Compilation Techniques for Partitioned Global Address Space Languages

Katherine Yelick

U.C. Berkeley and Lawrence Berkeley National Lab

http://titanium.cs.berkeley.edu
http://upc.lbl.gov

HPC Programming: Where are We?

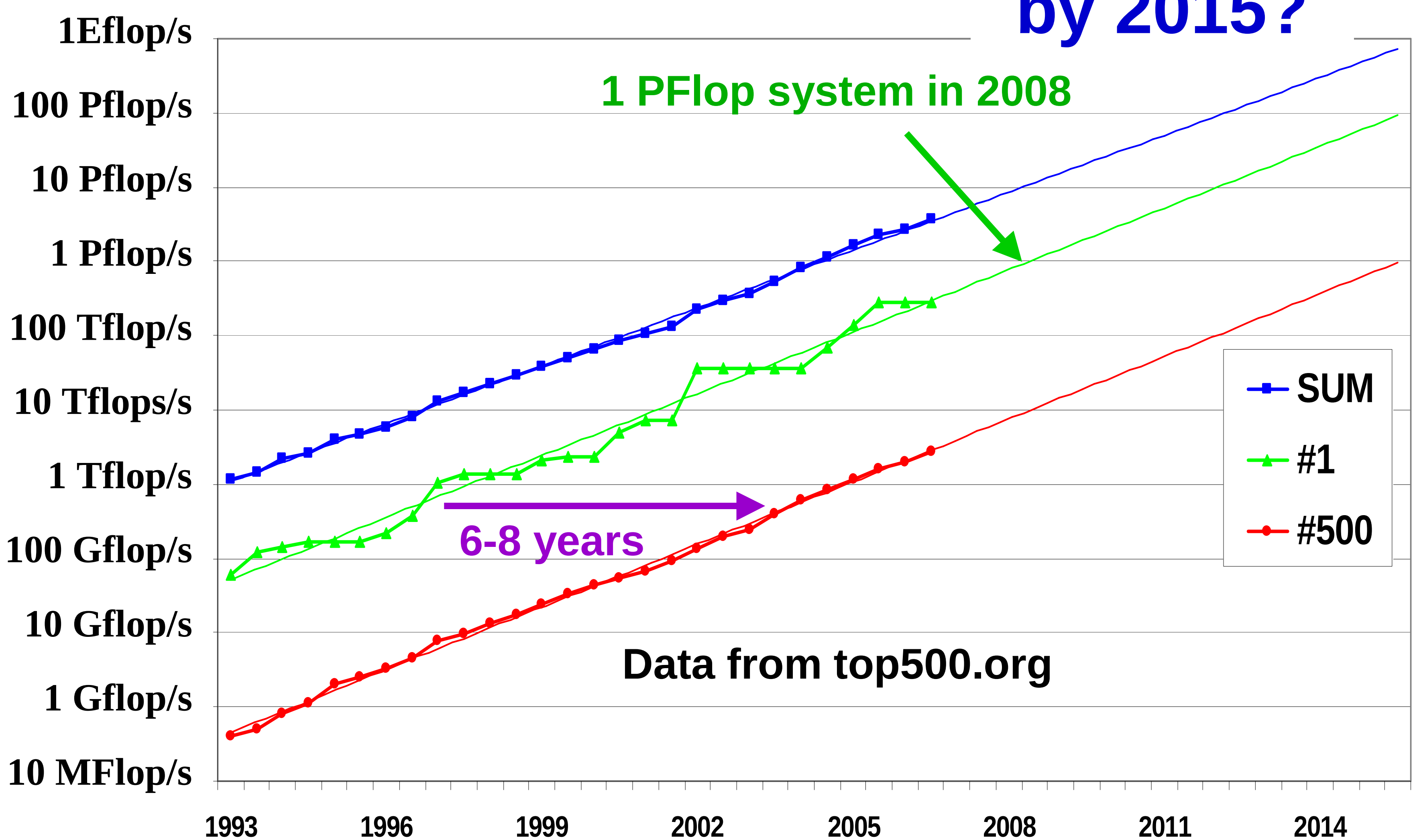
- IBM SP at NERSC/LBNL has as 6K processors
  - There were 6K transistors in the Intel 8080a implementation
- BG/L at LLNL has 64K processor cores
  - There were 68K transistors in the MC68000
- A BG/Q system with 1.5M processors may have more processors than there are logic gates per processor

- HPC Applications developers today write programs that are as complex as describing where every single bit must move between the 6,000 transistors of the 8080a
- We need to *at least* get to "assembly language" level

Slide source: Horst Simon and John Shalf, LBNL/NERSC

Petaflop with ~1M Cores

Common by 2015?



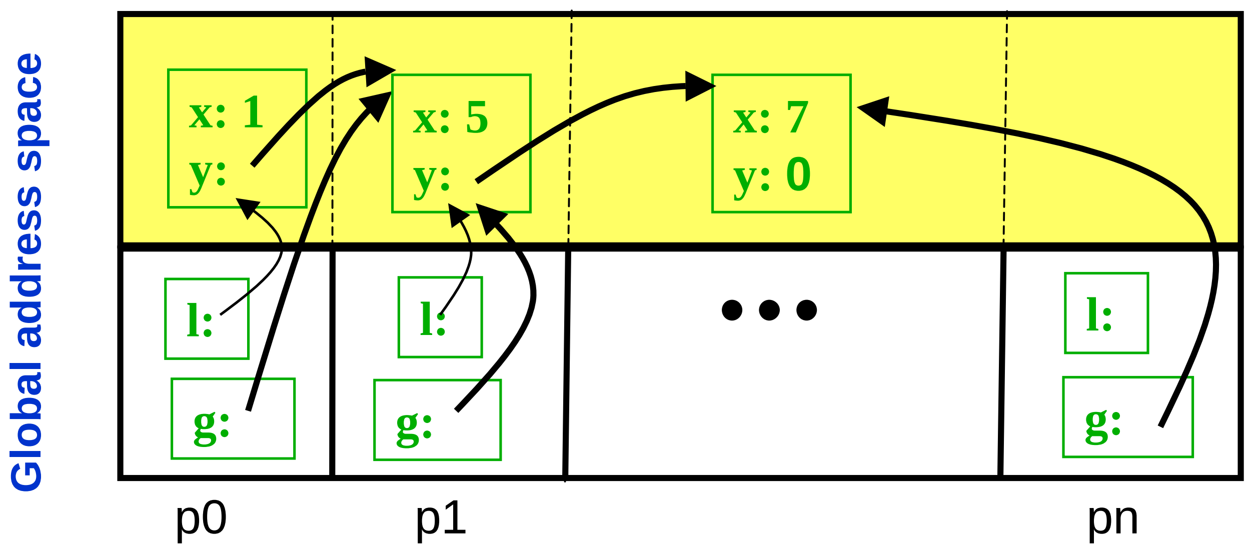
Predictions

- **Parallelism will explode**
  - Number of cores will double every 12-24 months
  - 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 key will be concerns for many programmers – not just an HPC problem
- **A new programming model will emerge for multicore programming**
  - Can one language cover laptop to top500 space?

