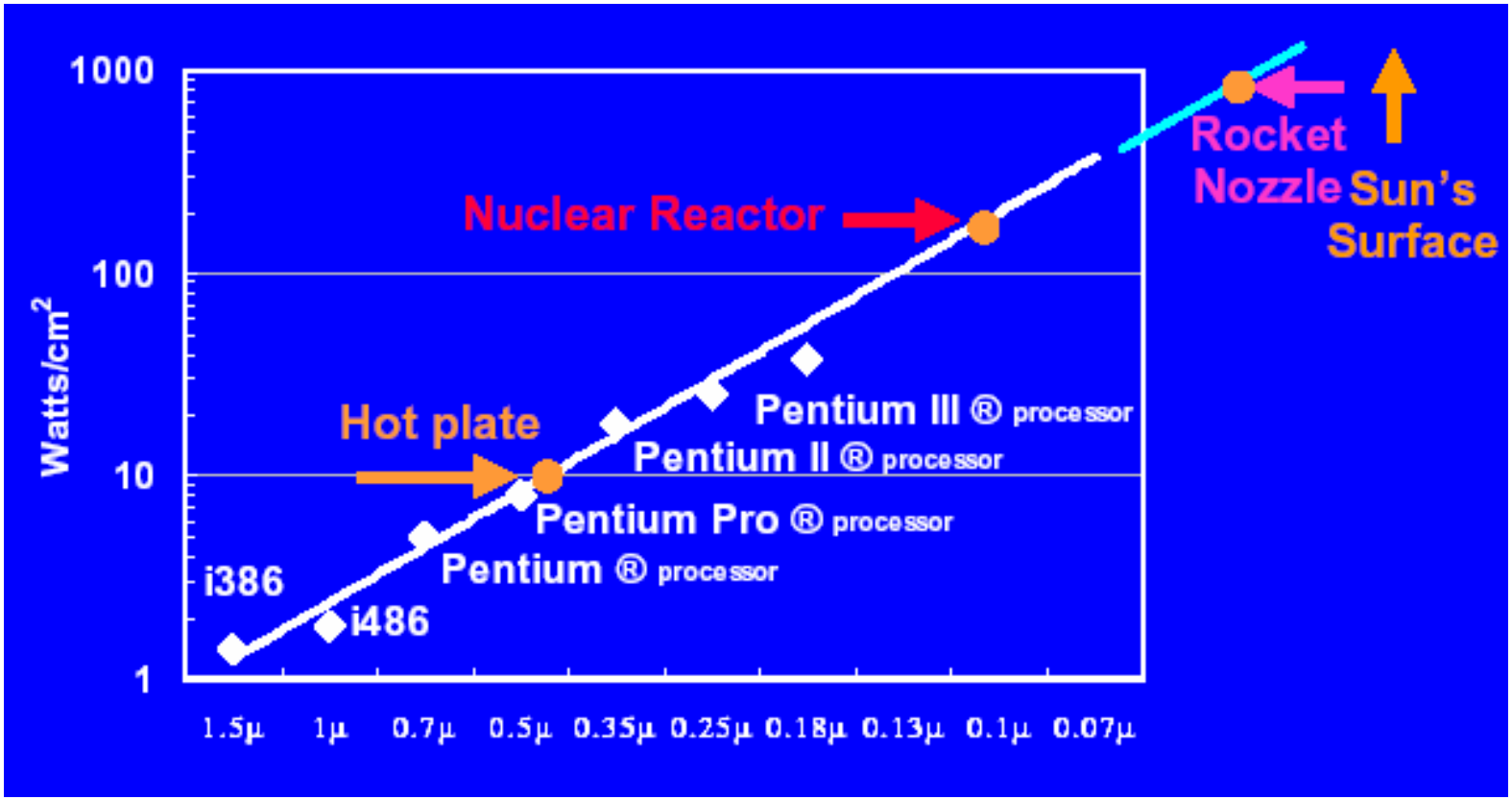
Architecture Trends and Implications for Algorithms

