Thirteen modern ways to fool the masses with performance results on parallel computers

Georg Hager

Erlangen Regional Computing Center (RRZE)

University of Erlangen-Nuremberg

12th Teraflop Workshop

HLRS, 15.03.2010

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.
5. Quote performance results projected to a full system.
6. Compare your 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.

At that time…

- The “oxen vs. chicken” debate was in full swing
- Cray was dominating in the “oxen” department
- People were more used/forced to system-specific optimizations
- The question whether to use 32-bit or 64-bit FP arithmetic was more important than it was today
  - However, GPUs have re-opened this can (and somebody should have long ago)

Today we have…

- **Hybrid, hierarchical systems**
  - Multi-socket, multi-core, ccNUMA, heterogeneous networks
- **Multi-core processors**
  - Shared/separate caches, shared data paths
- **Fledglings all over the place**
  - Cell, Clearspeed, GPUs... (peep, peep)
- **Commodity everywhere**
  - x86-type processors, cost-effective interconnects, GNU/Linux

The landscape of High Performance Computing and the way we think about HPC has changed over the last 19 years, and we need an update!

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

Stunt 1

- **Report scalability, not absolute performance.**

Speedup:

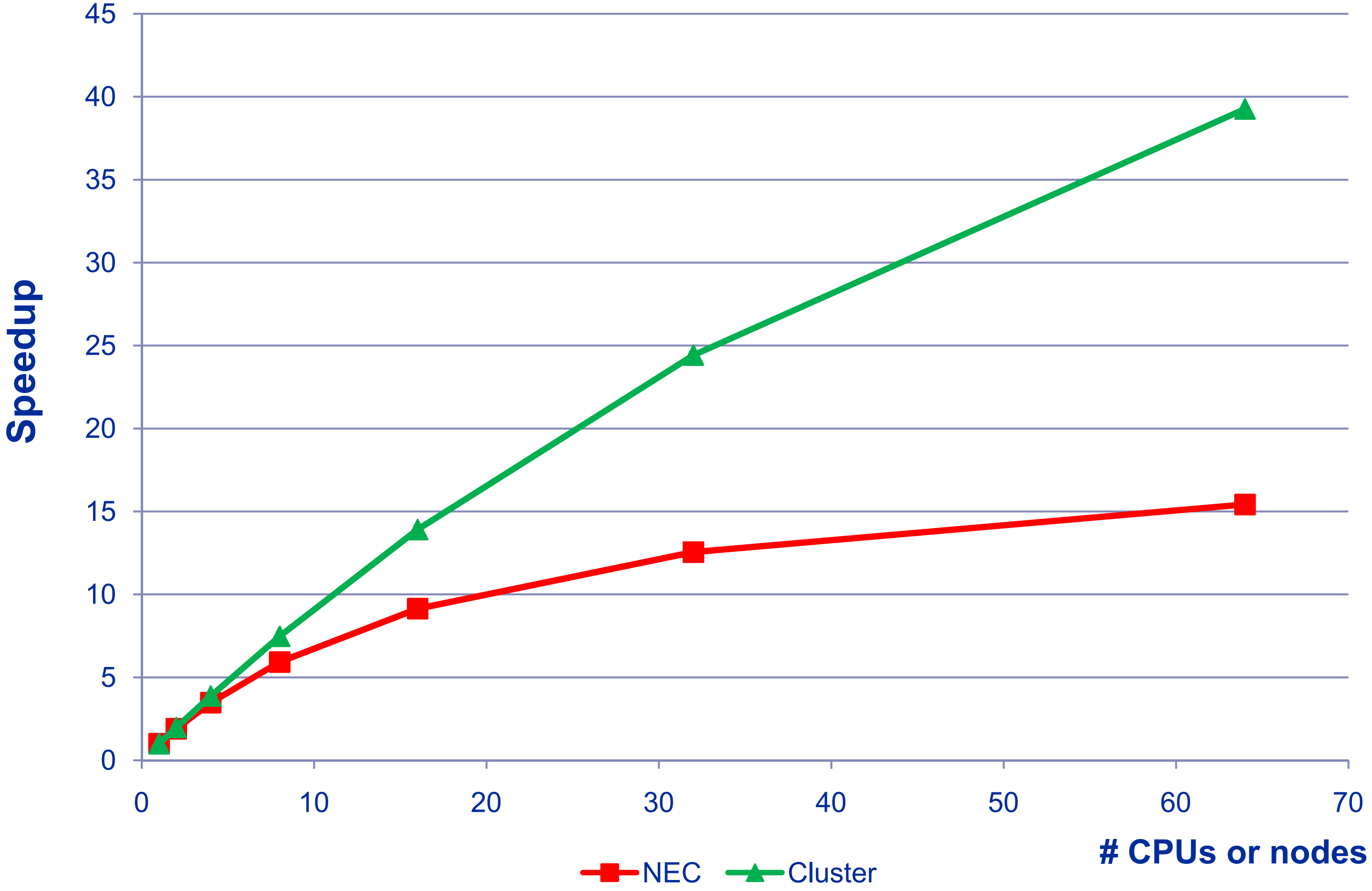
$$S(N) = \frac{\text{work/time with } N \text{ workers}}{\text{work/time with } 1 \text{ worker}}$$

“Good” scalability ↔ $S(N) \approx N$, but there is no mention of how fast you can solve your problem!

Consequence: Comparing different systems is much easier when using scalability instead of work/time directly

Stunt 1: Scalability vs. performance

- **And… instant success!**



Stunt 2

- **Slow down code execution.**

This is useful whenever there is some noticeable “non-execution” overhead

Parallel speedup with work ~ N^α: (α=0: strong, α=1: weak scaling)

$$S(N) = \frac{s+(1-s)N^{\alpha}}{s+(1-s)N^{\alpha-1}+c_{\alpha}(N)}$$

Now let’s slow down execution by a factor of μ>1 (and set α=0):

$$S_{\sigma}(N) = \frac{\mu}{\mu\big(s+(1-s)/N\big)+c(N)} = \frac{1}{s+(1-s)/N+c(N)/\mu}$$

I.e., if there is overhead, the slow code/machine scales better:

$$S_{\mu}(N) > S_{\mu=1}(N) \quad \text{if} \quad c(N) > 0$$

Stunt 2: Slow computing

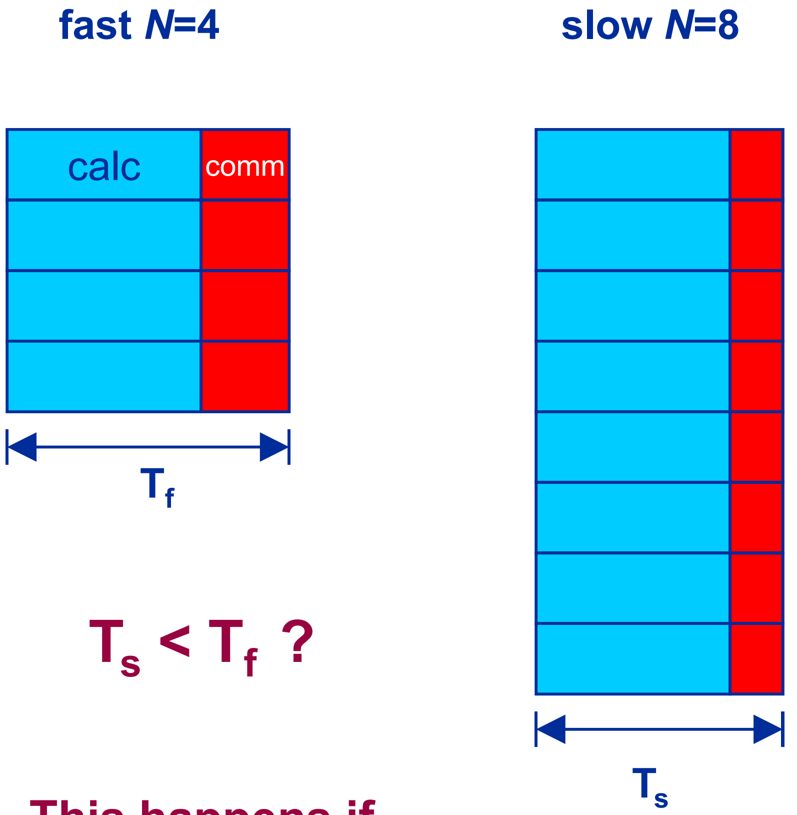
- **Corollaries:**

  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.

