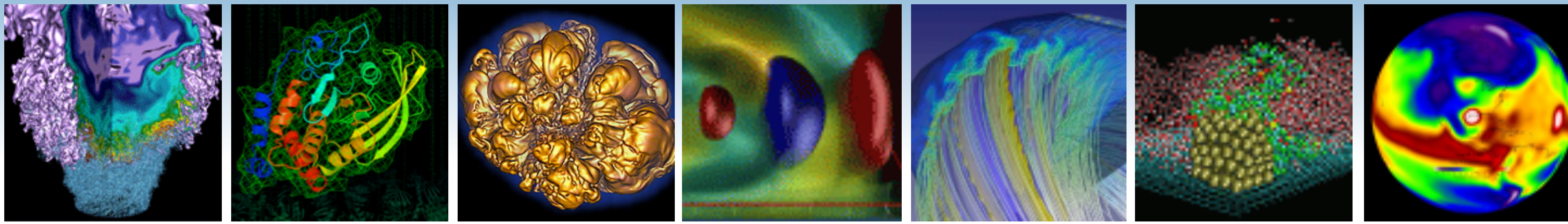


Beyond UPC

Kathy Yelick

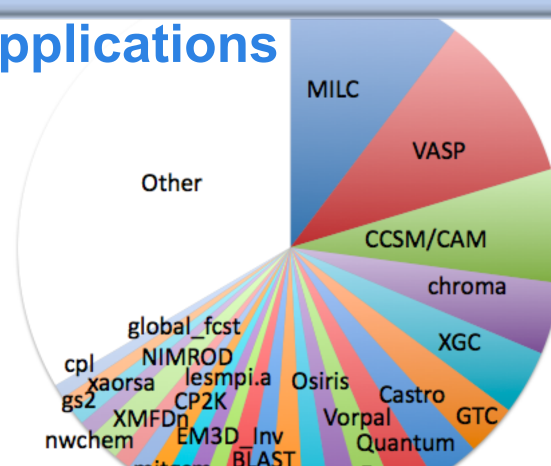
EECS Professor, UC Berkeley

Associate Laboratory Director for Computing Sciences
Lawrence Berkeley National Laboratory

The National Energy Research Scientific Computing Center Enables Science

1500+ publications per year

4500 users ~600 applications



25 applications = 2/3s of workload

Petaflop and Petabyte systems for science

Requirements For Future

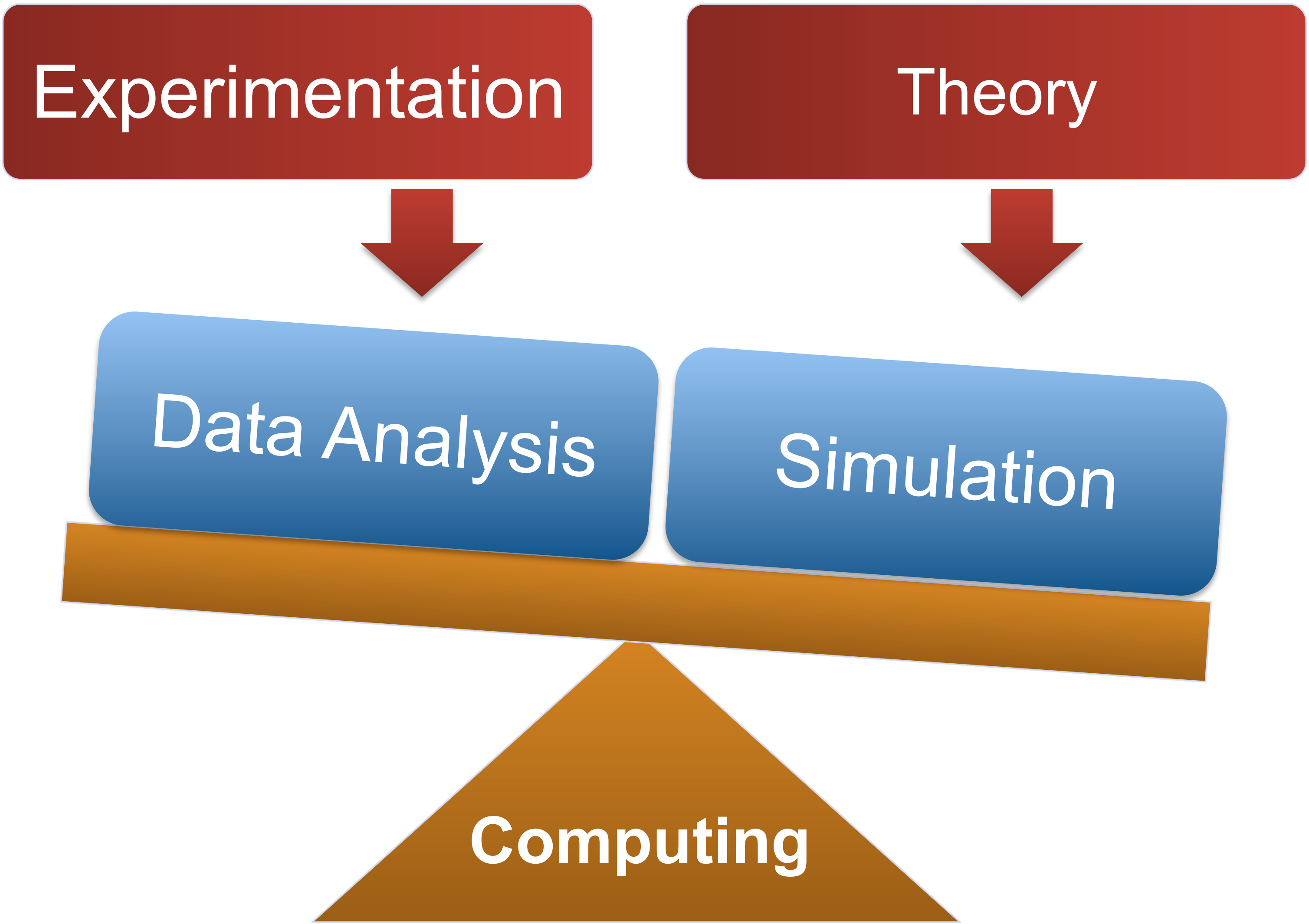
- 2x gap in demand vs. capability across centers
- 10x gap by 2015 (NERSC)
- ~650 applications with these programming models
  - 75% Fortran, 45% C/C++, 10% Python
  - 85% MPI, 25% with OpenMP
  - 10% PGAS or global objects

These are self-reported, likely low and may miss future users

http://science.energy.gov/ascr/news-and-resources/

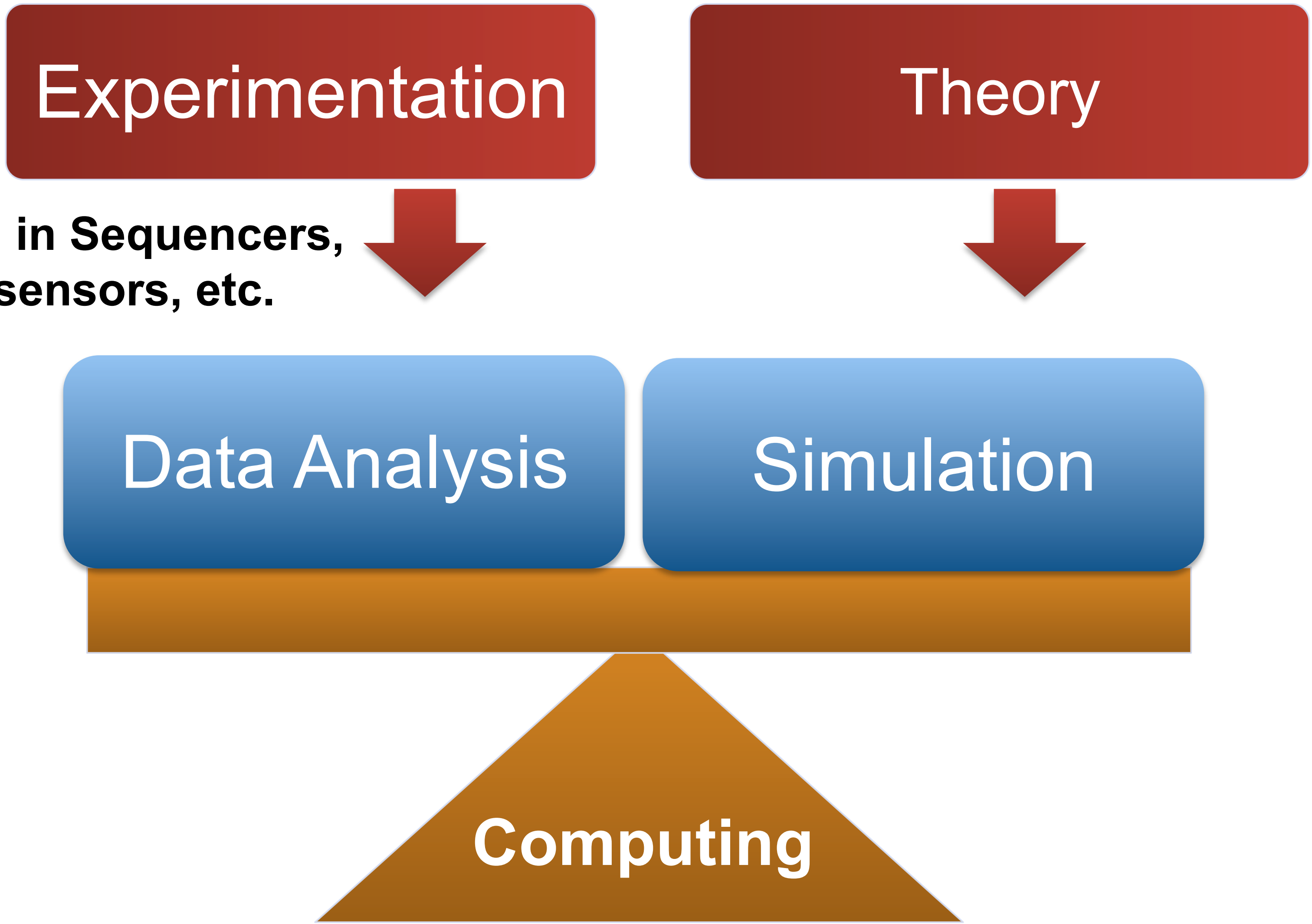
http://www.nersc.gov/science/requirements-reviews/final-reports/

Computing = Data Analysis and Simulation

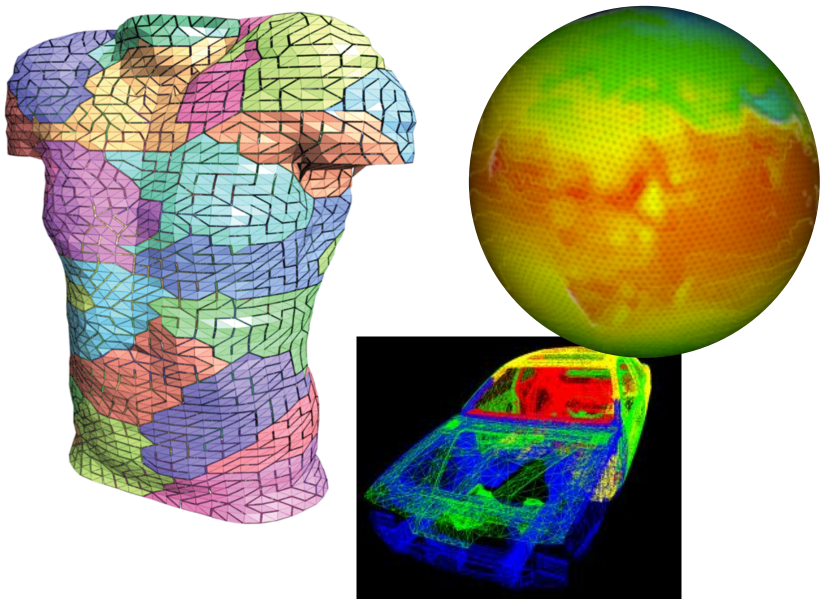


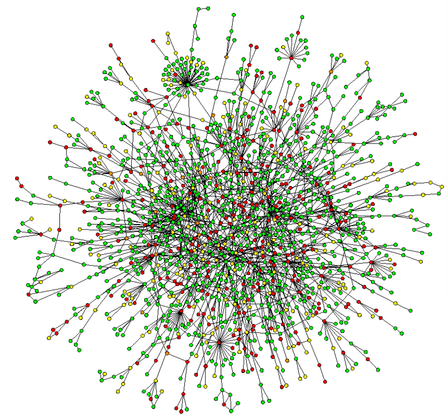
Data analysis is equally important in Science

Growth in Sequencers, CCDs, sensors, etc.



Programming Challenges and Solutions





Message Passing Programming

Divide up domain in pieces

Each compute one piece

Exchange (send/receive) data

PVM, MPI, and many libraries

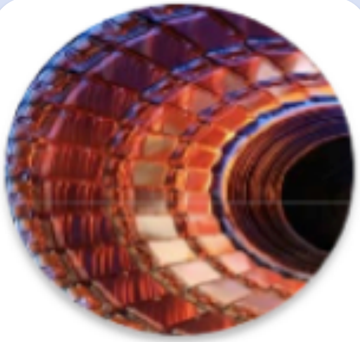
Global Address Space Programming

Each start computing

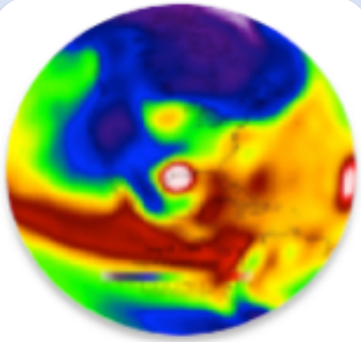
Grab whatever you need whenever

Global Address Space Languages and Libraries

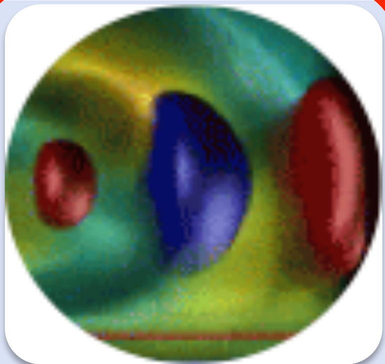
Science Across the "Irregularity" Spectrum



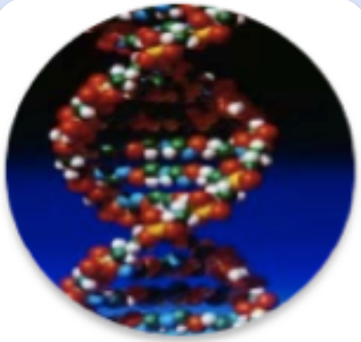
Massive Independent Jobs for Analysis and Simulations



Nearest Neighbor Simulations



All-to-All Simulations

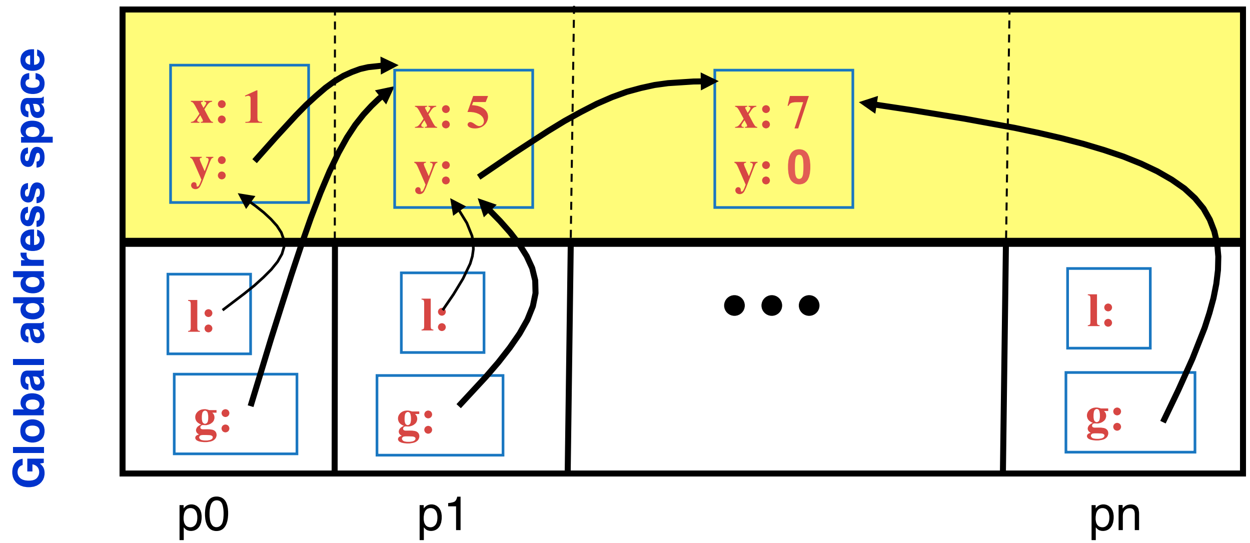


Random access, large data Analysis

Data analysis and simulation

PGAS Languages

- ***Global address space:*** thread may directly read/write remote data
  - Convenience and low overhead
- ***Partitioned:*** data is designated as local or global
  - Locality and scalability



UPC: A PGAS language based on C

See CS267 UPC Lectures for more details

Or attend SC13 tutorial on advanced UPC!