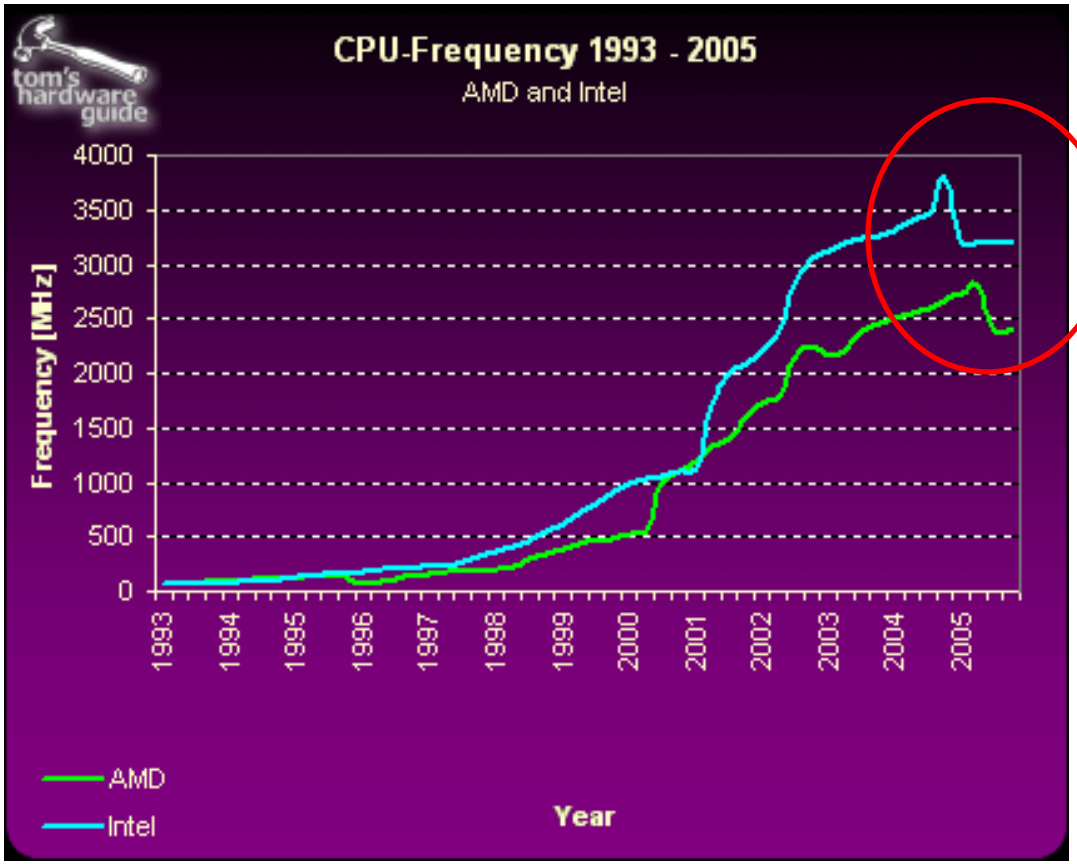
William Gropp
www.mcs.anl.gov/~gropp

Power Dissipation Trends

Or, why I can't just wait for computers to get faster

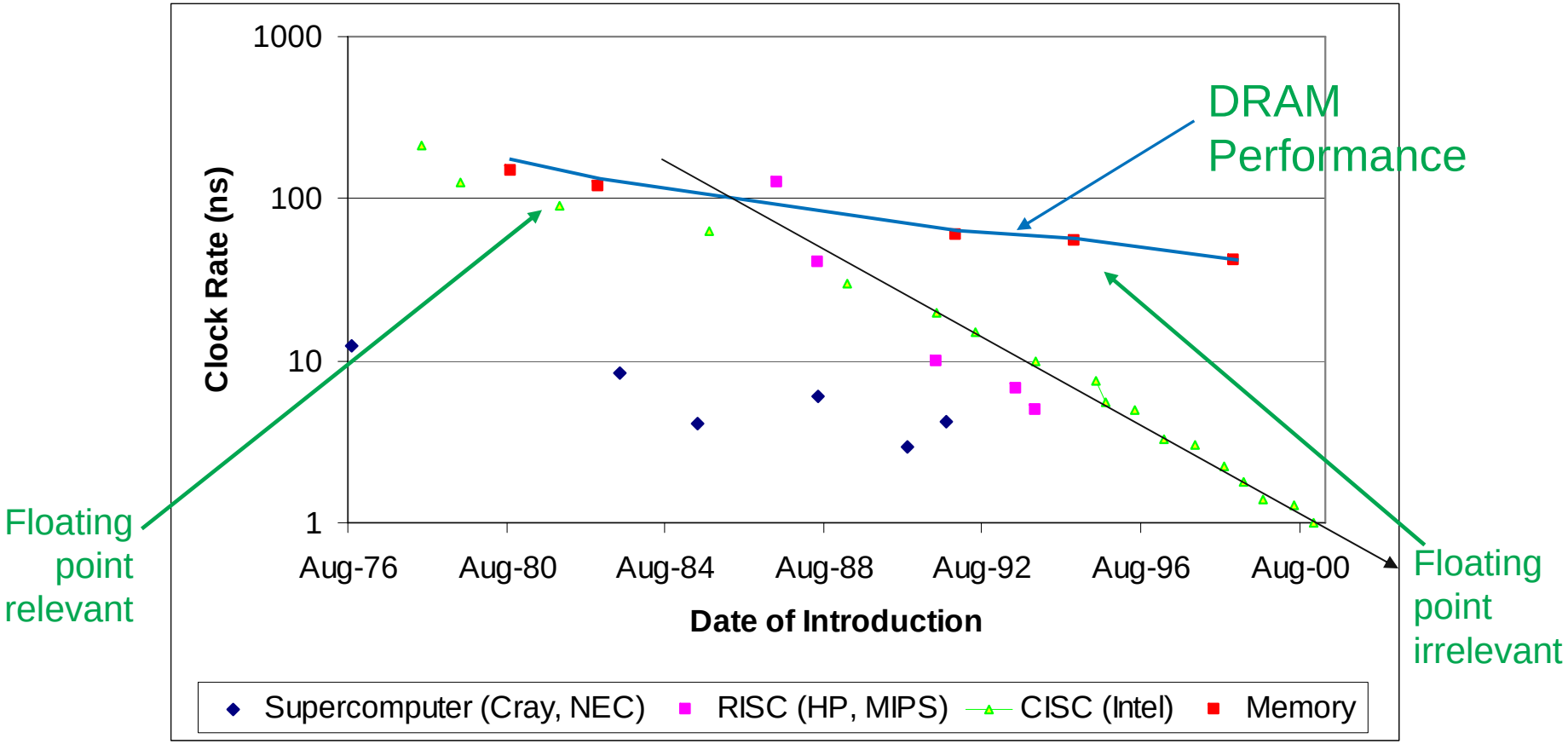


Peak CPU speeds are stable



- From http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/

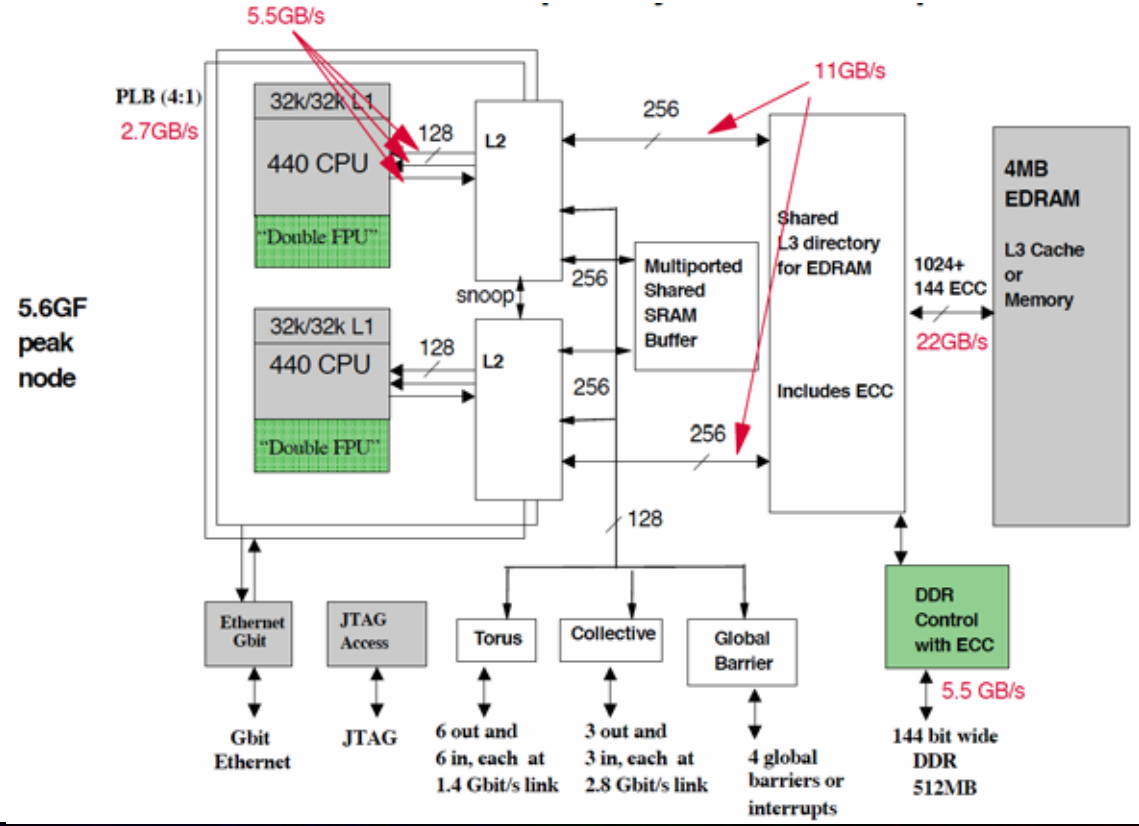
Why is achieved performance on a single node so poor?



Fixing the Performance Gap

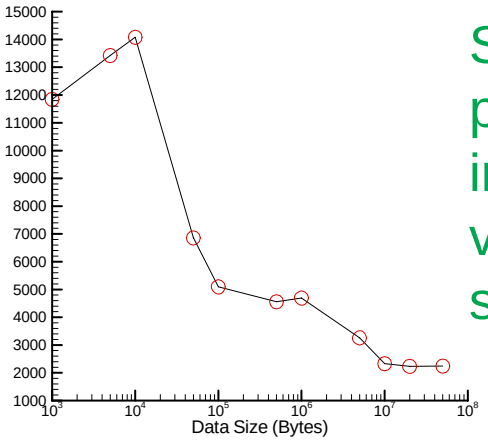
- Large gap in performance forces a design to “impedance match” CPU to memory
- Computer systems have complex, multilevel memory hierarchies
- Complexity results in nonlinear behavior for simple operations

