Performance Measurement & Presentation ("Fooling the Masses")

CPSC 424/524
Lecture #12
December 5, 2018

Credit: These slides are based on slides from Prof. Gerhard Wellein, who developed them for use in HPC programming courses at University of Erlangen, Germany

Performance: Why thoroughly measure and report it?

- **Determine which computer is best suited for a given (set of) application(s)?**
  - Gaming PC or Atom based Laptop?
  - Cluster or fat server? Fast CPU? Intel or AMD or may be GPU???
  - Which applications? Which input/data sets?

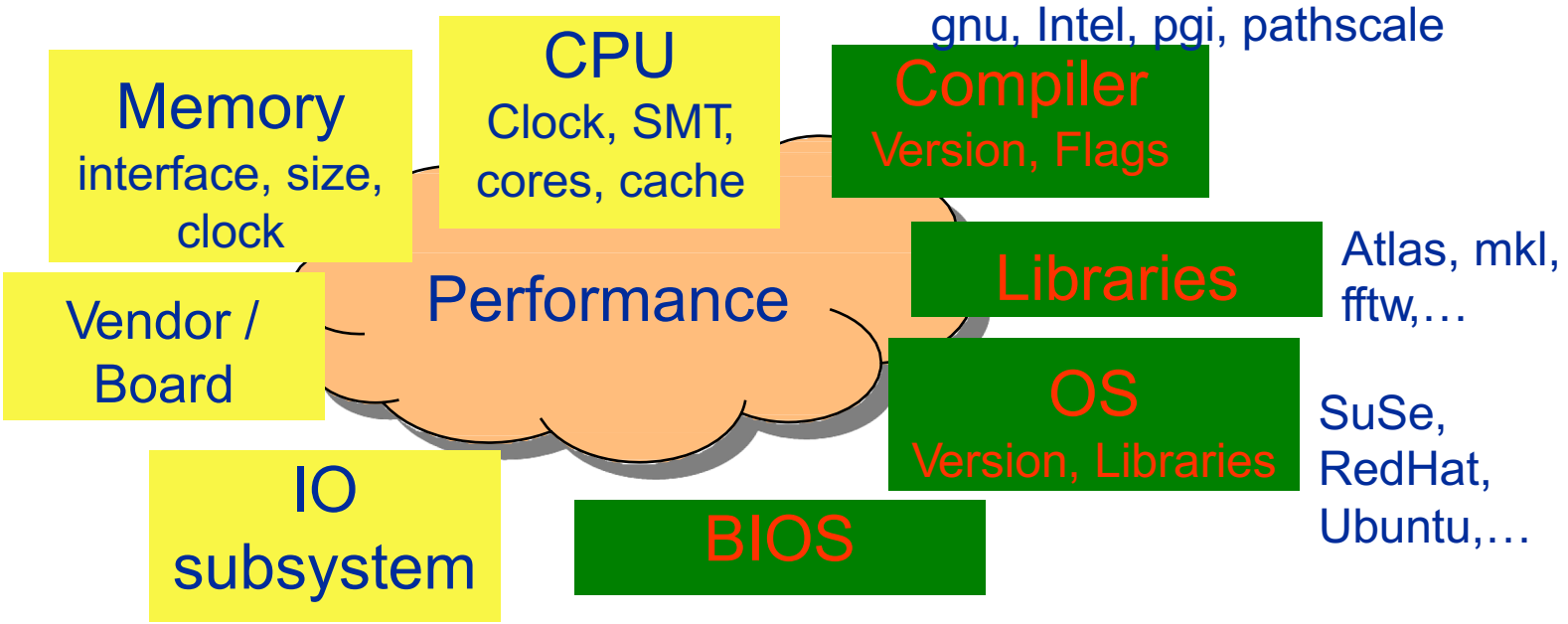
- **Present result of new optimization / implementation / parallelization strategy to others**
  - Results need to be interpreted and potentially reproduced by external people
  - Compare with other / previous work

- **Determine capabilities for individual parts of the computer with (simple) kernels**
  - Data transfer capabilities
  - IO / computational capabilities
  - Often required to guide optimization strategies → Performance Modeling

Courtesy of Prof. G. Wellein, U. of Erlangen

Performance: Impact factors

- **For a given code/problem performance may be influenced by many factors**



- **Code performance may also be influenced by**
  - programming language used (Java ←→ fortran77)
  - methods/algorithms/solver
  - serial / parallel execution
  - specific code implementation (untuned, … , highly tuned/optimized)

Courtesy of Prof. G. Wellein, U. of Erlangen

1991

- ***David H. Bailey***

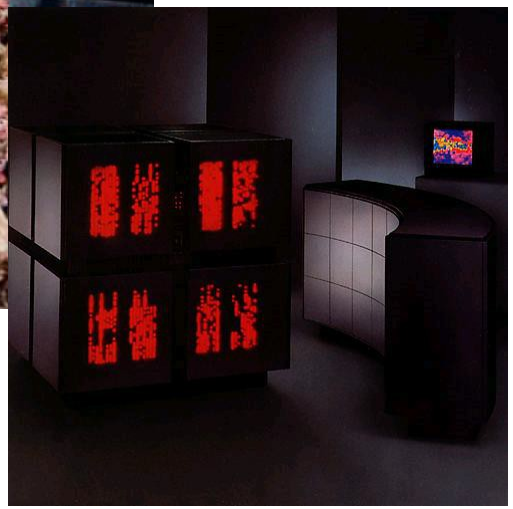
  Supercomputing Review, August 1991, p. 54-55

  "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers"

1. Quote only 32-bit performance results, not 64-bit results.
2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
3. Quietly employ assembly code and other low-level language constructs.
4. Scale up the problem size with the number of processors, but omit any mention of this fact. (Gustafson's Law; Weak Scalability.)
5. Quote small-scale performance results projected to a full system.
6. Compare your highly-tuned code results against scalar, unoptimized code on Crays.
7. When direct run time comparisons are required, compare with an old code on an obsolete system.
8. If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.
9. Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.
10. Mutilate the algorithm used in the parallel implementation to match the architecture.
11. Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment.
12. If all else fails, show pretty pictures and animated videos, and don't talk about performance.

What supercomputing was like in 1991

If you were plowing a field, which would you rather use?



Two strong oxen or 1024 chickens?

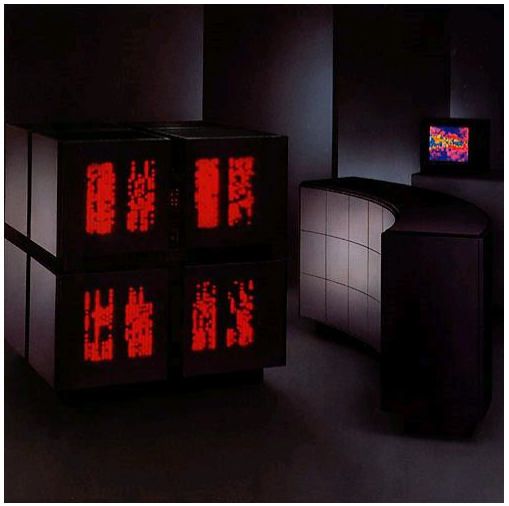
(Attributed to Seymour Cray)

What supercomputing was like in 1991

Strong I/O facilities

32-bit vs. 64-bit FP arithmetic

SIMD/MIMD parallelism



Vectorization
(the real thing, not the SSE/AVX c**p)

