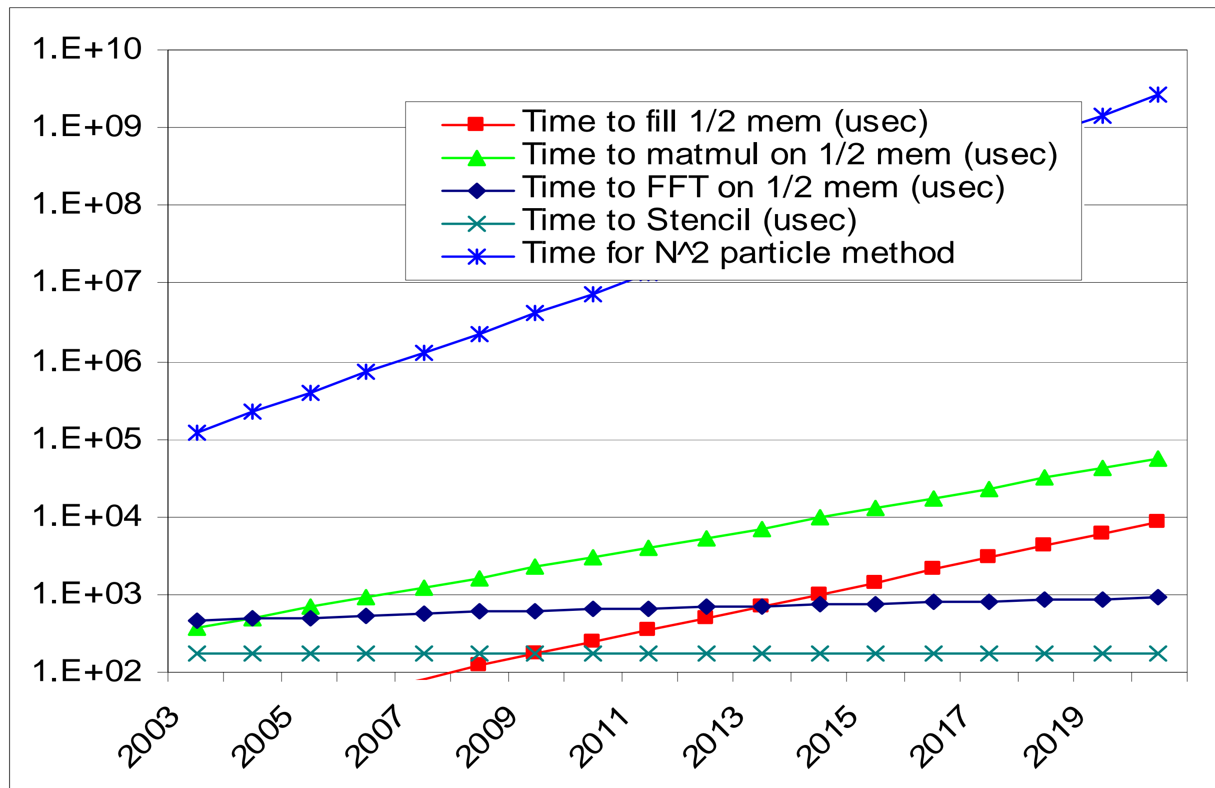
Programming Techniques to Harness Exaflops

Kathy Yelick
U.C. Berkeley and LBNL

How to Waste a Zettaflop machine

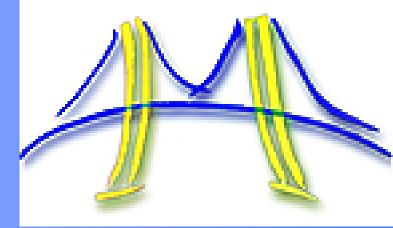
#1: Insufficient Memory Bandwidth

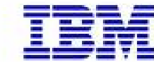
- Required bandwidth depends on the algorithm
- Need hardware designed to algorithmic needs