- This diagram leaves off the most important information - latency between components
- Stated values are *never* observed by the programmer

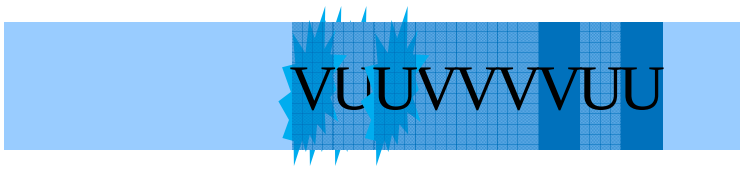


Challenges in Creating a Performance Model Based on Memory Accesses

- Different levels of the memory hierarchies have significantly different performance
- Time (work) is a *nonlinear* function of copy size
  - Source of “superlinear” speedup - that is *real*
- Cache behavior sensitive to details of data layout
- Still no good calculus for *predicting* performance
  - But all hope is not lost



STREAM performance in MB/s versus data size



Interleaved data causes data to be displaced while still needed for later steps

All this makes prediction hard

- But in best applied math tradition, ***bounds*** are possible and valuable
- Example: Sparse Matrix-Vector Product
  - Common operation for optimal (in floating-point operations) solution of linear systems
  - Sample code (in C):

```
for row=1,n
    m   = i[row] - i[row-1];
    sum = 0;
    for k=1,m
        sum += *a++ * x[*j++];
    y[i] = sum;
```

  - Data structures are a[nnz], j[nnz], i[n], x[n], y[n]

Simple Performance Analysis

- Memory motion:
  - nnz (sizeof(double) + sizeof(int)) +
    n (2*sizeof(double) + sizeof(int))
  - Assume a perfect cache (never load same data twice; only compulsory loads)
- Computation
  - nnz multiply-add (MA)
- Roughly 12 bytes per MA
- Typical WS node can move 1-4 bytes/MA
  - Maximum performance is 8-33% of peak

