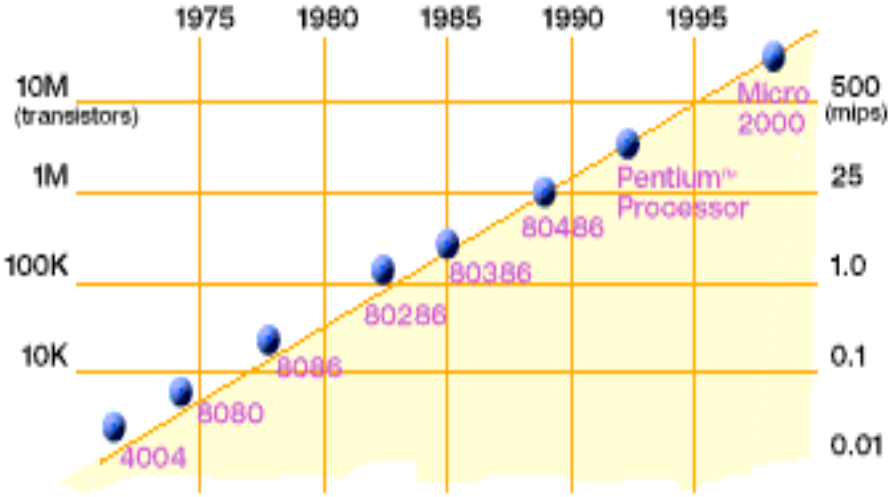
Multicore Meets Petascale: Catalyst for a Programming Model Revolution

Kathy Yelick

U.C. Berkeley and

Lawrence Berkeley National Laboratory

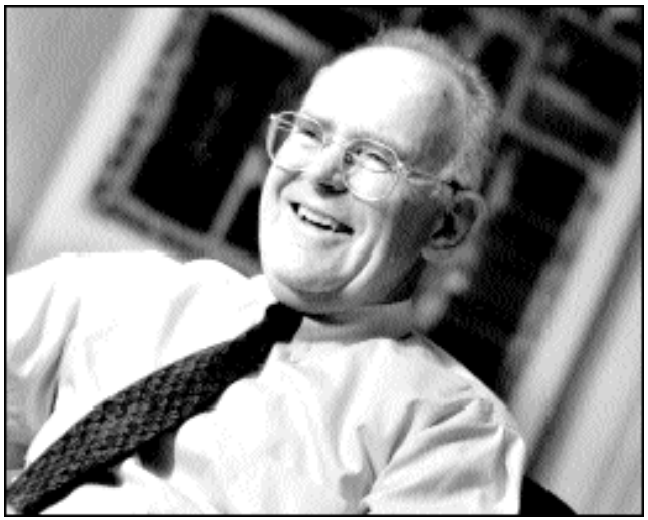
Moore's Law is Alive and Well



2X transistors/Chip Every 1.5 years

Called "Moore's Law"

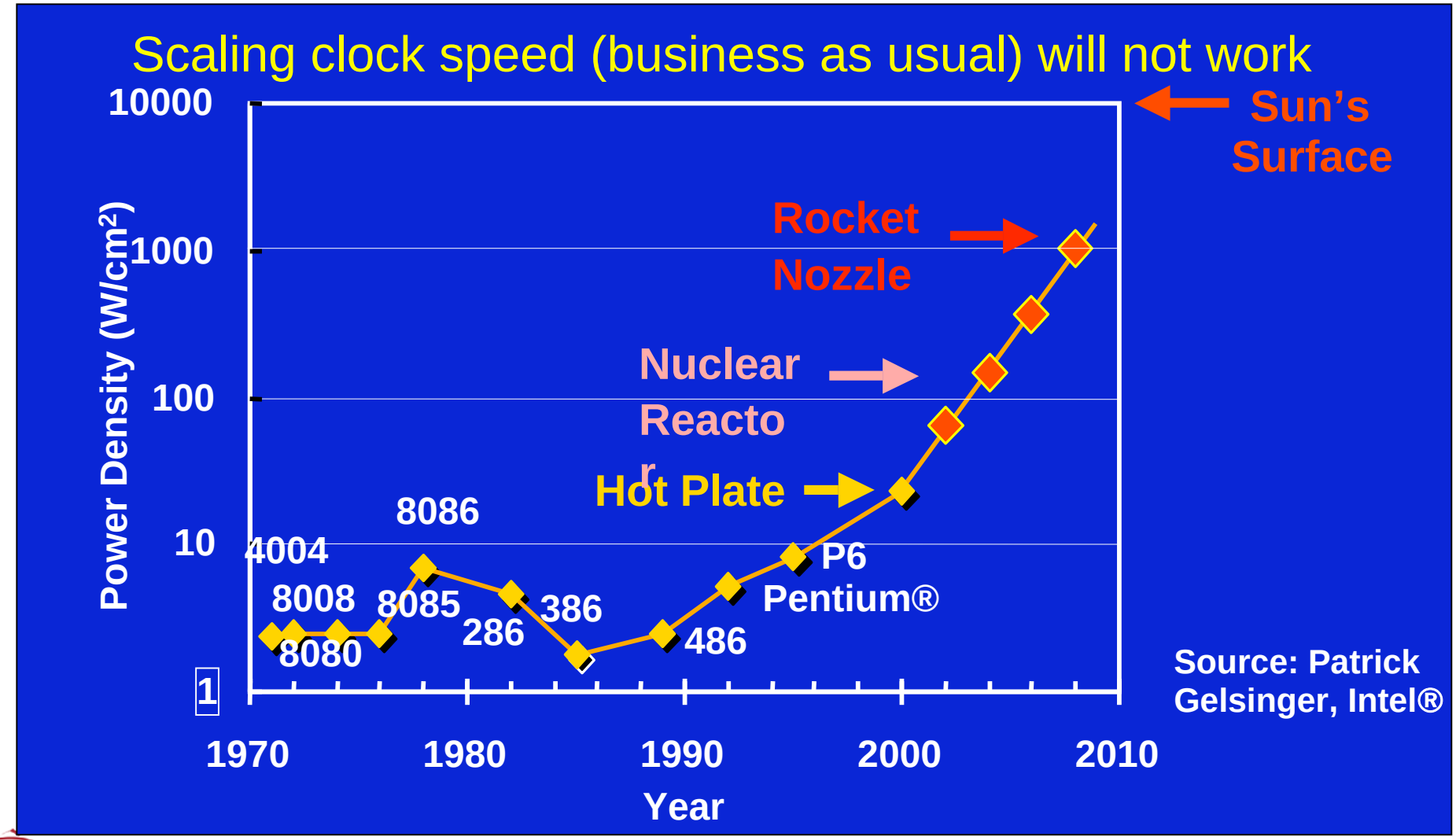
Microprocessors have become smaller, denser, and more powerful.



Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

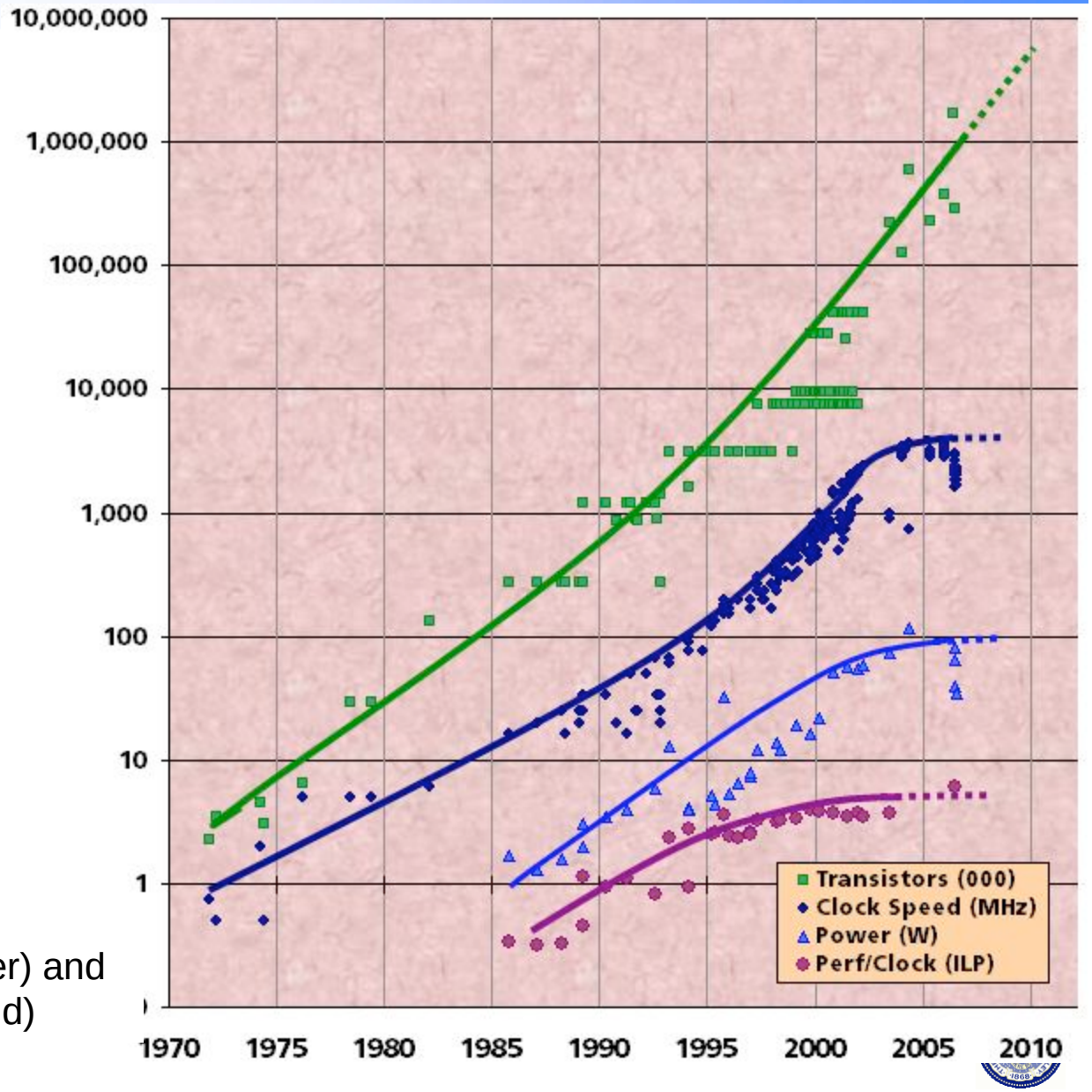
Slide source: Jack Dongarra

Clock Scaling Hits Power Density Wall



Multicore Revolution

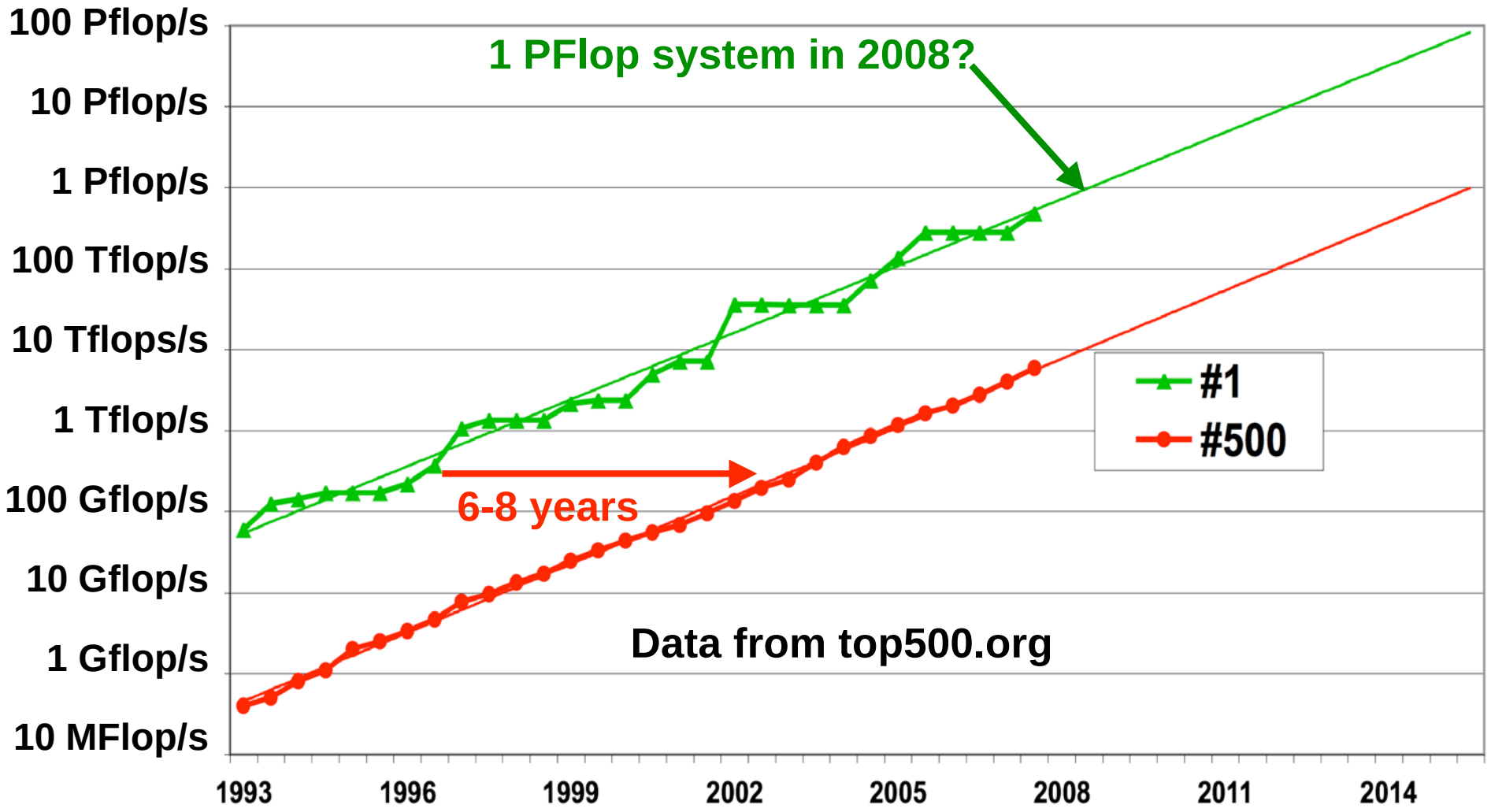
- **Chip density is continuing increase ~2x every 2 years**
  - **Clock speed is not**
  - **Number of processor cores may double instead**
- **There is little or no hidden parallelism (ILP) to be found**
- **Parallelism must be exposed to and managed by software**



Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)