PGAS Languages: What, Why, and How

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**

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

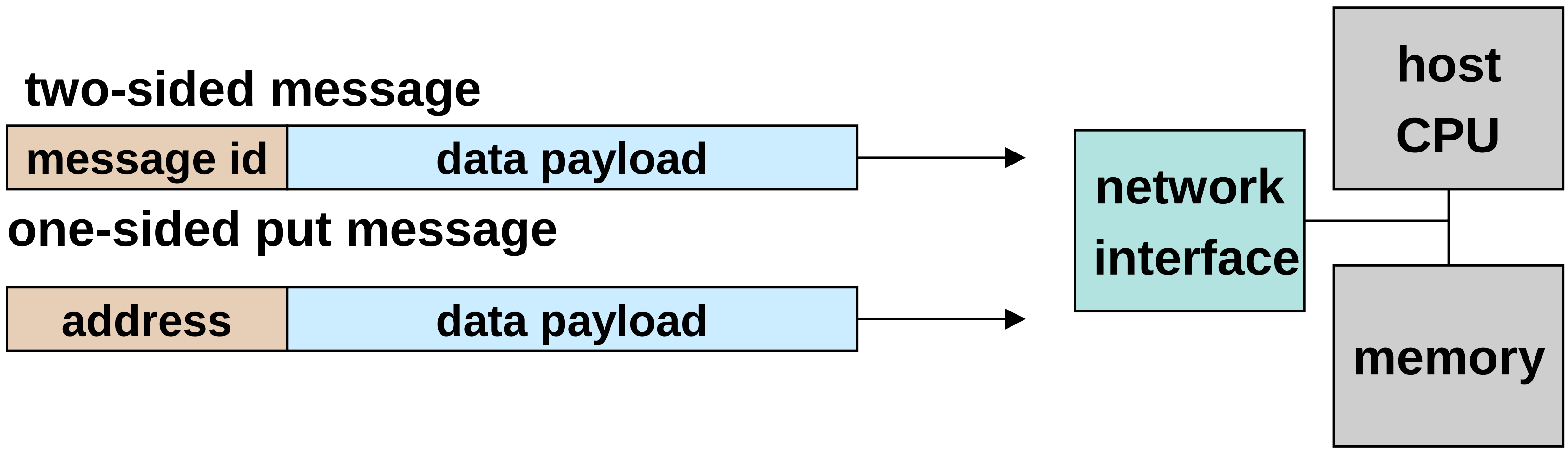
PGAS Language Overview

- **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 Language for Multicore

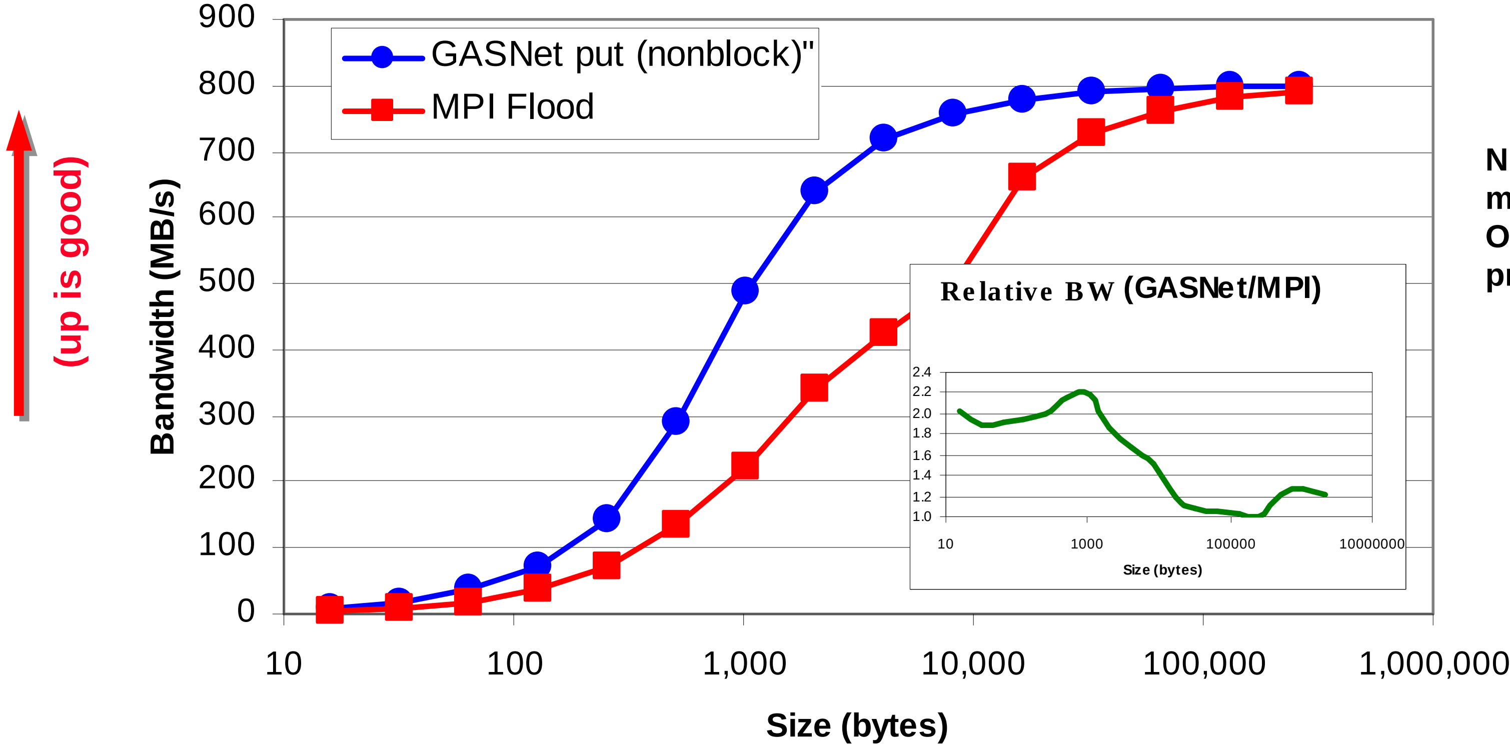
- **PGAS languages are a good fit to shared memory machines**
  - Global address space implemented as reads/writes
  - Current UPC and Titanium implementation uses threads
  - Working on System V shared memory for UPC
- **“Competition” on shared memory is OpenMP**
  - PGAS has locality information that may be important when we get to >100 cores per chip
  - Also may be exploited for processor with explicit local store rather than cache, e.g., Cell processor
  - SPMD model in current PGAS languages is both an advantage (for performance) and constraining

PGAS Languages on Clusters: 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)

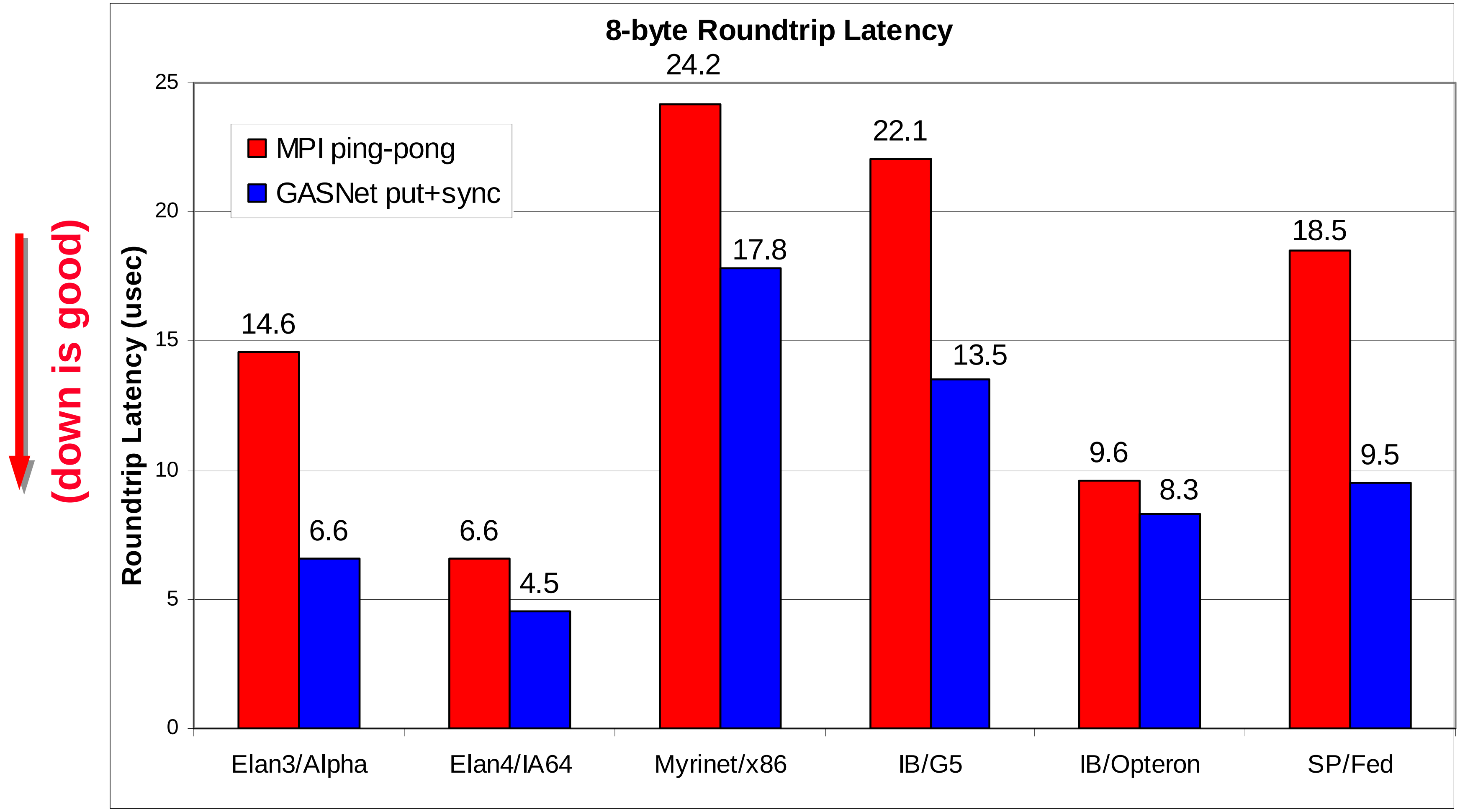
One-Sided vs. Two-Sided: Practice



NERSC Jacquard machine with Opteron processors

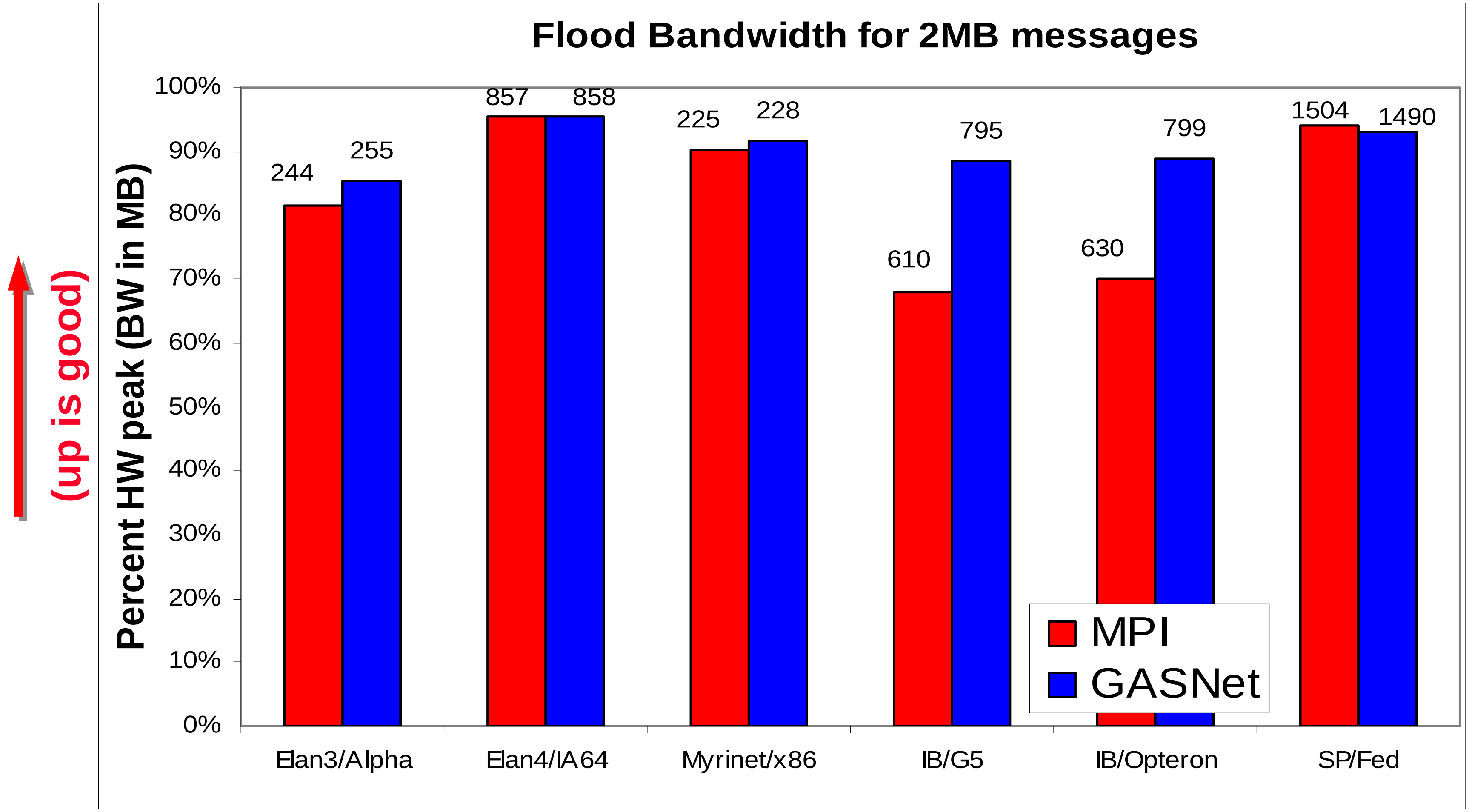
- **InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5**
- **Half power point (N ½) differs by *one order of magnitude***
- **This is not a criticism of the implementation!**

