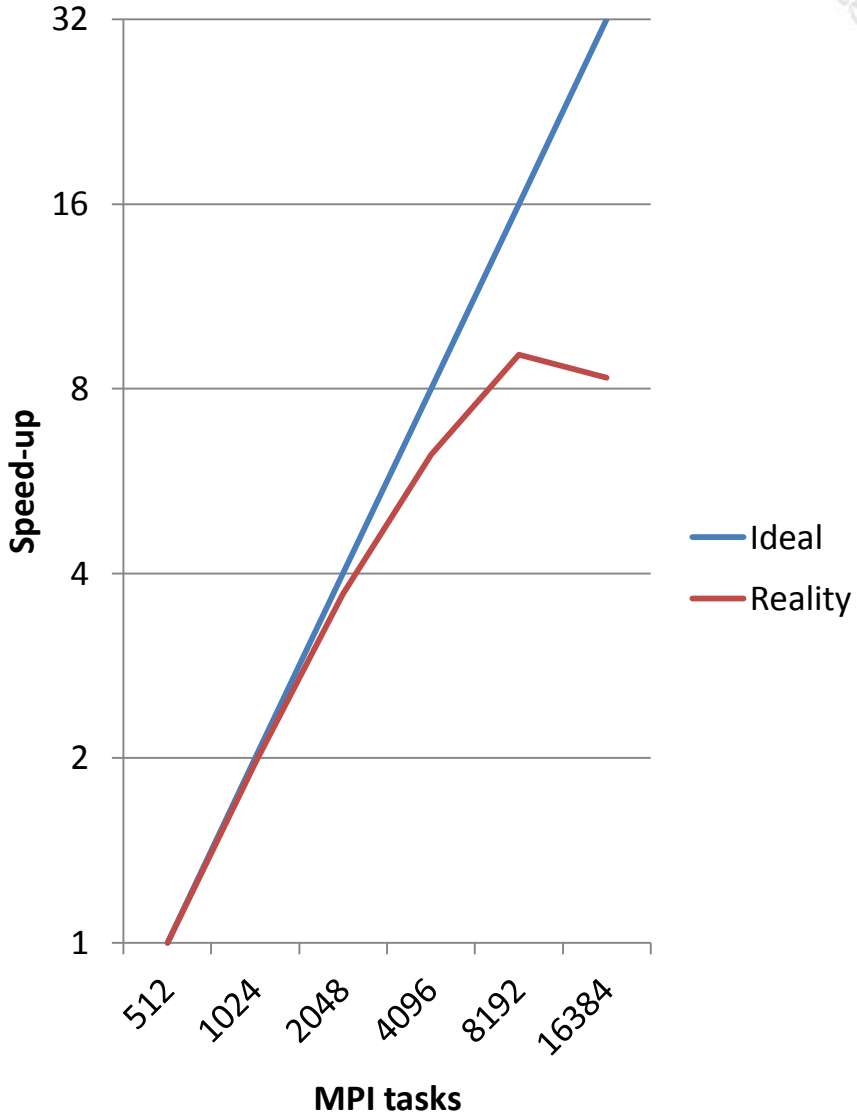
- **Adapting the problem to the underlying hardware**
- **Combination of many aspects**
  - Effective algorithms
  - Implementation: Processor utilization & efficient memory use
  - Parallel scalability
- **Important to understand interactions**
  - Algorithm – code – compiler – libraries – hardware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

Memory hierarchy



Why does scaling end?





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 of network – too many messages or lots of data**

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

Compiler man pages (on Cray)

- **The cc, CC, and ftn man pages contain information about the compiler driver commands**
- **Cray compiler: man craycc, crayCC, and crayftn**
- **GNU compiler: man gcc, g++, and gfortran**
- **PGI compiler: man pgf90, man pgcc, man pgCC**
- **Intel: no man page available, but do `ifort/icc –help`**
- **To verify that you are using the correct version of a compiler, use:**
  - -V option on a cc, CC, or ftn command with CCE
  - --version option on a cc, CC, or ftn command with GNU

Recommended compiler optimization flags

Compiler	Safe compromise	Aggressive
PGI	-fast	-fast -O3 -Mipa=fast,inline
Cray	(default level)	-O3 -hfp3
Intel	-O3	-Ofast -ipa -unroll-aggressive -align -fp-model fast=2
GNU	-O3	-Ofast -funroll-all-loops

On Cray systems, the CPU-specific optimizations are controlled with craype-* modules, e.g. craype-mc12

Libraries

- **Most computational kernels in scientific computing are available in ready-to-run libraries**
  - Tested - less latent bugs
  - Performance optimized
- **On Cray XE/XC the following system-tuned libraries are available**
  - module cray-libsci : BLAS, CBLAS, LAPACK, ScaLAPACK (dense linear algebra), CRAFFT (FFT)
  - module fftw: FFTW 2.x and 3.x
  - module cray-petsc: PETSc (sparse linear algebra, PDE solvers, etc)
  - module cray-trilinos: Trilinos
  - module cray-tpsl: MUMPS, ParMetis, SuperLU, SuperLU_DIST, Hypre, Scotch, and Sundials
  - module cray-hdf5(-parallel): HDF5 (I/O library)

I/O optimization

- **Tuning filesystem (Lustre) parameters may improve application performance**
  - 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

MPI parameters (MPICH-based MPI libraries)

- **Consult man mpi for how to employ these and their proper description**
- **Typically the best impact is seen from the variables**
  - MPICH_GNI_MAX_EAGER_MSG_SIZE
    - Controls the used protocol for point-to-point message transmission
    - Usually increasing the default value improves
  - MPICH_NEMESIS_ASYNC_PROGRESS
    - May improve sc. overlapping when using non-blocking communication
  - MPICH_RANK_REORDER_METHOD
    - Controls the placement of MPI tasks over the nodes
    - Usually requires an optimized placement from CrayPAT suite
  - MPICH_USE_DMAPP_COLL
    - Controls the implementation of collective operations
    - Requires the use of huge pages

Lecture 2:

PERFORMANCE ANALYSIS

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
  - 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
  - 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

Profiling

- **Purpose of the profiling is to find the "hot spots" of the program**
  - Usually execution time, also memory
- **Usually the code has to be recompiled or relinked, sometimes also small code changes are needed**
- **Often several profiling runs with different techiques is needed**
  - Identify the hot spots with one approach, identify the reason for poor performance

Profiling: sampling

The application execution is interrupted at constant intervals and the program counter and call stack is examined