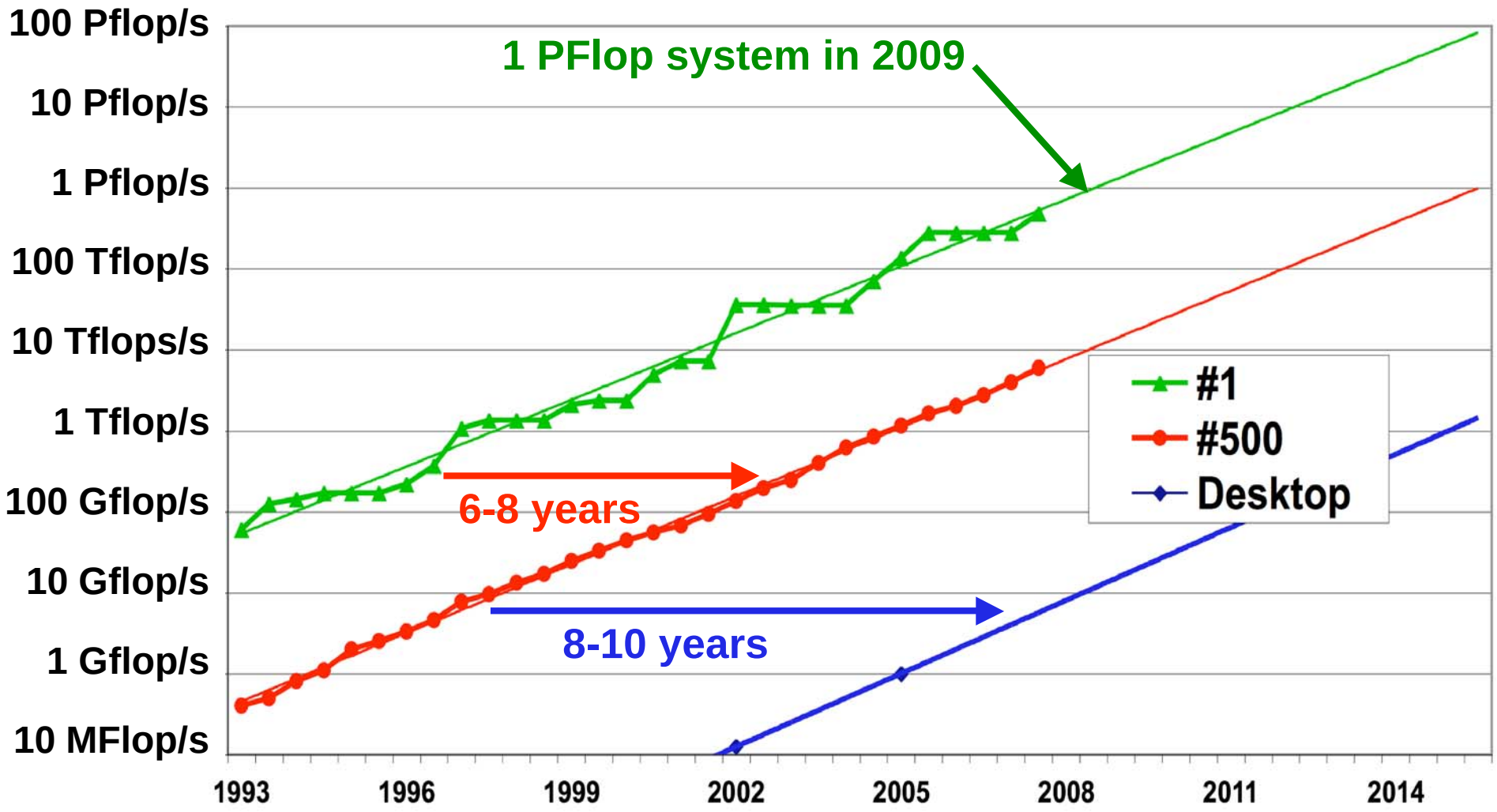
Petaflop with ~1M Cores

Common by 2015?



Petaflop with ~1M Cores

On your desk in 2025?



Need a Fundamentally New Approach

- **Rethink hardware**
  - What limits performance
  - How to build efficient hardware
- **Rethink software**
  - Massive parallelism
  - Eliminate scaling bottlenecks replication, synchronization
- **Rethink algorithms**
  - Massive parallelism and locality
  - Counting Flops is the wrong measure

Rethink Hardware

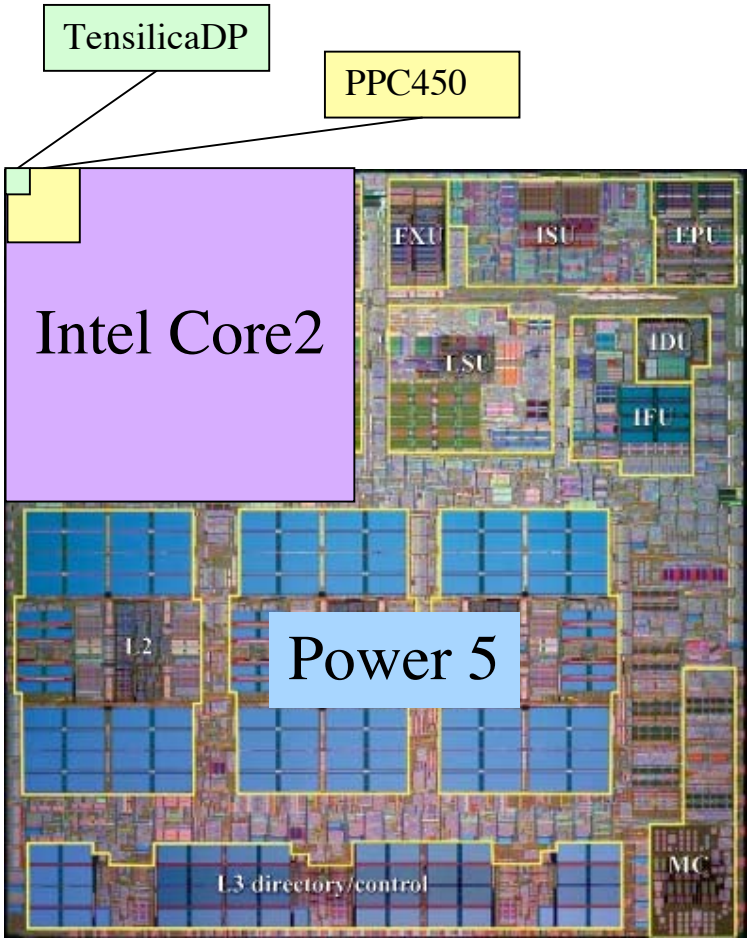
(Ways to Waste $50M)

Waste #1: Ignore Power Budget

Power is top concern in hardware design

- **Power density within a chip**
  - **Led to multicore revolution**
- **Energy consumption**
  - **Always important in handheld devices**
  - **Increasingly so in desktops**
  - **Soon to be significant fraction of budget in large systems**
- **One knob: increase concurrency**

Optimizing for Serial Performance Consumes Power



- **Power5 (Server)**
  - 389 mm²
  - 120 W @ 1900 MHz
- **Intel Core2 sc (Laptop)**
  - 130 mm²
  - 15 W @ 1000 MHz
- **PowerPC450 (BlueGene/P)**
  - 8 mm²
  - 3 W @ 850 MHz
- **Tensilica DP (cell phones)**
  - 0.8 mm²
  - 0.09 W @ 650 MHz

Each core operates at 1/3 to 1/10th efficiency of largest chip, but you can pack 100x more cores onto a chip and consume 1/20 the power!

BERKELEY LAB

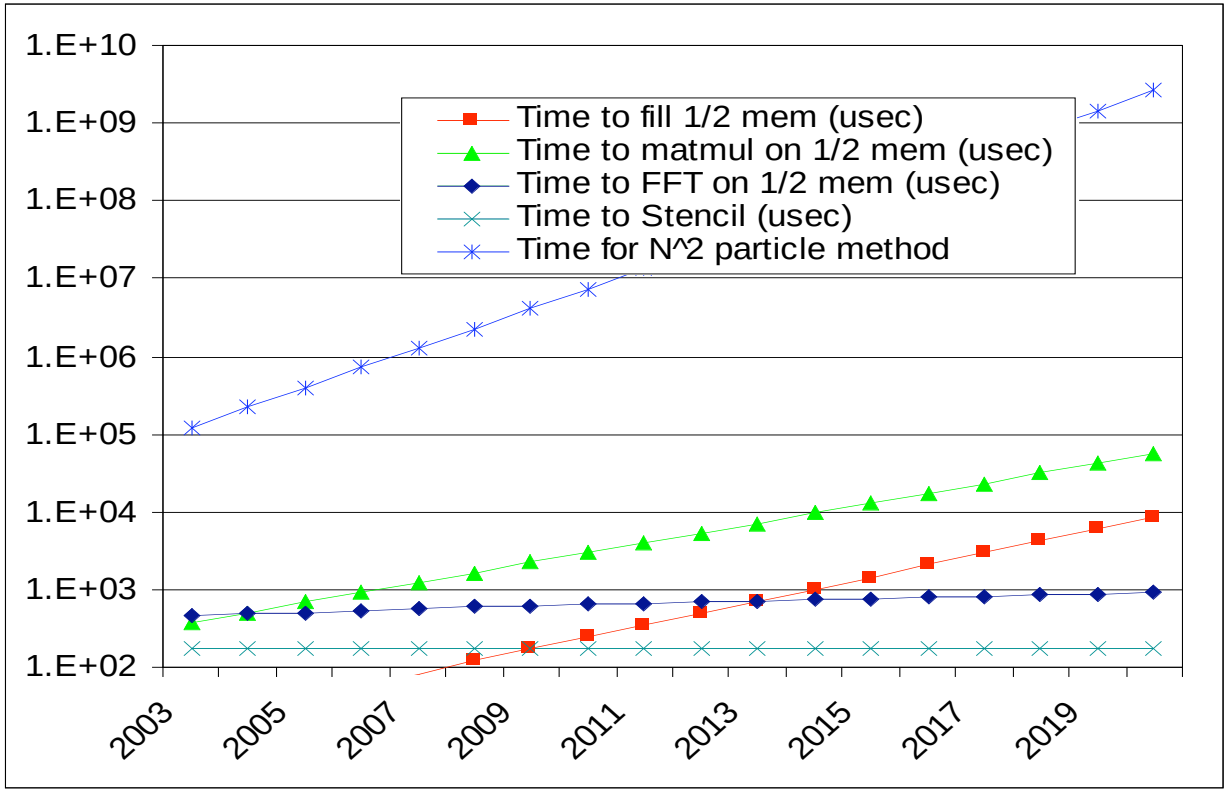
Power Demands Threaten to Limit the Future Growth of Computational Science

Looking forward to Exascale (1000x Petascale)

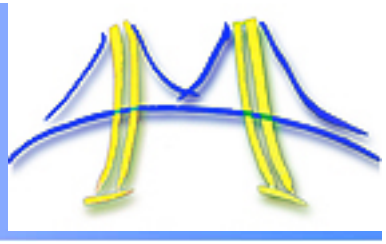
- **DOE E3 Report**
  - Extrapolation of existing design trends
  - Estimate: 130 MW
- **DARPA Exascale Study**
  - More detailed assessment of component technologies
    - Power-constrained design for 2014 technology
    - 3 TF/chip, new memory technology, optical interconnect
  - Estimate:
    - 40 MW plausible target, not business as usual
    - Billion-way concurrency with cores and latency-hiding
- **NRC Study**
  - Power and multicore challenges are not just an HPC problem

Waste #2: Buy More Cores than Memory can Feed

- **Required bandwidth depends on the algorithm**
- **Need hardware designed to algorithmic needs**



Waste #3: Ignore Little's Law-- Latency also Matters

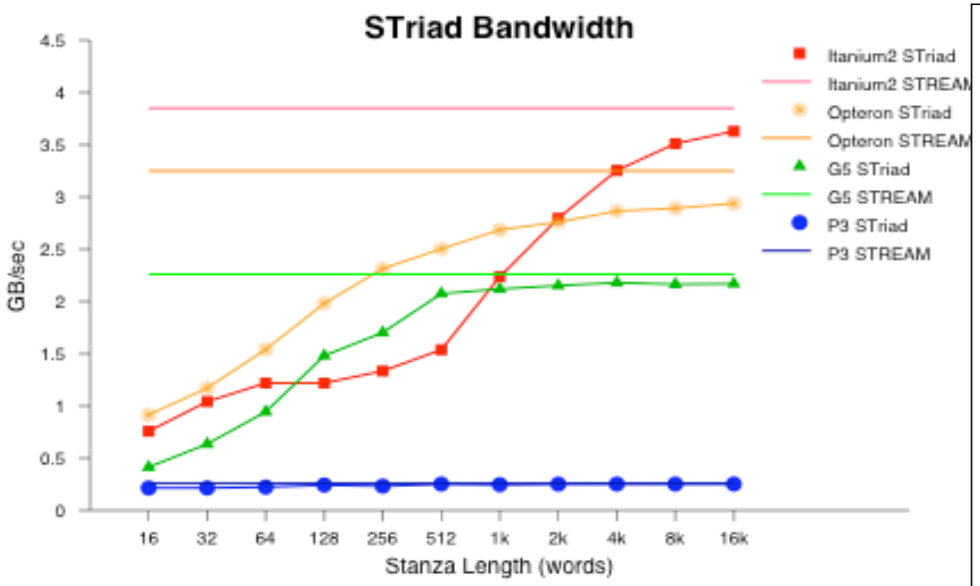


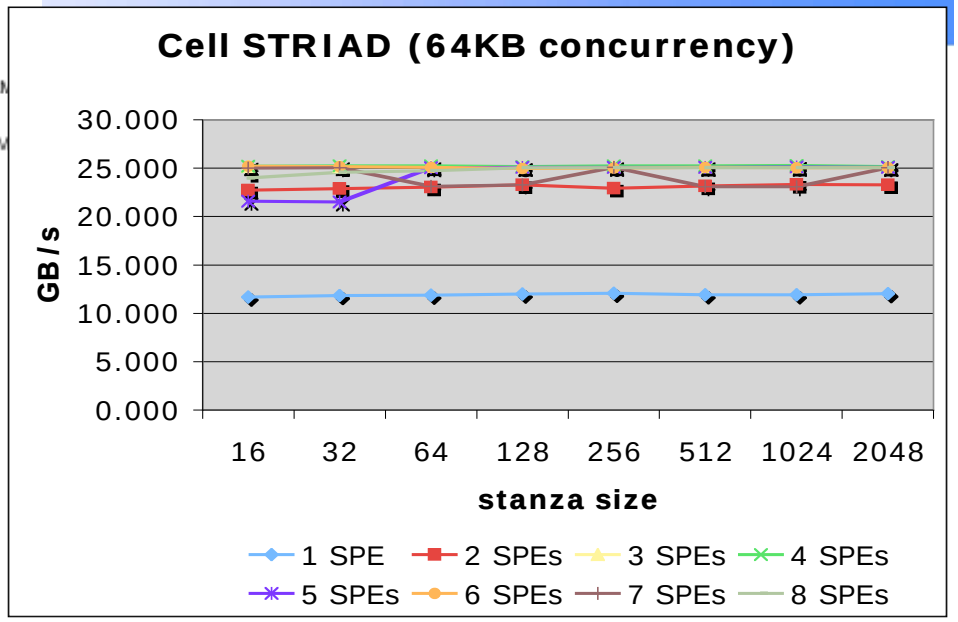
Sparse Matrix-Vector Multiply (2flops / 8 bytes) should be BW limited

Name	Clovertown	Opteron	Cell
Chips*Cores	2*4 = 8	2*2 = 4	1*8 = 8
Architecture	4-/3-issue, 2-/1-SSE3, OOO, caches, prefetch		2-VLIW, SIMD, local RAM, DMA
Clock Rate	2.3 GHz	2.2 GHz	3.2 GHz
Peak MemBW	21.3 GB/s	21.3	25.6 GB/s
Peak GFLOPS	74.6 GF	17.6 GF	14.6 (DP Fl. Pt.)
Naïve SpMV (median of many matrices)	1.0 GF	0.6 GF	--
Efficiency %	1%	3%	--

Why is the STI Cell So Efficient?

