Programming Models: Petascale and Beyond

Kathy Yelick

U.C. Berkeley and

Lawrence Berkeley National Laboratory

Software Issues at Scale

- **DARPA Study on Exascale**
  - Power discussion dominates all others; concurrency is the only significant approach:
    - lower clock, increases parallelism
  - Power density and system power (20-155MW)
- **Summary Issues for Software**
  - Exascale will require billion-way concurrency with 1K cores per chip
  - Departmental scale (1 PF) systems will require millions of threads
  - The memory/core ratio will drop by at least an order of magnitude across machine size
    - Note: Weak Scaling at Risk!
  - A new model for fault tolerant software is needed; checkpoints to disk will be impractical
- **These issues will creep into Petascale**

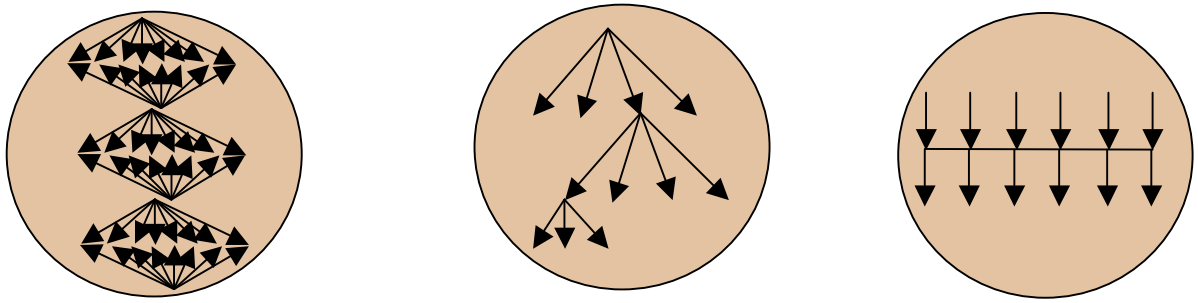
Need a Fundamentally New Approach

- **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

Rethinking Programming Models

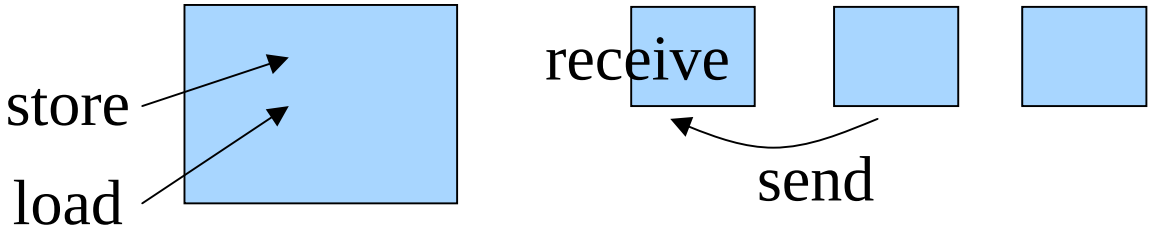
Two Parallel Language Questions

- **What is the parallel control model?**



data parallel (singe thread of control) | dynamic threads | single program multiple data (SPMD)

- **What is the model for sharing/communication?**



shared memory | message passing

implied synchronization for message passing, not shared memory

A Brief History of Languages

- **When vector machines were king**
  - Parallel “languages” were loop annotations (IVDEP)
  - Performance was fragile, but there was good user support
- **When SIMD machines were king**
  - Data parallel languages popular and successful (CMF, *Lisp, C*, …)
  - Quite powerful: can handle irregular data (sparse mat-vec multiply)
  - Irregular computation is less clear (multi-physics, adaptive meshes, backtracking search, sparse matrix factorization)
- **When shared memory machines (SMPs) were king**
  - Shared memory models, e.g., OpenMP, Posix Threads, are popular
- **When clusters took over**
  - Message Passing (MPI) became dominant

We are at the mercy of HW, but SW takes the blame.

To Virtualize or Not

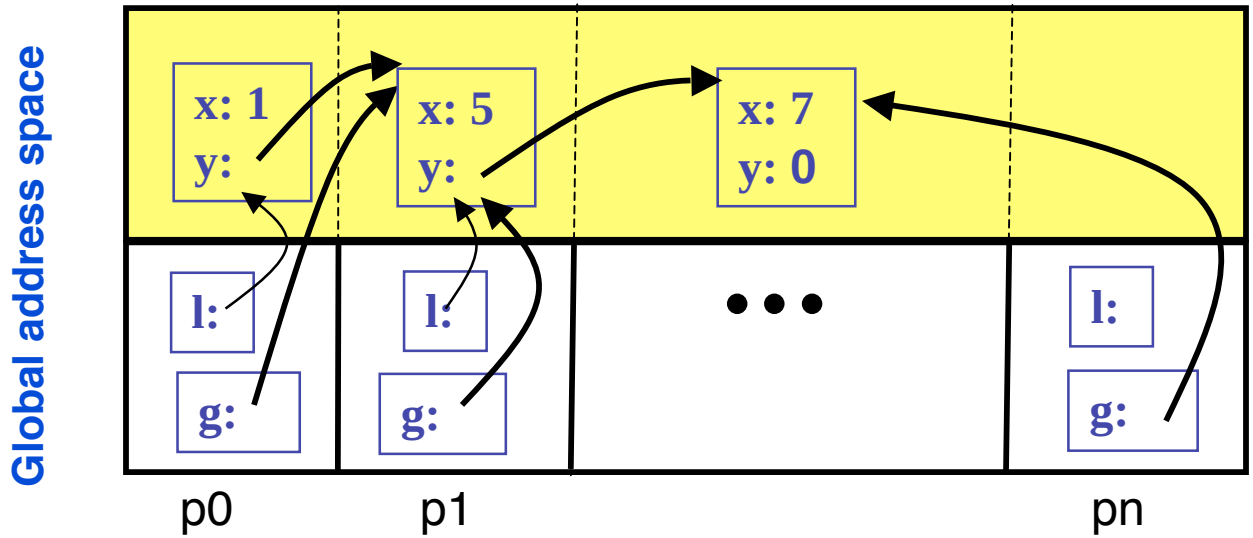
- The fundamental question facing in parallel programming models is:

  What should be virtualized?

- Hardware has finite resources
  - Processor count is finite
  - Registers count is finite
  - Fast local memory (cache and DRAM) size is finite
  - Links in network topology are generally $< n^2$
- Does the programming model (language+libraries) expose this or hide it?
  - E.g., one thread per core, or many?
    - Many threads may have advantages for load balancing, fault tolerance and latency-hiding
    - But one thread is better for deep memory hierarchies
- How to get the most out of your machine?

PGAS Languages

- *Global address space:* **thread may directly read/write remote data**
  - **"Virtualizes" or hides the distinction between shared/distributed memory**
- *Partitioned:* **data is designated as local or global**
  - **Does not hide this: critical for locality and scaling**

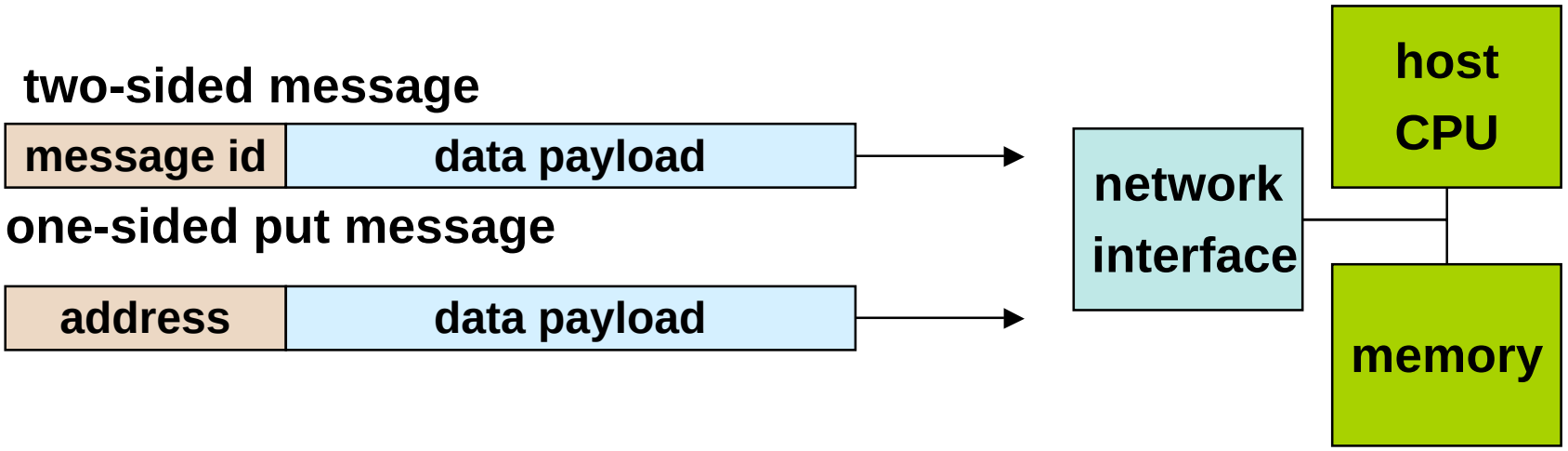


- *UPC, CAF, Titanium: Static parallelism (1 thread per proc)*
  - **Does not virtualize processors; main difference from HPCS languages which have many/dynamic threads**

What Makes a Language/Library PGAS?

- **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: memory copy or array copy
  - Optional: remote invocation of functions
- **Control over data layout**
  - PGAS is not the same as (cache-coherent) “shared memory”
  - Remote data stays remote in the performance model
- **Synchronization**
  - Global barriers, locks, memory fences
- **Collective Communication, IO libraries, etc.**

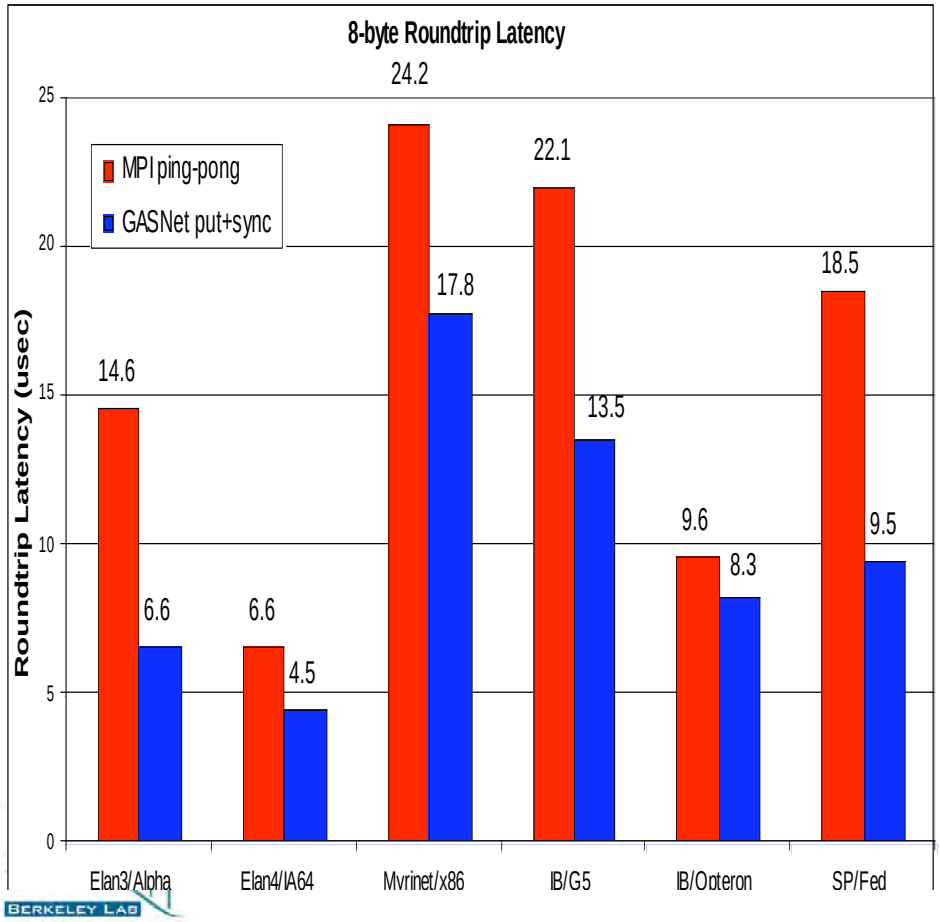
What Make Communication One-Sided?