BERKELEY LAB

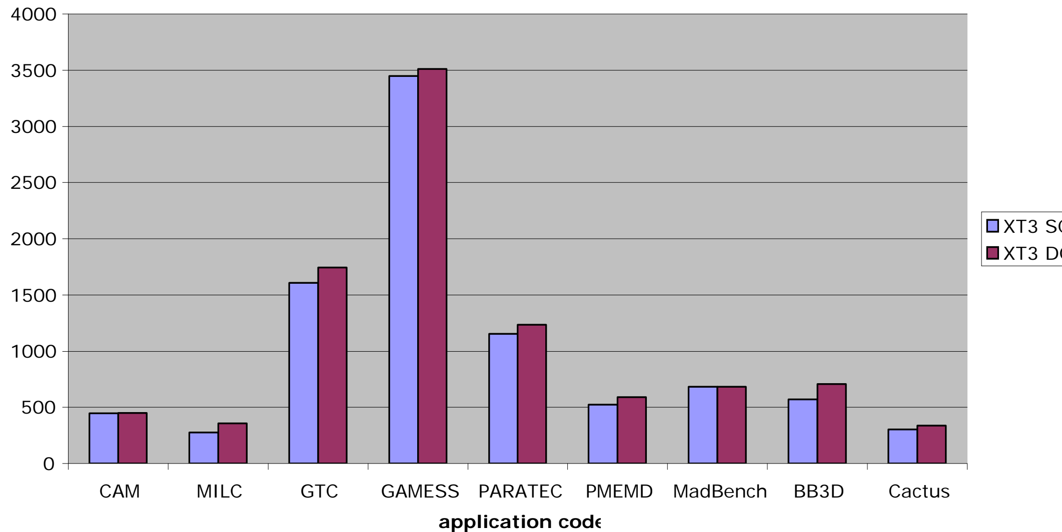
#2: Ignore Little's Law

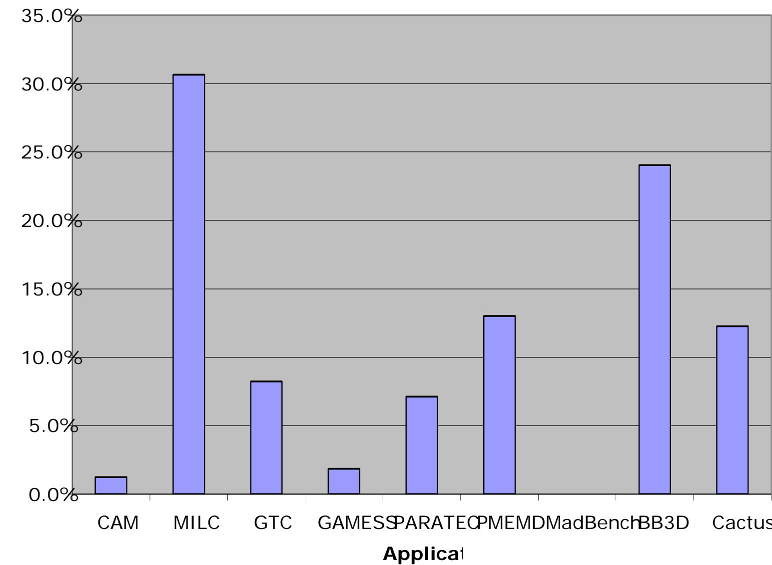




Name	Clovertown	Opteron	Cell
Chips*Cores	2*4 = 8	2*2 = 4	1*8 = 8
Architecture	4-/3-issue, 2-/1-SSE3, OOO, caches, prefetch		2-VLIW, SIMD, local RAM, DMA
Clock Rate	2.3 GHz	2.2 GHz	3.2 GHz
Peak MemBW	21.3 GB/s	21.3	25.6 GB/s
Peak GFLOPS	74.6 GF	17.6 GF	14.6 (DP Fl. Pt.)
Naïve SpMV (median of many matrices)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

NERSC SSP Applications

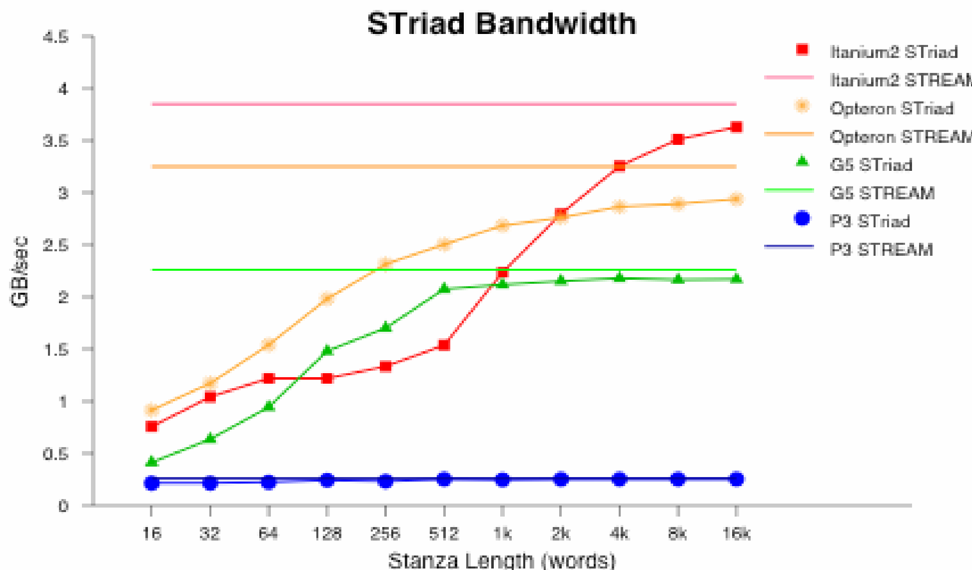


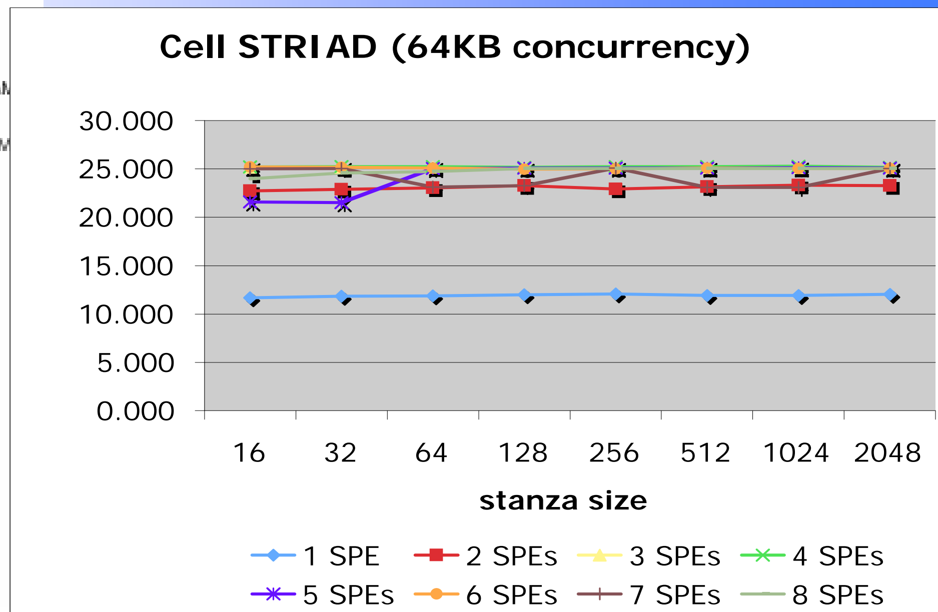


- **Still 10% drop on average when halving memory bandwidth!**
  - **#$%^&* application developers write crummy code!**
  - **Lets pick an application that *I KNOW* is memory bandwidth bound!**

Why is the STI Cell So Efficient?

(Latency Hiding with Software Controlled Memory)