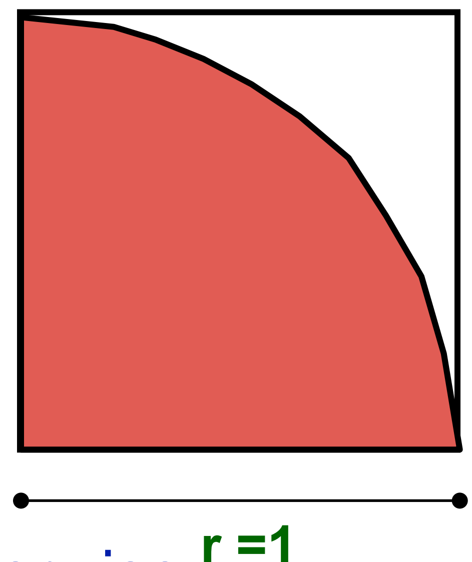
UPC Execution Model

- A number of threads working independently in a SPMD fashion
  - Number of threads specified at compile-time or run-time; available as program variable **`THREADS`**
  - **`MYTHREAD`** specifies thread index (**`0..THREADS-1`**)
  - **`upc_barrier`** is a global synchronization: all wait
  - There is a form of parallel loop for distributing work
- UPC has locks to protect shared variables: **`upc_lock_t`**

```
upc_lock_t *myLock = upc_all_lock_alloc();
upc_lock(myLock)
    critical region
upc_unlock(myLock)
upc_lock_free(myLock);
```

Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} * \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- Compute ratio:
  - # points inside / # points total
  - $\pi = 4*\text{ratio}$



- Assume serial function:

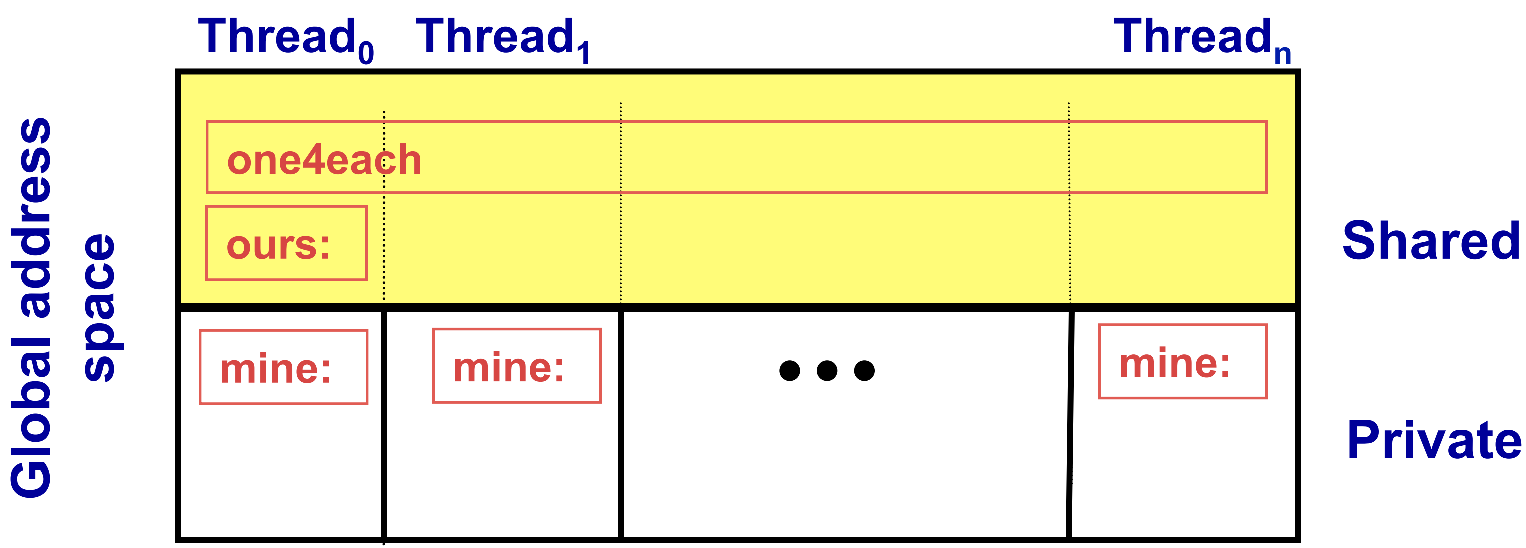
  int hits ()

  - for x, y, return 1 if $x^2 + y^2 < 1$, 0 otherwise

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0

```
shared int ours;  // use sparingly: performance
int mine;
int one4each [THREADS]; // cyclic layout
```



Pi in UPC: Shared Memory Style

- **Parallel computing of pi, without the bug**

```
shared int hits;
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
       my_hits += hit();
    upc_lock(hit_lock);
    hits += my_hits;
    upc_unlock(hit_lock);
    upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*hits/trials);
}
```

create a lock

accumulate hits locally

accumulate across threads

Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
  ... declarations and initialization code omitted
  for (i=0; i < my_trials; i++)
     all_hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++) hits += all_hits[i];
    printf("PI estimated to %f.", 4.0*hits/trials);
  }
}
```

all_hits is shared by all processors

update element with local affinity

Common Uses for UPC Pointer Types

```
int *p1;
```

- These pointers are fast (just like C pointers)
- Use to access local data in part of code performing local work
- Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

```
shared int *p2;
```

- Use to refer to remote data
- Larger and slower due to test-for-local + possible communication
- Typical implementation has a thread ID + address + phase

```
int *shared p3;
```

- Not recommended

```
shared int  *shared p4;
```

- Use to build shared linked structures, e.g., a linked list

UPC Arrays and Collectives

Gather threads together for data-parallel style operations

Pi in UPC: Data Parallel Style

- The previous version of Pi works, but is not scalable:
  - On a large # of threads, the locked region will be a bottleneck
- Use a reduction for better scalability

```
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
    ...
    for (i=0; i < my_trials; i++)
       my_hits += hit();
    my_hits =           // type, input, thread, op
       bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*my_hits/trials);
}
```

Berkeley collectives

no shared variables

barrier implied by collective

Vector Addition with upc_forall

- The vector addition can be written as follows
  - The code would be correct but slow if the affinity expression were `i+1` rather than `i`.
  - Equivalent code could use "`&sum[i]`" for affinity
  - Better style: if `sum` layout changes, still get good affinity

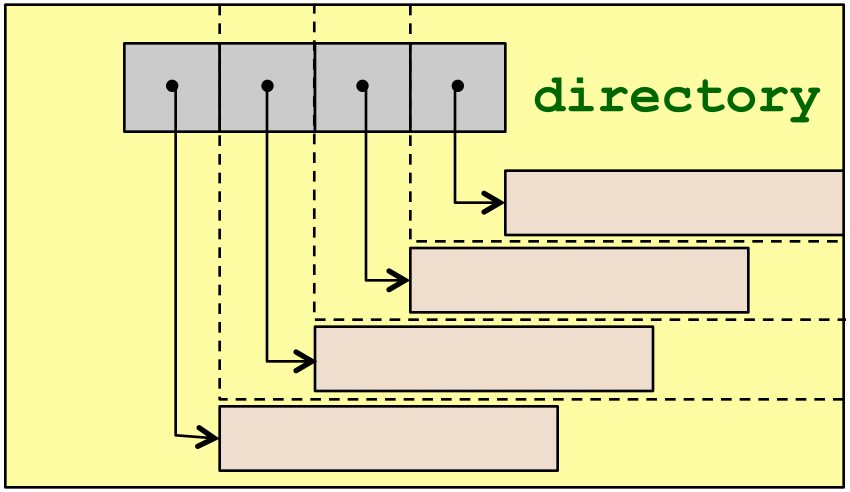
```
#define N 100*THREADS

  shared [100] int v1[N], v2[N], sum[N];

  void main() {
      int i;
      upc_forall(i=0; i<N; i++; i)
                sum[i]=v1[i]+v2[i];
  }
```

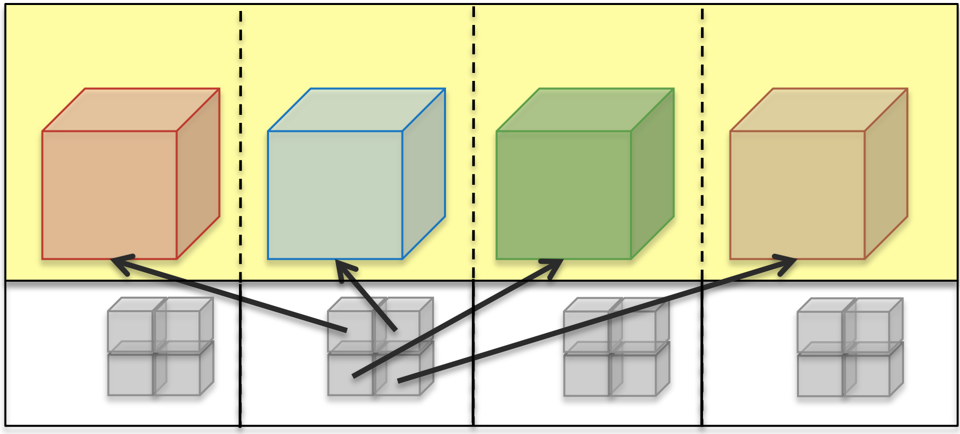
&sum[i]

Distributed Arrays Directory Style

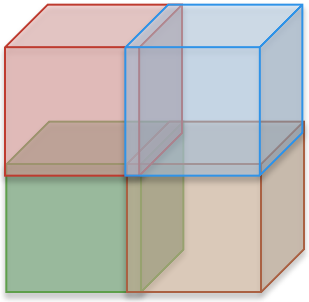
- Many UPC programs avoid the UPC style arrays in factor of directories of objects

```
typedef shared [] double *sdblptr;
shared sdblptr directory[THREADS];
directory[i]=upc_alloc(local_size*sizeof(double));
```



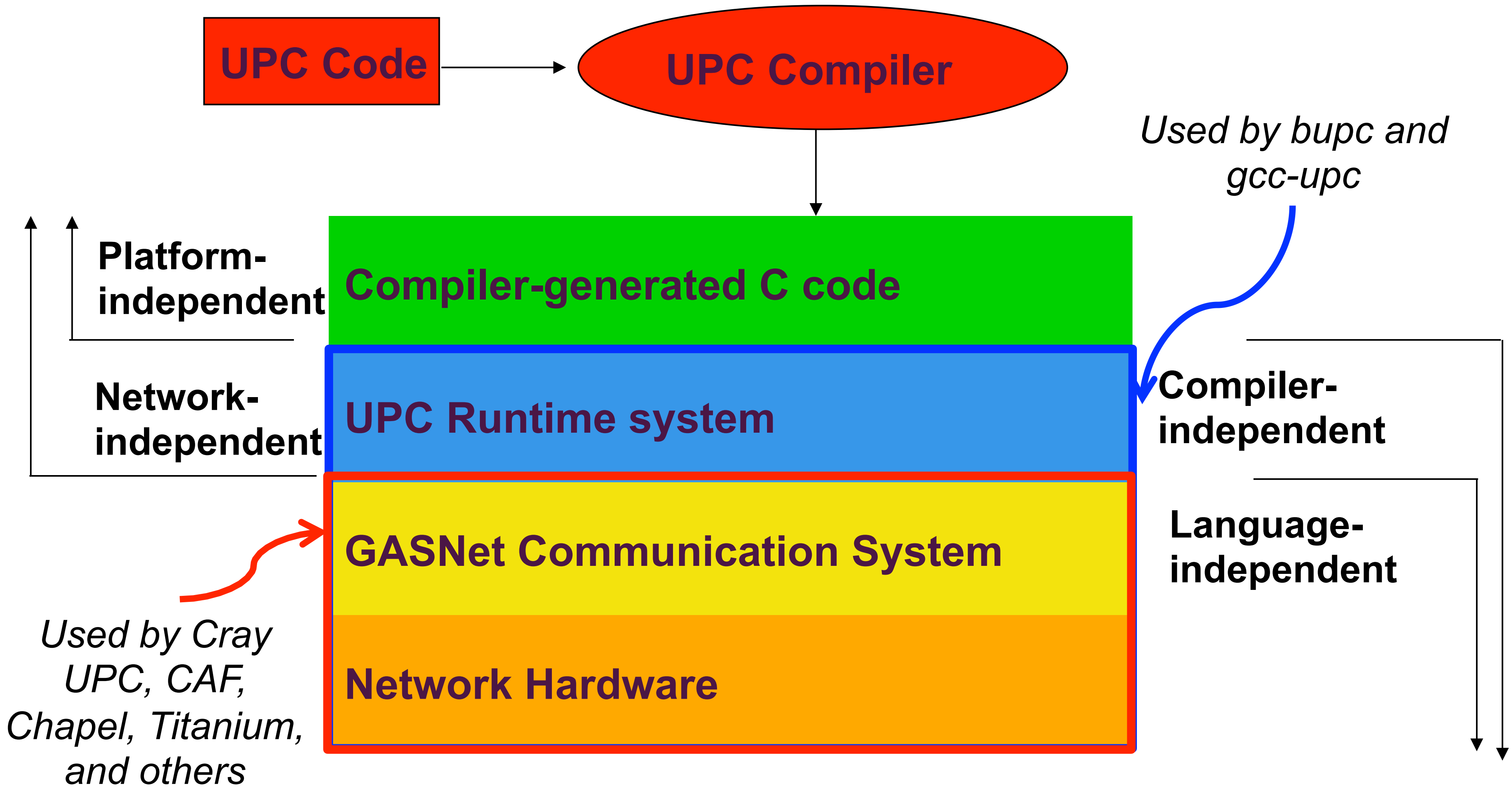


- These are also more general:
  - Multidimensional, unevenly distributed
  - Ghost regions around blocks

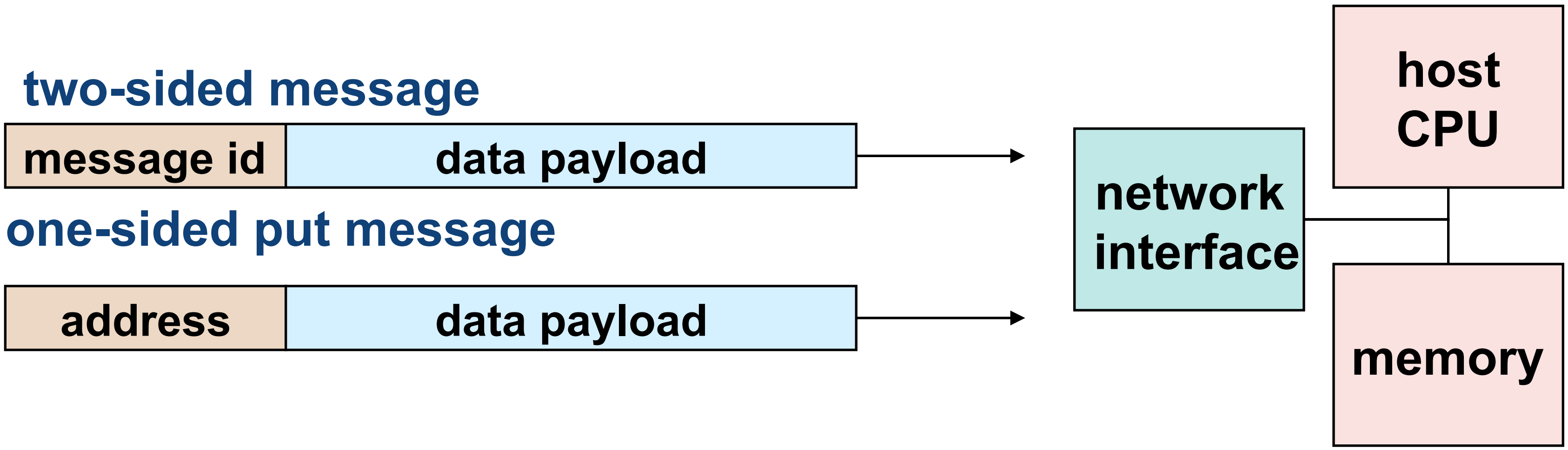


Performance of UPC

Berkeley UPC Compiler

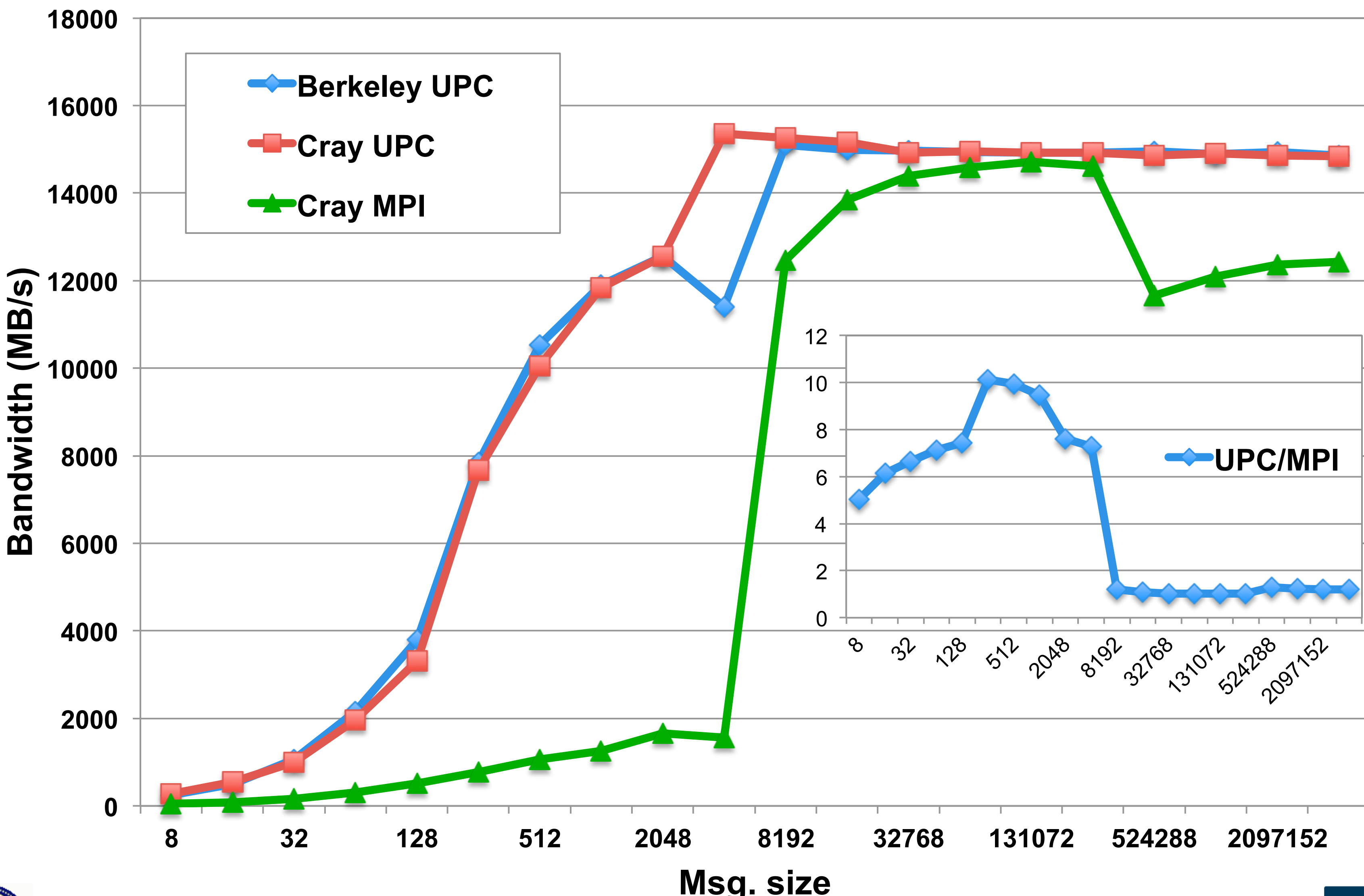


Avoiding Synchronization in Communication

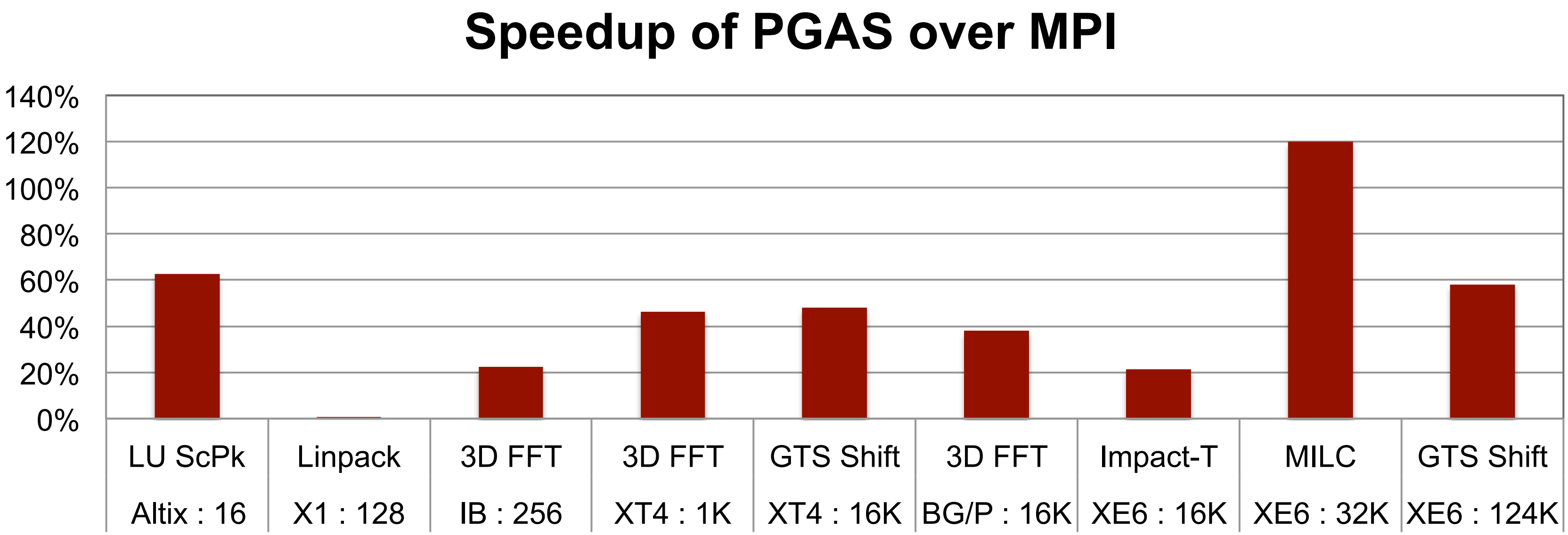


- Two-sided message passing (e.g., MPI) requires a matching receive to identify memory address to put data
  - Couples data transfer with synchronization (but it ain't free!)
- Global address space decouples synchronization
  - Separately synchronize as needed
  - Never have to say "receive"
- NB: MPI 1-sided can have same performance advantages

Bandwidths on Cray XE6 (Hopper)



PGAS's One-sided communication has performance advantages



Performance advantages for PGAS over MPI from

- Lower latency and overhead
- Better pipeline (overlapping communication with communication)
- Overlapping communication with computation
- Use of bisection bandwidth

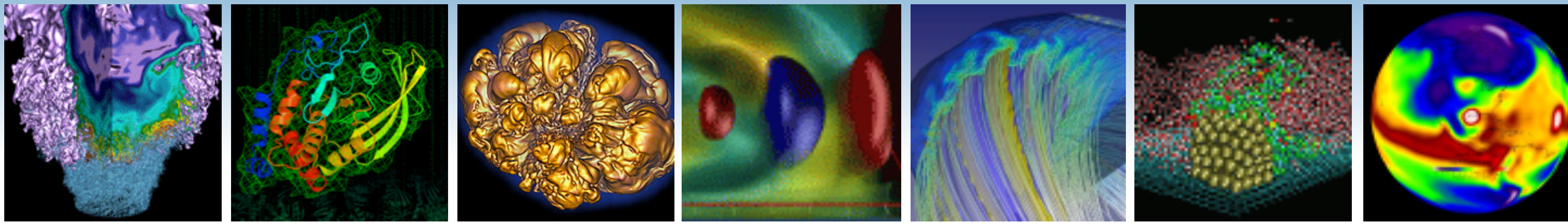
PyGAS: Combine two popular ideas

- Python
  - No. 6 Popular on http://langpop.com and extensive libraries, e.g., Numpy, Scipy, Matplotlib, NetworkX
  - 10% of NERSC projects use Python
- PGAS
  - Convenient data and object sharing
- PyGAS : Objects can be shared via *Proxies* with operations intercepted and dispatched over the network:

```
num = 1+2*j
    = share(num, from=0)
```

```
print pxy.real # shared read
pxy.imag = 3   # shared write
print pxy.conjugate() # invoke
```

- Leveraging duck typing:
  - *Proxies* behave like original objects.
  - Many libraries will automatically work.



Antisocial Parallelism: Avoiding, Hiding and Managing Communication

Kathy Yelick

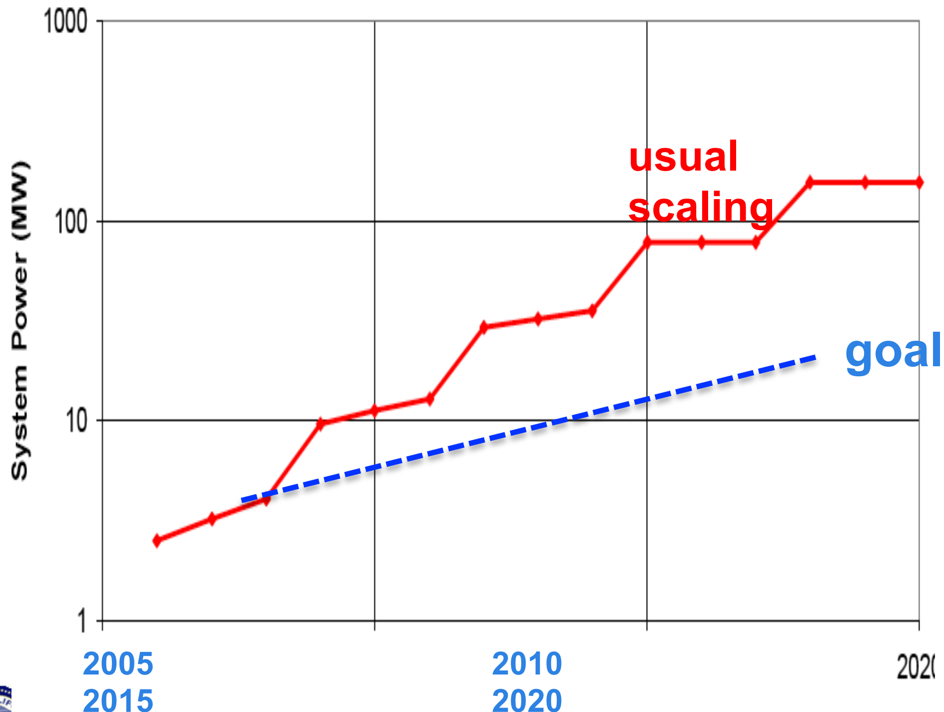
EECS Professor, UC Berkeley

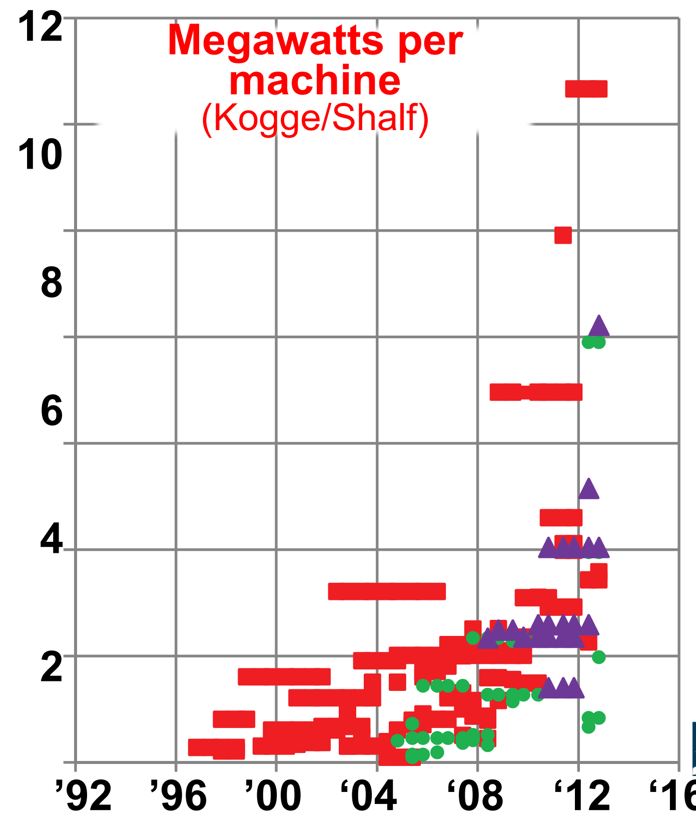
Associate Laboratory Director for Computing Sciences
Lawrence Berkeley National Laboratory

Challenge #1: Computing is energy-constrained

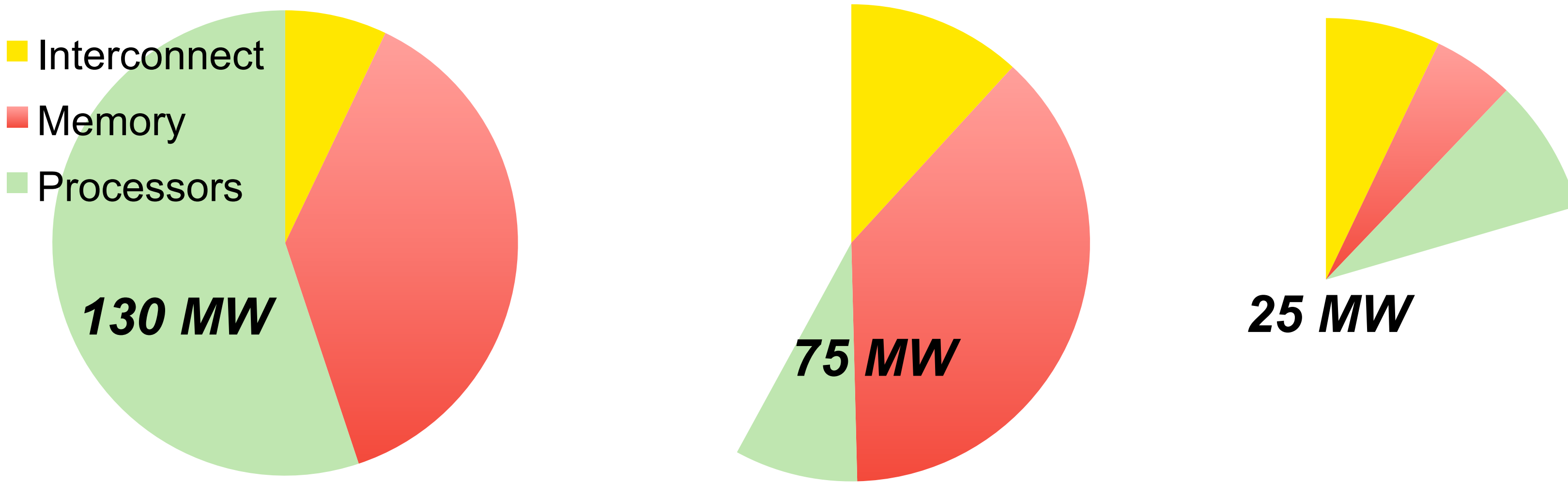
At ~$1M per MW, energy costs are substantial

- 1 petaflop in 2008 used 3 MW
- 1 exaflop in 2018 possible in 200 MW with “usual” scaling
- Goal: 1 exaflop in 20 MW = 20 pJ / operation





New Processors Means New Software

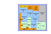


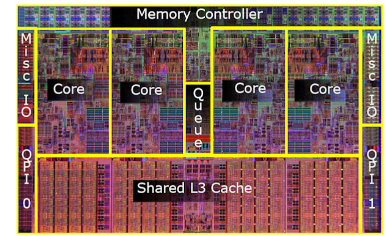
Server Processors ***Manycore*** ***Low power memory and interconnect***

- Exascale will have chips with thousands of tiny processor cores, and a few large ones
  - Sea of lighweight cores with heavyweight “service” nodes
  - Or lightweight cores as accelerators to CPUs
- Low power memory and storage technology are essential
  - Probably with more software management to avoid waste

Challenge #2: Nodes with Heterogeneity and Locality

Cell phone processor (0.1 Watt, 4 Gflop/s)





Server processor (100 Watts, 50 Gflop/s)

- Case for heterogeneity
  - Many small cores and SIMD for energy efficiency; few CPUs for OS / speed
- Local store, explicitly managed memory
  - More efficient (get only what you need) and simpler hardware
- Split memory between CPU and “Accelerators”
  - Driven by market history and simplicity, but may not last
  - Communication: The bus is a significant bottleneck.
- Co-Processor interface between CPU and Accelerator
  - Default is on CPU, only run “parallel” code in limited regions
  - Why are the minority CPUs in charge?

Avoid vicious cycle: Programming model should be designed for future, not for current/past constraints

Memory Speed vs. Capacity Conundrum

Cost (increases for higher capacity and cost/bit increases with bandwidth)

Power

Bandwidth\Capacity	16 GB	32 GB	64 GB	128 GB	256 GB	512 GB	1 TB
4 TB/s							
2 TB/s	Stack/PNM						
1 TB/s			Interposer				
512 GB/s				HMC organic			
256 GB/s							NVRAM
128 GB/s						DIMM	

optical

block

- Because of cost and power issues, we cannot have both high memory bandwidth and large memory capacity
- The colored region is feasible in 2017

Compute intensive architecture focus on upper-left

Data Intensive architecture focus on lower right

Slide source: John Shalf

Compiler-free "UPC++" eases interoperability

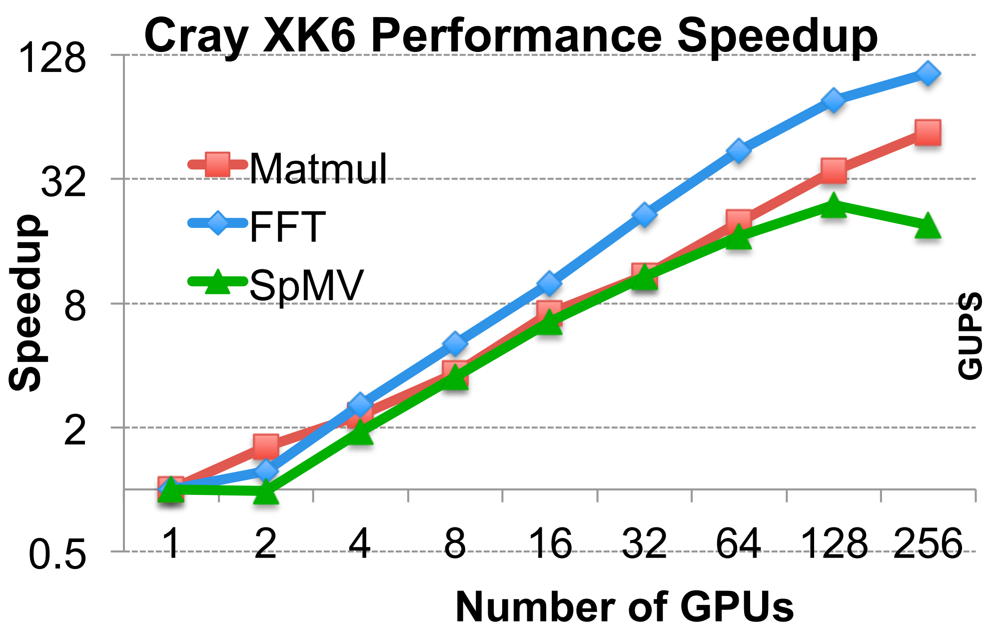
global_array_t<int, 1> A(10); // shared [1] int A[10];

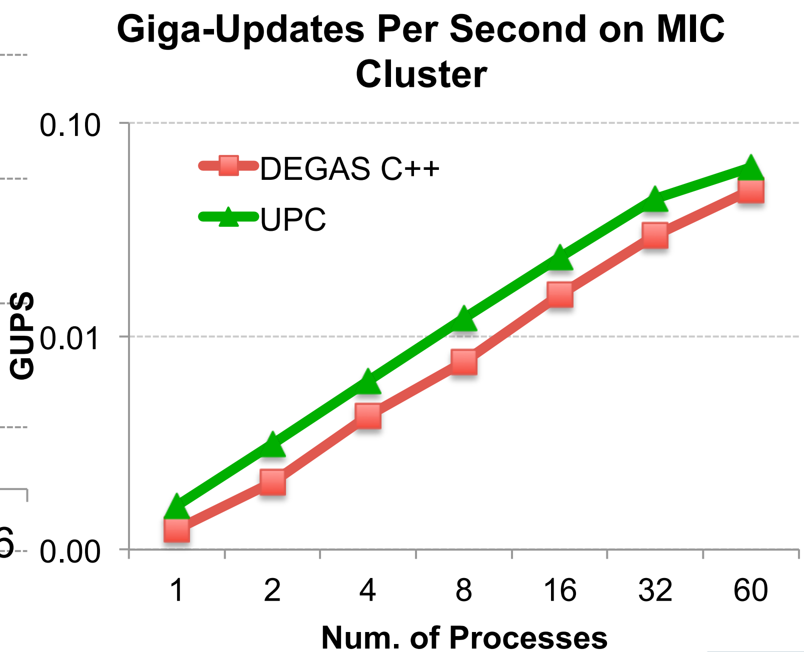
L-value reference (write/put)

A[1] = 1; // A[1] -> global_ref_t ref(A, 1); ref = 1;

R-value reference (read/get)

int n = A[1] + 1; // A[1] -> global_ref_t ref(A, 1); n = (int)ref + 1;

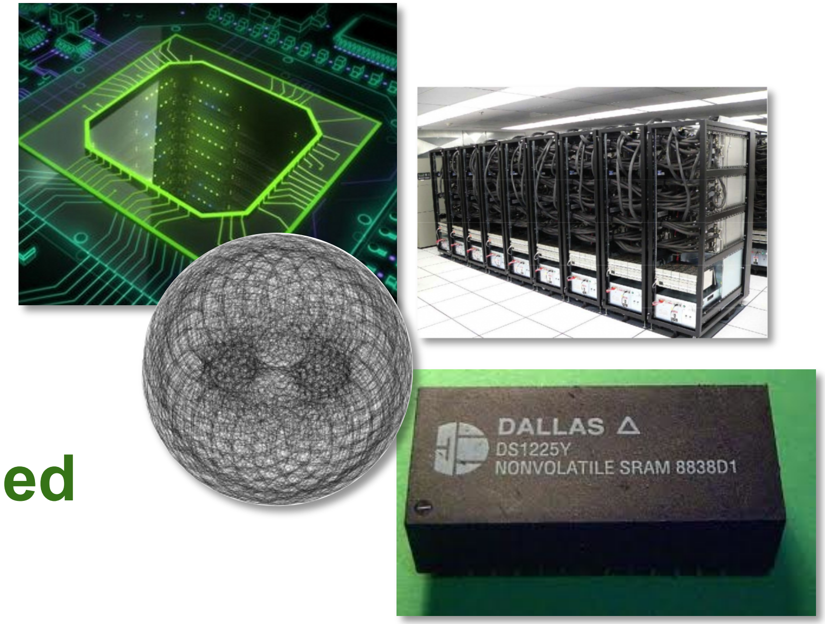




One-sided communication works everywhere

PGAS programming model