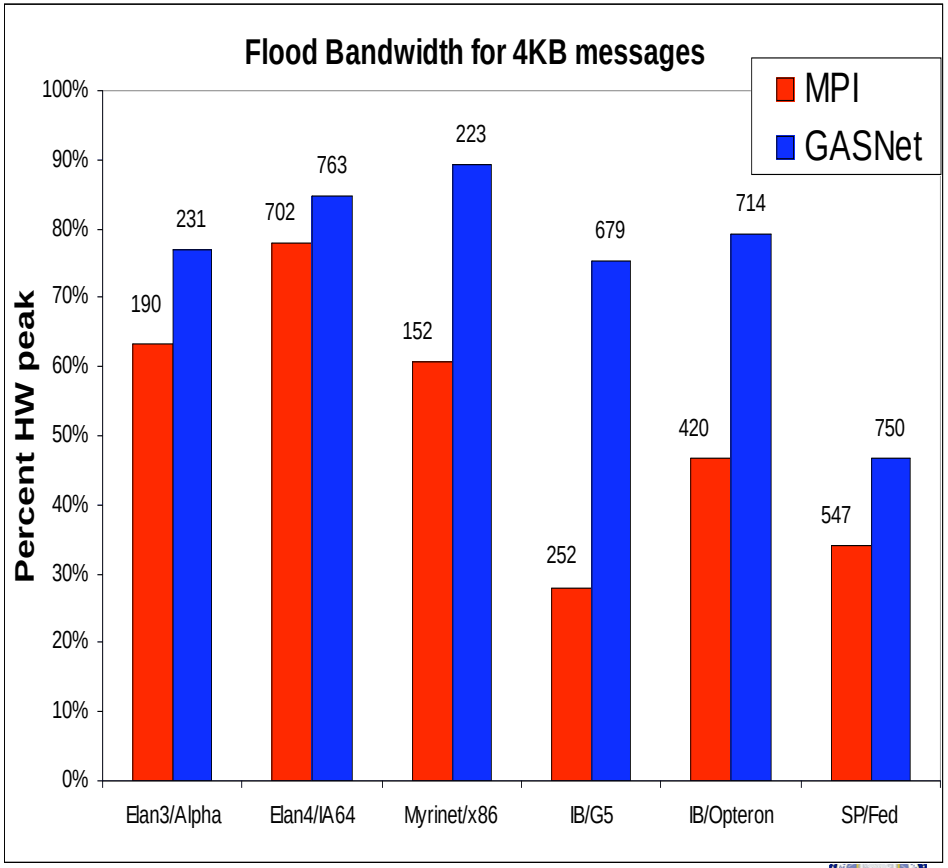


- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks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

Performance Advantage of One-Sided Communication

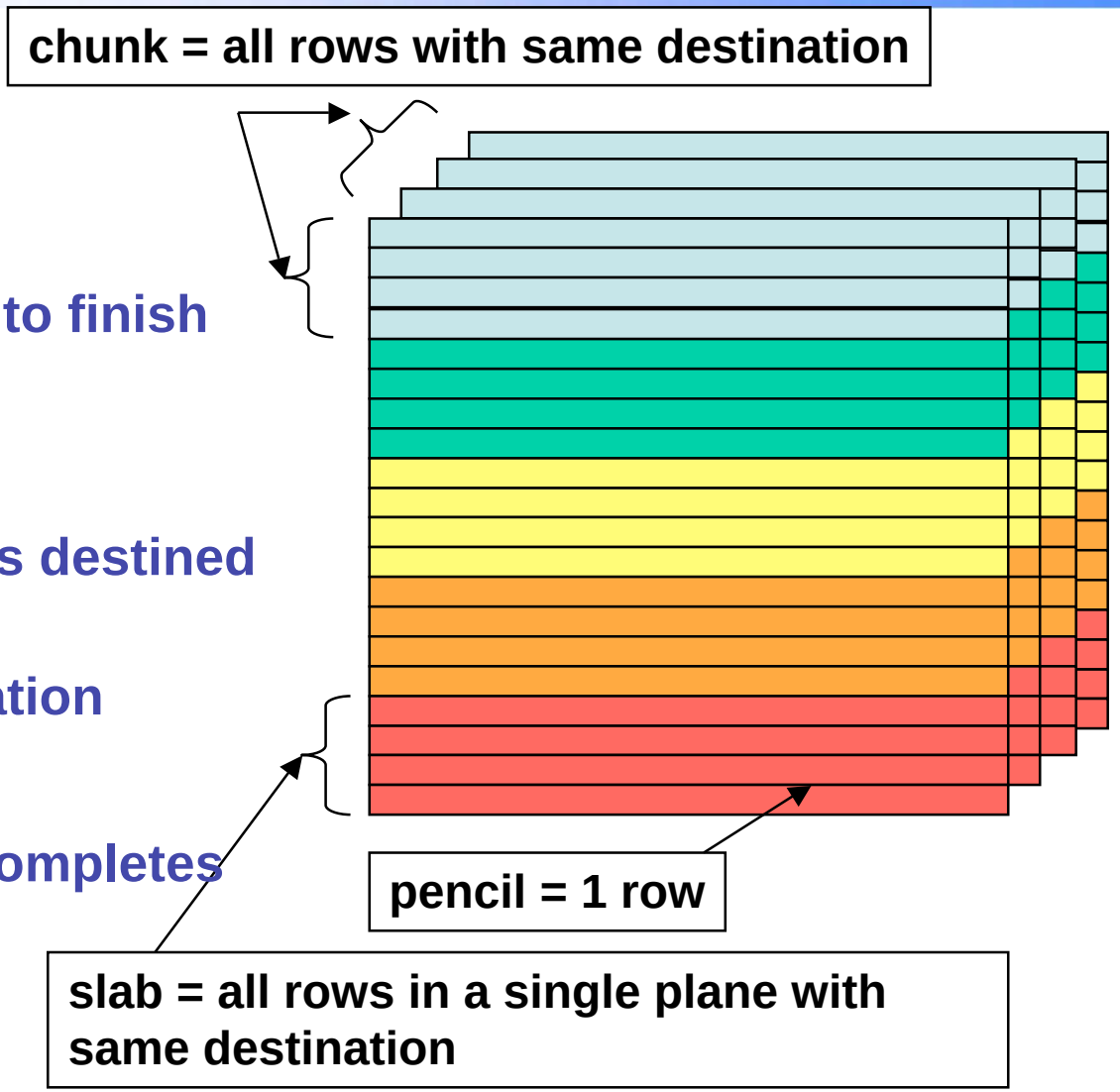
- The put/get operations in PGAS languages (remote read/write) are one-sided (no required interaction from remote proc)
- This is faster for pure data transfers than two-sided send/receive