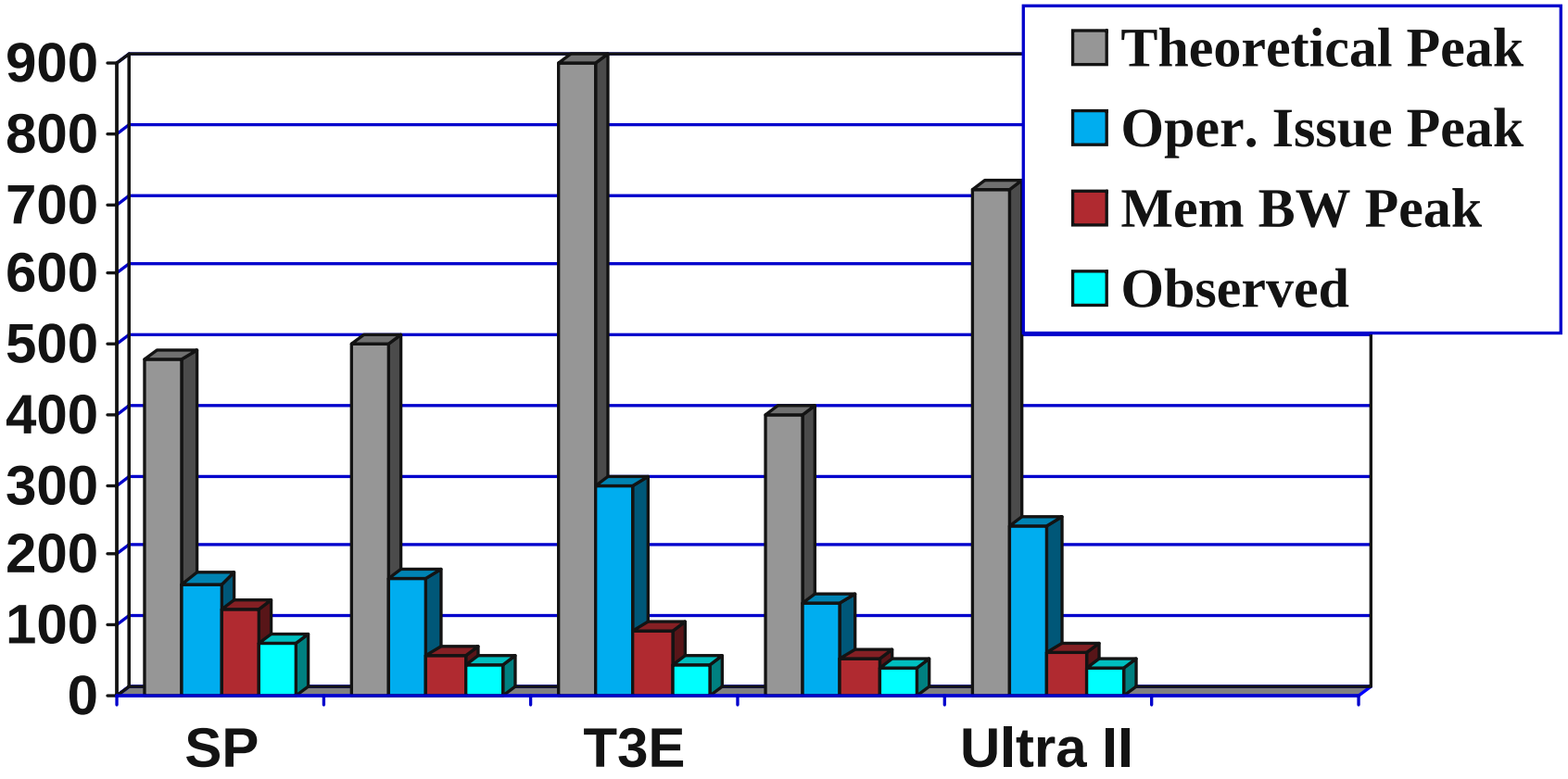
More Performance Analysis

- Instruction Counts:
  - nnz (2*load-double + load-int + mult-add) +
    n (load-int + store-double)
- Roughly 4 instructions per MA
- Maximum performance is 25% of peak (33% if MA overlaps one load/store)
  - (wide instruction words can help here)
- Changing matrix data structure (e.g., exploit small block structure) allows reuse of data in register, eliminating some loads (x and j)
- Implementation improvements (tricks) cannot improve on these limits

Realistic Measures of Peak Performance

Sparse Matrix Vector Product

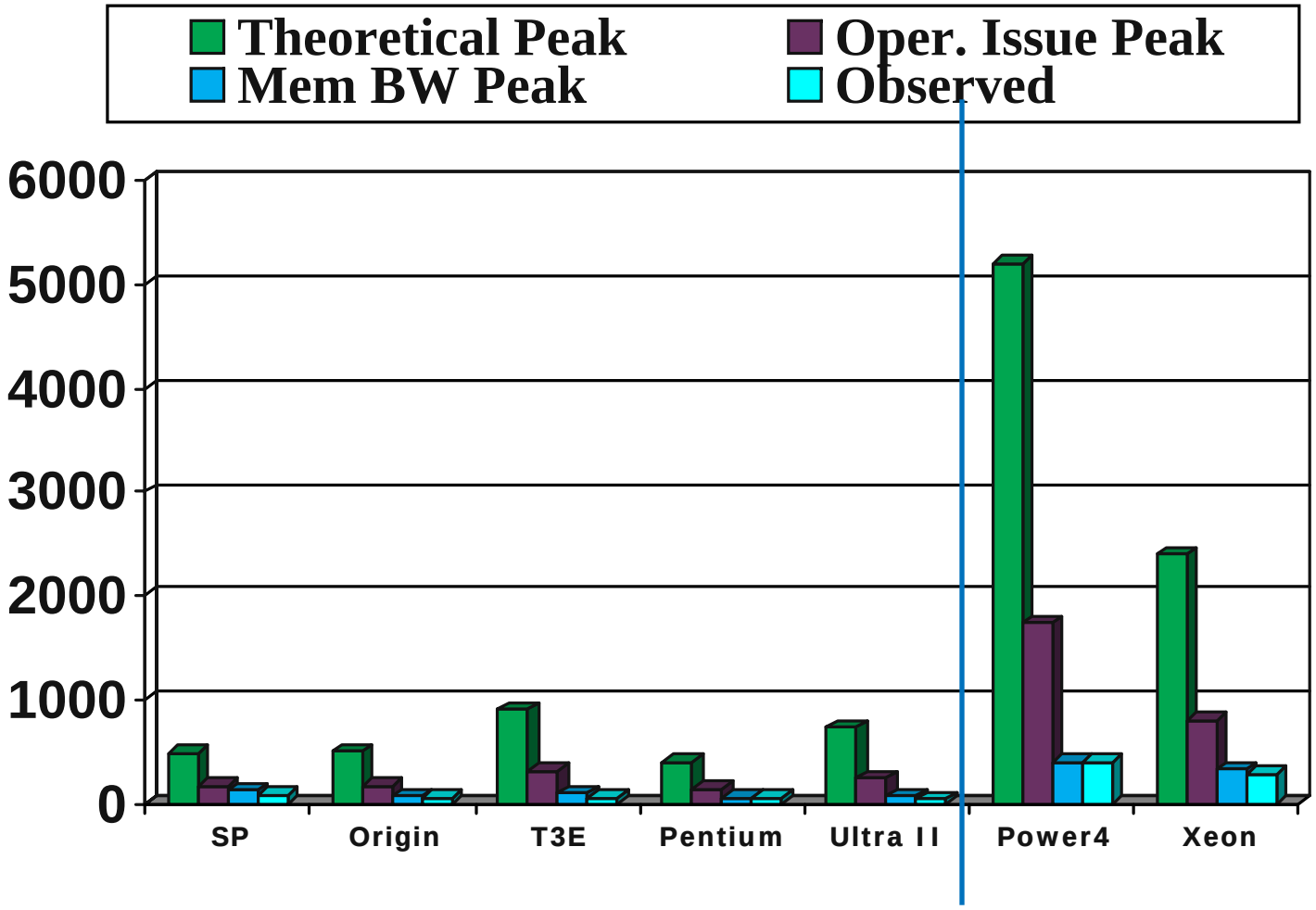
one vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



