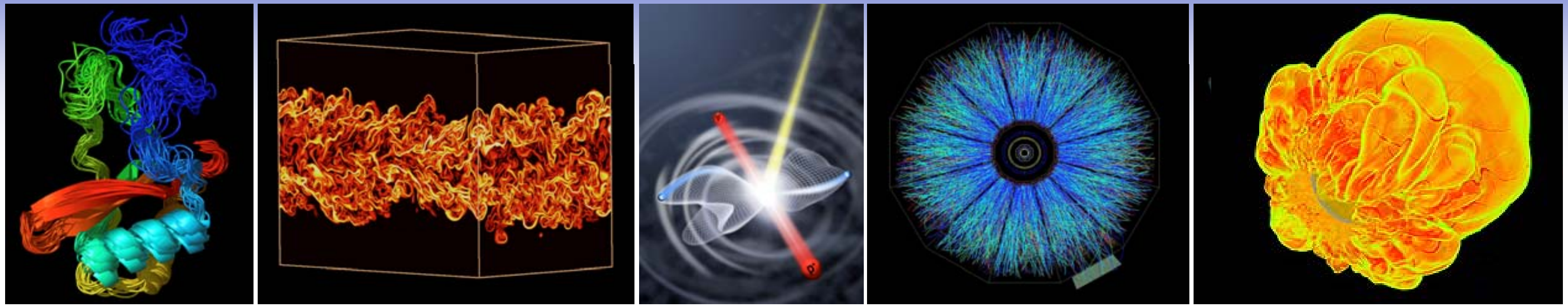


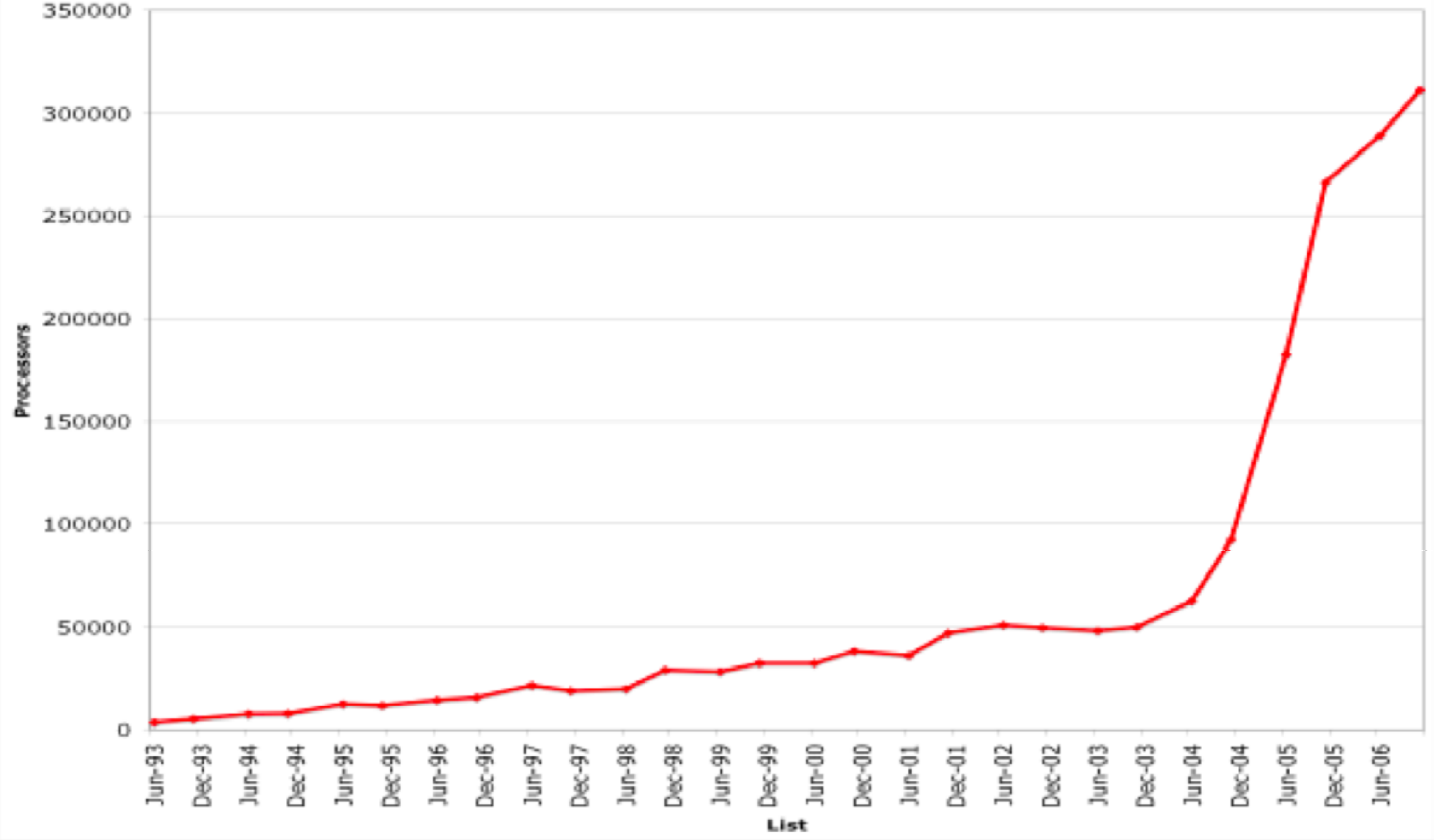
Multicore/Manycore: What Can We Expect from the Software?

Kathy Yelick

National Energy Research Supercomputing Center
Lawrence Berkeley National Laboratory and
EECS Department, University of California, Berkeley

This has Also Impacted HPC System Concurrency

Sum of the # of cores in top 15 systems (from top500.org)