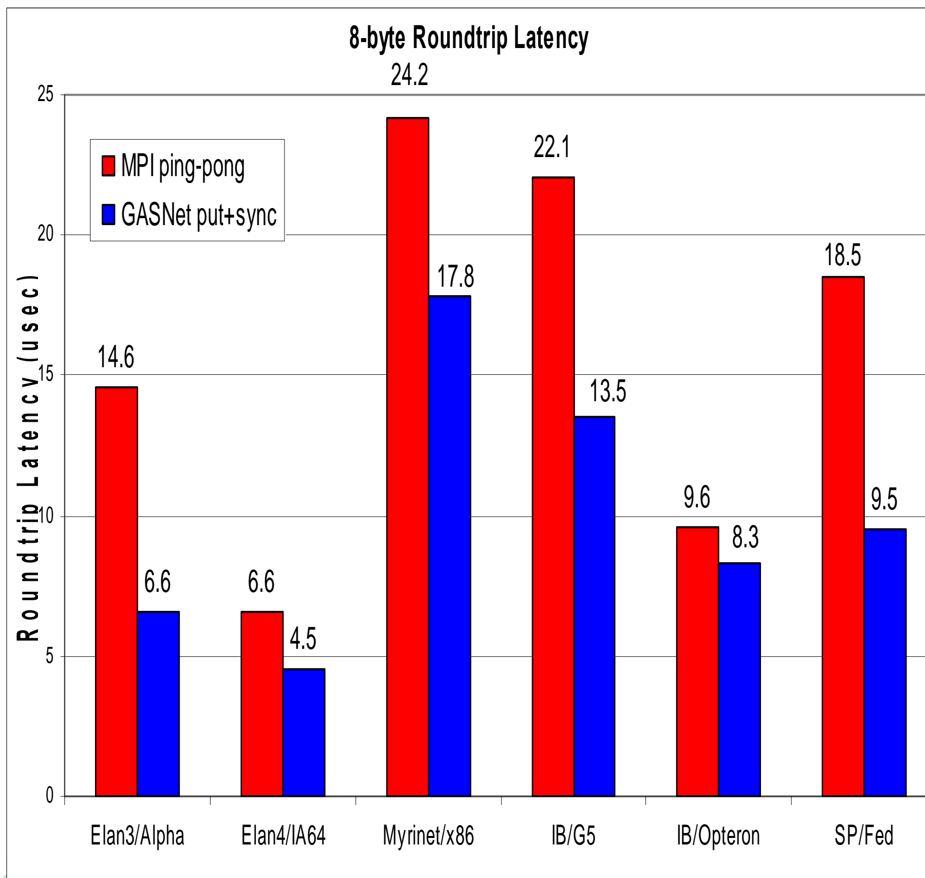


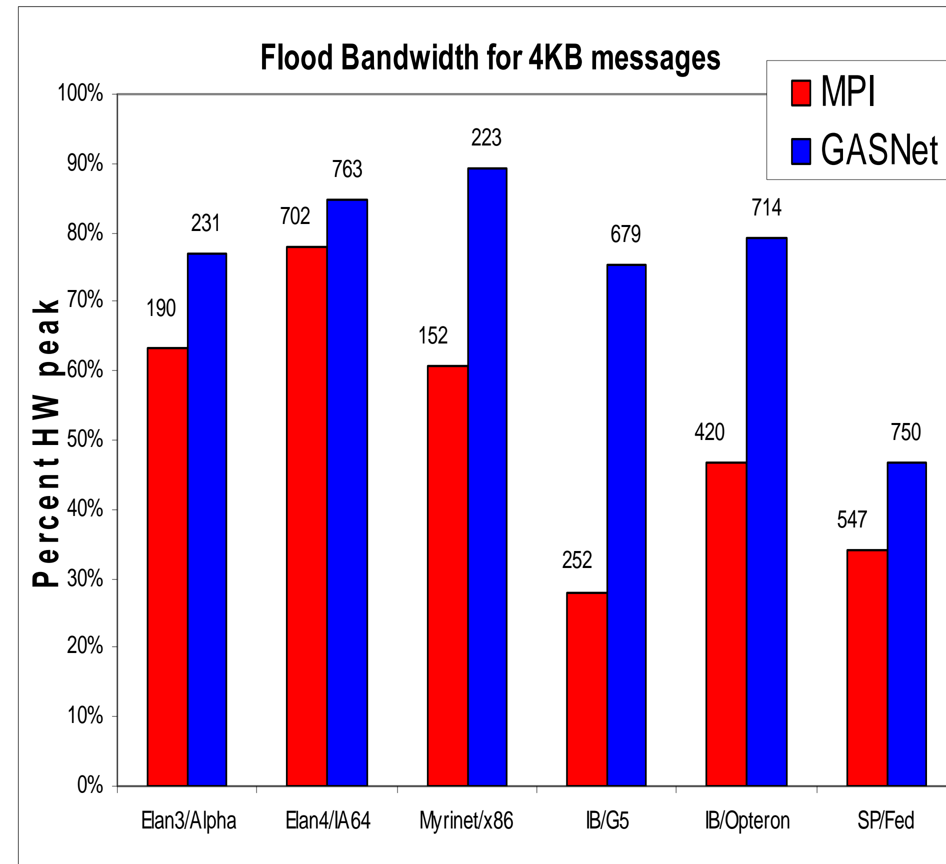


- **Performance of Standard Cache Hierarchy**
  - **Cache hierarchies underutilize memory bandwidth due to inability to tolerate latency**
  - **Hardware prefetch prefers long unit-stride access patterns (optimized for STREAM)**
  - **But in practice, access patterns are for shorter stanzas: so never reaches peak bandwidth (still latency limited)**
- **Cell "explicit DMA"**
  - **Cell software controlled DMA engines can provide nearly flat response for a variety of access patterns**
  - **Response is nearly full memory bandwidth can be utilized for all access patterns**
  - **Cell memory requests can be nearly completely hidden behind the computation due to asynchronous DMA engines**
  - **Performance model is simple and deterministic (much simpler than modeling a complex cache hierarchy),** min{time_for_memory_ops, time_for_core_exec}
  - **Problem: lack of tractable/broadly applicable programming model**