Stunt 2: Slow computing

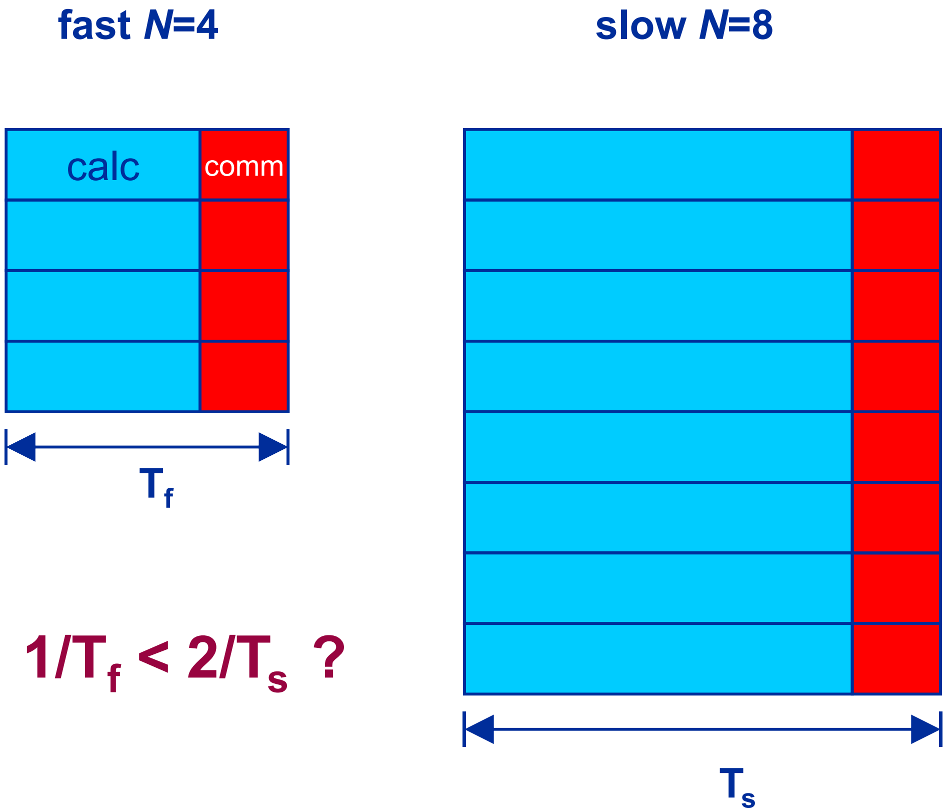
- “Slow” machines have some surprises in store…
  - Let’s look at μ=2:



$T_s < T_f$?

This happens if

$$c'(N) < 0 \quad @ \quad s = 0$$



$1/T_f < 2/T_s$?

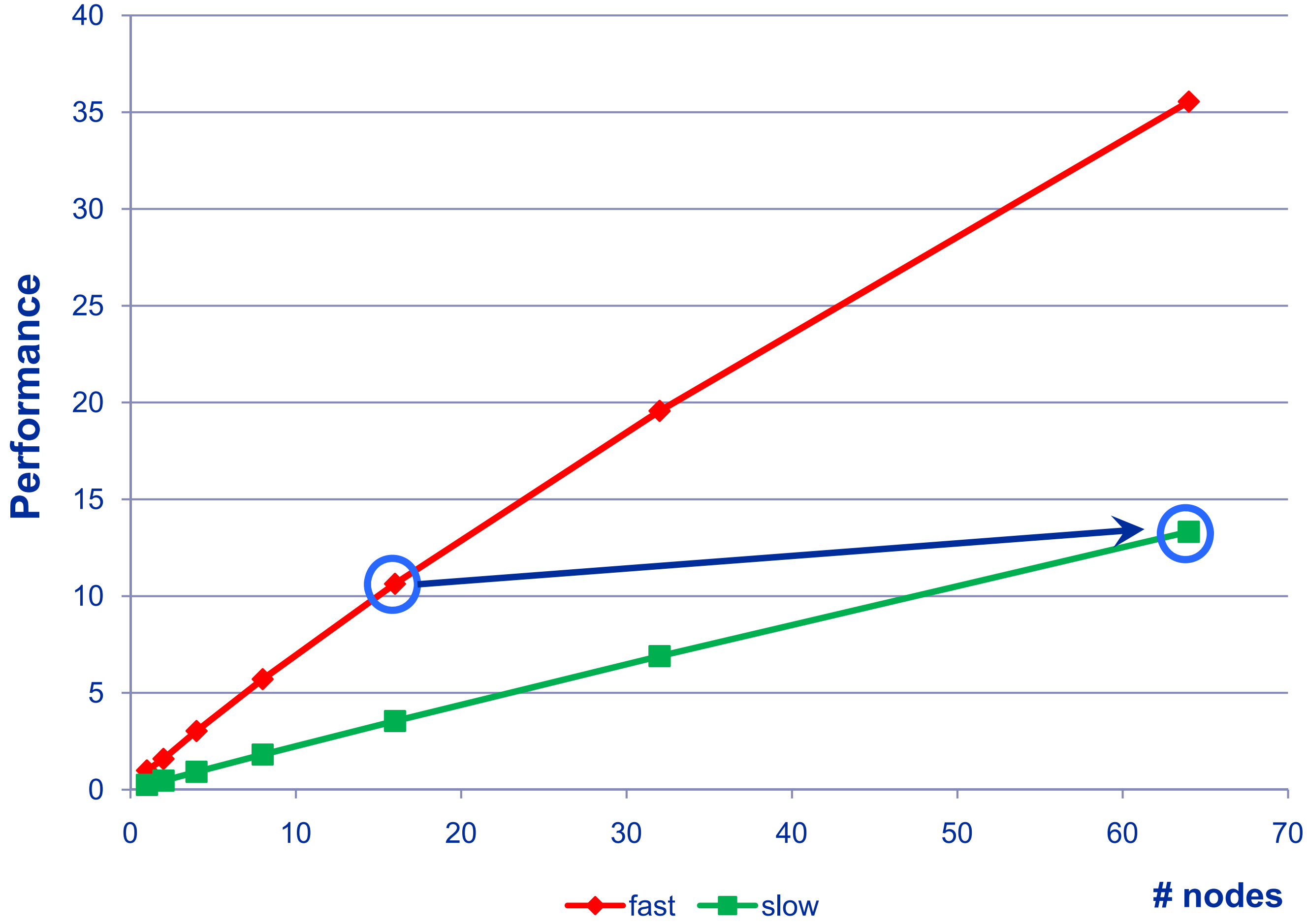
This happens if

$$c(N) > \frac{c(\mu N)}{\mu} \quad @ \quad s = 0$$

What’s the catch?

Stunt 2: Slow computing

- **Example for μ=4 and $c(N)$~$N^{-2/3}$ at strong scaling:**



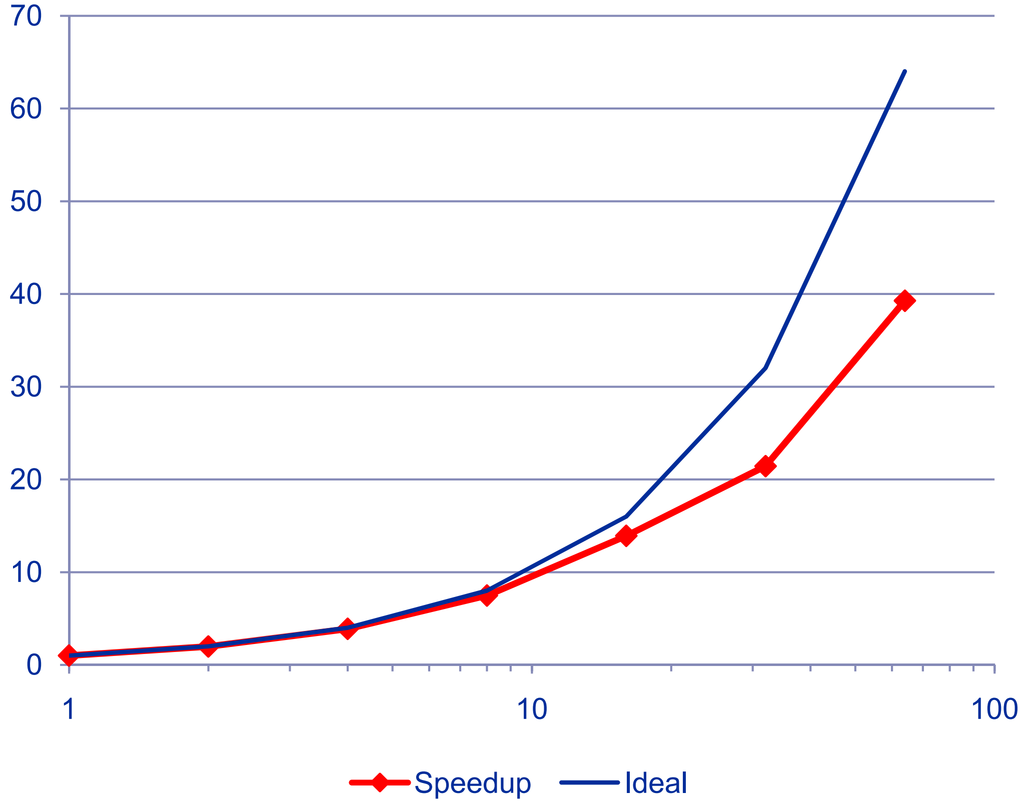
- The performance is better with μN slow CPUs than with N fast CPUs
- "Slow computing" can effectively lessen the impact of communication overhead
- We assume that the network is the same in both machines

