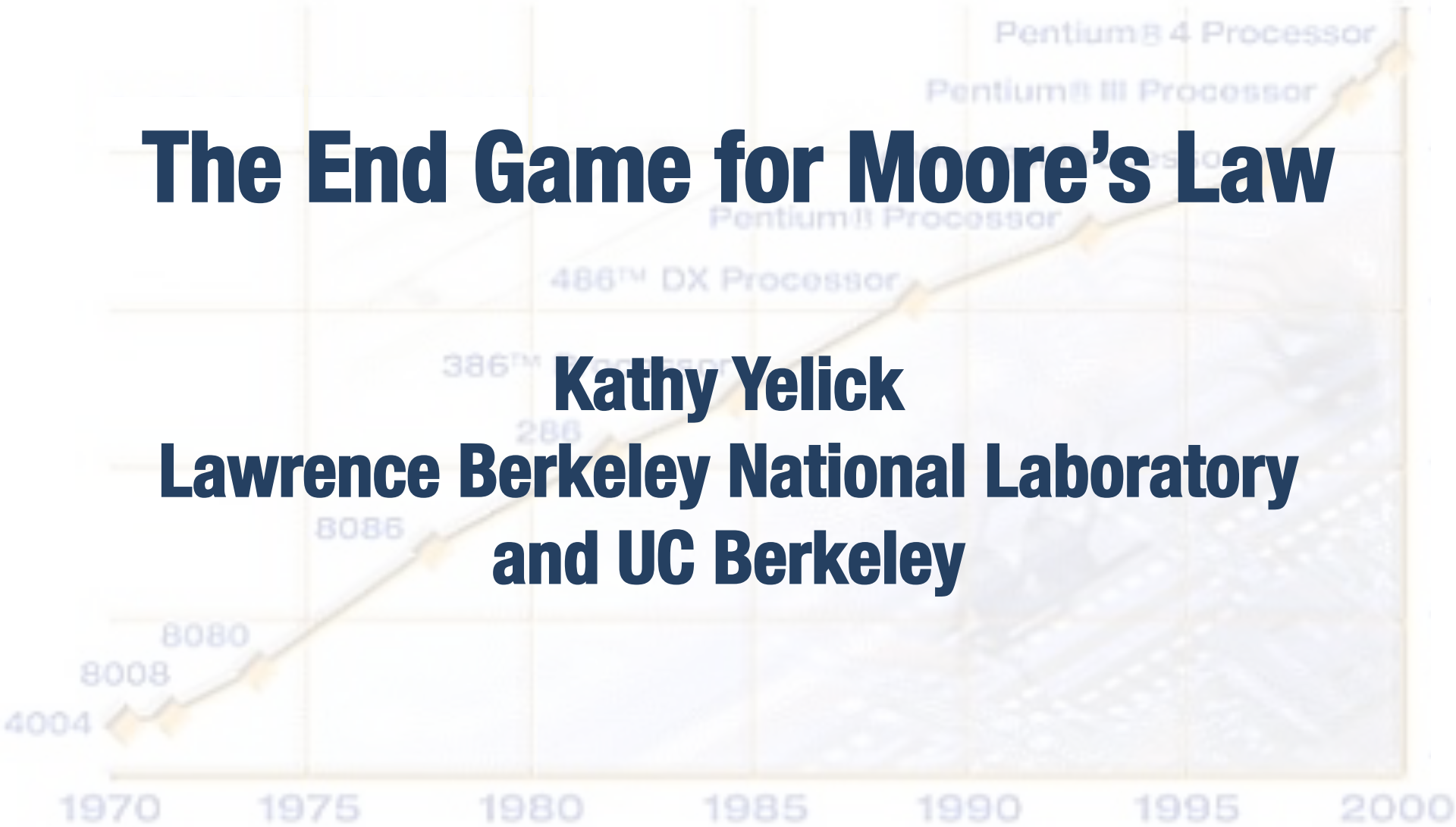


The End Game for Moore's Law

Kathy Yelick

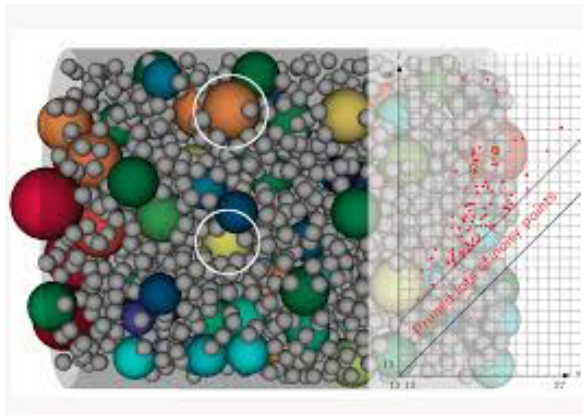
**Lawrence Berkeley National Laboratory
and UC Berkeley**



A Focus on Science



The changing nature of scientific discovery



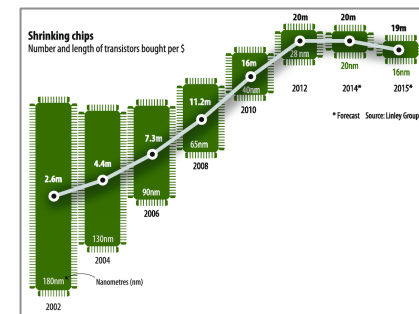
New methods for feature identification and data discovery