```
*p1 = *p2 + 1;
A[i] = B[i];

upc_memput(A,B,64);
```



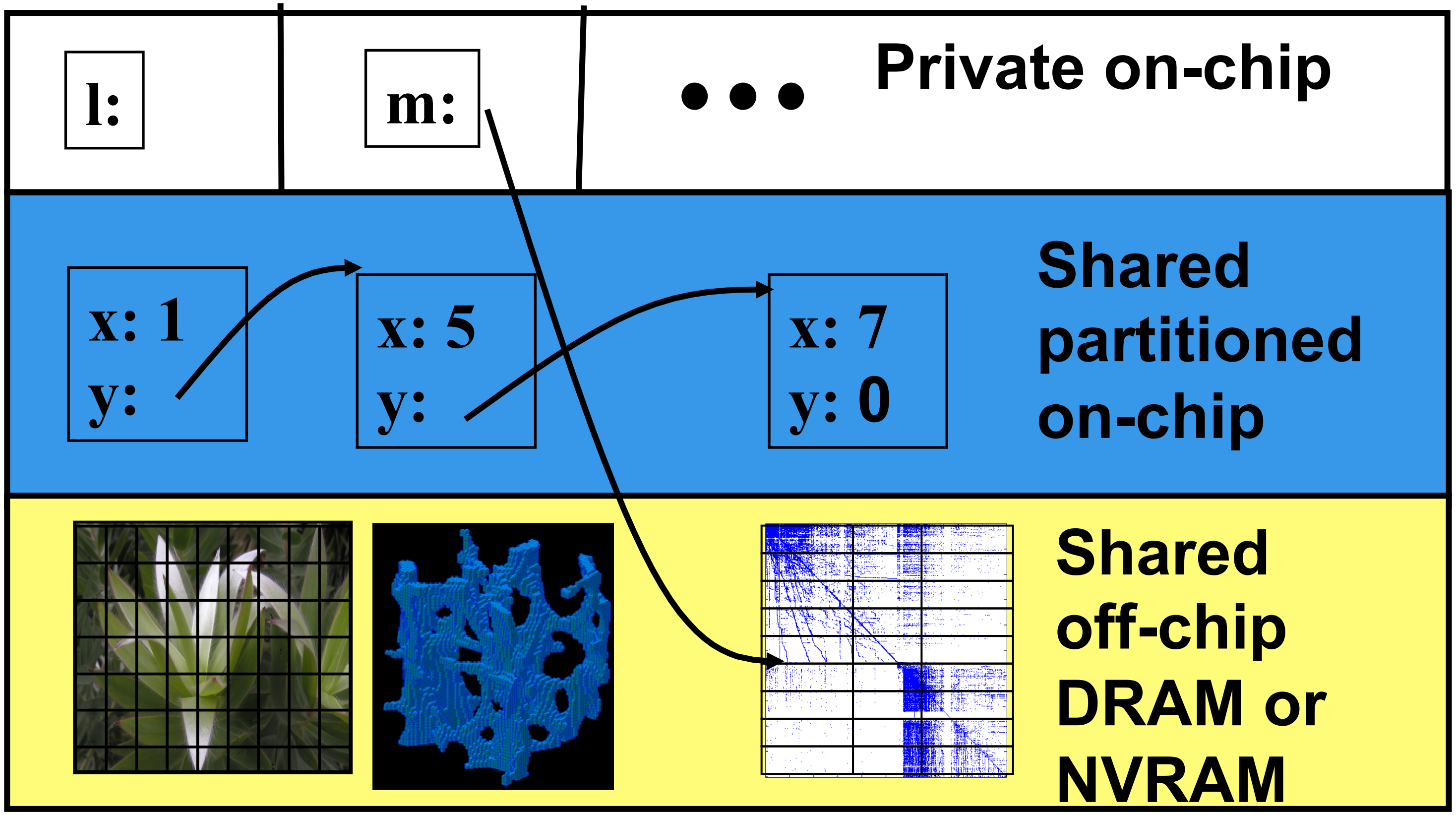
It is implemented using one-sided communication: put/get

Support for one-sided communication (DMA) appears in:

- Fast one-sided network communication (RDMA, Remote DMA)
- Move data to/from accelerators
- Move data to/from I/O system (Flash, disks,..)
- Movement of data in/out of local-store (scratchpad) memory

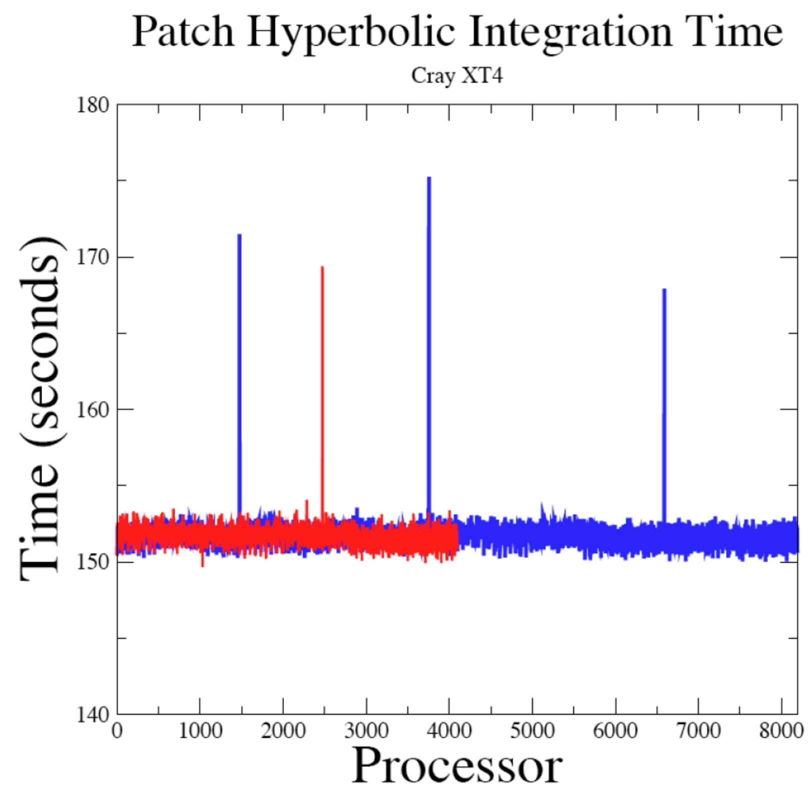
Vertical PGAS

- **New type of wide pointer?**
  - Points to slow (offchip memory)
  - The type system could get unwieldy quickly



Challenge #3: Synchronization is Expensive

- Machines will have Frequent Faults and "Performance Instability"
- Do all applications become "irregular"?
- Locality-Load balance trade-off
  - Most work on dynamic scheduling is inside a shared memory node
  - Largest variability will be between nodes

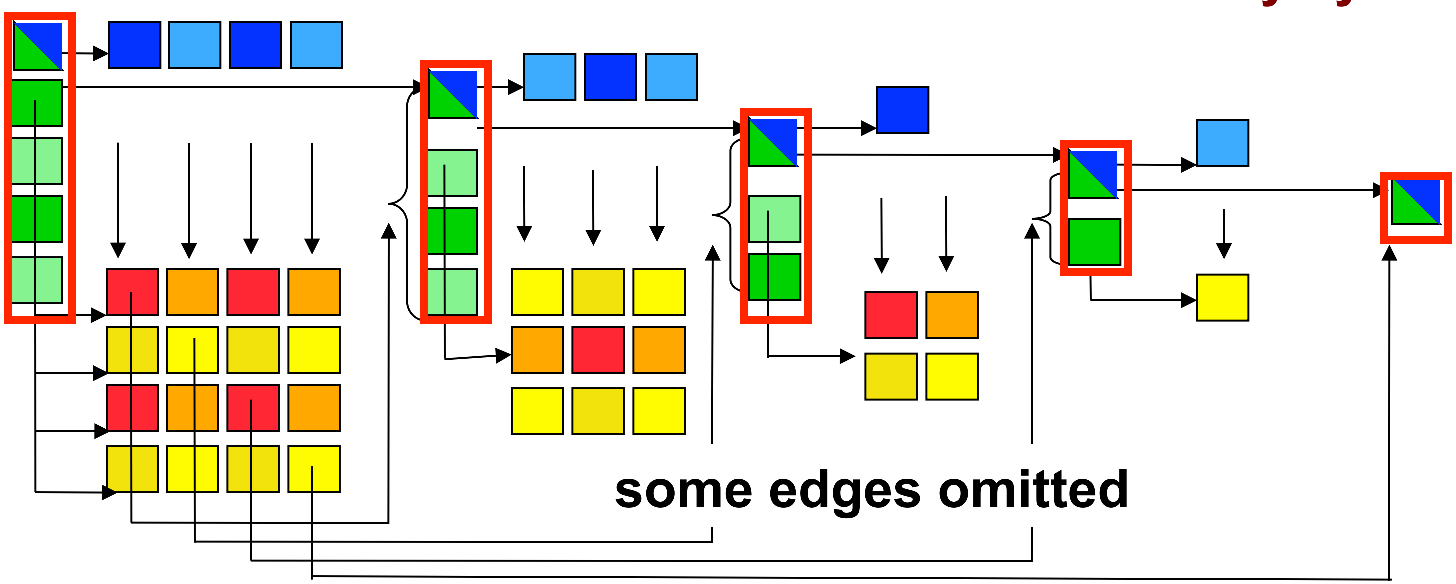


Brian van Straalen, DOE Exascale Research Conference, April 16-18, 2012. *Impact of persistent ECC memory faults.*

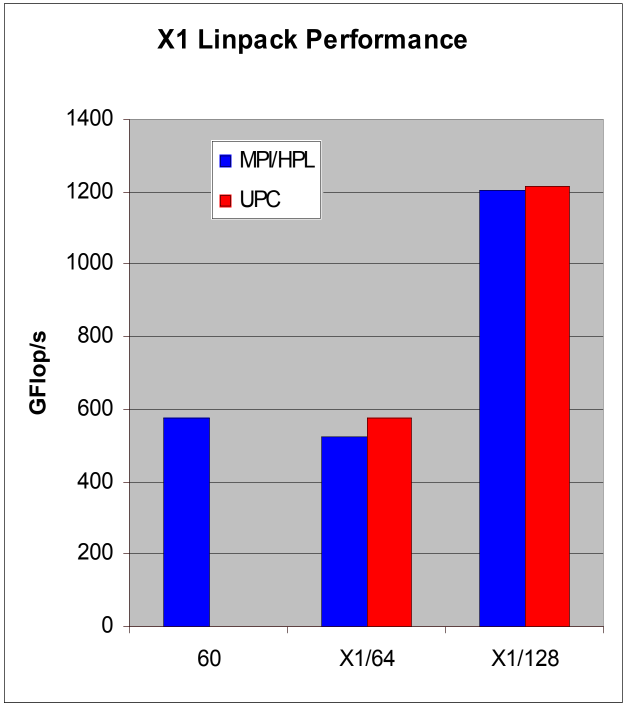
Event Driven LU in UPC

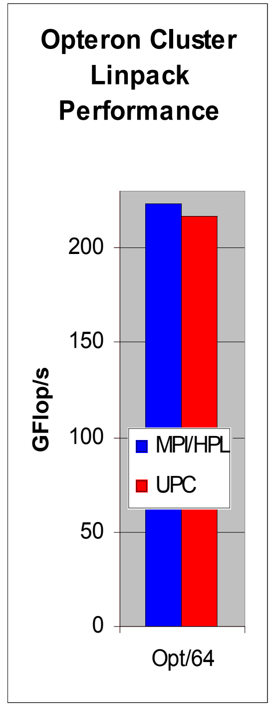
- DAG Scheduling in a distributed (partitioned) memory context
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - “memory constrained” lookahead

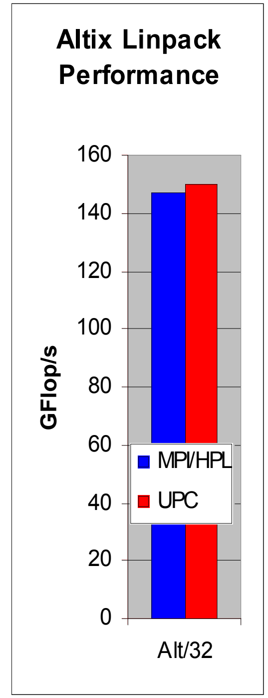
Uses a Berkeley extension to UPC to remotely synchronize



UPC HPL Performance







- **MPI HPL numbers from HPCC database**
- **Large scaling:**
  - 2.2 TFlops on 512p,
  - 4.4 TFlops on 1024p (Thunder)

- Comparison to ScaLAPACK on an Altix, a 2 x 4 process grid
  - ScaLAPACK (block size 64) 25.25 GFlop/s (tried several block sizes)
  - UPC LU (block size 256) - 33.60 GFlop/s, (block size 64) - 26.47 GFlop/s
- n = 32000 on a 4x4 process grid
  - ScaLAPACK - **43.34 GFlop/s** (block size = 64)
  - UPC - **70.26 Gflop/s** (block size = 200)

Joint work with Parry Husbands

Two Distinct Parallel Programming Questions

- What is the parallel control model?

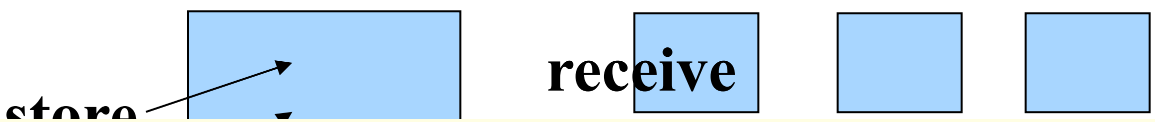
SPMD "default" plus data parallelism through collectives and dynamic tasking within nodes or between nodes through libraries

data parallel
(singe thread of control)

dynamic
threads

single program
multiple data (SPMD)

- What is the model for sharing/communication?



sy

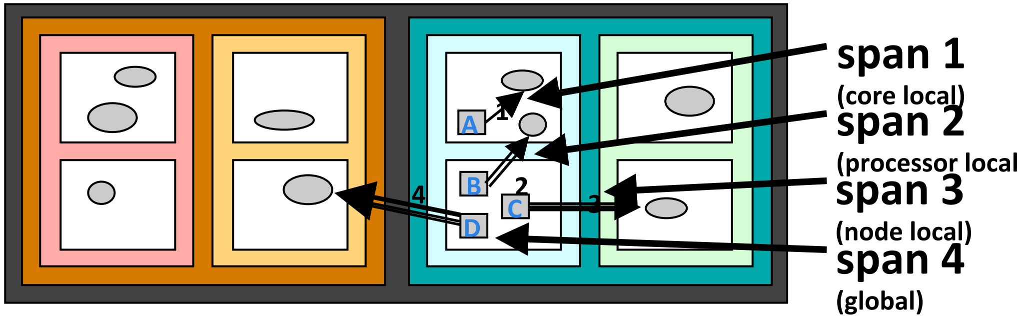
PGAS load/store with partitioning for locality, but need a "signaling store" for producer consumer parallelism

Hierarchical SPMD (demonstrated in Titanium)

- Thread teams may execute distinct tasks

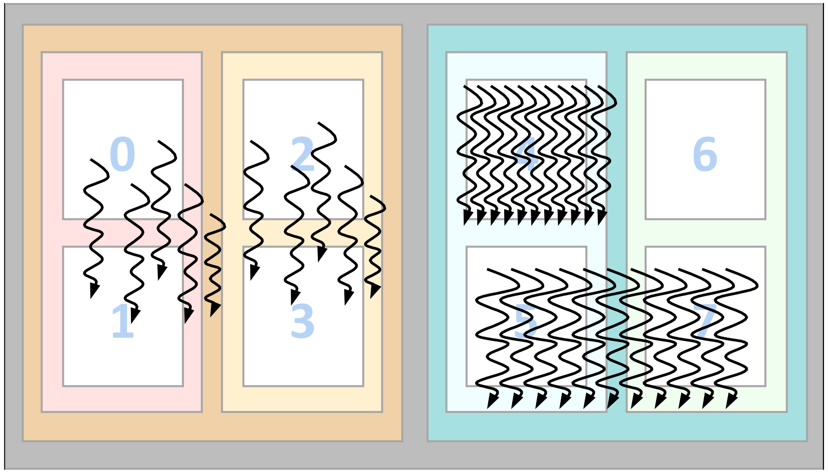
```
partition(T) {
  { model_fluid(); }
  { model_muscles(); }
  { model_electrical(); }
}
```

- Hierarchy for machine / tasks
  - Nearby: access shared data
  - Far away: copy data



- Advantages:
  - Provable pointer types
  - Mixed data / task style
  - Lexical scope prevents some deadlocks

Hierarchical machines → Hierarchical programs



- **Hierarchical memory model may be necessary (what to expose vs hide)**
- **Two approaches to supporting the hierarchical control**

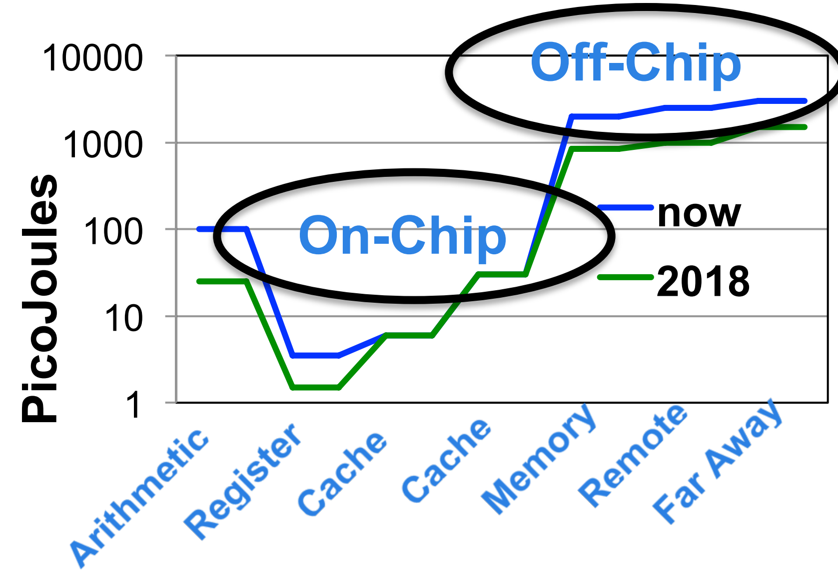
- Option 1: Dynamic parallelism creation
  - Recursively divide until… you run out of work (or hardware)
  - Runtime needs to match parallelism to hardware hierarchy
- Option 2: Hierarchical SPMD with “Mix-ins”
  - Hardware threads can be grouped into units hierarchically
  - Add dynamic parallelism with voluntary tasking on a group