System-specific optimizations

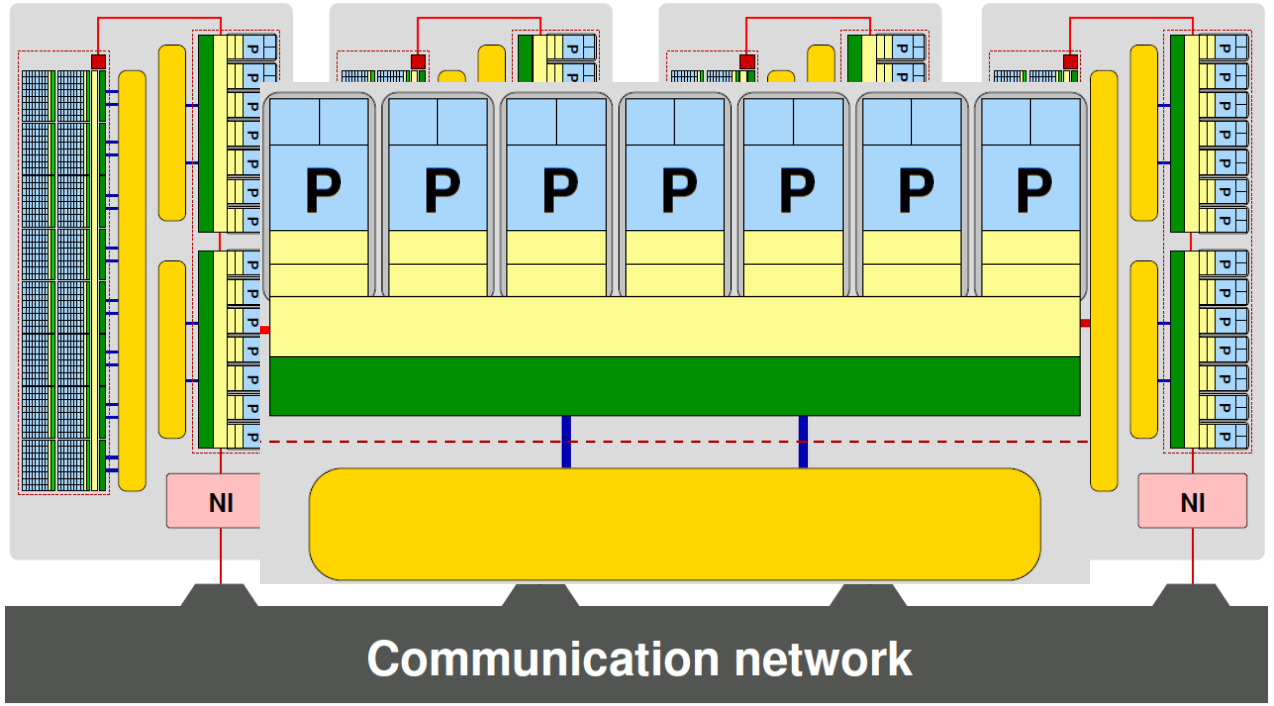
No parallelization standards

Courtesy of Prof. G. Wellein, U. of Erlangen

Today we have…

Multicore processors

with shared/separate caches, shared data paths



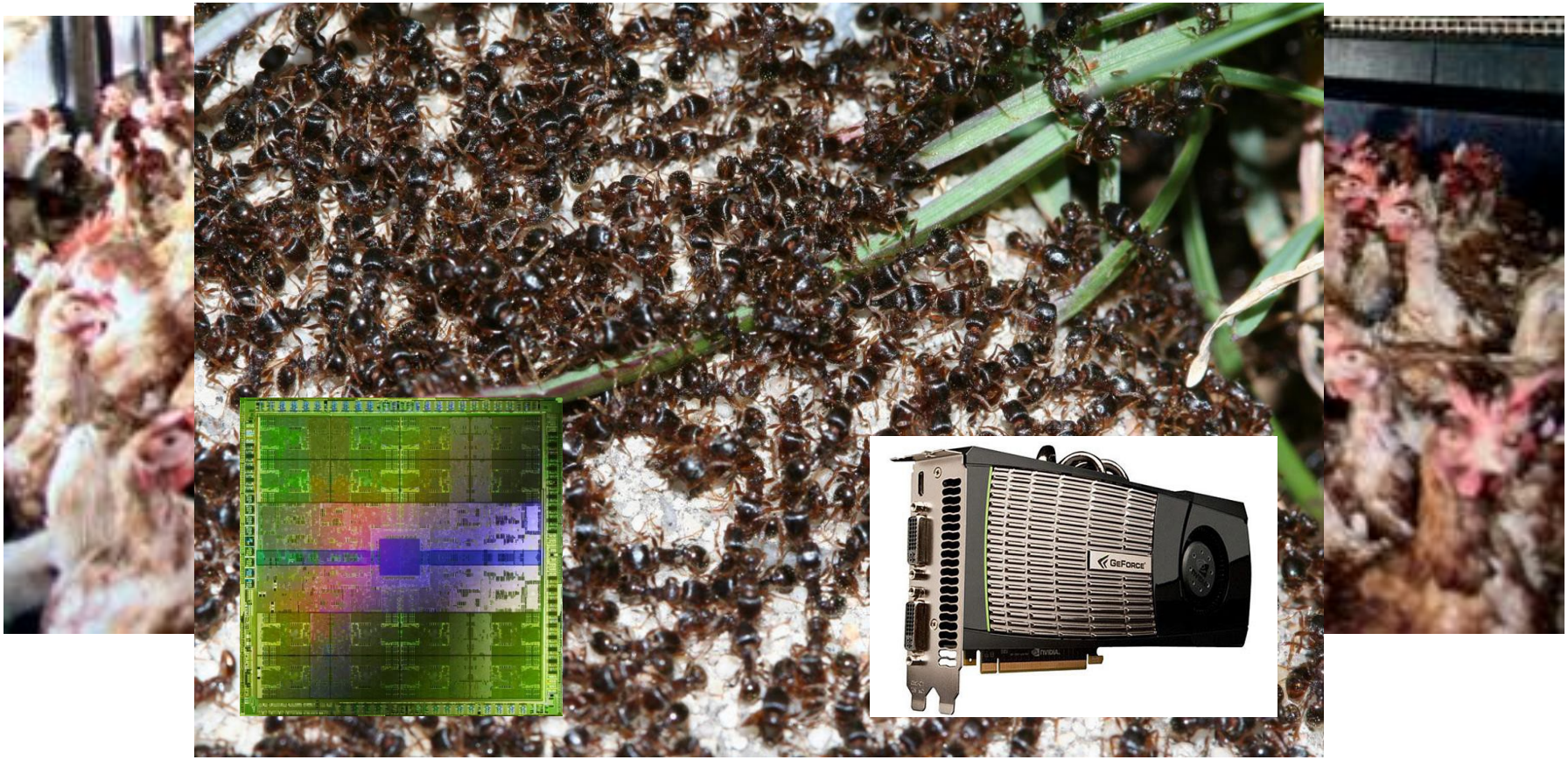
Hybrid, hierarchical systems

with multi-socket, multi-core, ccNUMA, accelerators, heterogeneous networks

Today we have…

Ants all over the place

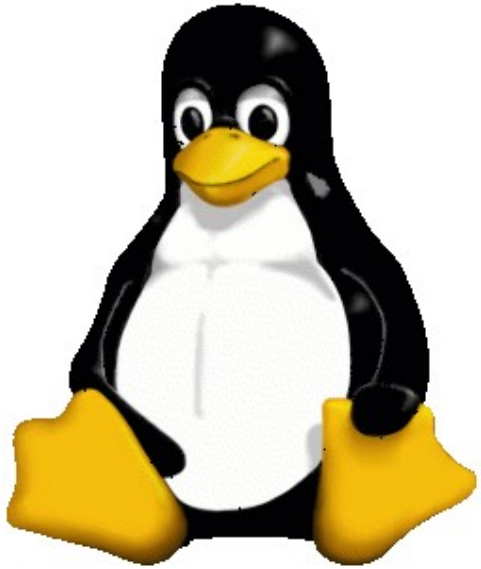
GPUs, Xeon Phi,...



Today we have…

Commodity everywhere

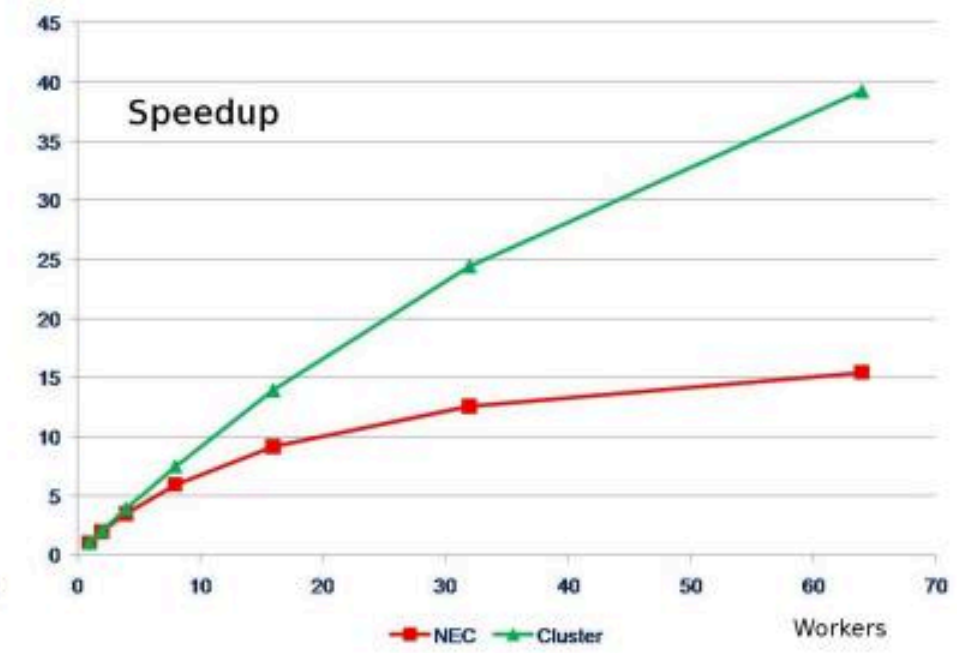
x86-type processors, cost-effective interconnects, GPUs, GNU/Linux



The landscape of High Performance Computing and the way we think about HPC has changed over the last 25 years, and we need an update!

Still, many of Bailey's points are valid without change

Report Speedup, Not Absolute Performance



Courtesy of Prof. G. Wellein, U. of Erlangen

Better: Report Speedup for a "Slowed-Down" Code

Basic Principle when reporting speedups:

If there is a significant serial part, artificially slow down the parallelizable part to reduce the relative impact of the serial part. (For example, insert some sleeps or other delays.)