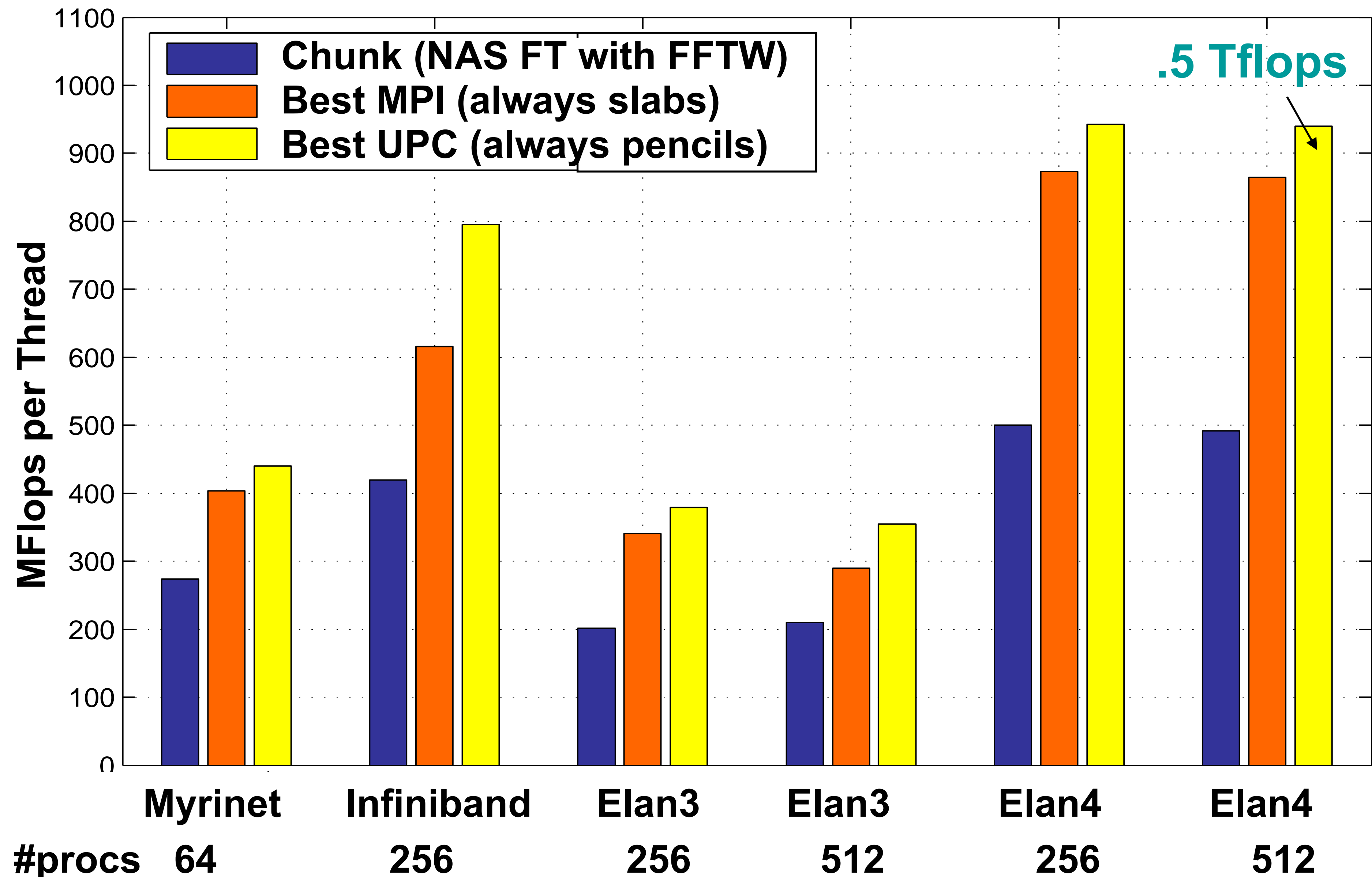
#3: Unnecessarily Synchronize Communication

Use a programming model in which you can't utilize bandwidth or "low" latency





NAS FT Variants Performance Summary



How to Efficiently Use a Zettascale System

- **Rethink hardware**
  - Parallelism is mainstream, but most cores are optimized for serial performance
  - Need to design hardware for power and parallelism
- **Rethink software**
  - Massive parallelism
  - Eliminate scaling bottlenecks replication, synchronization
- **Rethink algorithms**
  - Massive parallelism and locality
  - Counting Flops is the wrong measure

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) size is finite
  - Links in network topology are generally $< n^2$

- Does the programming model (language+libraries) expose this or hide it?

A Brief History of Languages

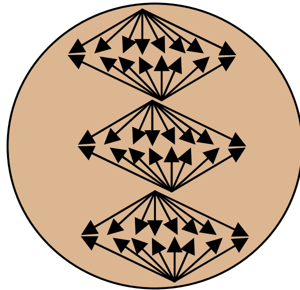
- When vector machines were king
  - Parallel "languages" were loop annotations (IVDEP)
  - Performance was fragile, but there was good user support
- When SIMD machines were king
  - Data parallel languages popular and successful (CMF, *Lisp, C*, …)
  - Quite powerful: can handle irregular data (sparse 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.

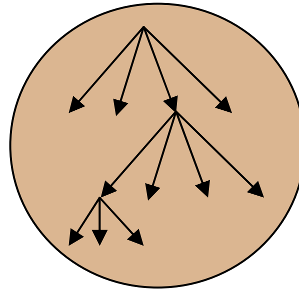
BERKELEY LAB

Two Parallel Language Questions

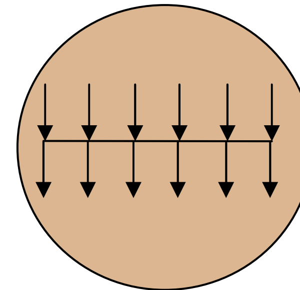
- **What is the parallel control model?**



data parallel
(singe thread of control)

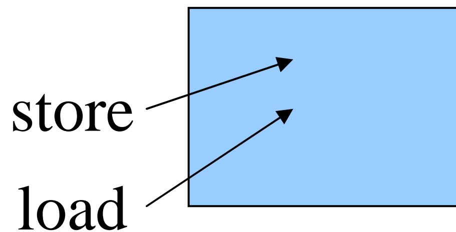


dynamic
threads

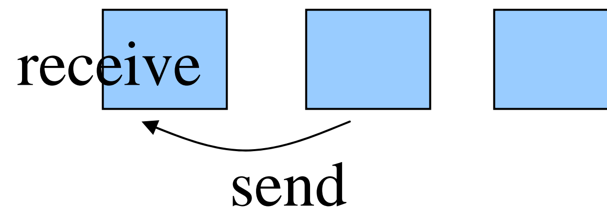


single program
multiple data (SPMD)

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



shared memory



message passing

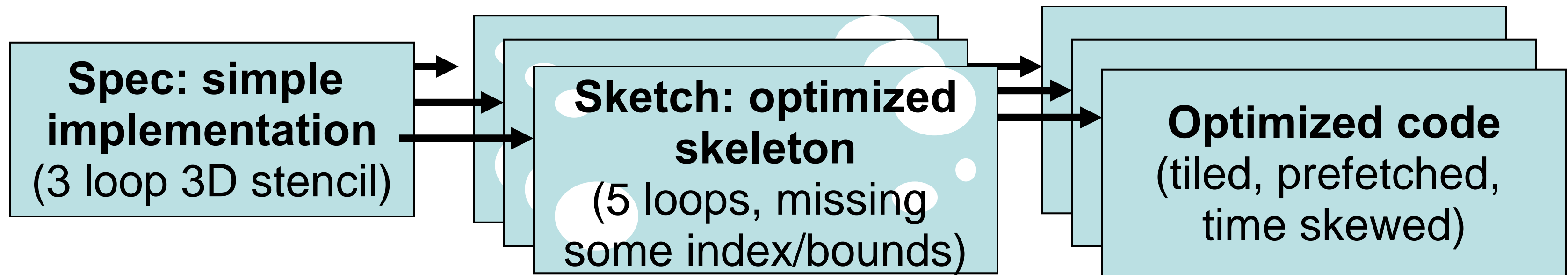
implied synchronization for message passing, not shared memory