  - Add data parallelism with collectives on a group

Option 1 spreads threads, option 2 collecte them together

Challenge #4: Communication is expensive

Annual improvements			
Flops		BW	Latency
59%	Network	26%	15%
	DRAM	23%	5%

Communication is expensive…
… time and energy



Cost components:

- **Bandwidth: # of words**
- **Latency: # messages**

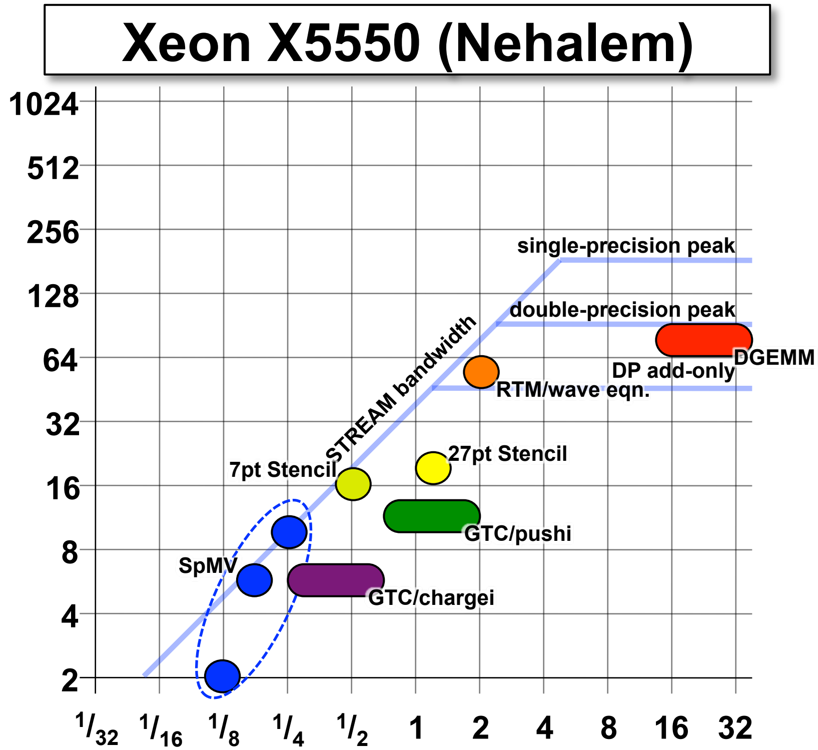
Strategies

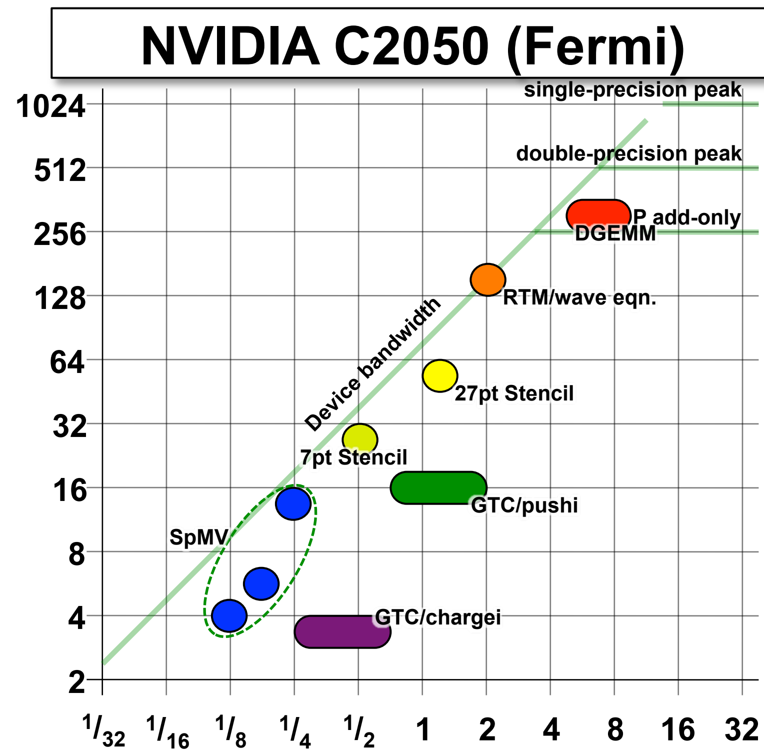
- **Overlap: hide latency**
- **Avoid: algorithms to reduce bandwidth use and number of messages (latency)**

Hard to change: Latency is physics; bandwidth is money!

Autotuning Gets Kernel Performance Near Optimal

- Roofline model captures bandwidth and computation limits
- Autotuning gets kernels near the roof





Work by Williams, Oliker, Shalf, Madduri, Kamil, Im, Ethier,…

Lessons Learned

- **Good news**
  - Although careful tuning is necessary
  - Autotuning helps save programmer time

- **But many kernels are bandwidth limited**
  - Stencils
  - Sparse matrix-vector multiply
  - Dense matrix-vector multiply

- **A problem for local memory and network**

Avoiding Communication in Iterative Solvers

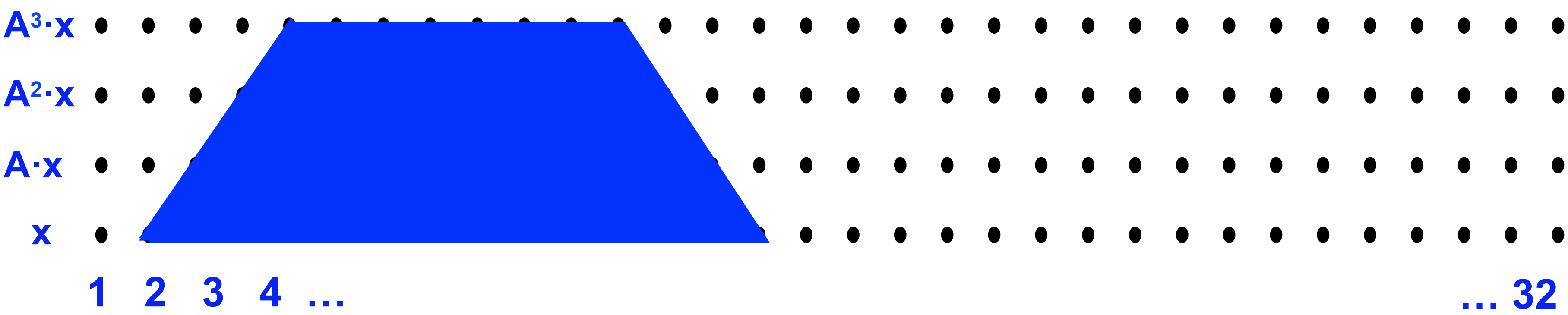
- Consider Sparse Iterative Methods for *Ax*=*b*
  - Krylov Subspace Methods: GMRES, CG,…
- Solve time dominated by:
  - Sparse matrix-vector multiple (SPMV)
    - Which even on one processor is dominated by "communication" time to read the matrix
  - Global collectives (reductions)
    - Global latency-limited
- Can we lower the communication costs?
  - Latency: reduce # messages by computing multiple reductions at once
  - Bandwidth to memory, i.e., compute Ax, A^2x, … A^kx with one read of A

Joint work with Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin

Communication Avoiding Kernels

The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of $y = A \cdot x$ with [Ax, A^2x, …, A^kx]

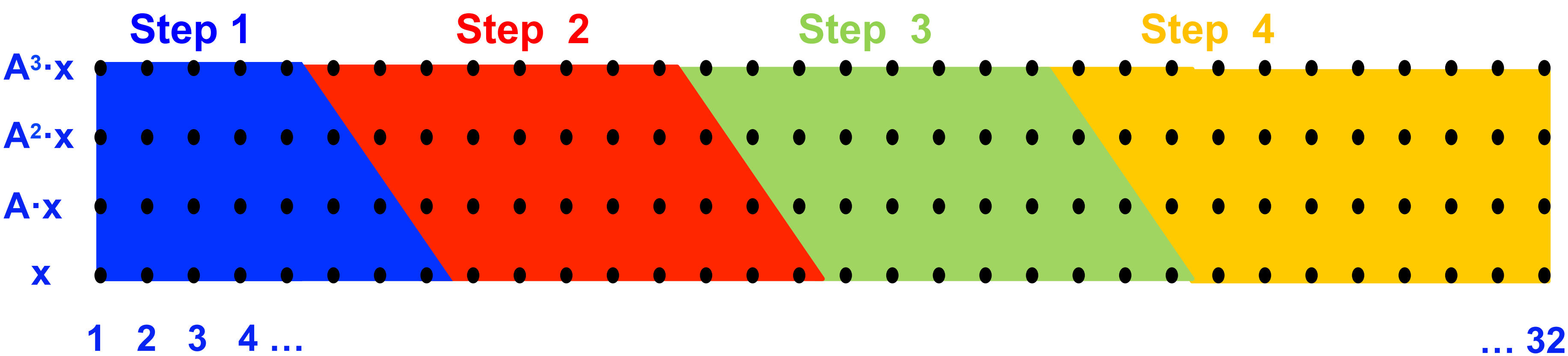


- Idea: pick up part of A and x that fit in fast memory, compute **each of k products**
- Example: A tridiagonal matrix (a 1D “grid”), n=32, k=3
- General idea works for any “well-partitioned” A

Communication Avoiding Kernels (Sequential case)

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- **Sequential Algorithm**

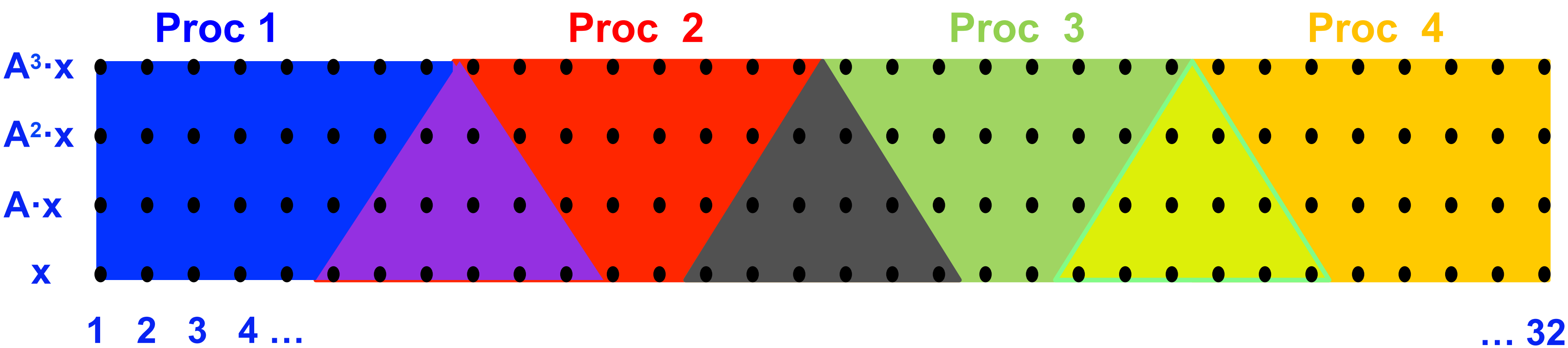


- Example: A tridiagonal, n=32, k=3
- **Saves bandwidth (one read of A&x for k steps)**
- **Saves latency (number of independent read events)**

Communication Avoiding Kernels: (Parallel case)

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

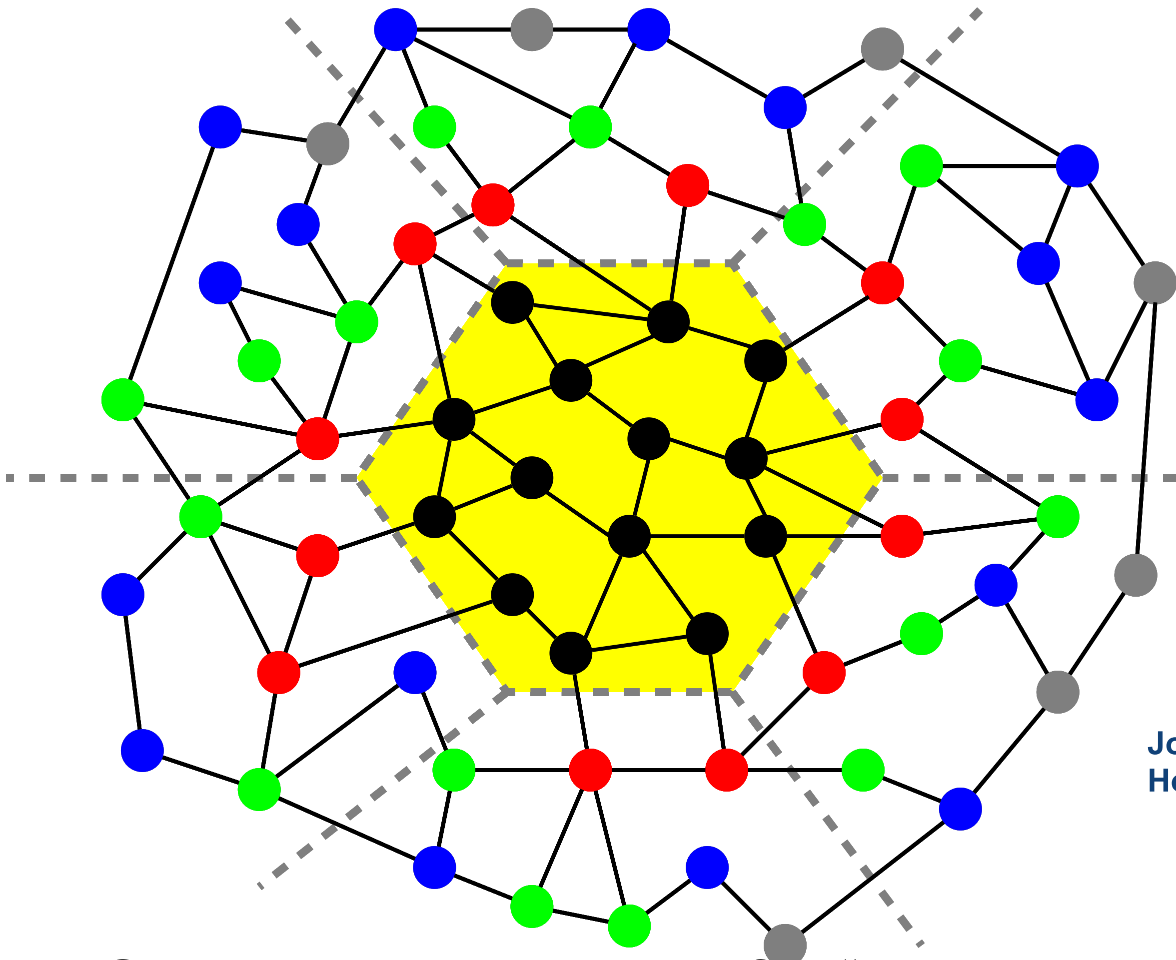
- Replace k iterations of $y = A\cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- **Parallel Algorithm**



- Example: A tridiagonal, n=32, k=3
- **Each processor works on (overlapping) trapezoid**
- Saves latency (# of messages); **Not bandwidth**

  But adds redundant computation

Matrix Powers Kernel on a General Matrix

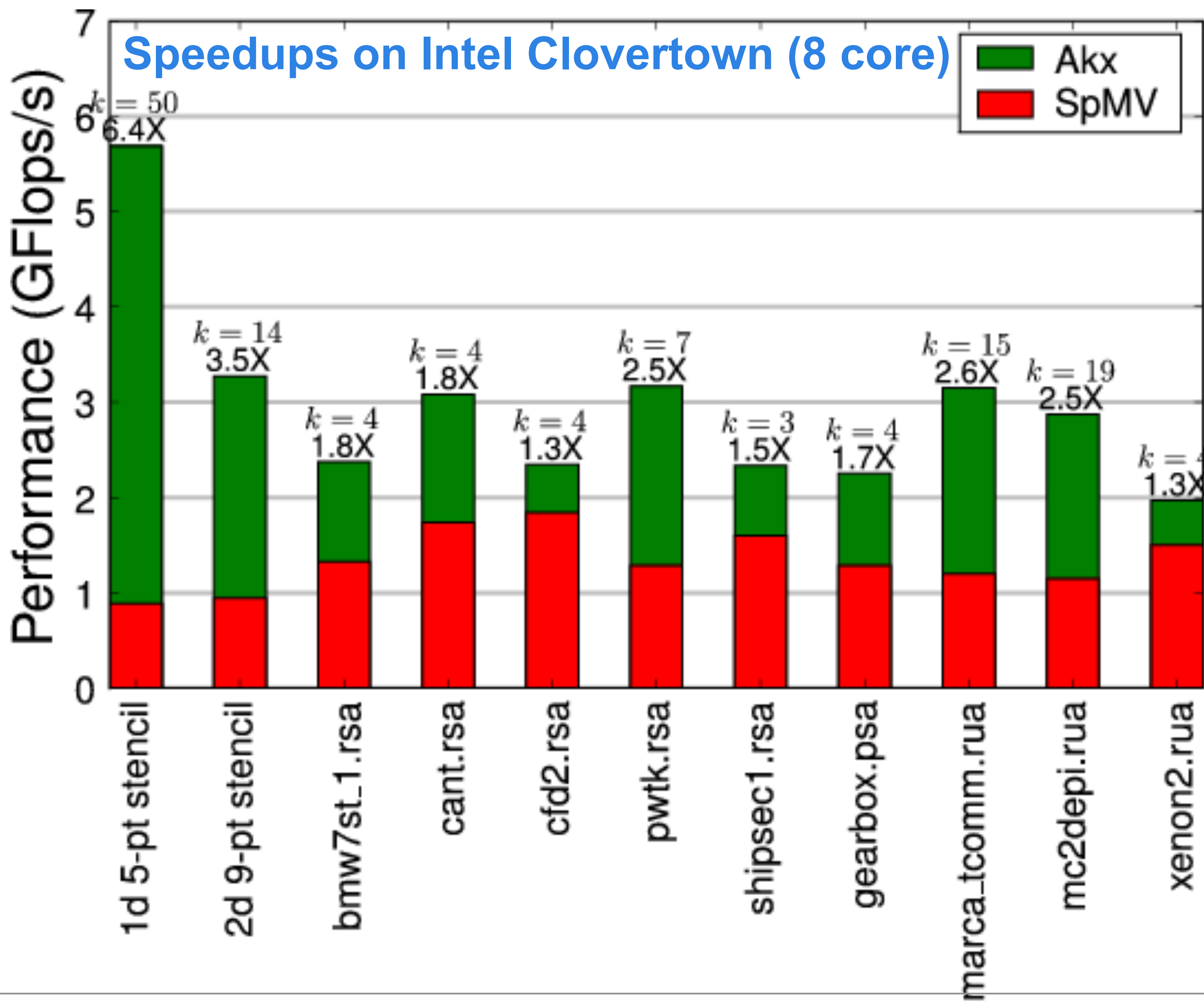


For implicit memory management (caches) uses a TSP algorithm for layout

Joint work with Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin

- Saves communication for "well partitioned" matrices
  - Serial: O(1) moves of data moves vs. O(k)
  - Parallel: O(log p) messages vs. O(k log p)

A^kx has higher performance than Ax



Minimizing Communication of GMRES to solve Ax=b

- **GMRES: find x in span{b,Ab,…,A^kb} minimizing || Ax-b $||_2$**

Standard GMRES