Realistic Measures of Peak Performance

Sparse Matrix Vector Product

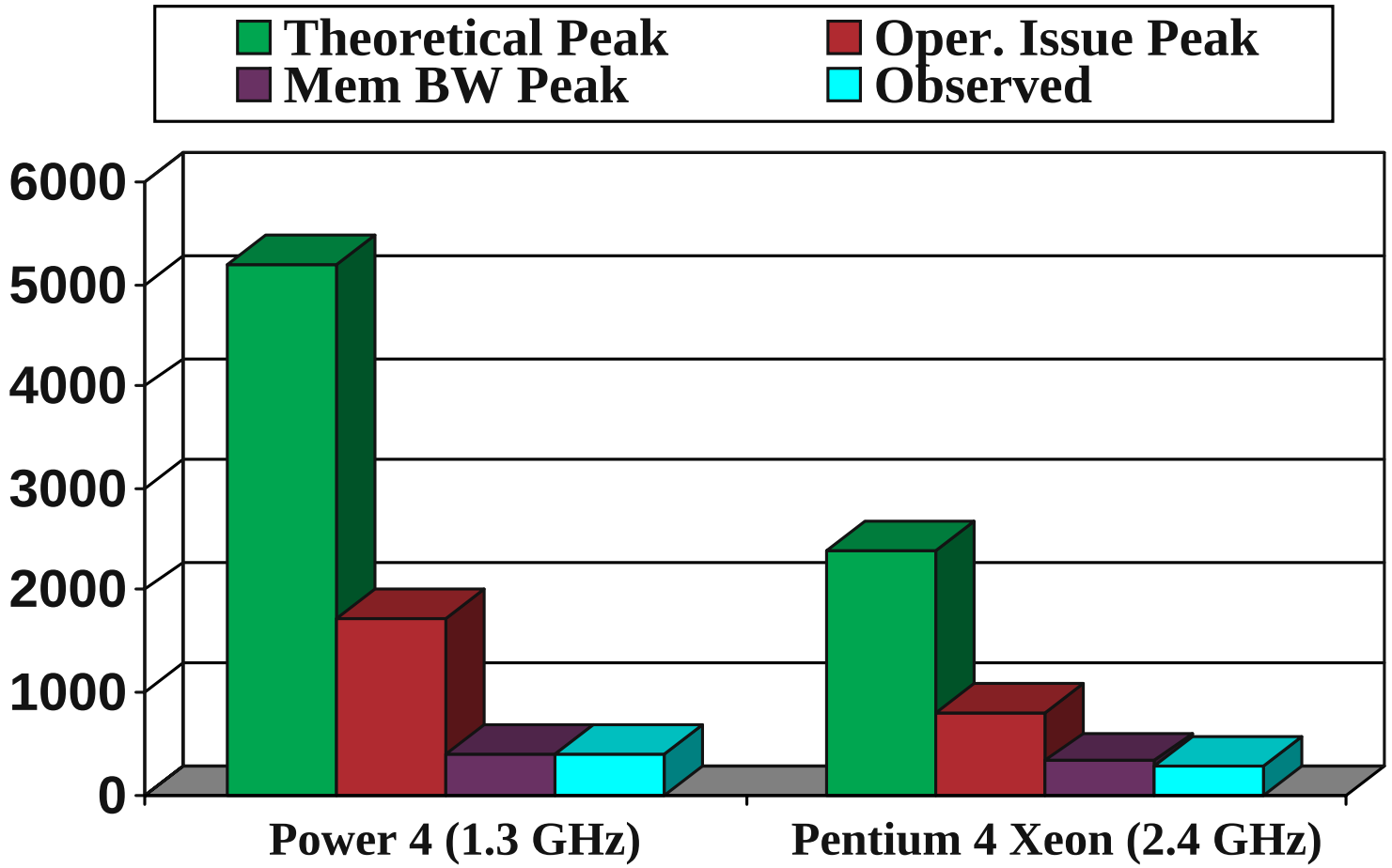
one vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



Realistic Measures of Peak Performance

Sparse Matrix Vector Product

One vector, matrix size, m = 90,708, nonzero entries nz = 5,047,120



Thanks to Dinesh Kaushik;
ORNL and ANL for compute time

Comments

- Simple model based on memory performance gives good ***bounds*** on performance
  - Detailed prediction requires much more work; often not necessary or relevant to the algorithm designer
- What do you do if observed performance is far short of predicted performance?
  - Take a closer look at the memory motion
  - Perfect cache assumption is often violated; nonlinear performance behavior gives significant cost to any miss
  - Define algorithm in terms of nested amounts of memory - create a *family* of algorithms
    - For example, in terms of blocks (matrices and meshes)
    - Note that they may not be numerically identical to the unblocked algorithm, so analysis is needed
  - One idea is *Cache oblivious algorithms*
    - Contain no parameters (or only a single minimum size)
    - Successful for dense matrix-matrix multiply