Stunt 3 (The power of obfuscation, part I)

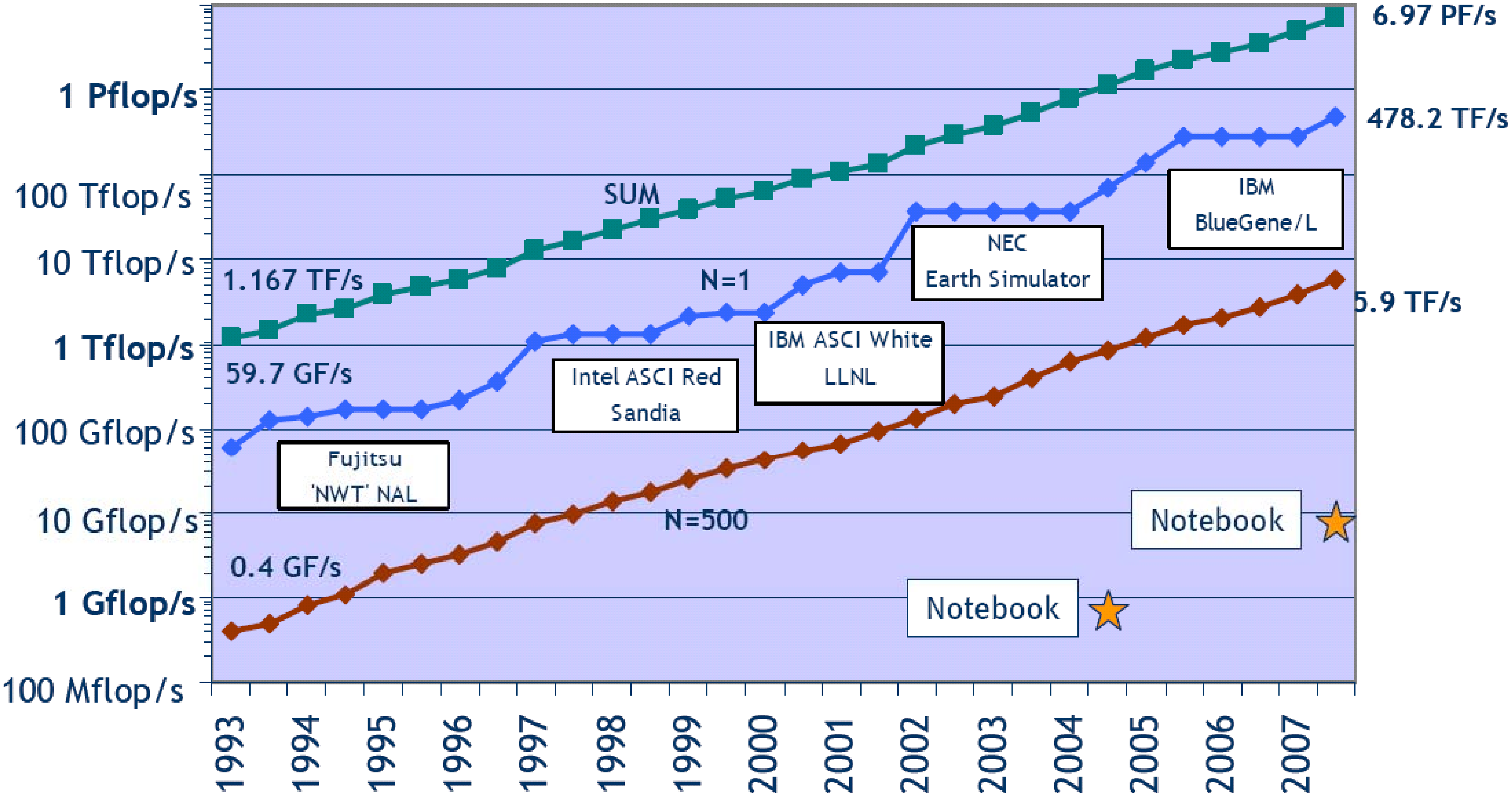
- **If scalability doesn't look good enough, use a logarithmic scale to drive your point home.**

Everything looks OK if you plot it the right way!

1. Linear plot: bad scaling, strange things at N=32
2. Log-log plot: better scaling, but still the N=32 problem
3. Log-linear plot: N=32 problem gone



Stunt 3: Log scale



© Top500 `08

Stunt 4

- **If you must report actual performance, *quietly* employ weak scaling to show off**

It’s all in that bloody denominator…

$$S(N)=\frac{s+(1-s)N^{\alpha}}{s+(1-s)N^{\alpha-1}+c_{\alpha}(N)}$$

At *α*=1 the world looks so much nicer:

$$S(N)=\frac{s+(1-s)N}{1+c(N)}$$

… but keep in mind: Do not mention the term “weak scaling” or you will be asked nasty questions about parallel efficiency.

Stunt 4: Weak scaling

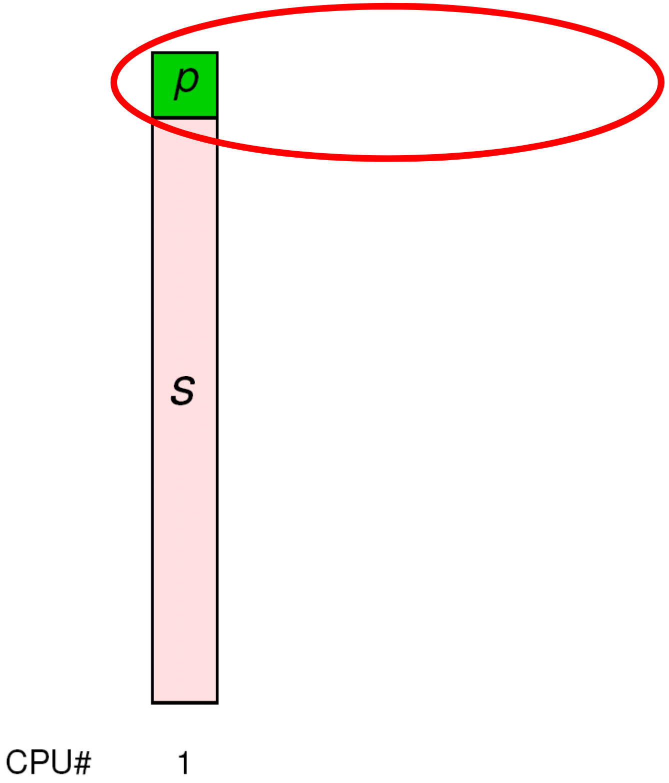
- But weak scaling gives us much more than just a “straight” graph. It gives us perfect scaling if we choose the right metric to look at!

- Assumption: Weak scaling with parallel efficiency $\varepsilon = S(N)/N << 1$ and no other overhead

→ $S(N) = s + (1 - s)N$ has a small slope

But: If we choose a metric for work that is applicable to the parallel part alone, work/time scales linearly.