(Latency Hiding with Software Controlled Memory)





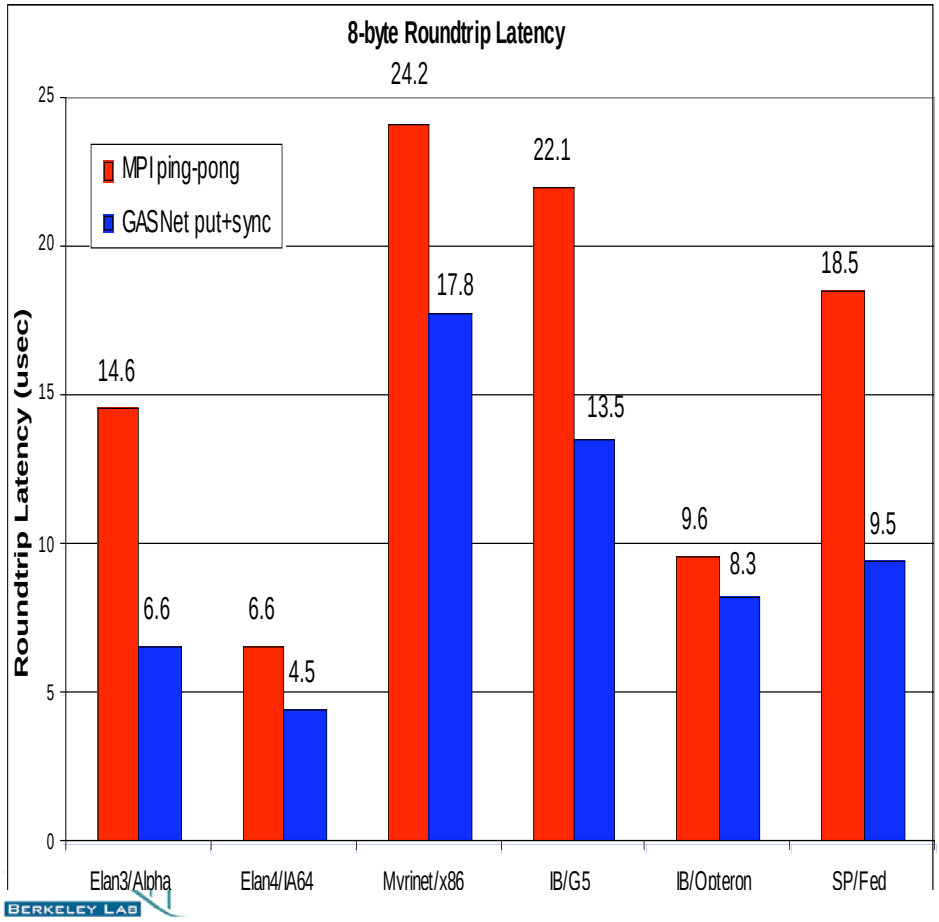
- **Performance of Standard Cache Hierarchy**
  - **Cache hierarchies are insufficient to tolerate latencies**
  - **Hardware prefetch prefers long unit-stride access patterns (optimized for STREAM)**
- **Cell "explicit DMA"**
  - **Cell software controlled DMA engines can provide nearly flat bandwidth**
  - **Performance model is simple and deterministic (much simpler than modeling a complex cache hierarchy),**

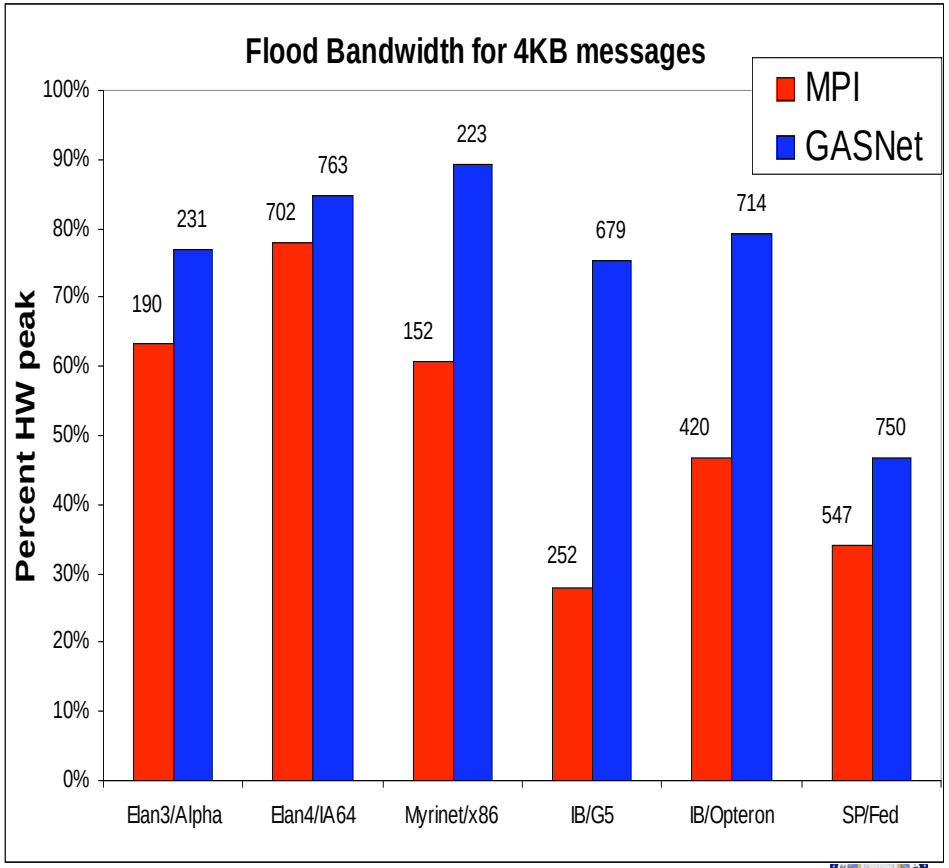
    min{time_for_memory_ops, time_for_core_exec}

BERKELEY LAB

Waste #4: Unnecessarily Synchronize Communication

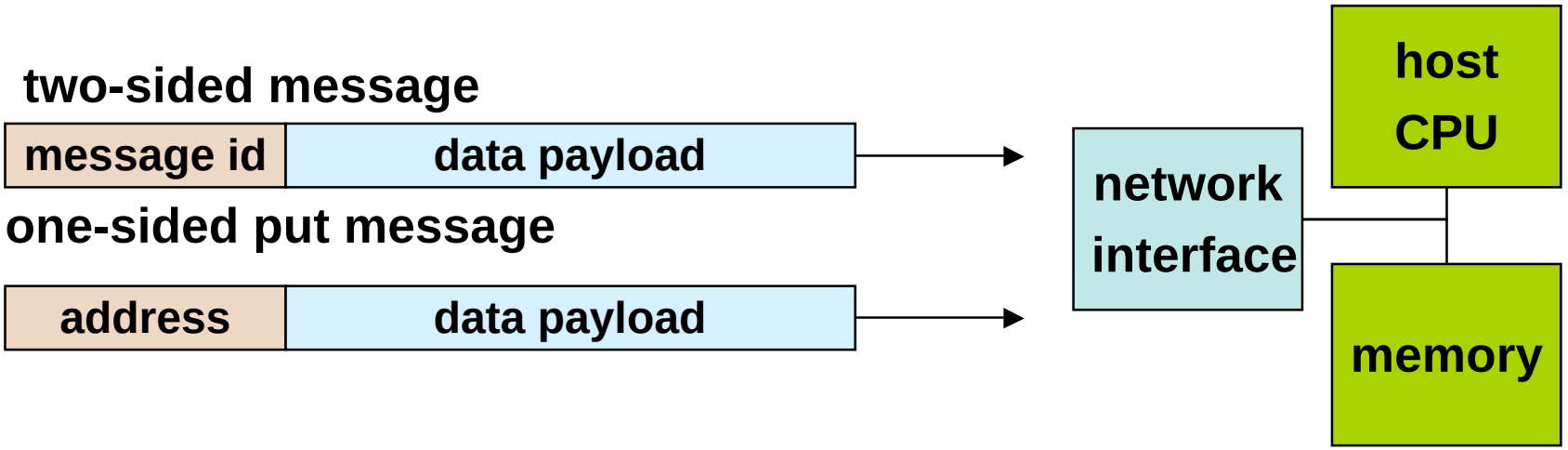
- **MPI Message Passing is two-sided: each transfer is tied to a synchronization event (message received)**
- **One-sided communication avoids this**





BERKELEY LAB

One-Sided vs Two-Sided Communication



- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)

Joint work with Dan Bonachea

Rethinking Programming Models

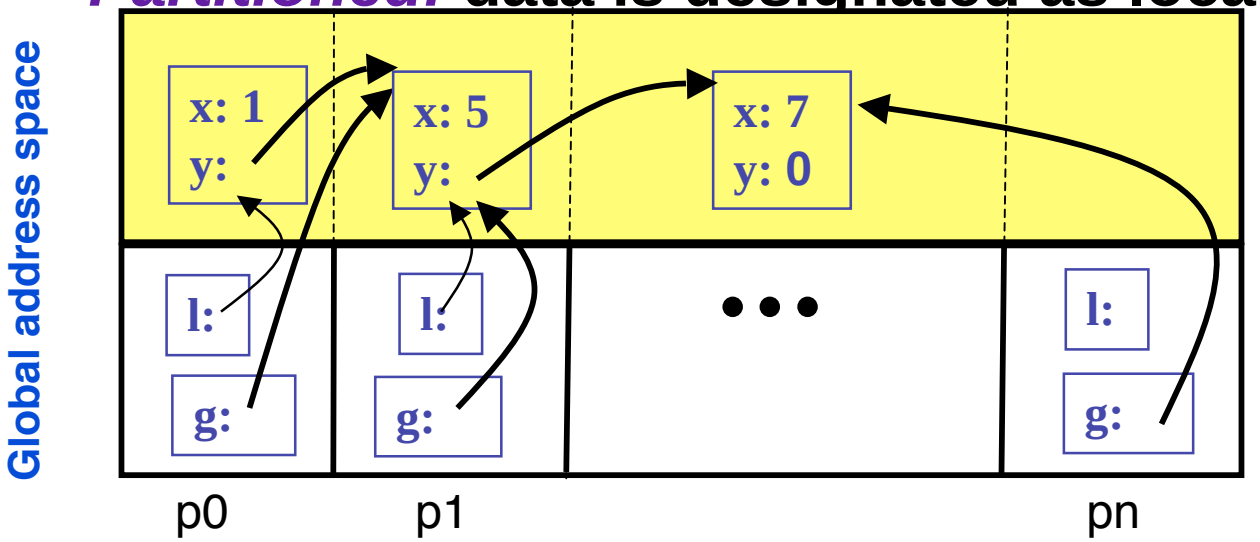
for
Massive concurrency
Latency hiding
Locality control

Parallel Programming Models

- **Most parallel programs are written using either:**
  - **Message passing with a SPMD model**
    - **Scientific applications; is portable and scalable**
    - **Success driven by cluster computing**
  - **Shared memory with threads in OpenMP or Threads**
    - **IT applications; easier to program**
    - **Used on shared memory architectures**
- **Future Petascale machines**
  - **Massively distributed, O(100K) nodes/sockets**
  - **Massively parallelism within a chip (2x every year, starting at 4-100 now)**
- **Main question: 1 programming model or 2?**
  - **MPI + OpenMP or something else**

Partitioned Global Address Space

- *Global address space:* any thread/process may directly read/write data allocated by another
- *Partitioned:* data is designated as local or global



By default:

- Object heaps are shared
- Program stacks are private

- *3 Current languages:* UPC, CAF, and Titanium
  - All three use an SPMD execution model
  - Emphasis in this talk on UPC and Titanium (based on Java)
- *3 Emerging languages:* X10, Fortress, and Chapel

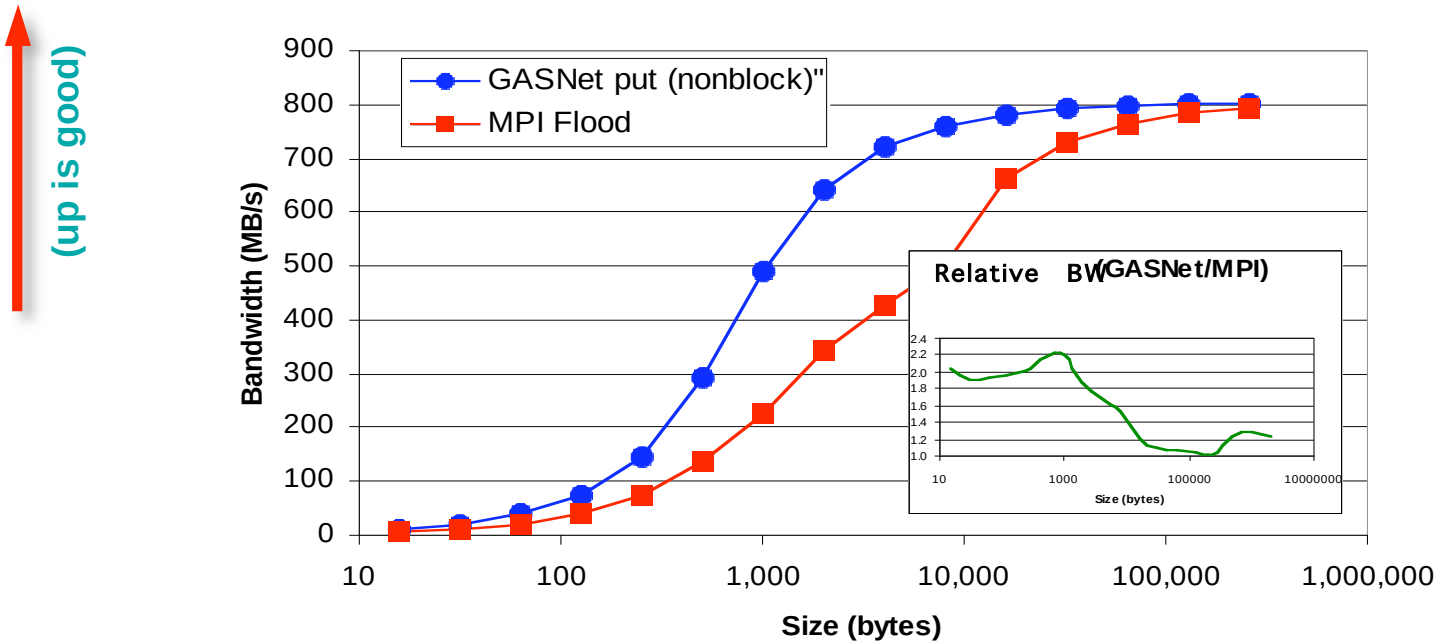
PGAS Language Overview

- **Many common concepts, although specifics differ**
  - **Consistent with base language, e.g., Titanium is strongly typed**
- **Both private and shared data**
  - int x[10]; **and** shared int y[10];
- **Support for distributed data structures**
  - **Distributed arrays; local and global pointers/references**
- **One-sided shared-memory communication**
  - **Simple assignment statements:** x[i] = y[i]; **or** t = *p;
  - **Bulk operations: memcpy in UPC, array ops in Titanium and CAF**
- **Synchronization**
  - **Global barriers, locks, memory fences**
- **Collective Communication, IO libraries, etc.**

PGAS Languages for Distributed Memory

- **PGAS languages are a good fit to distributed memory machines and clusters of multicore**
  - **Global address space uses fast one-sided communication**
  - **UPC and Titanium use GASNet communication**
- **Alternative on distributed memory is MPI**
  - **PGAS partitioned model scaled to 100s of nodes**
  - **Shared data structure are only in the programmer's mind in MPI; cannot be programmed as such**