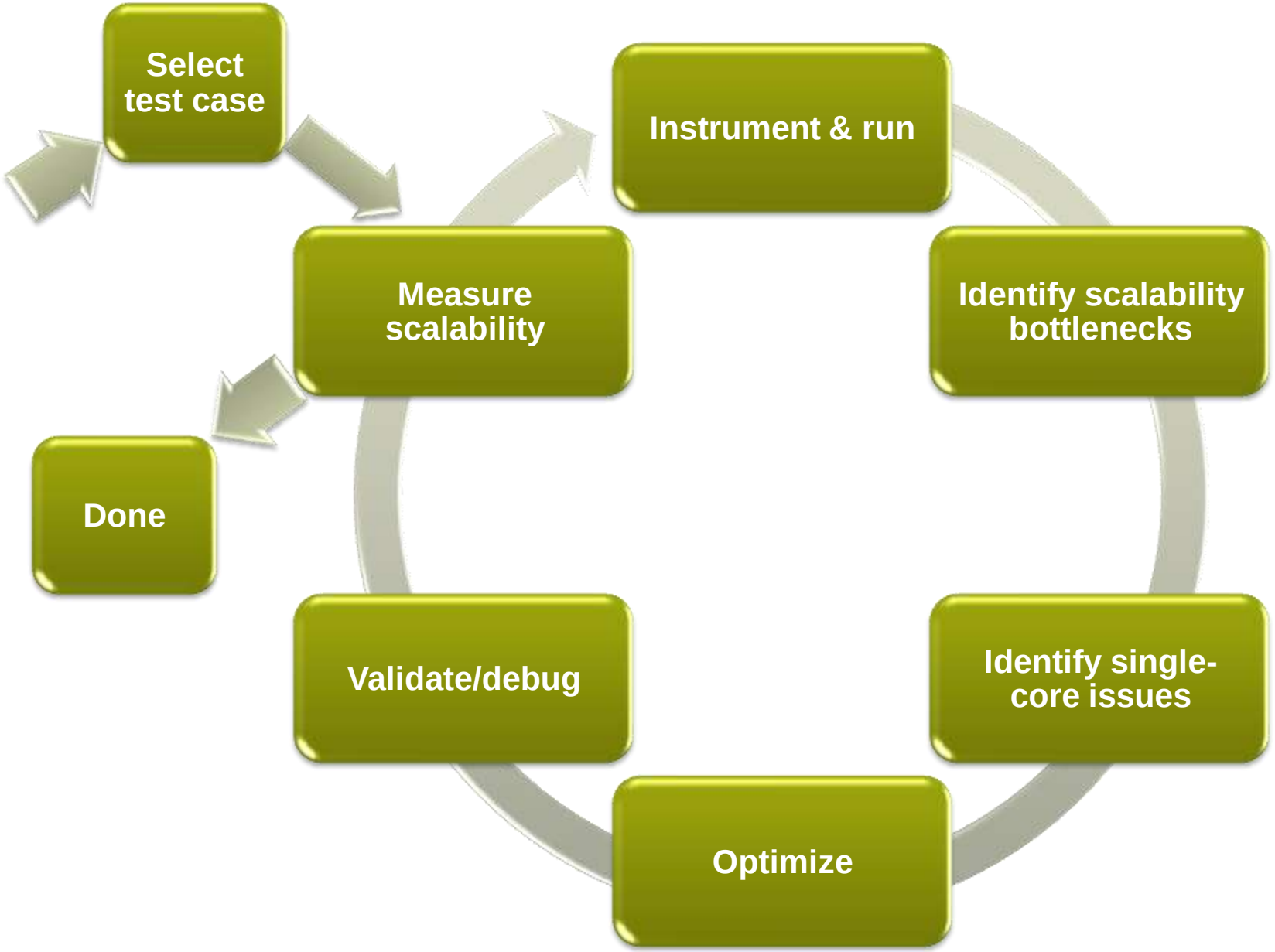
- **Pros**
  - Lightweight
  - Does not interfere the code execution too much
- **Cons**
  - Not always accurate
  - Difficult to catch small functions
  - Results may vary between runs

Profiling: tracing

Hooks are added to function calls (or user-defined points in program) and the required metric is recorded

- **Pros**
  - Can record the program execution accurately and repeatedly
- **Cons**
  - More intrusive
  - Can produce infeasible large log files
  - May change the performance behavior of the program

Code optimization cycle

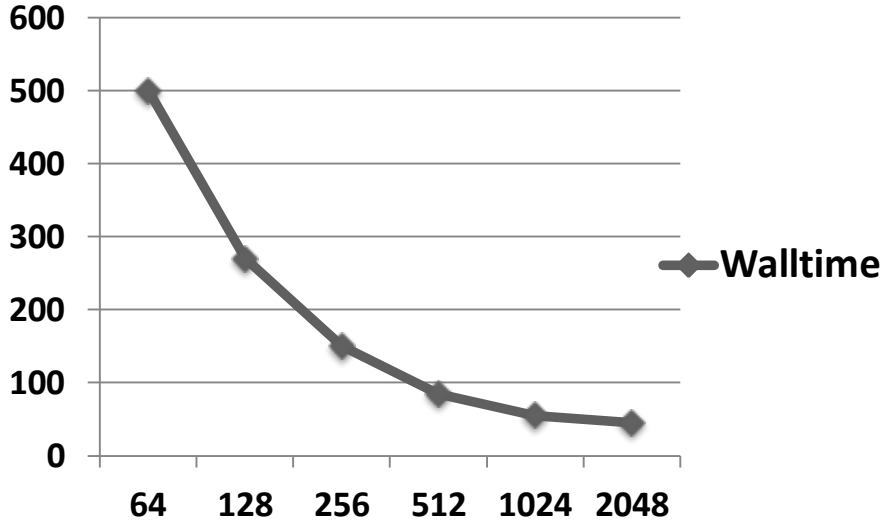


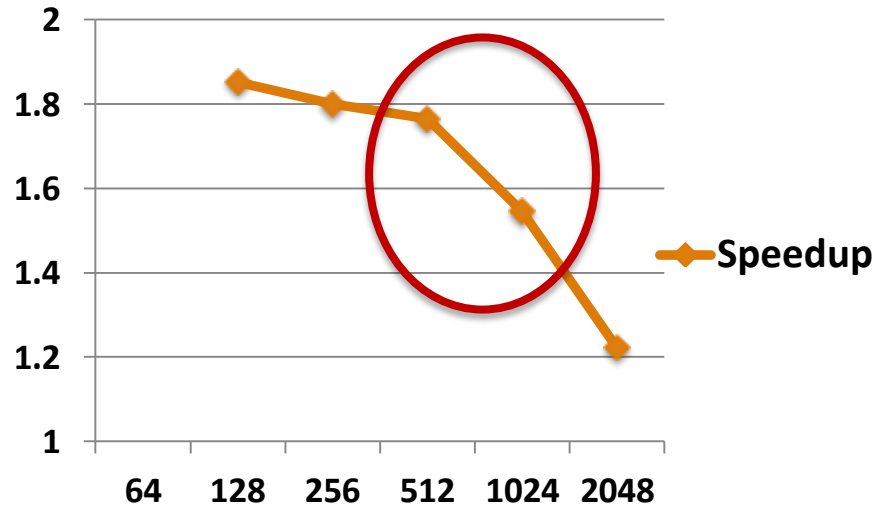
Step 1: Choose a test problem

- **The dataset used in the analysis should**
  - Make 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
- Often we look at strong scaling
  - Also weak scaling is definitely of interest





Step 3: Instrument & run

- **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
  - Tracing also 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**
  `pat_build a.out`
  - You should get an instrumented program `a.out+pat`
- **Run the instrumented application (...+pat) to get a sampling profile**
  - You should get a performance file ("`<sdatafile>.xf`") or multiple files in a directory `<sdatadir>`