So all you need to do is plot Mflop/s, MLUP/s, or anything that doesn't happen in the serial part and you can even show real performance numbers! → See also stunt #10

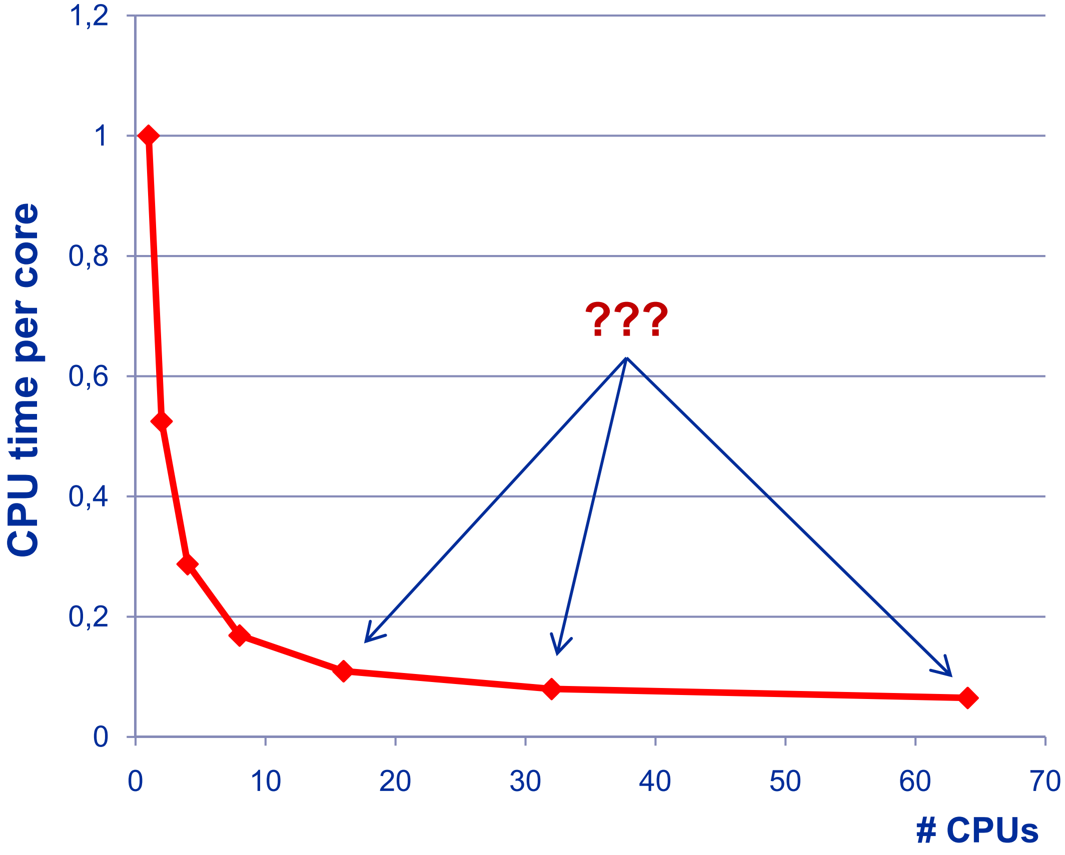


Stunt 5 (The power of obfuscation, part II)

- **Instead of performance, plot absolute runtime vs. CPU count**

Very, very popular indeed!

Nobody will be able to tell whether your code actually Scales



Corollary:

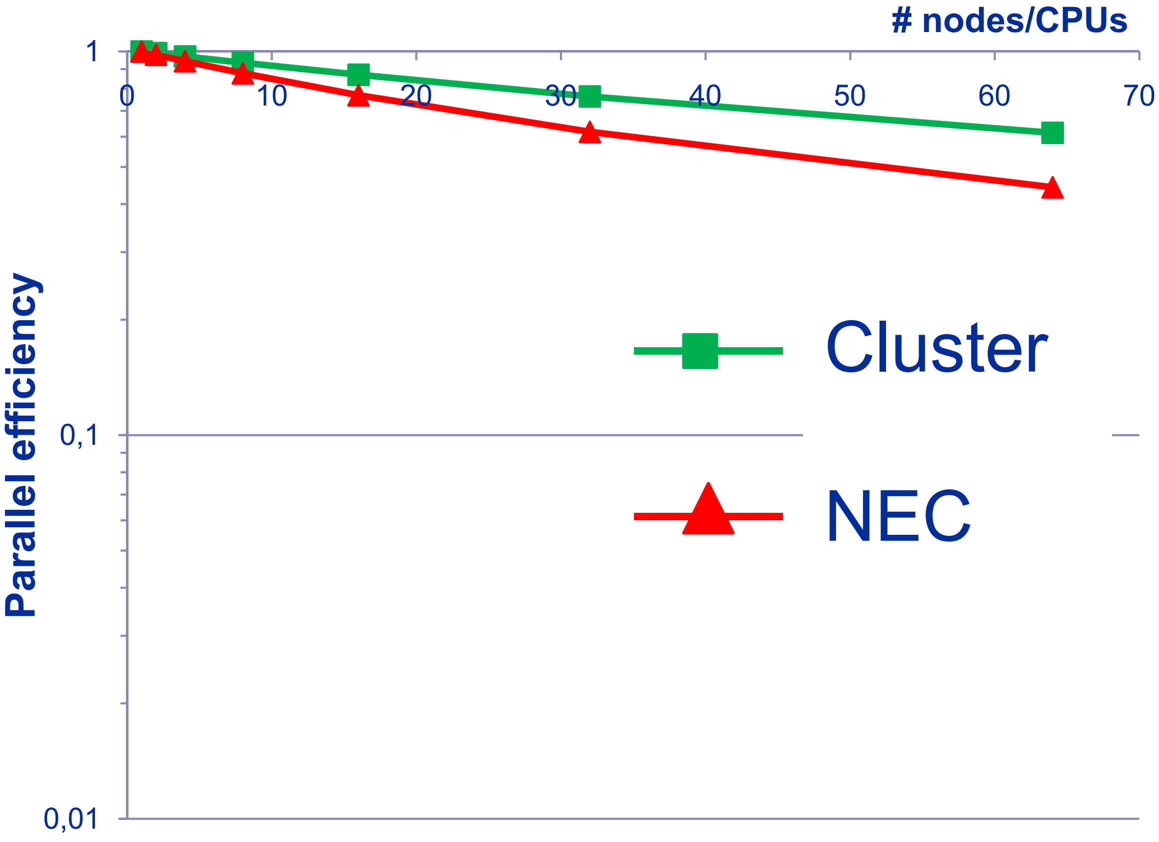
CPU time per core is even better because it omits most overheads…

Stunt 6 (The power of obfuscation, part III)

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

Unusual ways of putting data together surprise and confuse your audience

Remember: Legends can be any size you like!



Stunt 7

- **Emphasize the quality of your shiny accelerator code by comparing it with scalar, unoptimized code on a single core of a standard CPU. *And use GCC 2.7.2*.**

This should be obvious! GPUs are leet, and you can’t waste your precious time on multi-core parallelization, OpenMP optmization, or even compiler flags.

And besides, don’t the compiler guys always say that the’re “multi-core enabled”?

Corollary:

Use single precision on the GPU but double precision on the CPU. This will cut on the effective bandwidths, cache size, and peak performance of the latter and let the former shine.

Stunt 8

- **Always quote GFlops, MIps, Watts per Flop or any other ~~irrelevant~~ interesting metric instead of inverse time to solution.**

Flops are so cool it hurts:

```
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];
```

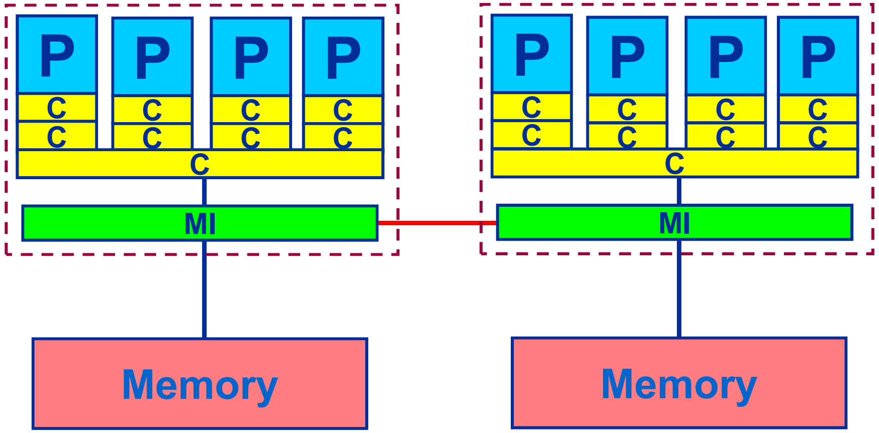
“Floptimization”

Watts/Flop are an ingenious fallback – who would dare question a truly “green” application/system? Except maybe some investors…

Stunt 9

- **Ignore affinity and topology issues. Real scientists are not bothered by such details.**

Multi-core, cache groups, ccNUMA, SMT, network hierarchies etc. are just parts of a vicious plot to take the fun out of computing. Ignoring those issues will make them go away. If people ask specific questions about it, answer that it's the OS's or the compiler's job.