GASNet: Portability and *High-Performance*



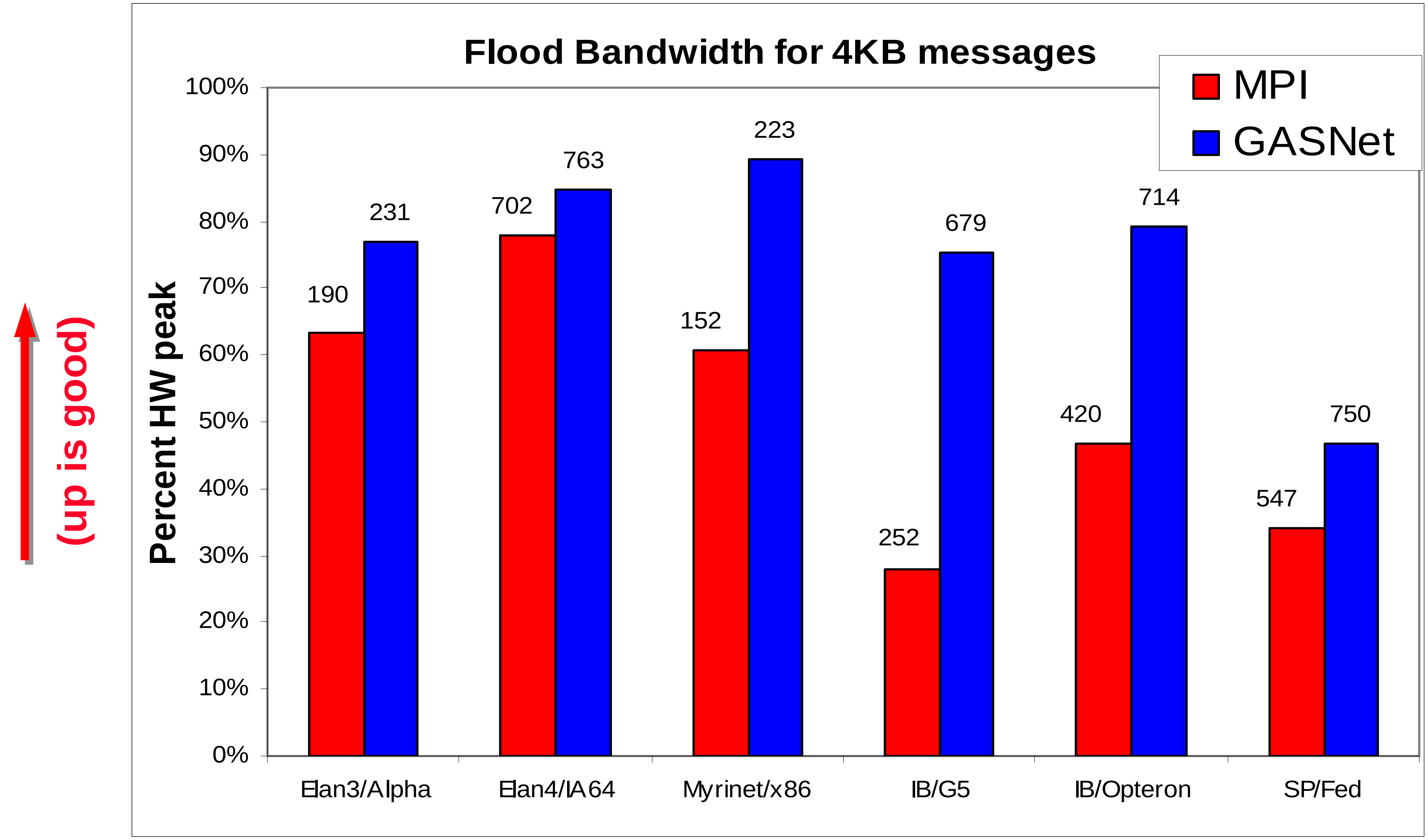
GASNet better for latency across machines

GASNet: Portability and *High-Performance*



GASNet at least as high (comparable) for large messages

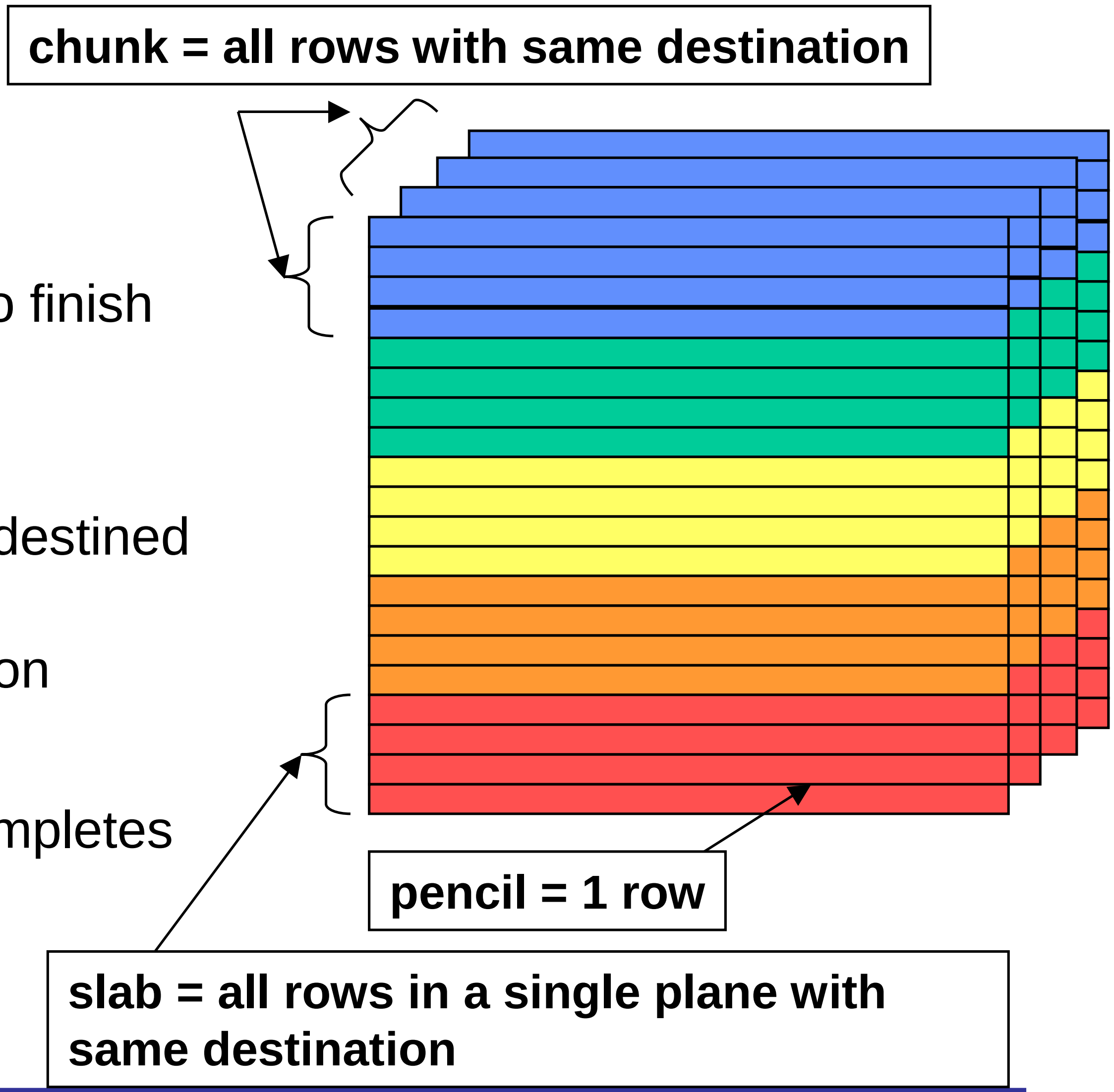
GASNet: Portability and *High-Performance*



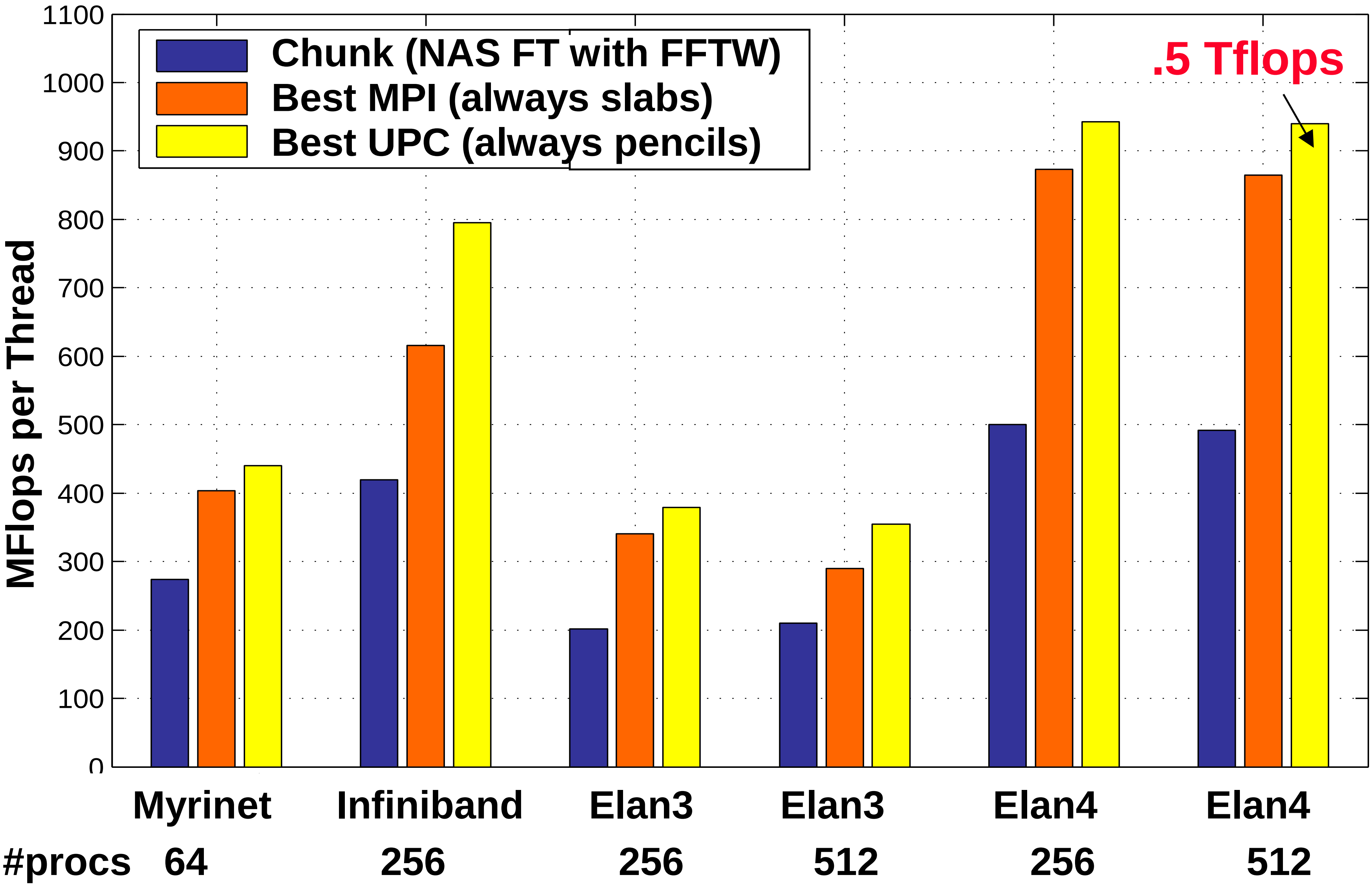
GASNet excels at mid-range sizes: important for overlap