```
for i=1 to k
   w = A · v(i-1)   … SpMV
   MGS(w, v(0),…,v(i-1))
   update v(i), H
endfor
solve LSQ problem with H
```

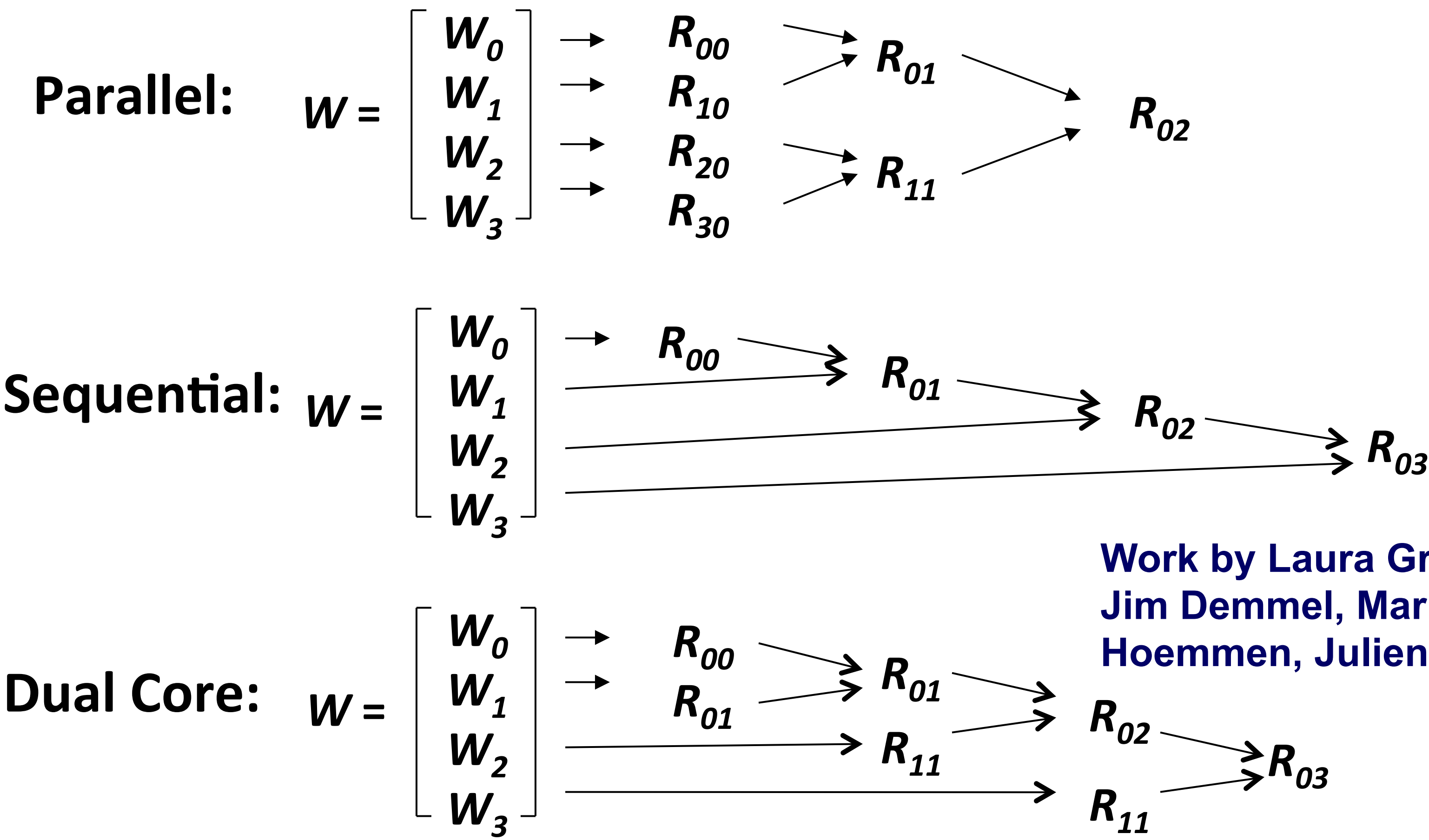
Communication-avoiding GMRES