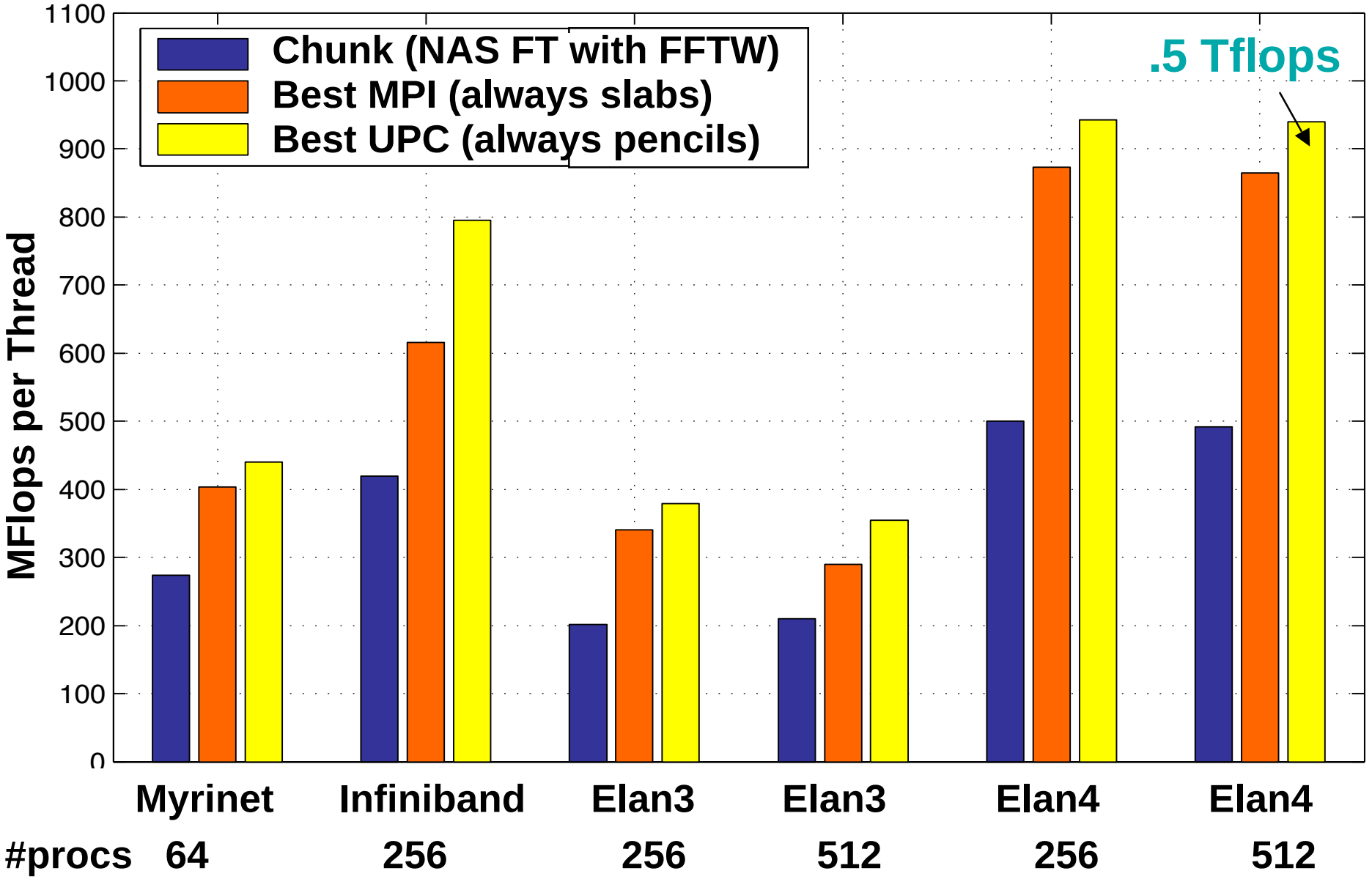
Communication Strategies for 3D FFT

- **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



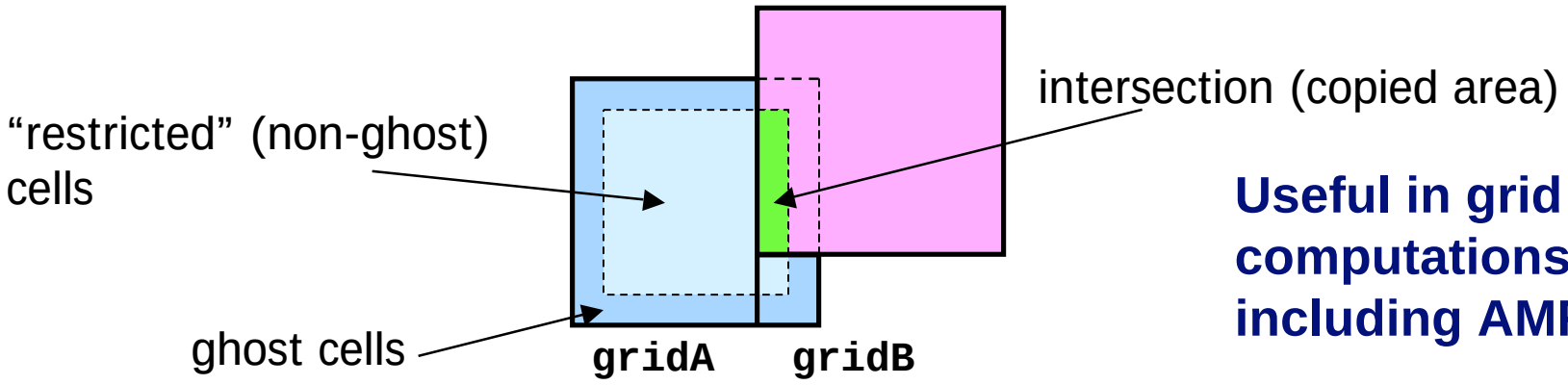
Arrays in a Global Address Space

- **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

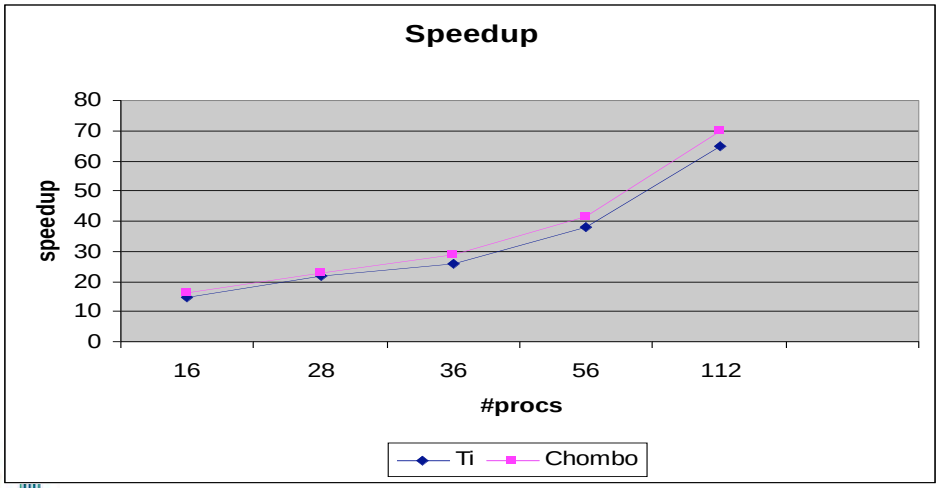
Languages Support Helps Productivity

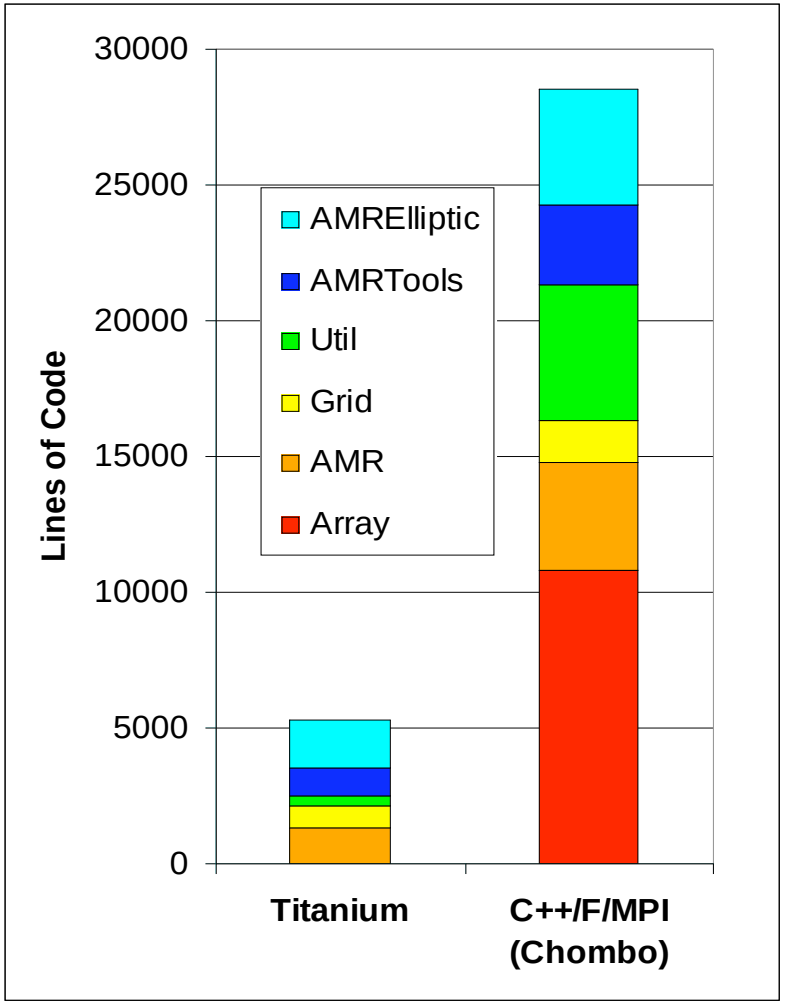
C++/Fortran/MPI AMR

- 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



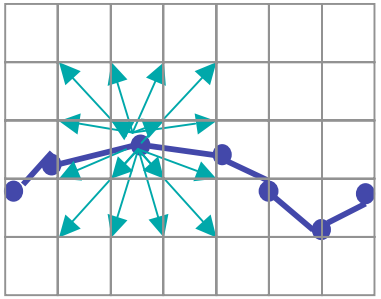


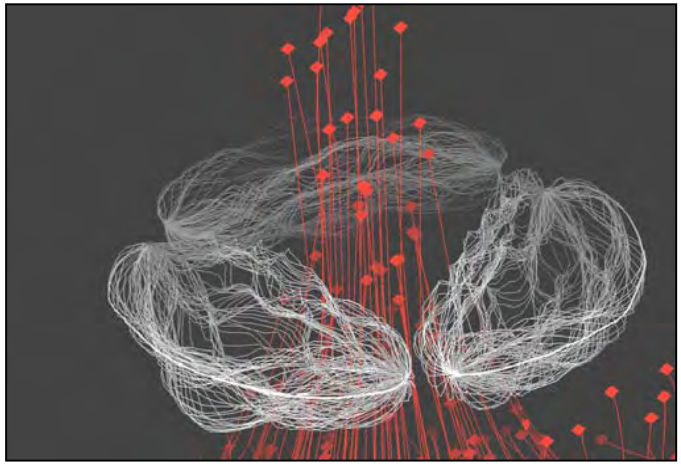
Work by Tong Wen and Philip Colella; Communication optimizations joint with Jimmy Su

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

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen

Beyond the SPMD Model: Dynamic Threads

- **UPC uses a static threads (SPMD) programming model**
  - **No dynamic load balancing built-in**
- **Berkeley compiler has some extensions**
  - **Allows programmers to execute active messages (AMs)**
  - **AMs have limited functionality (no messages except acks) to avoid deadlock in the network**
- **A more dynamic runtime would have many other uses**
  - **Application load imbalance, OS noise, fault tolerance**
- **Two extremes are well-studied**
  - **Dynamic parallelism without locality**
  - **Static parallelism (with threads = processors) with locality**
- **What issues do we run into if we want dynamic threads with locality?**

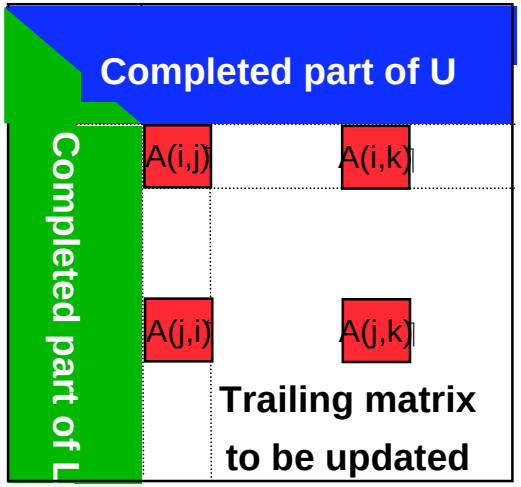
LU Factorization with Partial Pivoting

- **Interesting, heavily used computational kernel**
  - **Non-trivial dependence patterns**
- **Available in Linpack/LAPACK/ScaLAPACK**
  - **LAPACK/ScaLAPACK are the second most popular mathematical libraries by NERSC users**
- **HPL benchmark**
  - **Highly tuned parallel block LU factorization with partial pivoting**
  - **Best optimized code around, written with MPI**
- **Rules for extreme performance**
  - **Locality and load balance (traditional blocked-cyclic layout)**
  - **Maximize cache re-use; use blocks (merge blocks when possible)**
  - **Avoid synchronization in the network: one-sided communication**
  - **Avoid waiting on algorithmic synchronization and data transfers using multithreading**

Joint work with Parry Husbands

LU Factorization with Partial Pivoting

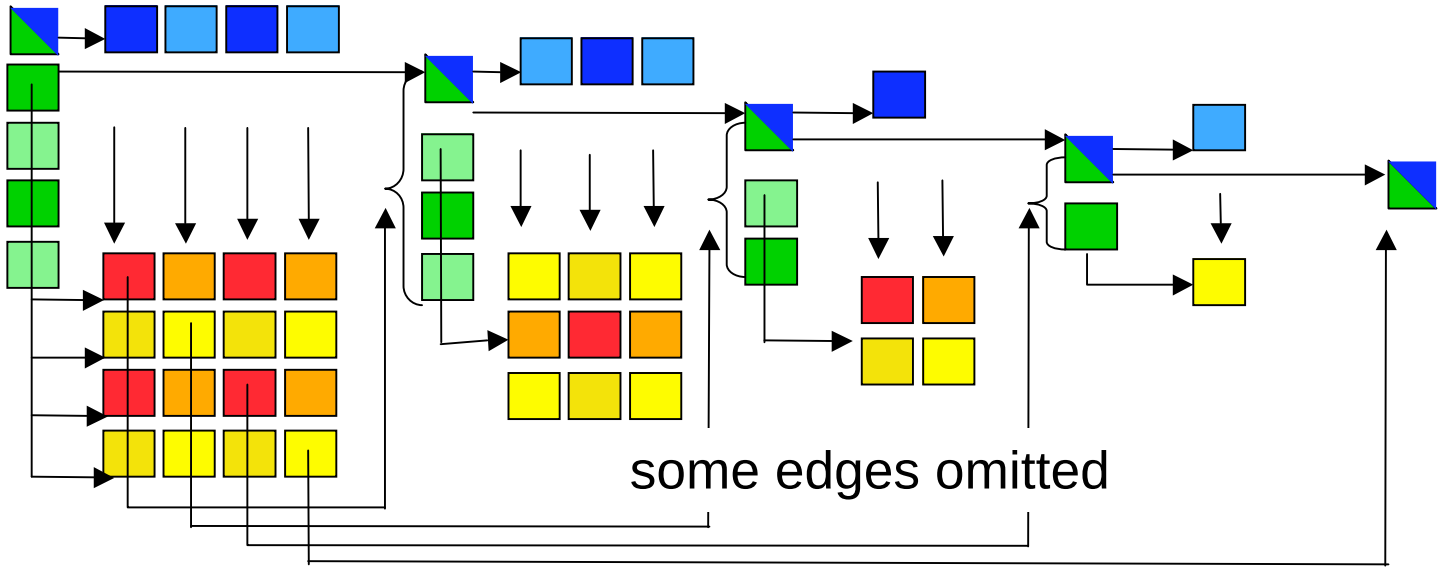
```
for i=1:n-1
  swap rows so |a(i,i)| = max{abs(a(:,i))}   // pivot
  for j=i+1:n
    l(j,i) = a(j,i)/a(i,i)                   // scale to get l
  for j=i:n
     u(i,j)=a(i,j)                           // update u
  for j=i+1:n
    for k=i+1:n                              // trailing matrix
      a(j,k) = a(j,k)-l(j,i)*u(i,k)          // update
```



