Performance Engineering

Dr Pekka Manninen
CSC – IT Center for Science Ltd (Finland)

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	Optimal porting
15.00-15.15	Break
15.15-17.00	Lab session

Wednesday

8.30-9.00	Interim wrap-up, 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

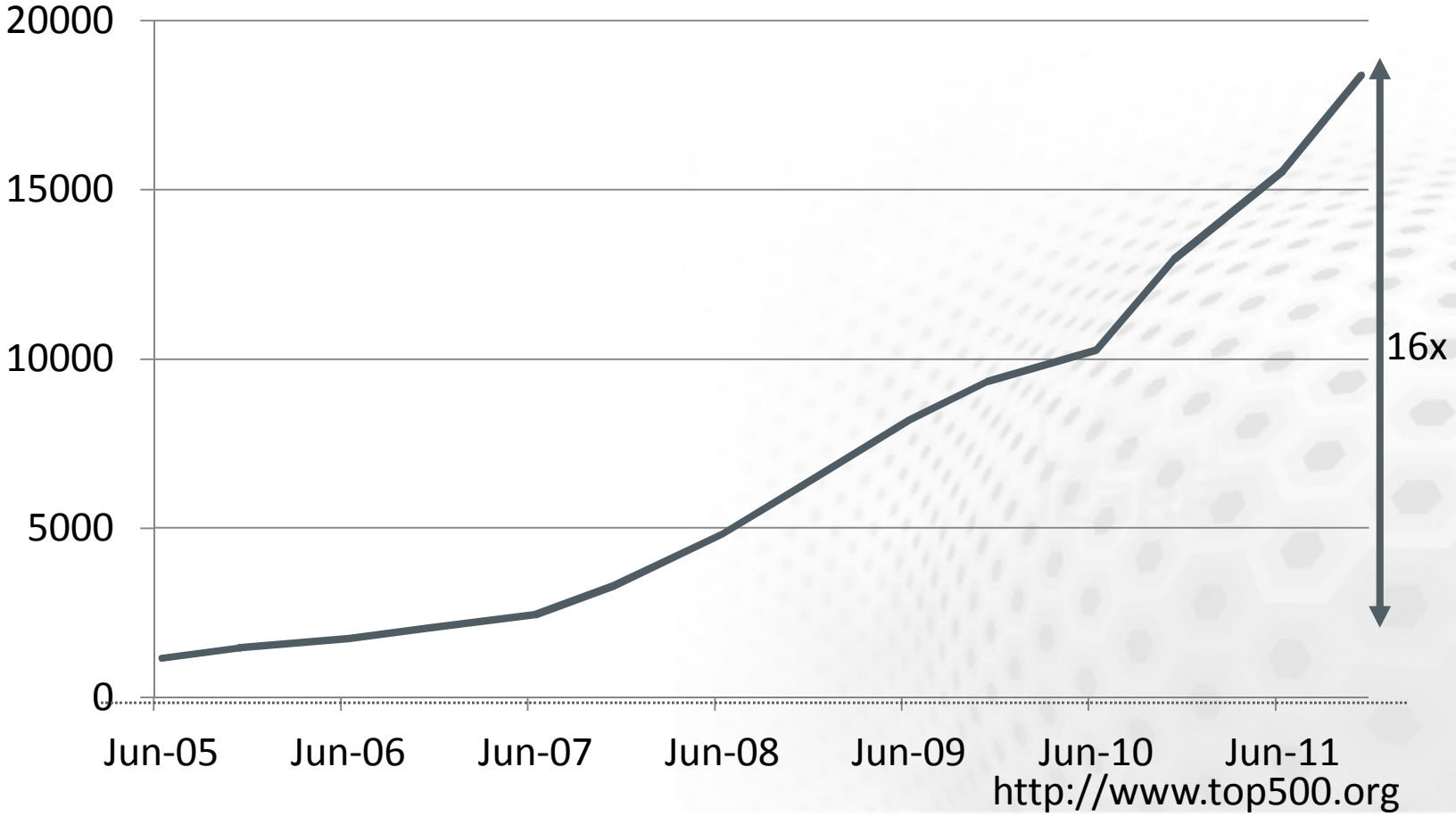
Improving application performance

- Obvious benefits
  - Better throughput => more science
  - Cheaper than new hardware
  - Save energy, compute quota etc.
- ..and some non-obvious ones
  - Potential cross-disciplinary research
  - Deeper understanding of application
- Several trends making optimization even more important

Four easy steps to better application performance

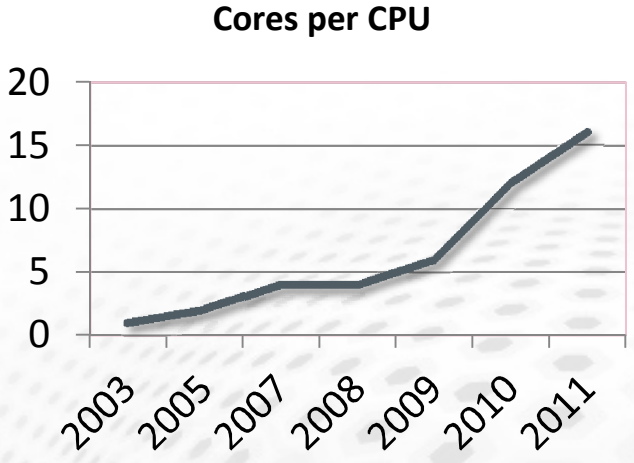
- Find best-performing **compilers** and **compiler flags**
- Employ **tuned libraries** wherever possible
- Find suitable settings for **environment parameters**
- Mind the **I/O**
  - Do not checkpoint too often
  - Do not ask for the output you do not need

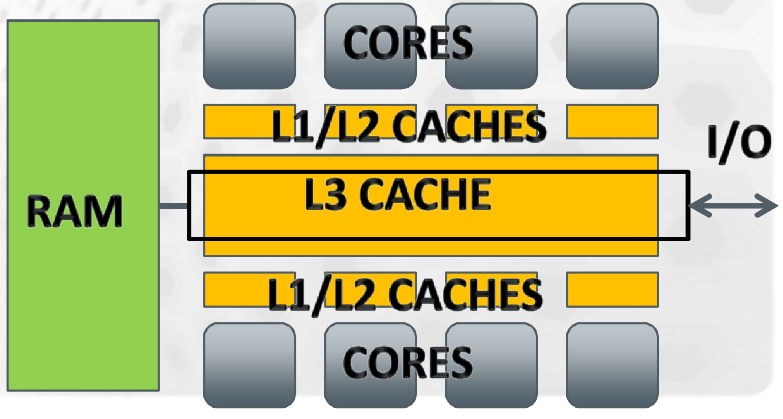
Average CPU cores per system in the Top500



Trend: More and more cores

- Concurrency increasing
  - Processes/threads per node
  - MPI tasks per job
- Complex topology of a node
- Optimization
  - Reconsidering algorithms
  - Efficient parallelization
  - System utilization





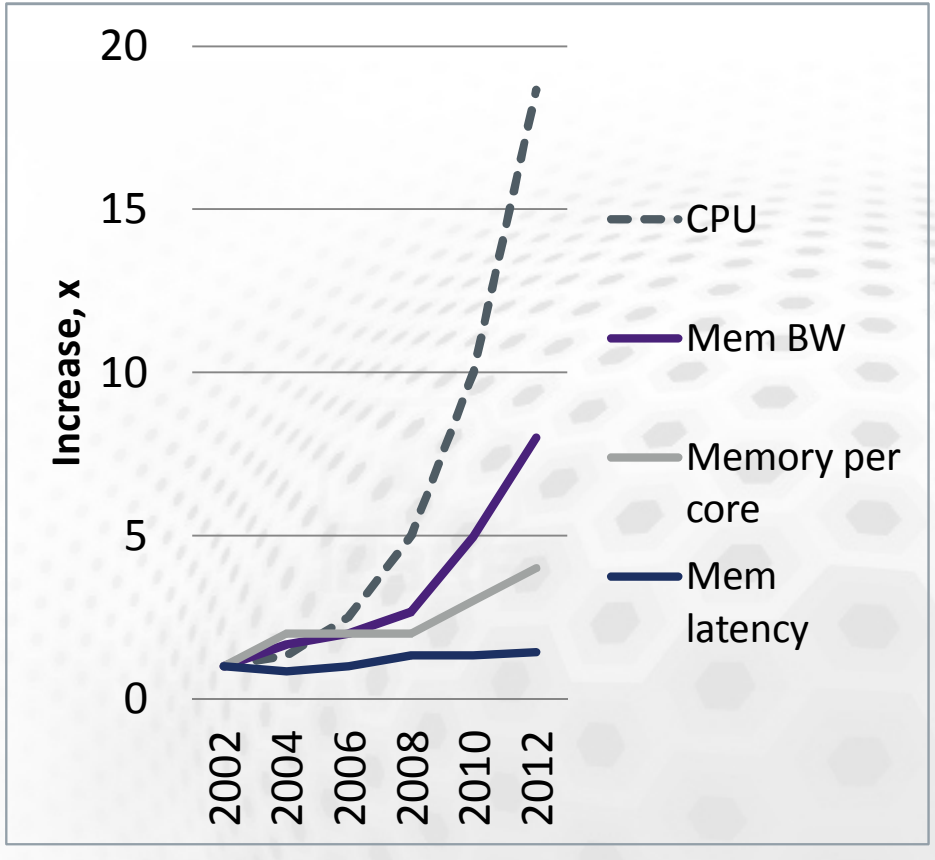
Trend: Fattening vectors

- Making CPUs smarter getting difficult
  - ”Low-hanging fruit” have been picked
    - Pipelining, out-of-order execution etc.
  - Trend: Wider vector units
    - New instruction sets (AVX, AVX2)
- Leveraging the wider vector unit
  - At minimum: Compiler optimization
    - Changes to code often needed
  - Worst case: Algorithm cannot benefit from wider vectors