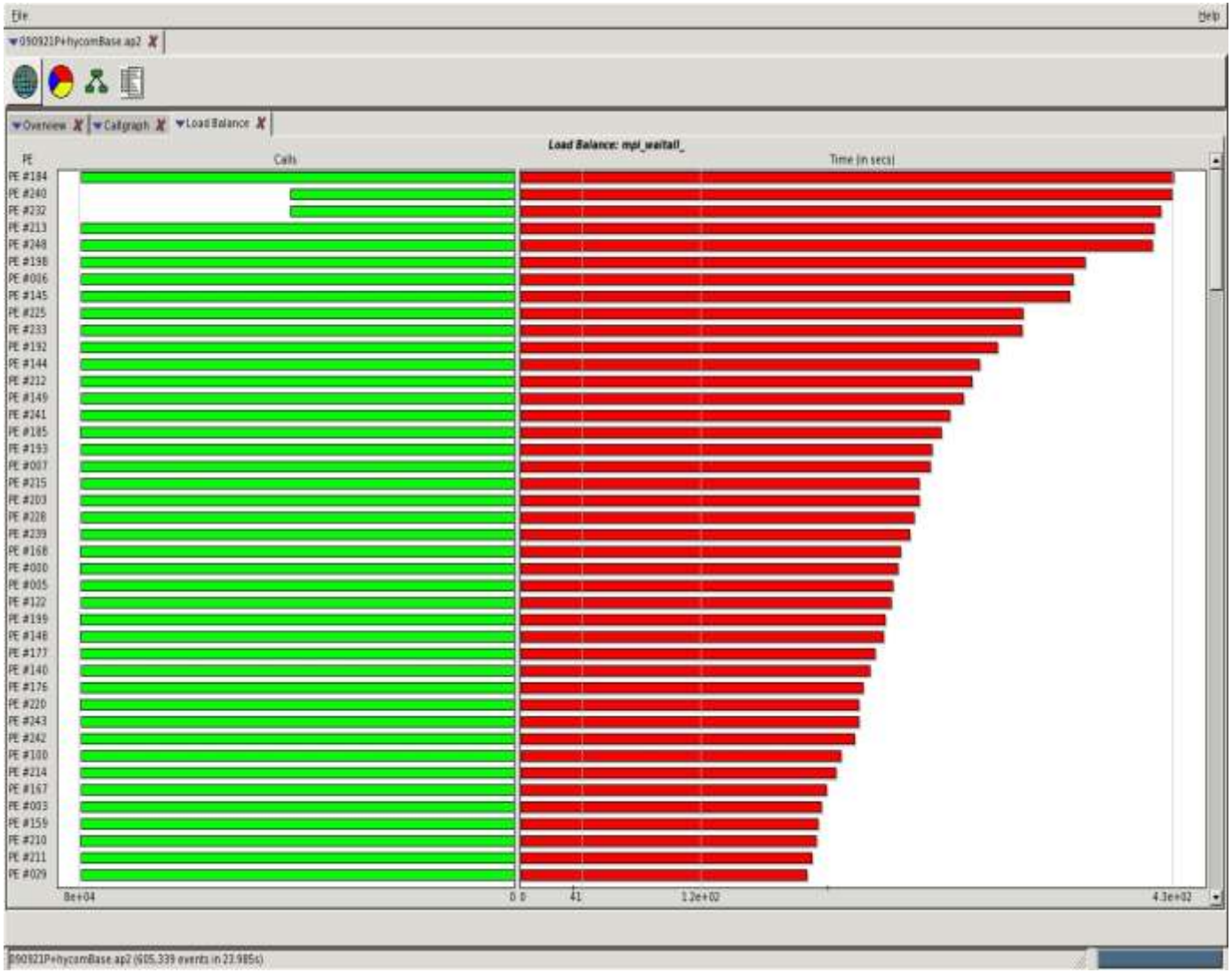
Example with CrayPAT (2/2)

- Generate text report and an `.apa` instrumentation file
  `pat_report <sdatafile>.xf -o sampling.txt`
- Inspect the generated `.apa` file and sampling report whether additional instrumentation is needed
- Instrument application for further analysis (a.out+apa)
  `pat_build -O <apafile>.apa`
- Re-run the newly instrumented application (...+apa)
- Generate text report and visualization file (.ap2)
  `pat_report <data.xf> -o tracing.txt`
- View report in text and/or with Cray Apprentice2
  `app2 <datafile>.ap2`

Step 4: Identify scalability bottlenecks

- **What communication pattern and routines are dominating the true time spent for communication (excluding the sync times)?**
- **How does the message-size profile look like?**
- **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

Example with CrayPAT



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
||||=============================================
...
```

Step 4: Identify scalability bottlenecks

- **Signature: User routines scaling but MPI time blowing up**
  - Issue: Not enough to compute in a domain
    - Weak scaling could still continue
  - Issue: Expensive collectives
  - Issue: Communication increasing as a function of tasks
- **Signature: 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?

Step 5: Find single-core hotspots

- **Remember: pay attention only to user routines that consume significant portion of the total time**
- **Collect the key hardware counters, for example**
  - L1 and L2 cache metrics (PAT_RT_PERFCTR=2)
  - use of vector (SSE/AVX) instructions (PAT_RT_PERFCTR=13)
  - Computational intensity (= ratio of floating point ops / memory accesses) (PAT_RT_PERFCTR=1, default)
- **Trace the “math” group to see if expensive operations (exp, log, sin, cos,...) have a significant role**

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
  ...
  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
  ...
```

Flat profile data

HW counter values

Derived metrics

Step 5: Find single-core hotspots

- **CrayPAT has mechanisms for finding “the” hotspot in one routine (e.g. in case the routine contains several and/or long loops)**
  - CrayPAT API
    - Possibility to introduce “PAT regions” to capture a certain piece of a function/subroutine
  - 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

Table 2: Loop Stats from -hprofile_generate

Loop Incl Time / Total	Loop Incl Time	Loop Incl Time / Hit	Loop Hit	Loop Trips Avg	Loop Notes	Function=/.LOOP\. PE='HIDE'
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 5: Find single-core hotspots

- **Signature: Low L1 and/or L2 cache hit ratios**
  - <96% for L1, <99% for L1+L2
  - Issue: Bad cache alignment
- **Signature: Low vector instruction usage**
  - Issue: Non-vectorizable (hotspot) loops
- **Signature: Traced ”math” group featuring a significant portion in the profile**
  - Issue: Expensive math operations

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.

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

Lecture 3:

IMPROVING NODE-LEVEL EFFICIENCY

Single-core performance analysis

- **Are the hotspot routines compute-bound or memory-bound?**
  - If computational intensity > 1.0 the routine is compute-bound, otherwise memory-bound
- **Signature: low L1 and/or L2 cache hit ratios**
  - <96% for L1, <99% for L1+L2
  - Issue: Bad cache alignment
- **Signature: low vector instruction usage**
  - Issue: Non-vectorizable (hotspot) loops
- **Signature: traced ”math” group featuring a significant portion in the profile**
  - Issue: Expensive operations

Doesn't the compiler do everything?

- **Not yet...**
  - Standard answer, unchanged for the last 50 years or so
- **You can make a big difference to code performance**
  - Helping the compiler spot optimisation opportunities
  - Using the insight of your application
  - Removing obscure (and obsolescent) “optimisations” in older code
    - Simple code is the best, until otherwise proven
- **No fixed rules: optimize on case-by-case basis**
  - But first, check what the compiler is already doing

Compiler feedback/output

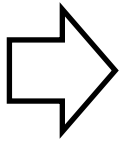
- **Cray compiler:** `ftn -rm …` **or** `cc/CC -hlist=m …`
  - Compiler generates an <source file name>.lst file that contains annotated listing of your source code
- **PGI compiler:** `ftn/cc -Minfo=all -Mneginfo`
- **Intel compiler:** `ftn/cc -opt-report 3 -vec-report 6`
  - If you want this into a file: add `-opt-report-file=filename`
  - See `ifort --help reports`
- **GNU compiler:** `ftn/cc -ftree-vectorizer-verbose=6`

Issue: Bad cache alignment

- **If multi-dimensional arrays are addressed in a wrong order, it causes a lot of cache misses = bad performance**
  - C is row-major, Fortran column-major
  - A compiler may re-order loops automatically (see output)

```
real a(N,M)
real sum = 0;

do i=1,N
  do j=1,M
    sum = sum + a(i,j)
  end do
end do
```