One-Sided vs. Two-Sided: Practice



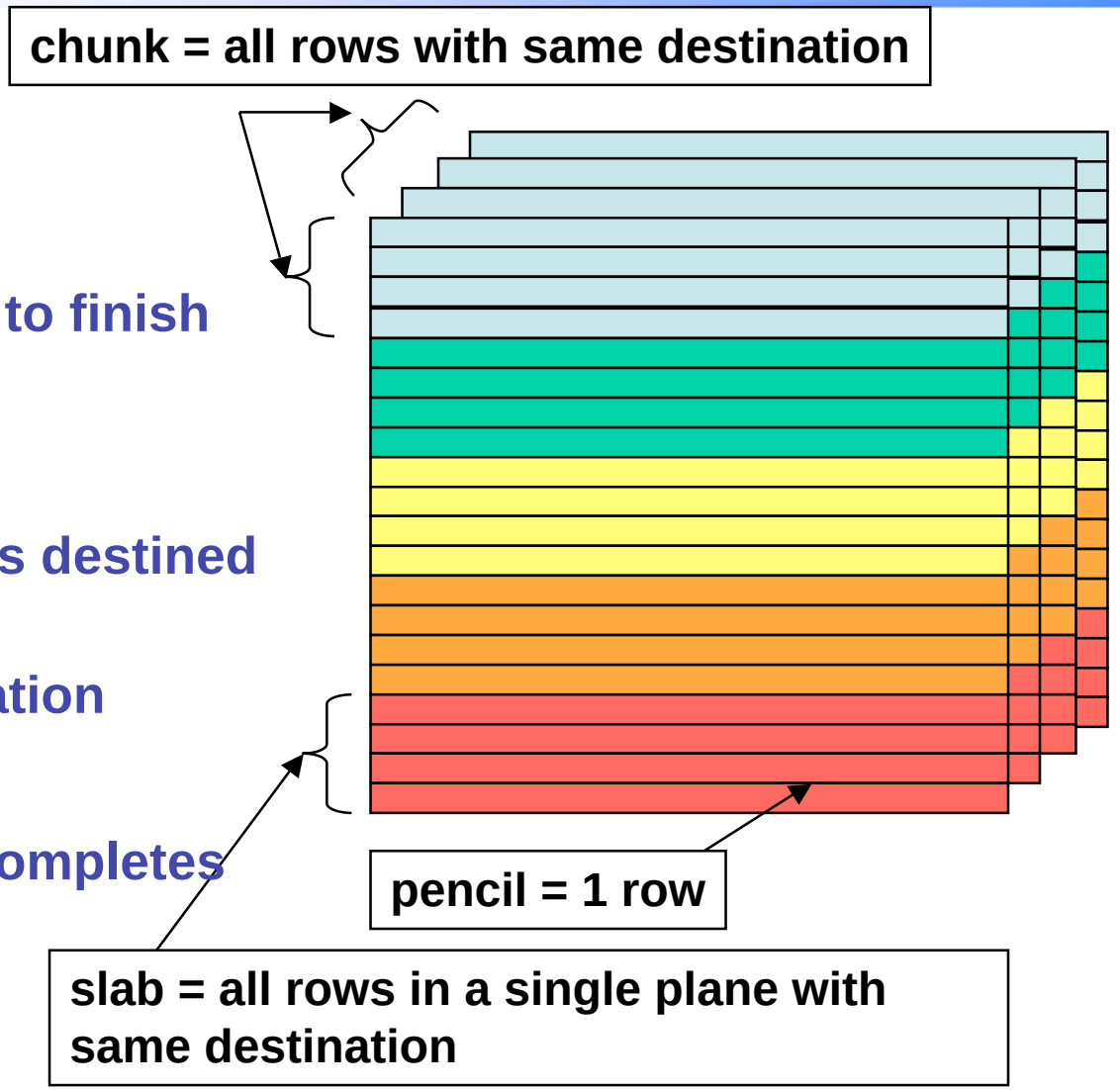
NERSC Jacquard machine with Opteron processors

- **InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5**
- **Half power point (N ½) differs by *one order of magnitude***
- **This is not a criticism of the implementation!**

Joint work with Paul Hargrove and Dan Bonachea

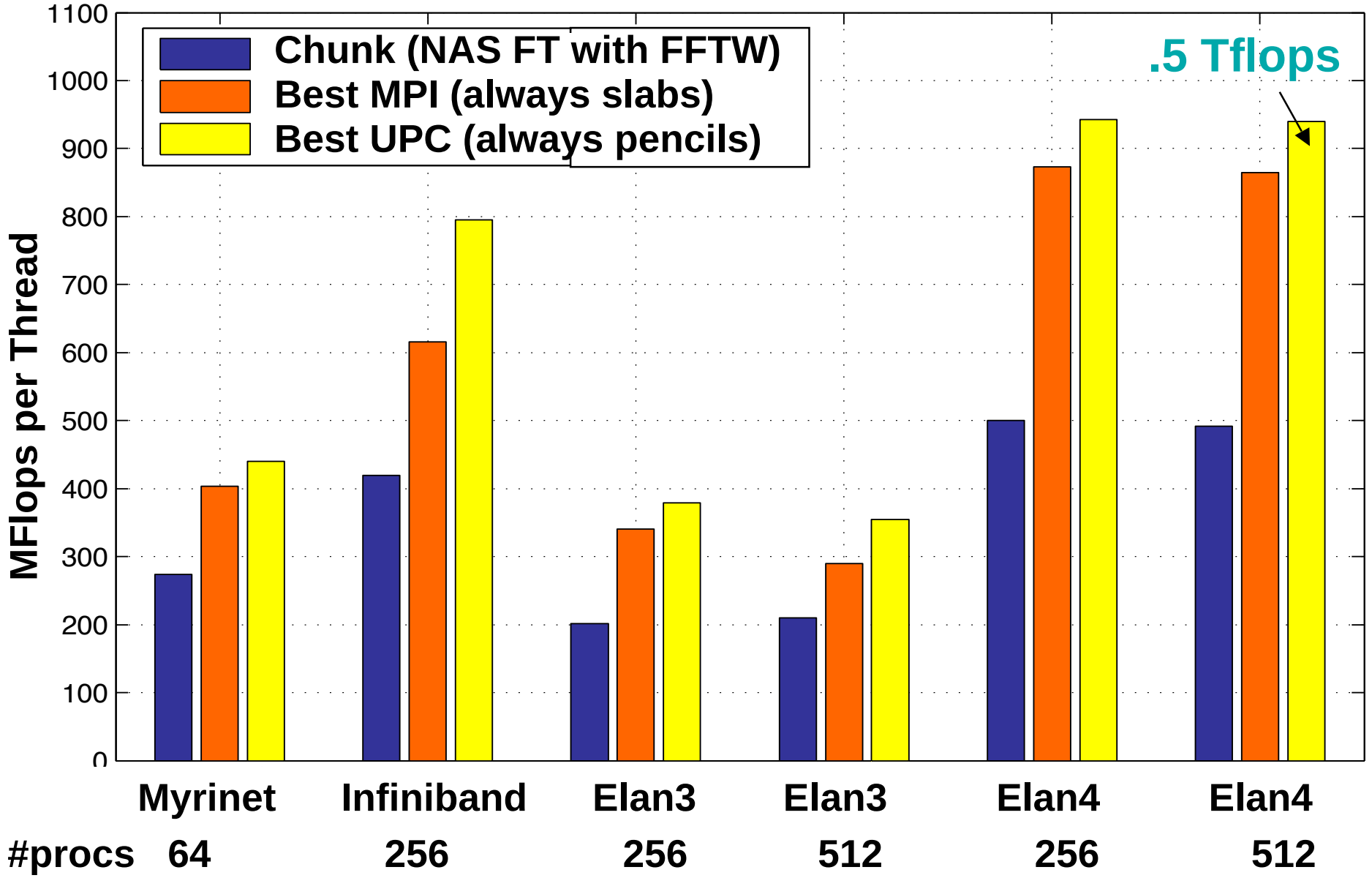
Communication Strategies for 3D FFT

- **Three approaches:**
  - –Chunk:
    - **Wait for 2nd dim FFTs to finish**
    - **Minimize # messages**
  - –Slab:
    - **Wait for chunk of rows destined for 1 proc to finish**
    - **Overlap with computation**
  - –Pencil:
    - **Send each row as it completes**
    - **Maximize overlap and**
    - **Match natural layout**



Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea

NAS FT Variants Performance Summary



PGAS Languages and Productivity

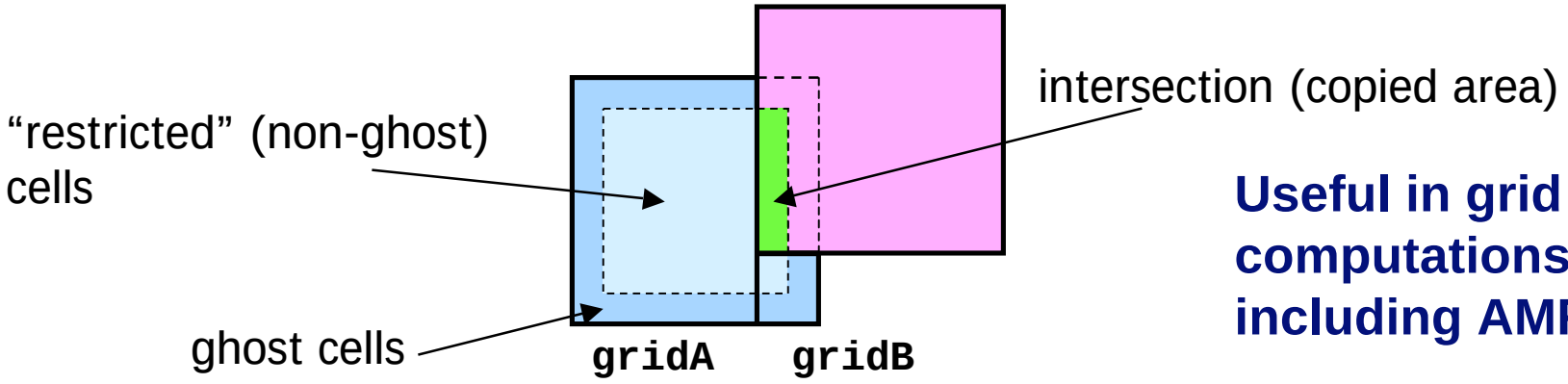
Arrays in a Global Address Space

- **Key features of Titanium arrays**
  - **Generality: indices may start/end and any point**
  - **Domain calculus allow for slicing, subarray, transpose and other operations without data copies**
- **Use domain calculus to identify ghosts and iterate:**

```
foreach (p in gridA.shrink(1).domain()) ...
```

- **Array copies automatically work on intersection**

```
gridB.copy(gridA.shrink(1));
```



Useful in grid computations including AMR

Joint work with Titanium group

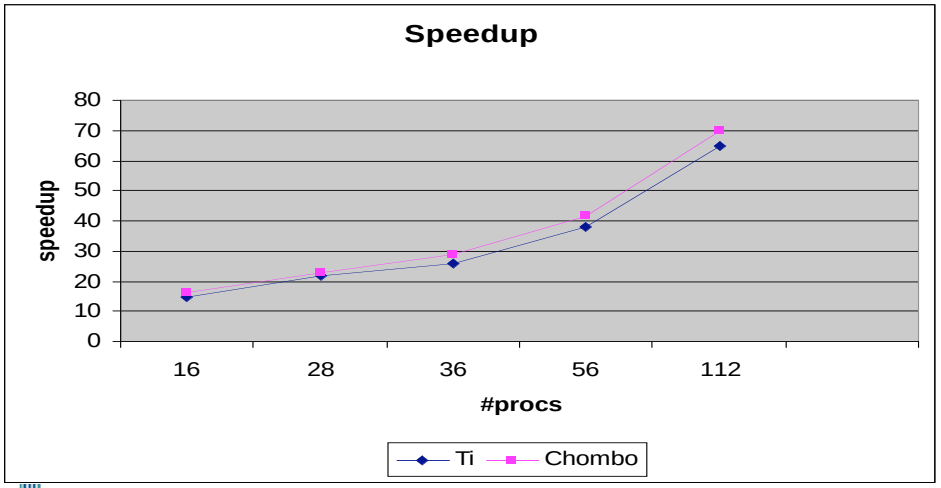
Languages Support Helps Productivity

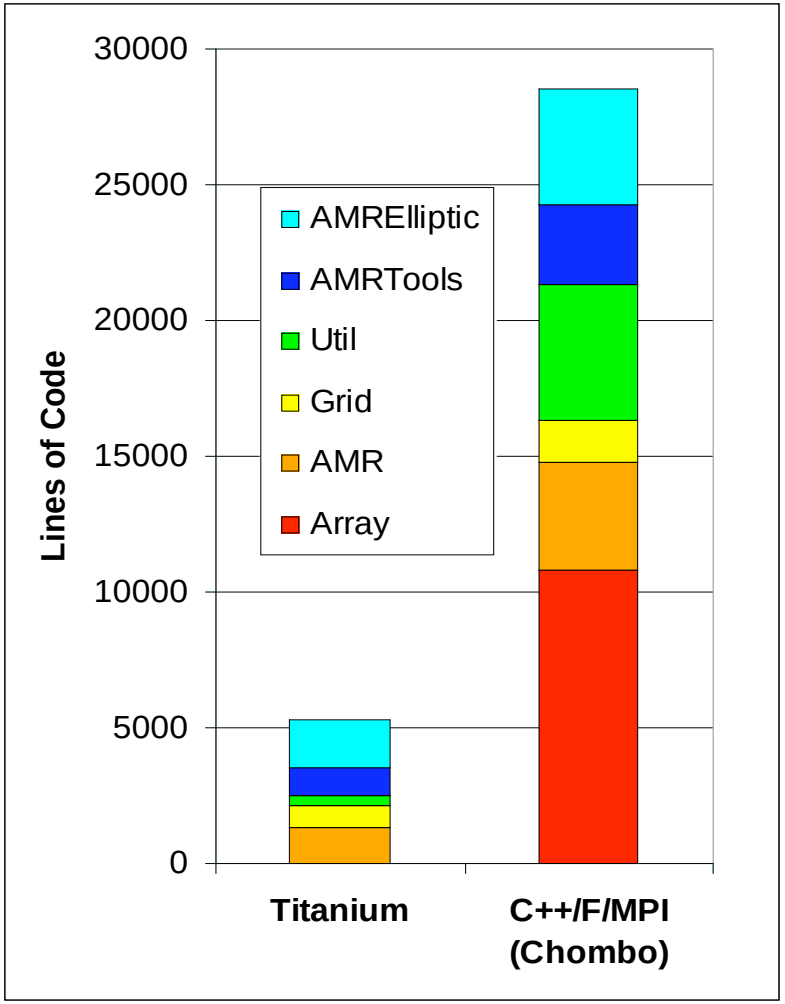
C++/Fortran/MPI AMR

- Chombo package from LBNL
- Bulk-synchronous comm:
  - Pack boundary data between procs
  - All optimizations done by programmer

Titanium AMR

- Entirely in Titanium
- Finer-grained communication
  - No explicit pack/unpack code
  - Automated in runtime system
- General approach
  - Language allow programmer optimizations
  - Compiler/runtime does some automatically



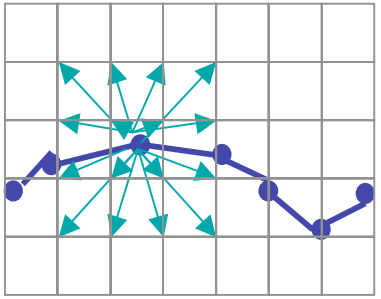


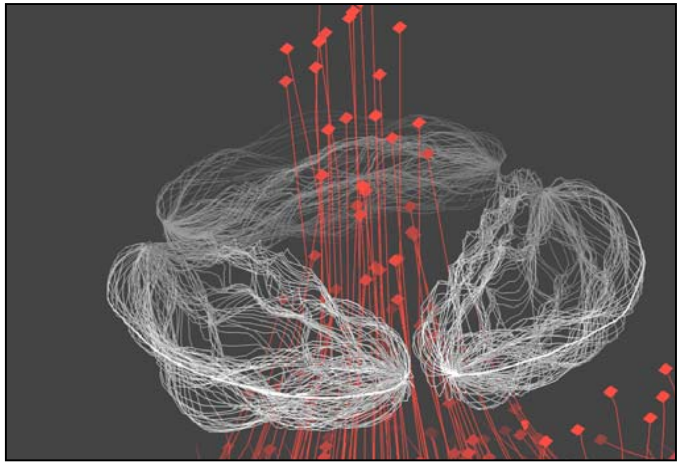
Work by Tong Wen and Philip Colella; Communication optimizations joint with Jimmy Su

Particle/Mesh Method: Heart Simulation

- **Elastic structures in an incompressible fluid.**
  - Blood flow, clotting, inner ear, embryo growth, …
- **Complicated parallelization**
  - Particle/Mesh method, but "Particles" connected into materials (1D or 2D structures)
  - Communication patterns irregular between particles (structures) and mesh (fluid)