Science at the boundary of simulation and observation

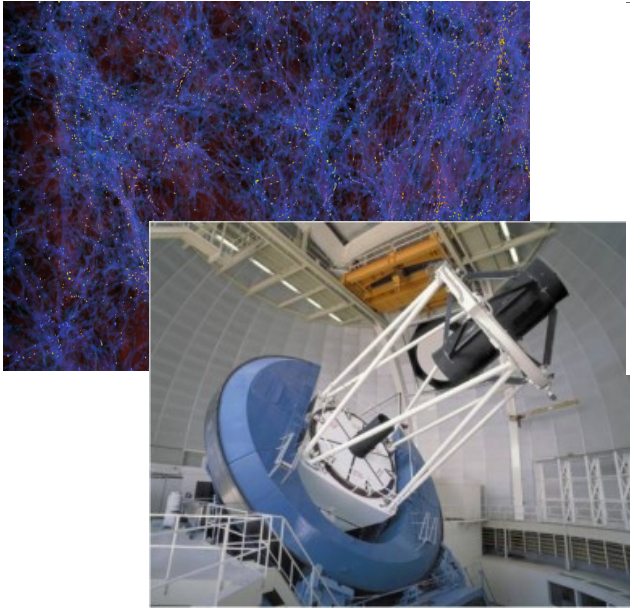


Automation, robotics and new input devices

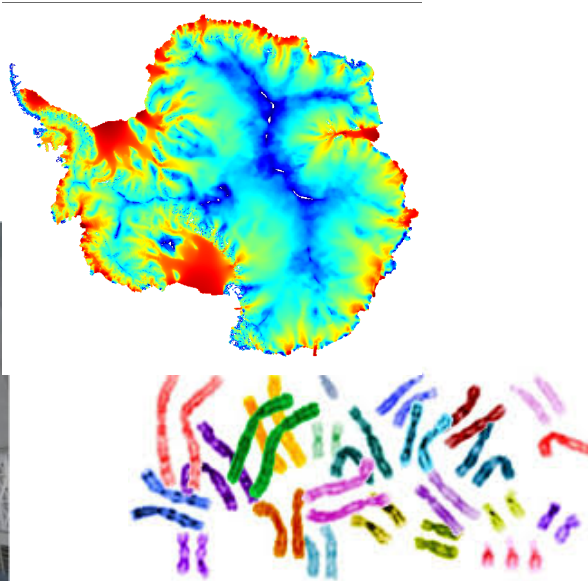


More computing for more complex science questions