Adapting the Algorithm to Architecture

- Problem:
  - $u \in \Re^n$, $F(u) = 0$, representing nonlinear PDE on Domain Ω . Discretize.
- Typical Algorithmic decomposition:
  - Nonlinear problem $\rightarrow$
    Newton method $\rightarrow$
    Linear system involving Jacobian matrix $\rightarrow$
    Solve linear system in parallel
- However, limited temporal locality for linear solves, particularly for solvers such as multigrid
- One Solution: Cross - iteration algorithms
  - Think in blocks involving time/iteration, not just slices
  - Examples - CG methods, Nonlinear Schwarz

Nonlinear Schwarz Brings Back Memory Reuse

- An alternate approach:
  - Divide Ω into overlapping domains Ω_i, boundaries $\partial\Omega_i$ Let u_i be u restricted to Ω_i. $\partial\Omega_i \cap \Omega$ set from Ω_j
  - For k=0, ...
    Solve $F(u_1^{k+1},u_2^k) = 0$ for u_1^{k+1}on Ω_1,
    Solve $F(u_1^k,u_2^{k+1}) = 0$ for u_2^{k+1} on Ω_2, ...
- Each subdomain involves local (cache resident) solve
  - Choose Ω_i to fit in fast memory
  - Nonlinear methods are not (yet) O(1)
    - *Permit temporal locality*
  - Linear solvers used are O(1)
- Memory hierarchy handled through multilevel version
  - Solve $F(u_1^{k+1},u_2^k) = 0$ for u_1^{k+1}on $\Omega_1$with nonlinear Schwarz, etc.
- For an intro to memory issues for algorithm designers, see
  - Karp in SIAM Review 1996
  - McGeoch, AMS Notices March 2001

The Dimensions of a Typical Cluster

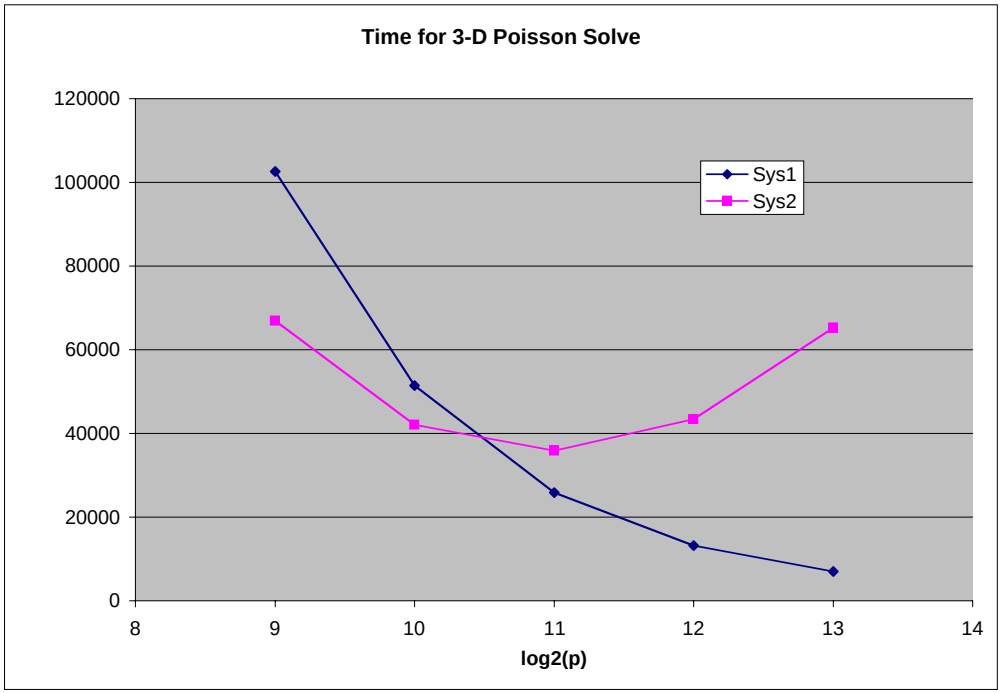
- 6.1 m x 2.1 m x 1m
- 1-norm size (airline baggage norm) = 9.2m
- At 2.4Ghz, =
  74 cycles
  (49 x 17 x 8)
- Real distance is greater
  - Routes longer
  - Signals slower than light in a vacuum
- PRAM (Parallel Random Access Memory) model is not helpful
  - Like using Newtonian mechanics for predicting behavior of near lightspeed particles
  - It is *too* simple

Two Challenges for Scalable Computing

- Amdahl's law
  - Maximum speed up = $1/(1-(T_s/(T_s+T_p)) = 1/(1-\text{serial\_fraction})$
  - For applications that require every bit of available memory (so called weak scaling), the serial_fraction is very small
  - For applications with fixed problem size (strong scaling), this is often already a problem
- Little's law
  - From queueing theory
    - *In a stable system, the arrival rate * the residency time equals the number in the queue*
  - For memory, we have
    - *Residency time = memory latency*
    - *Arrival rate = 1/clock*
    - *Thus number = memory latency in clocks*
  - This number is the number of outstanding operations, such as loads, or the number of concurrent operations needed to avoid waiting on memory
  - Typical values are 100-250