OpenMP overhead

Shared cache re-use

OS buffer cache

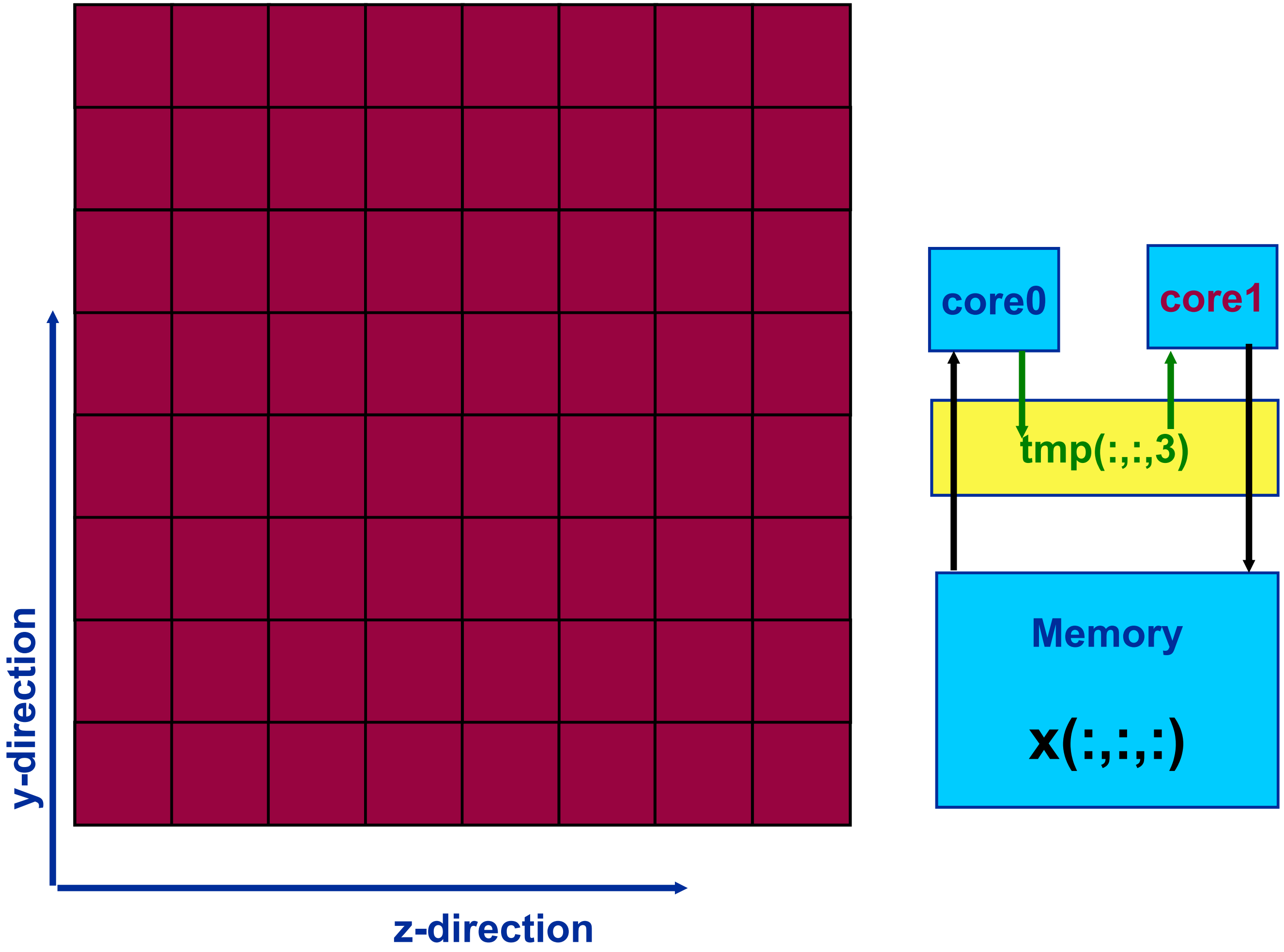
Bandwidth contention

Intra-node MPI

ccNUMA page placement

Stunt 9: Affinity issues

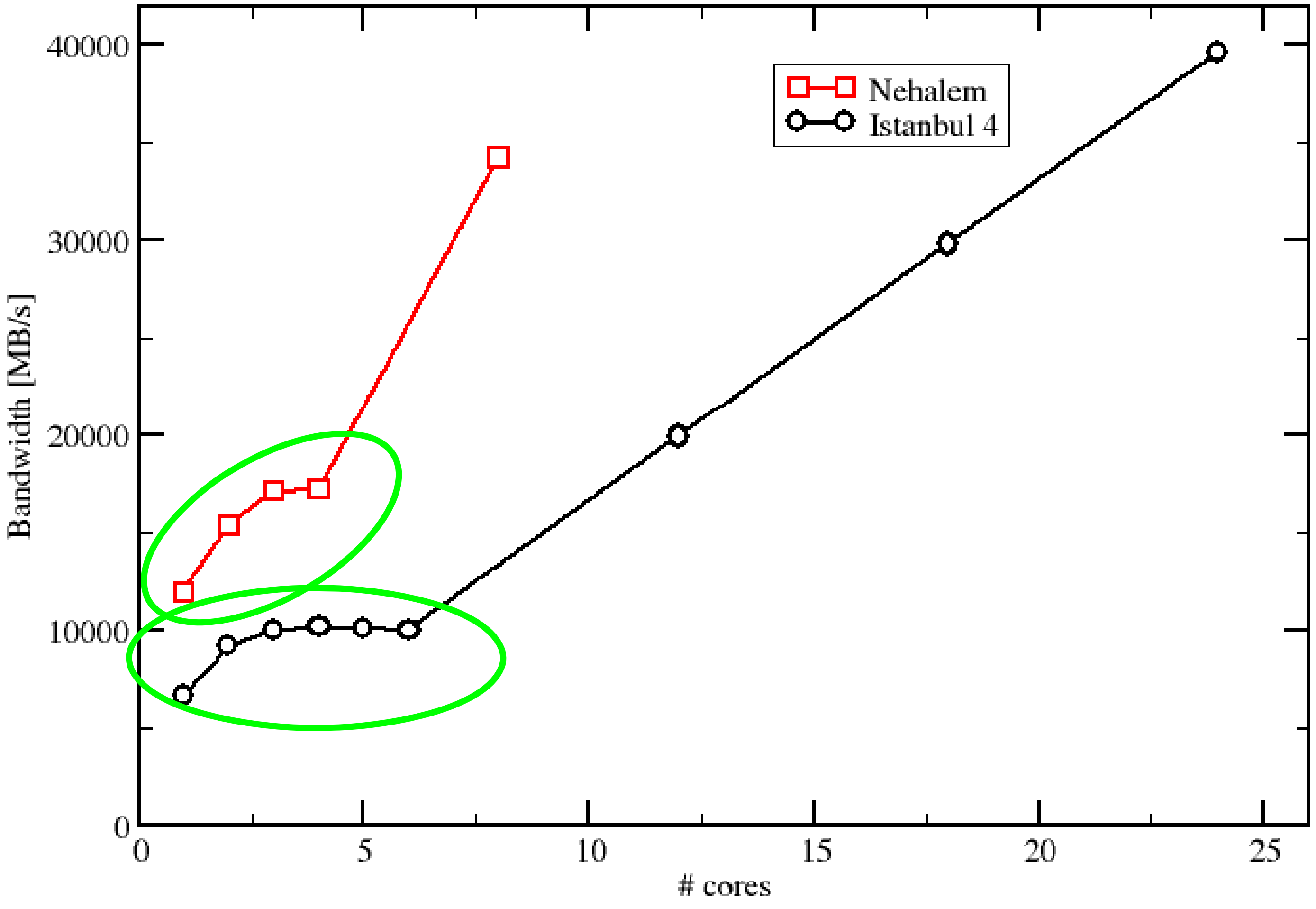
- **Re-using shared cache** on multi-core CPUs? More cores mean more performance, do they not?