Year	Name	FP / Hz
2001	SSE2	4
2012	AVX	8
2014?	AVX2	16

Trend: CPU-memory gap growing

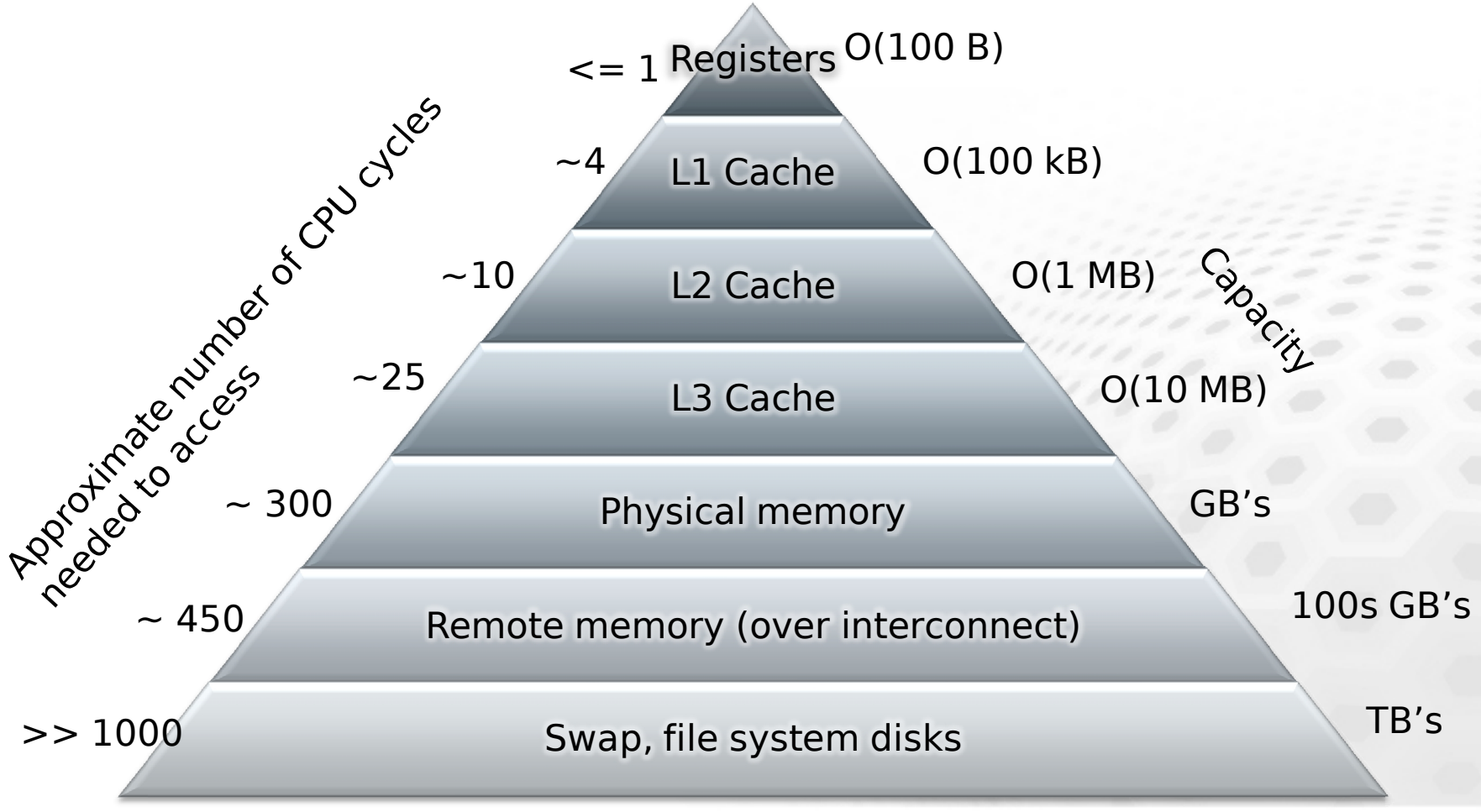
- Memory lagging behind in performance
  - No silver bullet in short term
- Optimization
  - Efficient memory hierarchy utilization
  - Latency hiding
  - Sparing memory use