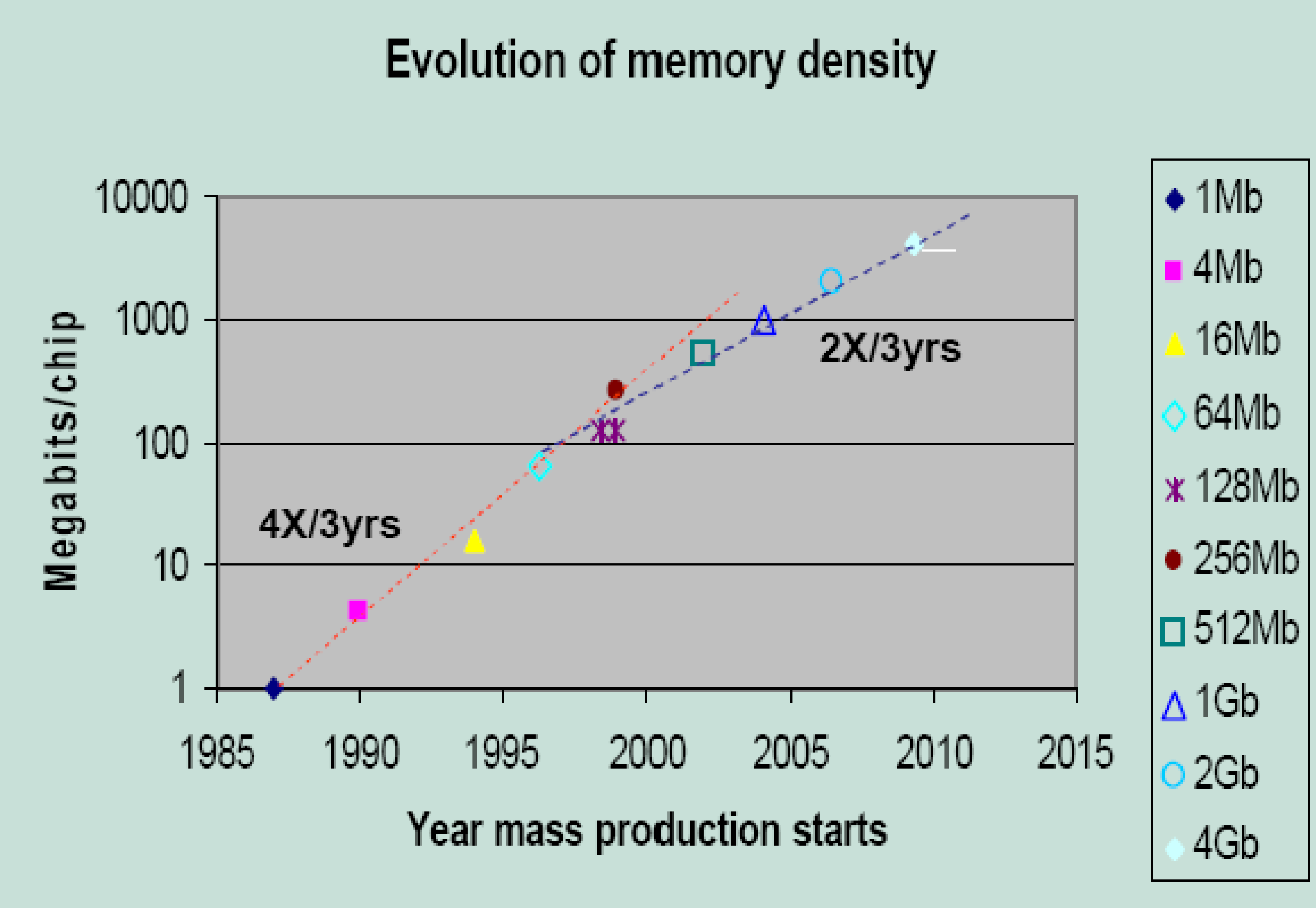


Exponential wave of increasing concurrency for forseeable future!

1M cores sooner than you think!

Berkeley ParLab

DRAM component density is *only* doubling every 3 years



Berkeley ParLab

Source: IBM

NERSC

BERKELEY LAB

Is MPI the Answer?

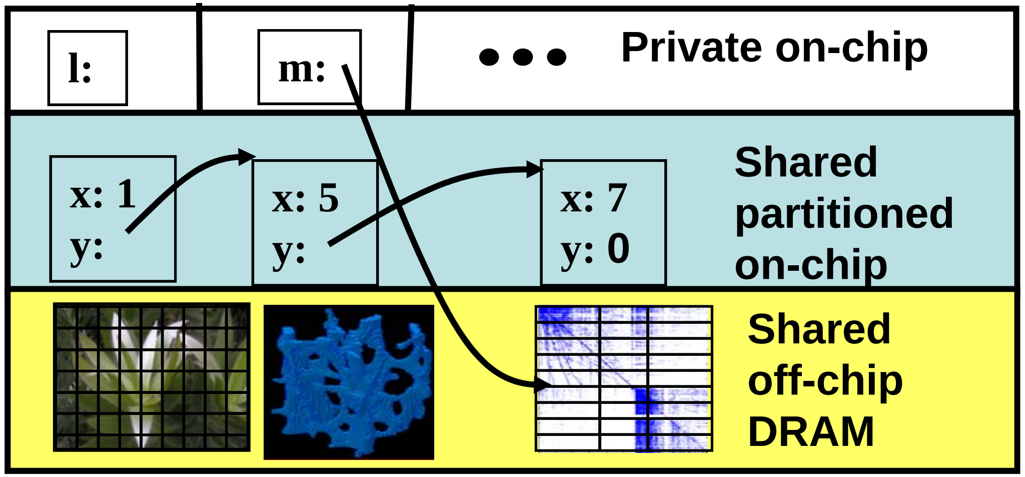
- We can run 1 MPI process per core
  - This works now (for CMPs) and will work for a while
- How long will it continue working?
  - 4 - 8 cores? Probably. 128 - 1024 cores? Probably not.
- What is the problem?
  - Latency: some copying required by semantics
  - Synchronization: notification required by semantics
  - Memory utilization: partitioning requires some replication
    - How big is your per core subgrid? At 10x10x10, over 1/2 of the points are surface points, probably replicated
  - Memory bandwidth: extra state means extra bandwidth
  - Weak scaling: success model for the “cluster era;” will not be for the many core era -- not enough memory per core
  - Heterogeneity: Is “core” really the right term or will these be a sea of functional units: MPI per CUDA thread-block?

What about Mixed MPI and Threads?

- Threads: OpenMP, PThreads, TBB, …
  - Will this work for 4-8 cores? Probably. More?
- What is the problem?
  - OpenMP leads programmers into Amdahl’s Law trap
    - Alternating serial and parallel code
    - Doesn’t encourage thinking in parallel (unlike MPI)
  - No direct control over locality
    - Memory affinity key on multiple sockets. Soon on-chip?
    - Can get this with extra help, static threads + pinning
  - Two programming models per application

~~PGAS~~ DMA Languages + Autotuning for Manycore/Multicore

- PGAS languages are a good fit to multicore
  - Global address space implemented as reads/writes
  - Also may be exploited for processor with explicit local store rather than cache, e.g., Cell, GPUs,…
- Open question in architecture
  - Hardware managed caches vs local stores (or hybrid)
  - Cache coherence shared memory vs. global addresssing
  - UPC demonstrate that the partitioned address space with DMA operations is useable (although not “high level”)



8 Things Software *Should* Do

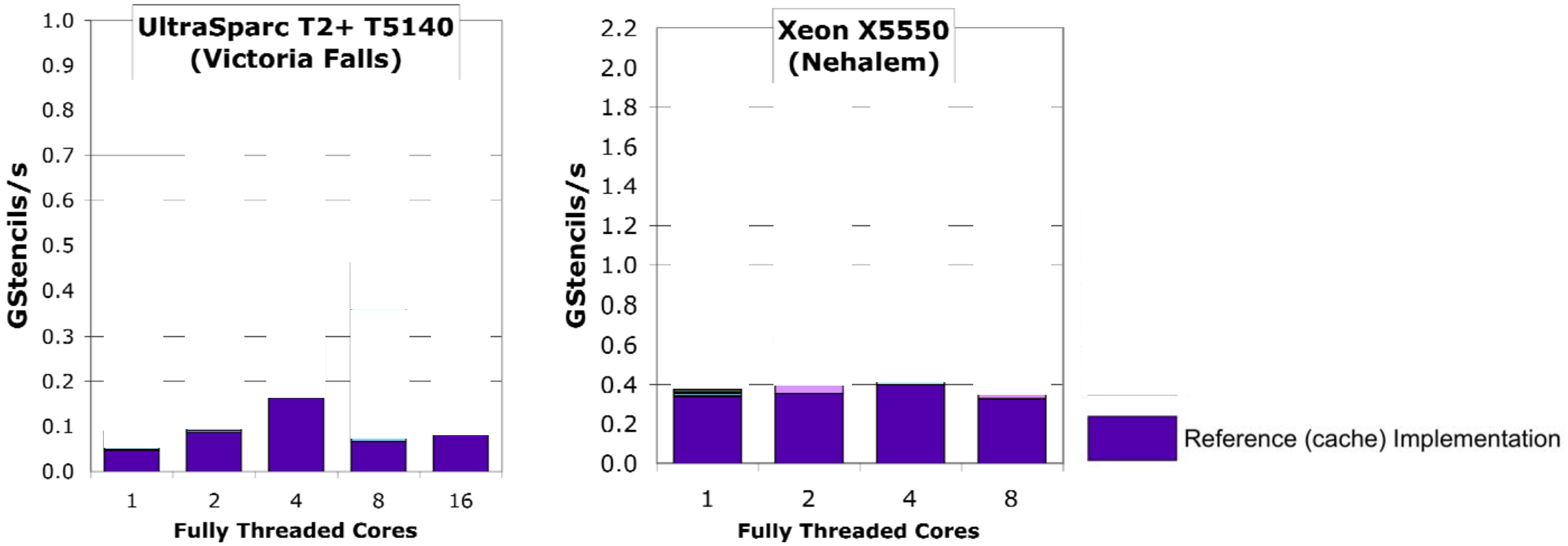
(And some encouraging evidence that it can)

#1) Software Needs to Avoid Unnecessary Bandwidth Use

Nearest-neighbor 7point stencil on a 3D array

Use Autotuning!