Code is blocked in practice, so that updates are matmuls

Joint work with Parry Husbands

Parallel Tasks in LU

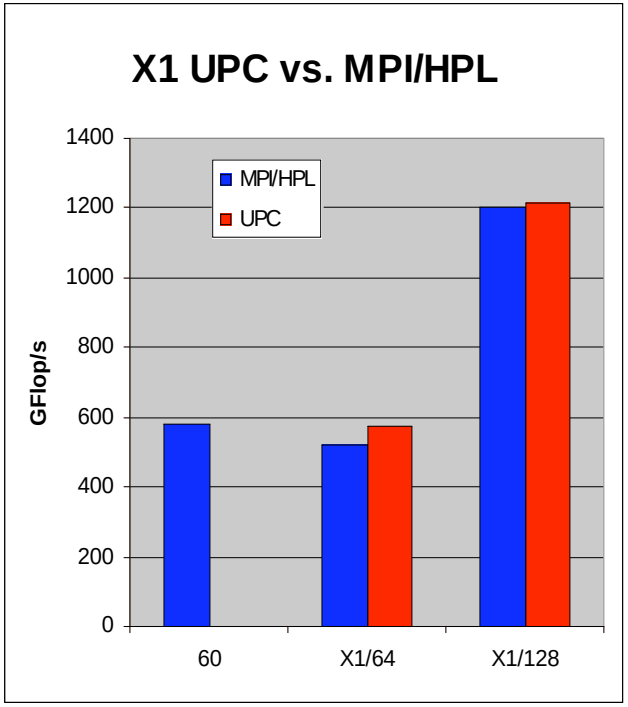


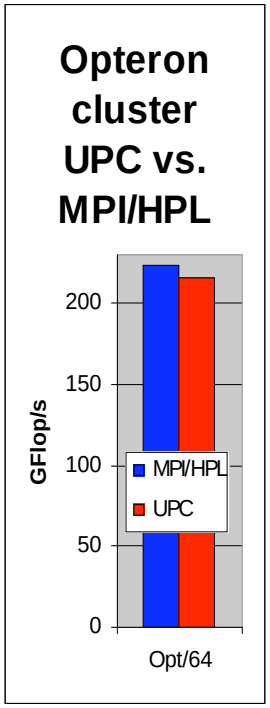
- **Panel Factorizations (parallel recursive formulation used)**
- **Pivot application and update to U**
- **Trailing matrix updates**

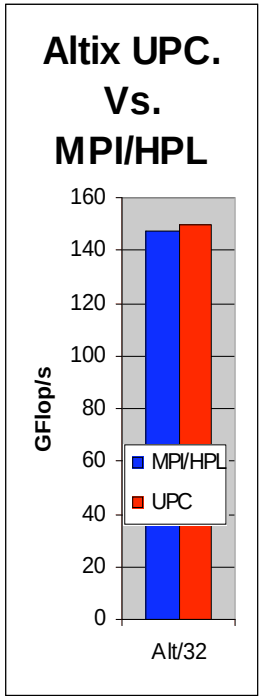
LU in UPC + Multi-threading

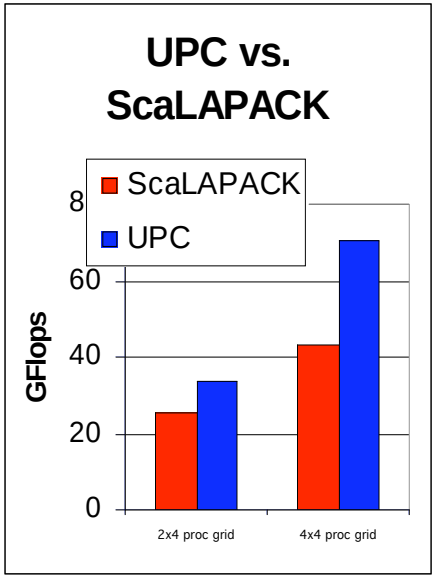
- UPC uses a static threads (SPMD) programming model
- Co-operative multi-threading used to mask latency and to mask dependence delays (home-grown package)
  - Thread volunteers to give up processor; never pre-empted
    - Important for deep memory hierarchies
  - Non-blocking (get) transfers to mask communication latency
  - Remote enqueue used to spawn remote threads.
  - Matrix blocks distributed in 2-d block-cyclic manner (fixed layout) and tuned for block size.
- Three levels of threads in LU code:
  - UPC threads (data layout, each runs an event scheduling loop)
  - Multithreaded BLAS (boost efficiency)
  - User level (non-preemptive) threads with explicit yield
- Operations "fire" when dependencies are satisfied.
  - Carefully tuned application-specific scheduler use to prioritize critical path and avoid memory allocator deadlock

UPC HP Linpack Performance









- **Faster than ScaLAPACK (less synchronization), comparable to MPI/HPL**
- **Large scaling of UPC code on Itanium/Quadrics (Thunder)**
  - 2.2 TFlops on 512p and 4.4 TFlops on 1024p

  91.8% of peak on 1p Itanium 2 1.5GHz, 81.9% on 1p Opteron 2.2GHz

Lessons Learned

- One-sided communication is faster than 2-sided
  - FFT example shown previous
- Global address space can ease programming
  - Dynamic scheduling can tolerate latencies
  - More adaptive and easier to use (fewer knobs)
- Principled scheduling that takes into account
  - Critical Path, Memory use, Cache, etc.
- Combination of dynamic loc balancing with locality control has new challenges
  - Previous work solve load balancing (Cilk) or locality (MPI) but not both together
- Current PGAS languages are not the final answer

Rethinking Software

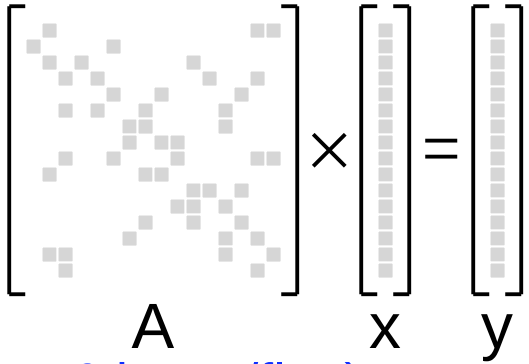
Performance from Multicore:
How hard can it be?

Autotuning: Extreme Performance Programming

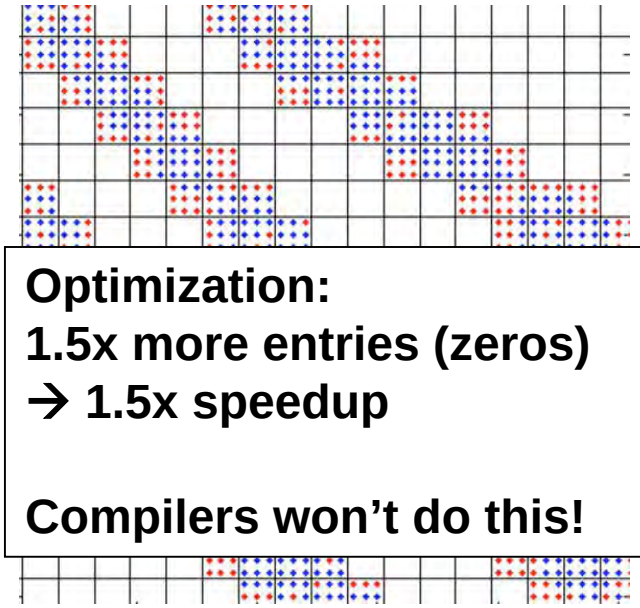
- **Automatic performance tuning**
  - **Use machine time in place of human time for tuning**
  - **Search over possible implementations**
  - **Use performance models to restrict search space**
- **Programmers should write programs to generate code, not the code itself**
- **Autotuning finds a good performance solution be heuristics or exhaustive search**
  - **Perl script generates many versions**
  - **Generate SIMD-optimized kernels**
  - **Autotuner anlyzes/runs kernels**
  - **Uses search and heuristics**
- **PERI SciDAC is including some of these ideas into compilers and domain-specific code generators libraries, e.g., OSKI for sparse matrices**

Sparse Matrix Vector Multiplication

- **Sparse matrix-vector multiply**
  - **Important to applications**



- **Challenges**
  - Very low computational intensity (often >6 bytes/flop)
    = likely memory bound
  - **Difficult to exploit Superscalar**
  - **Difficult to exploit SIMD**
  - **Irregular memory access**
  - **Difficult to load balance**
- **Optimizations depend on matrix**

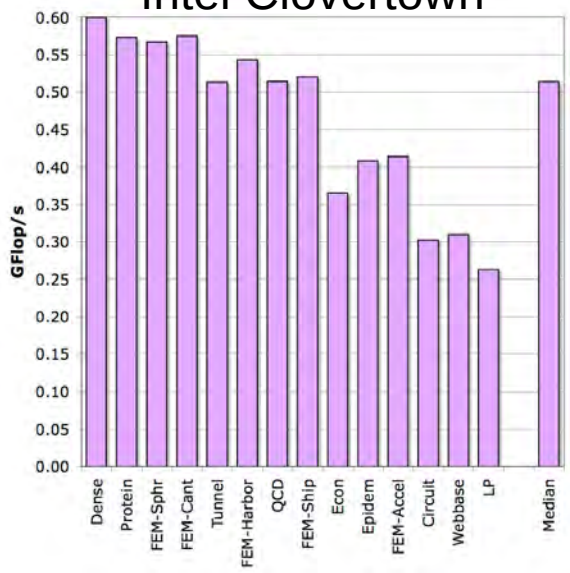


Optimization:
1.5x more entries (zeros)
→ 1.5x speedup

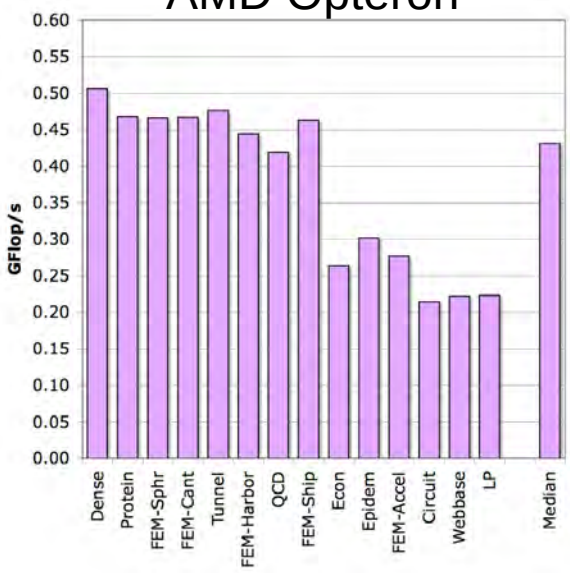
Compilers won't do this!

Naïve Serial Implementation

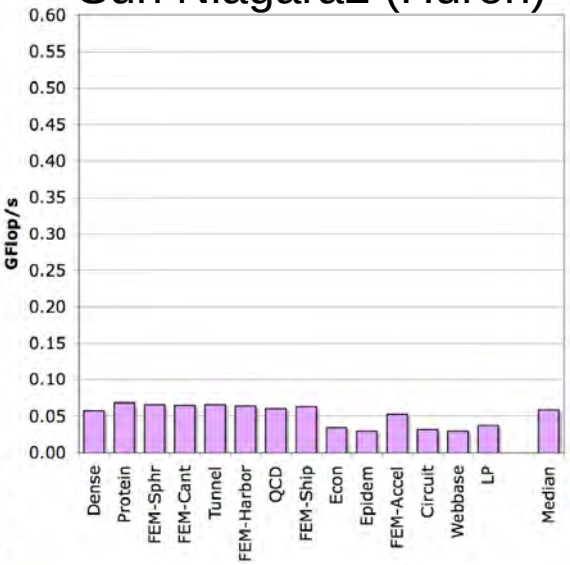
Intel Clovertown



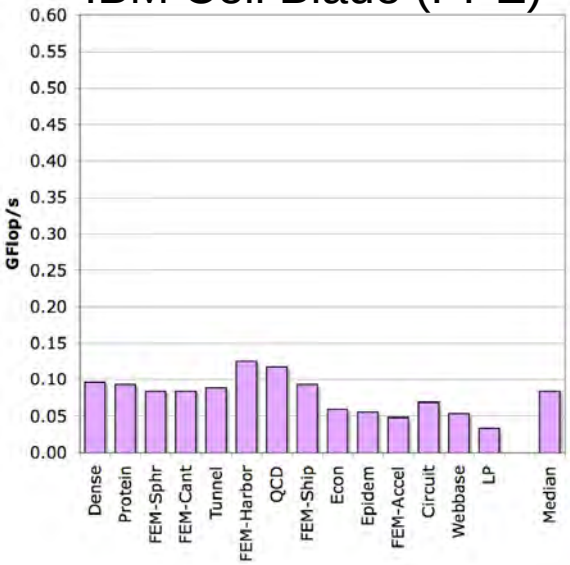
AMD Opteron



Sun Niagara2 (Huron)