Corollaries:

1. Do not use high compiler optimization levels or the latest compiler versions.
2. If scalability is still bad, parallelize some short loops with OpenMP. That way you can get some extra bonus for a scalable hybrid code.

If someone asks for time to solution, answer that if you had a bigger machine, you could get the solution as fast as you want. This is of course due to the superior scalability of your code.

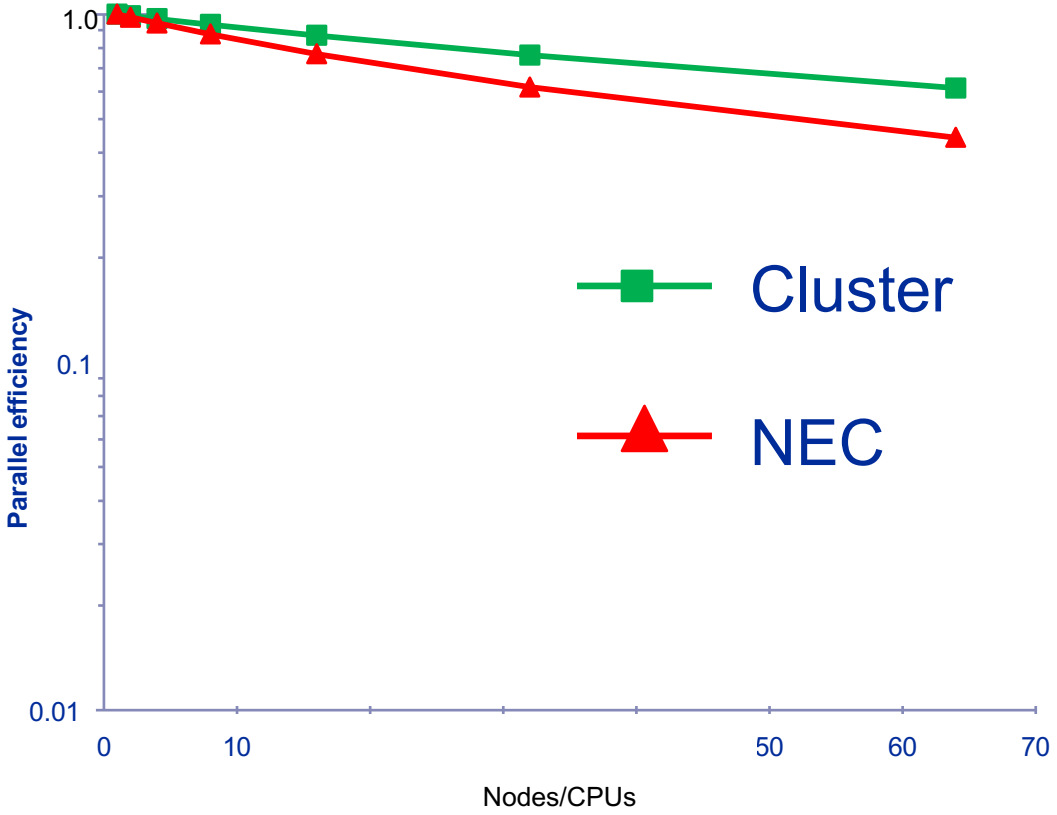
Courtesy of Prof. G. Wellein, U. of Erlangen

Divert attention from critical details

Compare different systems by showing the log of parallel efficiency vs. CPU count

Unusual ways of putting data together will surprise and confuse your audience

Remember: Legends can be any size you like!

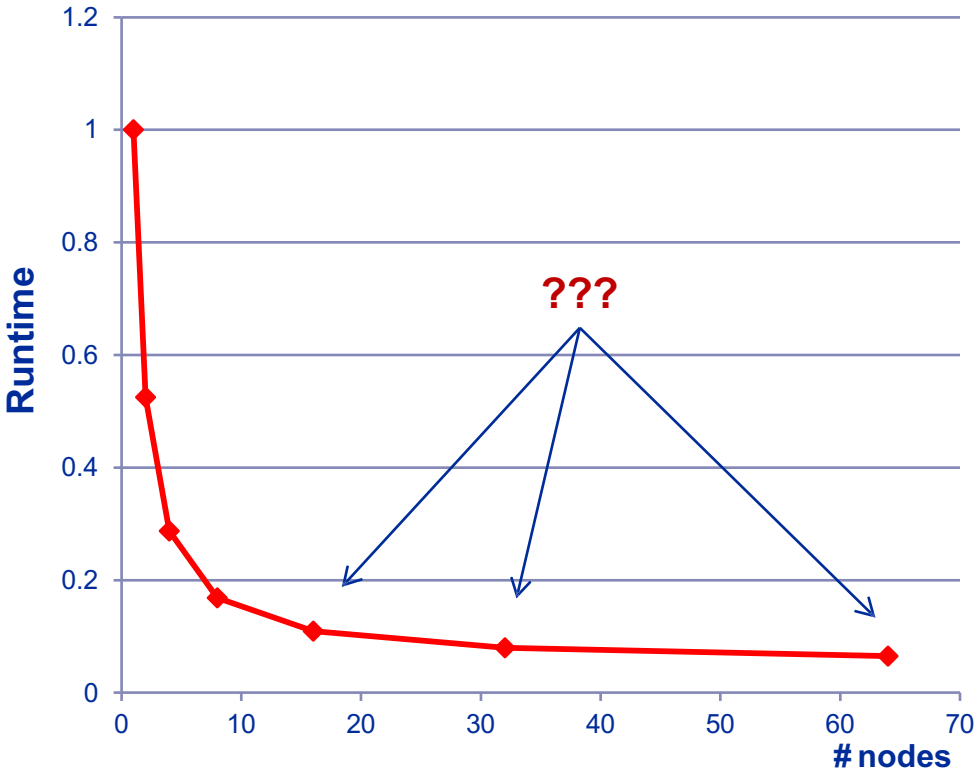


Instead of performance tables, plot runtime vs. CPU count

Very, very popular indeed!

Nobody will be able to tell whether your code actually scales

Caveat: Make sure to use a linear y scale!

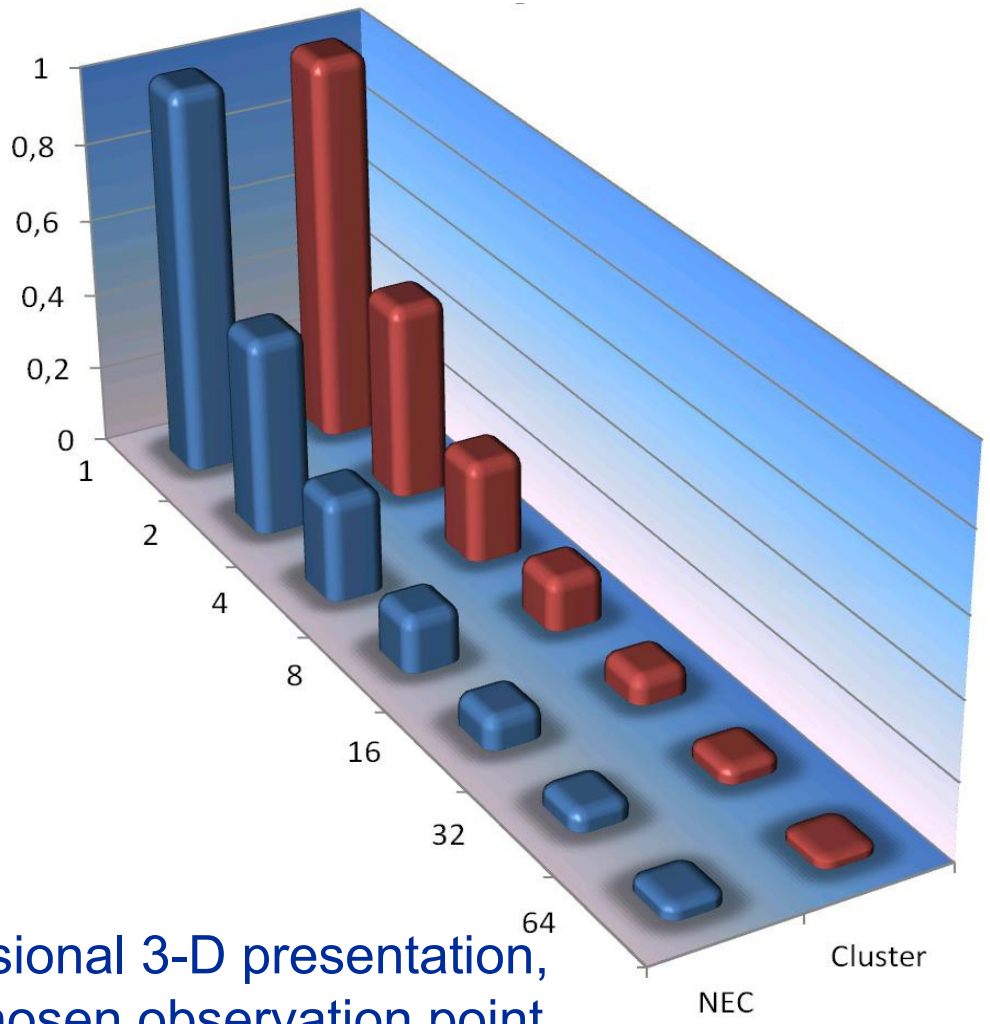


Courtesy of Prof. G. Wellein, U. of Erlangen

Do not show “hard” (precise) numbers

Even better: Divert attention with a fancy plot

Nobody will be able to tell anything concrete about your code’s performance…



Corollaries:

CPU time per core is even better because it omits most overheads…

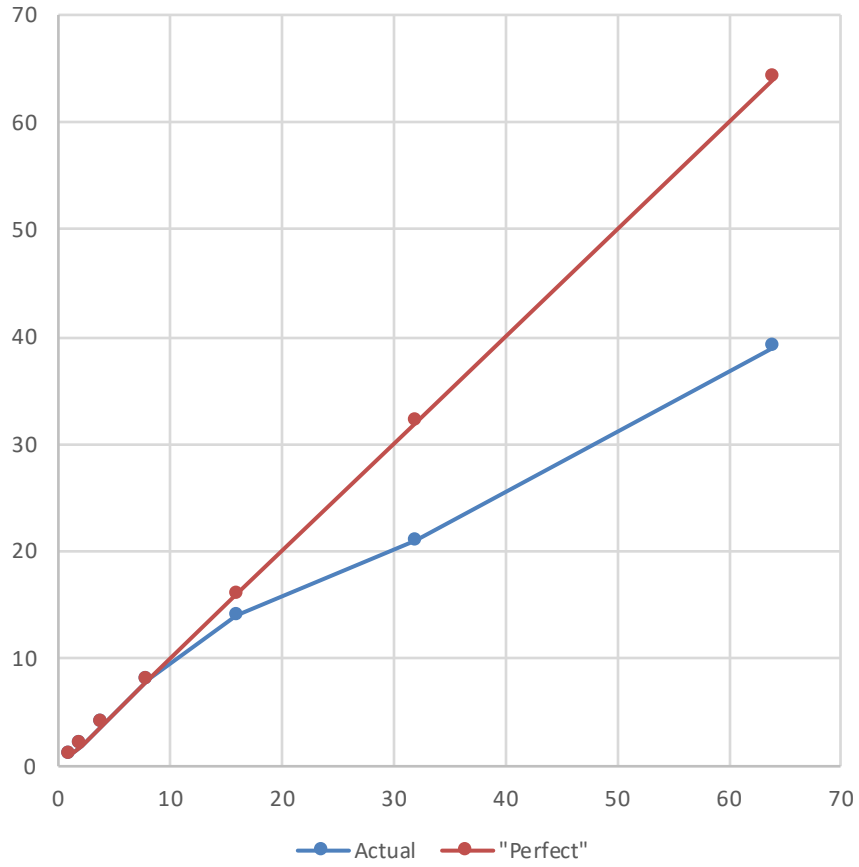
It’s essential to use a professional 3-D presentation, preferably with a carefully-chosen observation point.

Courtesy of Prof. G. Wellein, U. of Erlangen

Choose your scale carefully (Obfuscation, Part I)

If scalability/speedup doesn't look good enough, use a suitable scale to drive your point home. Everything looks OK if you plot it the right way!

1. Linear plot: reveals bad scaling, strange things at N=16

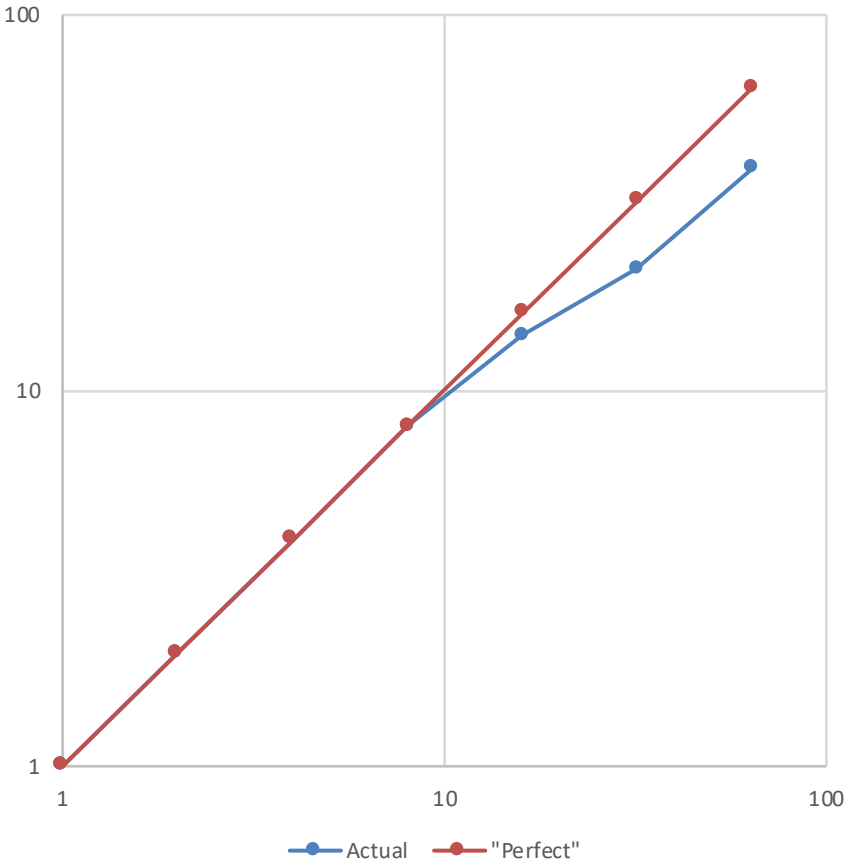


Choose your scale carefully (Obfuscation, Part I)

If scalability/speedup doesn't look good enough, use a suitable scale to drive your point home. Everything looks OK if you plot it the right way!

1. Linear plot: reveals bad scaling, strange things at N=16

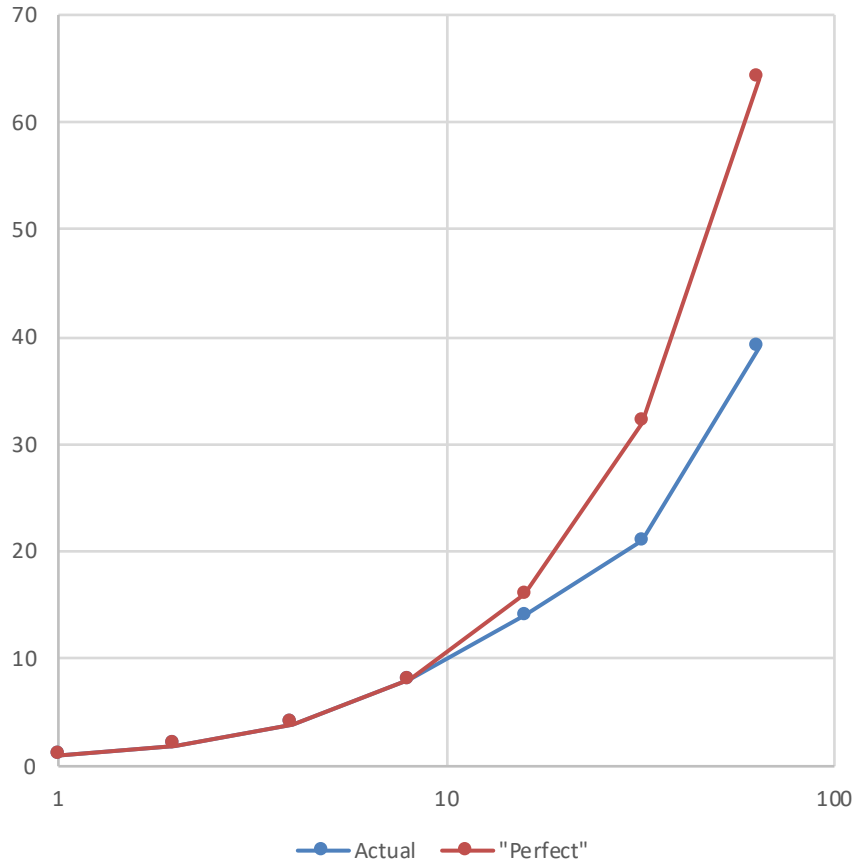
2. Log-log plot: scaling looks better, but still the N=16 problem



Choose your scale carefully (Obfuscation, Part I)

If scalability/speedup doesn't look good enough, use a suitable scale to drive your point home. Everything looks OK if you plot it the right way!