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



NAS FT Variants Performance Summary



Top Ten PGAS Problems

1. **Pointer localization**
2. **Automatic aggregation of communication**
3. **Synchronization strength reduction**
4. **Automatic overlap of communication**
5. **Collective communication scheduling**
6. **Data race detection**
7. **Deadlock detection**
8. **Memory consistency**
9. **Global view → local view**
10. **Mixed Task and Data Parallelism**

optimization

analysis

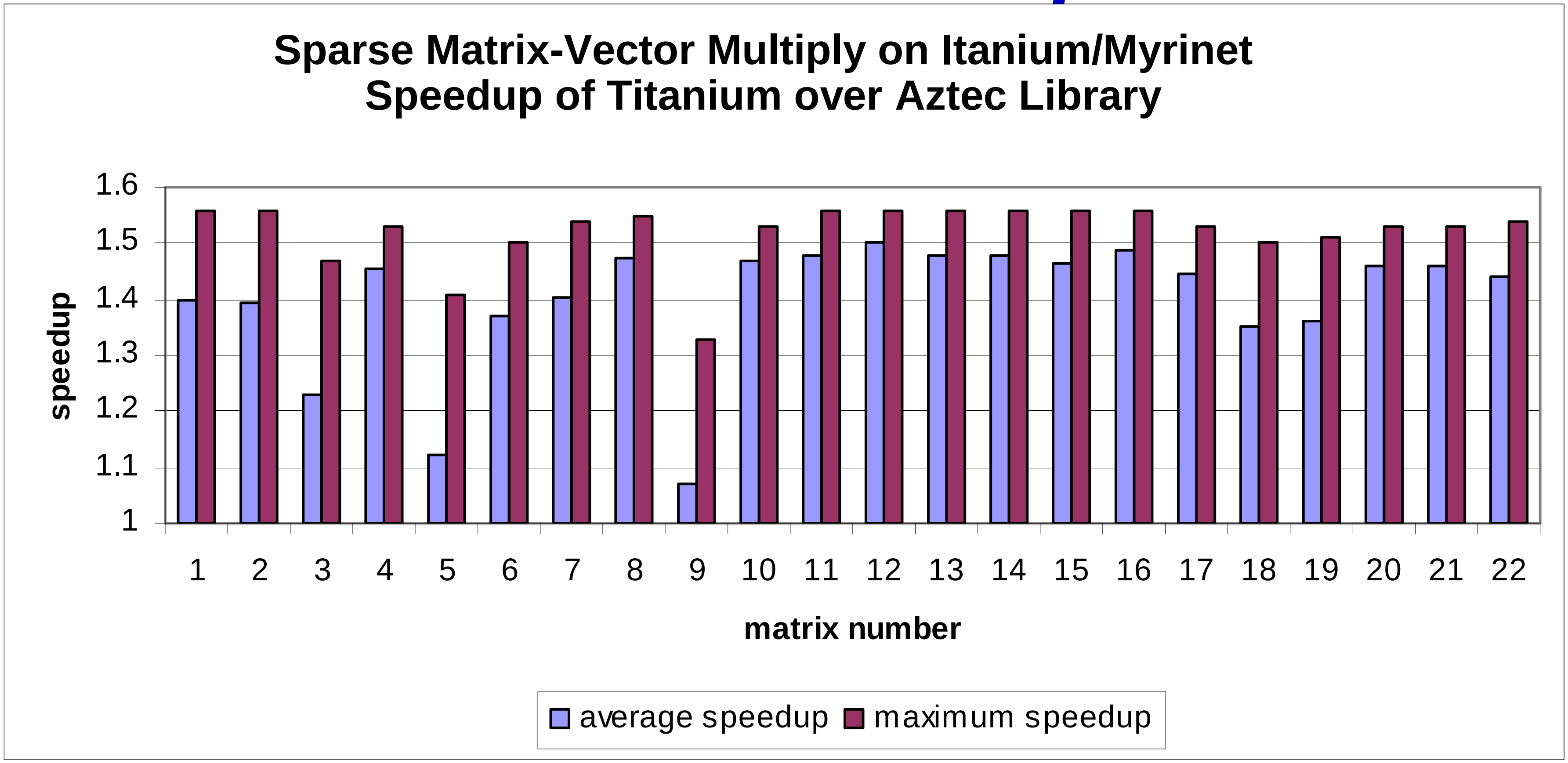
language

Optimizations in Titanium

- **Communication optimizations are done**
- **Analysis in Titanium is easier than in UPC:**
  - Strong typing helps with alias analysis
  - Single analysis identifies global execution points that all threads will reach “together” (in same synch phase)
    - I.e., a barrier would be legal here
- **Allows global optimizations**
  - Convert remote reads to remote writes by other side
  - Perform global runtime analysis (inspector-executor)
  - Especially useful for sparse matrix code with indirection:

```
y [i] = … a[b[i]]
```

Global Communication Optimizations



- Titanium code is written with fine-grained remote accesses
- Compile identifies legal “inspector” points
- Runtime selects (pack, bounding box) per machine / matrix / thread pair

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 in Titanium

- Relies on Titanium's *textual barriers* and *single-valued* expressions
- Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers (this is illegal)