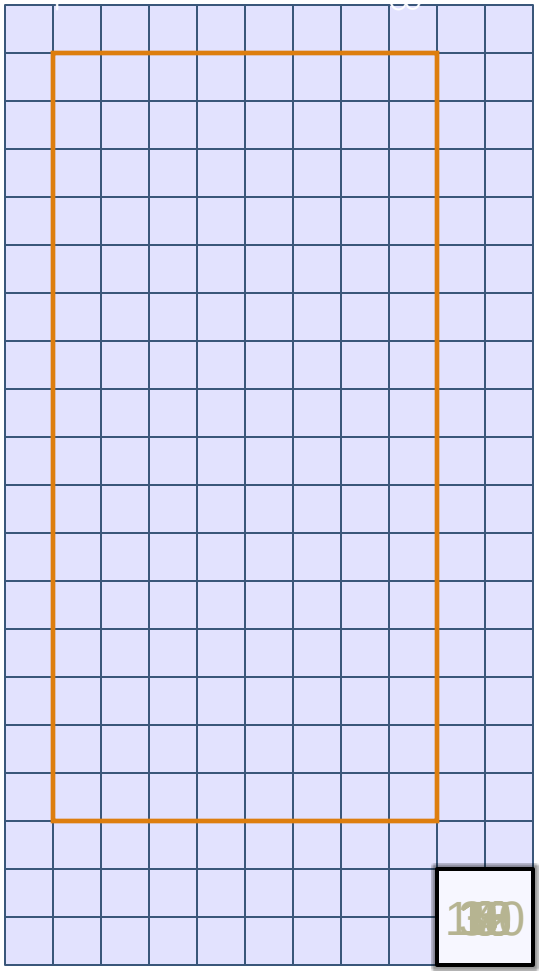
```
real a(N,M)
real sum = 0

do j=1,M
  do i=1,N
    sum = sum + a(i,j)
  end do
end do
```

Issue: Bad cache alignment

- **Loop blocking = Large loops are partitioned by hand such that the data in inner loops stays in caches**
  - A prime example is matrix-matrix multiply coding
- **Complicated optimization: optimal block size is a machine dependent factor as there is a strong connection to L1 and L2 cache sizes**
- **Some compilers do loop blocking automatically**
  - See the compiler output
  - You can assist it using compiler pragmas/directives

Cache Use in Stencil Computations

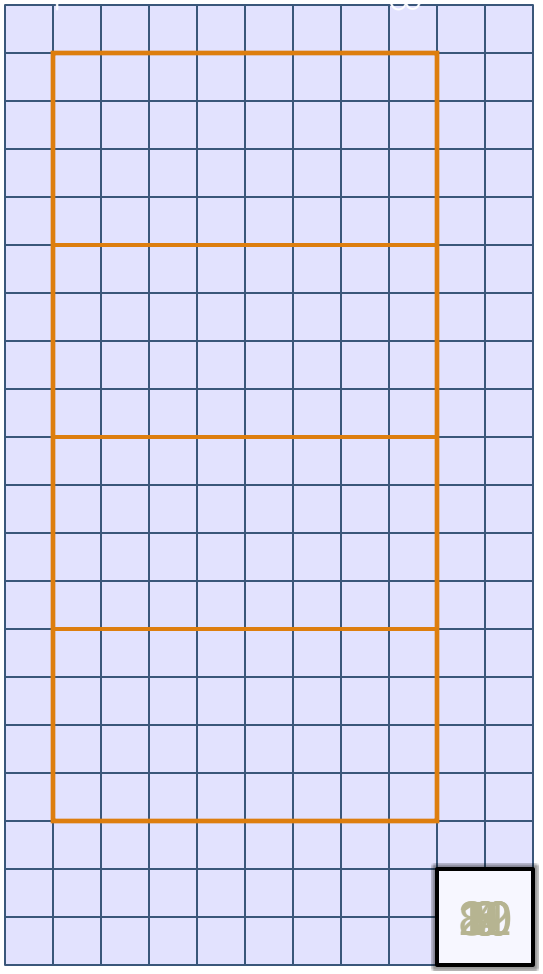


- **2D Laplacian**

```
do j = 1, 8
   do i = 1, 16
      a = u(i-1,j) + u(i+1,j)   &
          - 4*u(i,j)            &
          + u(i,j-1) + u(i,j+1)
   end do
end do
```

- **Cache structure for this example:**
  - Each line holds 4 array elements
  - Cache can hold 12 lines of `u` data
- **No cache reuse between outer loop iterations**

Blocking to Increase Reuse

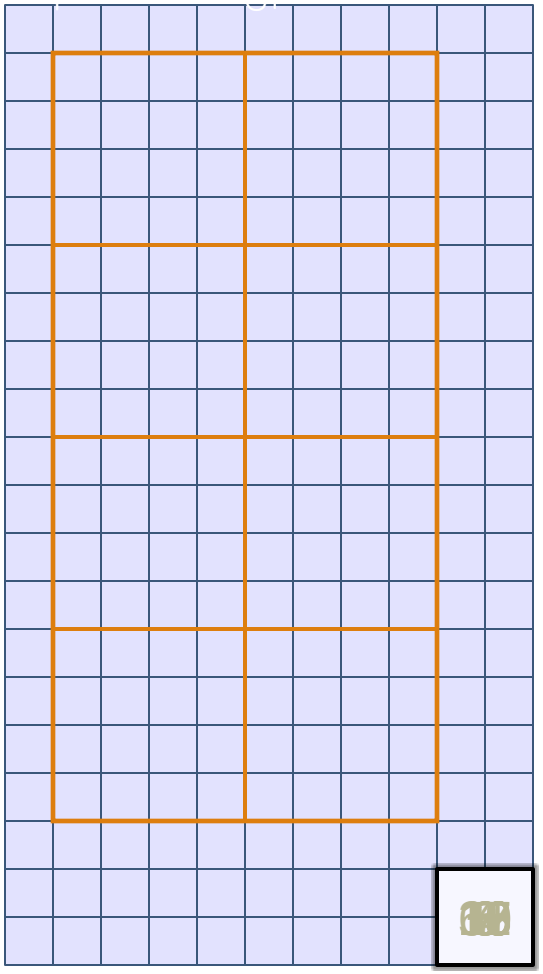


- Unblocked loop: 120 cache misses
- Block the inner loop

```
do IBLOCK = 1, 16, 4
   do j = 1, 8
      do i = IBLOCK, IBLOCK + 3
         a(i,j) = u(i-1,j) + u(i+1,j)   &
                  - 4*u(i,j)            &
                  + u(i,j-1) + u(i,j+1)
      end do
   end do
end do
```

- Now we have reuse of the "j+1" data

Blocking to Increase Reuse



- One-dimensional blocking reduced misses from 120 to 80
- Iterate over 4×4 blocks

```
do JBLOCK = 1, 8, 4
   do IBLOCK = 1, 16, 4
      do j = JBLOCK, JBLOCK + 3
         do i = IBLOCK, IBLOCK + 3
            a(i,j) = u(i-1,j) + u(i+1,j)   &
                     - 4*u(i,j)            &
                     + u(i,j-1) + u(i,j+1)
         end do
      end do
   end do
end do
```

- Better use of spatial locality (cache lines)

Issue: Bad cache alignment

Original loopnest	Blocking with compiler directives	Equivalent explicit code

do k = 6, nz-5
  do j = 6, ny-5
  do i = 6, nx-5
  ! *stencil*
  enddo
  enddo
enddo
	!dir$ blockable(j,k)
	!dir$ blockingsize(16)
	do k = 6, nz-5
	  do j = 6, ny-5
	  do i = 6, nx-5
	  ! *stencil*
	  enddo
	  enddo
	enddo
	