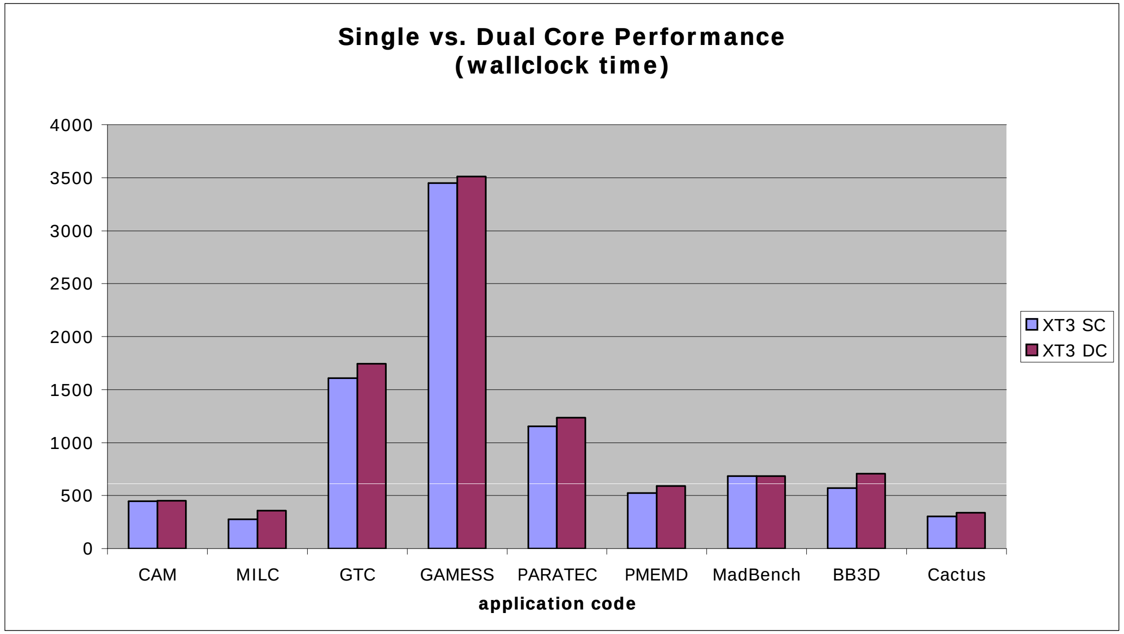
Write code generators and let computers do tuning



#2) Software Needs to Address Little’s Law

Little’s Law: required concurrency = bandwidth * latency

#outstanding_memory_fetches = bandwidth* latency



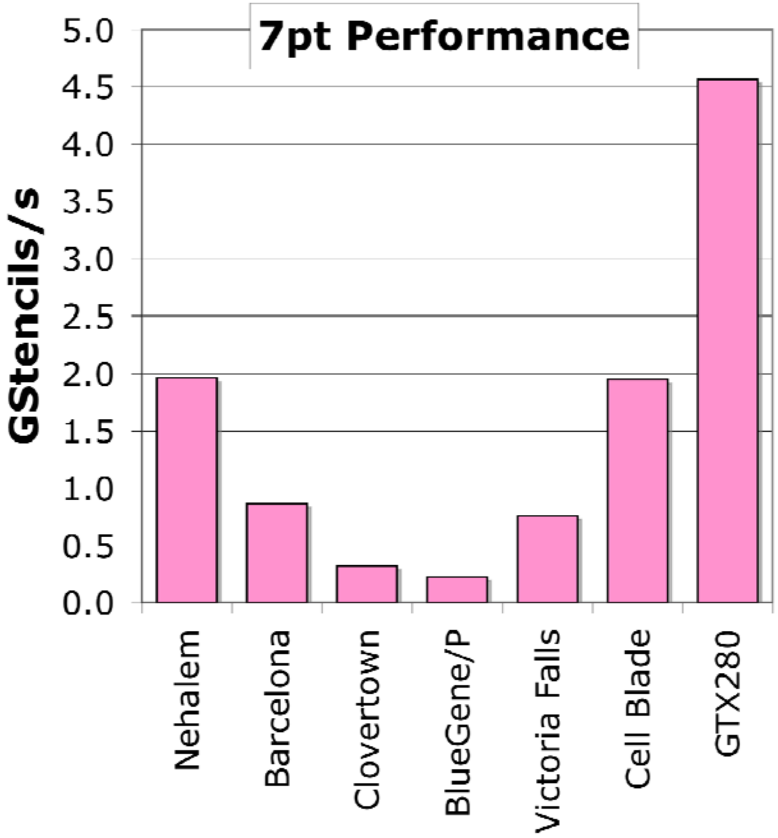
NERSC application benchmarks
Shalf et al

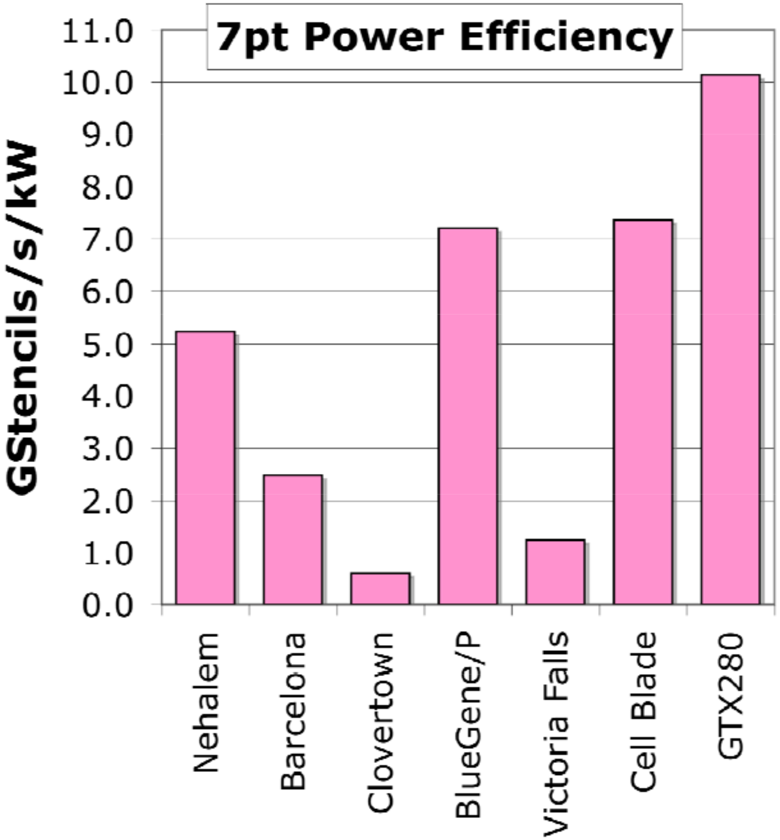
Experiment: Running on a fixed number of cores