Strategies for Zettascale Software

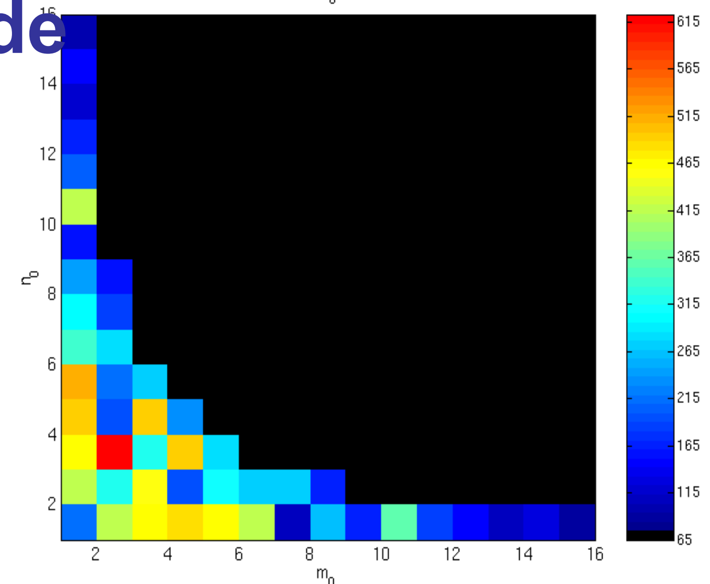
Rethink our Software Models

Program Synthesis

- Needs extensive tuning knobs for writing basic code
- Don't do this by hand: tools for tuning

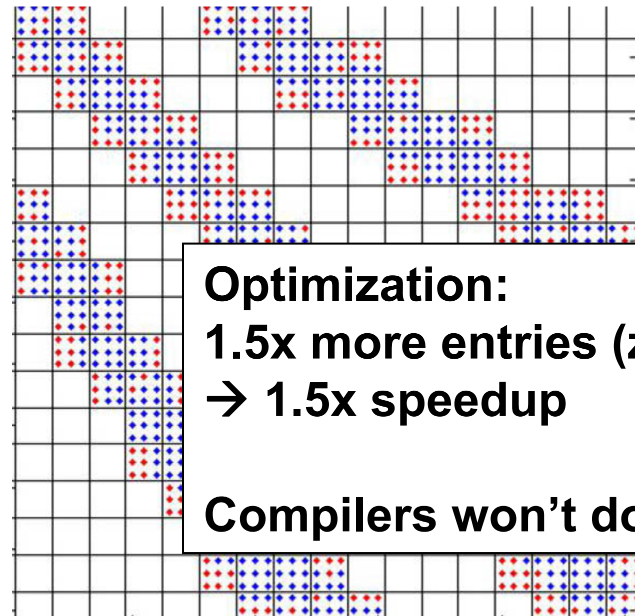


- Autotuning: self-tuning code
  - Can select from algorithms/data structures changes not producible by compiler transform



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!

Clovertown

1Core Naïve
1Core PF
1Core PF,RB
1Core PF,RB,CB
2Core
4Core

GFlop/s

2.5
2.0
1.5
1.0
0.5
0.0

Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median

AMD X2

1Core Naïve
1Core PF
1Core PF,RB
1Core PF,RB,CB
2Core

GFlop/s

3.0
2.5
2.0
1.5
1.0
0.5
0.0

Autotuning boosts performance of single and multiple cores

Clovertown

1Core Naïve
1Core PF
1Core PF,RB
1Core PF,RB,CB
2Core
4Core
2Socket x 4Core

GFlop/s

0.0
0.5
1.0
1.5
2.0
2.5

Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median

AMD X2

1Core Naïve
1Core PF
1Core PF,RB
1Core PF,RB,CB
2Core
2Socket x 2Core

GFlop/s

0.0
0.5
1.0
1.5
2.0
2.5
3.0

Autotuning boosts performance of multiple socket SMPs

Clovertown

1Core Naïve
1Core PF
1Core PF,RB
1Core PF,RB,CB
2Core
4Core
2Socket x 4Core
2Core Naïve
MPI+OSKI

GFlop/s
2.5
2.0
1.5
1.0
0.5
0.0

Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median

AMD X2

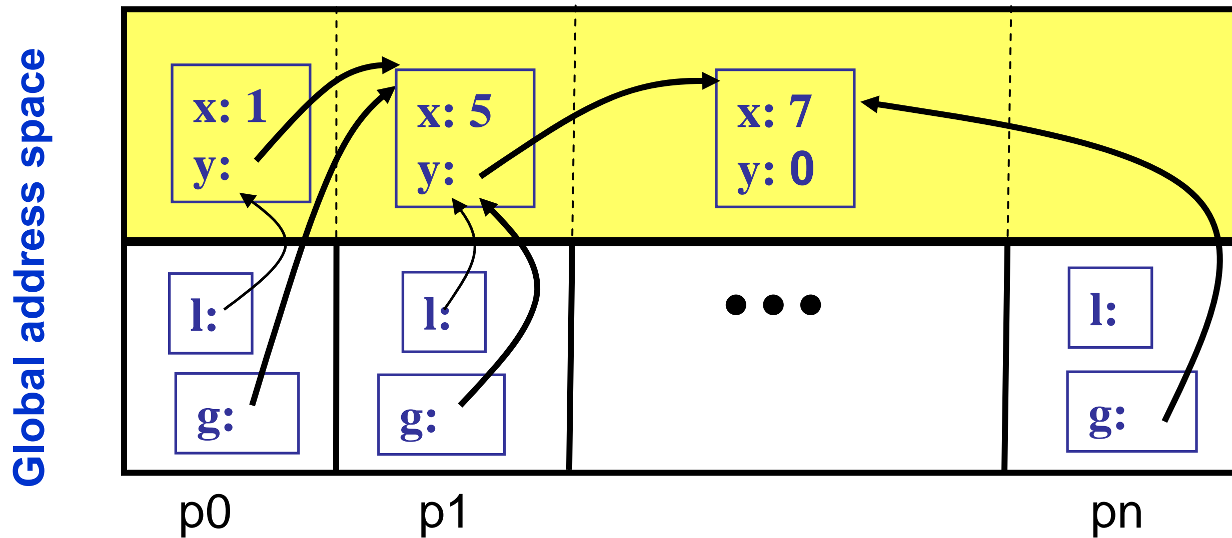
1Core Naïve
1Core PF
1Core PF,RB
1Core PF,RB,CB
2Core
2Socket x 2Core
2Core Naïve
MPI + OSKI

GFlop/s
3.0
2.5
2.0
1.5
1.0
0.5
0.0

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

What About PGAS Languages?