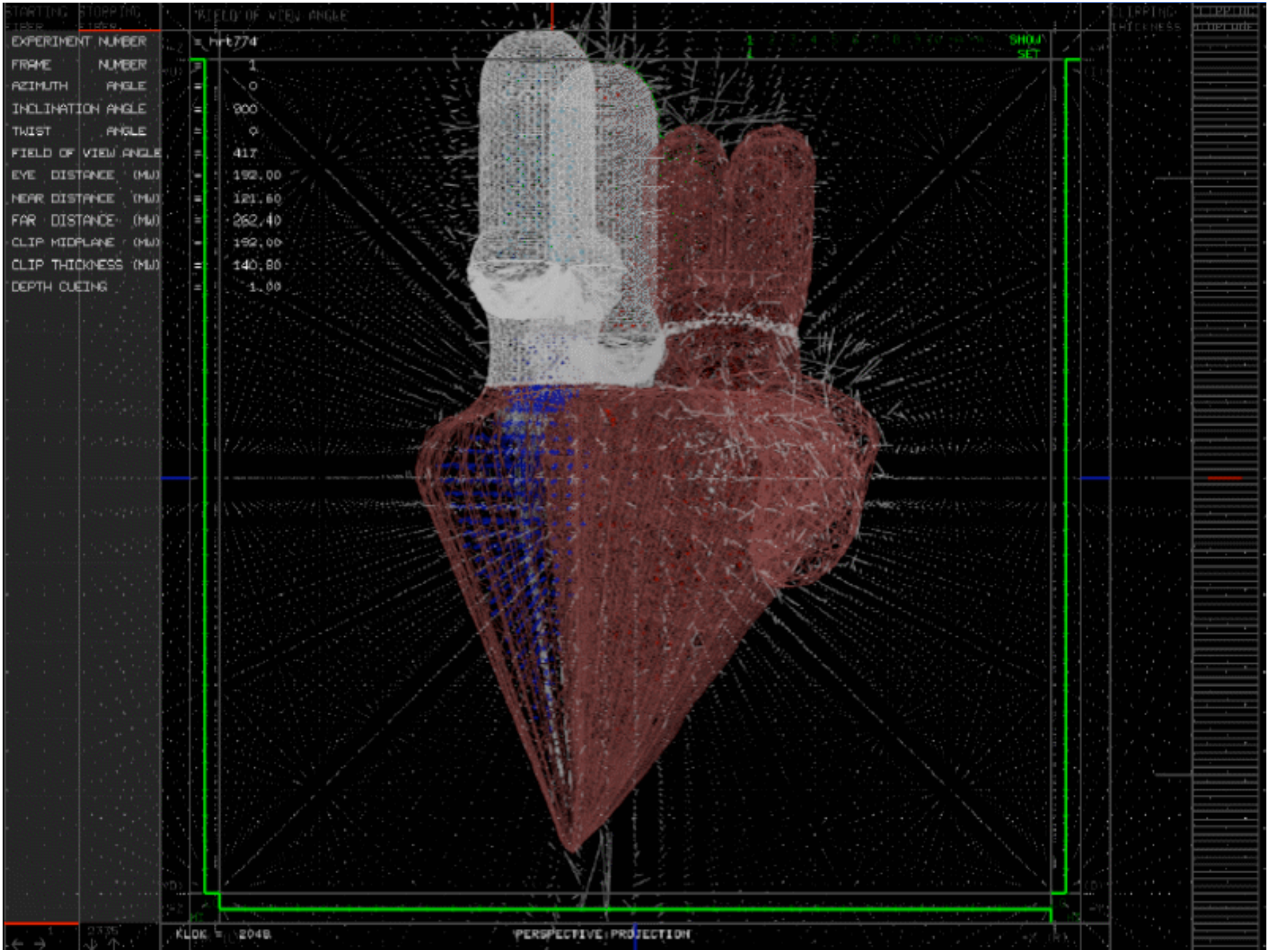
2D Dirac Delta Function



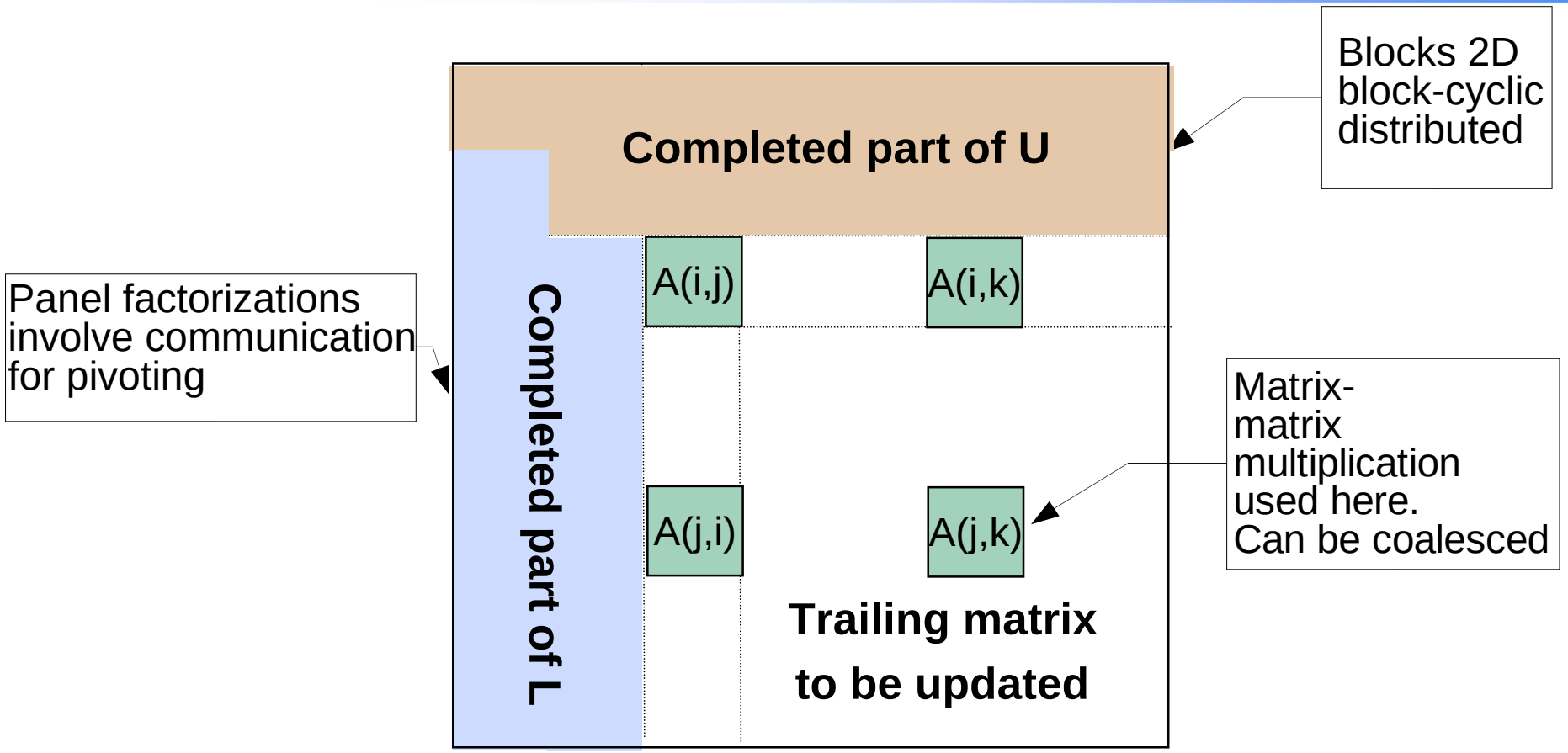


Code Size in Lines	
Fortran	Titanium
8000	4000

Note: Fortran code is not parallel


STARTING STOPPING
FIBER FIBER
FIELD OF VIEW ANGLE
EXPERIMENT NUMBER
FRAME NUMBER
AZIMUTH ANGLE
INCLINATION ANGLE
TWIST ANGLE
FIELD OF VIEW ANGLE
EYE DISTANCE (MU)
NEAR DISTANCE (MU)
FAR DISTANCE (MU)
CLIP MIDPLANE (MU)
CLIP THICKNESS (MU)
DEPTH CUEING
= mrt774
1
0
900
0
417
192.00
121.60
262.40
192.00
140.80
1.00
SHOW
SET
CLIPPING
THICKNESS
KLOK = 2048
PERSPECTIVE PROJECTION

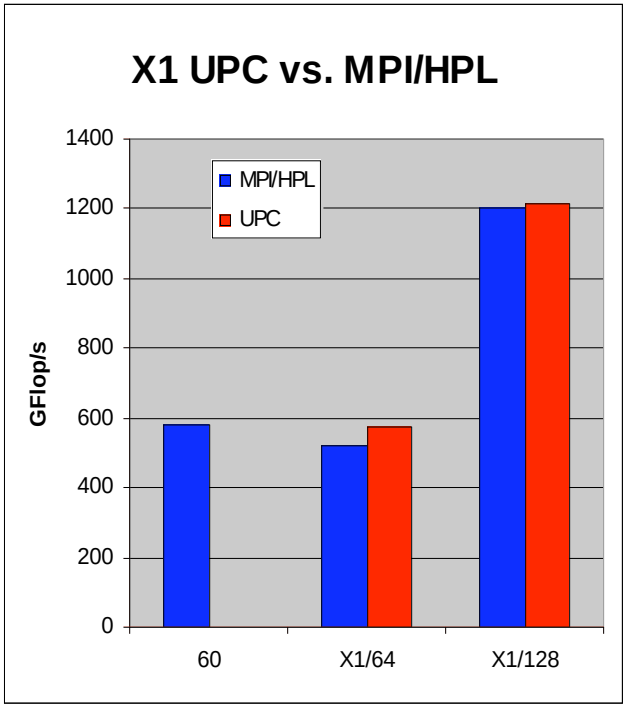
Dense and Sparse Matrix Factorization

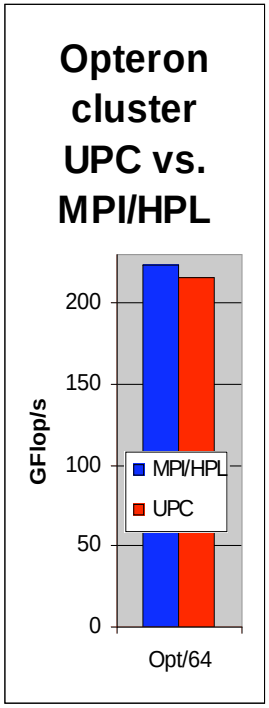


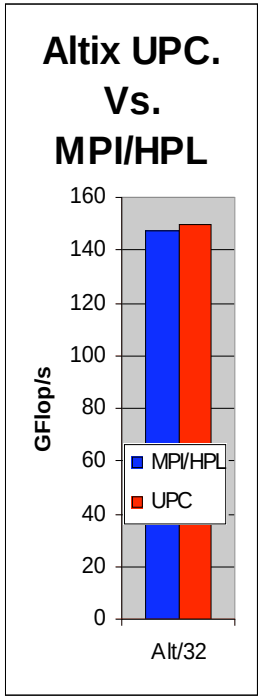
Matrix Factorization in UPC

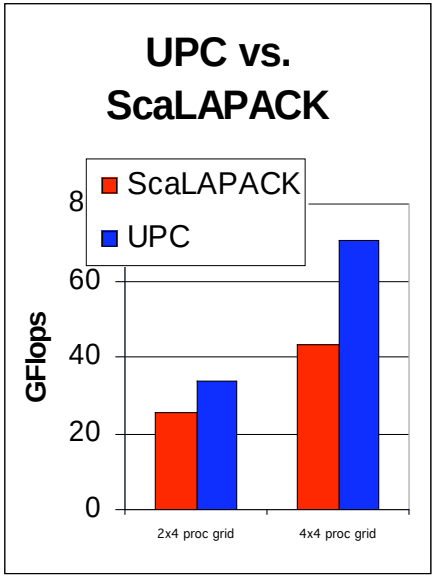
- **UPC factorization uses a highly multithreaded style**
  - Used to mask latency and to mask dependence delays
  - Three levels of threads:
    - UPC threads (data layout, each runs an event scheduling loop)
    - Multithreaded BLAS (boost efficiency)
    - User level (non-preemptive) threads with explicit yield
  - No dynamic load balancing, but lots of remote invocation
  - Layout is fixed (blocked/cyclic) and tuned for block size
- **Same framework being used for sparse Cholesky**
- **Hard problems**
  - Block size tuning (tedious) for both locality and granularity
  - Task prioritization (ensure critical path performance)
  - Resource management can deadlock memory allocator if not careful

UPC HP Linpack Performance









- **Comparable to MPI HPL (numbers from HPCC database)**
- **Faster than ScaLAPACK due to less synchronization**
- **Large scaling of UPC code on Itanium/Quadrics (Thunder)**
  - 2.2 TFlops on 512p and 4.4 TFlops on 1024p

Joint work with Parry Husbands

BERKELEY LAB

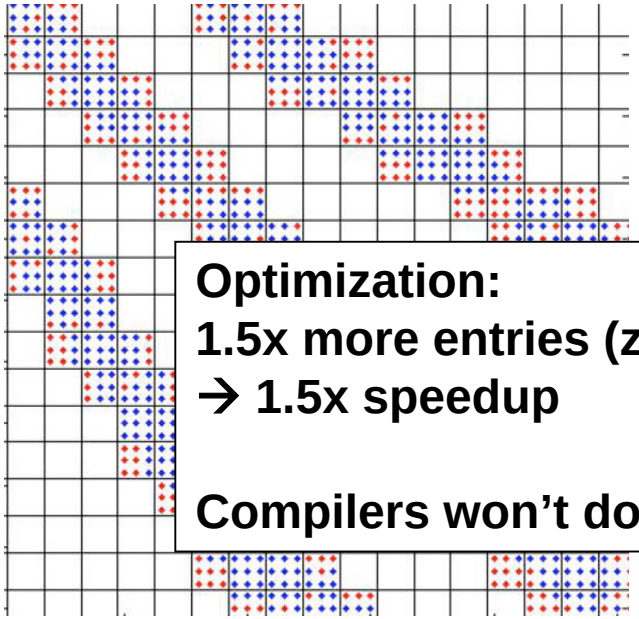
~~PGAS Languages~~ + Autotuning for Multicore

DMA

- **PGAS languages are a good fit to shared memory machines, including multicore**
  - **Global address space implemented as reads/writes**
  - **Current UPC and Titanium implementation uses threads**
- **Alternative on shared memory is OpenMP or threads**
  - **PGAS has locality information that is important on multi-socket SMPs and may be important as #cores grows**
  - **Also may be exploited for processor with explicit local store rather than cache, e.g., Cell processor**
- **Open question in architecture**
  - **Cache-coherence shared memory**
  - **Software-controlled local memory (or hybrid)**
- **How do we get *performance*, not just parallelism on multicore?**