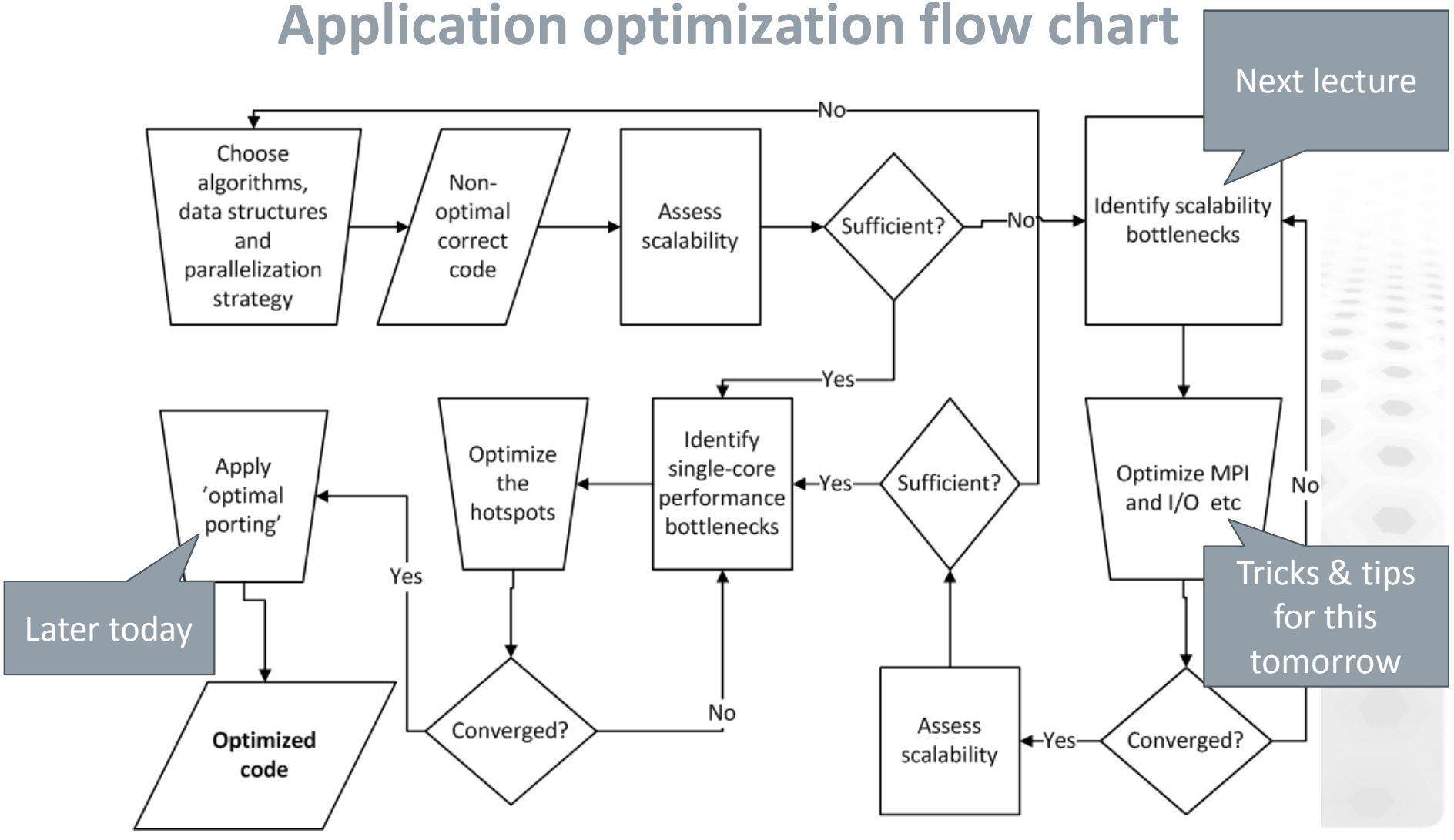
Code optimization

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

Memory hierarchy



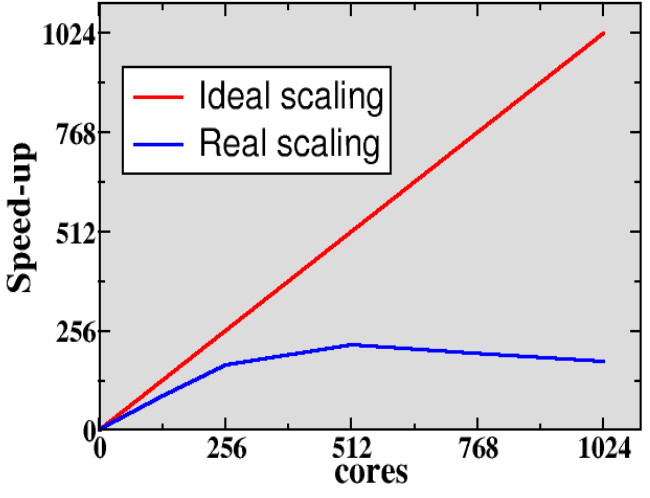
Application optimization flow chart



PART II: APPLICATION PERFORMANCE ANALYSIS

Why does scaling end?

- Amount of data per process small - computation takes little time compared to communication
- 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
- Amdahl's law in general
  - E.g., I/O



Performance measurement

- 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 measurement

- For more insight we need to employ **performance analysis tools**
  - Top time consuming routines (**profile**)
  - **Load balance** across processes and threads
  - Parallel overhead
  - Communication patterns
  - Hardware utilization details
- HPC platforms usually have performance analysis suites
  - CrayPAT, Scalasca, Paraver, Tau,...

Performance analysis

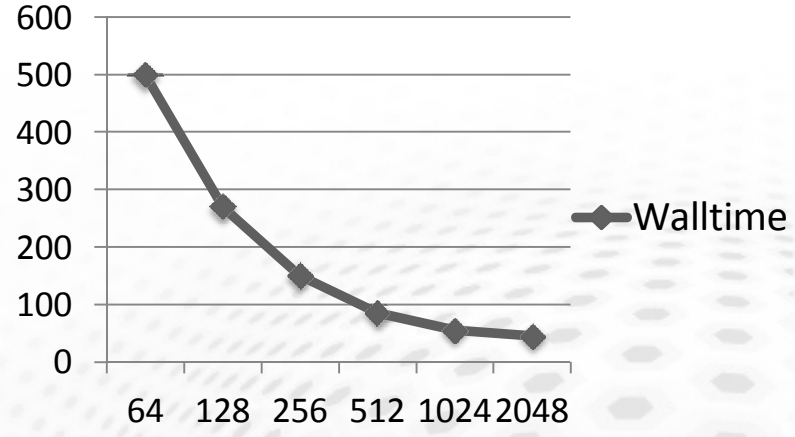
- **Instrumentation** of code
  - adding special measurement code to binary
    - special commands, compiler/linker wrappers
    - automatic or manual
  - normally all routines do not need to be measured
- **Measurement**: running the instrumented binary
  - profile: sum of events over time
  - trace: sequence of events over time
- **Analysis**
  - text based analysis reports
  - visualization

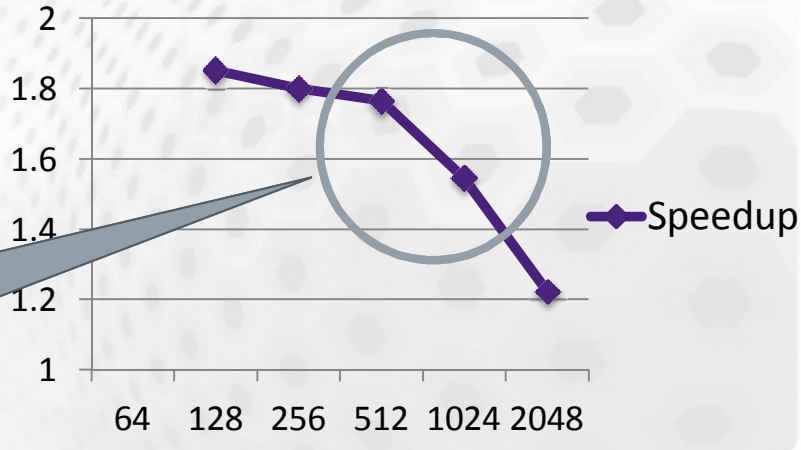
Step 1: Choose a test problem

- The dataset used in the analysis should
  - Make scientific sense
  - 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, however weak scaling is definitely also of interest





We should focus here: what is happening?

Step 3: Instrument the application

- Obtain first a sampling profile to find which user functions should be traced
  - One should not trace them all, it causes excessive overhead
- Make an instrumented exe with tracing 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

Step 4: Assessing the big picture

- What are 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?

Step 5: Analyze load imbalance

- What do the imbalanced routines wait for?
  - Data from other tasks?
  - Imbalanced amount of computation in different tasks?
  - I/O?
- If your code is using also OpenMP
  - Trace for OpenMP API and run-time library too, and regather data

Step 6: Analyze communication

- What is dominating the true time spent for MPI (excluding the sync times)
  - Collectives?
  - Point-to-point communication?
- Note that the analysis tools may report load imbalances as ”real” communication
  - Put an MPI_Barrier before the suspicious routine
- How does the message size profile look like?
  - Are there a lot of small messages?

Step 7: Analyze I/O

- How much I/O?
  - Do the I/O operations take a significant amount of time?
    - Trace POSIX I/O calls (fwrite, fread, write, read,…)
- Are some of the load imbalances etc. MPI hotspots due to I/O?
  - For example, data being gathered to a single task for writing
  - Insert MPI_Barriers to investigate this

Step 8: Analyze single-core bottlenecks

- Check the fraction of the peak and computational intensity
- Obtain different HW counters, for example
  - L1 and L2 cache metrics
  - memory bandwidth information
  - use of SSE instructions
  - conditionals and branching
- Analyze these, whether the major bottleneck is in memory or some floating point issue

Example: Using CrayPAT

- Load necessary modules

```
% module load perftools
```

- Build the application normally (make clean; make, cc, ftn, etc)
- Instrument the application

```
% pat_build -w -u -g mpi,io myexe
```

Tracing user functions, MPI calls and I/O. For other tracing options, see `man pat_build`

- Run the instrumented program

```
% aprun -n 16 ./myexe+pat
```

Example: Using CrayPAT

- Analyse & visualize the results

```
> pat_report -O profile,mpi,io,lb myexe+pat+NNNNNN.xf2
> app2 myexe+pat+NNNNNN.ap2
```

- Further info: man craypat, man pat_build, man pat_report
- For large real-world applications we should use a bit more elaborated approach called *automatic profiling analysis* (APA) that avoids tracing all functions. Refer to CrayPAT documentation.

Asking for a function profile, MPI & I/O statistics and load balance information. See `pat_report -O help` for more options.

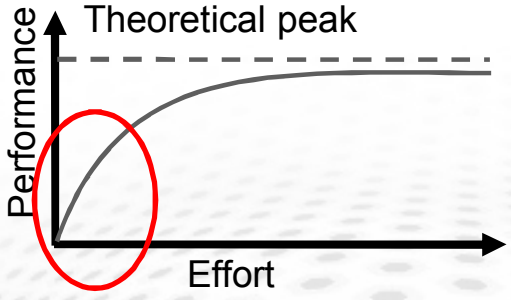
Web resources

- 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

PART III: OPTIMAL PORTING

Optimal Porting

- ”Improving application performance without touching the source code”
- Potential to get significant performance improvements with little effort
- Should be revisited routinely
  - Hardware, OS, compiler and library upgrades
  - Can be automated



Compilers
Compiler flags
Numerical libraries
Intranode placement
Internode placement
Parallel I/O

Choosing a compiler

- Many different choices
  - GNU, PGI, Intel, Pathscale, IBM, Cray etc.
- There is no universally fastest compiler
  - Depends on the application or even input
- Correctness
  - Aggressive optimization may alter results or even break the code - check against reference results!
  - Compiler bugs are not that uncommon

Compiler optimization

- Modern compilers can make sophisticated optimizations
  - At best much more effective than a human
  - In some cases need a lot of assistance
- By default compilers cannot/should not
  - Ignore language standard restrictions
  - Take shortcuts that modify output
  - Make any assumptions on input data