- ***Global address space:*** **any thread/process may directly read/write data allocated by another**
- ***Partitioned:*** **data is designated as local (near) or global (possibly far); programmer controls layout**

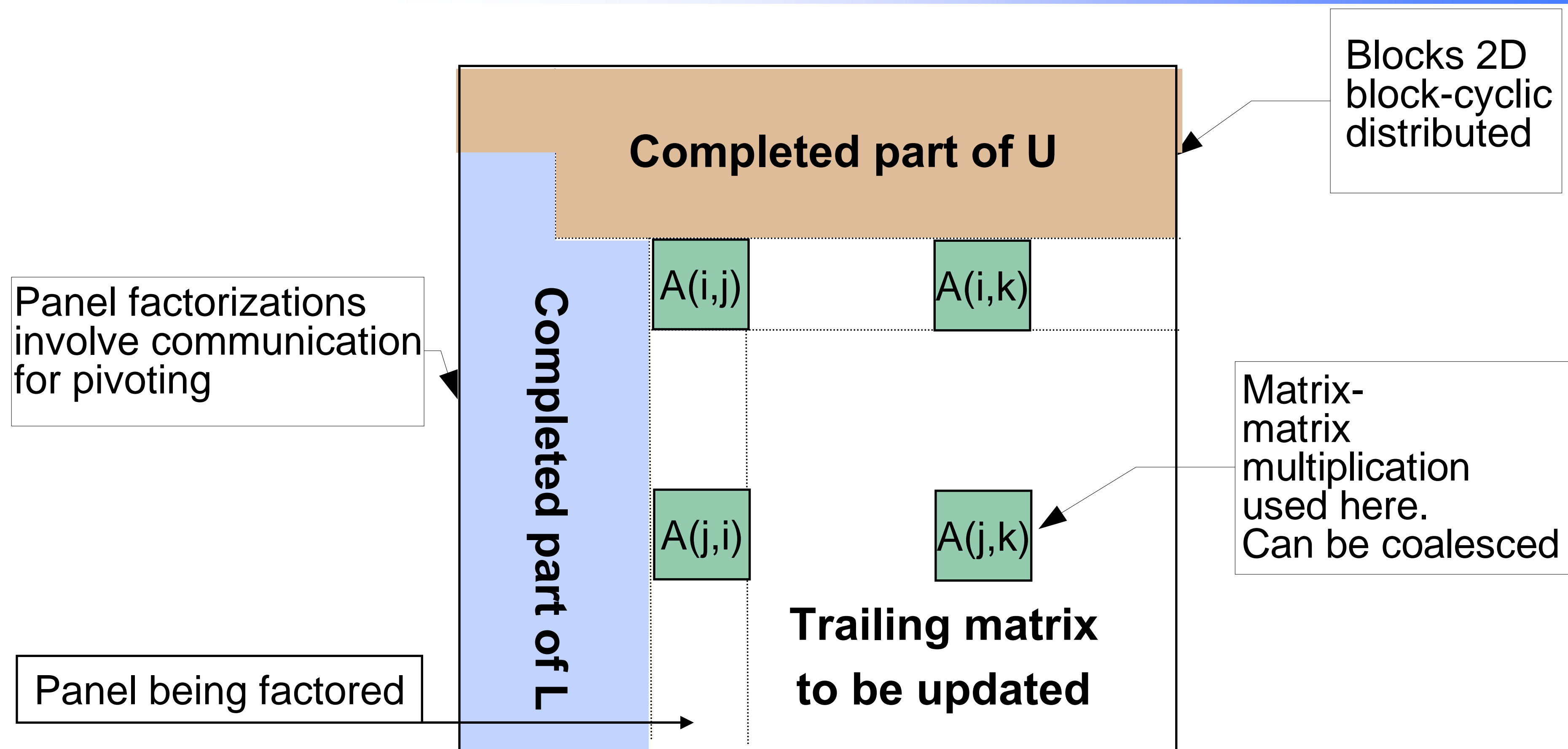


- ***Processor per core will shrink:*** **need to avoid replication of system and application structures**

What about HPCS Languages?

- Chapel (Cray), X10 (IBM), and Fortress (Sun) were developed in HPCS program
- Are these languages "compilable"?
  - Yes, in principle; easier than HPF
  - Source-to-source model is proven
  - But still need work
- Several interesting differences (Java, all-new, etc.)
- Several interesting similarities
  - PGAS memory model
  - Dynamic (non SPMD) execution model
  - Mixed task and data parallelism
- When does this dynamic model help?
  - It does create a runtime challenge not yet "solved"