1 core per socket vs 2 cores per socket

Only 10% performance drop from sharing (halving) bandwidth

7 Point Stencil Revisited

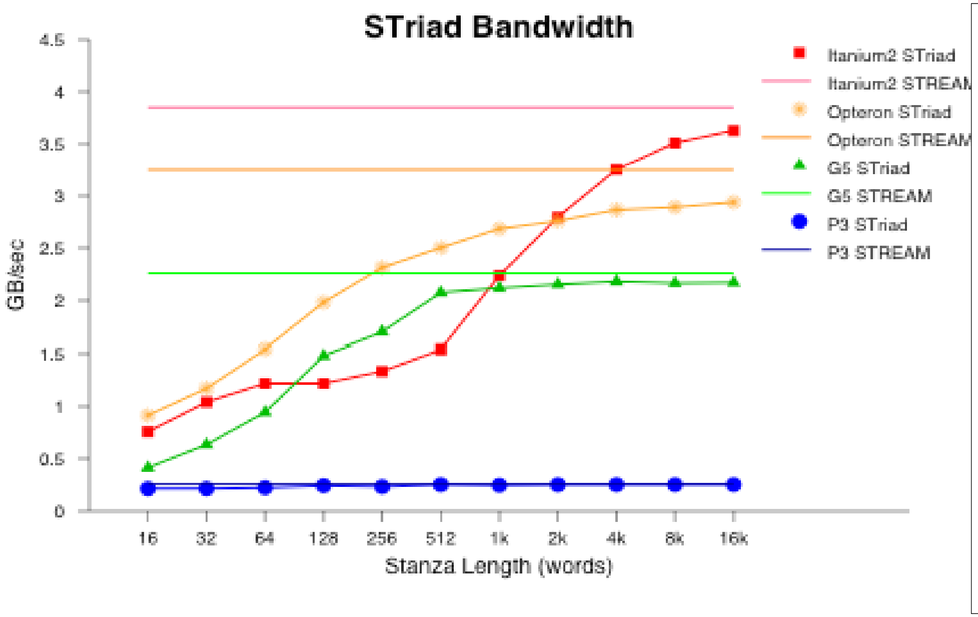


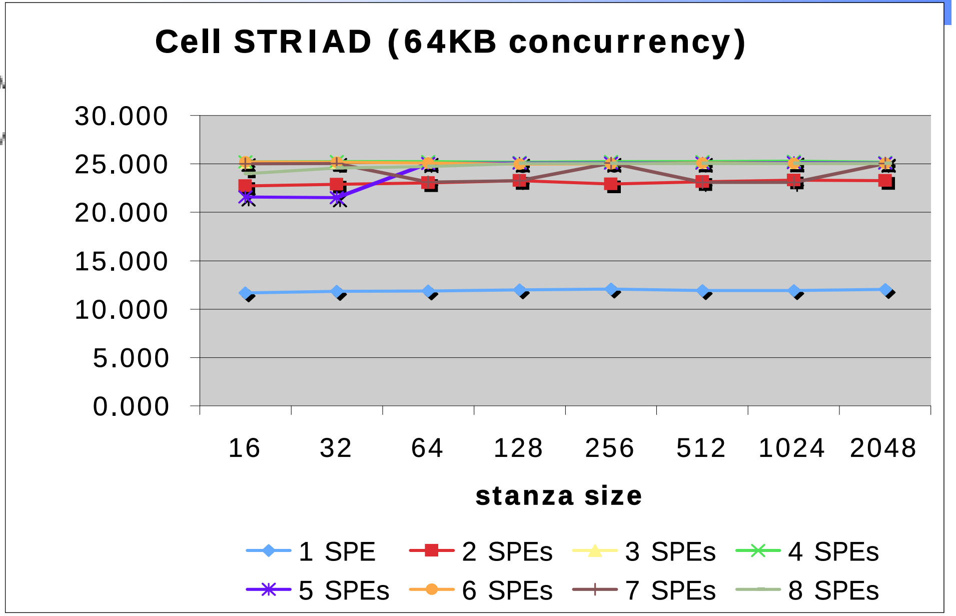


- Cell and GTX280 are notable for both performance and energy efficiency

Joint work with Kaushik Datta, Jonathan Carter, Shoaib Kamil, Lenny Oliker, John Shalf, and Sam Williams

Why is the STI Cell So Efficient?



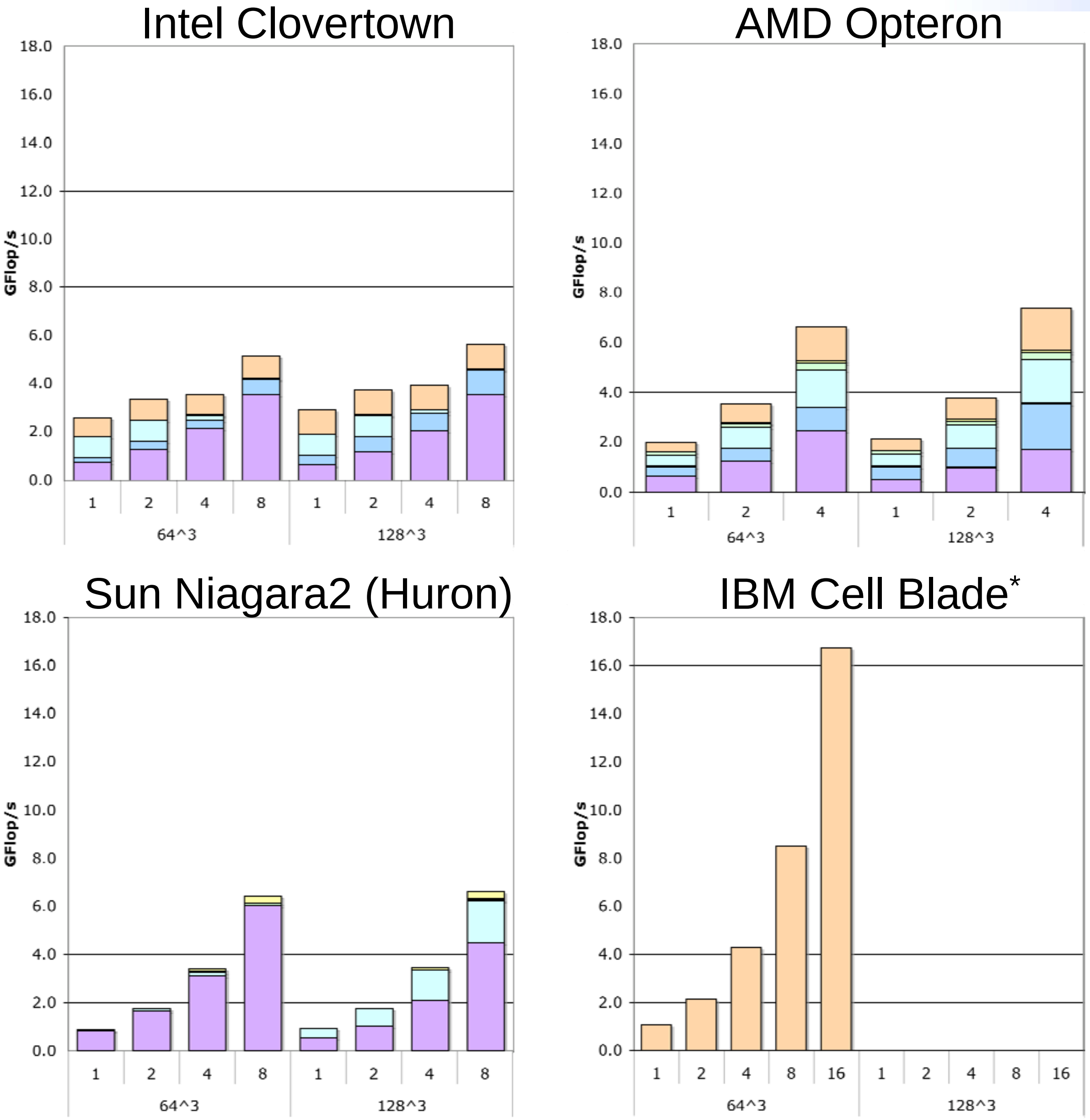


- Unit stride access is as important as cache utilization on processors that rely on hardware prefetch
  - Tiling in unit stride direction is counter-productive: improves reuse, but kills prefetch effectiveness
- Software controlled memory gives programmers more control
  - Spend bandwidth on what you use; bulk moves (DMA) hide latency