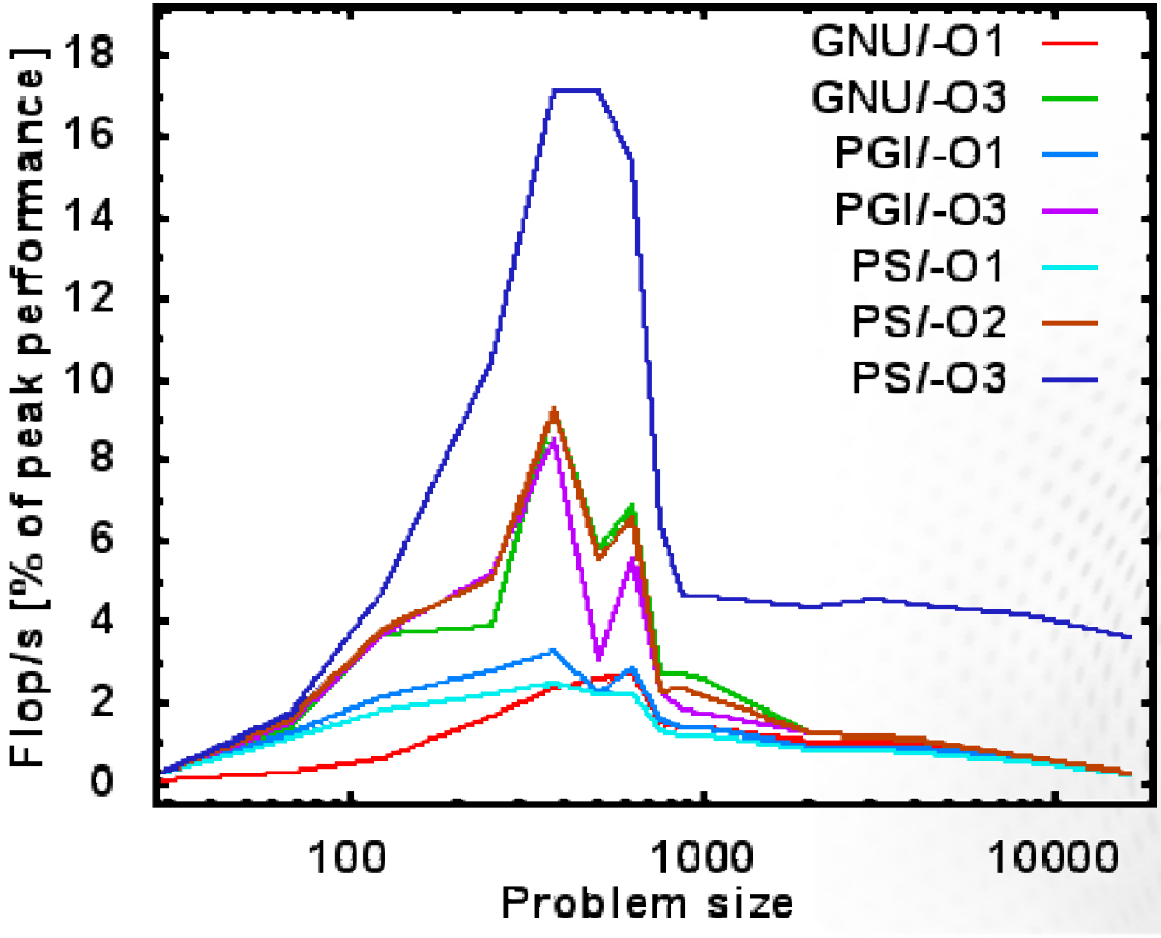
Compiler flags

- Compilers typically have >100 command-line parameters (aka **compiler flags**)
  - Many affect compiler optimization behavior
  - Ridiculously large search space
  - Best flags **not portable**: may vary by application or even input
  - Some flags are generally beneficial
- Some flags are potentially dangerous
  - May lower numerical precision (e.g. ”fast math”)
  - May break the code

The ”dash O” flags

- De-facto standard flags for enabling typical optimizations
  - ’`-O[0-4]`’, sometimes also ’`fast`’
    - For example `gcc -O3` or `icc -fast`
  - The higher the level, the more aggressive optimization
    - Compilers default to some ”safe” level (typically ’`-O2`’)
    - ’`-O0`’ disables optimizations completely
- Typically improves performance but not always
- No standardized definition what the flags actually mean

Example of the ”dash O” Flags



A Conjugate Gradient solver for **Ax**=**b**, using 4 MPI tasks on a XT4 node (AMD Barcelona)

Compiler optimization techniques

- Architecture-specific tuning
  - Tunes all applicable parameters to the defined architecure
- Vectorization
  - Exploiting the vector units of the CPU (SSE, AVX etc.)
  - Improves performance in most cases
- Loop transformations
  - Fusing, splitting, interchanging, unrolling etc.
  - Effectiveness varies

Compiler optimization techniques

- Interprocedural Optimization (IPO or IPA)
  - Analyzes dependencies between functions and modules and possibly even between source files
- Profile Guided Optimization (PGO)
  - Optimizing based on feedback from training runs
  - Process takes time and manual effort
- Fast math
  - Reduces the error checking and/or numerical precision of floating point operations. Use with caution!

Optimization flag examples

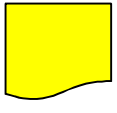
	PGI	GNU
Architecture-specific tuning	`-tp=`*`arch`*	`-march=native`
PGO training	`-Mpfi`	`-fprofile-generate`
PGO optimize	`-Mpfo`	`-fprofile-use`

IPA	`-Mipa=`*`option`*	`-combine`
		`-fwhole-program`
		`-fipa-*`

Fast math	`-Mfprelaxed`	`-ffast-math`

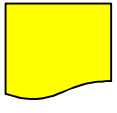
Interesting flags: PGI

- **-fast** is a flag that turns on a set of flags that generally provide good performance. In practice it often provides good, but not the best performance. -fast turns on: -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align -Mflushz
- **-Mipa=fast** Enable InterProcedural Analysis (IPA). Also activates -O2, at a minimum. fast chooses generally optimal -Mipa flags. May increase compile time significantly.
- **-Mvect=sse** Enables the usage of SSE instructions (included in -fast)
- **-Msmart** Enable AMD64-specific post-pass instruction scheduling.
- **-Msmartalloc=huge** Add a call to the routine mallopt in the main routine. Link in the huge page runtime library, so dynamic memory will be allocated in huge pages.
- **-Mfprelaxed** Performs some floating point operations using relaxed precision



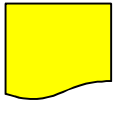
Interesting flags: PGI

- **-Mnoprefetch** In some cases it can be beneficial to turn prefetching off, to avoid evicting data from cache too early.
- **-Mprefetch=plain** Turn on prefetching. In addition to plain there are also other options.
- **-Mvect=noaltcode** Do not generate alternative code for SSE vectorized loops.
- **-Minline=levels:n** Inline n levels. Default n is one, it can be beneficial to attempt higher levels (e.g. 3,5 or even 10)
- **-Munroll=c:u** Try with u=4 or higher
- **-Munroll=n:u** Try with u=4,8 or 16.
- **-Munroll=m:u** Try with u=4,8 or 16.
- **-Mmovnt** Force generation of nontemporal moves.
- **-Mnozerotrip** Don't include a zero-trip test for loops. Use only when all loops are known to execute at least once.



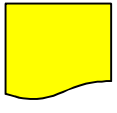
Interesting flags GNU

- **-O2** GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff
- **-O3** Optimize yet more.
- **-Os** Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. May be faster than O2 & O3
- **-ftree-ch** Perform loop header copying on trees. On by default, except when Os is active in which case this often should be activated
- **-funroll-loops** Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop.
- **-frename-register** Avoid false dependencies by making use of registers left over after register allocation. Enabled by default with -funroll-loops.