Stunt 9: Affinity issues

- **Memory bandwidth saturation? ccNUMA effects? Shouldn’t the OS put the threads and pages where they are supposed to be?**

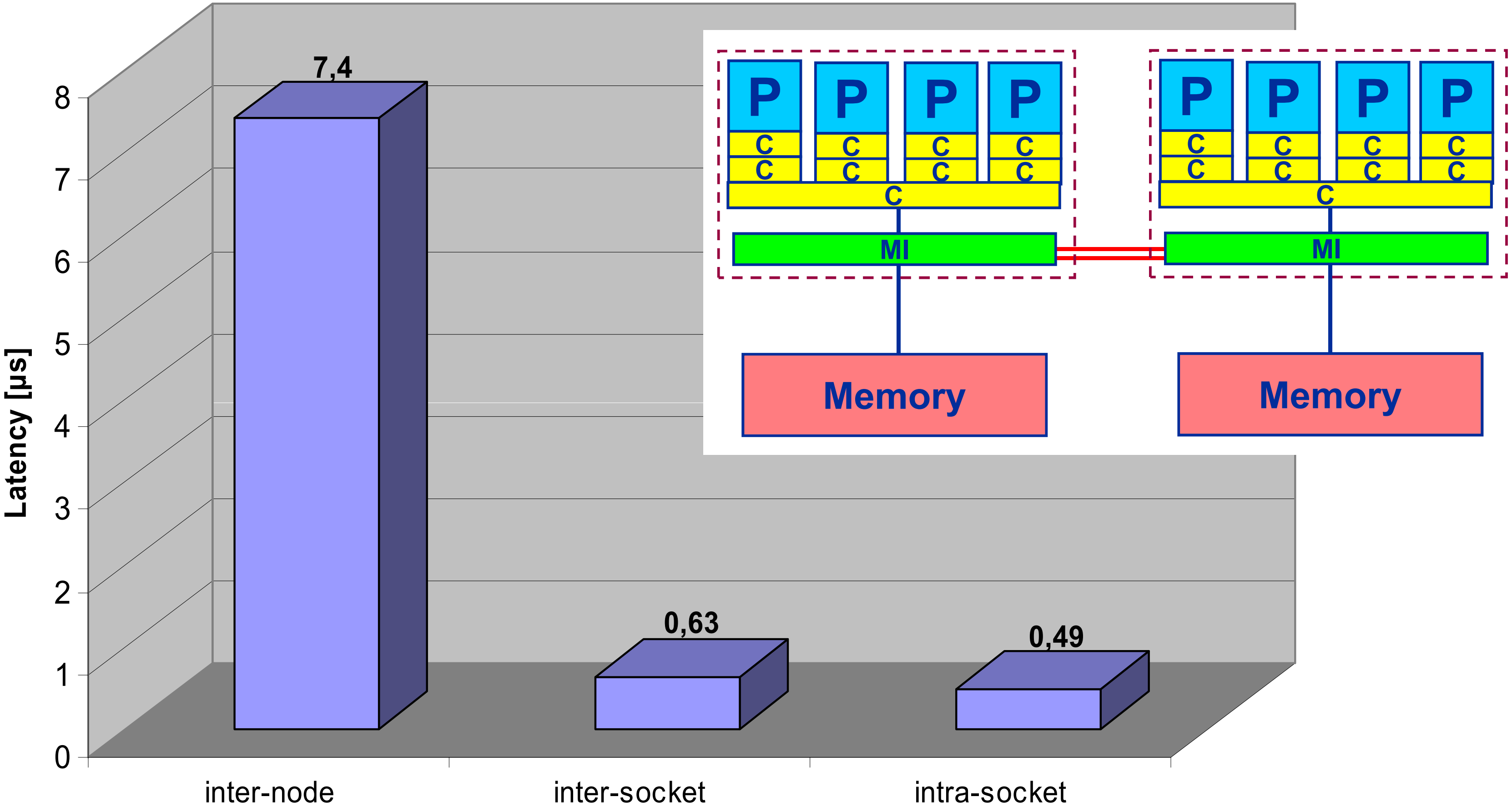
Parallel STREAM performance



Stunt 9: Affinity issues

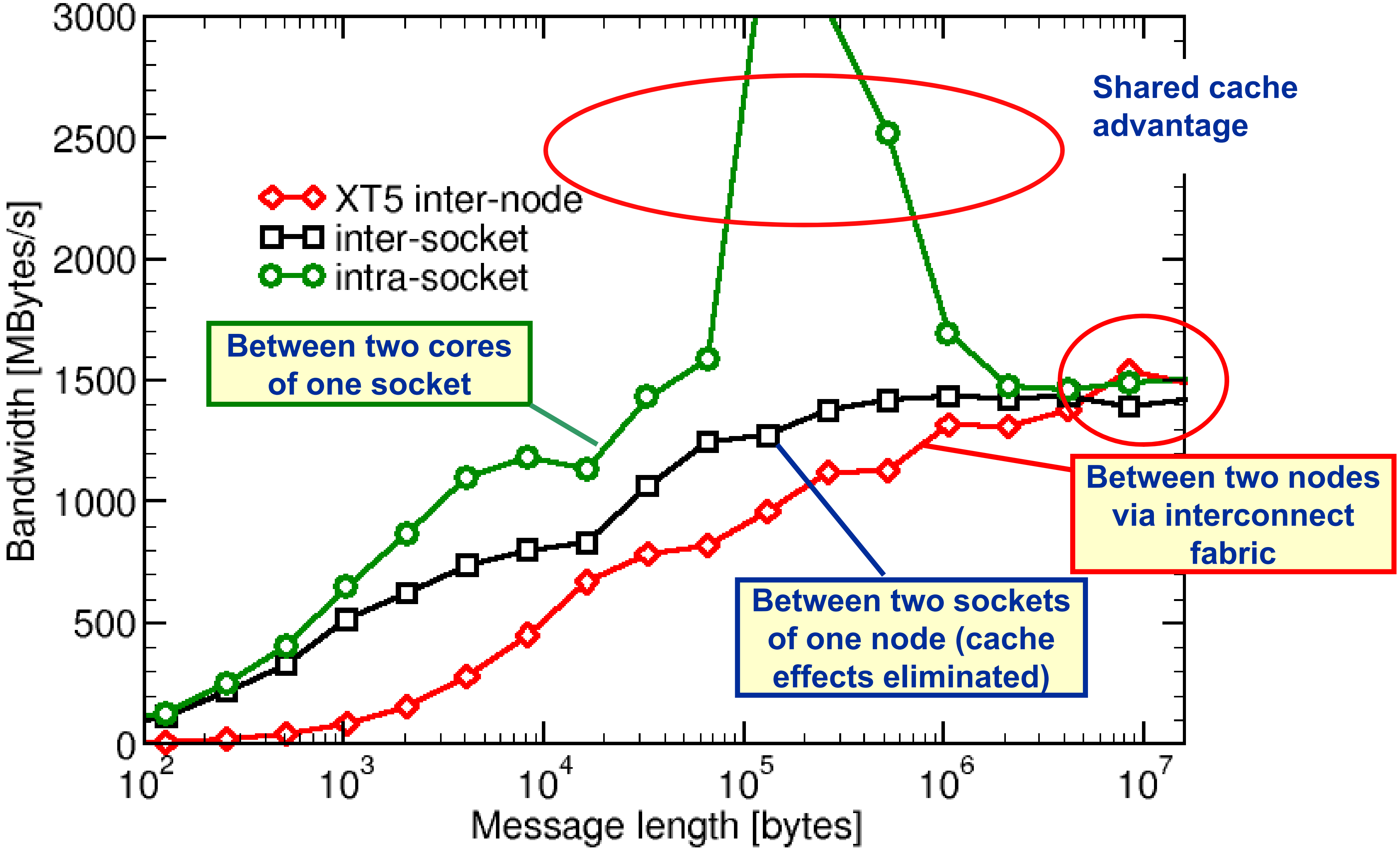
- Intra-node MPI is infinitely fast! Look at those latencies!