	C:
	#pragma blockable(2)
	#pragma blockingsize(16)
		do kb = 6,nz-5,16
		  do jb = 6,ny-5,16
		  do k = kb,MIN(kb+16-1,nz-5)
		  do j = jb,MIN(jb+16-1,ny-5)
		  do i = 6, nx-5
		  ! *stencil*
		  enddo
		  enddo
		  enddo
		  enddo
		enddo

Loop depth

Issue: Bad cache alignment

- Loop fusion: Useful when the same data is used e.g. in two separate loops: cache-line re-use

Original code	Complete fusion	Partial fusing

do j = 1, Nj
  do i = 1, Ni
  a(i,j)=b(i,j)*2
  enddo
enddo
 	
do j = 1, Nj
  do i = 1, Ni
  a(i,j)=a(i,j)+1
  enddo
enddo
	do j = 1, Nj
	  do i = 1, Ni
	  a(i,j)=b(i,j)*2
	  a(i,j)=a(i,j)+1
	  enddo
	enddo
		do j = 1, Nj
		  do i = 1, Ni
		  a(i,j)=b(i,j)*2
		  enddo
		  do i = 1, Ni
		  a(i,j)=a(i,j)+1
		  enddo
		enddo

Issue: Non-vectorizable loops

- See compiler feedback on why some loops were not vectorized
  - CCE: -hlist=a
  - Intel: -vec-report[0..5]
  - GNU: -ftree-vectorizer-verbose=5

```
16.  + 1-------<    do j = 1,N
17.    1              x = xinit
18.  + 1 r4----<      do i = 1,N
19.    1 r4             x = x + vexpr(i,j)
20.    1 r4             y(i) = y(i) + x
21.    1 r4---->      end do
22.    1------->    end do
```

```
ftn-6254 ftn: VECTOR File = bufpack.F90, Line = 16
  A loop starting at line 16 was not vectorized
because a recurrence was found on "y" at line 20.
ftn-6005 ftn: SCALAR File = bufpack.F90, Line = 18
  A loop starting at line 18 was unrolled 4 times.
ftn-6254 ftn: VECTOR File = bufpack.F90, Line = 18
  A loop starting at line 18 was not vectorized
because a recurrence was found on "x" at line 19.
```

Issue: Non-vectorizable loops

- The compiler will only vectorize loops
- Constant (unit) strides are best
- Indirect addressing will not vectorize (efficiently)
- Can vectorize across inlined functions but not if a procedure call is not inlined
- Needs to know loop tripcount (but only at runtime)
  - i.e. DO WHILE style loops will not vectorize
- No recursion allowed

Issue: Non-vectorizable loops

- **Does the non-vectorized loop have true dependencies?**
  - No: add the pragma/directive `ivdep` on top of the loop
  - Yes: Rewrite the loop
    - Convert loop scalars to vectors
    - Move if-statements out of the loop
- **If you cannot vectorize the entire loop, consider splitting it - so as much of the loop is vectorized as possible**

Issue: Non-vectorized loops

```
38.    Vf------<   do i = 1,N
39.    Vf           x(i) = xinit
40.    Vf------>   end do
41.
42.    ir4-----<   do j = 1,N
43.    ir4 if--<     do i = 1,N
44.    ir4 if          x(i) = x(i) + vexpr(i,j)
45.    ir4 if          y(i) = y(i) + x(i)
46.    ir4 if-->     end do
47.    ir4----->   end do
```

x promoted to vector: trade slightly more memory for better performance

ftn-6007 ftn: SCALAR File = bufpack.F90, Line = 42

A loop starting at line 42 was **interchanged** with the loop starting at line 43.

ftn-6004 ftn: SCALAR File = bufpack.F90, Line = 43

A loop starting at line 43 was **fused** with the loop starting at line 38.

ftn-6204 ftn: VECTOR File = bufpack.F90, Line = 38

A loop starting at line 38 was **vectorized**.

ftn-6208 ftn: VECTOR File = bufpack.F90, Line = 42

A loop starting at line 42 was **vectorized** as part of the loop starting at line 38.

ftn-6005 ftn: SCALAR File = bufpack.F90, Line = 42

A loop starting at line 42 was **unrolled 4 times**.

Issue: Expensive operations

- Cost of different scalar FP operations is roughly as follows:
  - ~1 cycle: +, *
  - ~20 cycles: /, sqrt()
  - ~100-300 cycles: sin, cos, exp, log, ...
- Note that there is also instruction latency and issues related to the pipelining

Issue: Expensive operations

- **Loop hoisting: try to get the expensive operations out of innermost loops**
- **Minimize the use of sin, cos, exp, log, pow, ...**
  - Consider precomputing values to lookup table
  - Use identities, e.g.
    - pow(x,2.5) = x*x*sqrt(x)
    - sin(x)*cos(x) = 0.5*sin(2*x)

Lecture 4:

IMPROVING PARALLEL SCALABILITY

Scalability bottlenecks

- **Signature: 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
- **Signature: 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