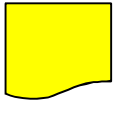
Interesting flags GNU

- **-fsched-stalled-insns=2** 2 instructions can be moved prematurely from the queue of stalled insns into the ready list, during the second scheduling pass. Can also try 4,6,... instructions.
- **-fno-guess-branch-probability** In some cases it can be useful to turn of branch guessing.
- **-ftree=vectorize** Perform loop vectorization on trees.
- **-fno-cprop-registers** After register allocation and post-register allocation instruction splitting, we perform a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.
- **-fprofile-generate**, **-fprofile-use** Create and use profile-based optimization



Interesting flags Intel

- **-O1** Optimize for size
- **-O3** More aggressive optimizations incl. vectorization (-O2 is default)
- **-fast** Collection of common optimizations
  - Contains: -O3 -ipo -no_prec_div -static -xW
- **-x*HOST*** compile for architecture specified by HOST
- **-no_prec_div** Relaxed precision division
- **-mp** Disable optimizations that can affect floating point accuracy
- **-ipo** Interprocedural optimizations
- **-prof-gen / -prof-use** Generate /use profile



Numerical libraries

- Some key numerical routines have de-facto standardized interfaces
  - BLAS, LAPACK, ScaLAPACK
  - FFT (nearly)
- There are multiple implementations of interfaces
  - Both commercial and open-source
  - The so-called ”reference” implementations are useful for checking correctness but have poor performance

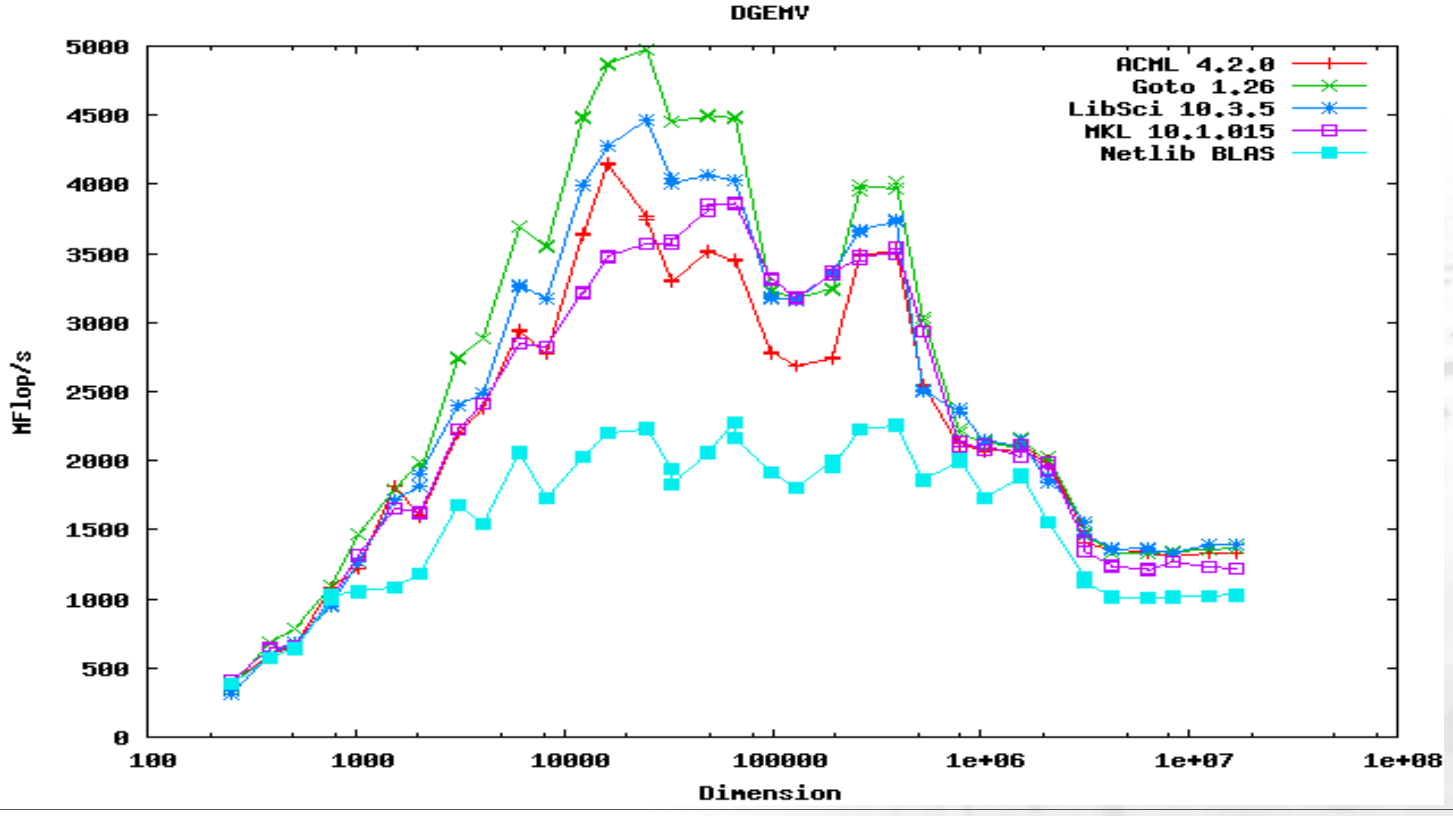
Numerical library collections

- Vendors provide numerical library collections
  - Optimized for the processor or the system architecture
  - Usually contains: BLAS, LAPACK, FFT, vector math, RNG
  - Possibly also sparse solvers and others
- CSC systems have several packages
  - AMD ACML (Louhi, Murska, Vuori)
  - Cray LibSci (Louhi)
  - Intel MKL (Louhi, Murska, Vuori)
  - Nvidia numerical libraries (Vuori)

Optimized BLAS

- Cornerstone of performance for many upper-level libraries and applications
- Many optimized implementations
  - GOTO, MKL, LibSci, ACML, ATLAS etc.
  - Also for GPUs: CUBLAS, ACML-GPU
- Some compilers support translating intrinsic operations (matmul etc.) into calls to a BLAS library
  - GNU Fortran >=4.3: -fexternal-blas

BLAS Performance



BLAS routine for double precision matrix-vector multiplication (“DGEMV”) with different libraries on quad-core Opteron Barcelona

Optimized FFT

- No standard interface
  - Similar function: Plan once, execute plan N times
- Many optimized implementations
  - Syntax look alike but not identical
  - CRAFFT, FFTW, ACML FFT, Intel FFT, CUFFT etc.
- No single best solution
  - Depends on architecture and type of FFT
- Consider saving the plans to a file and reusing them
  - Only useful if FFT dimensions are consistent across runs

Fast math libraries

- Vector math routines
  - Operates on complete arrays
- Intrinsic math libraries
  - Libraries that replace the standard libm intrinsics
    - sin, exp, log etc.
- No standard on naming
  - Good idea to use macros

11.30s

```
for(i=0;i<n;i++){
  b[i]=exp(a[i]);
}
```

6.73s

```
vrda_exp(n,a,b);
```

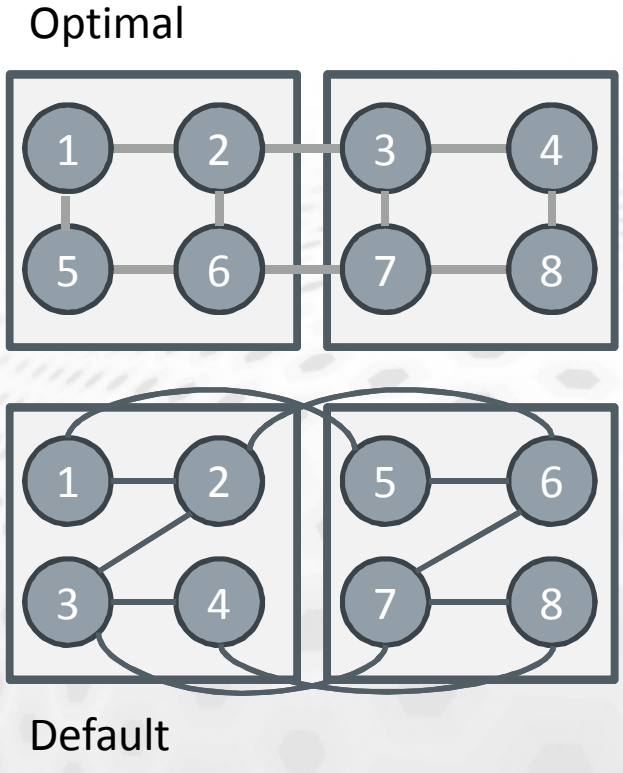
68% speedup

Interesting libraries on the Cray XT/XE

- Iterative Refinement Toolkit (IRT)
  - Subset of LAPACK solvers implemented
  - Uses a mixed-precision approach
  - Close to 2x speedup for well-conditioned problems
- Adaptive Sparse Kernels (CASK)
  - Autoselects compute kernel based on sparsity pattern
  - Integrated with PETSc
- Adaptive FFT (CRAFFT)
  - Autoselects best implementation for a given FFT type

MPI placement

- The layout of MPI processes can be defined
  - On MPICH based libraries (e.g. Cray MPI) the environment variable is `MPICH_RANK_REORDER_METHOD`
- Goal to minimize inter-node communication
  - On Cray, CrayPAT can be used to automate this



Other MPI parameters

- Different MPI implementations have different parameters
- **Protocol limits** are important parameters
  - More about these tomorrow
- Other important parameters vary
  - Buffer sizes, mechanisms for matching unexpected messages, message patterns for collectives etc.
  - E.g. on the Cray XT: consult `'man mpi'`

Summary

- Choice of **compiler** and selection of **flags** may provide a significant performance boost with very little effort
  - See the manual pages for the compilers for full information on flags
- Modern HPC programming builds a lot upon **libraries** - make sure you find the best performing ones
- There are also other levers we can pull: MPI library and other **platform parameters**
- None of the above cannot alleviate a poor algorithm or bad implementation!