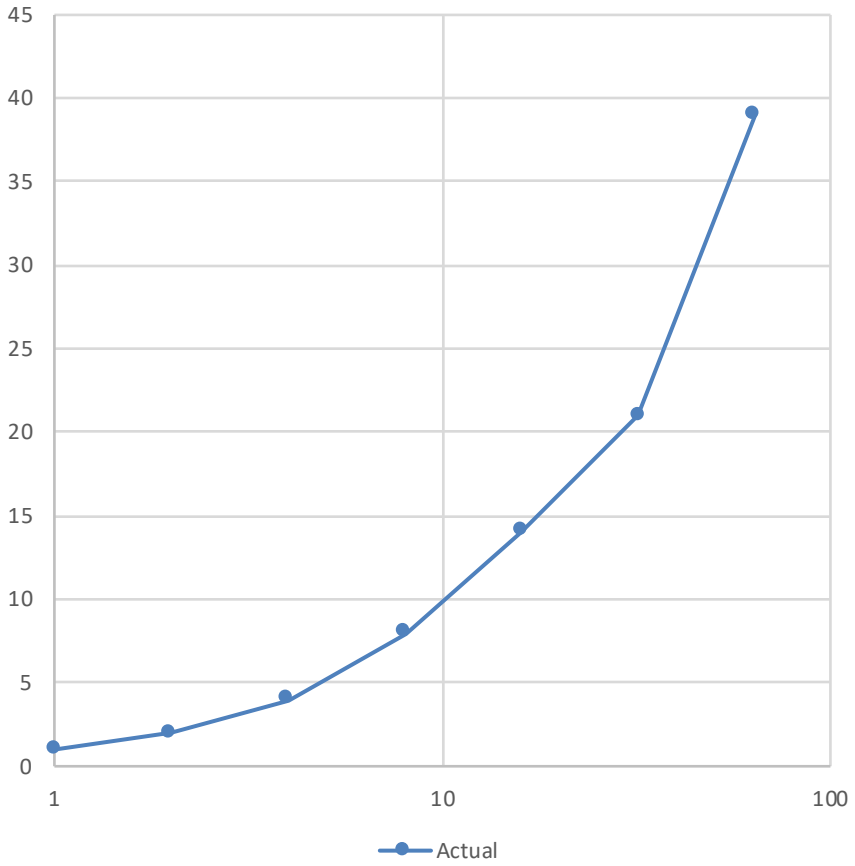
1. Linear plot: reveals bad scaling, strange things at N=16
2. Log-log plot: scaling looks better, but still the N=16 problem
3. Log-linear plot: N=16 problem gone! (Who can tell!)



Choose your scale carefully (Obfuscation, Part I)

If scalability/speedup doesn't look good enough, use a suitable scale to drive your point home. Everything looks OK if you plot it the right way!

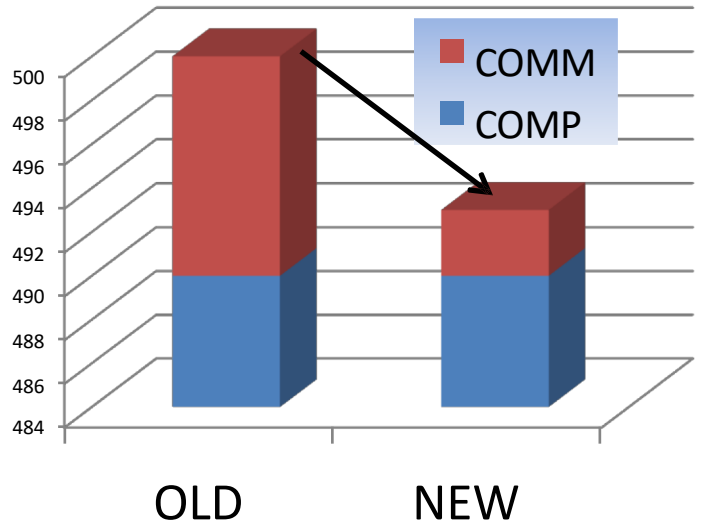
1. Linear plot: reveals bad scaling, strange things at N=16
2. Log-log plot: scaling looks better, but still the N=16 problem
3. Log-linear plot: N=16 problem gone! (Who can tell!)
4. … and remove the ideal scaling line to make it perfect!

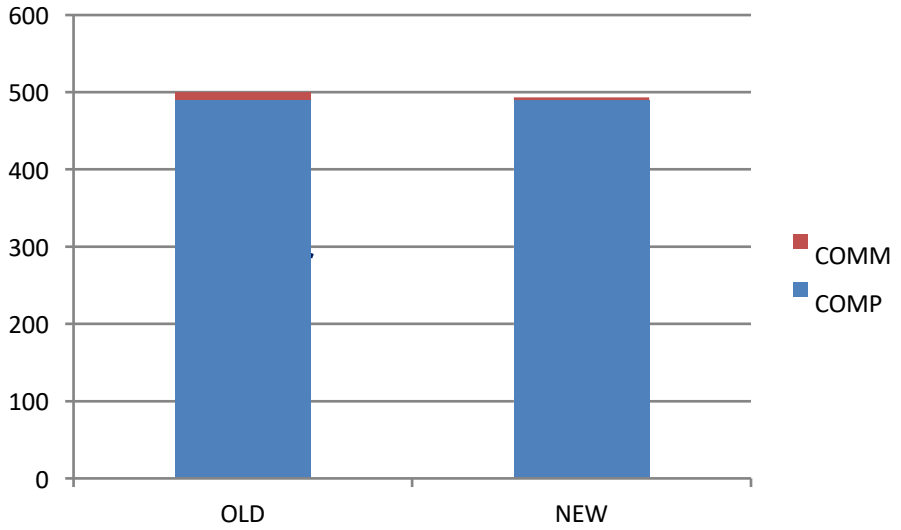


Careful Focus (Obfuscation, Part II)

Keep graphs simple. And focus on the data region that is most supportive of your point.

"Fig. 3 demonstrates the benefit of our new communication scheme which reduces overall communication volume by 70%"





Adding a strong/bold arrow further emphasizes the importance of your achievement and 3D bars really look professional.

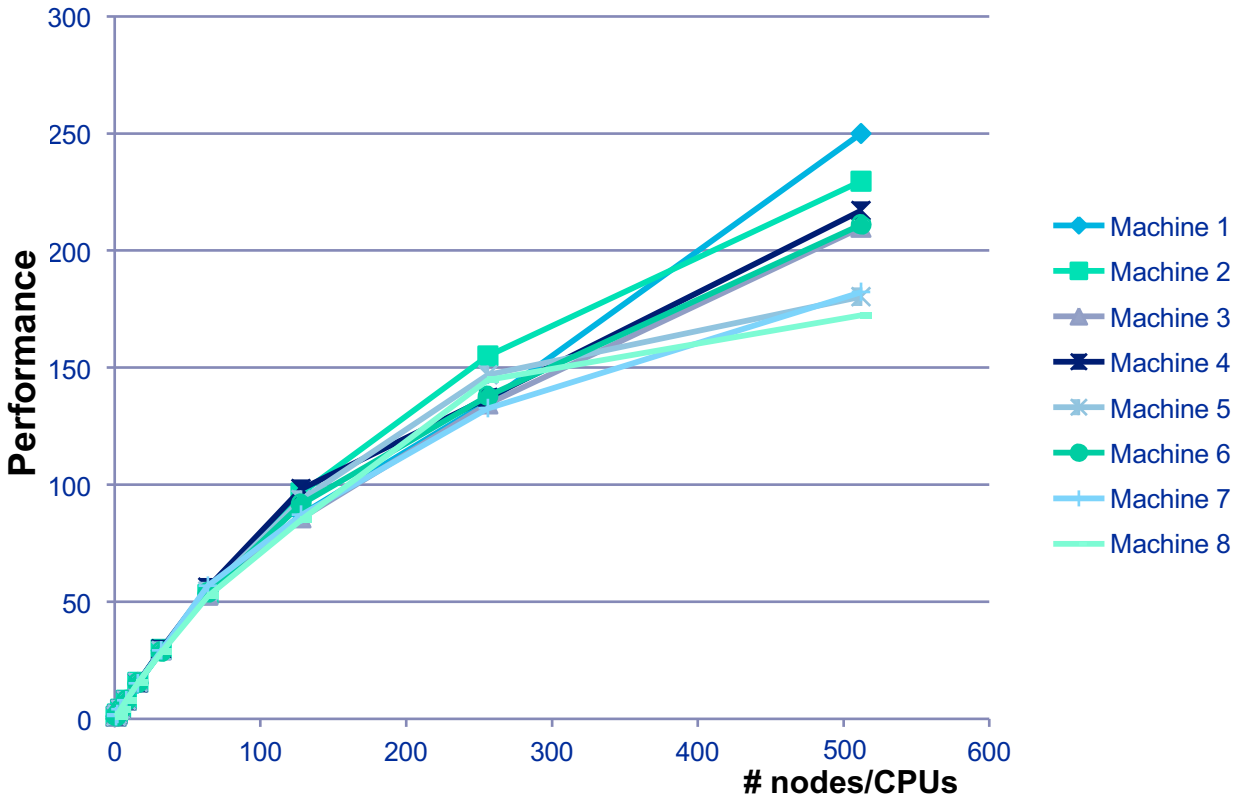
Overwhelm with data (Obfuscation, Part III)

Performance modeling is for wimps. Show real data. Plenty. And then some.