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  Ti.barrier(); // odd ID threads
```

- *Single-valued* expressions used to enforce textual barriers while permitting useful programs

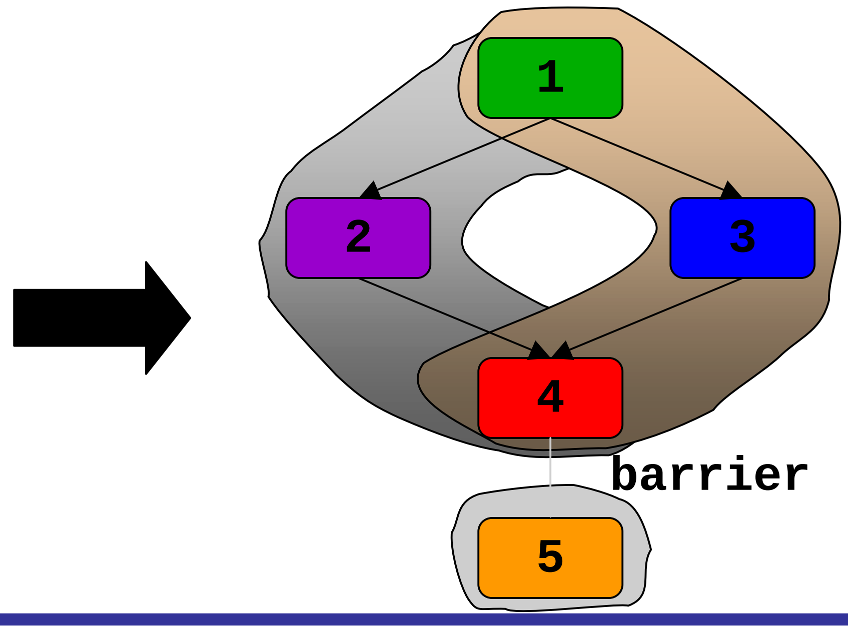
```
single boolean allGo = broadcast go from 0;
if (allGo) Ti.barrier();
```

- May also be used in loops to ensure same number of iterations

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)

Benchmarks

Benchmark	Lines[1]	Description
`pi`	56	Monte Carlo integration
`demv`	122	Dense matrix-vector multiply
`sample-sort`	321	Parallel sort
`lu-fact`	420	Dense linear algebra
`3d-fft`	614	Fourier transform
`gsrb`	1090	Computational fluid dynamics kernel
`spmv`	1493	Sparse matrix-vector multiply
`amr-gas`	8841	Hyperbolic AMR solver for gas dynamics
`amr-poisson`	4700	AMR Poisson (elliptic) solver

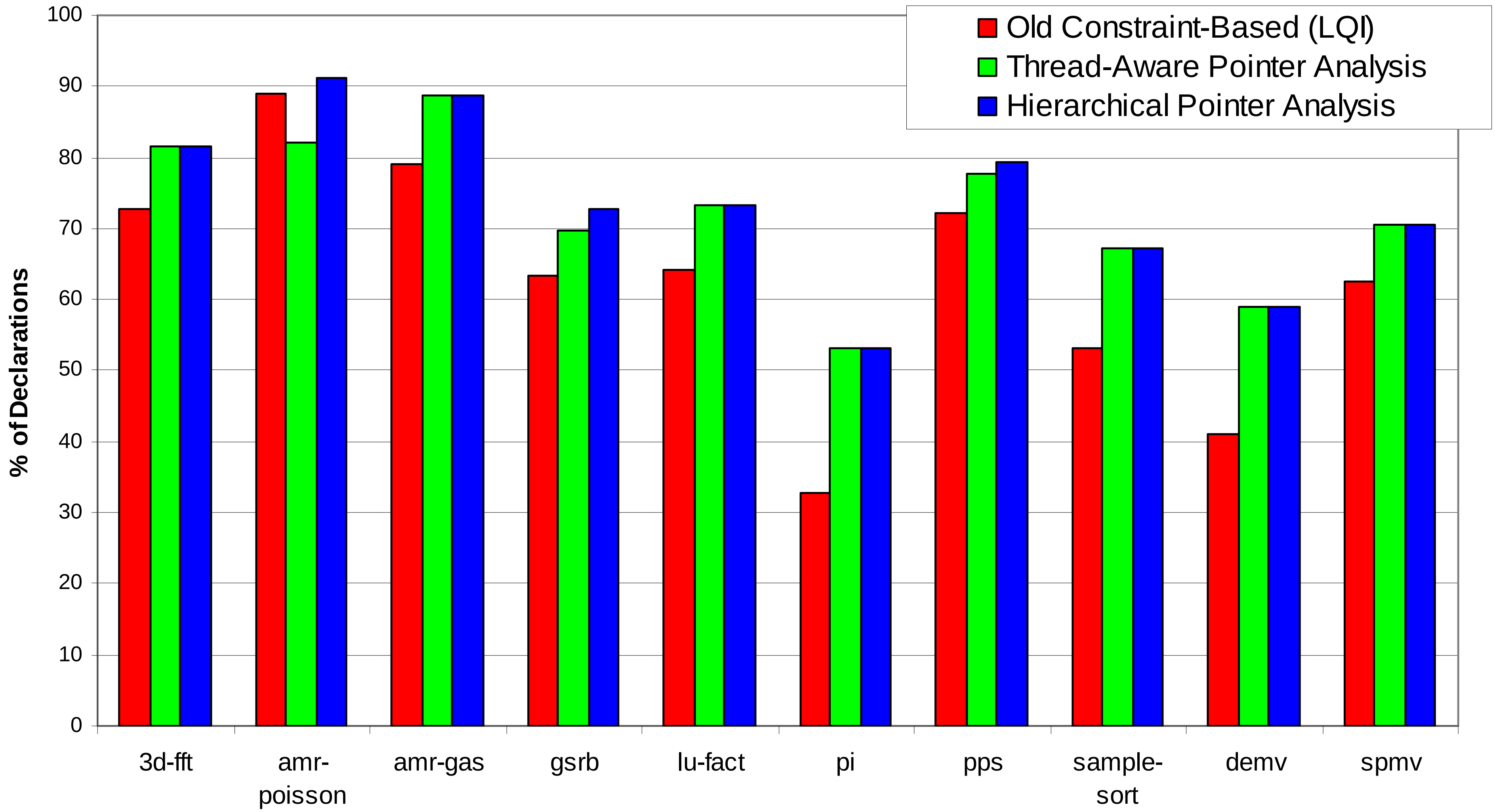
[1] Line counts do not include the reachable portion of the 37,000 line Titanium/Java 1.0 libraries

Analysis Levels

- **Analyses of varying levels of precision**

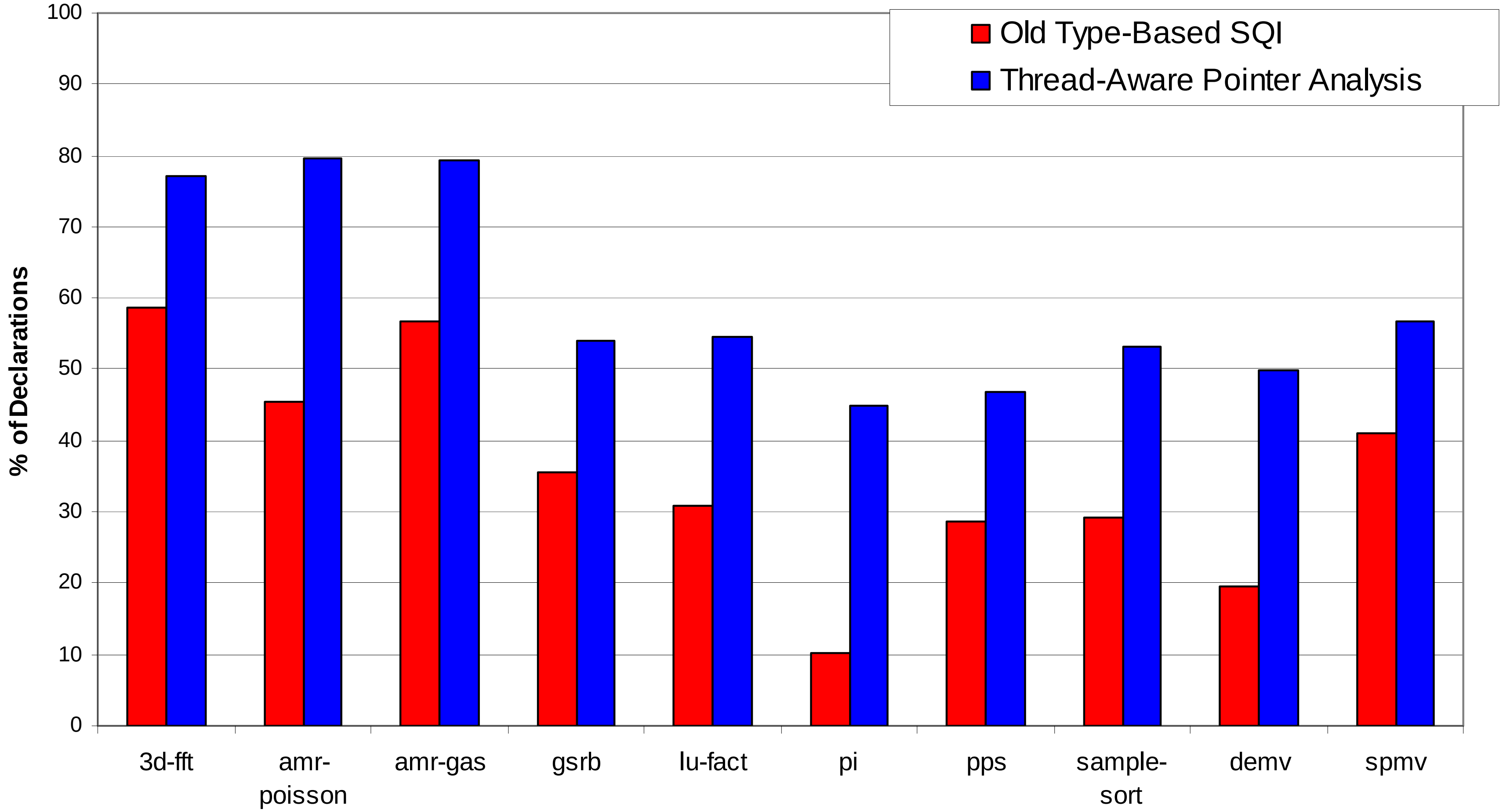
Analysis	Description
`naïve`	All heap accesses
`old LQI/SQI/Sharing`	Previous constraint-based type analysis by Aiken, Gay, and Liblit (different versions for each client)
`concur-multi-level-pointer`	Concurrency analysis + hierarchical (on and off node) thread-aware alias analysis

Declarations Identified as "Local"



Local pointers are both faster and smaller

Declarations Identified as Private

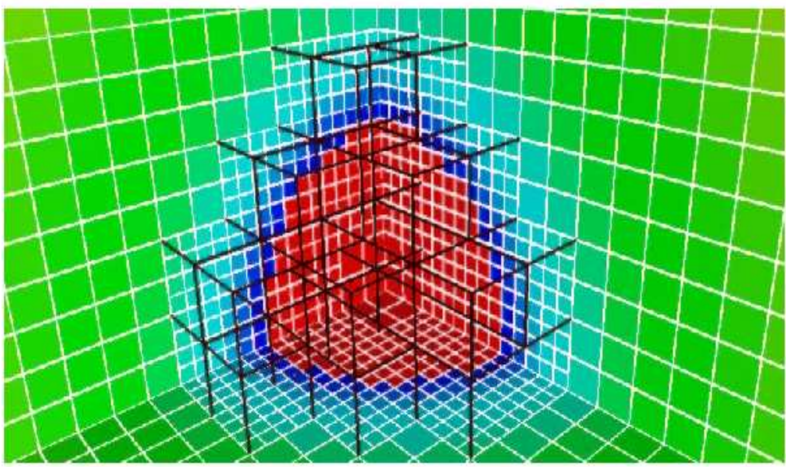


Private data may be cached and is known not to be in a race

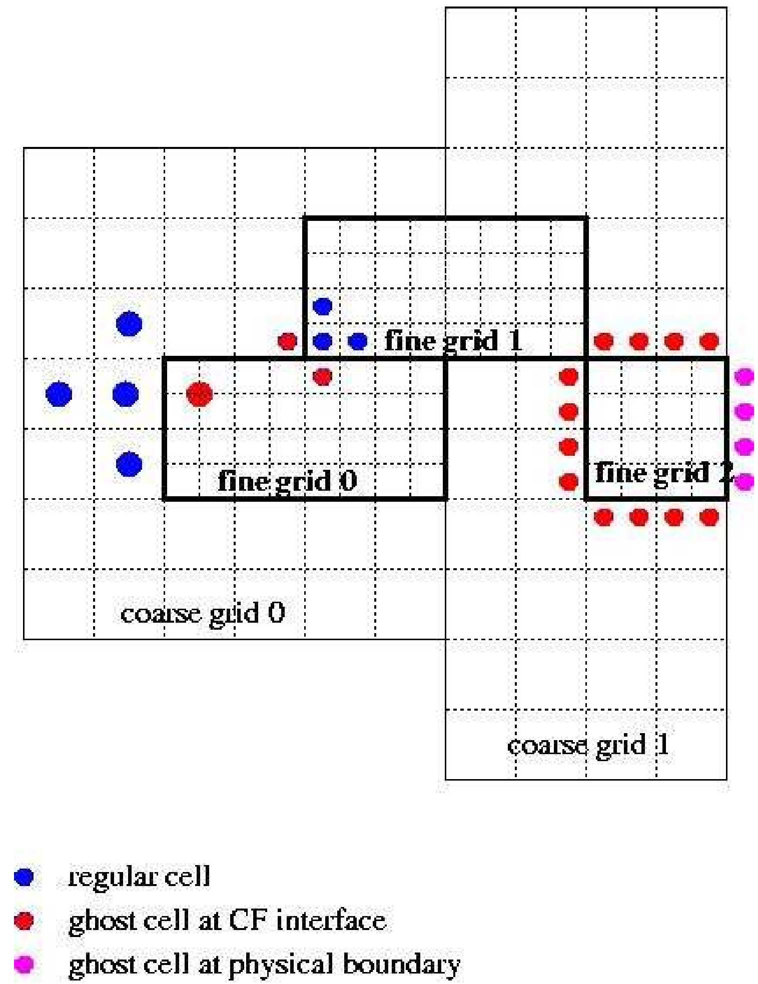
Making PGAS Real: Applications and Portability

Coding Challenges: Block-Structured AMR

- **Adaptive Mesh Refinement (AMR) is challenging**
  - Irregular data accesses and control from boundaries
  - Mixed global/local view is useful



Titanium AMR benchmark available



AMR Titanium work by Tong Wen and Philip Colella

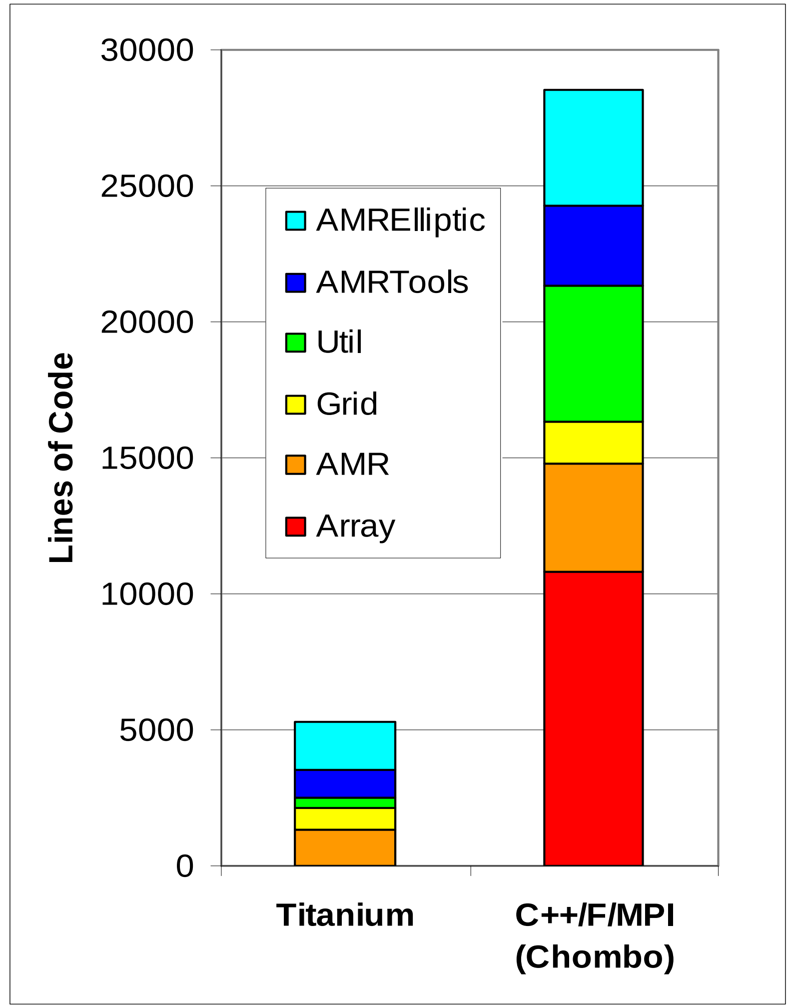
Languages Support Helps Productivity