Web resources

- Wikipedia article about compiler optimization
  http://en.wikipedia.org/wiki/Compiler_optimization
- How to make the best use of Cray MPI on the XT
  http://www.csc.fi/kurssit/arkisto/aineisto/hpce2-workshop/hpce2-workshop-derose-mpi
- Tool for optimizing runtime parameters of Open MPI
  http://www.open-mpi.org/projects/otpo/
- Porting applications to BlueGene/P
  http://www.fz-juelich.de/jsc/datapool/page/3365/BGP_porting.pdf

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
    % aprun -n 4 ./gol 100 1000 1000
    ```

    Develop a 1000x1000 board for 100 iterations
  - You need to do (once) `module load ImageMagick`
  - Then you can visualise the board development with

    ```
    % convert -delay 40 -geometry 512x512 life_*.pbm life.gif
    % animate life.gif
    ```

Lab session: Performance analysis

- By carrying out performance analysis, find out the reasons why the provided version of the GoL code does not scale
  - Indeed it should scale, it is a simple domain decomposition with thin halos
  - We are going to overcome these tomorrow
- Alternatively, you can carry out the eight-step procedure discussed earlier for your own application!

PART IV: IMPROVING PARALLEL SCALABILITY

Improving parallel scalability

- Unfortunately, quite often achieving better scalability requires algorithm and data-structure level changes
  - However, usually it is possible to do something without touching the big picture
- Review the performance measurements: is the communication bottleneck in
  - Load imbalance?
    - Large Sync times
    - Large differences in MPI_Wait times
  - Just large amount of communication?

Basic considerations

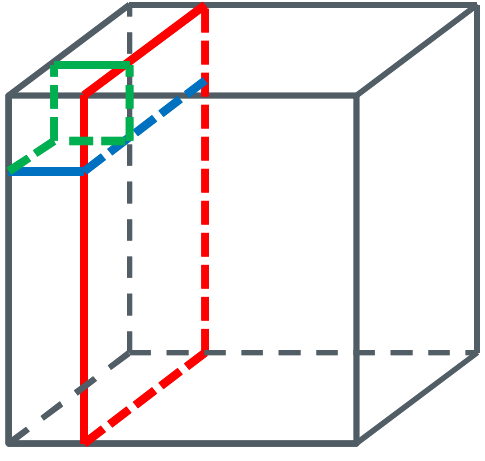
- Message transfer time ∝ latency + message length / bandwidth
  - Latency: Startup for message handling
  - Bandwidth: Network BW / number of messages using the same link
- Bandwidth and latency depend on the used **protocol**
  - **Eager** or **rendezvous**
    - Latency *and* bandwidth higher in rendezvous
    - Eager has an issue with sc. unexpected message buffers
  - The platform will select the protocol basing on the message size, these limits can be adjusted

Basic considerations

- Reduce latency: Send one big message instead of several small messages
- Minimize the amount of bytes send over the interconnect
- Quick tricks
  - Experiment with the rank placement
  - Experiment with the protocol limits and other MPI library parameters

Problem decomposition

- 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 $\propto n^2 * w * 2$

w = halo width
p = number of MPI tasks

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

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

Efficient MPI programming style

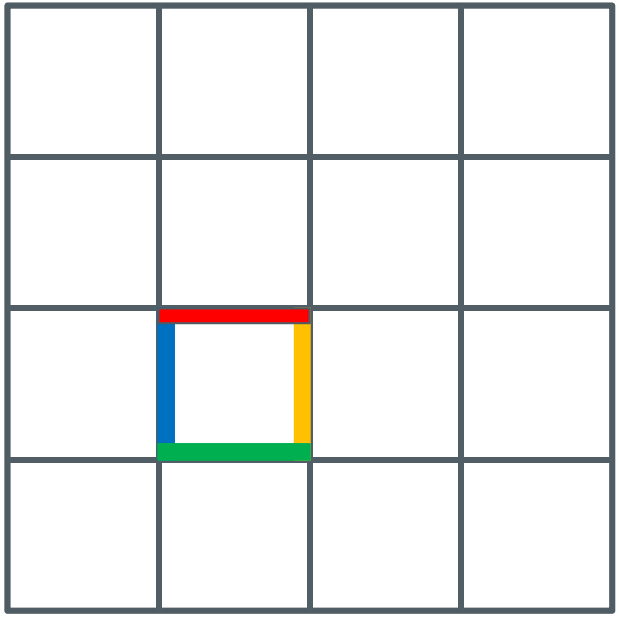
- Use collectives!
  - If a collective call can do it for you, it will outperform all point-to-point constructs
- And use them right
  - See if every all-to-all collective operation needs to be all-to-all rather than one-to-all or all-to-one
    - Often encountered case: convergence checking
  - See if you can live with the basic version of a routine instead of a variable-width version (`MPI_Gatherv`, `MPI_Alltoallv` etc)

Efficient MPI programming style

- Do not ask for the stuff you do not need
  - Do not send dummy messages but use `MPI_PROC_NULL`
  - Do not request for a status if you don’t employ it (but use `MPI_STATUS_IGNORE`)
- Avoid unnecessary memory copies
  - User-defined datatypes are much faster

Point-to-point performance

- Case study: halo exchange in 2D decomposition



Blocking 1

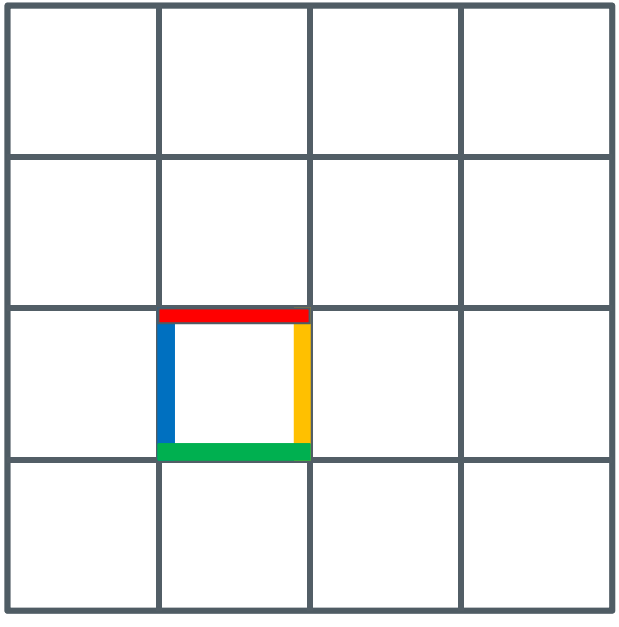
```
Send(to left)
Recv(from left)
Send(to right)
Recv(from right)
Send(to up)
Recv(from up)
Send(to down)
Recv(from down)
```