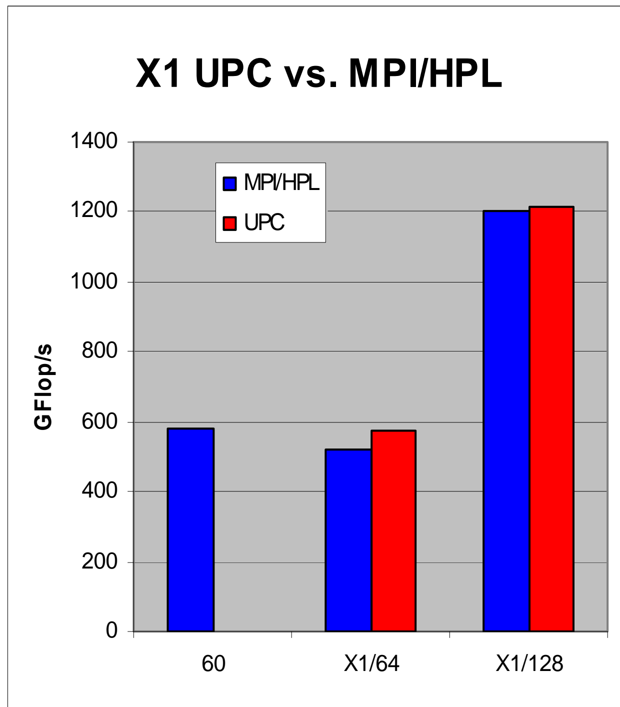
Matrix Factorization: Cautionary Tale

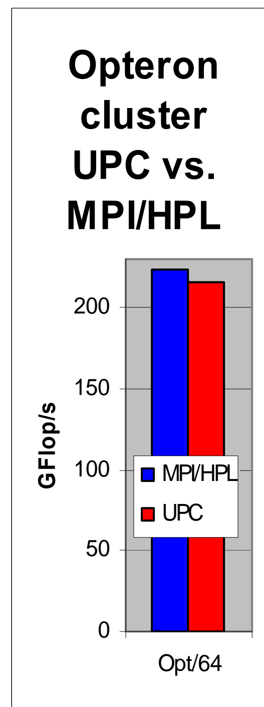


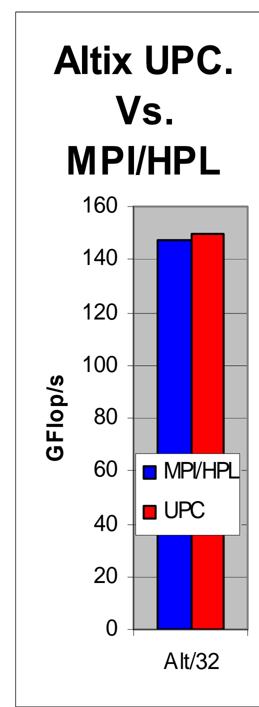
- **Current PGAS languages use static parallelism model (SPMD)**
- **Parallel matches address space partition**
- **LU factorization (sparse and dense) has highly irregular task graph**

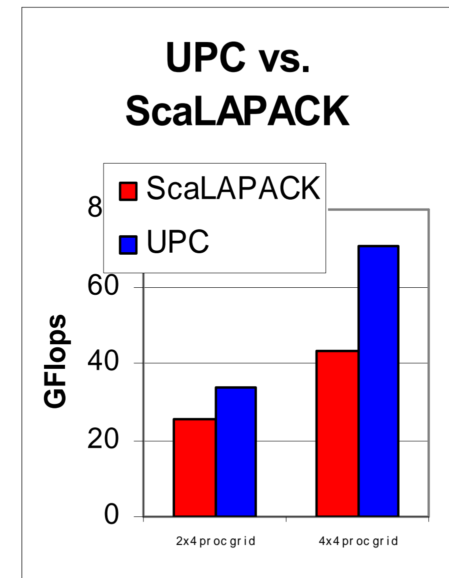
Joint work with Parry Husbands

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

Virtualization of Processors

- Highly multithreaded LU code is:
  - Roughly 2x faster than bulk-synchronous
  - As fast as MPI with limited asynchrony
- Experience
  - Highly tuned thread schedule for LU
  - Not yet there for sparse Cholesky
- Conclusion: multithreading is great, but software needs control over scheduling

New Programming Models

- Science is currently limited by the difficulty of programming
  - Codes get written somehow, but barrier to algorithm experimentation is too high
  - Multicore shift will make this worse: The application scientists have other things to do that worry about the next hw revolution
- Want an integrated programming model:
  - Express many levels and kinds of parallelism for multiple machine generations
  - Painful reprogramming should only be done once

Technical Challenges in Programming Models

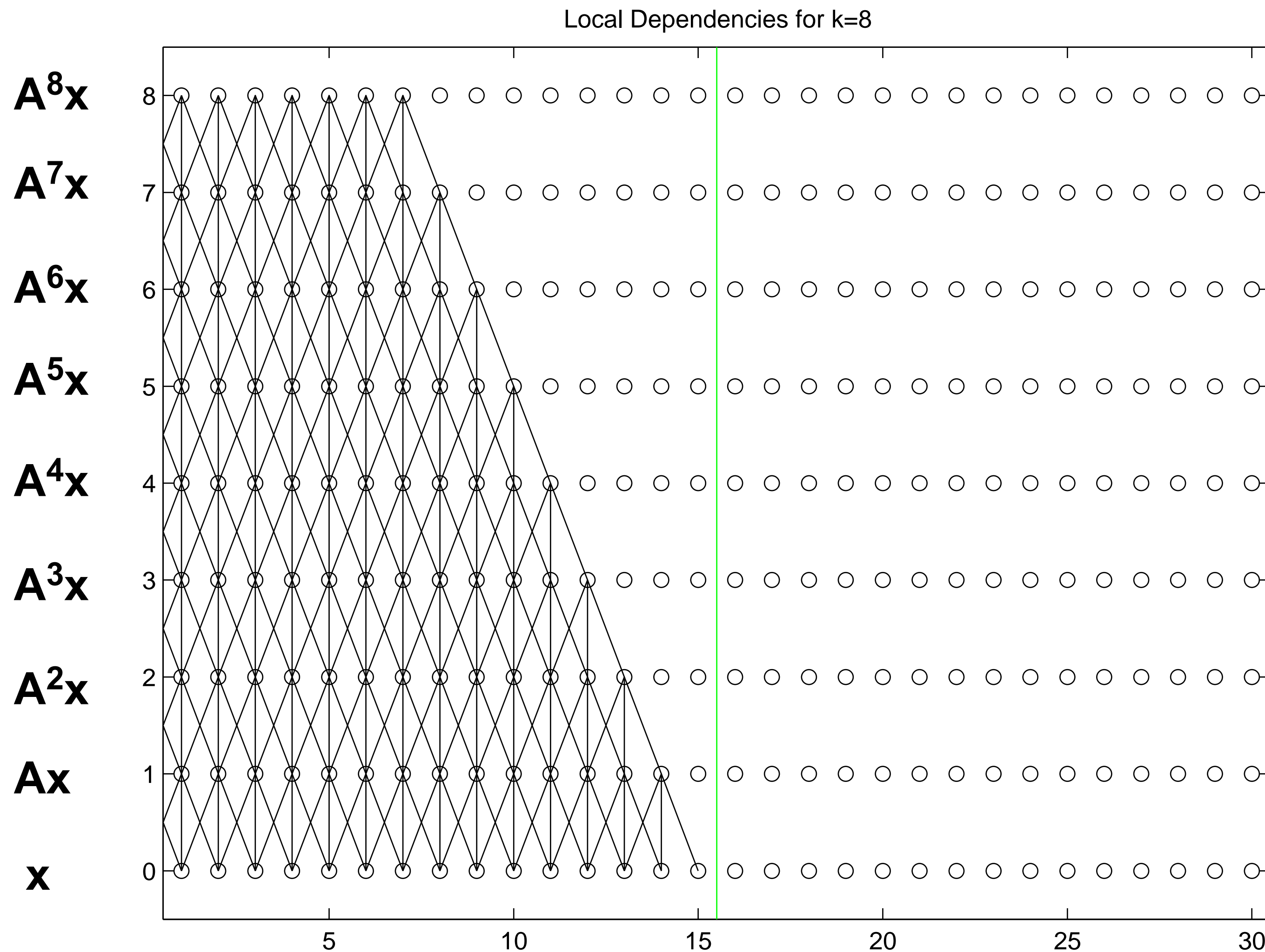
- Open problems in language runtimes
  - Virtualization: away from SPMD model for load balance, fault tolerance, OS noise, etc.
  - Resource management: thread scheduler
- What we do know how to do:
  - Build systems with dynamic load balancing (Cilk) that do not respect locality
  - Build systems with rigid locality control (MPI, UPC, etc.) that run at the speed of the slowest component
  - Put the programmer in control of resources: message buffers, dynamic load balancing