```
W = [ v, Av, A²v, … , Aᵏv ]
[Q,R] = TSQR(W)
      … "Tall Skinny QR"
build H from R
solve LSQ problem with H
```

Sequential case: #words moved decreases by a factor of k
Parallel case: #messages decreases by a factor of k

- **Oops – W from power method, precision lost!**

TSQR: An Architecture-Dependent Algorithm

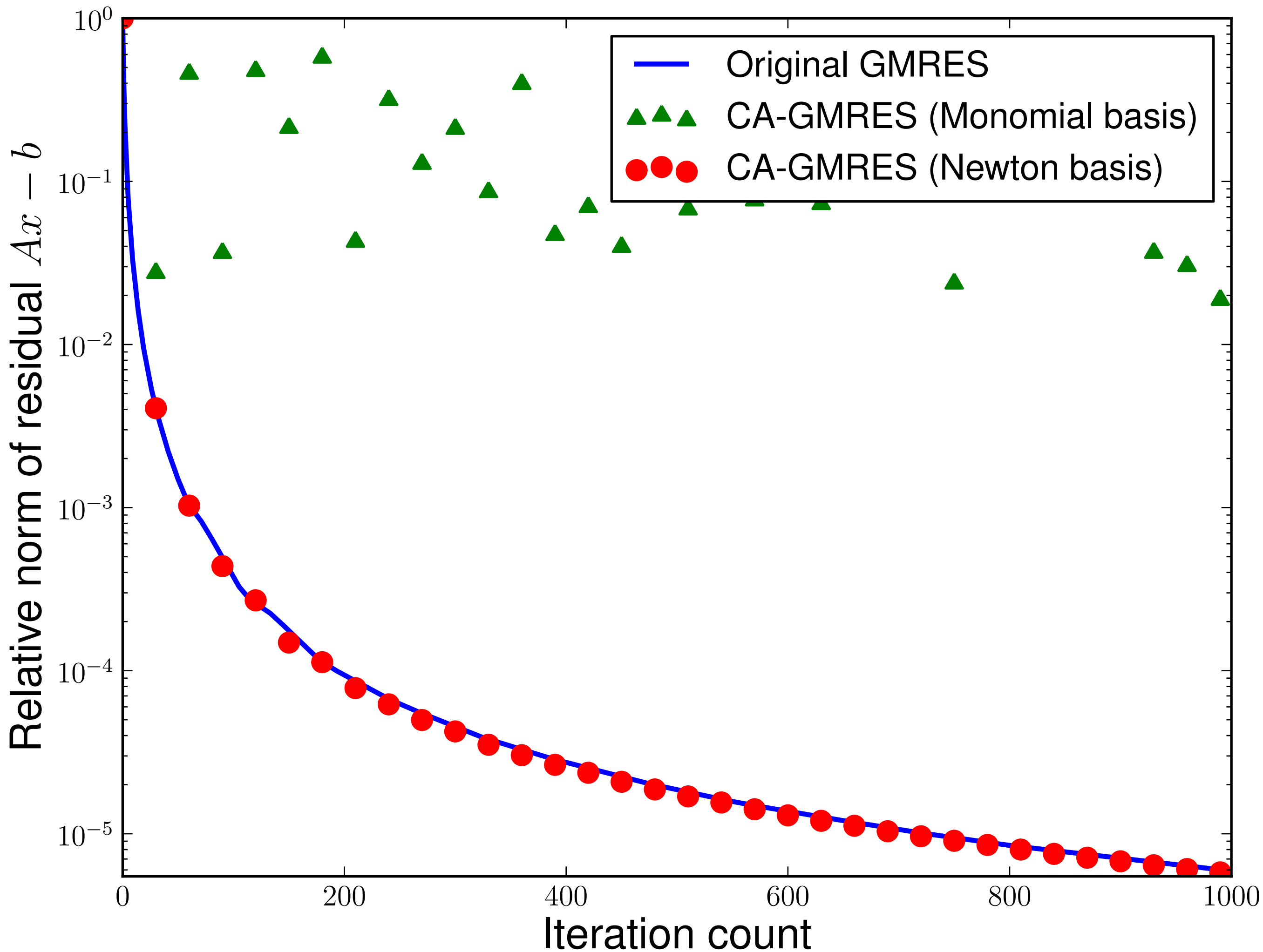


Work by Laura Grigori, Jim Demmel, Mark Hoemmen, Julien Langou

Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

Can choose reduction tree dynamically

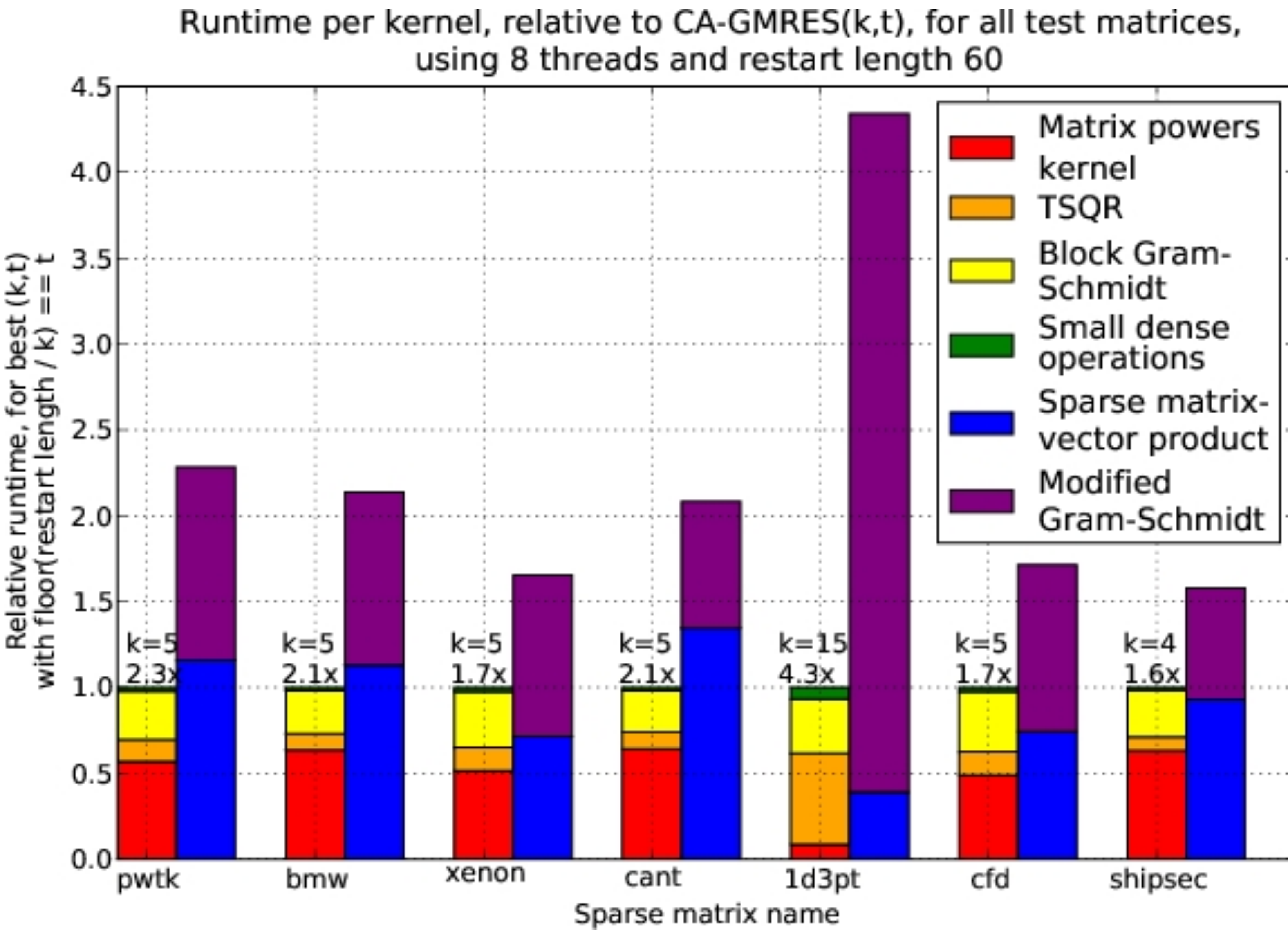
Matrix Powers Kernel (and TSQR) in GMRES



Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Kathy Yelick

Communication-Avoiding Krylov Method (GMRES)

Performance on 8 core Clovertown



Optimality of Communication

Lower bounds, (matching) upper bounds (algorithms) and a question:

Can we train compilers to do this?

See: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-61.pdf

Beyond Domain Decomposition

2.5D Matrix Multiply on BG/P, 16K nodes / 64K cores

c = 16 copies

Matrix multiplication on 16,384 nodes of BG/P

Surprises:

- **Even Matrix Multiply had room for improvement**
- **Idea: make copies of C matrix (as in prior 3D algorithm, but not as many)**
- **Result is provably optimal in communication**

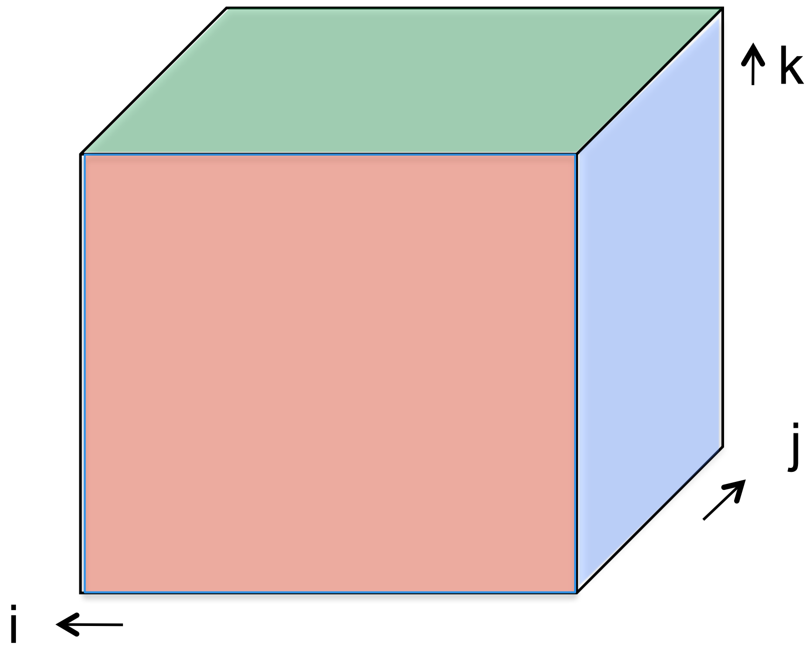
Lesson: Never waste fast memory

Can we generalize for compiler writers?

EuroPar'11 (Solomonik, Demmel)

SC'11 paper (Solomonik, Bhatele, Demmel)

Towards Communication-Avoiding Compilers: Deconstructing 2.5D Matrix Multiply



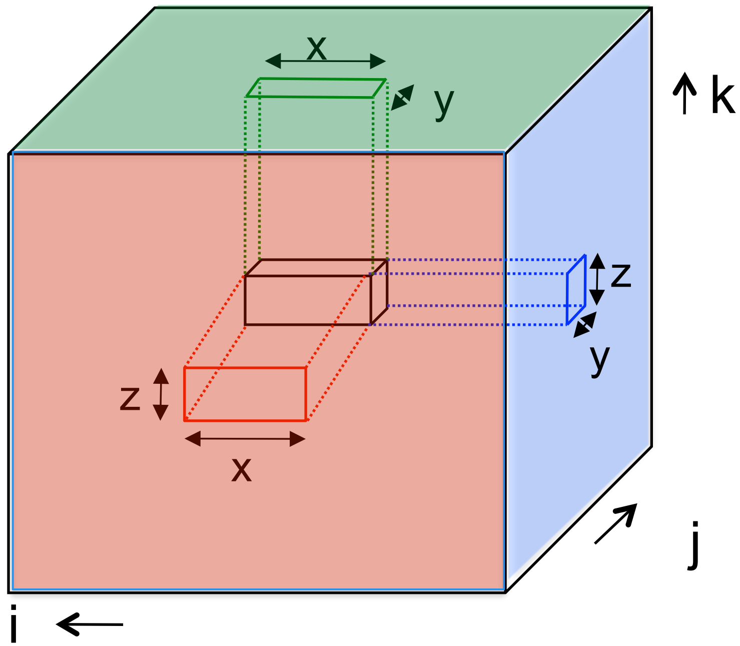
Tiling the iteration space
- **Compute a subcube**
- **Will need data on faces (projection of cube, subarrays)**
- **For s loops in the nest ➔ s dimensional space**
- **For x dimensional arrays, project to x dim space**

Matrix Multiplication code has a 3D iteration space
Each unit cube in the space is a constant computation (*/+)