MPI intra-node and inter-node latencies on Cray XT5



Stunt 9: Affinity issues

- **Intra-node MPI is infinitely fast! Low-level benchmarking is unreliable!**



Shared cache advantage

Between two cores of one socket

Between two nodes via interconnect fabric

Between two sockets of one node (cache effects eliminated)

Stunt 9: Affinity issues

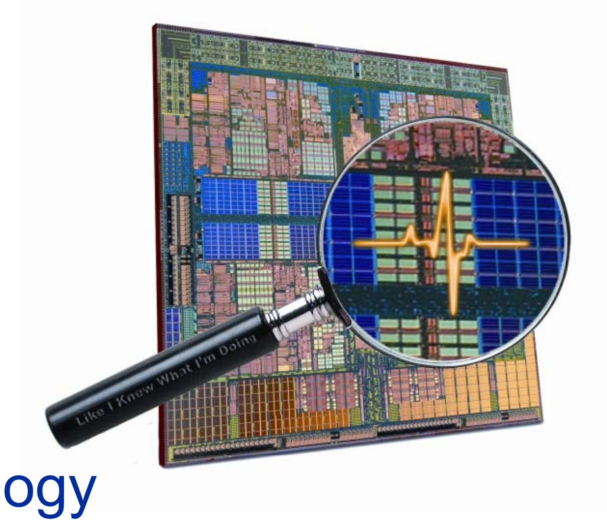
- **Why should you reverse engineer the overcomplicated cache topology of those modern systems?**

Xeon E5420 **2 Threads**	**shared L2**	**same socket**	**different socket**
pthreads_barrier_wait	5863	27032	**27647**
omp barrier (icc 11.0)	576	760	**1269**
Spin loop	259	485	11602

Nehalem **2 Threads**	**Shared SMT threads**	**shared L3**	**different socket**
pthreads_barrier_wait	**23352**	4796	49237
omp barrier (icc 11.0)	**2761**	479	1206
Spin loop	17388	267	787

Stunt 9: Affinity – if you still insist…

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - support Intel and AMD CPUs



- **Current tools:**
  - **likwid-topology**: Print thread and cache topology
  - likwid-perfCtr: Measure performance counters
  - likwid-features: View and enable/disable hardware prefetchers
  - **likwid-pin**: Pin threaded application without touching code

Open source project (GPL v2):

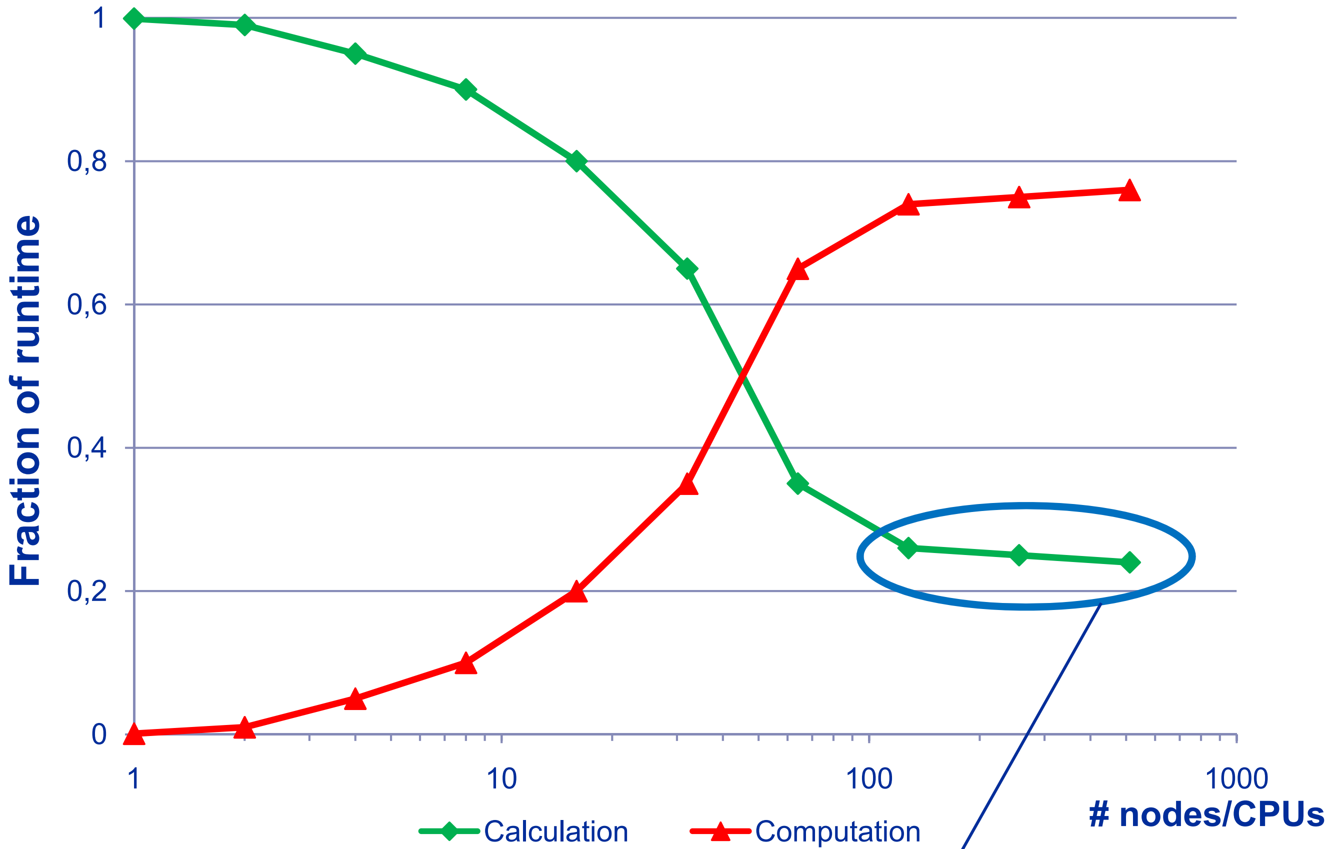
`http://code.google.com/p/likwid/`

Stunt 10

- **If you really can’t reduce communication overhead, argue in favor of “reliable inefficiency.”**

Even if you spend 80% of time communicating, that’s ok if the ratio stays constant – it means you can scale to any size!

And fill any machine.