Issue: Load imbalances

- **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

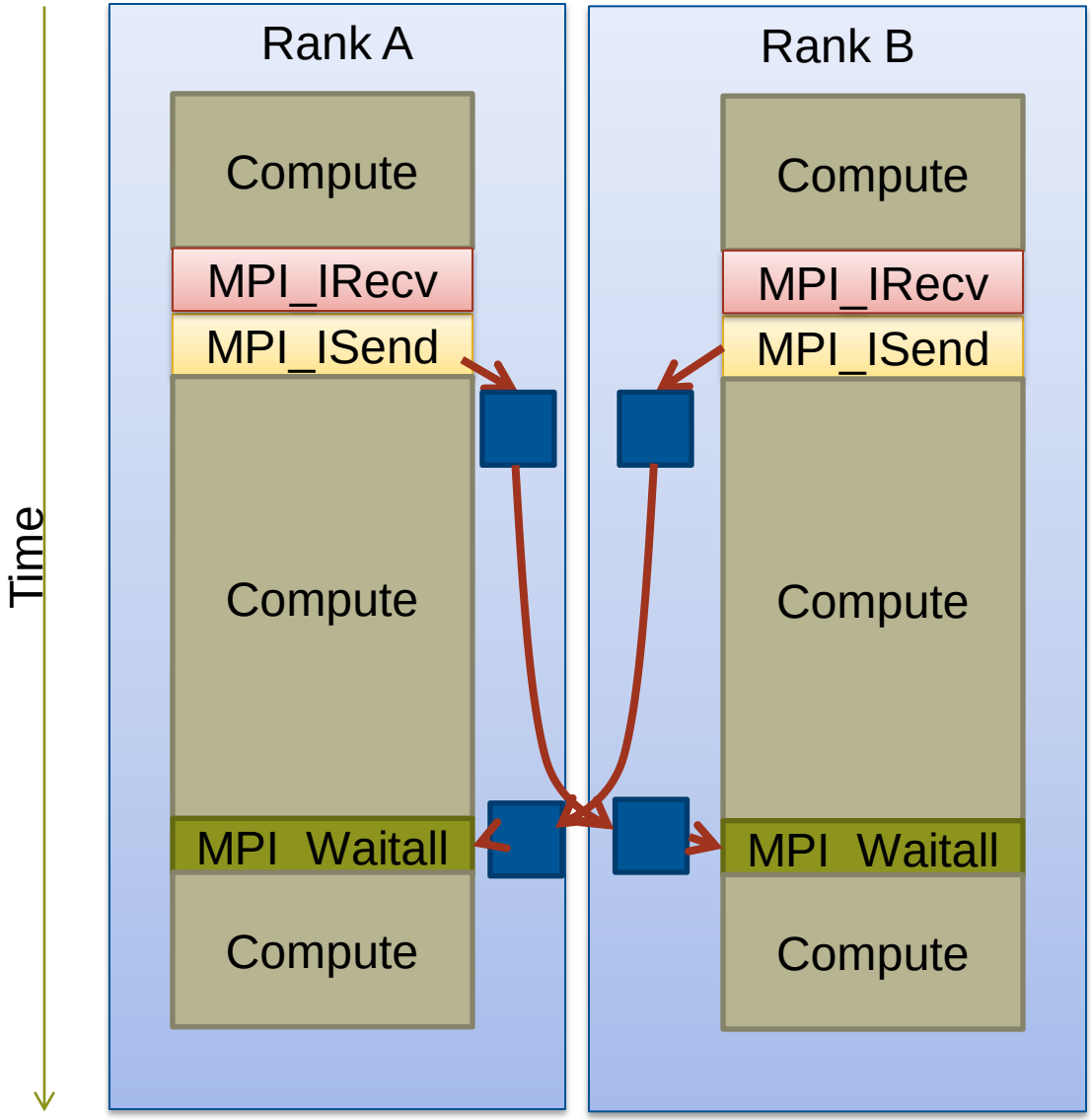
Issue: Load imbalances

- **Changing rank placement**
  - So easy to experiment with that it should be tested with every application!
  - CrayPAT is able to make suggestions for optimal rank placement:
    `pat_report -O mpi_rank_order datafile.xf`
    - This output can then be copied or written into a file named MPICH_RANK_ORDER and used with `MPICH_RANK_REORDER_METHOD=3`

Issue: Point-to-point communication consuming time

- **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, even when using non-blocking communication routines
  - The platform will select the protocol basing on the message size, these limits can be adjusted
- **See the CrayPAT report for the message size profile, which is the dominant protocol**

EAGER potentially allows overlapping

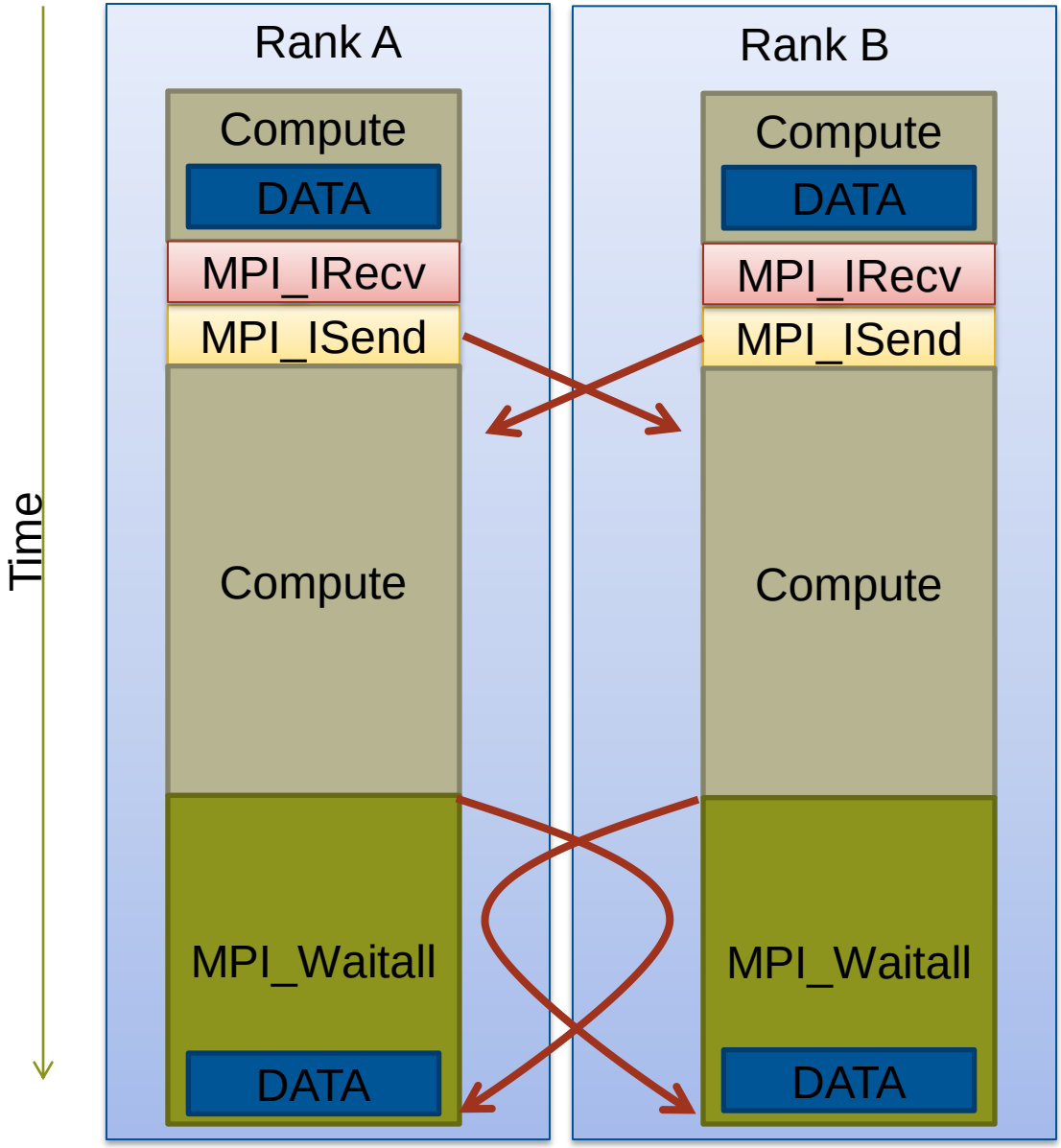


Data is pushed into an empty buffer(s) on the remote processor.

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.

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.

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)
- **Have MPI_Irecv calls open before the corresponding MPI_Isend calls to avoid unnecessary buffer copies and buffer overflows**