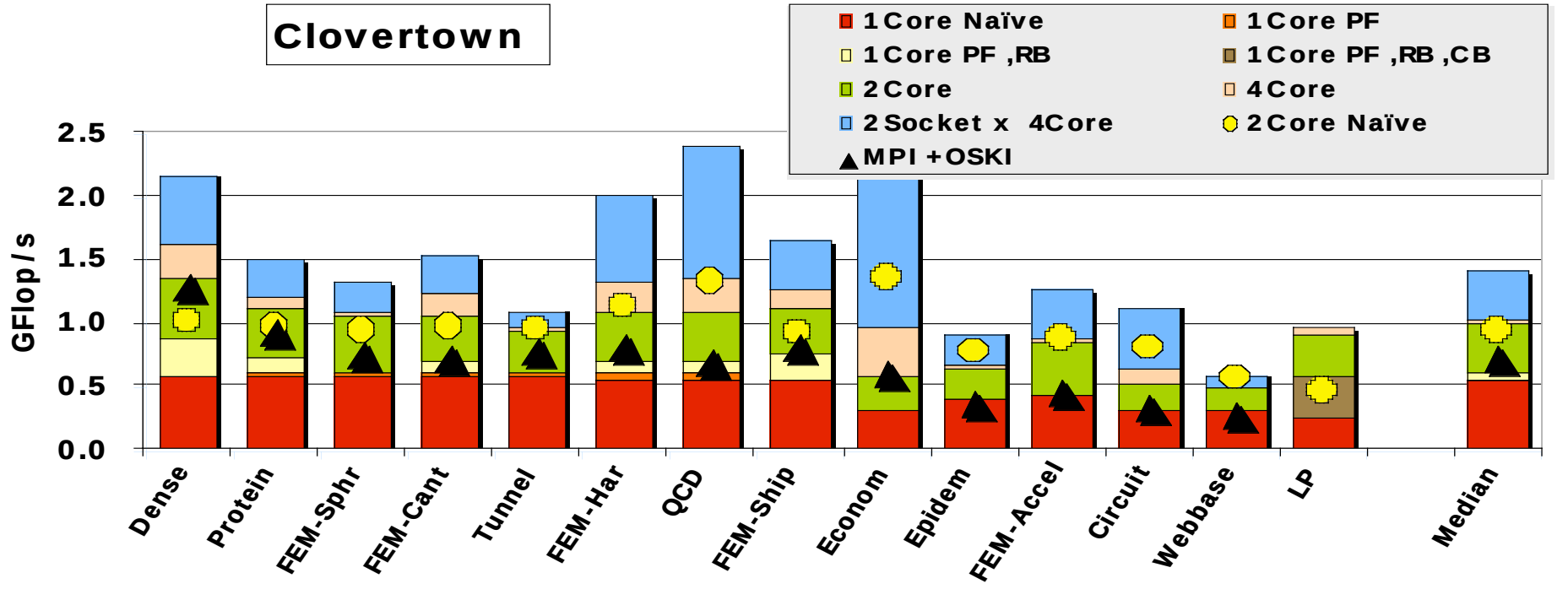
Tools for Efficiency: Autotuning

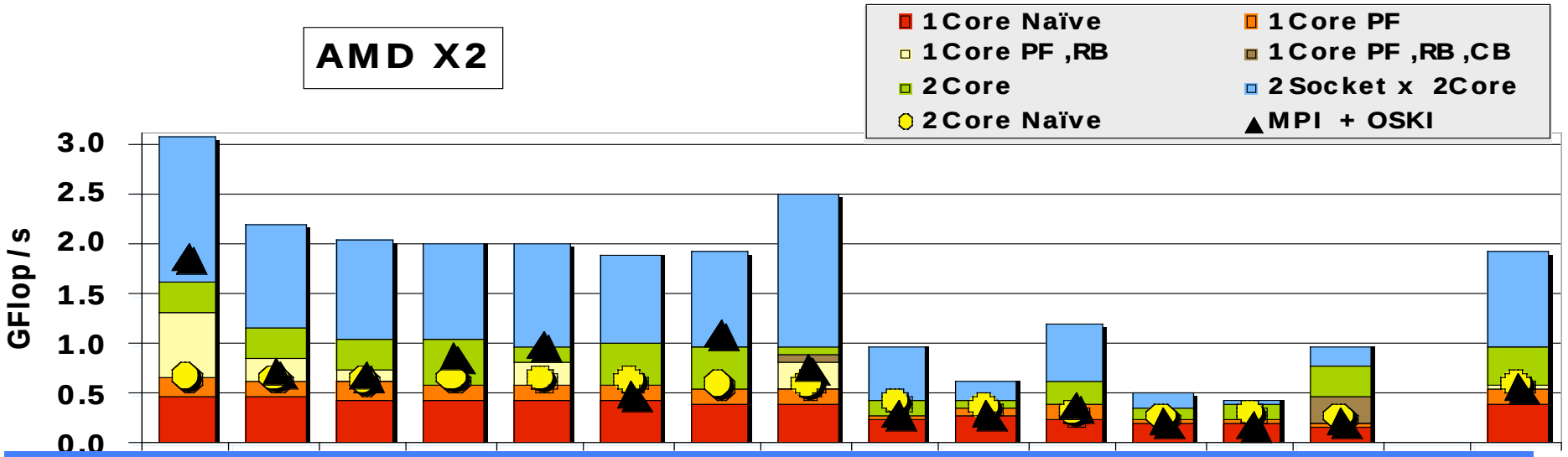
- **Automatic performance tuning**
  - Use machine time in place of human time for tuning
  - Search over possible implementations
  - Use performance models to restrict search space
  - Autotuned libraries for dwarfs (up to 10x speedup)
    - Spectral (FFTW, Spiral)
    - Dense (PHiPAC, Atlas)
    - Sparse (Sparsity, OSKI)
    - Stencils/structured grids
  - **Are these compilers?**
    - Don't transform source
    - There are compilers that use this kind of search
    - But not for the sparse case (transform matrix)



Optimization:
1.5x more entries (zeros)
→ 1.5x speedup

Compilers won't do this!





And don't think running MPI process per core is good enough for performance.

Rethink Compilers and Tools

Compilers

- **Challenges of writing optimized and portable code suggest compilers should:**
  - **Analyze and optimize parallel code**
    - **Eliminate races**
    - **Raise level of programming to data parallelism and other higher level abstractions**
    - **Enforce easy-to-learn memory models**
  - **Write code generators rather than programs**
    - **Compilers are more dynamic than traditional**
    - **Use partial evaluation and dynamic optimizations (used in both examples)**

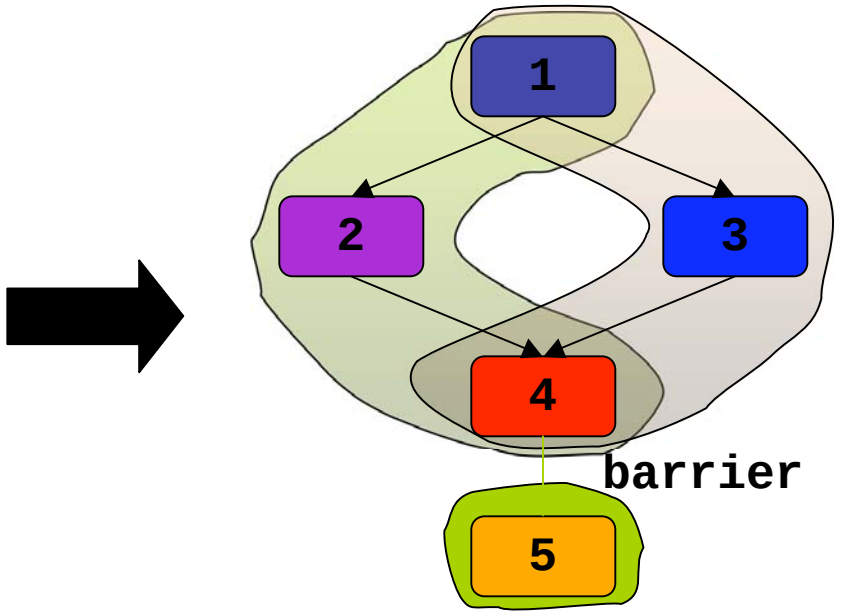
Parallel Program Analysis

- To perform optimizations, new analyses are needed for parallel languages
- In a data parallel or serial (auto-parallelized) language, the semantics are serial
  - Analysis is “easier” but more critical to performance
- Parallel semantics requires
  - Concurrency analysis: which code sequences may run concurrently
  - Parallel alias analysis: which accesses could conflict between threads
- Analysis is used to detect races, identify localizable pointers, and ensure memory consistency semantics (if desired)

Concurrency Analysis

- **Graph generated from program as follows:**
  - Node for each code segment between barriers and single conditionals
  - Edges added to represent control flow between segments
  - Barrier edges removed
- **Two accesses can run concurrently if:**
  - They are in the same node, or
  - One access's node is reachable from the other access's node

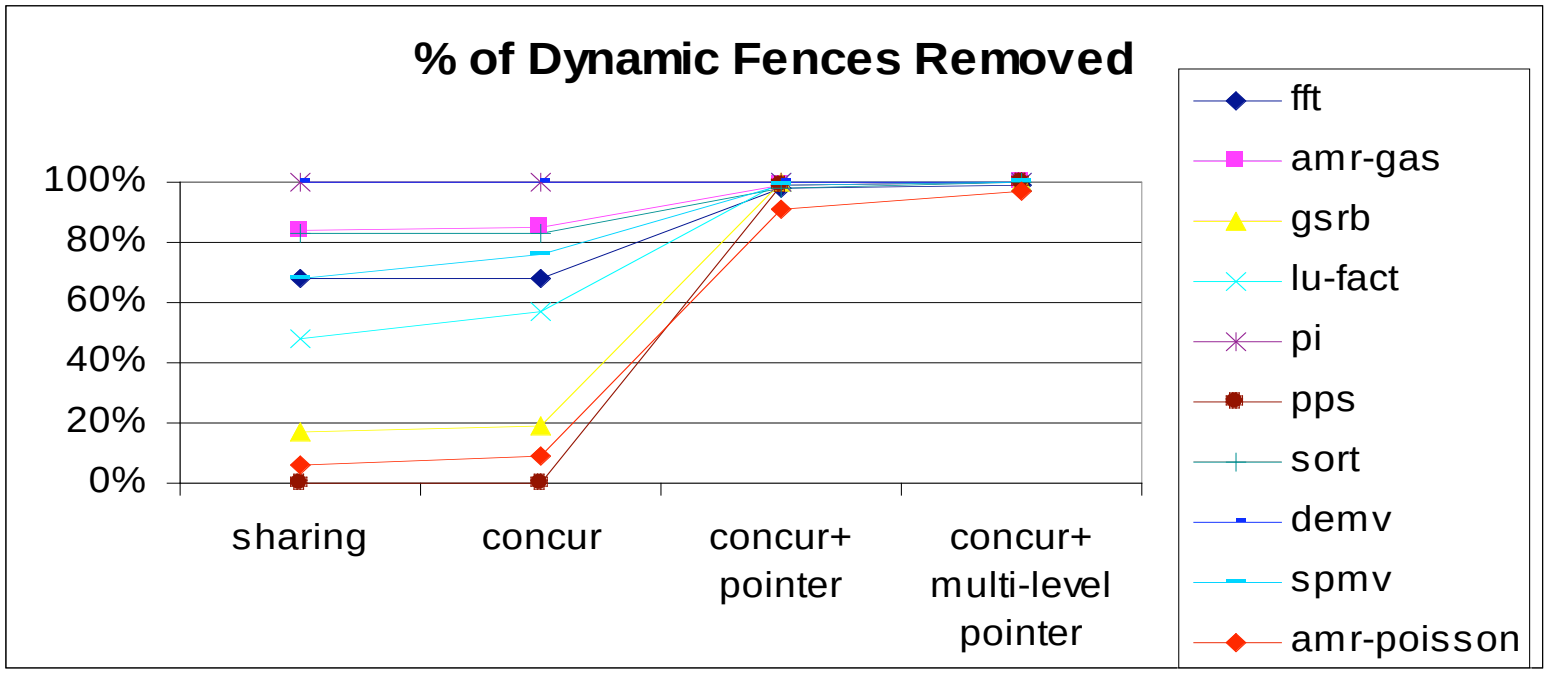
```
// segment 1
if ([single])
  // segment 2
else
  // segment 3
// segment 4
Ti.barrier()
// segment 5
```



Alias Analysis

- **Allocation sites correspond to *abstract locations (a-locs)***
  - **Abstract locations (a-locs) are typed**
- **All explicit and implicit program variables have *points-to sets***
  - **Each field of an object has a separate set**
  - **Arrays have a single points-to set for all elements**
- **Thread aware: Two kinds of abstract locations: local and remote**
  - **Local locations reside in local thread's memory**
  - **Remote locations reside on another thread**
  - **Generalizes to multiple levels (thread, node, cluster)**

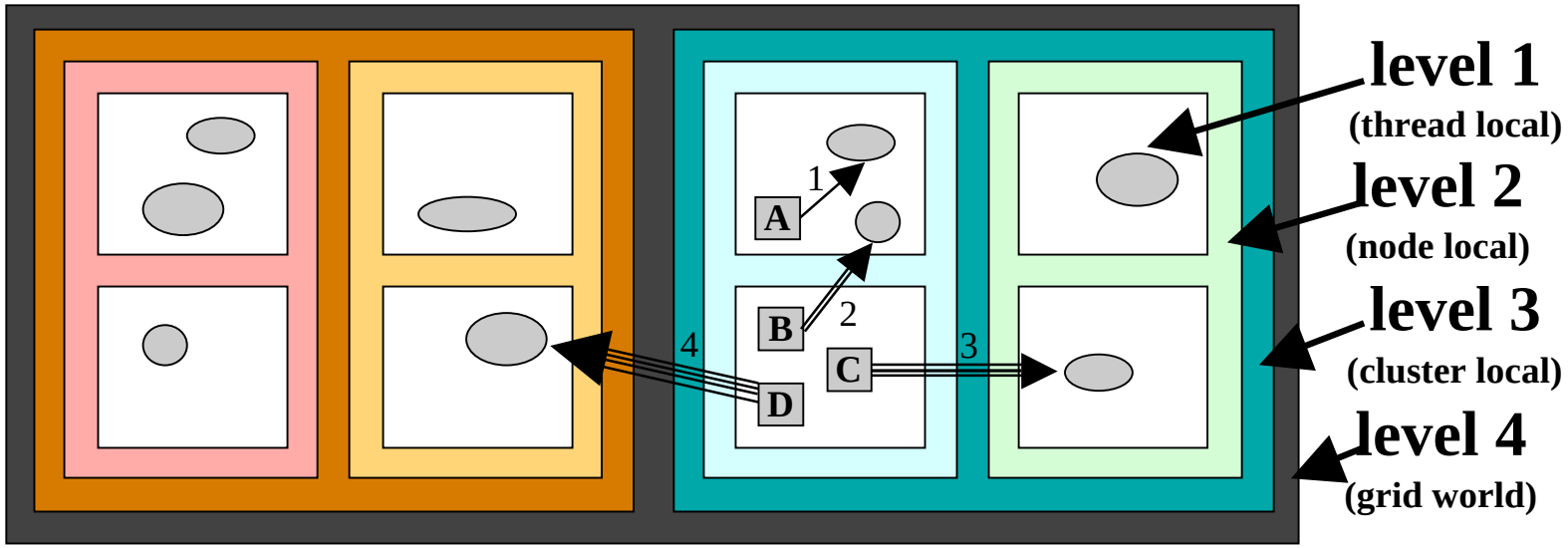
Implementing Sequential Consistency with Fences and Analysis