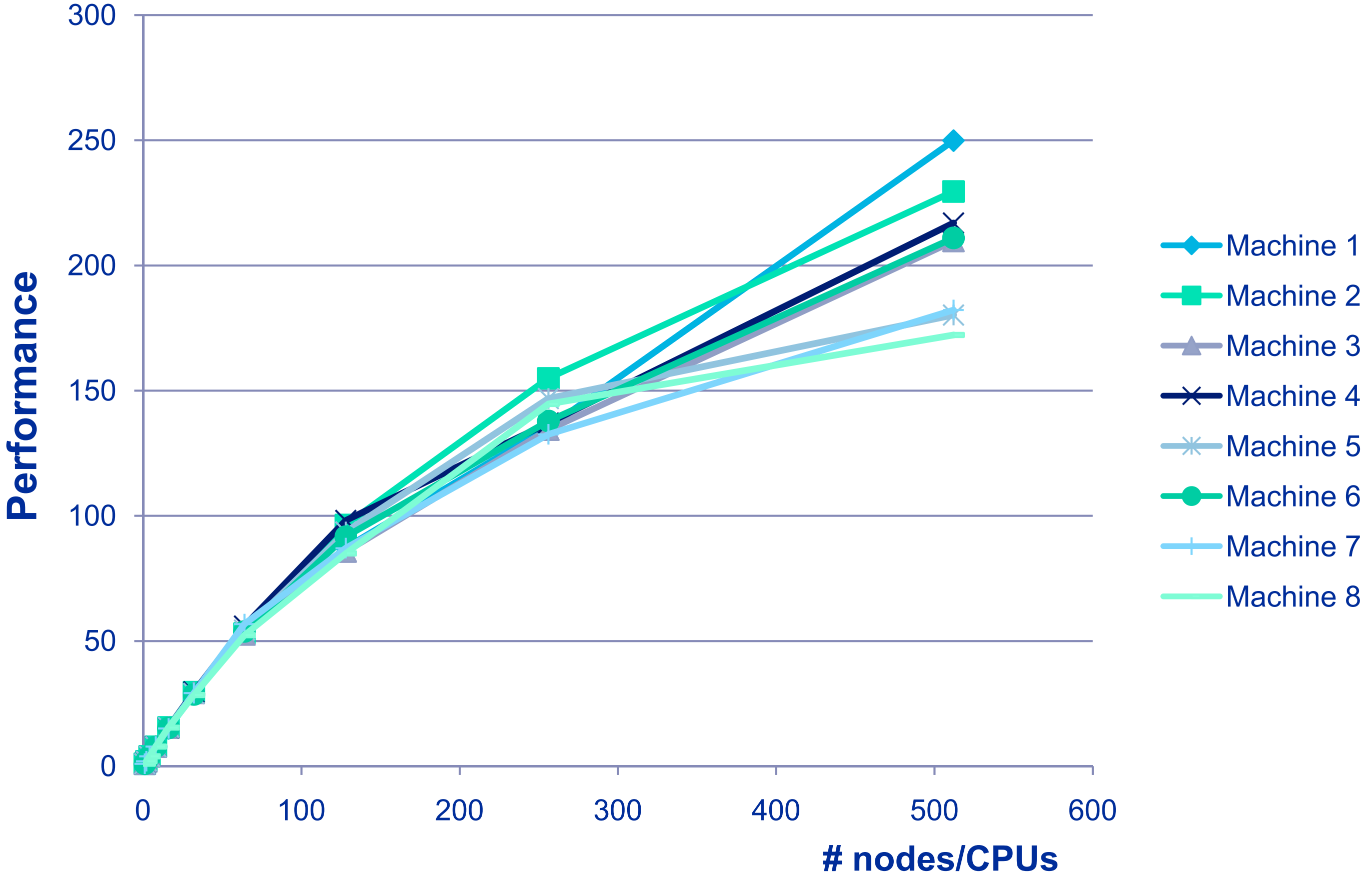


Efficiency constant for large N

Stunt 11 (The power of obfuscation, part IV)

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

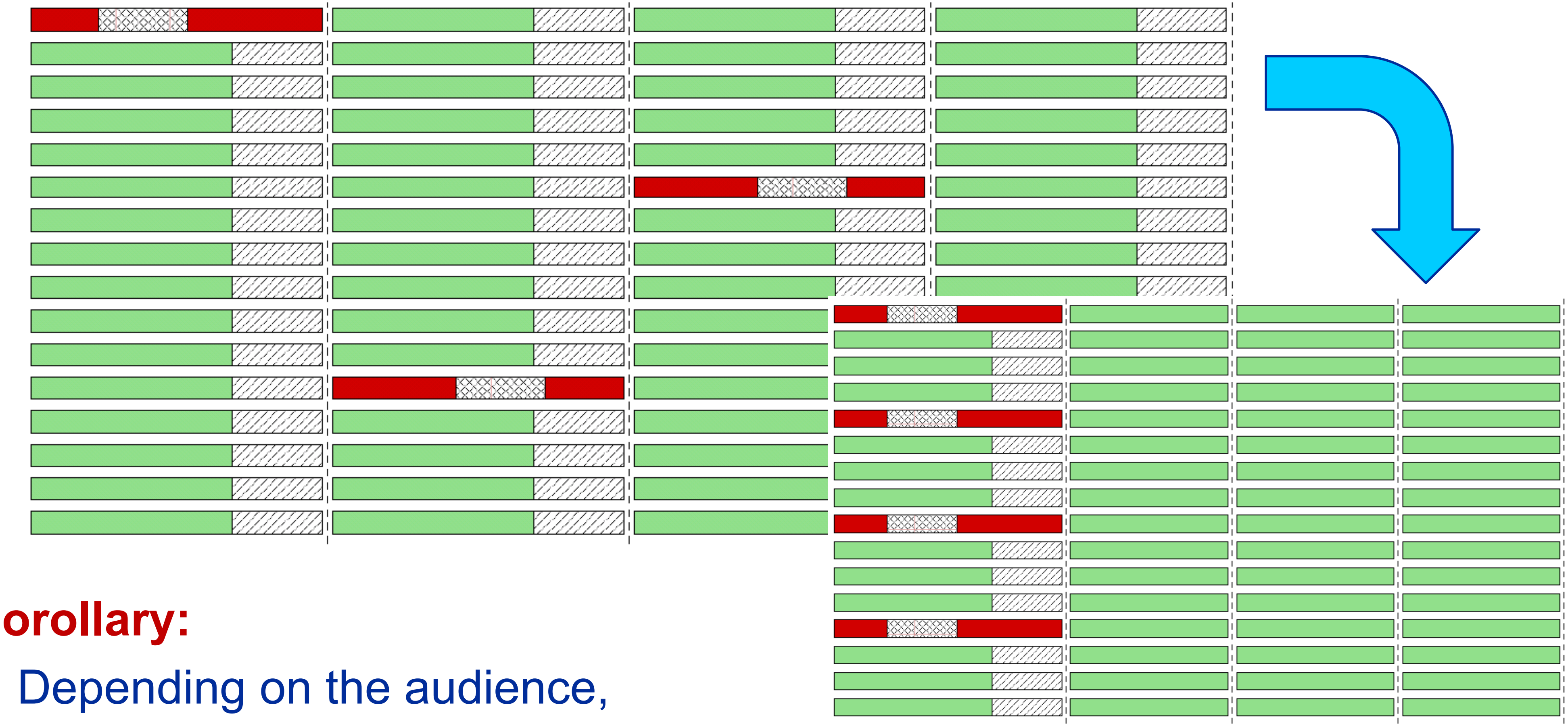


If nasty questions pop up, say your code is so complex that no model can describe it.

Stunt 12

- **If they get you cornered, blame it all on OS jitter.**

They will understand and nod knowingly.



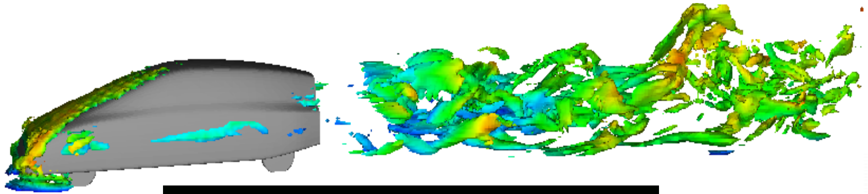
Corollary:

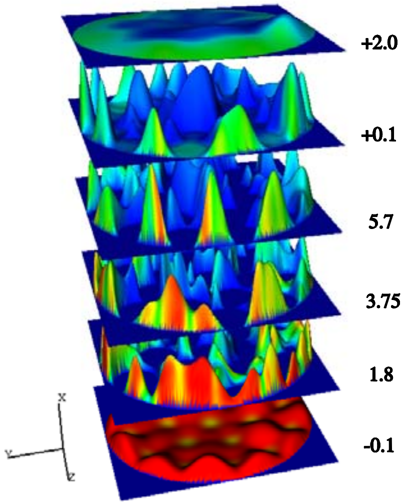
Depending on the audience,
TLB misses may work just as fine.

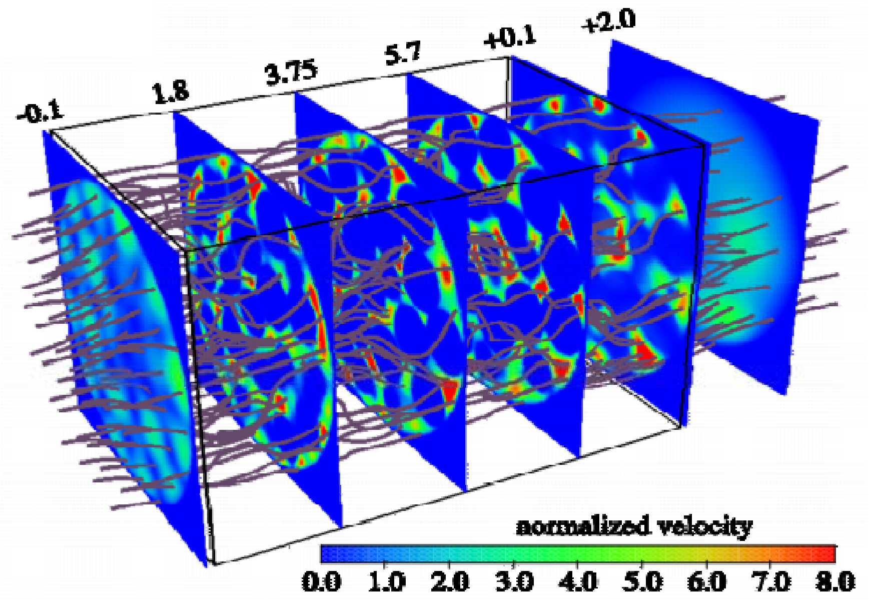
Stunt 13

- **If all else fails, show pretty pictures and animated videos, and don’t talk about performance.**

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







THANK YOU