C++/Fortran/MPI AMR

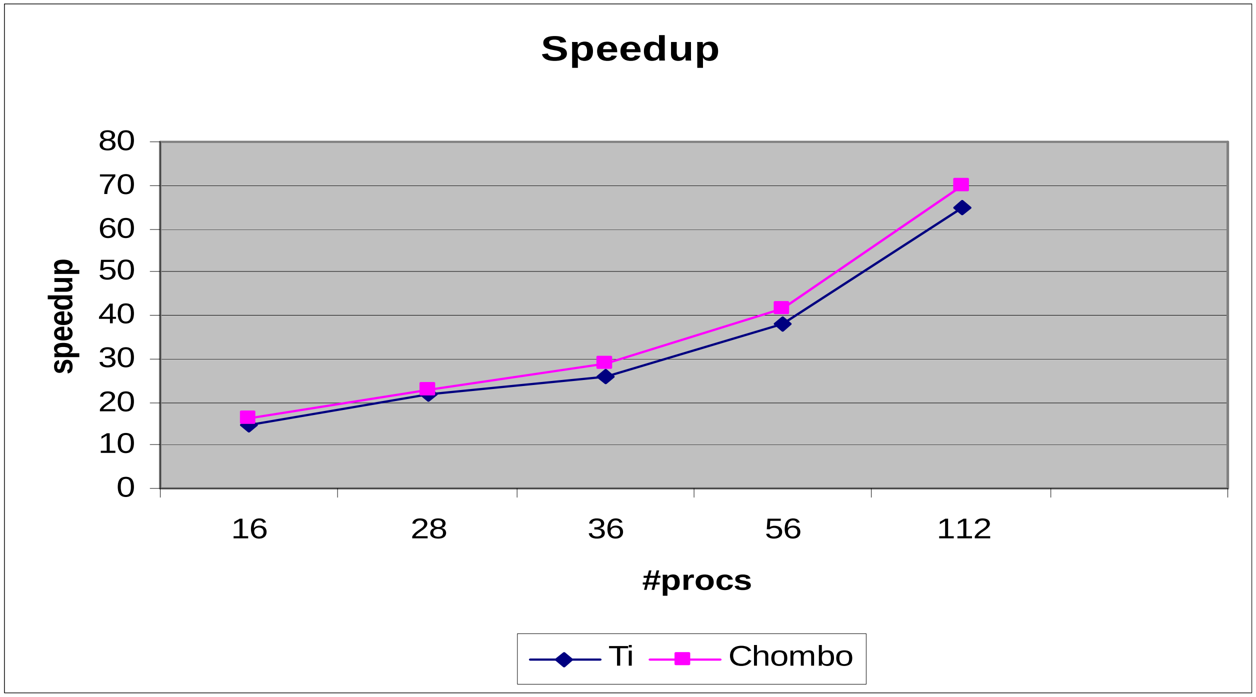
- **Chombo package from LBNL**
- **Bulk-synchronous comm:**
  - Pack boundary data between procs
  - All optimizations done by programmer

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



Performance of Titanium AMR



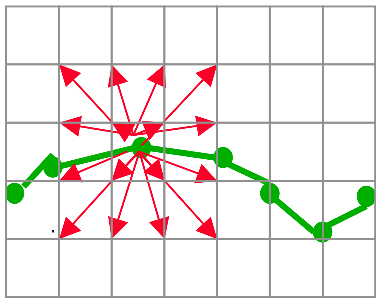
Comparable parallel performance

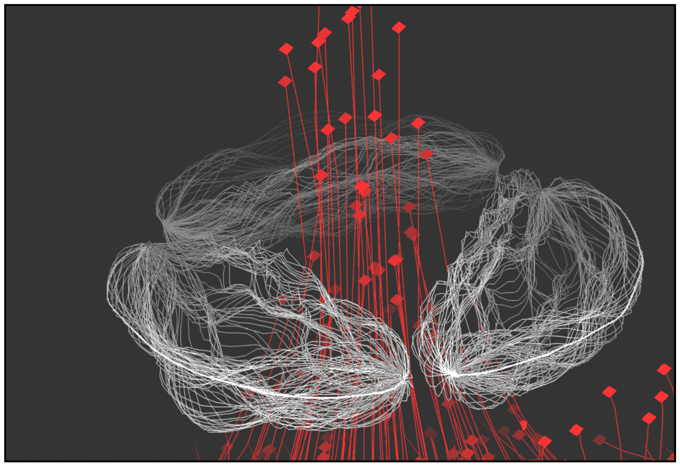
- **Serial: Titanium is within a few % of C++/F; sometimes faster!**
- **Parallel: Titanium scaling is comparable with generic optimizations**
  - **- optimizations (SMP-aware) that are not in MPI code**
  - **- additional optimizations (namely overlap) not yet implemented**

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

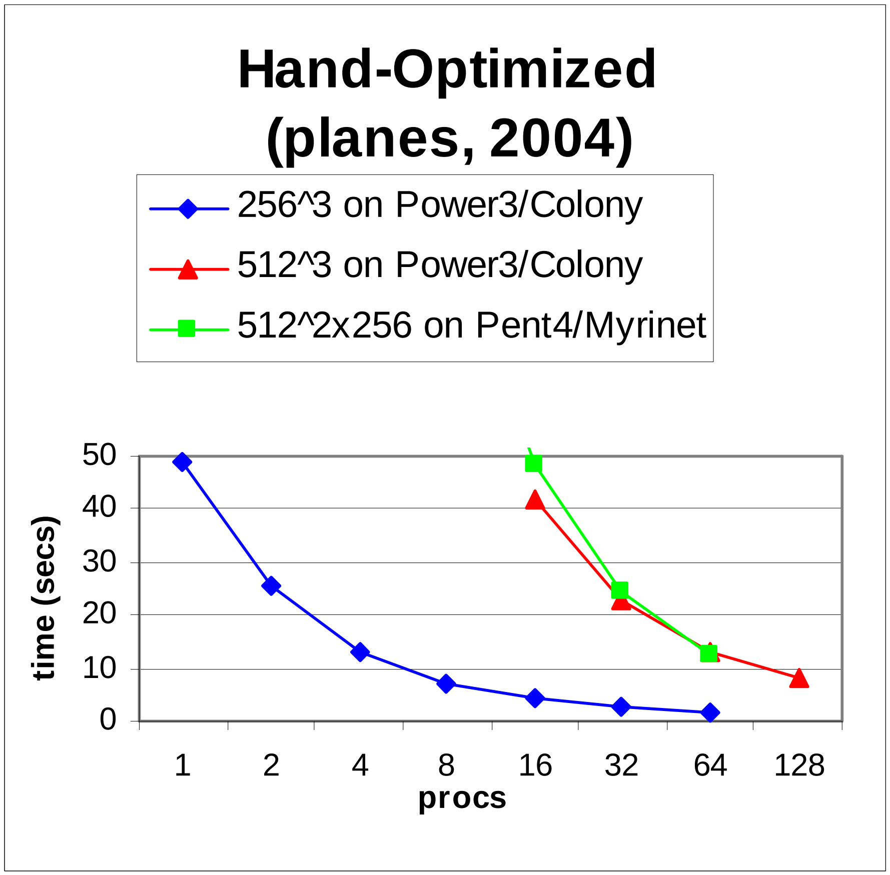


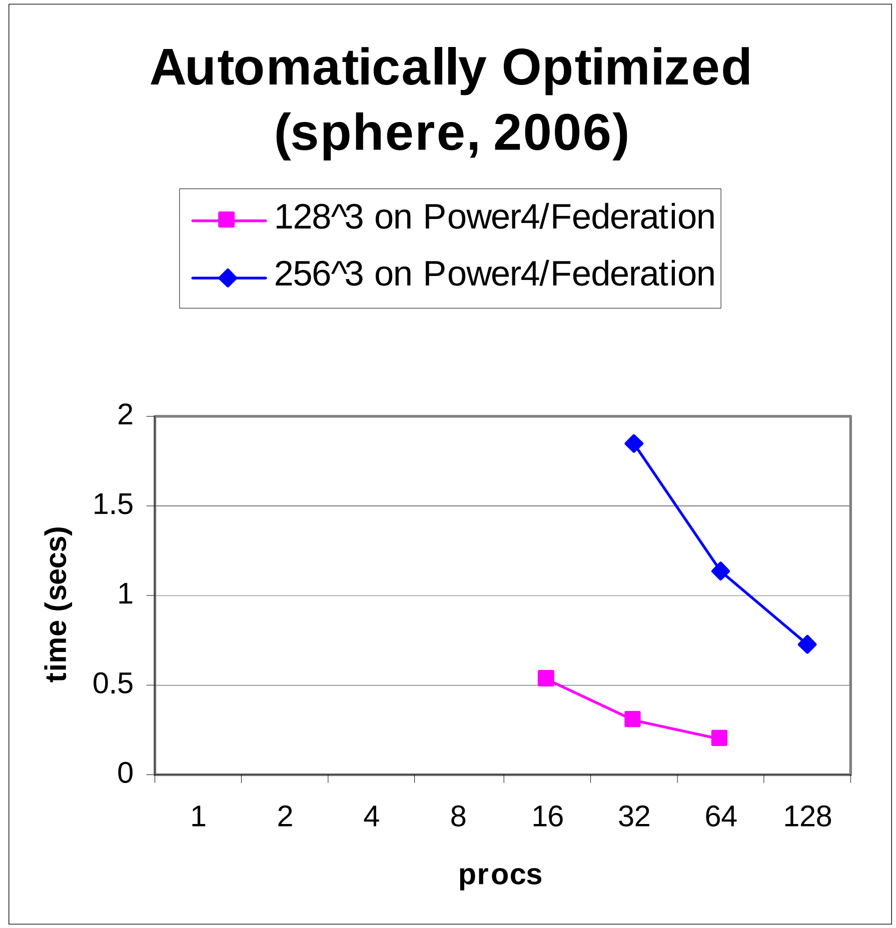


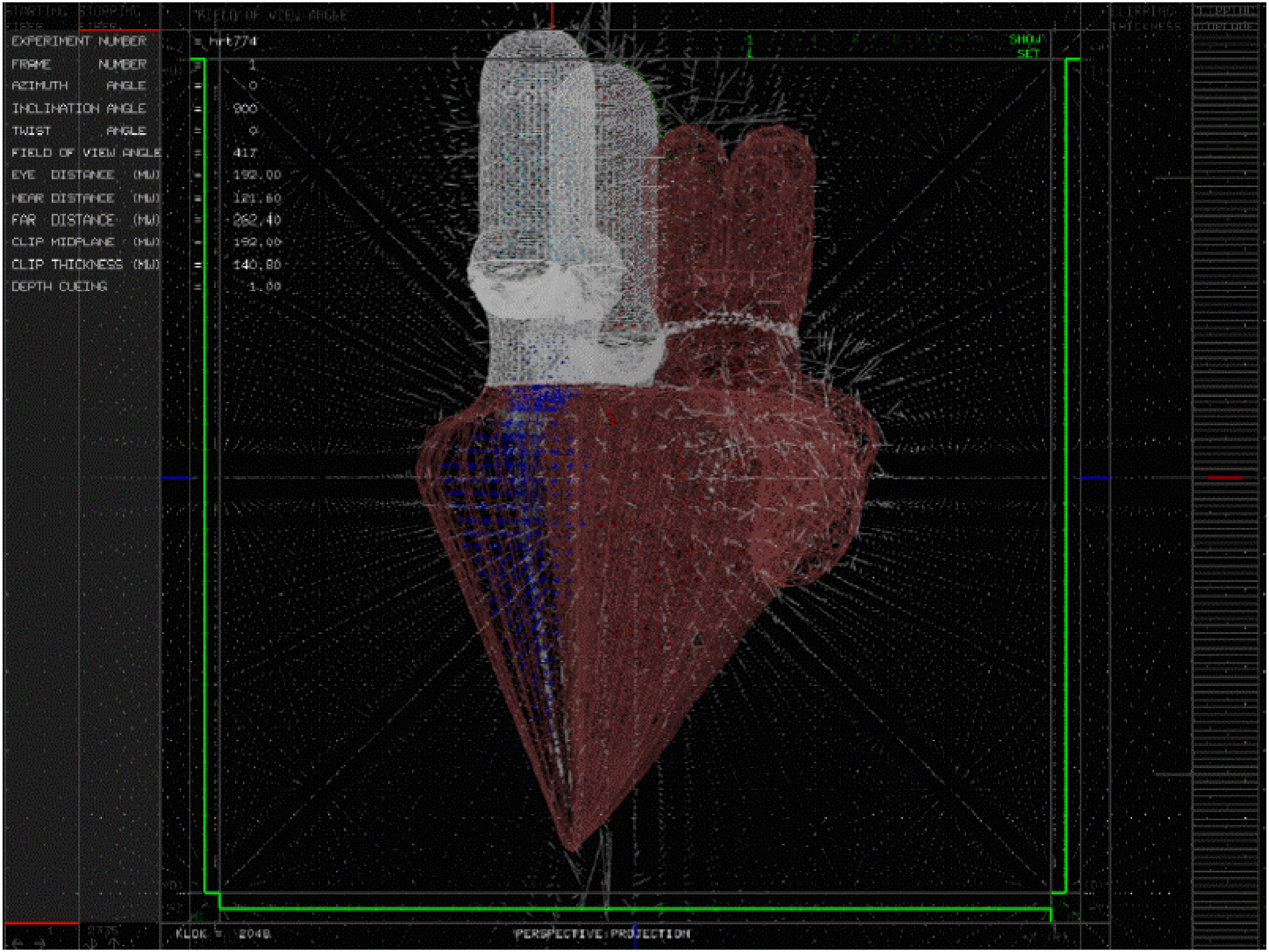
Code Size in Lines	
Fortran	**Titanium**
8000	**4000**

Note: Fortran code is not parallel

Immersed Boundary Method Performance






EXPERIMENT NUMBER = hrt774
FRAME NUMBER = 1
AZIMUTH ANGLE = 0
INCLINATION ANGLE = 900
TWIST ANGLE = 0
FIELD OF VIEW ANGLE = 417
EYE DISTANCE (MW) = 192.00
NEAR DISTANCE (MW) = 121.60
FAR DISTANCE (MW) = 262.40
CLIP MIDPLANE (MW) = 192.00
CLIP THICKNESS (MW) = 140.80
DEPTH CUEING = 1.00
SHOW
SET
KLOK = 2048
PERSPECTIVE PROJECTION

Beyond the SPMD Model: Mixed Parallelism

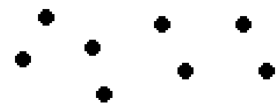
- **UPC and Titanium uses a static threads (SPMD) programming model**
  - General, performance-transparent
  - Criticized as “local view” rather than “global view”
    - “for all my array elements”, or “for all my blocks”
- **Adding extension for data parallelism**
  - Based on collective model:
    - Threads gang together to do data parallel operations
    - Or (from a different perspective) single data-parallel thread can split into P threads when needed
  - Compiler proves that threads are aligned at barriers, reductions and other collective points
    - Already used for global optimizations: read → writes transform