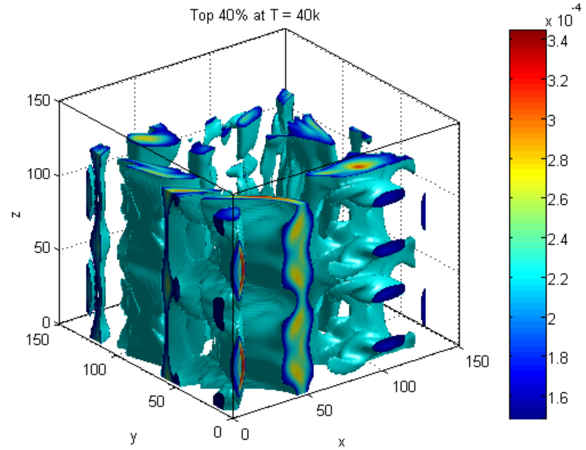
Berkeley ParLab

Joint work with Shoaib Kamil, Lenny Oliker, John Shalf, Kaushik Datta

#3) Use Novel Hardware Features Through Code Generators (Autotuning)



LBMHD is not always bandwidth limited: used SIMD, etc.

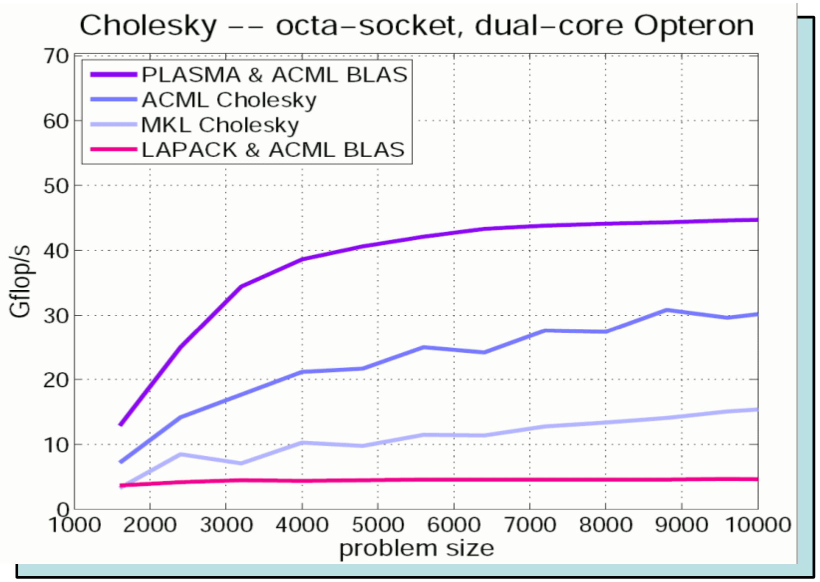


- 🟧 +SIMDization
- 🟨 +SW Prefetching
- 🟩 +Unrolling
- ▢ +Vectorization
- 🟦 +Padding
- 🟪 Naïve+NUMA

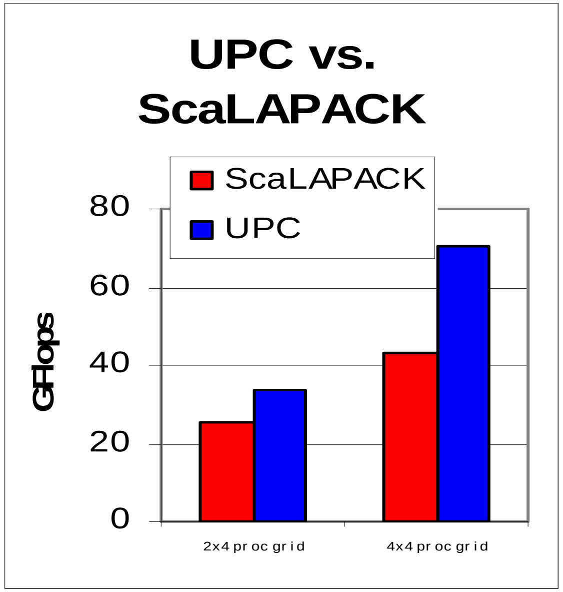
Joint work with Sam Williams, Lenny Oliker, John Shalf, and Jonathan Carter

#4) Software Should Avoid Unnecessary Global Synchronization

PLASMA on shared memory



UPC on partitioned memory

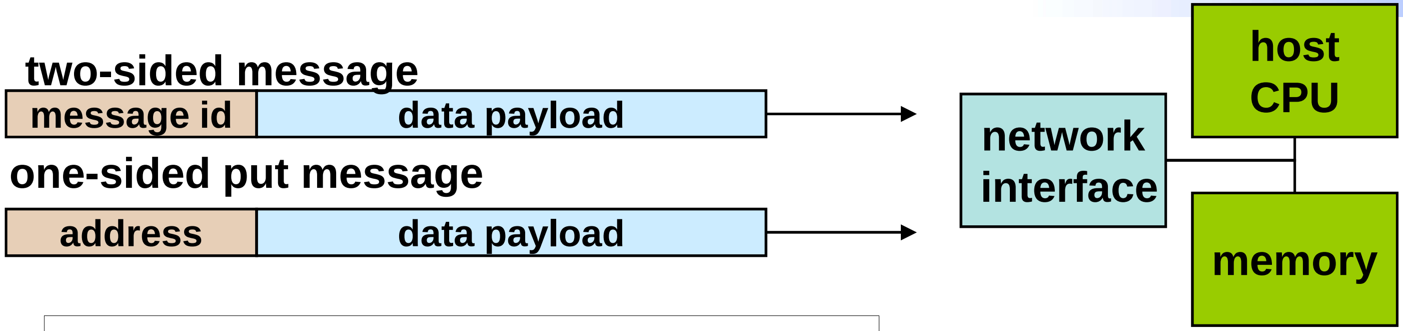


- UPC LU factorization code adds cooperative (non-preemptive) threads for latency hiding
  - New problem in partitioned memory: allocator deadlock
  - Can run on of memory locally due tounlucky execution order