IBM Cell Blade (PPE)

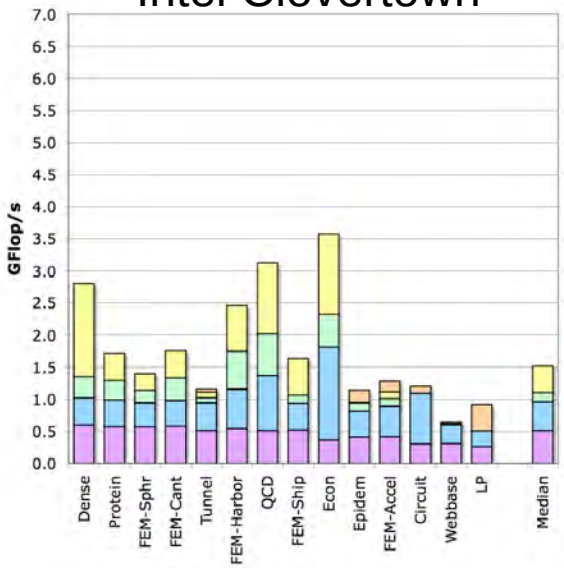


- **Vanilla C implementation**
- **Matrix stored in CSR (compressed sparse row)**
- **Explored compiler options, but only the best is presented here**
- **x86 core delivers > 10x the performance of a Niagara2 thread**

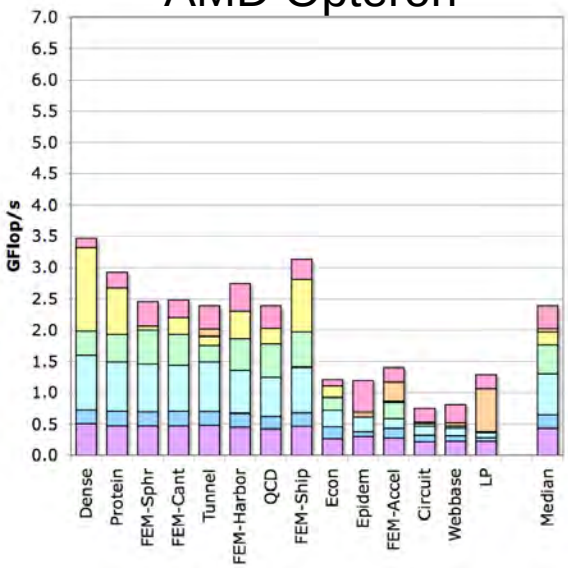
- **Work by Sam Williams with Vuduc, Oliker, Shalf, Demmel, Yelick**

Autotuned Performance

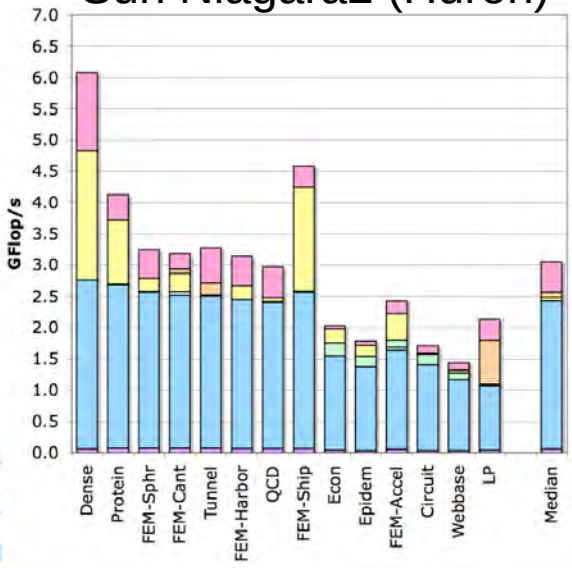
Intel Clovertown



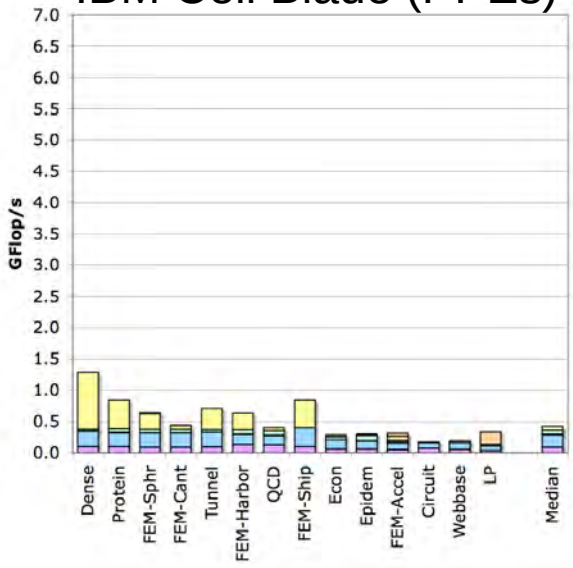
AMD Opteron



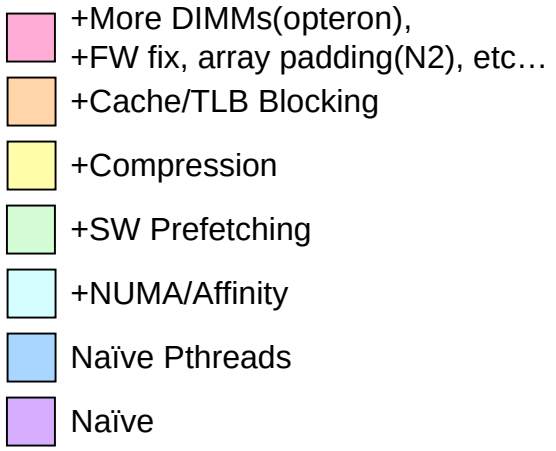
Sun Niagara2 (Huron)



IBM Cell Blade (PPEs)



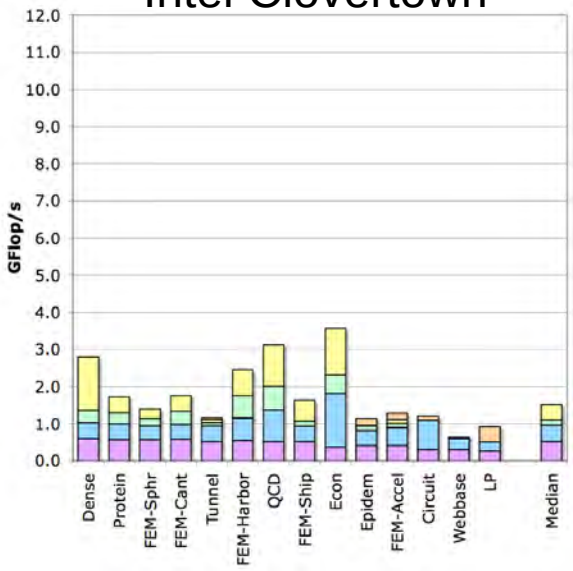
- **Threads alone doesn't show much benefit on AMD; need to control memory layout**



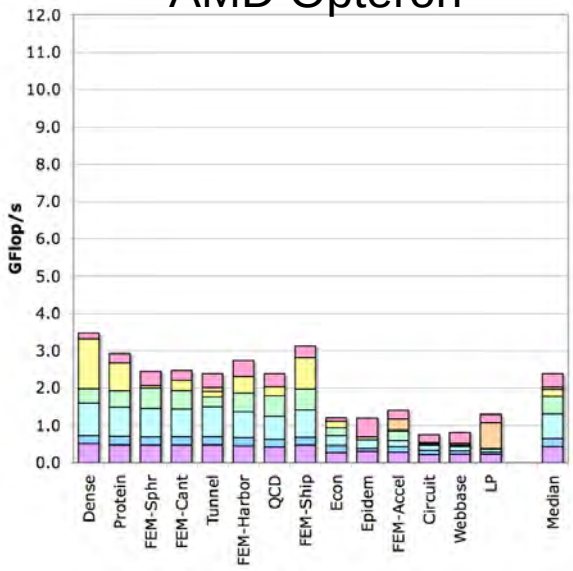
Autotuned Performance

(+Cell/SPE version)

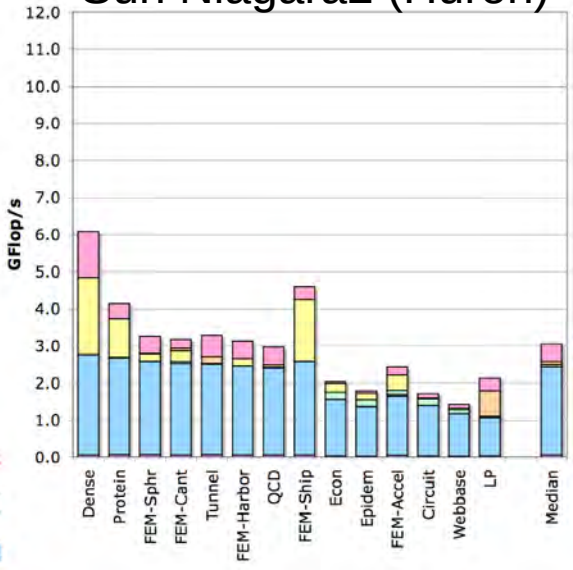
Intel Clovertown



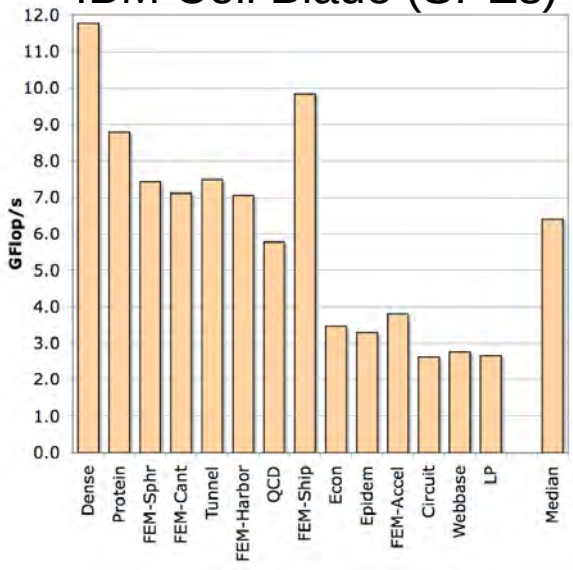
AMD Opteron



Sun Niagara2 (Huron)



IBM Cell Blade (SPEs)



- **Wrote a double precision Cell/SPE version**
- **DMA, local store blocked, NUMA aware, etc…**
- **Only 2x1 and larger BCOO**
- **Only the SpMV-proper routine changed**
- **About 12x faster (median) than using the PPEs alone.**