    - Adding support for other data parallel operations

Beyond the SPMD Model: Dynamic Threads

- **UPC uses a static threads (SPMD) programming model**
  - No dynamic load balancing built-in, although some examples (Delaunay mesh generation) of building it on top
  - Berkeley UPC model extends basic memory semantics (remote read/write) with active messages
  - AM have limited functionality (no messages except acks) to avoid deadlock in the network
- **A more dynamic runtime would have many uses**
  - Application load imbalance, OS noise, fault tolerance
- **Two extremes are well-studied**
  - Dynamic load balancing (e.g., random stealing) without locality
  - Static parallelism (with threads = processors) with locality
- **Charm++ has virtualized processes with locality**
  - How much “unnecessary” parallelism can it support?

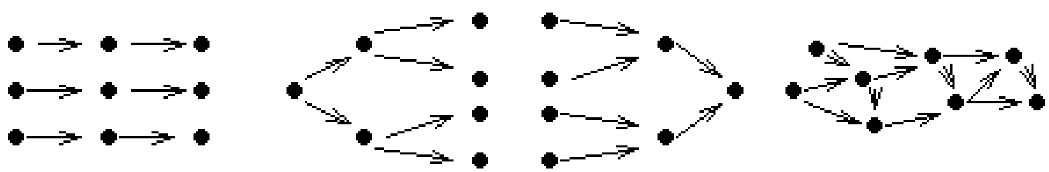
Task Scheduling Problem Spectrum

Easy: The tasks can execute in any order.



dependence free loops

Harder: The tasks have a predictable structure.



wave−front　　out−tree　　in−tree　　general dag

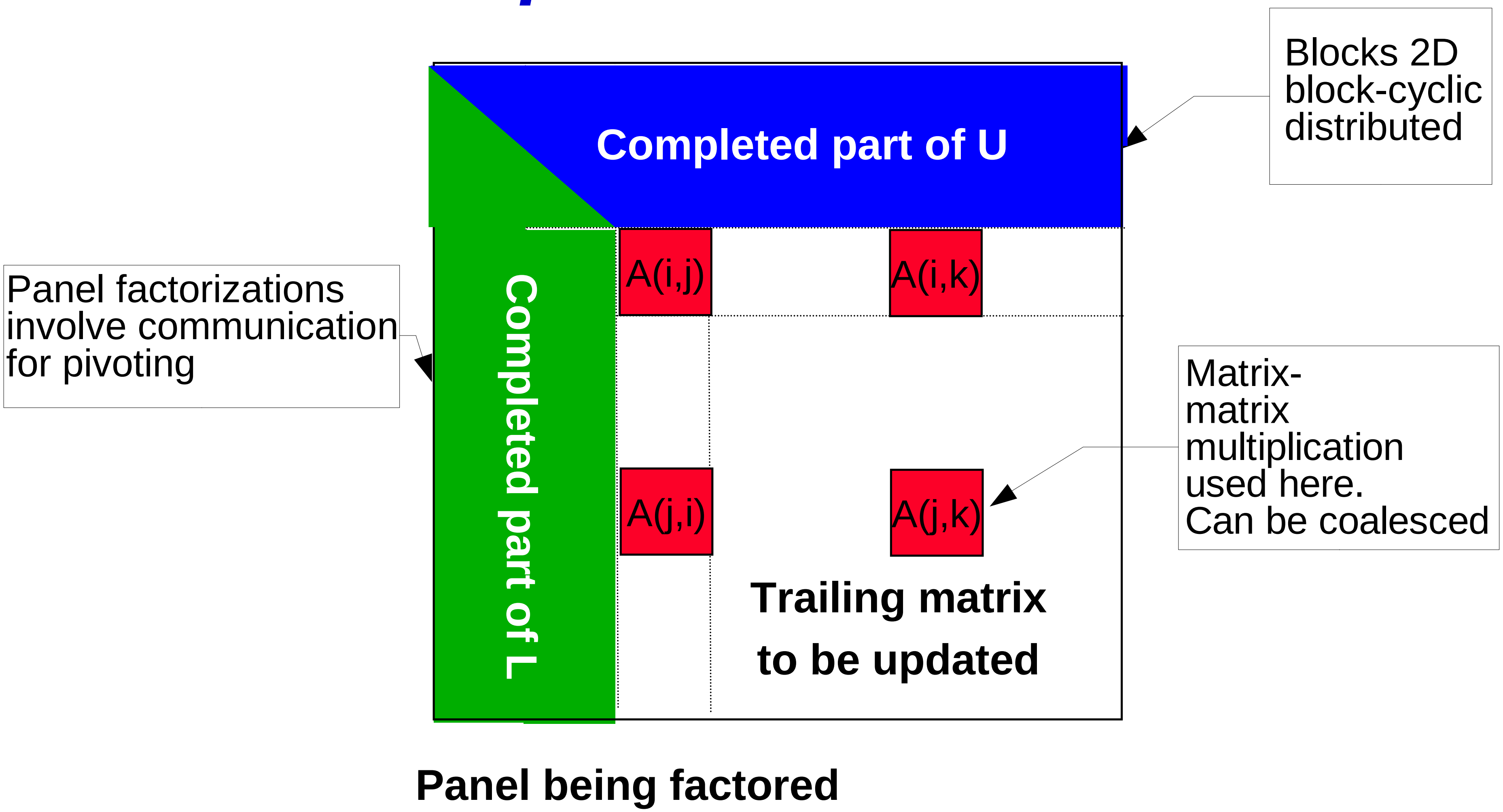
balanced or unbalanced

matrix computations (dense, and some sparse, Cholesky)

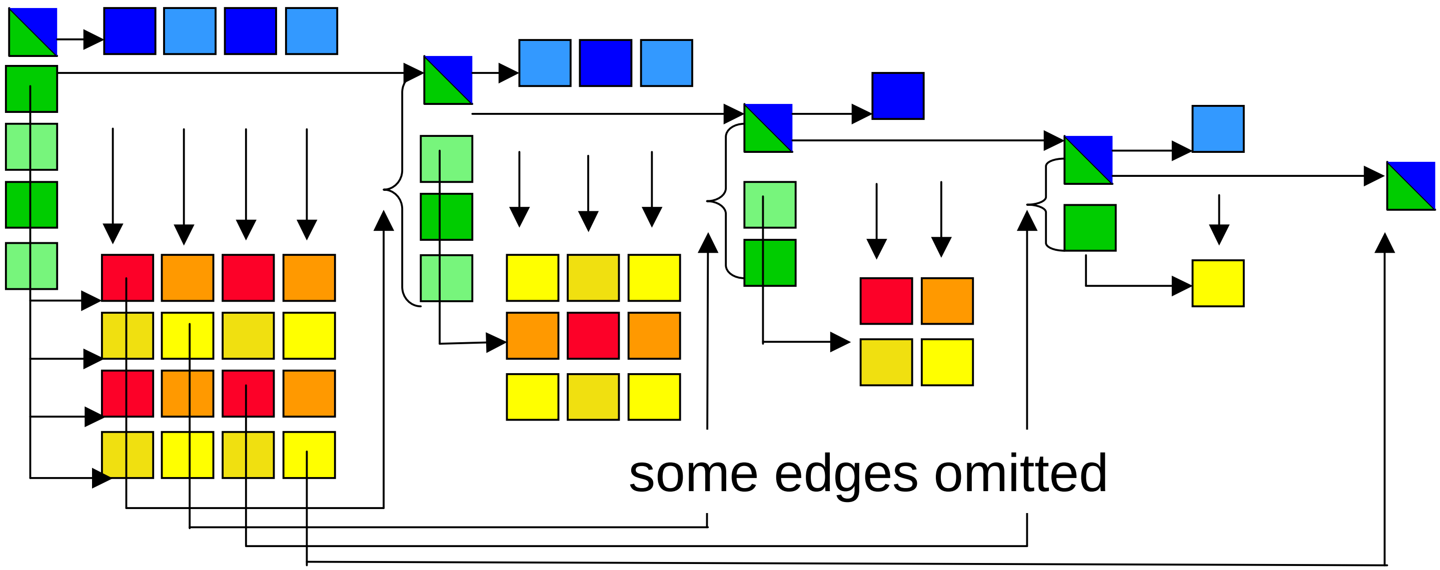
Hardest: The structure changes dynamically (slowly or quickly) search, sparse LU

- How important is locality and what is locality relationship?
- Some tasks must run with dependent tasks to re-use state
- If data is small or compute:communicate ratio large, locality less important
- Can we build runtimes that work for the hardest case: general dag with large data and small compute

Dense and Sparse Matrix Factorization



Parallel Tasks in LU

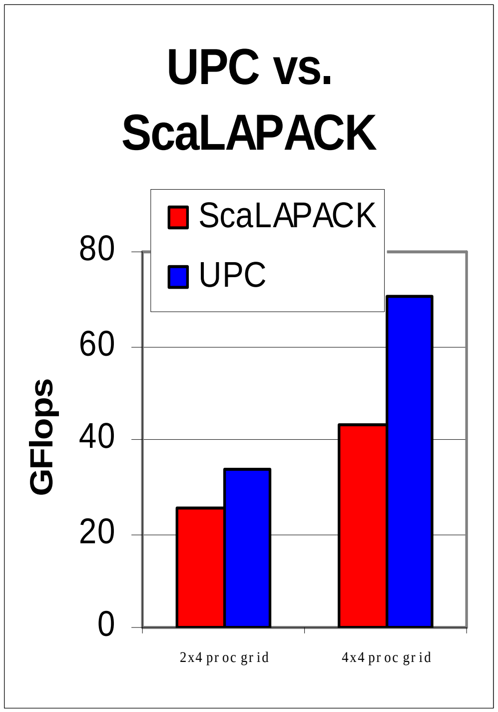


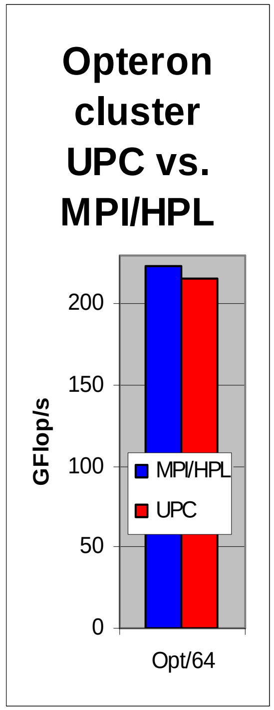
- **Theoretical and practical problem: Memory deadlock**
  - Not enough memory for all tasks at once. (Each update needs two temporary blocks, a green and blue, to run.)
  - If updates are scheduled too soon, you will run out of memory