Issue: Point-to-point communication consuming time

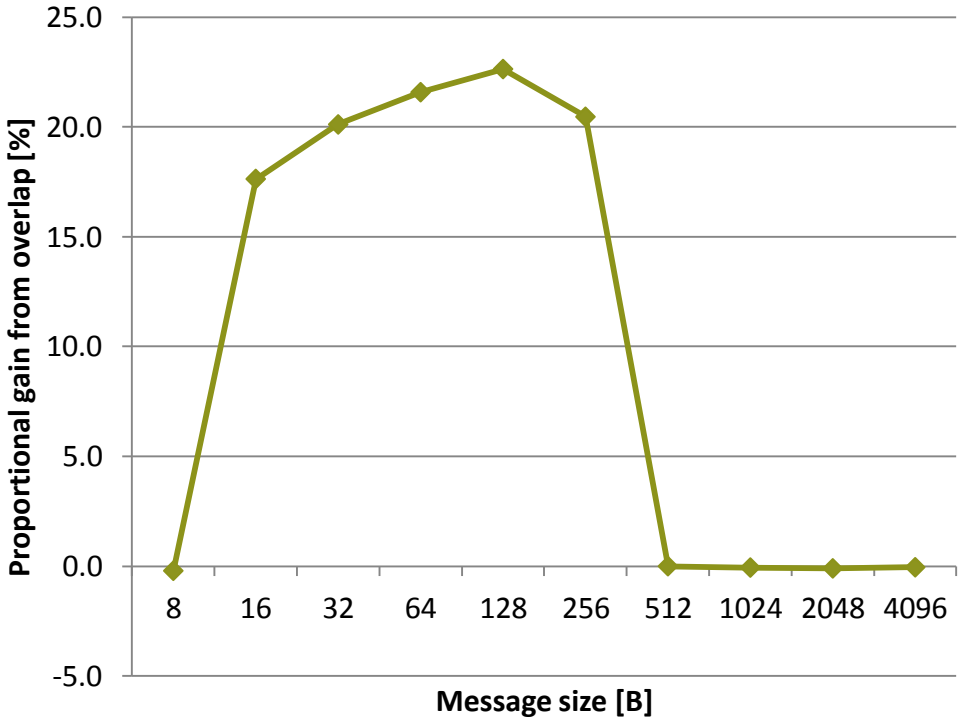
- **On Cray XE & XC: Asynchronous Progress Engine**
- **Progresses rendezvous messages on the background by launching an extra helper thread to each MPI task**
  - This will need spare cores or hyperthreads
  - Works only when running without hyperthreading (aprun -j1)
    - or with using specialized cores (aprun -r1)
- **Consult ‘man mpi’ and there the variable `MPICH_NEMESIS_ASYNC_PROGRESS`**

Issue: Expensive collectives

- **Reducing MPI tasks by hybridizing with OpenMP often helps**
- **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 not referenced later
- **In case of (very) sparse Alltoallv's, point-to-point communication may outperform the collective**

Issue: Expensive collectives

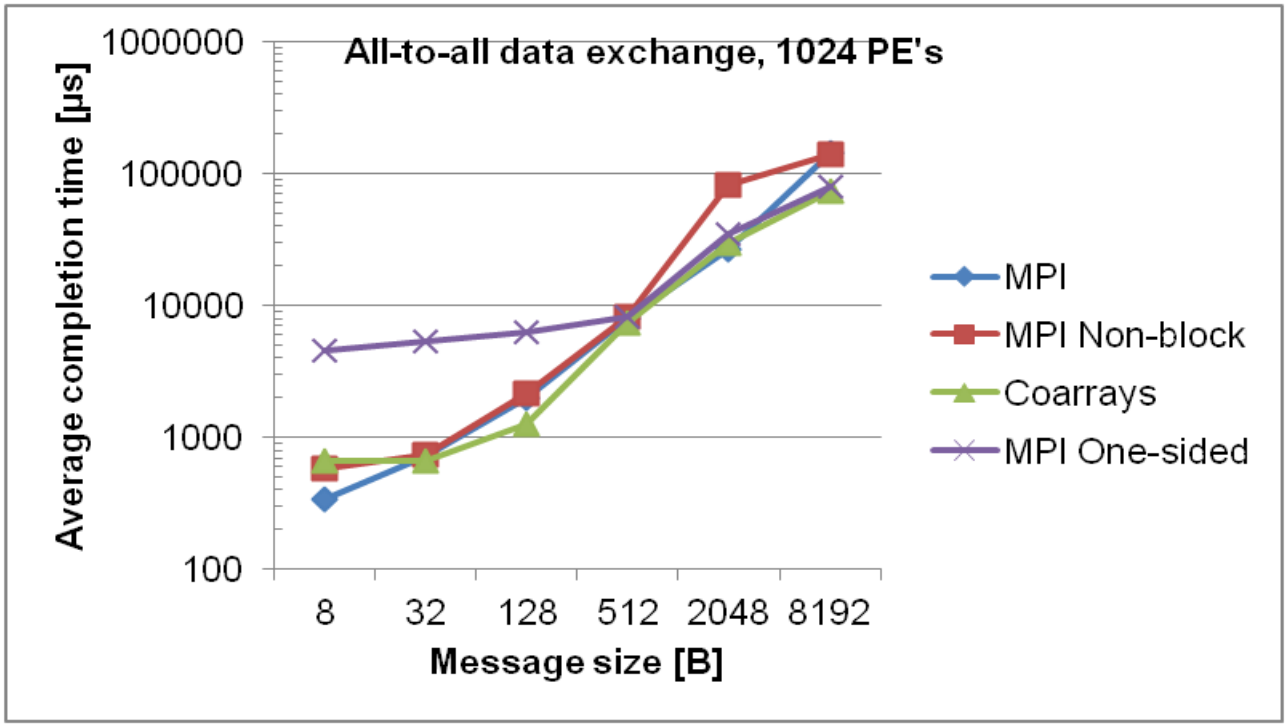
- **Use non-blocking collectives (MPI_Ialltoall,...)**
  - Allow for overlapping collectives with other operations, e.g. computation, I/O or other communication
  - Are in most cases faster than the blocking corresponds even without the overlap
  - Replacement is trivial



MPI_Ialltoall, 1024 cores Cray XC30

Issue: Expensive collectives

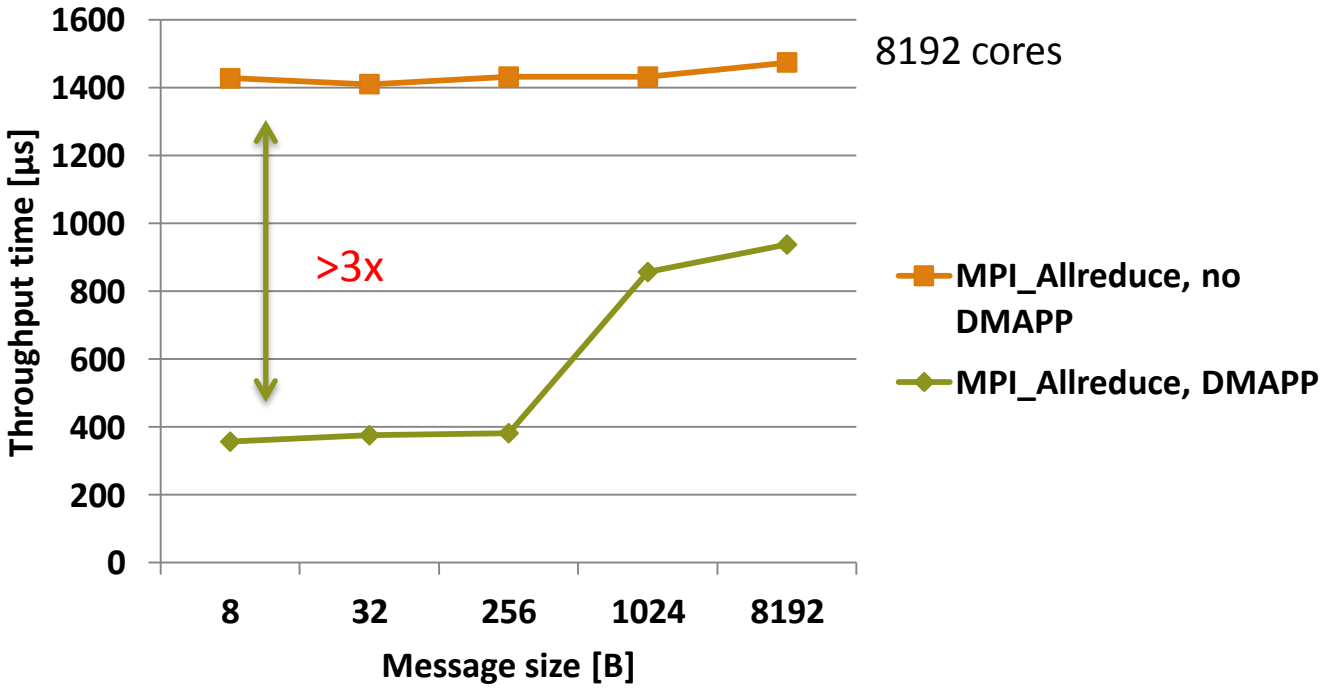
- **Hand-written RDMA collectives may outperform those of the MPI library**
  - Fortran coarrays, Unified Parallel C, MPI one-sided communication