Don't try to make sense of your data by fitting it to a performance model.

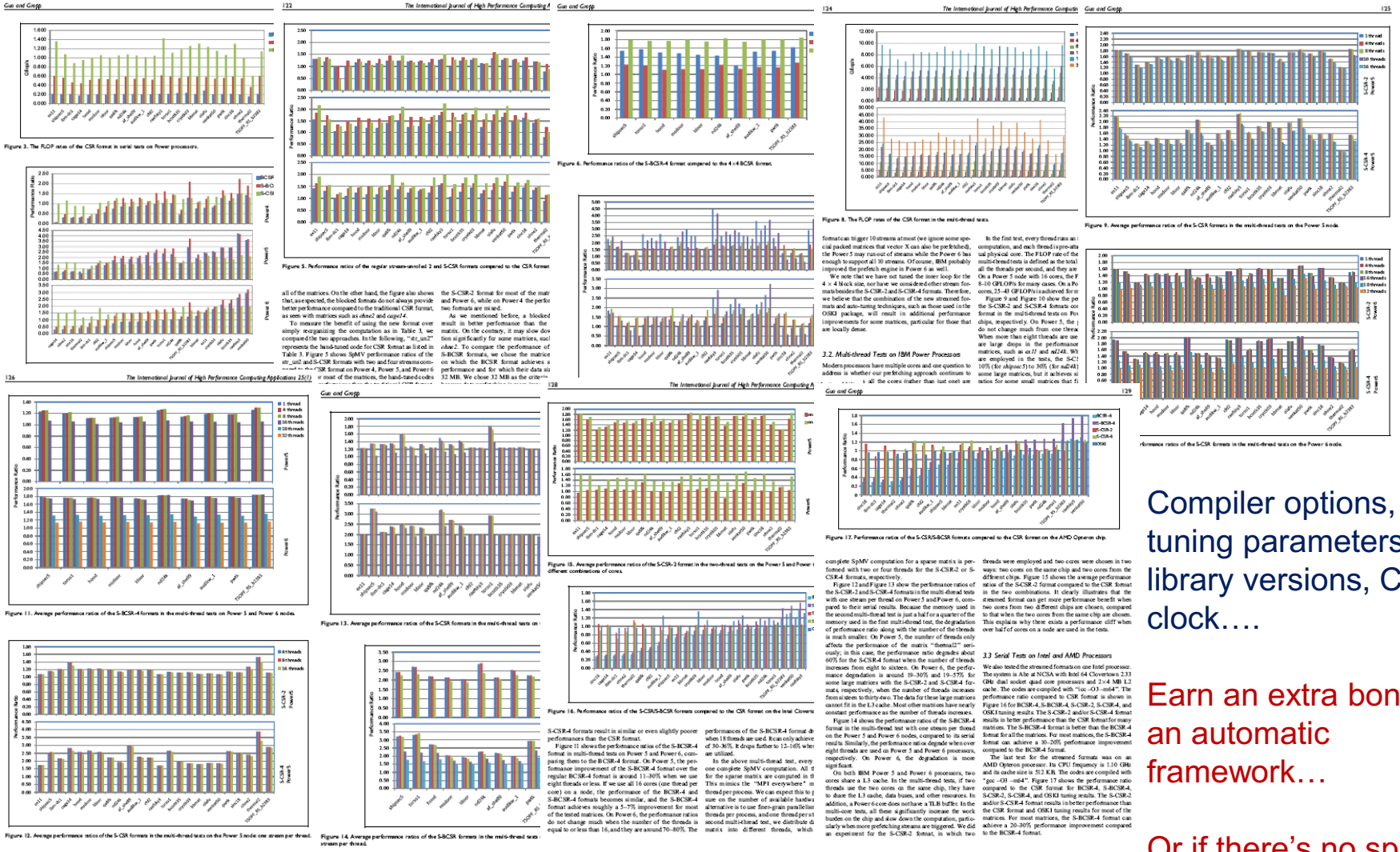
Instead, show at least 8 graphs per plot, all in bright pastel colors, with different symbols.

If nasty questions pop up, say your code is so complex that no model can describe it.



Courtesy of Prof. G. Wellein, U. of Erlangen

Overwhelm with data (Obfuscation, Part IV)



Compiler options, tuning parameters, library versions, CPU clock….

Earn an extra bonus for an automatic framework…

Or if there's no space for explanations!

Courtesy of Prof. G. Wellein, U. of Erlangen

Make Use of a Modern, Automated Data Science Framework

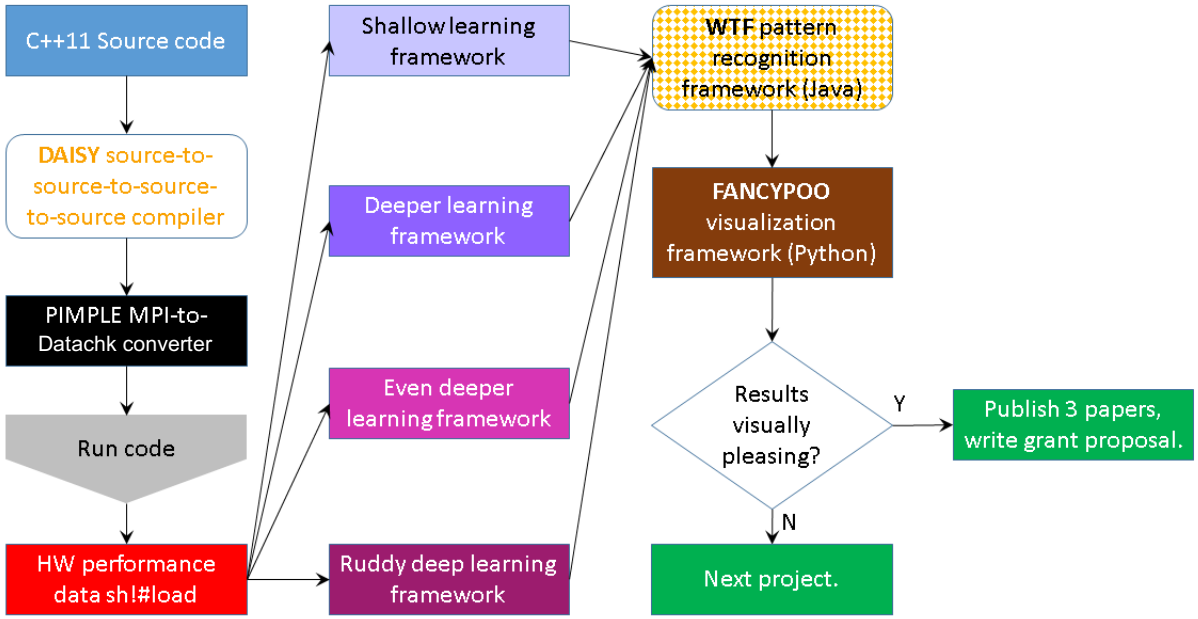
Computers are for automating tasks. Why not automate the process of performance analysis?

Automate everything. Use as many tools as possible and plug them together.

Use machine learning. Always.

Use at least three different languages.

Give the whole thing a catchy name.