Blocking 2

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

Point-to-point performance

- Case study: halo exchange in 2D decomposition

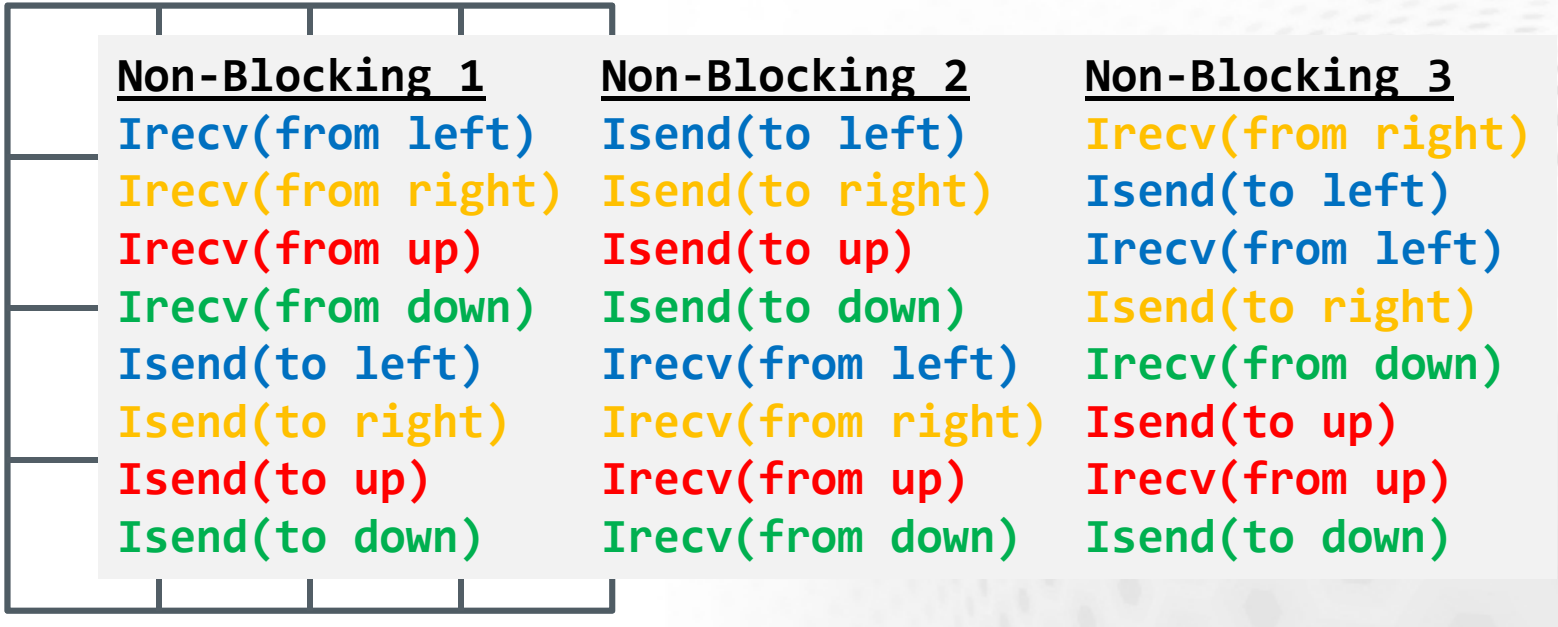


Sendrecv

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

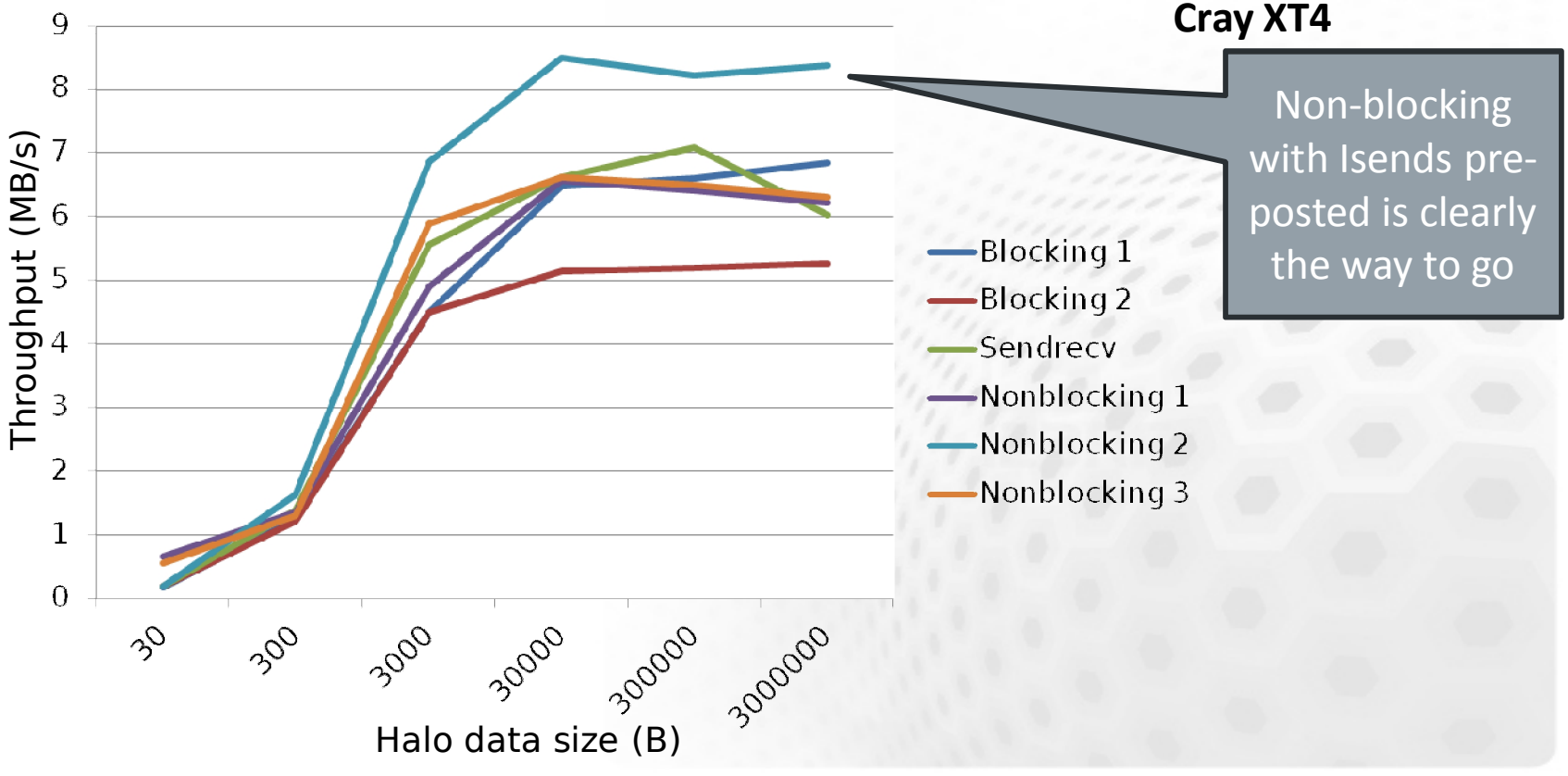
Point-to-point performance

- Case study: halo exchange in 2D decomposition



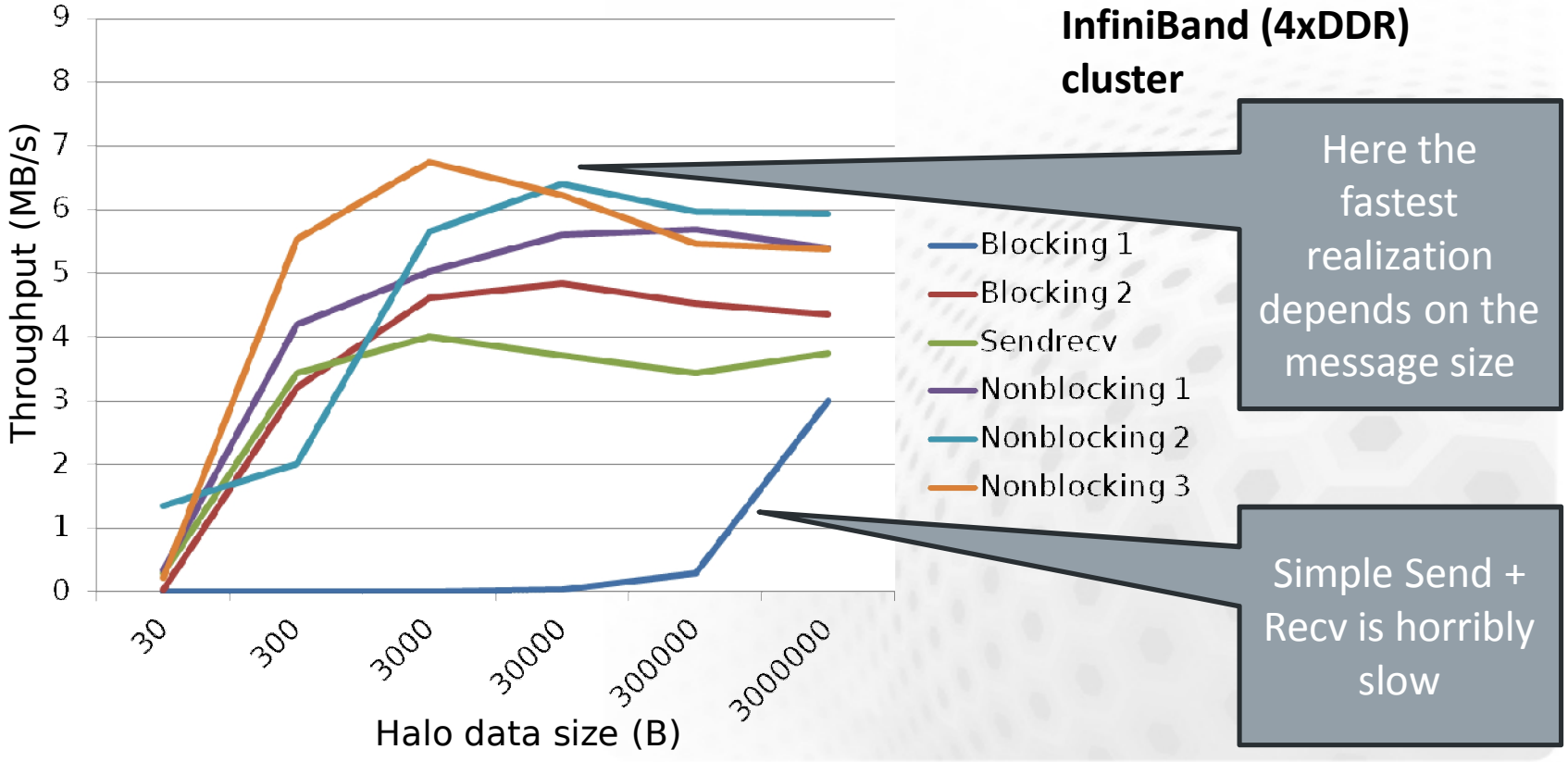
Point-to-point performance

2D halo exchange



Point-to-point performance

2D halo exchange



Overlapping computation and communication

- Compute while the communication takes place
  - Hides communication overhead
- Realization
  - Non-blocking communication
  - Persistent communication
  - Hybrid OpenMP+MPI