  - If updates are scheduled too late, critical path will be delayed.

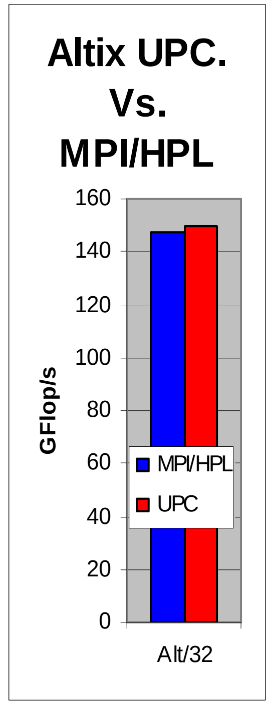
LU in UPC + Multithreading

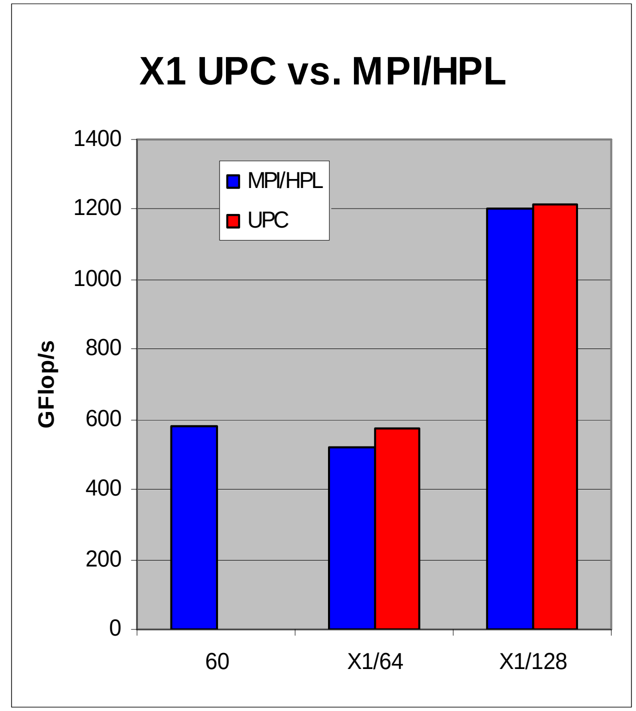
- **UPC uses a static threads (SPMD) programming model**
  - Used to mask latency and to mask dependence delays
  - Three levels of threads:
    - UPC threads (data layout, each runs an event scheduling loop)
    - Multithreaded BLAS (boost efficiency)
    - User level (non-preemptive) threads with explicit 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
  - Collectives (asynchronous reductions for pivoting) need high priority

UPC HP Linpack Performance









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

HPCS Languages

- **DARPA HPCS languages**
  - X10 from IBM, Chapel from Cray, Fortress from Sun
- **Many interesting differences**
  - Atomics vs. transactions
  - Remote read/write vs. remote invocation
  - Base language: Java vs. a new language
  - Hierarchical vs. flat space of virtual processors
- **Many interesting commonalities**
  - Mixed task and data parallelism
    - Data parallel operations are “one-sided” not collective: one thread can invoke a reduction without any help from others
  - Distributed arrays with user-defined distributions
  - Dynamic load balancing built in

Conclusions and Open Questions

- **Best time ever for a new parallel language**
  - Community is looking for parallel programming solutions
  - Not just an HPC problem
- **Current PGAS Languages**
  - Good fit for shared and distributed memory
  - Control over locality and (for better or worse) SPMD
- **Need to break out of strict SPMD model**
  - Load imbalance, OS noise, faults tolerance, etc.
  - Managed runtimes like Charm++ add generality
- **Some open language questions**
  - Can we get the best of global view (data-parallel) and local view in one efficient parallel language
  - Will non-SPMD languages have sufficient resource control for applications with complex task graph structures?