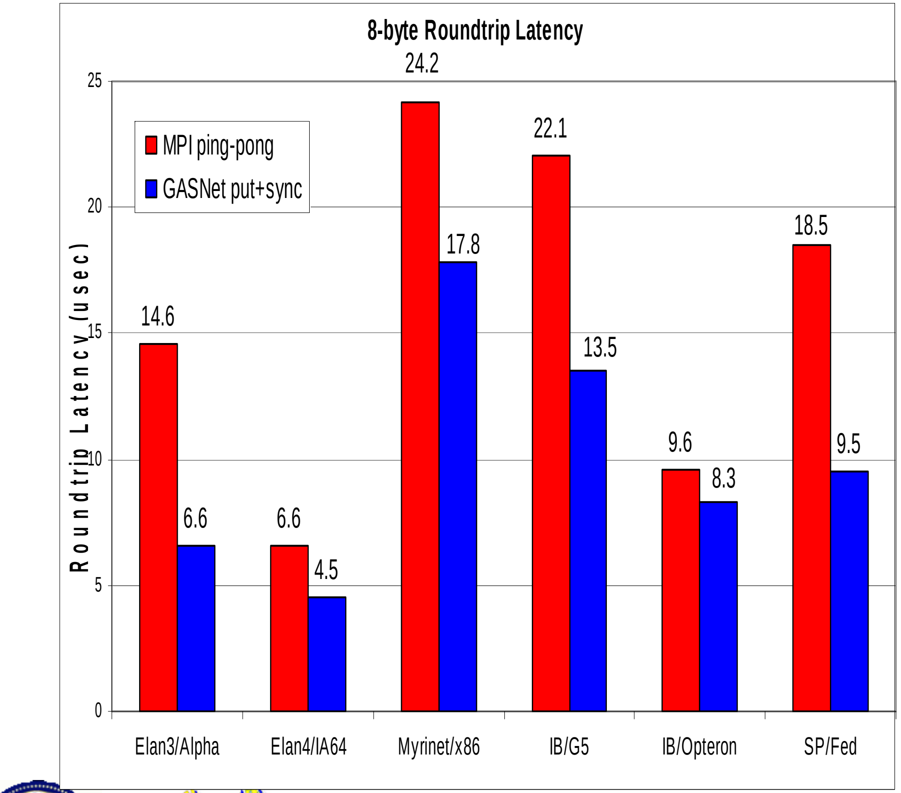
PLASMA by Dongarra et al; UPC LU joint with Parray Husbands

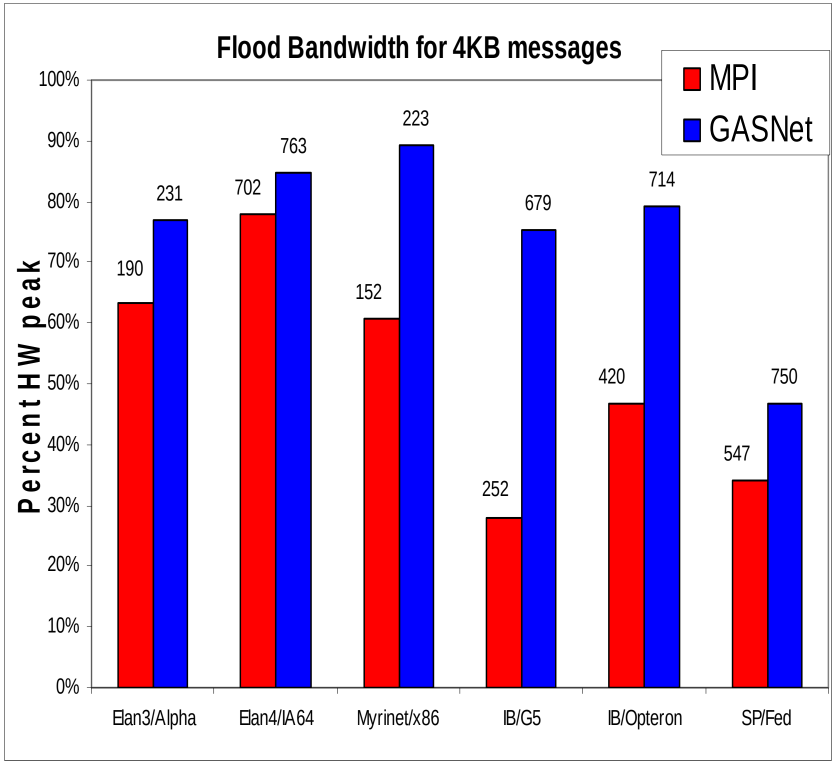
Berkeley ParLab

#5) Software Should Avoid Unnecessary Point-to-Point Communication



Pay only for what you need

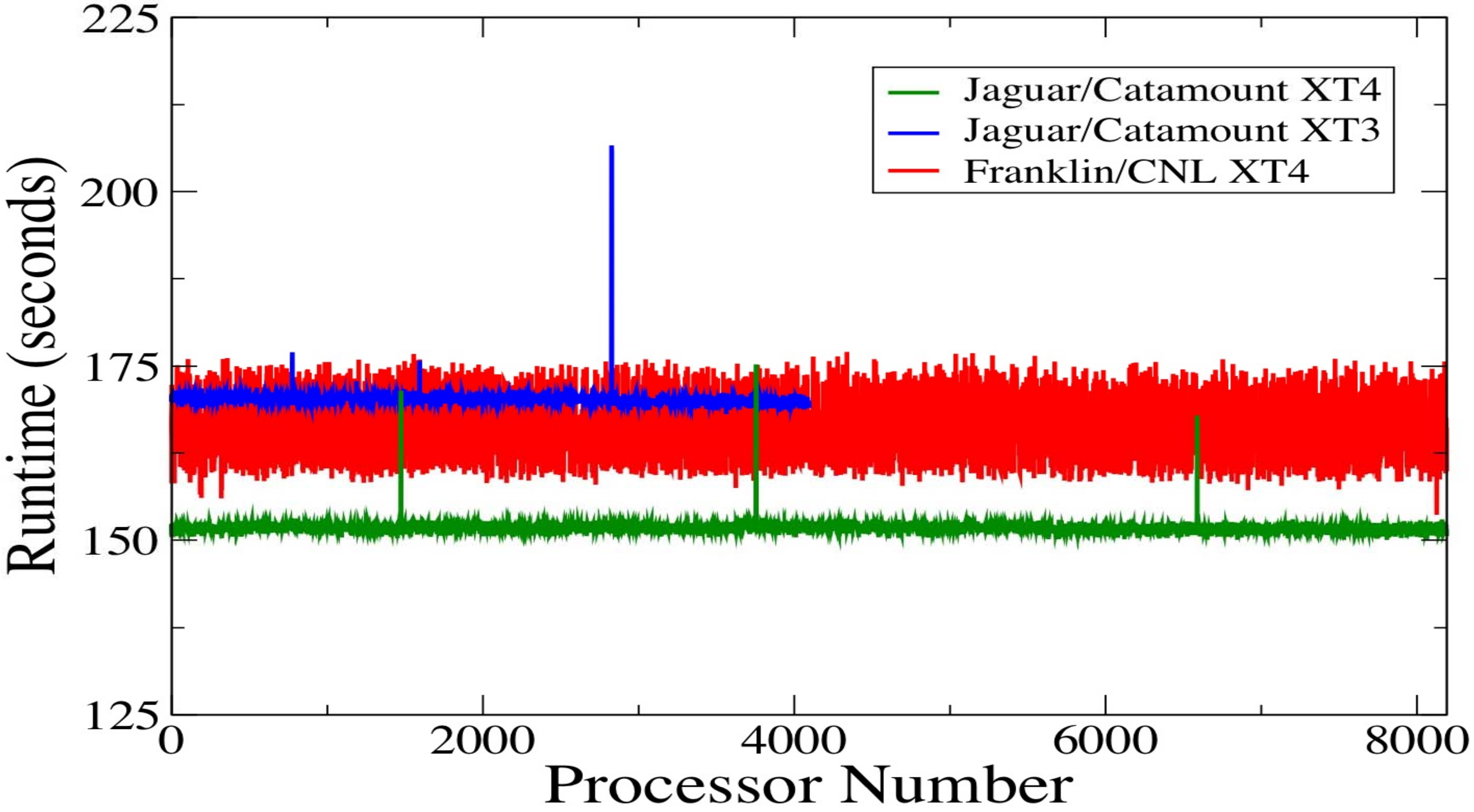




Joint work with Berkeley UPC Group

#6) Alas, Software Needs to Deal with Faults

- **Fault resilience introduces inhomogeneity in execution rates *(error correction is not instantaneous)***