Rethink Algorithms

Latency and Bandwidth-Avoiding

- 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 $[x, Ax, \ldots, A^8x]$, A tridiagonal



Can be computed without communication

k=8 fold reuse of A

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

Type (1) Remote Dependencies for k=8

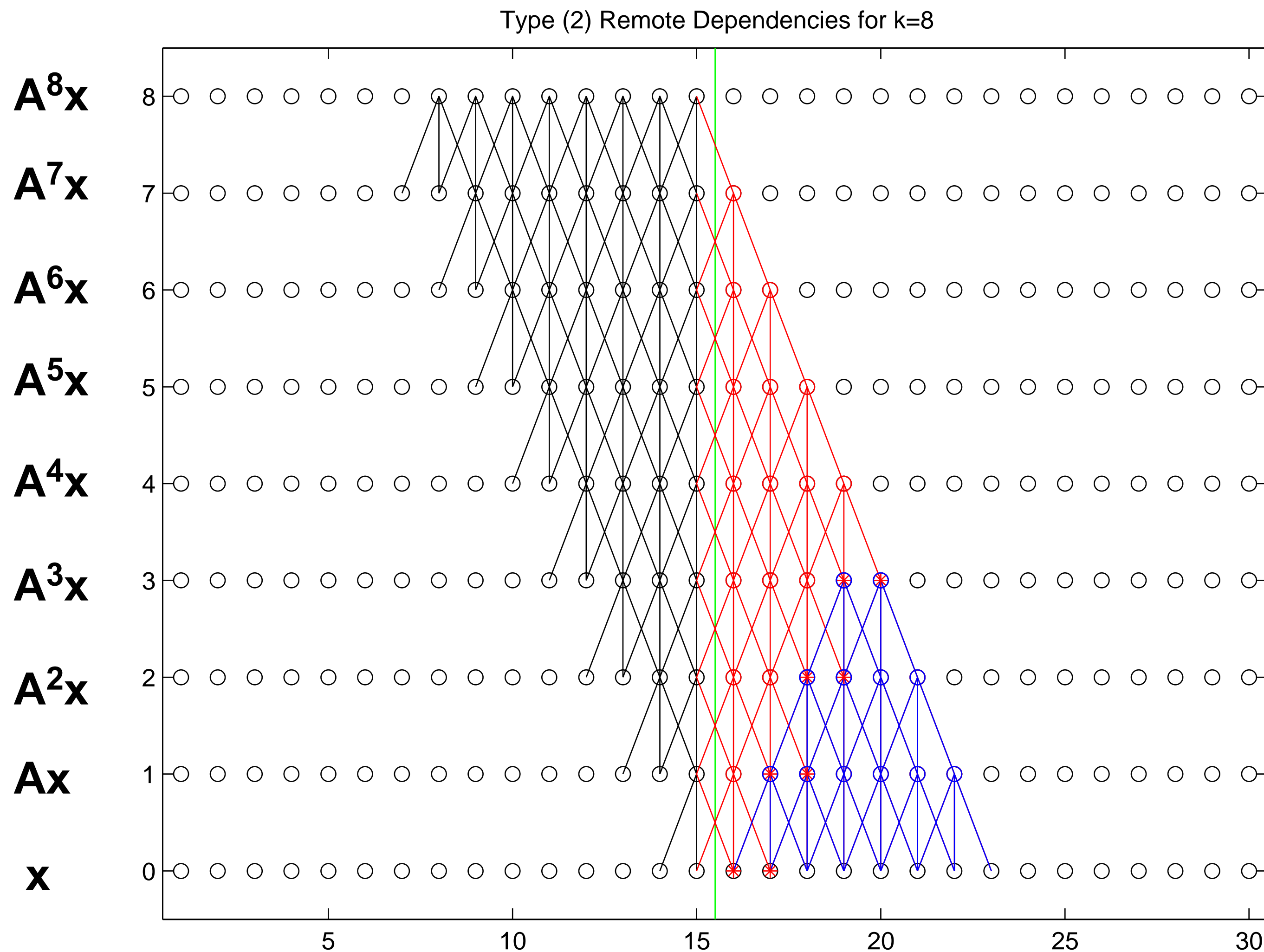
A^8x	8
A^7x	7
A^6x	6
A^5x	5
A^4x	4
A^3x	3
A^2x	2
Ax	1
x	0

5 10 15 20 25 30

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

Price: redundant work

Fewer Remotely Dependent Entries for $[x, Ax, \ldots, A^8x]$, 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

BERKELEY LAB

Fundamental Algorithm Change

- Use of matrix powers kernels is useful for performance
  - Reduces message count
  - Reduces memory bandwidth and latency
- Using traditional iterative methods, loses stability
  - Change of algorithm (basis) reclaims stability

- Conclusion: rethink algorithms from the ground up

Approach for Programming Models

- Work on programming model now
- Start with two Zettaflop apps
  - One easy: if anything scale, this will
  - One hard: plenty of parallelism, but it's irregular, adaptive, asynchronous
- Rethink algorithms
  - Scalability at all levels (including algorithmic)
  - Reducing bandwidth (compress data structures); reducing latency requirements
- Design programming model to express this parallelism
  - Develop technology to automate as much as possible (parallelism, HL constructs, search-based optimization)
- Consider spectrum of hardware possibilities
  - Analyze at various levels of detail (eliminating when they are clearly infeasible)
  - Early prototypes (expect 90% failures) to validate

Questions?