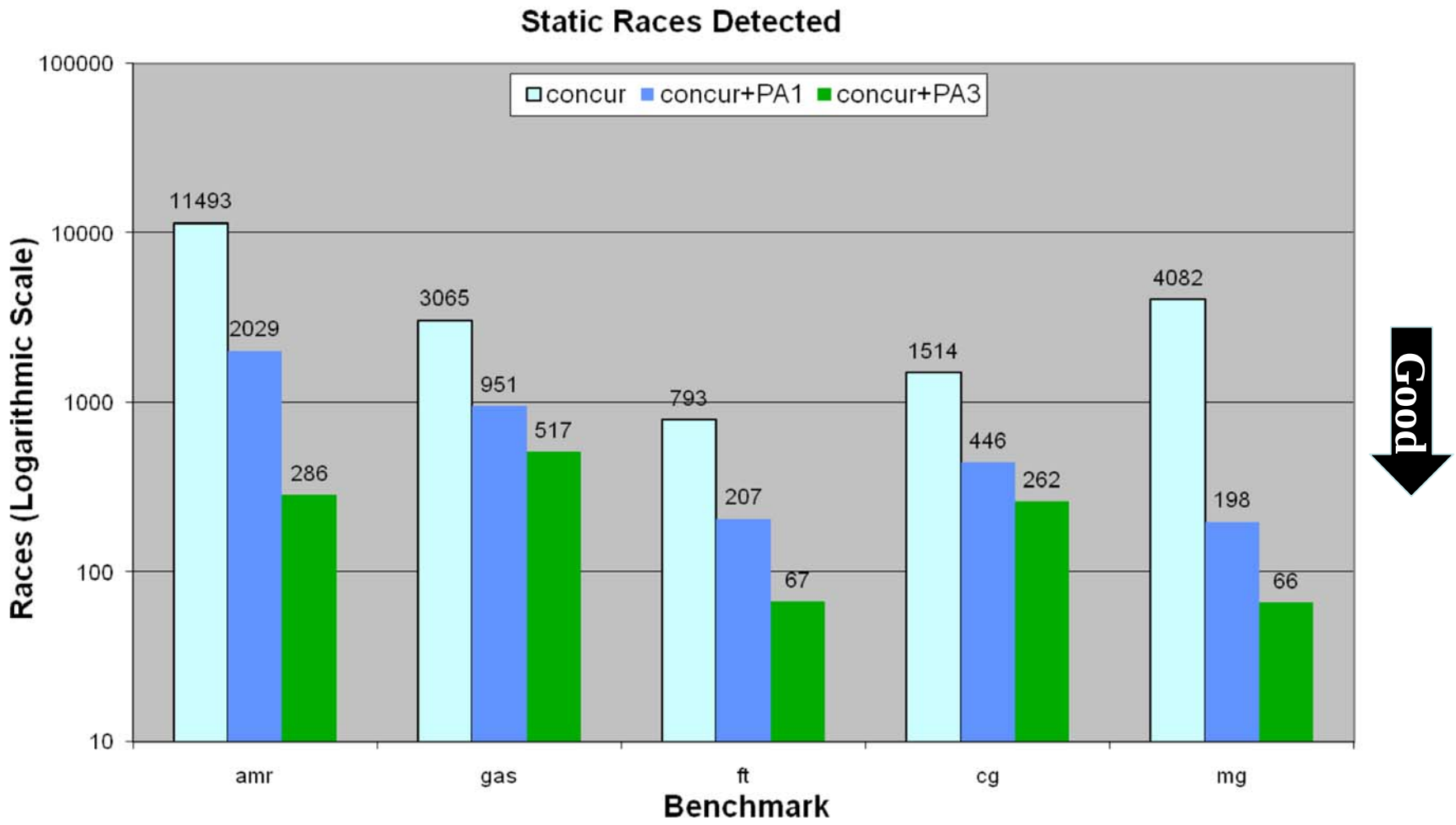
Joint work with Amir Kamil

Hierarchical Machines

- **Parallel machines often have hierarchical structure**



Race Detection Results

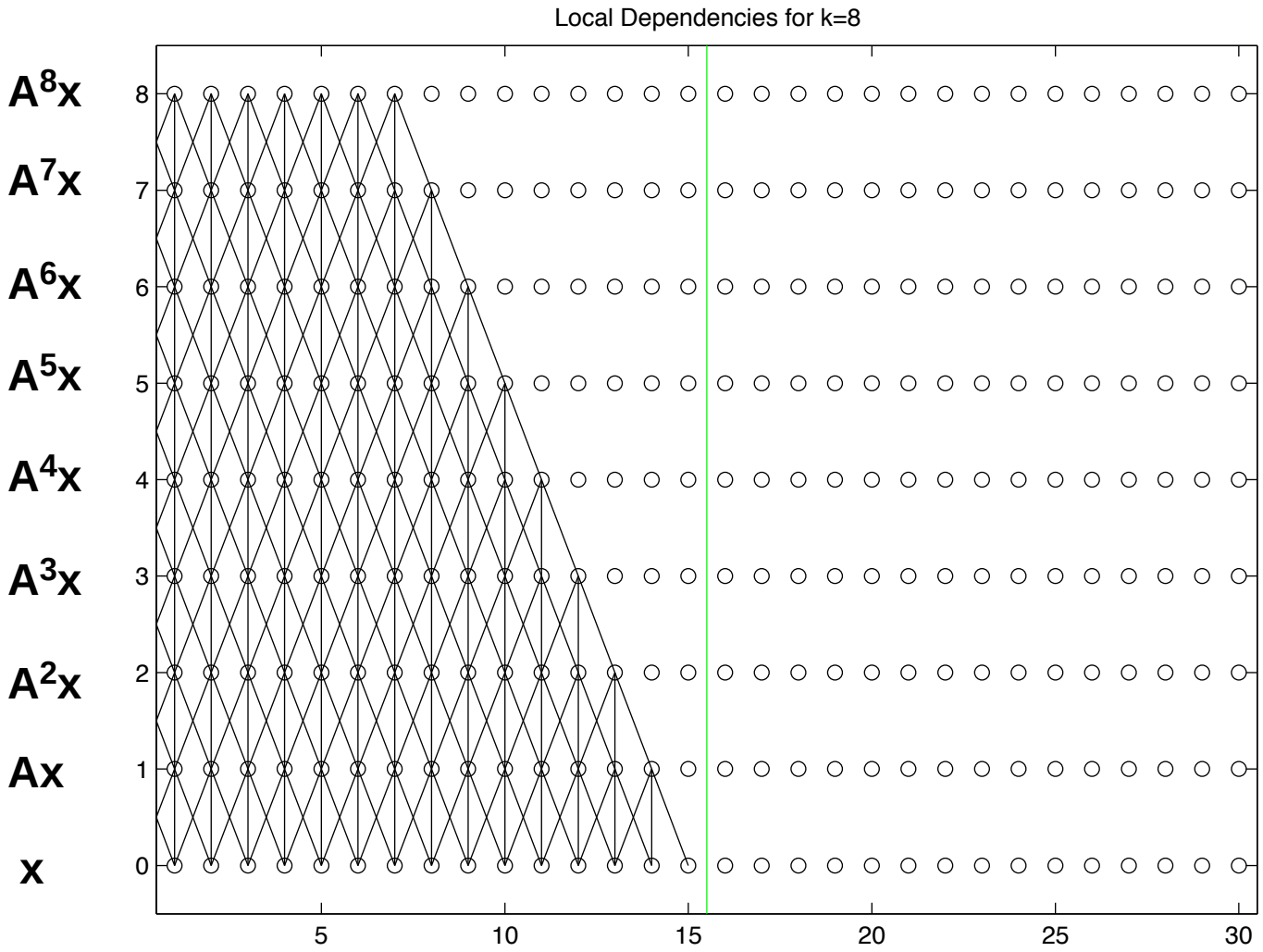


Rethink Algorithms

Latency and Bandwidth-Avoiding

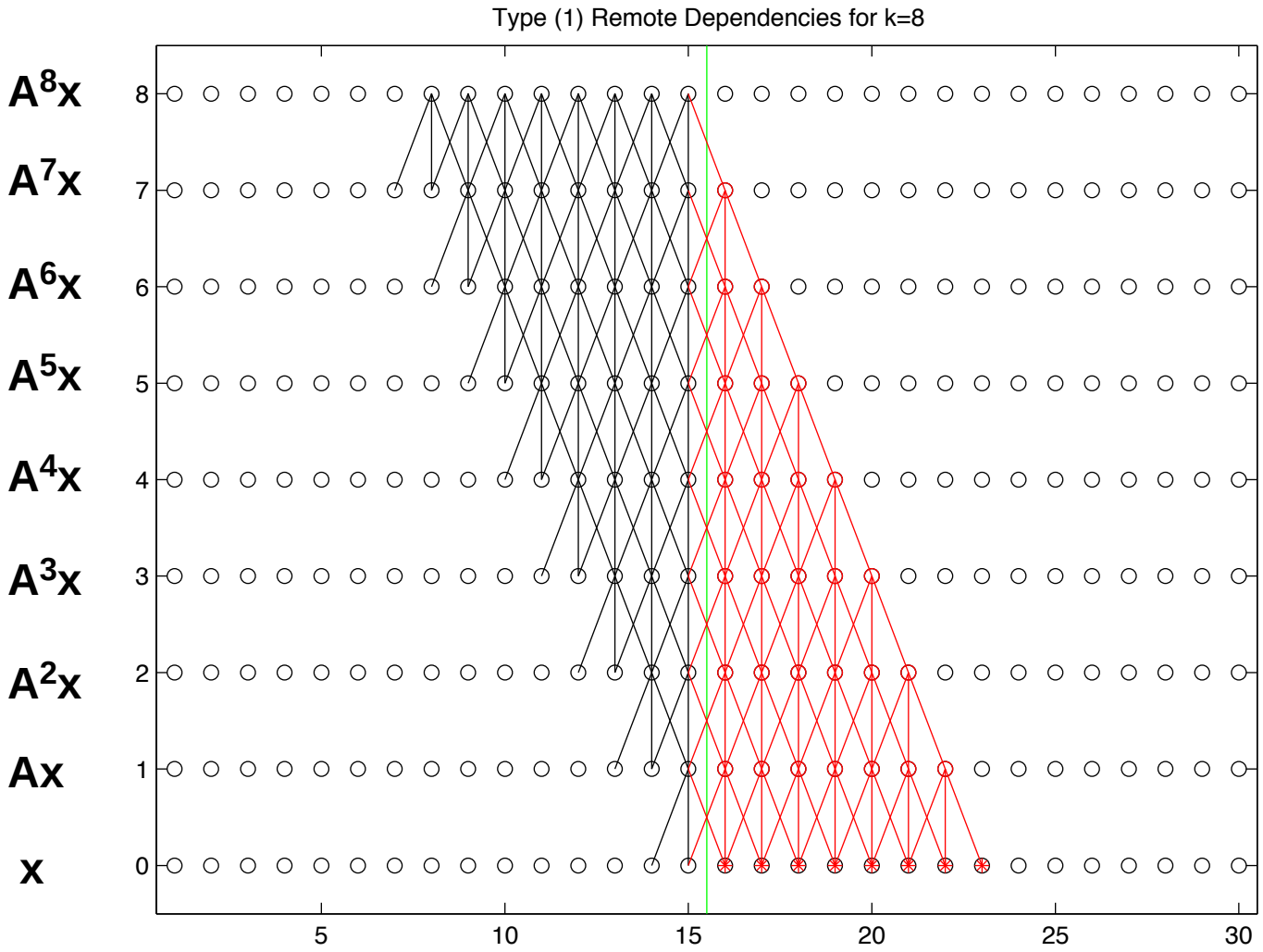
- Many iterative algorithms are limited by
  - Communication latency (frequent messages)
  - Memory bandwidth
- New optimal ways to implement Krylov subspace methods on parallel and sequential computers
  - Replace $x \rightarrow Ax$ by $x \rightarrow [Ax, A^2x, \ldots A^kx]$
  - Change GMRES, CG, Lanczos, … accordingly
- Theory
  - Minimizes network latency costs on parallel machine
  - Minimizes memory bandwidth and latency costs on sequential machine
- Performance models for 2D problem
  - Up to 7x (overlap) or 15x (no overlap) speedups on BG/P
- Measure speedup: 3.2x for out-of-core

Locally Dependent Entries for [x,Ax,…,A^8x], A tridiagonal



Can be computed without communication
k=8 fold reuse of A

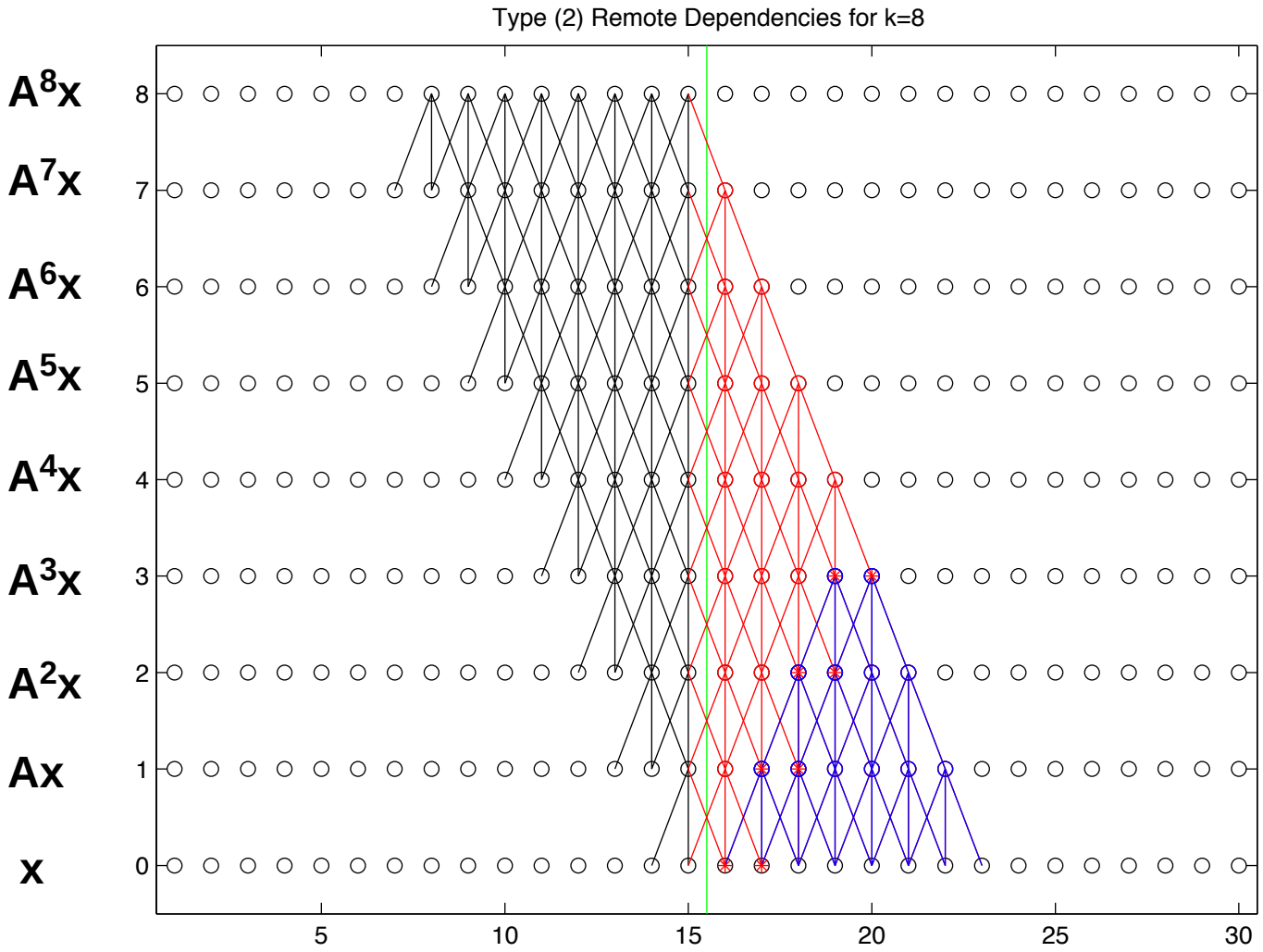
Remotely Dependent Entries for [x,Ax,…,A^8x], A tridiagonal