Manninen, CRESTA D4.5.3 (2013)

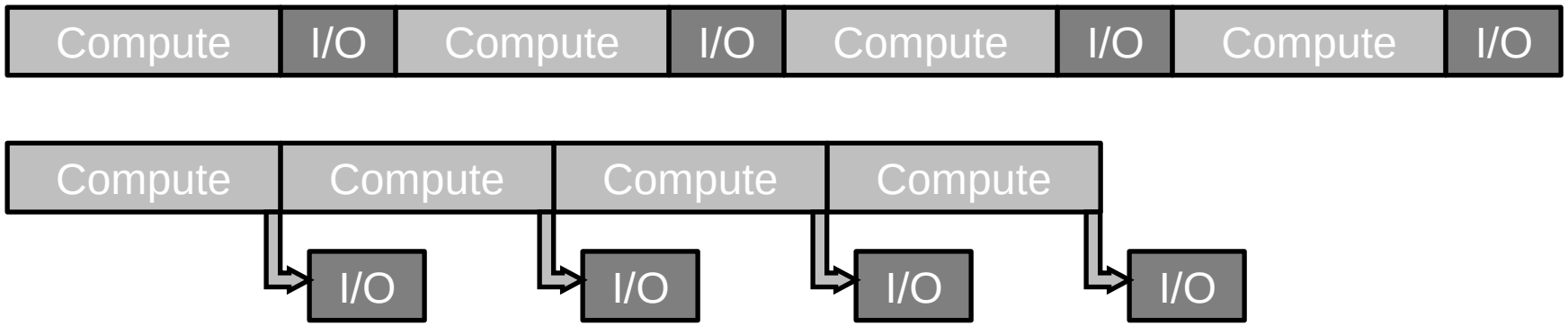
Issue: Expensive collectives

- **On Cray XC, the sc. DMAPP implementation of collectives will (usually significantly) improve the performance of the expensive collectives**
  - Enabled by the variable `MPICH_USE_DMAPP_COLL`
  - Consult 'man mpi'



Issue: Performance bottlenecks due to I/O

- **Parallelize your I/O !**
  - MPI I/O, I/O libraries (HDF5, NetCDF), hand-written schemas,...
  - 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

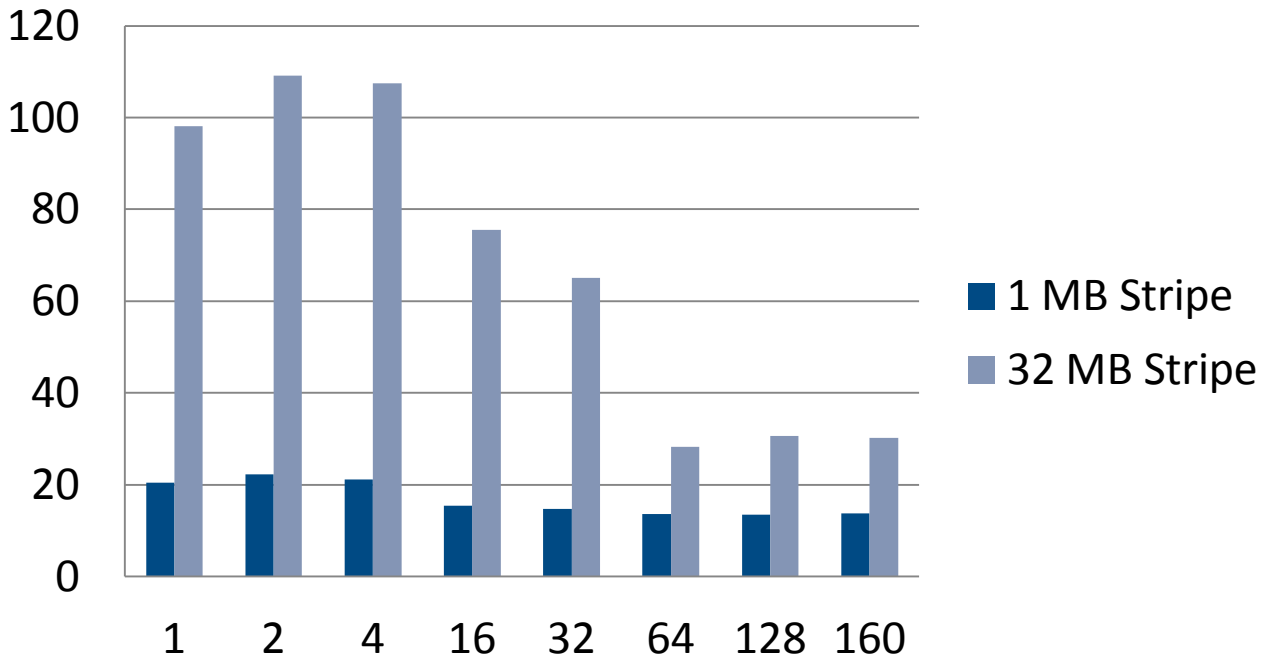


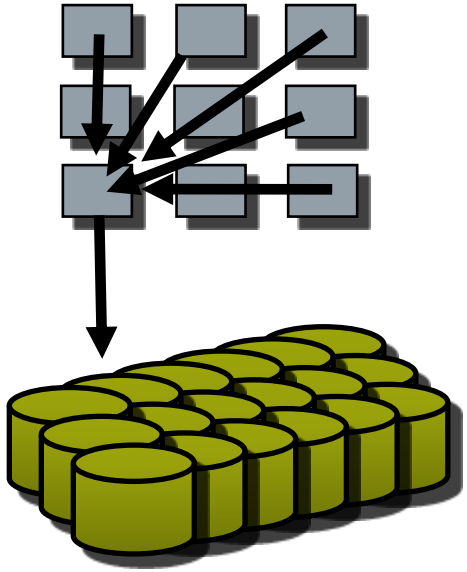
Issue: Performance bottlenecks due to I/O

- **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

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





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)

Issue: Performance bottlenecks due to I/O

- **Use I/O buffering for all sequential I/O**
  - On Cray: 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
  - Insert ”export IOBUF_PARAMS=’*’” to your job script
    - These params control e.g. which files are being buffered, how many bufferes are used etc.
  - See man iobuf

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

Performance Engineering: Concluding remarks

- **Scaling up is the most important consideration in HPC**
- **Possible approaches for alleviating typical scalability bottlenecks**
  - Find the optimal rank placement
  - Use non-blocking communication operations for both p2p and collective communication
  - 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

Performance Engineering: Concluding remarks

- **Mind your I/O!**
  - Use parallel I/O
  - Tune filesystem parameters
- **Node-level performance considerations**
  - Good data locality and cache optimization crucial for performance, together with vectorization
  - Usually an interplay with the compiler - see the compiler feedback and restructure loops to allow for compiler optimization

Introduction to the lab session

- We will analyse and optimize applications throughout the afternoon - more detailed instructions provided as handouts.
- You are encouraged to work with your own application! In case you don’t have one there’s a toy code for 2D heat equation solver being provided.
- The lab is quite inexact - discuss with the instructors about different strategies good for your code.