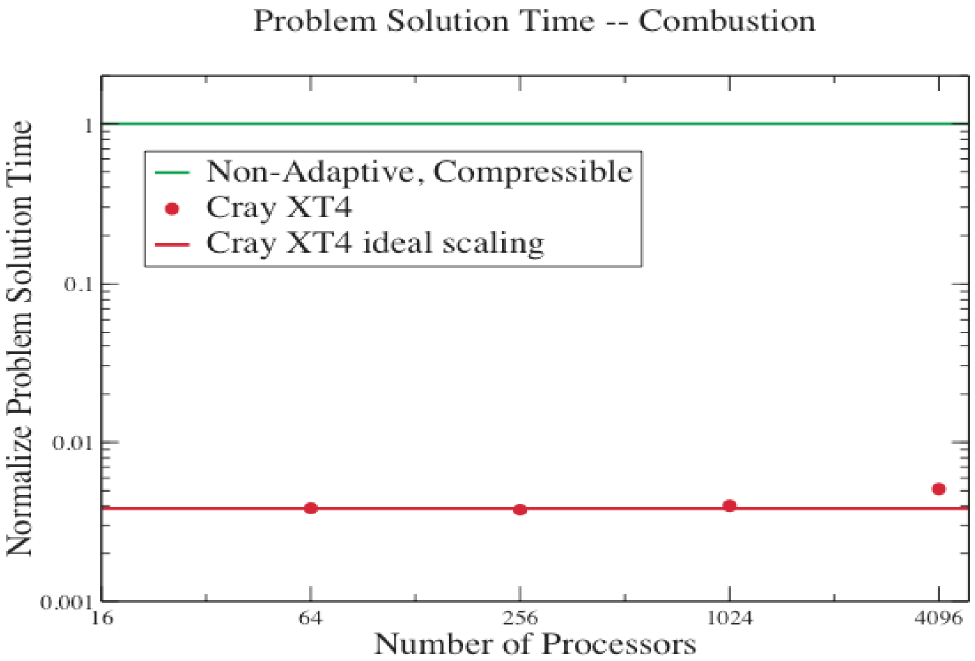


Berkeley ParLab

Slide source: John Shalf

#7) Software should make use of Good Algorithms

- Algorithmic gains in last decade have far outstripped Moore's Law
  - –Adaptive meshes rather than uniform
  - –Sparse matrices rather than dense
  - –Reformulation of problem back to basics



- Example of canonical "Poisson" problem on n points:
  - –Dense LU: most general, but $O(n^3)$ flops on $O(n^2)$ data
  - –Multigrid: fastest/smallest, $O(n)$ flops on $O(n)$ data

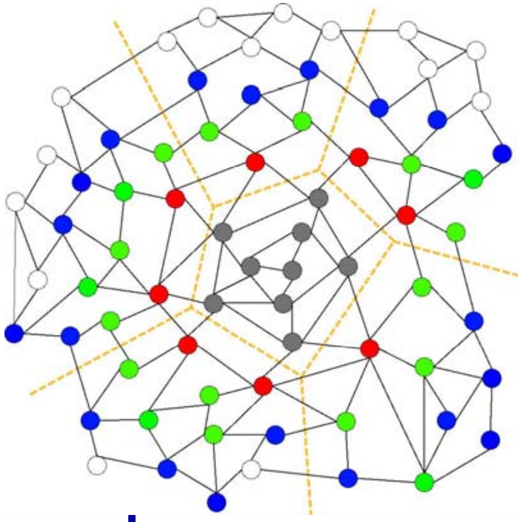
Berkeley ParLab

Performance results: John Bell et al

#8) Algorithm Developers should Avoid Communication, not Flops

- **Consider Sparse Iterative Methods**
  - **Nearest neighbor communication on a mesh**
  - **Dominated by time to read matrix (edges) from DRAM**
  - **And (small) communication and global synchronization events at each step**

  – **Can we lower data movement costs?**



- **Take *k* steps "at once" with one matrix read from DRAM and one communication phase**

  – **Parallel implementation**

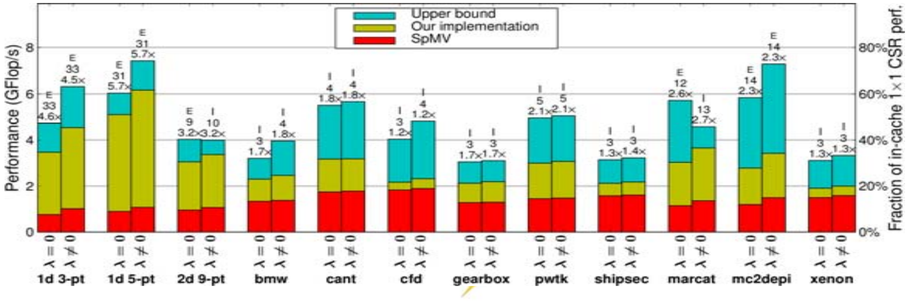
  - **O(log p) messages vs. O(k log p)**