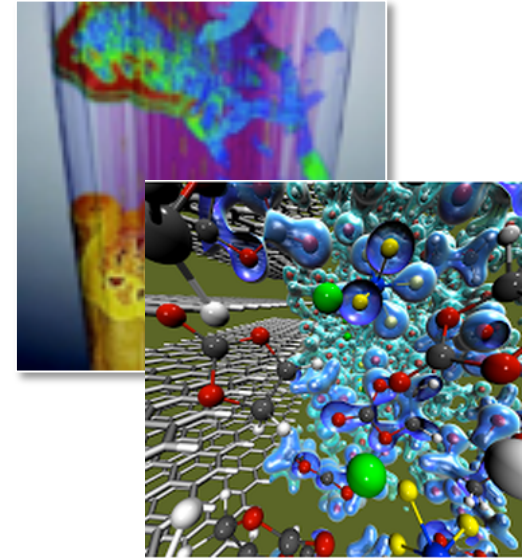
Science at the Boundary of Simulation and Observation



Cosmology



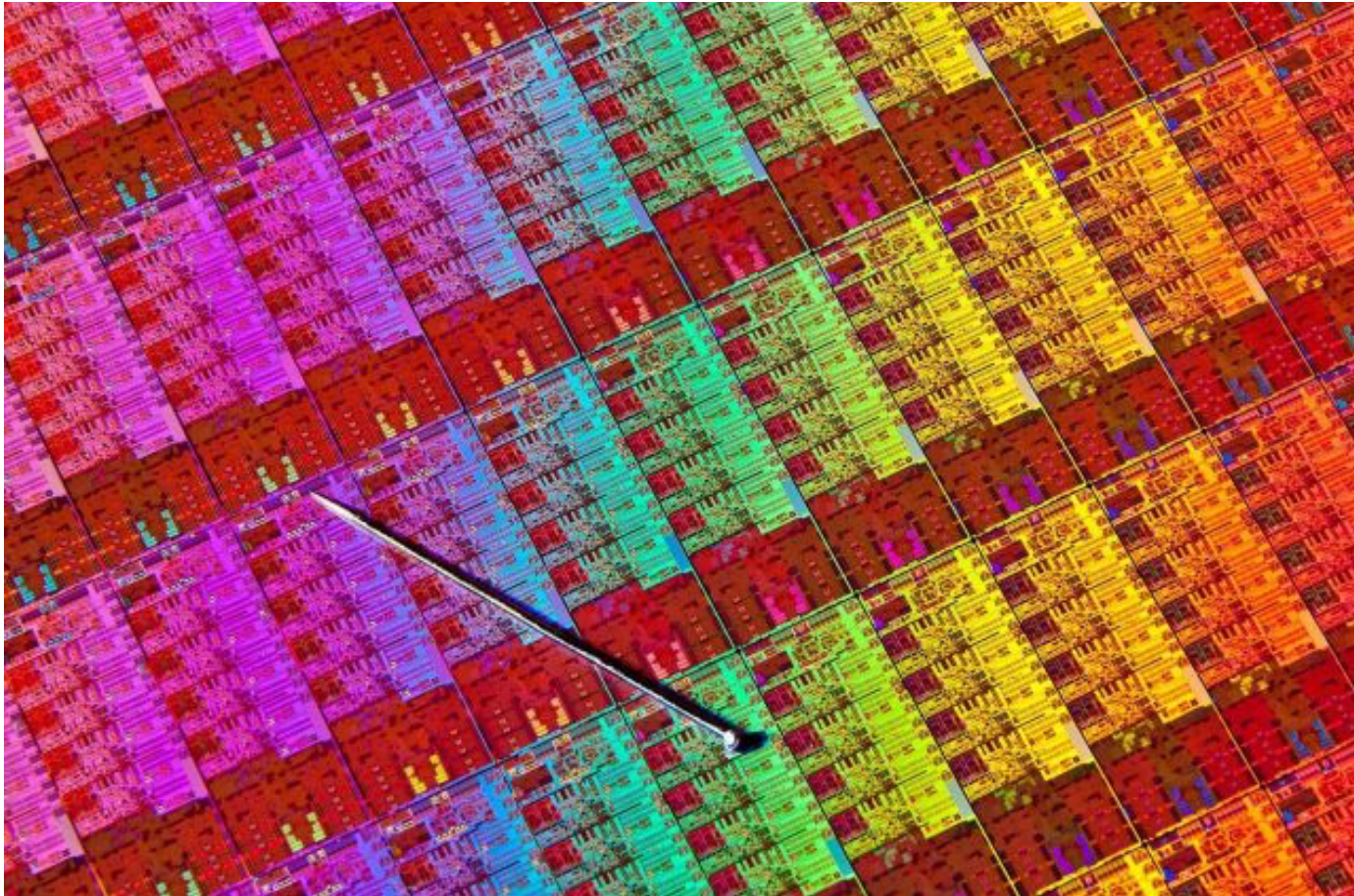
Environment



Materials

In many areas, there are opportunities to combine simulation and observation for new discoveries.