```
for i
  for j
    for k
      C[i,j] …  A[i,k] …  B[k,j] …
```

Deconstructing 2.5D Matrix Multiply

Solomonik & Demmel



Tiling in the k dimension

- k loop has dependencies because C (on the top) is a Left-Hand-Side variable
  C += ..
- Advantages to tiling in k:
  - More parallelism → Less synchronization
  - Less communication

What happens to these dependencies?

- All dependencies are vertical k dim (updating C matrix)
- Serial case: compute vertical block column in order
- Parallel case:
  - 2D algorithm (and compilers): never chop k dim
  - 2.5 or 3D: Assume + is associative; chop k, which implies replication of C matrix

Beyond Domain Decomposition

x += …

x += …

x += …

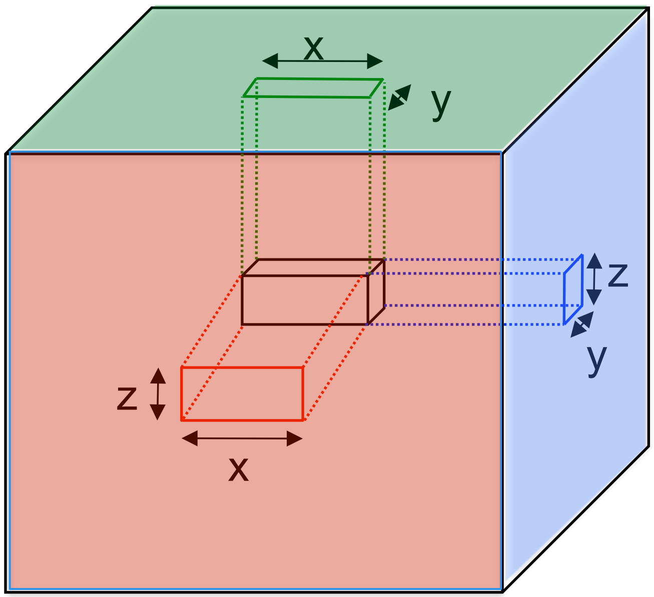
x += …

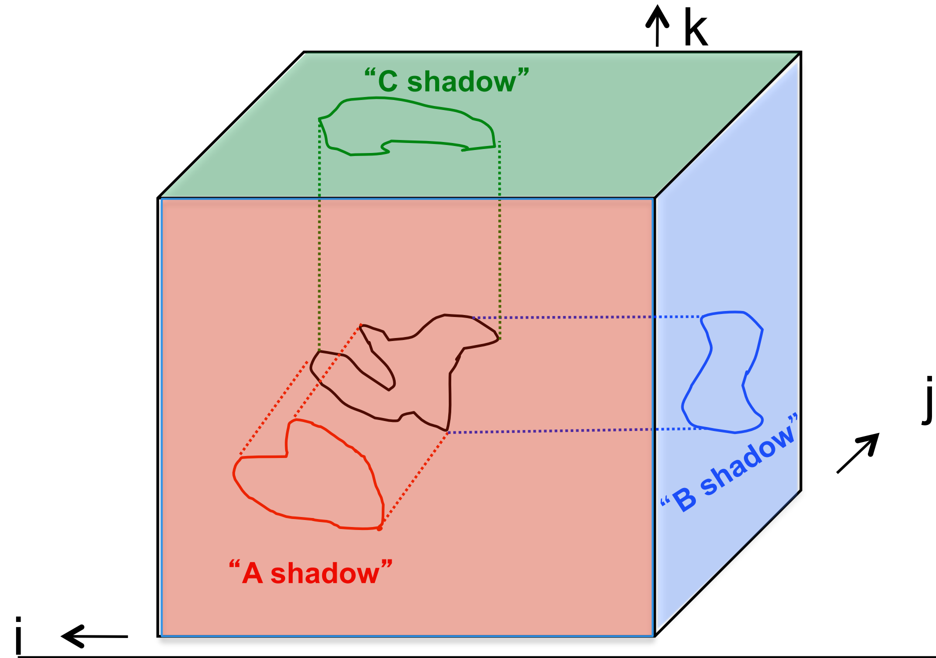
Using x for C[i,j] here

- Much of the work on compilers is based on owner-computes
  - For MM: Divide C into chunks, schedule movement of A/B
  - In this case domain decomposition becomes replication
- Ways to compute C "pencil"
  1. Serially
  2. Parallel reduction ***Standard vectorization trick***
  3. Parallel asynchronous (atomic) updates
  4. Or any hybrid of these
- For what types / operators does this work?
  - "+" is associative for 1,2 rest of RHS is "simple"
  - and commutative for 3

Lower Bound Idea on C = A*B

Iromy, Toledo, Tiskin





"Unit cubes" in black box with side lengths x, y and z

= Volume of black box

= x*y*z

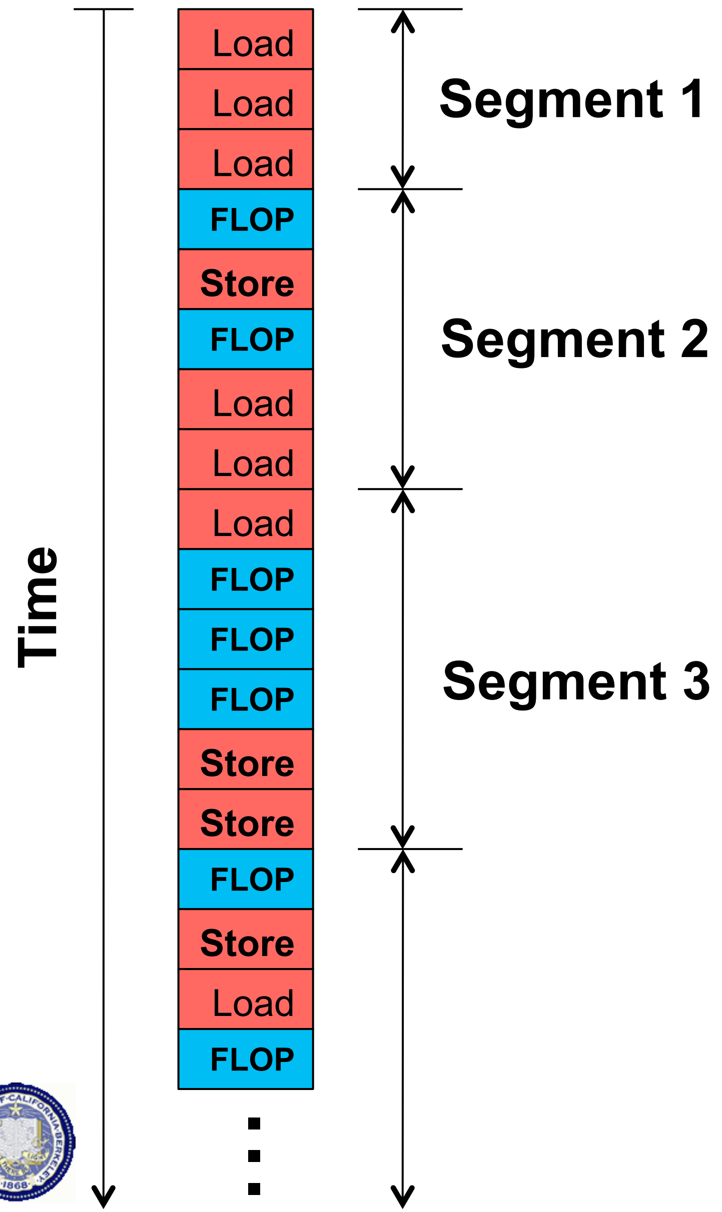
= (#A□s * #B□s * #C□s)$^{1/2}$

= (xz * zy * yx)$^{1/2}$

(i,k) is in "A shadow" if (i,j,k) in 3D set
(j,k) is in "B shadow" if (i,j,k) in 3D set
(i,j) is in "C shadow" if (i,j,k) in 3D set

Thm (Loomis & Whitney, 1949)
  \# cubes in 3D set = Volume of 3D set
  ≤ (area(A shadow) * area(B shadow) * area(C shadow)) $^{1/2}$

Lower Bound: What is the minimum amount of communication required?



- Proof from Irony/Toledo/Tiskin (2004)
- Assume fast memory of size M
- Outline (big-O reasoning):
  - Segment instruction stream, each with M loads/stores
  - Somehow bound the maximum number of flops that can be done in each segment, call it F
  - So F · # segments ≥ T = total flops $= 2 \cdot n^3$, so # segments ≥ T / F
  - So # loads & stores = M · #segments ≥ M · T / F
- How much work (F) can we do with O(M) data?

Recall optimal sequential Matmul

- Naïve code

  for i=1:n, for j=1:n, for k=1:n, C(i,j)+=A(i,k)*B(k,j)

- "Blocked" code

  for i1 = 1:b:n, for j1 = 1:b:n, for k1 = 1:b:n

  $$\left.\begin{array}{l}\text{for i2 = 0:b-1, for j2 = 0:b-1, for k2 = 0:b-1} \\ \quad \text{i=i1+i2, j = j1+j2, k = k1+k2} \\ \quad \text{C(i,j)+=A(i,k)*B(k,j)}\end{array}\right\} \text{b x b matmul}$$

- Thm: Picking $b = M^{1/2}$ attains lower bound:

  #words_moved = $\Omega(n^3/M^{1/2})$

- Where does **1/2** come from? Can we compute these for arbitrary programs?

Generalizing Communication Lower Bounds and Optimal Algorithms

- For serial matmul, we know #words_moved = $\Omega(n^3/M^{1/2})$, attained by tile sizes $M^{1/2} \times M^{1/2}$

- **Thm (Christ,Demmel,Knight,Scanlon,Yelick):** *For any program that "smells like" nested loops, accessing arrays with subscripts that are linear functions of the loop indices, #words_moved = $\Omega(\#\text{iterations}/M^e)$, for some e we can determine*
- **Thm (C/D/K/S/Y): Under some assumptions, we can determine the optimal tiles sizes**
- Long term goal: All compilers should generate communication optimal code from nested loops

New Theorem applied to Matmul

- for i=1:n, for j=1:n, for k=1:n, C(i,j) += A(i,k)*B(k,j)
- Record array indices in matrix Δ

$$\begin{array}{cccc} & i & j & k & \\ \Delta = & \left(\begin{array}{ccc} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{array}\right) & \begin{array}{c} A \\ B \\ C \end{array} \end{array}$$

- Solve LP for $x = [xi,xj,xk]^T$: $\max \mathbf{1}^T x$ s.t. $\Delta\, x \le \mathbf{1}$
  - Result: $x = [1/2, 1/2, 1/2]^T$, $\mathbf{1}^T x = 3/2 = s_{HBL}$

- Thm: #words_moved $= \Omega(n^3/M^{s_{HBL}-1}) = \Omega(n^3/M^{1/2})$
  Attained by block sizes $M^{xi}, M^{xj}, M^{xk} = M^{1/2}, M^{1/2}, M^{1/2}$

New Theorem applied to Direct N-Body

- for i=1:n, for j=1:n, F(i) += force(P(i) , P(j))
- Record array indices in matrix Δ

$$\Delta = \begin{array}{cc} & \begin{array}{cc} i & j \end{array} \\ \begin{array}{c} \\ \\ \\ \end{array} & \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} F \\ P(i) \\ P(j) \end{array}$$

- Solve LP for $x = [xi,xj]^T$: max $\mathbf{1}^T x$ s.t. $\Delta\, x \le \mathbf{1}$
  - Result: $x = [1,1]$, $\mathbf{1}^T x = 2 = s_{HBL}$
- Thm: #words_moved = $\Omega(n^2/M^{s_{HBL}-1}) = \Omega(n^2/M^1)$

  Attained by block sizes $M^{xi}, M^{xj} = M^1, M^1$

New Theorem applied to Random Code

- for i1=1:n, for i2=1:n, … , for i6=1:n

  A1(i1,i3,i6) += func1(A2(i1,i2,i4),A3(i2,i3,i5),A4(i3,i4,i6))

  A5(i2,i6) += func2(A6(i1,i4,i5),A3(i3,i4,i6))

- Record array indices

  in matrix Δ

$$
\Delta = \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}
\begin{array}{cccccc}
i1 & i2 & i3 & i4 & i5 & i6
\end{array}
$$

$$
\Delta = \begin{pmatrix}
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0
\end{pmatrix}
\begin{array}{l}
A1 \\ A2 \\ A3 \\ A3,A4 \\ A5 \\ A6
\end{array}
$$

- Solve LP for $x = [x1,\dots,x7]^T$: max $\mathbf{1}^T x$ s.t. $\Delta\, x \le \mathbf{1}$
  - Result: $x = [2/7,3/7,1/7,2/7,3/7,4/7]$, $\mathbf{1}^T x = 15/7 = s_{HBL}$
- Thm: #words_moved = $\Omega(n^6/M^{S_{HBL}-1}) = \Omega(n^6/M^{8/7})$

  Attained by block sizes $M^{2/7}, M^{3/7}, M^{1/7}, M^{2/7}, M^{3/7}, M^{4/7}$

General Communication Bound

- Given S subset of Z^k, group homomorphisms φ_1, φ_2, …, bound $|S|$ in terms of $|\varphi_1(S)|$, $|\varphi_2(S)|$, … , $|\varphi_m(S)|$
- Def: Hölder-Brascamp-Lieb LP (HBL-LP) for $s_1,\ldots,s_m$:

  for all subgroups $H < Z^k$, $\quad \mathrm{rank}(H) \le \Sigma_j\, s_j{*}\mathrm{rank}(\varphi_j(H))$

- Thm (Christ/Tao/Carbery/Bennett): Given $s_1,\ldots,s_m$

$$|S| \le \Pi_j\, |\varphi_j(S)|^{s_j}$$

- Thm: Given a program with array refs given by φ_j, choose s_j to minimize $s_{HBL} = \Sigma_j\, s_j$ subject to HBL-LP. Then

$$\#\mathrm{words\_moved} = \Omega\,(\#\mathrm{iterations}/M^{s_{HBL}-1})$$

Comments

- Thm: (bad news) HBL-LP reduces to Hilbert's 10^{th} problem over Q (conjectured to be undecidable)
- Thm: (good news) Another LP with same solution is decidable (but expensive, so far)
- Thm: (better news) Easy to write down LP explicitly in many cases of interest (eg all φ_j = {subset of indices})
- Thm: (good news) Easy to approximate, i.e. get upper or lower bounds on s_{HBL}
  - If you miss a constraint, the lower bound may be too large (i.e. s_{HBL} too small) but still worth trying to attain
  - Tarski-decidable to get superset of constraints (may get s_{HBL} too large)

Comments

- Attainability depends on loop dependencies
- Best case: none, or associate operators (matmul, nbody)

- Thm: When all φ_j = {subset of indices}, dual of HBL-LP gives optimal tile sizes:

  HBL-LP: minimize 1^T*s s.t. $s^T*\Delta \geq 1^T$

  Dual-HBL-LP: maximize 1^T*x s.t. $\Delta*x \leq 1$

  Then for sequential algorithm, tile i_j by M^{x_j}
- Ex: Matmul: $s = [\ 1/2\ ,\ 1/2\ ,\ 1/2\ ]^T = x$
- Generality:
  - Extends to unimodular transforms of indices
  - Does not require arrays (as long as the data structures are injective containers)
  - Does not require loops as long as they can model computation

In theory there is no difference between theory and practice, but in practice there is.

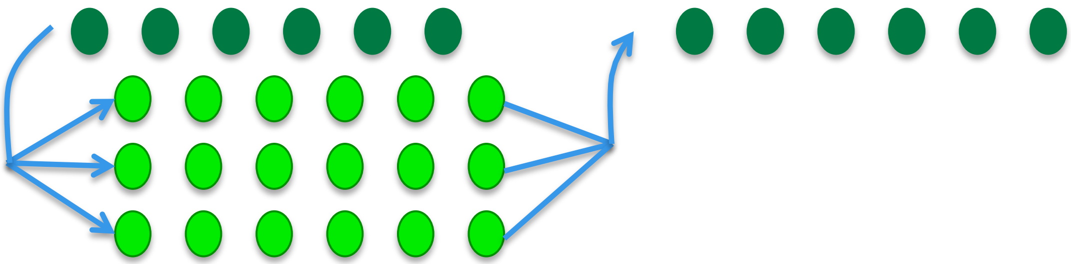
-- Jan L. A. van de Snepscheut, Computer Scientist

or

-- Yogi Berra, Baseball player and manager

Generalizing Communication Optimal Transformations to Arbitrary Loop Nests

1.5D N-Body: Replicate and Reduce



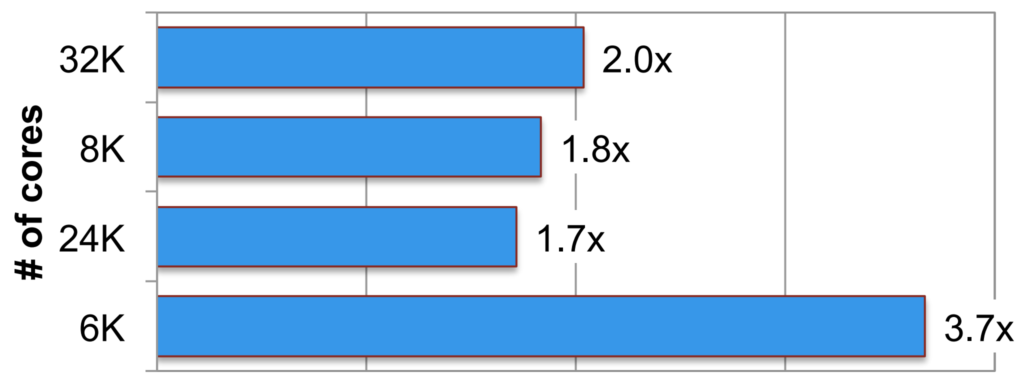
The same idea (replicate and reduce) can be used on (direct) N-Body code:

1D decomposition → "1.5D"

Does this work in general?

- ***Yes, for certain loops and array expressions***
- ***Relies on basic result in group theory***
- ***Compiler work TBD***

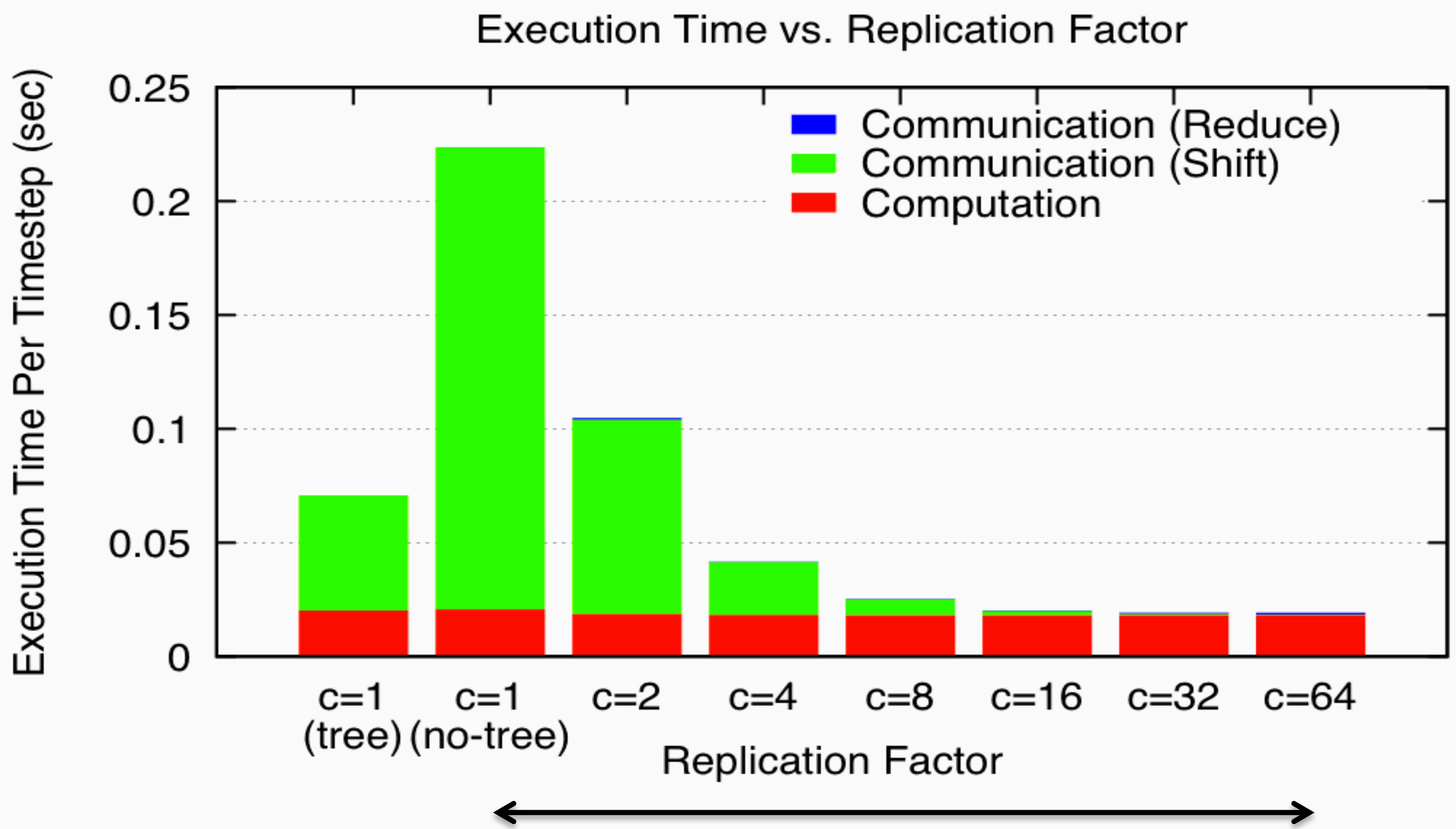
Speedup of 1.5D N-Body over 1D



IPDPS'13 paper (Driscoll, Georganas, Koanantakool, Solomonik, Yelick)

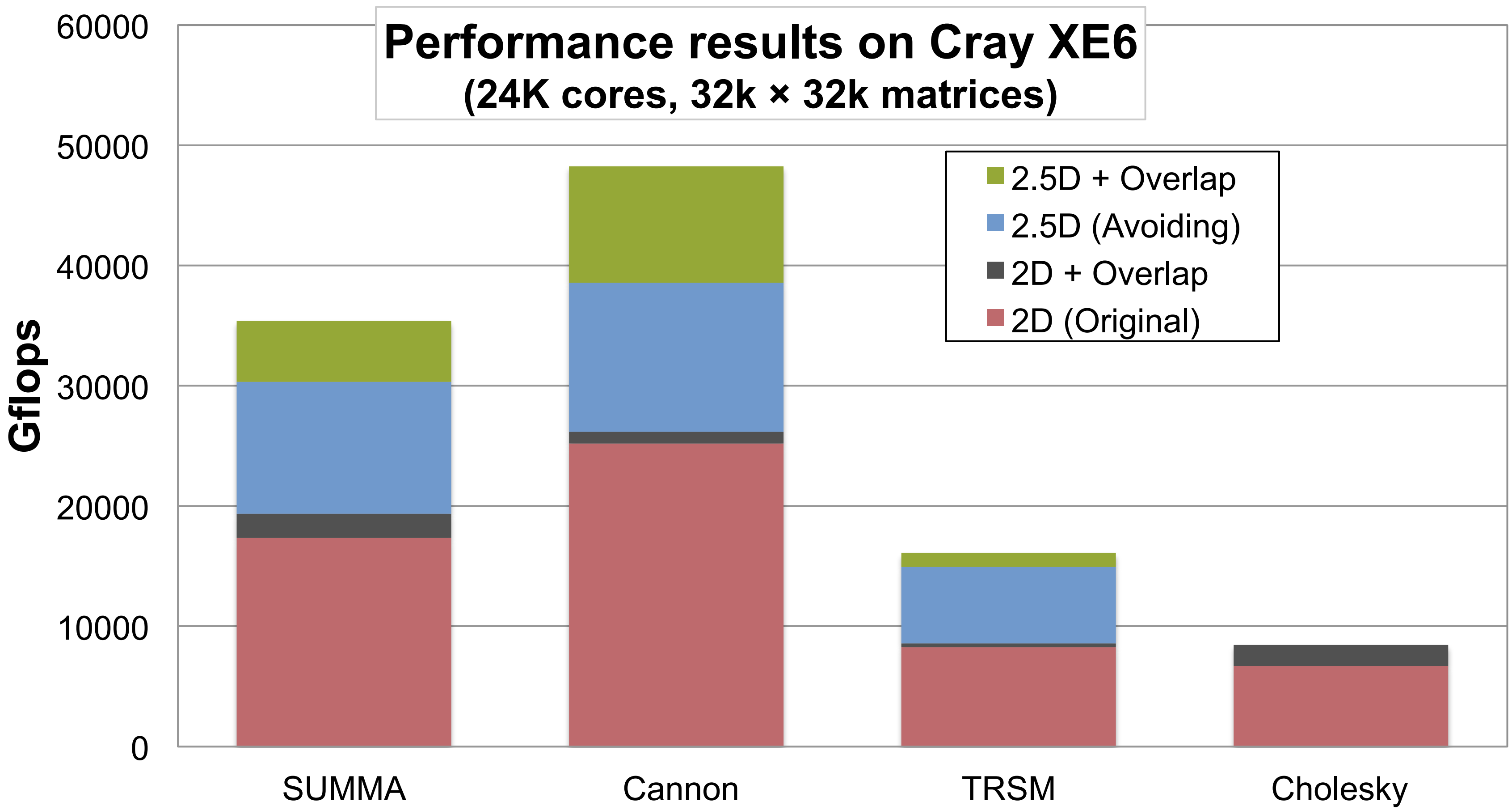
N-Body Speedups on IBM-BG/P (Intrepid)
8K cores, 32K particles

K. Yelick, E. Georganas, M. Driscoll, P. Koanantakool, E. Solomonik



11.8x speedup

Communication Overlap Complements Avoidance



- Even with communication-optimal algorithms (minimized bandwidth) there are still benefits to overlap and other things that speed up networks
- *Communication Avoiding and Overlapping for Numerical Linear Algebra*, Georganas et al, SC12

Stepping Back

- Communication avoidance as old at tiling
- Communication optimality as old as Hong/Kung
- What's new?
  - Raising the level of abstraction at which we optimize
  - BLAS2 → BLAS3 → LU or SPMV/DOT → Krylov
  - Changing numerics in non-trivial ways
  - Rethinking methods to models
- Communication and synchronization avoidance
- Software engineering: breaking abstraction
- Compilers: inter-procedural optimizations

Are there Exascale Algorithms?

Yes, but when you're worrying about

- Scaling
- Synchronization,
- Dynamic system behavior
- Irregular algorithms
- Resilience

don't forget what's important

Location, Location, Location