One message to get data needed to compute remotely dependent entries, not k=8

Price: redundant work

Fewer Remotely Dependent Entries for [x,Ax,…,A^{8}x], A tridiagonal

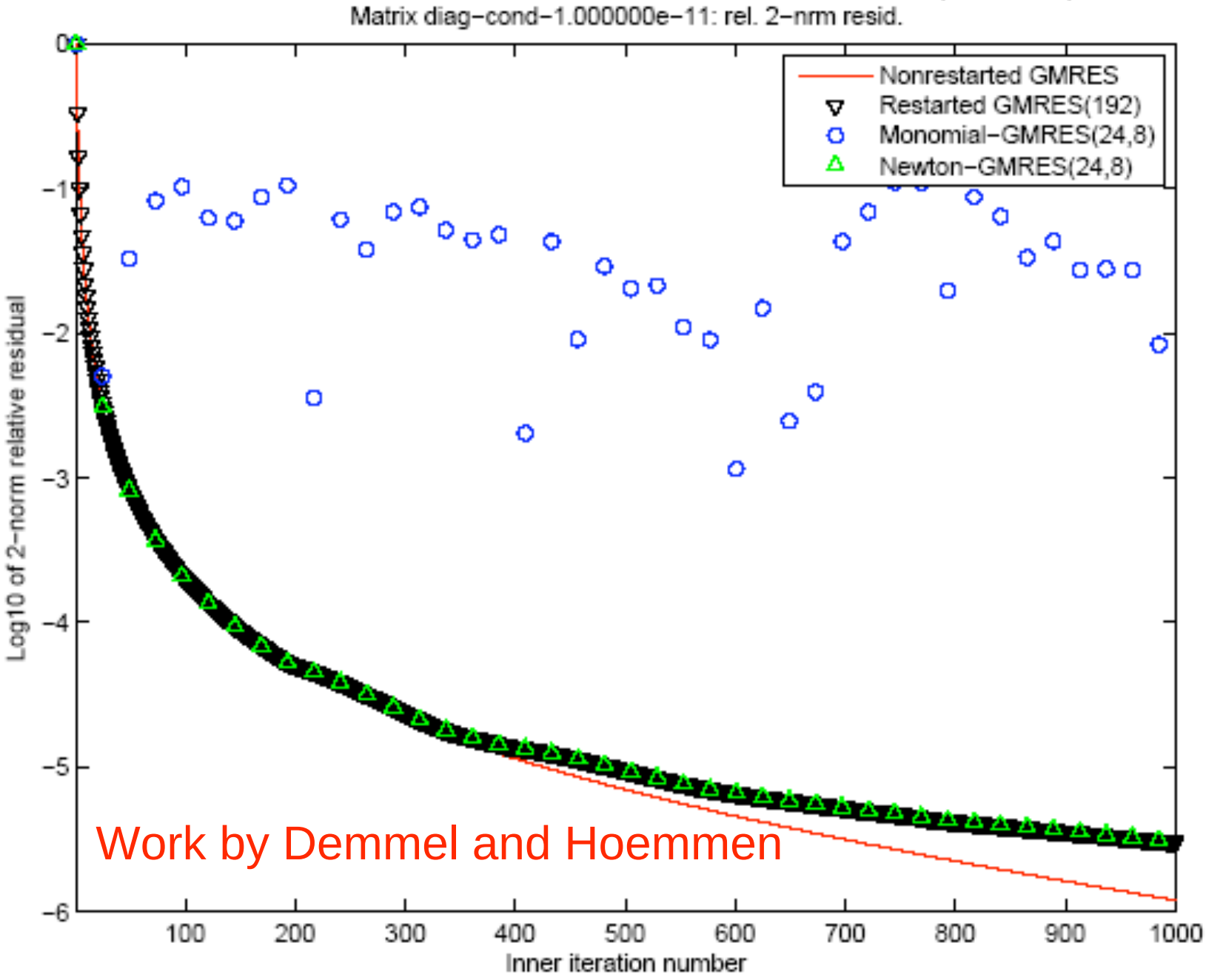


Reduce redundant work by half

Latency Avoiding Parallel Kernel for $[x, Ax, A^2x, \ldots, A^kx]$

- Compute locally dependent entries needed by neighbors
- Send data to neighbors, receive from neighbors
- Compute remaining locally dependent entries
- Wait for receive
- Compute remotely dependent entries

Can use Matrix Power Kernel, but change Algorithms



Work by Demmel and Hoemmen

Predictions and Conclusions

- **Parallelism will explode**
  - **Number of cores will double every ~2 years**
  - **Petaflop (million processor) machines will be common in HPC by 2015 (all top 500 machines will have this)**
- **Performance will become a software problem**
  - **Parallelism and locality are fundamental; can save power by pushing these to software**
- **Locality will continue to be important**
  - **On-chip to off-chip as well as node to node**
  - **Need to design algorithms for what counts (communication not computation)**
- **Massive parallelism required (including pipelining and overlap)**

Conclusions

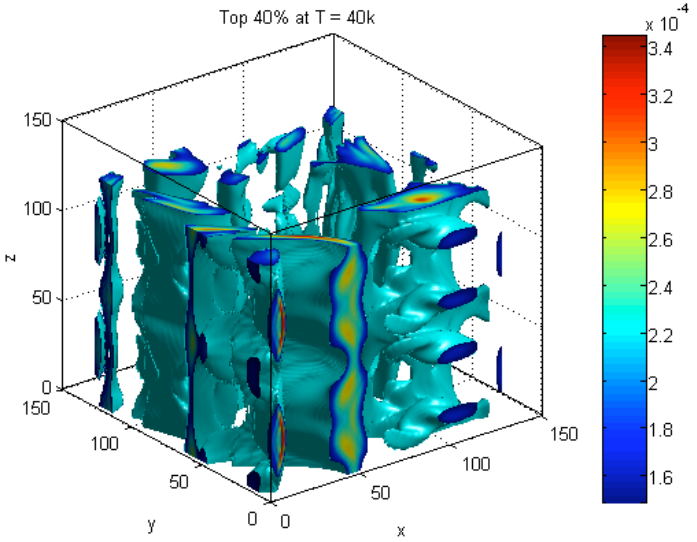
- **Parallel computing is the future**
- **Re-think Hardware**
  - **Hardware to make programming easier**
  - **Hardware to support good performance tuning**
- **Re-think Software**
  - **Software to make the most of hardware**
  - **Software to ease programming**
    - **Berkeley UPC compiler: http://upc.lbl.gov**
    - **Titanium compiler: http://titanium.cs.berkeley.edu**
- **Re-think Algorithms**
  - **Design for bottlenecks: latency and bandwidth**

Concurrency for Low Power

- **Highly concurrent systems are more power efficient**
  - ***Dynamic power is proportional to V^2fC***
  - ***Increasing frequency (f) also increases supply voltage (V): more than linear effect***
  - ***Increasing cores increases capacitance (C) but has only a linear effect***
- **Hidden concurrency burns power**
  - **Speculation, dynamic dependence checking, etc.**
- **Push parallelism discovery to software to save power**
  - **Compilers, library writers and applications programmers**

- **Challenge: *Can you double the concurrency in your algorithms every 2 years?***

LBMHD: Structure Grid Application



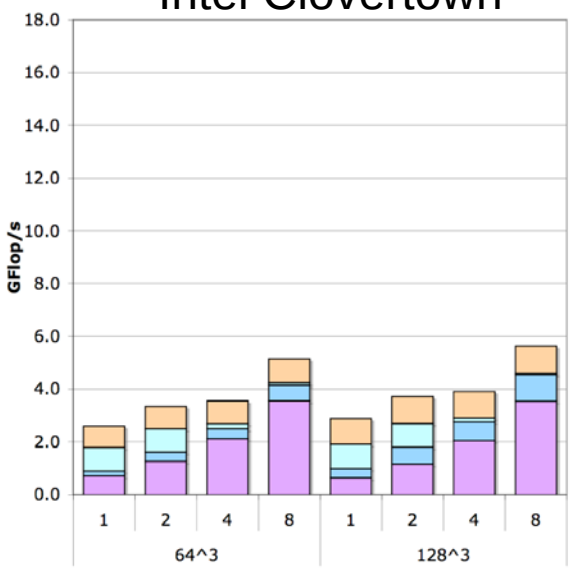
- Plasma turbulence simulation
- Two distributions:
  - momentum distribution (27 componen
  - magnetic distribution (15 vector compo
- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)
- Must read 73 doubles, and update(write) 79 doubles per point in space
- Requires about 1300 floating point operations per point in space
- Just over 1.0 flops/byte (ideal)
- No temporal locality between points in space within one time step

BERKELEY LAB

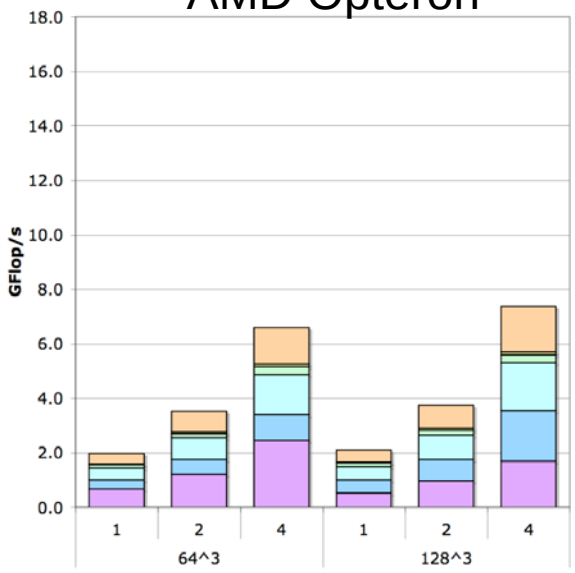
Autotuned Performance

(Cell/SPE version)

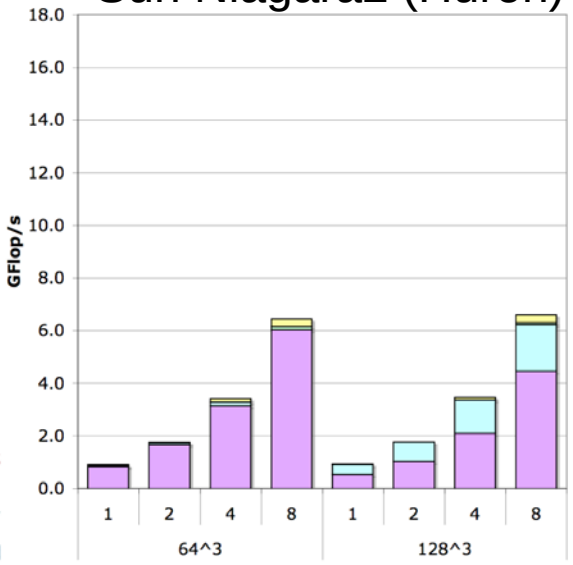
Intel Clovertown



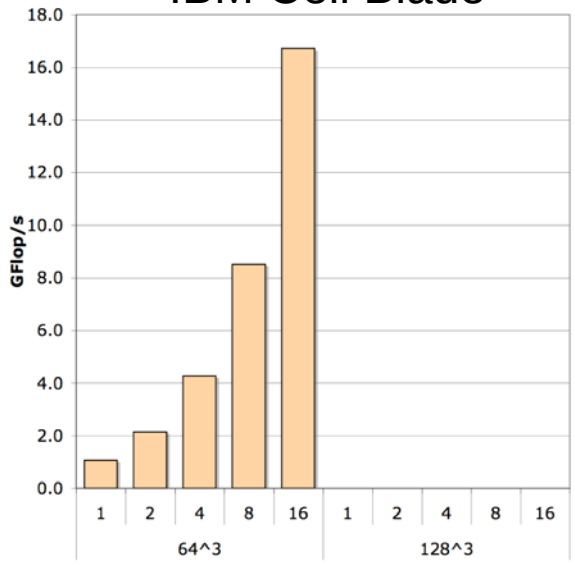
AMD Opteron



Sun Niagara2 (Huron)



IBM Cell Blade*



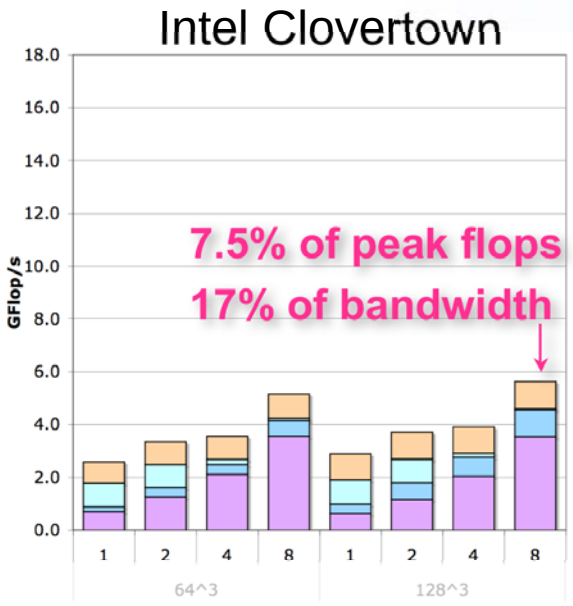
- **First attempt at cell implementation.**
- **VL, unrolling, reordering fixed**
- **Exploits DMA and double buffering to load vectors**
- **Straight to SIMD intrinsics.**
- **Despite the relative performance, Cell's DP implementation severely impairs performance**

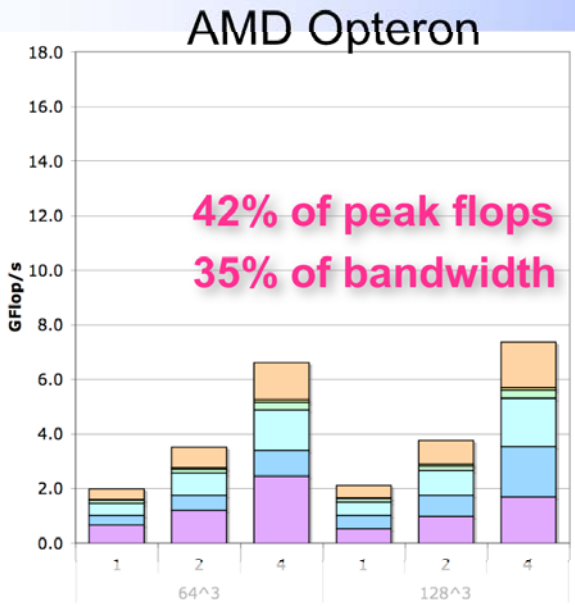
+SIMDization
+SW Prefetching
+Unrolling
+Vectorization
+Padding
Naïve+NUMA

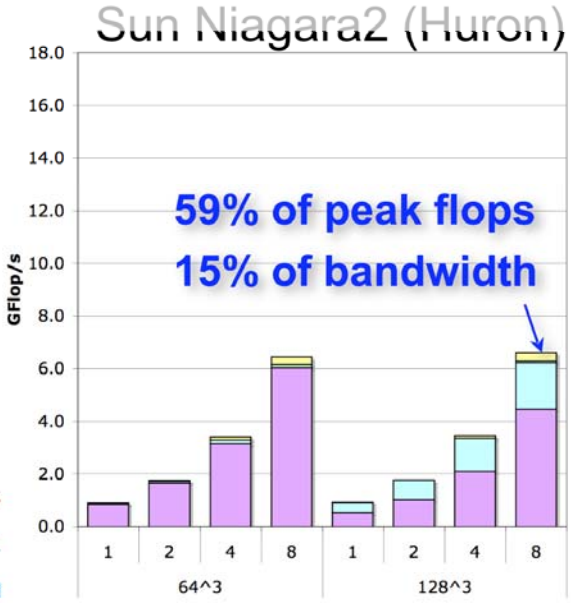
*collision() only

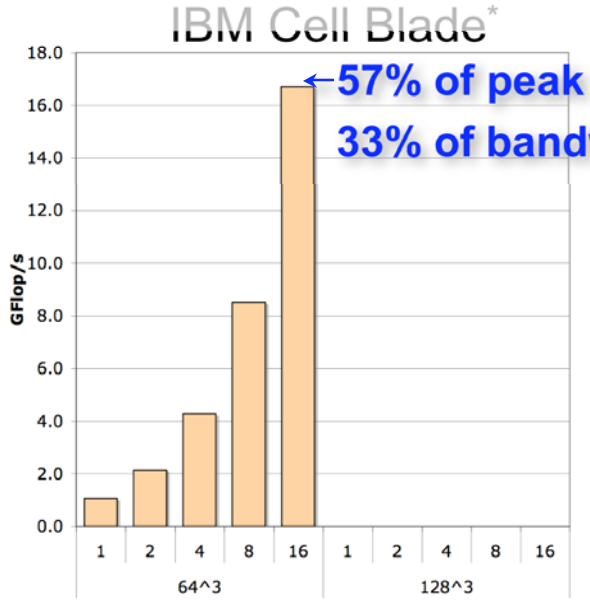
Autotuned Performance

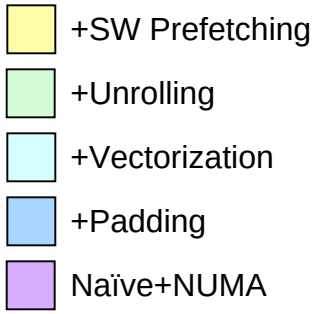
(Cell/SPE version)











*collision() only