+More DIMMs(opteron),
+FW fix, array padding(N2), etc…

+Cache/TLB Blocking

+Compression

+SW Prefetching

+NUMA/Affinity

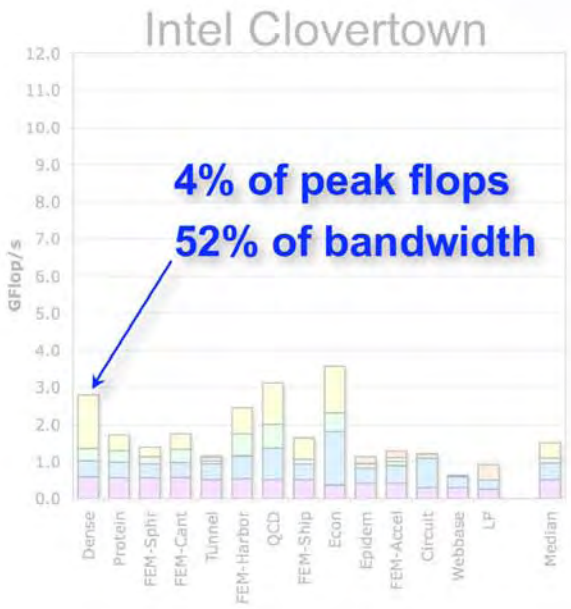
Naïve Pthreads

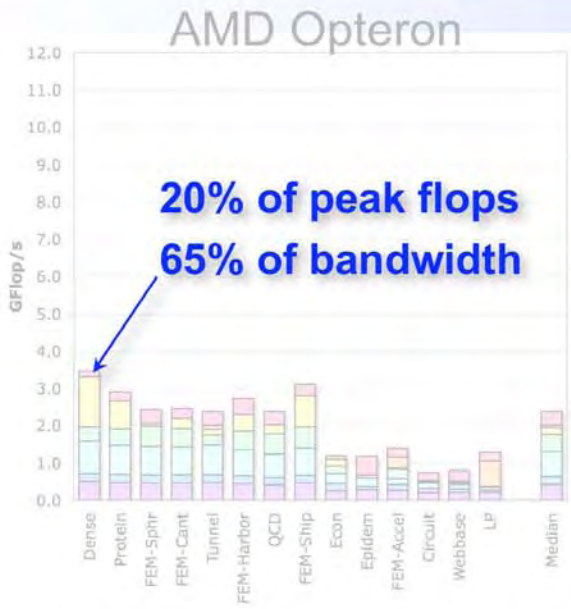
Naïve

BERK

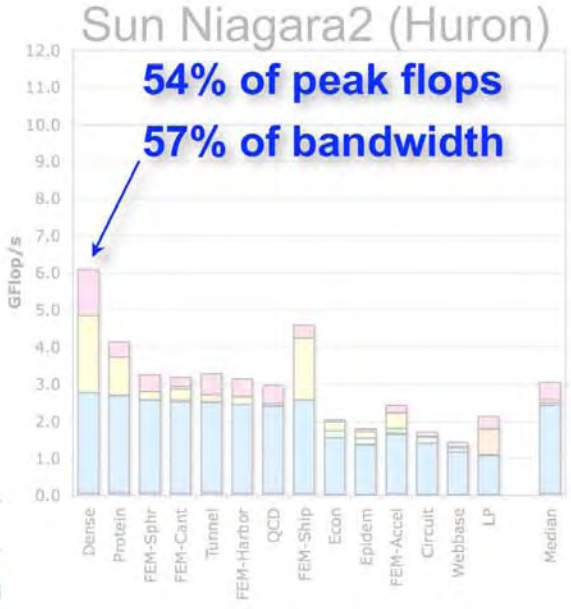
Autotuned Performance

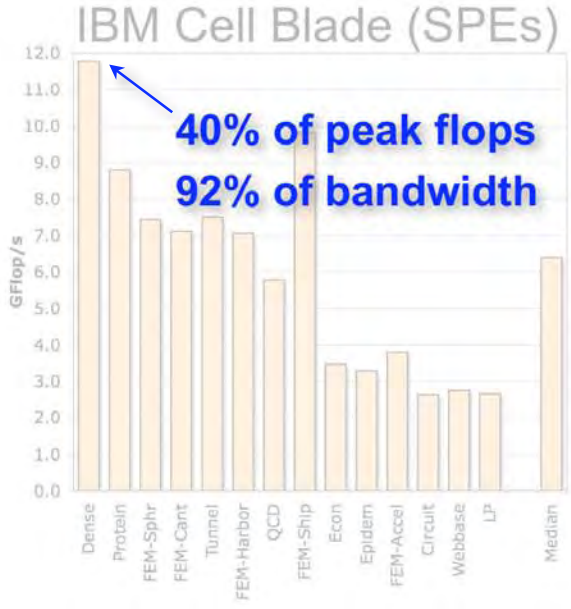
(+Cell/SPE version)





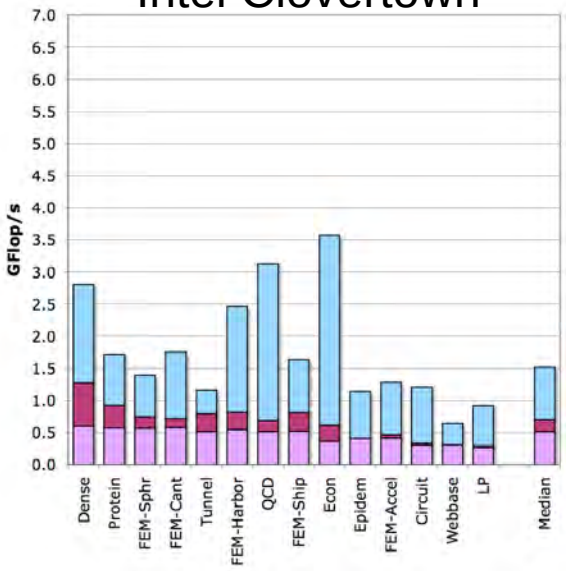
- Wrote a double precision Cell/SPE version
- DMA, local store blocked, NUMA aware, etc…
- Only 2x1 and larger BCOO
- Only the SpMV-proper routine changed
- About 12x faster than using the PPEs alone.



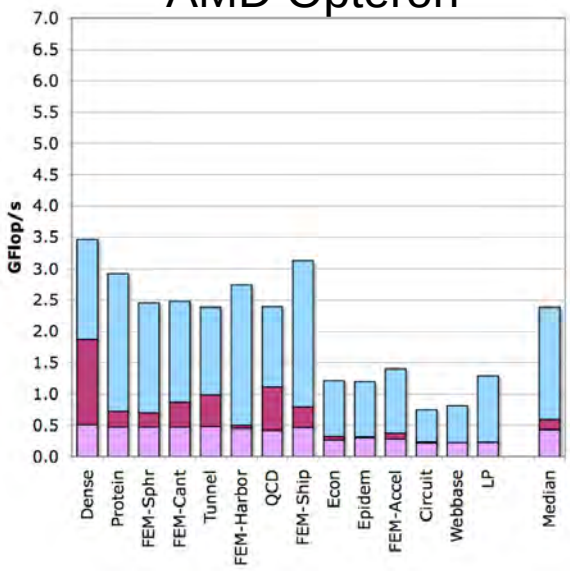


MPI vs. Threads

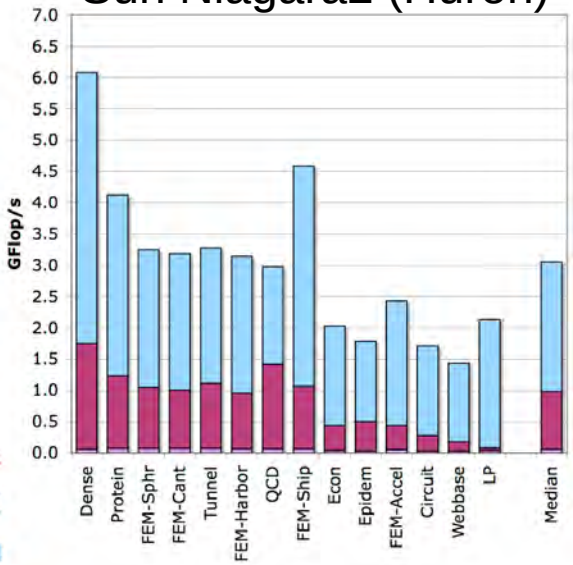
Intel Clovertown

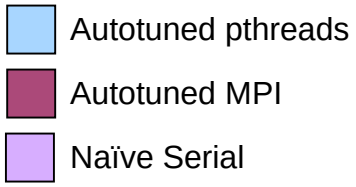


AMD Opteron



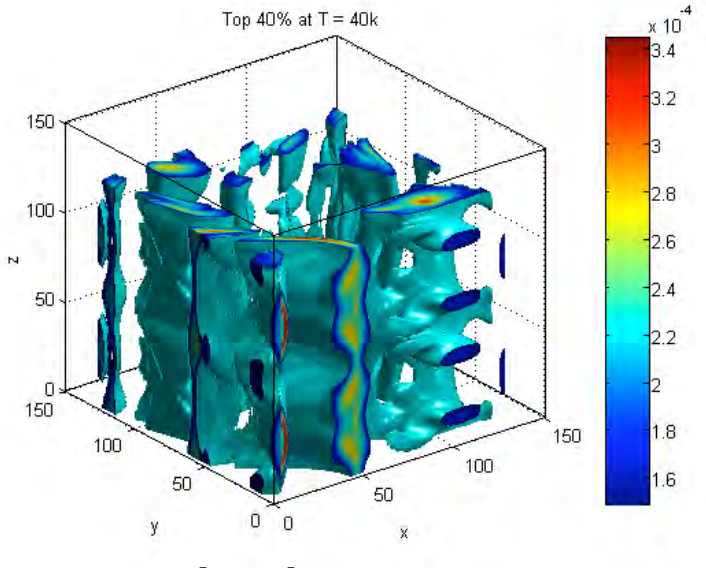
Sun Niagara2 (Huron)





- **On x86 machines, autotuned shared memory MPICH implementation rarely scales beyond 2 threads**
- **Still debugging MPI issues on Niagara2, but so far, it rarely scales beyond 8 threads.**

LBMHD: Structure Grid Application



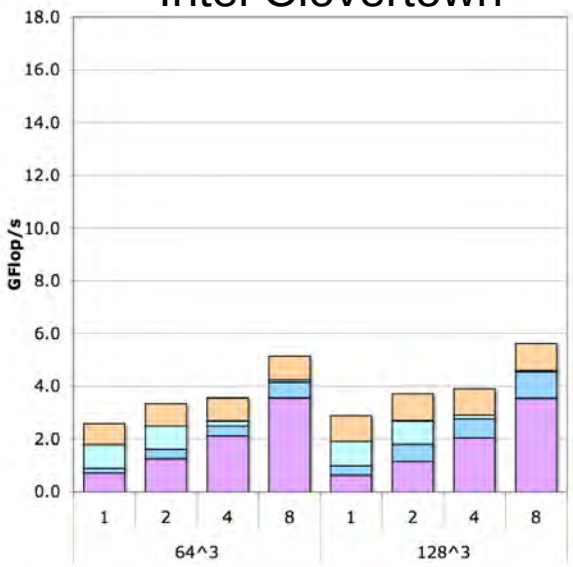
- Plasma turbulence simulation
- Two distributions:
  - momentum distribution (27 components)
  - magnetic distribution (15 vector componen
- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)
- Must read 73 doubles, and update(write) 7 space
- Requires about 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

- Work by Sam Williams with Jonathan Carter, Lenny Oliker, John Shalf, and Kathy Yelick

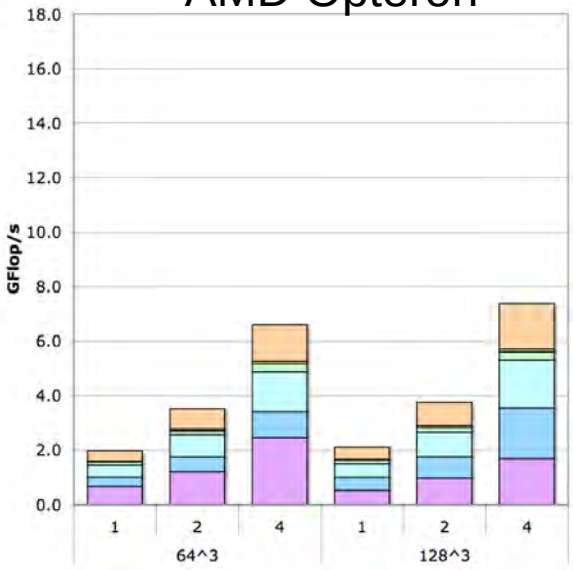
Autotuned Performance

(Cell/SPE version)