Consider hybridization

- Improves load balance
- Decreases replicated data
- Reduces the amount of messages over the interconnect
- One thread per core, one MPI task per CPU
  - Not always the best ratio, however
  - Experiment with other possibilities too

Achieving good I/O performance

- Do not perform I/O from one process only, but **use parallel I/O!**
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call
- Experiment with MPI I/O hints

Giving hints to MPI I/O

- Hints may enable the implementation to optimize performance
- MPI-2 standard defines several hints via the MPI_Info object
  - `MPI_INFO_NULL` : no info
  - Functions MPI_Info_create and MPI_Info_set allow one to create and set hints
- Effect of hints on performance is implementation and application dependent

Giving hints to MPI I/O

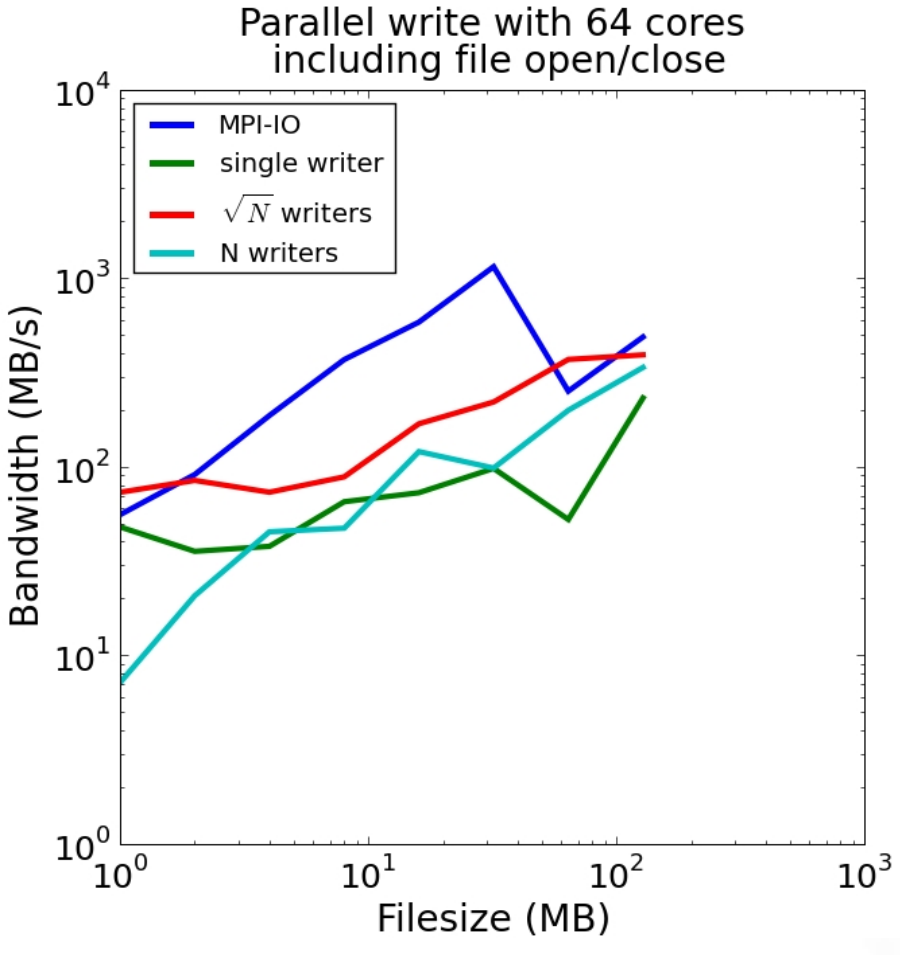
- For example, Cray XT systems support the following hints
  `striping_factor, striping_unit, direct_io, romio_cb_read, romio_cb_write, cb_buffer_size, cb_nodes, cb_config_list, romio_no_indep_rw, romio_ds_read, ind_rd_buffer_size, ind_wr_buffer_size`
- Consult ”man mpi” for their meaning and default values

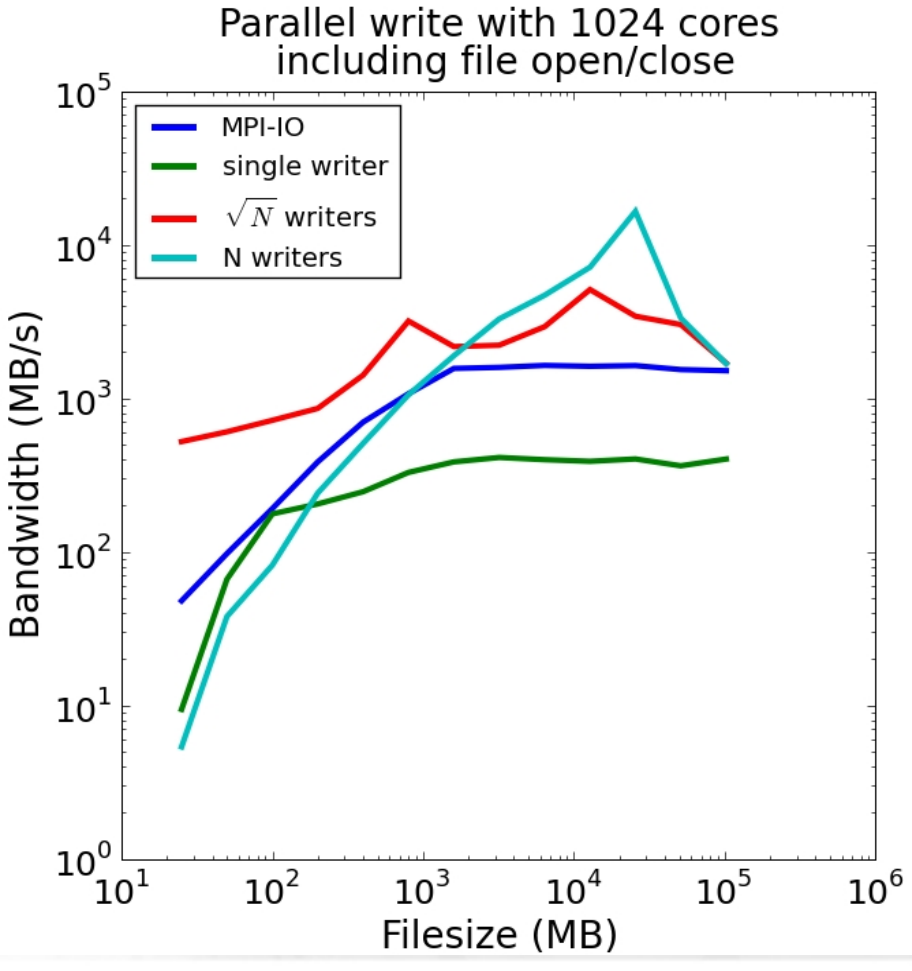
Giving hints to MPI I/O

- Some implementations allow setting of hints via environment variables
  - e.g. `MPICH_MPIIO_HINTS`
  - Example: for file “test.dat”, in collective I/O aggregate data to 32 nodes

```
export MPICH_MPIIO_HINTS=”test.dat:cb_nodes=32”
```

Parallel I/O performance (Cray XT)





Concluding remarks

- Apply the scientific method to performance engineering: make hypotheses and measurements!
- Scaling up is the most important consideration in HPC
  - Find the optimal decomposition
  - Optimize MPI
  - Optimize for message sizes in the bottleneck routines
  - Overlap computation & communication
  - Hybridize the code
- Mind your I/O!

Lab session: Performance optimization

- Following the optimization flow chart, optimize the GoL program
  1. Improve scalability
     - For your convenience, versions employing nonblocking communication as well as parallel I/O of the program are provided – see the makefile (type `make help`)
  2. Optimize the single core hotspots
  3. Try to do ”optimal porting” tricks for the GoL program

Alternatively, you can work with your own application!