  – **Serial implementation**

  - **O(1) moves of data moves vs. O(k)**

- **Performance of A^kx operation relative to Ax and upper boun**

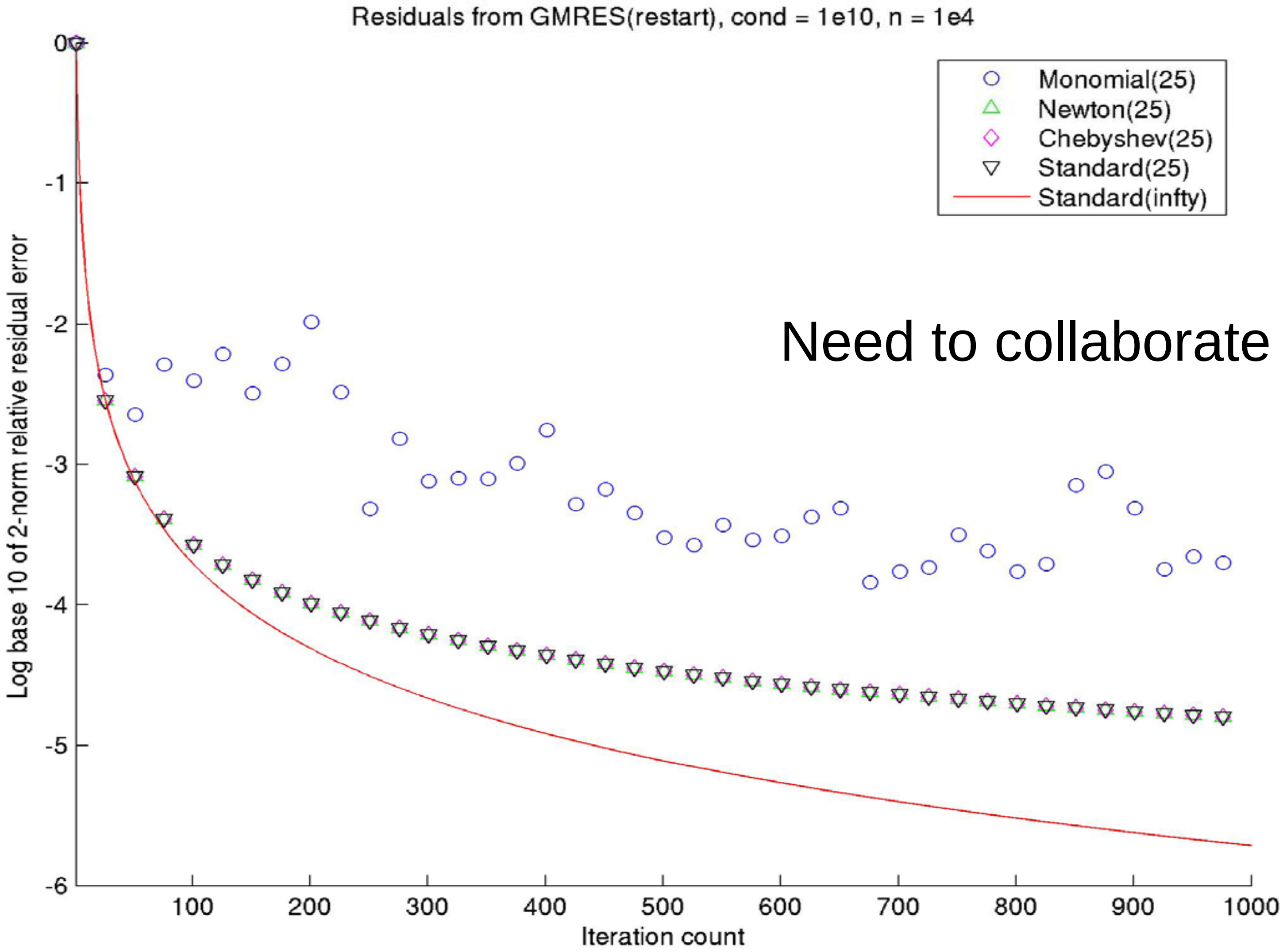
  – **Runs up to 5x faster on SMP**



Joint work with Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin

Berkeley ParLab

But the Numerics have to Change!



Need to collaborate

Work by Jim Demmel and Mark Hoemman

Berkeley ParLab

8 Rules for Software (and Algorithms and Applications)

1) Don’t waste memory bandwidth
2) Remember Little's Law
3) Use novel hardware features
4) Avoid global synchronization
5) Avoid point-to-point synchronization
6) Deal with faults throughout software
7) Choose efficient algorithms
8) Rethink algorithms to avoid data movement

Conclusions

- Enable programmers to get performance
  - – Expose features for performance
  - – Don’t hide them
- Go Green
  - – Enable energy-efficient computers and software
- Work with experts on software, algorithms, applications

Software Issues at Scale

- Power concerns will dominates all others;
  - Concurrency is the most significant knob we have: lower clock, increase parallelism
  - Power density and facility energy
- Summary Issues for Software
  - 1EF system: Billion-way concurrency, O(1K) cores per chip
  - 1 PF system: millions of threads and O(1K) cores per chip
  - The memory capacity/core ratio may drop significantly
  - Faults will become more prevalent
  - Flops are cheap relative to data movement

How to Waste an Exascale Machine

- Ignore Little’s Law (waste bandwidth)
- Over-synchronize (unnecessary barriers)
- Over-synchronize communication (two-sided vs. one-sided)
- Waste bandwidth: ignore locality
- Use algorithms that minimize flops rather than data movement
- Add a high-overhead runtime system when you don’t need it

To Virtualize or Not

- The fundamental question facing in parallel programming models is:

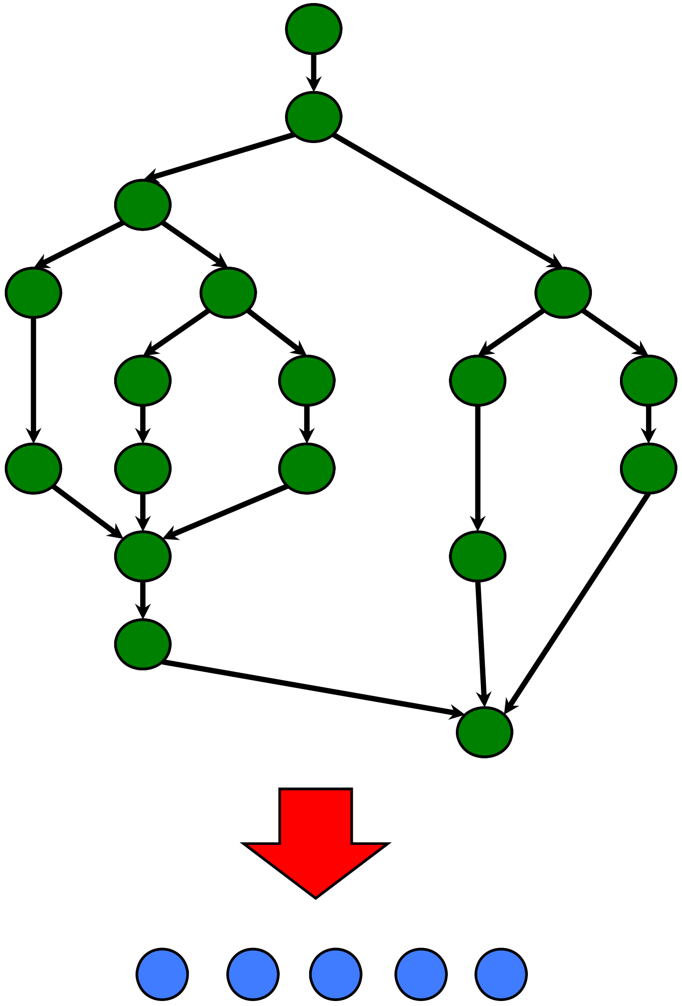
  What should be virtualized?

- 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?

Reasons to Virtualize

- Simplicity for Programmer
- Potential to hide problems:
  - load imbalance in hardware, e.g., jitter
  - faults
  - wierd memory structures (local stores)
- Effective use of system resources
  - in a space-shared environment
  - multiple jobs sharing resources

Virtualization of Processors



- **A parallel computation is defined by its task graph**
- **Many possible graphs, depending on how much parallelism is exposed**
- **Where does the mapping of the graph to a particular number of processors happen?**
  - **The compiler: auto parallelization, NESL, ZPL**
  - **The runtime system : Cilk, Charm++ (sometimes), OpenMP, X10**
  - **The programmer: MPI, UPC**

Irregular vs. Regular Parallelism

- Computations with known task graphs can be mapped to resources in an offline manner (before computation starts)
  - Regular graph: By a compiler (static) or runtime (semi-static)
  - Irregular graphs: By a DAG scheduler
  - No need for online scheduling
- If graphs are not known ahead of time (structure, task costs, communication costs), then dynamic scheduling is needed
  - Task stealing / task sharing
  - Demonstrated on shared memory
- Conclusion: If your task graph is dynamic, the runtime needs to be, but what if it static?

Load Balancing with Locality

- Locality is important:
  - When memory hierarchies are deep
  - When computational intensity is low (expensive move cost cannot be amortized)
- Most (all?) successful examples of locality-important applications/machines use static scheduling
  - Unless they have a irregular/dynamic task graph so it is impossible
- Two extremes are well-studied
  - Dynamic parallelism without locality
  - Static parallelism (with threads = processors) with locality
- Dynamic scheduling (task stealing) with locality control can cause problems
  - Locality control can cause non-optimal task schedule, which can blow up memory use (breadth vs. depth first traversal)
  - Can run out of memory locally when you don’t globally

New World Order

- **Goal: performance through parallelism**
- **Power is overriding hardware concern:**
  - **Power density limits clock speed**
  - **Handheld devices limited by battery life**
  - **HPC systems may be >100 MW in 10 years**
- **Performance is now a software concern**
  - **Not just in HPC**
- ***How can we lose performance and therefore lose the case for parallelism?***