Intel Clovertown

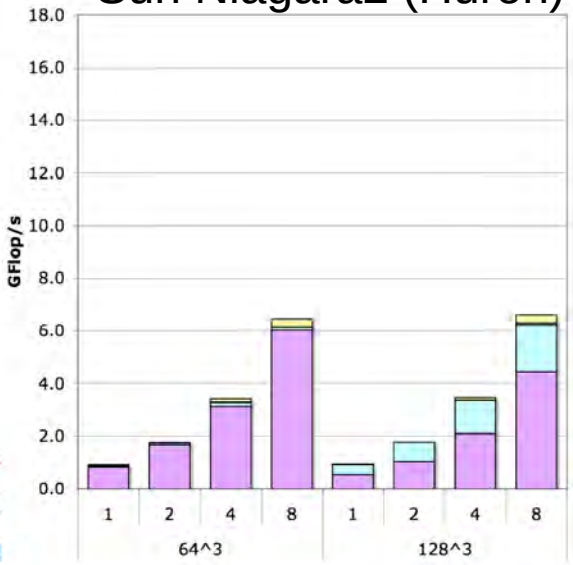


AMD Opteron

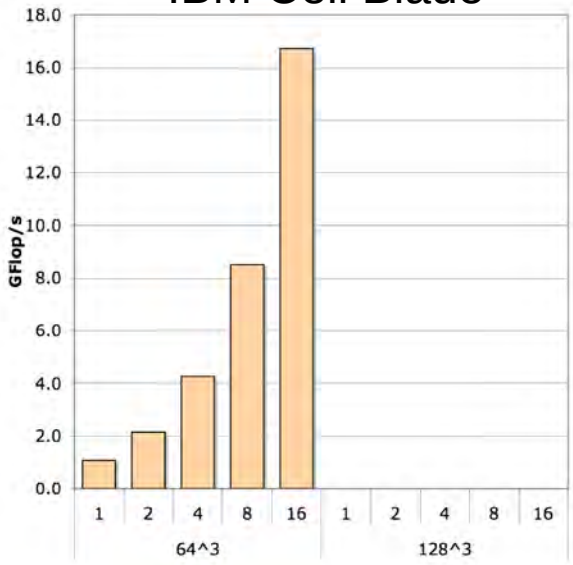


- **First attempt at cell implementation.**
- **VL, unrolling, reordering fixed**
- **Exploits DMA and double buffering to load vectors**
- **Straight to SIMD intrinsics.**
- **Despite the relative performance, Cell's DP implementation severely impairs performance**

Sun Niagara2 (Huron)



IBM Cell Blade*



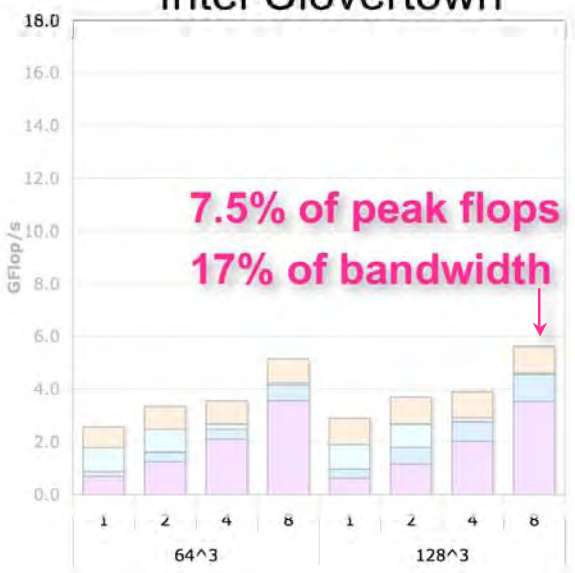
+SIMDization
+SW Prefetching
+Unrolling
+Vectorization
+Padding
Naïve+NUMA

*collision() only

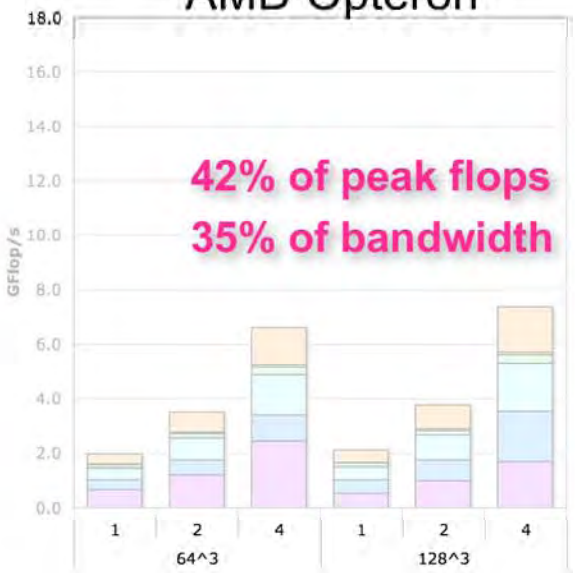
Autotuned Performance

(Cell/SPE version)

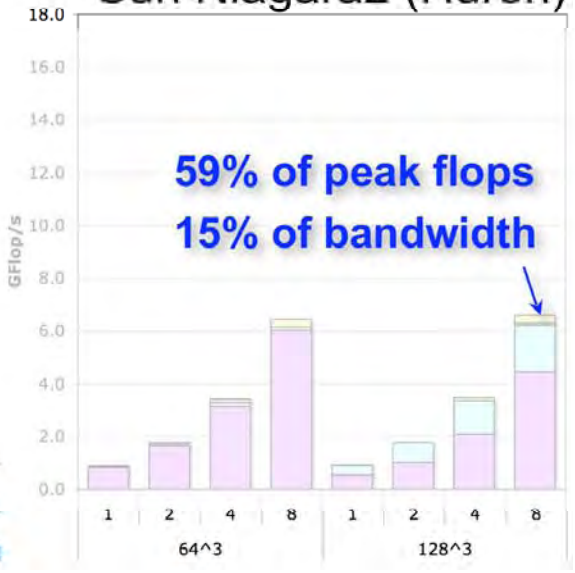
Intel Clovertown



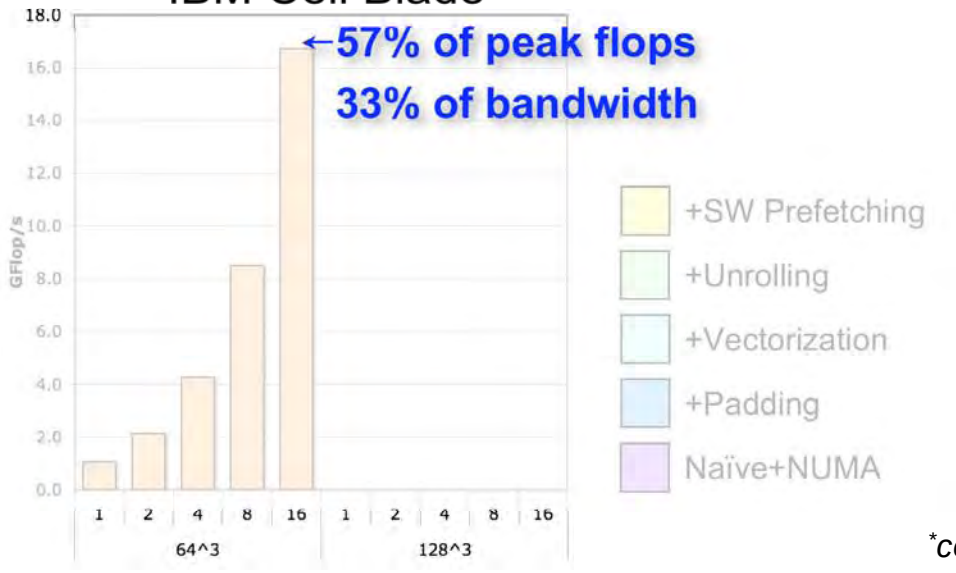
AMD Opteron



Sun Niagara2 (Huron)



IBM Cell Blade*



*collision() only

Lessons Learned

- Number of cores/chip will grow with Moore's Law
- Experience on multicore today is limited
  - Current multicore (AMD and Intel) look like SMPs, but GPUs, games, and others will influence them
- Explicitly manage memory is easier to tune for
  - Put/Get (DMA) operations are a good fit (PGAS?)
- MPI is not a viable model for Exascale
  - Does not get best performance
  - Does not permit sharing, which will be critical
    - Sharing data will be critical as cores count grows
    - A 10x10x10 grid subdomain has >50% surface area; we can't afford "ghost" regions at this scale or below
  - Even threads may be too heavy-weight for 1K cores
- Need a new dynamic model for software
  - Write programs to write applications and libraries