Courtesy of Prof. G. Wellein, U. of Erlangen

Fabricate a usefully slow baseline - I

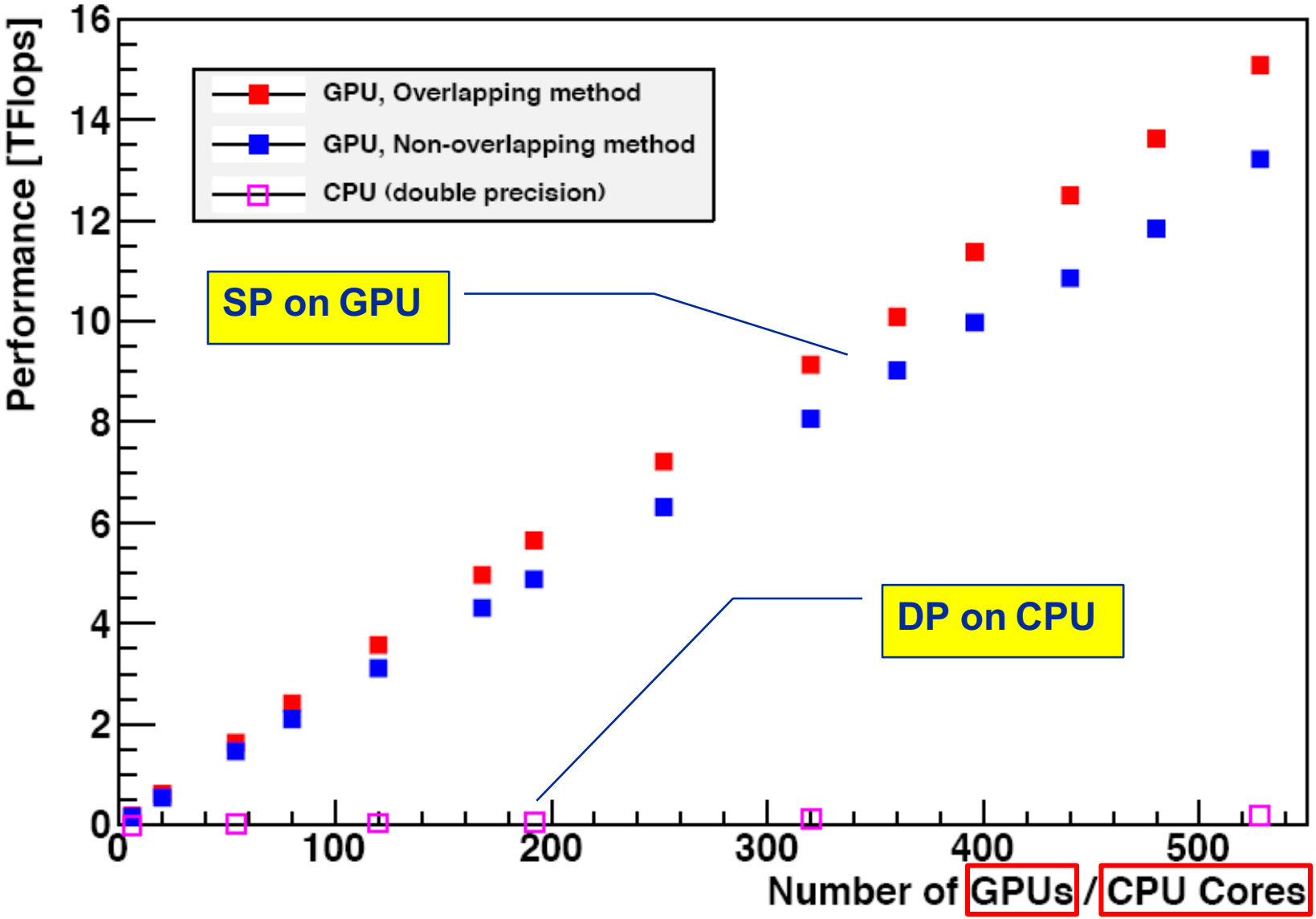
SC10 best student paper finalist

An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code

Takashi Shimokawabe*, Takayuki Aoki*‡,
Chiashi Muroi†, Junichi Ishida†, Kohei Kawano†,
Toshio Endo*‡, Akira Nukada*‡, Naoya Maruyama*‡, and Satoshi Matsuoka*‡§
* Tokyo Institute of Technology
† Japan Meteorological Agency
‡ Japan Science and Technology Agency, CREST
§ National Institute of Informatics
shimokawabe@sim.gsic.titech.ac.jp, taoki@gsic.titech.ac.jp

Courtesy of Prof. G. Wellein, U. of Erlangen

Fabricate a usefully slow baseline - II



Pick the right metric - I

Quote GFlops, MIps, or any other ~~irrelevant~~ interesting metric instead of (inverse) time to solution.

Flops are so cool, it makes you cry:

```
for(i=0; i<N; ++i)
  for(j=0; j<N; ++j)
    b[i][j] = 0.25*(a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1]);
```

```
for(i=0; i<N; ++i)
  for(j=0; j<N; ++j)
    b[i][j] = 0.25*a[i-1][j]+0.25*a[i+1][j]
              +0.25*a[i][j-1]+0.25*a[i][j+1];
```

"Floptimization"

Courtesy of Prof. G. Wellein, U. of Erlangen

Pick the right metric - II: Pure metric may fool…

- **"My vector update code runs at 2,000 MFlop/s on a 2GHz processor!**
- **Great – isn't it?**

```
for(i=0; i<n; i++)
{
    a[i]= 3.0*c0+c1*c2 +c3*c4*a[i] -1.d0 *a[i];
}
```

→ #FLOP = 8 * n

Same execution time but…

… but my MFlop/s rate is only ¼!

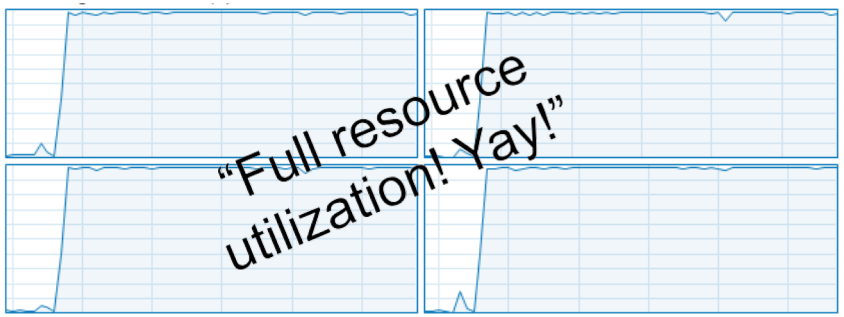
```
d0 = 3.0*c0+c1*c2;
d1 = c3*c4-1.d0;

for(i=0; i<n; i++)
{
    a[i]= d0 + d1*a[i];
}
```

→ #FLOP = 2* n + 5

Courtesy of Prof. G. Wellein, U. of Erlangen

Pick the right metric - III

Redefine "performance" appropriately:



"Our new algorithm shows a factor of 5 fewer cache misses compared to the baseline."

"Lower Watts/Flop!"

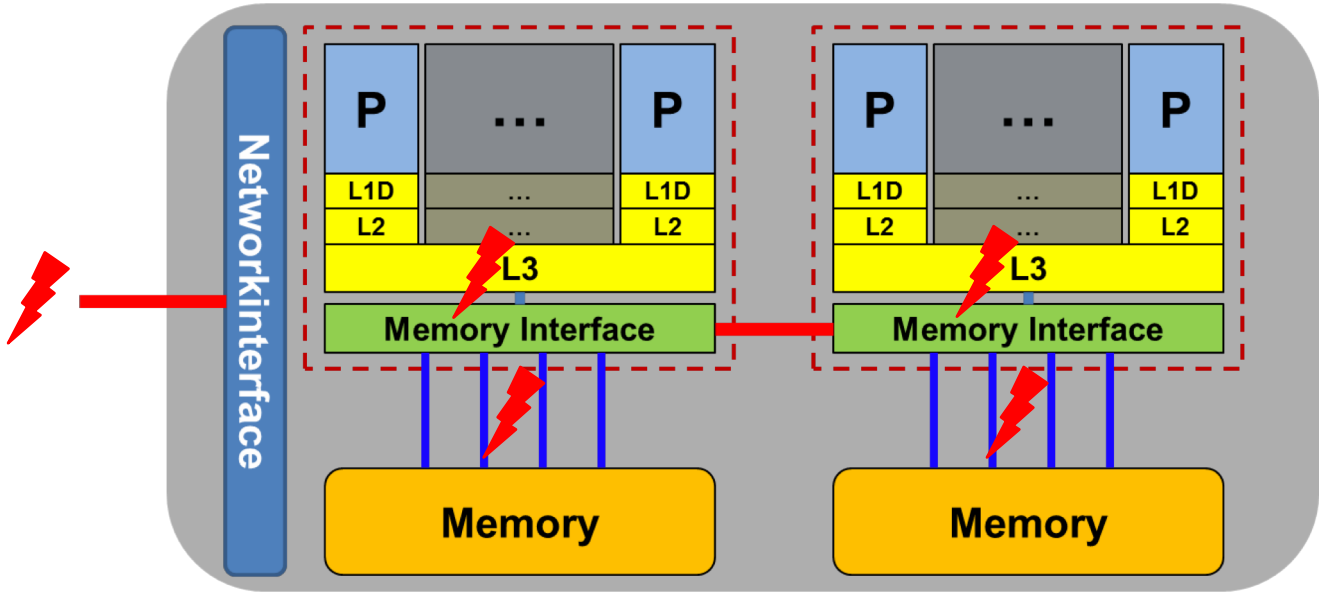
"24% less coding person-hours per MPI call!"

Courtesy of Prof. G. Wellein, U. of Erlangen

If you're cornered: Blame it all on "contention"

And throw in some "architectural complexity".

They will understand and nod knowingly.



Corollary:

Depending on the audience,
"bad prefetching efficiency" may work just as well.