Achieving Good Scaling

- Solve a 3-D Poisson Problem as part of a larger application
- Algorithm is Multigrid preconditioned CG
- One system shows good (predicted) scaling
- The other eventually shows a slowdown
- Why?
  - System 1 has a special network for MPI_Allreduce. Cost is low and nearly constant
  - System 2 does not. Cost is (relatively) high



Reorganizing Conjugate Gradient for High Performance

- Problem:
  - Solve a linear system Ax = b
- Conjugate Gradient Method:
  - Iterate, computing $Ap^{(n)}$ at the n^{th} step
  - Form new approximate solution using dot products and vector operations
- Performance Problems
  - Dot products cause synchronization
  - Sparse matrix-vector products strain memory system

Effect of Inner Products

- Typical Krylov method

$\beta = r^T z$
$\rho = \beta / \beta_{old}$
$\beta_{old} = \beta$

$p = z + \rho\, p$
$z = Ap$
$\alpha = \beta / p^T z$
$x = x + \alpha\, p$
$r = r - \alpha\, z$
$z = Mr$

- Rearange to

start $\beta = r^T z$
$z_1 = Az$
end $\beta = r^T z$
$\rho = \beta / \beta_{old}$
$\beta_{old} = \beta$
$p = z + \rho p$
$z = z_1 + \rho z$ $(=Az + \rho Ap_{old} = Ap)$
$\alpha = \beta / p^T z$
$x = x + \alpha\, p$
$r = r - \alpha\, z$
$z = Mr$

- ***Not*** numerically identical (compiler *must not* do this)
- Deeper pipelining possible by further loop unrolling (also not numerically identical)

Comments

- No claim that this is the right thing to do
  - Illustrates the opportunities
- Must analyze tradeoffs
  - More floating point operations
  - More data motion
  - Less waiting for inner product
- General Ideas
  - Overlap communication with useful work
  - Consider cross (sub)step and cross iteration transformations
  - Initiate early, wait late
- This is a simple algebraic approach
  - The real solution is to apply these principles at the algorithmic level to gain much greater benefit

Massive Scale

- Load Balancing
  - Work virtualization
    - *Give the system flexibility to allocate resources*
- Fault Management
  - Fault detection
    - *Check, do not impose, conservation properties*
    - *Symmetries imply conservation principles - exploit them*
    - *“Assessing Fault Sensitivity in MPI Applications," SC2004 Best Technical Paper*
  - Fault Recovery
    - *Compact representations for checkpointing*
      - E.g., do you need every bit for a valid simulation, or could you store a set of coefficients for a FE representation to the computational accuracy? Is there a natural different representation that could be stored?
    - *What is the minimum amount of information that is needed?*
      - Don’t forget to cost-weight the information!
    - *Is there redundant information that can be used to reconstruct lost data?*

Not Everything is PetaScale computing

- Make better use of otherwise idle processors for laptop/deskktop computations
  - Error estimates and bounds
  - Sensitivity and Uncertainty Quantification
  - Serendipity
    - *Data mine previous results*
- Ubiquitous computing
  - Low power - results per watt-second (Joule) may be more important
- What Does Multicore/Manycore Imply?
  - It is and is not SMP
    - *Very high bandwidth, relatively low latency for memory "on chip"*
    - *Same problems (or worse!) to off-chip memory*
  - Using the extra processors with the same (on chip) memory
    - *Helper threads*
    - *Concurrent computation on same data*

Is Performance Everything?

"In August 1991, the Sleipner A, an oil and gas platform built in Norway for operation in the North Sea, sank during construction. The total economic loss amounted to about $700 million. After investigation, it was found that the failure of the walls of the support structure resulted from a serious error in the finite element analysis of the linear elastic model." (http://www.ima.umn.edu/~arnold/disasters/sleipner.html)

Even better, more accurate computations not only predicted the failure, they predicted the depth at which the structure would fail within 5%

What Can You Do?

- Re-examine old algorithms
  - Performance costs have changed
  - Increase memory efficiency of algorithms (higher order, better analysis instead of resolution refinement, …)
  - But some algorithms are still bad ideas
- Use performance bounds that include memory, CPU, and instruction mix
  - Consider performance an interval or a distribution, not a single number
  - Insist on relevant work bounds (estimates) in research papers and text books
- Reconsider problem decompositions
  - Overly simple decompositions (e.g., stratified solvers) waste CPU resources
  - Design around memory motion
    - *All machines suffer from memory wall*
      - Techniques to deal with memory wall differ but none eliminate it
  - Split operations that make use of “far away” resources
- Create families of algorithms that adapt to memory and concurrency
  - These provide a way for CS folks to help adapt the algorithm to the complexities of the hardware