Rethinking Algorithms

Count memory, not Flops

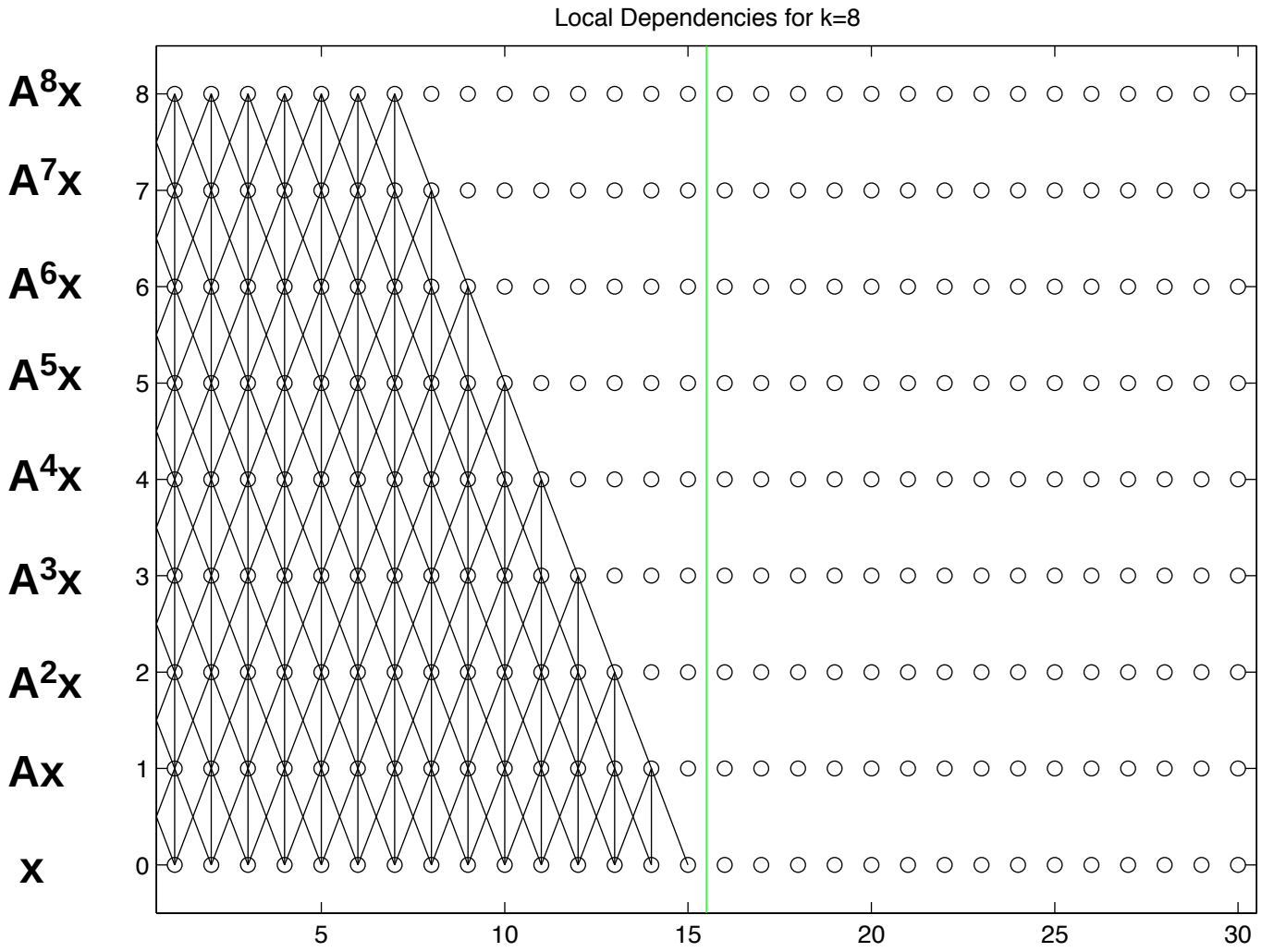
Latency and Bandwidth-Avoiding

- 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

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

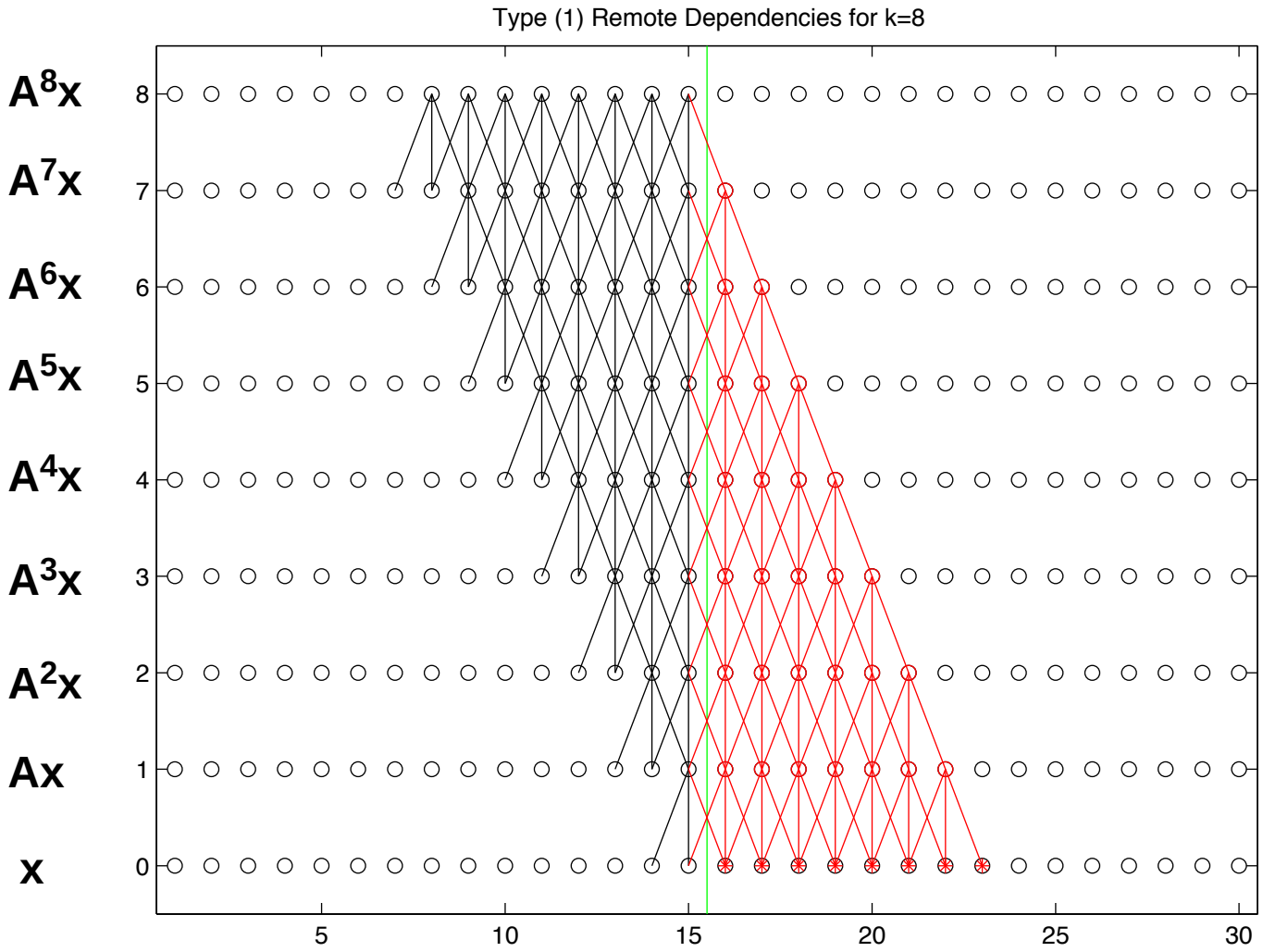
- Compute locally dependent entries needed by neighbors
- Send data to neighbors, receive from neighbors
- Compute remaining locally dependent entries
- Wait for receive
- Compute remotely dependent entries

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



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

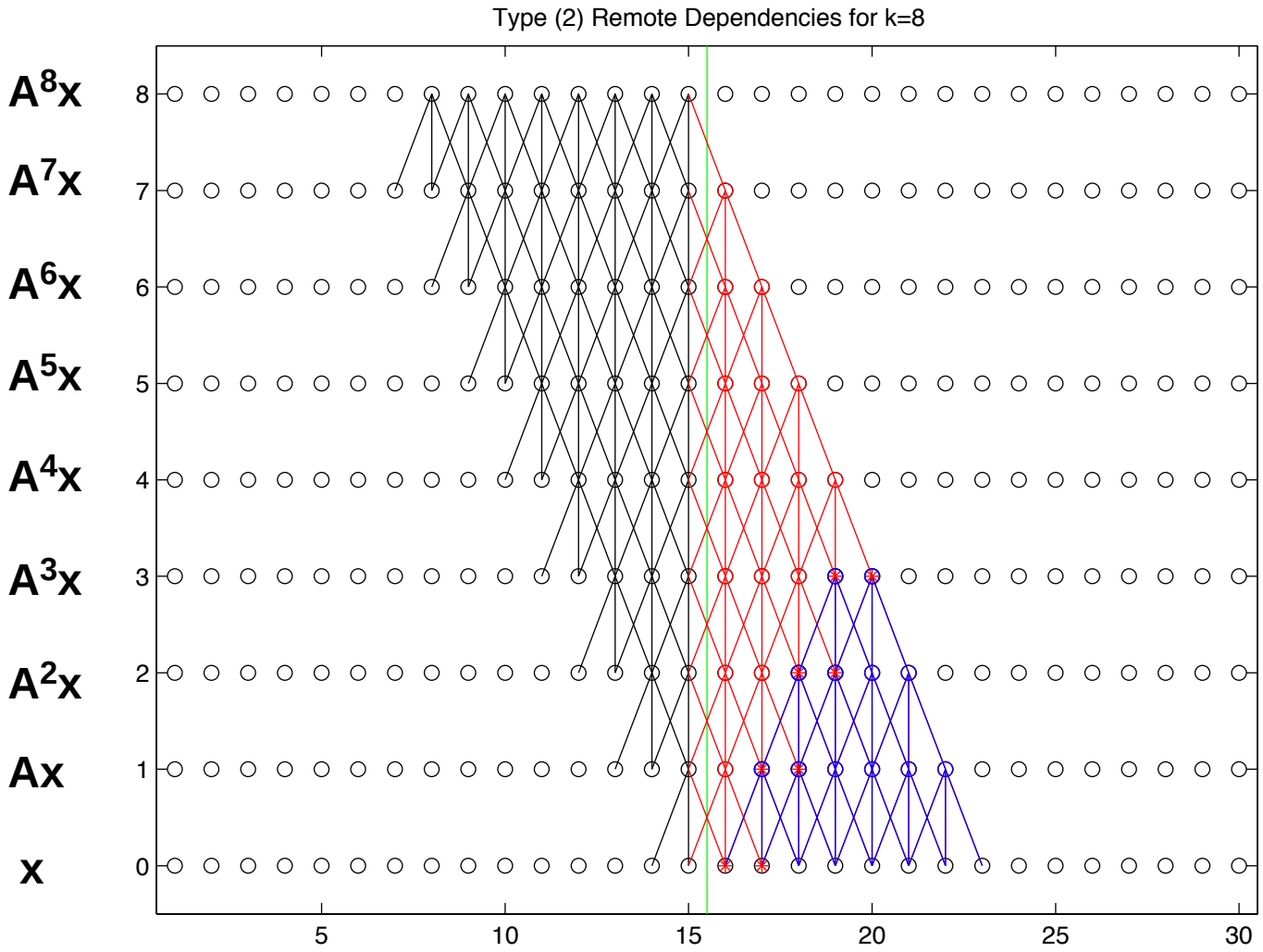
Remotely Dependent Entries for [x,Ax,…,A^8x], A tridiagonal



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

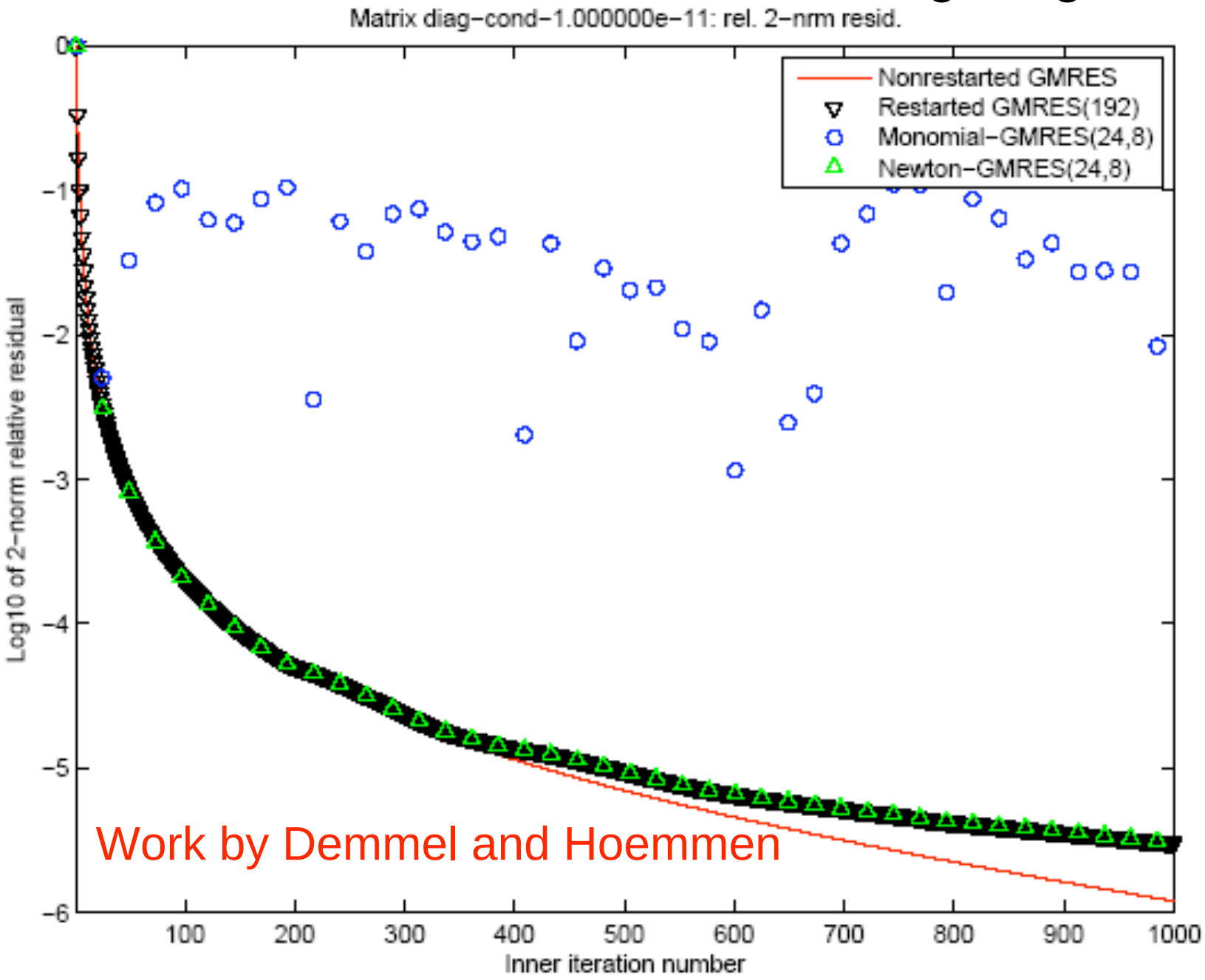
Price: redundant work

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



Reduce redundant work by half

Can use Matrix Power Kernel, but change Algorithms



Work by Demmel and Hoemmen

Conclusions

- **Re-think Programming Models**
  - **Software to make the most of hardware**
    - **One-sided communication to avoid synchronization**
    - **Global address space to increase sharing (re-use) and for productivity**
- **Re-think software for libraries/applications**
  - **Write self-tuning applications**
- **Re-think Algorithms**
  - **Design for bottlenecks: latency and bandwidth**