Courtesy of Prof. G. Wellein, U. of Erlangen

If you're cornered #2: Blame it on *some* technical issue

"Technical-detail-not-under-my-control":

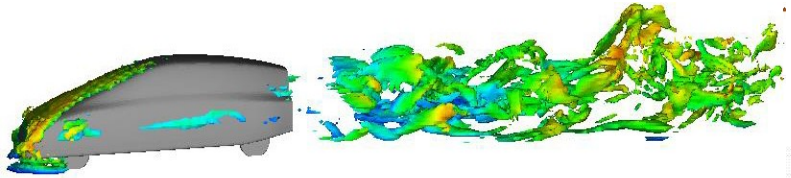
- **Stupid compilers**: "*Our version of the code shows slightly worse single-thread performance, which is presumably due to the limited optimization capabilities of the compiler.*"
- **Out-of-order execution (or lack thereof)**: "*Processor A shows better performance than processor B possibly because of A's superior out-of-order processing capabilities.*"
- **L1 instruction cache misses**: "*As shown in Table 1, our optimized code version B is faster because it has 20% fewer L1 instruction cache misses than version A.*"
- **TLB misses**: *"Performance shows a gradual breakdown with growing problem size. This may be caused by excessive penalties due to TLB misses."*
- **Bad prefetching**: "*Performance does not scale beyond four cores on a socket. We attribute this to problems with the prefetching hardware.*"
- **Bank conflicts**: "*Processor X has only* [sic!] *eight cache banks, which may explain the large fluctuations in performance vs. problem size.*"
- **Transient network errors**: "*In contrast to other high-performance networks such as Cray's Gemini, InfiniBand does not have link-level error detection, which impacts the scalability of our highly parallel code.*"
- **OS jitter**: "*Beyond eight nodes our implementation essentially stops scaling. Since the cluster runs vanilla* [insert your dearly hated distro here] *Linux OS images, operating system noise ("OS jitter") is the likely cause for this failure.*"

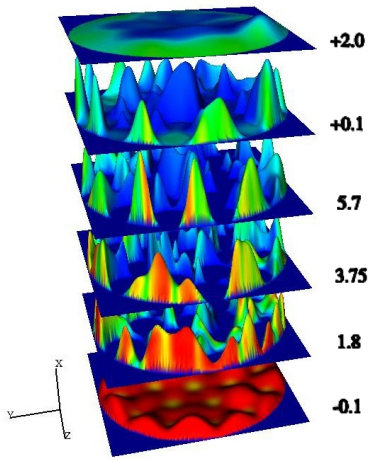
Courtesy of Prof. G. Wellein, U. of Erlangen

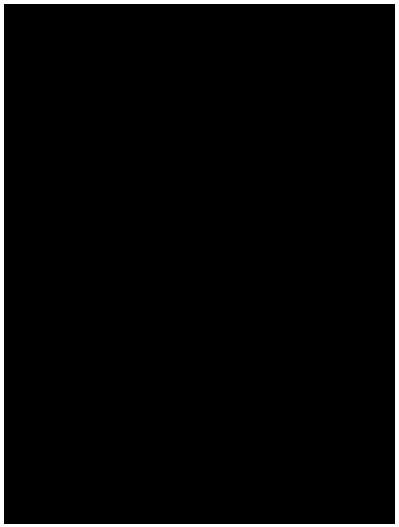
And if all else fails …

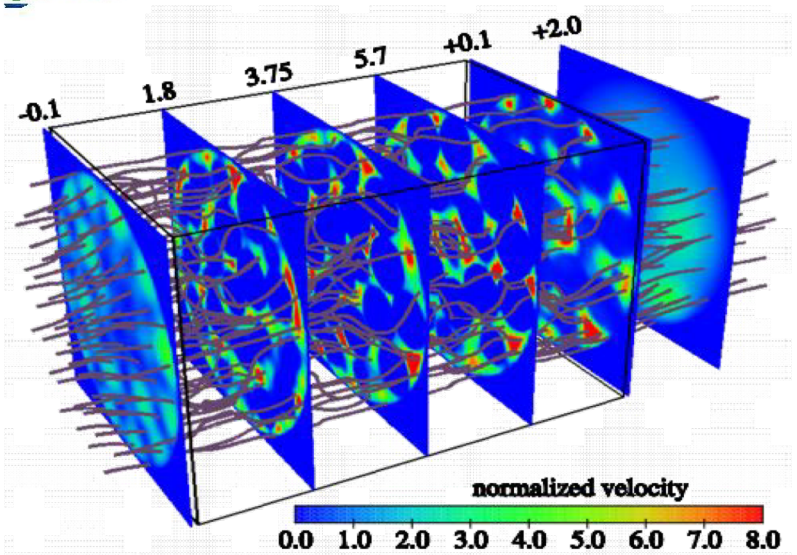
Show plenty of pretty pictures and animated videos, and don't talk about performance.

In five decades of supercomputing, this was always the best-selling plan, and it will (probably) stay that way forever.









Courtesy of Prof. G. Wellein, U. of Erlangen

Read more at:

http://tiny.cc/foolingthemasses

Courtesy of Prof. G. Wellein, U. of Erlangen

How to (REALLY) present performance data

1. **Analyze your data carefully on consistency and rationality - do not speculate or ignore data you do not like.**
2. **Check for trivial dependencies – Compiler version, different machines.**
3. **Determine a maximum / minimum performance level and check your measurements for consistency.**
4. **Try to correlate your performance with hardware features and performance.** (Don't claim that your code is memory bound if it only uses 10% of the available main memory bandwidth.)
5. **Present your data in ways that allow others to reproduce or understand it (in terms of performance models).** (Try to convince your friends/colleagues first!)
6. **Do not try to hide awkward data, but present them clearly, even if they do not look outstanding. Sometimes, even suboptimal results can be important in practice (e.g. a 2x performance gain on a critical application---even if 8 or 16 processors are needed so that parallel efficiency is poor).**

Performance: Report for professional benchmark

Hardware		Software	
CPU Name:	Intel Xeon X5670	Operating System:	SUSE Linux Enterprise Server 11 (x86_64), kernel 2.6.27.19-5-default
CPU Characteristics:	Intel Turbo Boost Technology up to 3.33 GHz	Compiler:	Intel C++ and Fortran Professional Compiler for IA32 and Intel 64, Version 11.1 Build 20091130 Package ID: l_cproc_p_11.1.064, l_cprof_p_11.1.064
CPU MHz:	2933	Auto Parallel:	Yes
FPU:	Integrated	File System:	ext3
CPU(s) enabled:	12 cores, 2 chips, 6 cores/chip	System State:	Multi-User Run Level 3
CPU(s) orderable:	1 chip		
Primary Cache:	32 KB I + 32 KB D on chip per core		
Secondary Cache:	256 KB I+D on chip per core		
Continued on next page		Continued on next page	

Hardware (Continued)		Software (Continued)	
L3 Cache:	12 MB I+D on chip per chip	Base Pointers:	64-bit
Other Cache:	None	Peak Pointers:	32/64-bit
Memory:	24 GB (6x4 GB PC3-10600E, 2 rank, CL9)	Other Software:	Binutils 2.18.50.0.7.20080502
Disk Subsystem:	1 x SATA II, 400 GB, 7200 rpm		
Other Hardware:	None		

Operating System Notes

```
'ulimit -s unlimited' was used to set the stacksize to unlimited prior to run
OMP_NUM_THREADS set to number of cores
KMP_AFFINITY set to granularity=fine,scatter
KMP_STACKSIZE set to 200M
```

Compiler Invocation

C benchmarks:

```
icc   -m64
```

Courtesy of Prof. G. Wellein, U. of Erlangen

Performance: Report for professional benchmark

Compiler Invocation (Continued)

C++ benchmarks:

```
icpc  -m64
```

Fortran benchmarks:

```
ifort -m64
```

Benchmarks using both Fortran and C:

```
icc   -m64 ifort -m64
```

Base Optimization Flags

C benchmarks:

```
-xSSE4.2 -ipo -O3 -no-prec-div -static -parallel -opt-prefetch
```

C++ benchmarks:

```
-xSSE4.2 -ipo -O3 -no-prec-div -static -parallel -opt-prefetch
```

Fortran benchmarks:

Peak Optimization Flags

C benchmarks:

433.milc: `-xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -static(pass 2) -prof-use(pass 2) -ansi-alias`

470.lbm: `-xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -static(pass 2) -prof-use(pass 2) -parallel -ansi-alias -auto-ilp32`

2 more pages…

Courtesy of Prof. G. Wellein, U. of Erlangen

Performance – TIME: which one?

- **LINUX / UNIX command `time` :**

```
>time ./test.x
>34.650u 0.612s 0:35.28 99.9%

>time ./testwIO.x
>33.802u 0.608s 0:43.64 78.8%
```

- **`> xxxu yyys mm:ss CPUratio%`**

`xxx` → USER CPU time [s] **`yyy` → SYSTEM CPU time [s]**
`mm:ss` → Elapsed time **`CPUratio` → `(xxx+yyy)/mm:ss`**

- **Performance metric ("Bestseller"): wallclock time**
  - **Measures time to solution → best metric thinkable, but not intuitive in all situations**
  - **Carefully specify the "problem" you solved!**
  - **Use dedicated compute nodes to avoid interference from other users!**

Performance – TIME: which one?

- **Measuring `walltime` within code on UNIX (-like) systems**
  - **Stay away from CPU time – it's evil!**

  - Use `gettimeofday()` to measure walltime:

```
#include <sys/time.h>

double timestamp(void){
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return((double)(tp.tv_sec + tp.tv_usec/1000000.0));
}
```

  - TIME:= Difference of two timestamps!

- **Works fine for serial timings – take care with parallel applications to be sure that you know what you're measuring**