End of Transistor Density Scaling



- **ITRS now sets the end of transistor shrinking to the year 2021**

Technology Scaling Trends

The many ends of "Moore's" Law

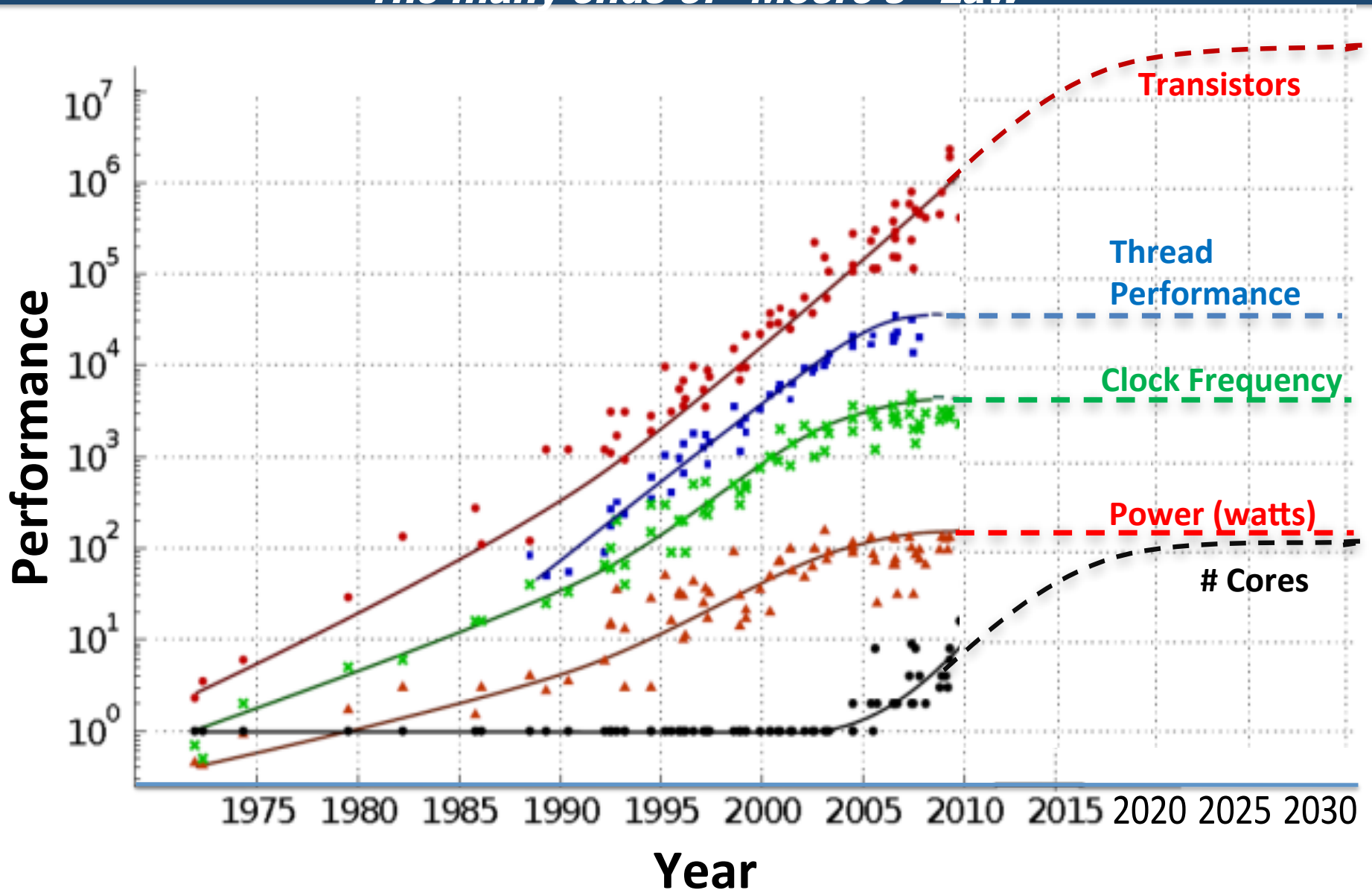


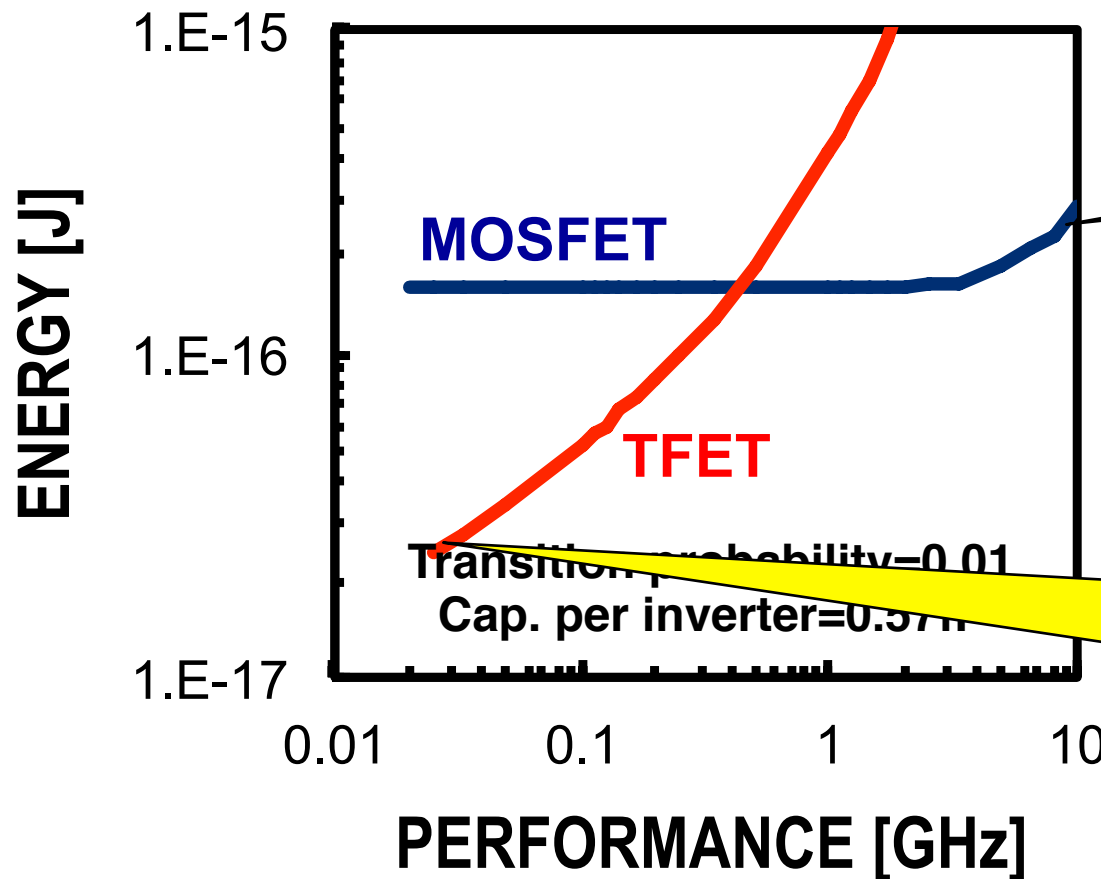
Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Energy Optimization: Alternatives to Conventional MOS

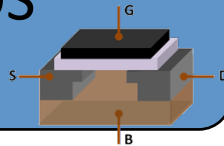
(all require lower clock rate, and much more parallelism)

Energy-Performance Comparison

(30-stage fanout-4 inverter chains)



Today's CMOS Technology



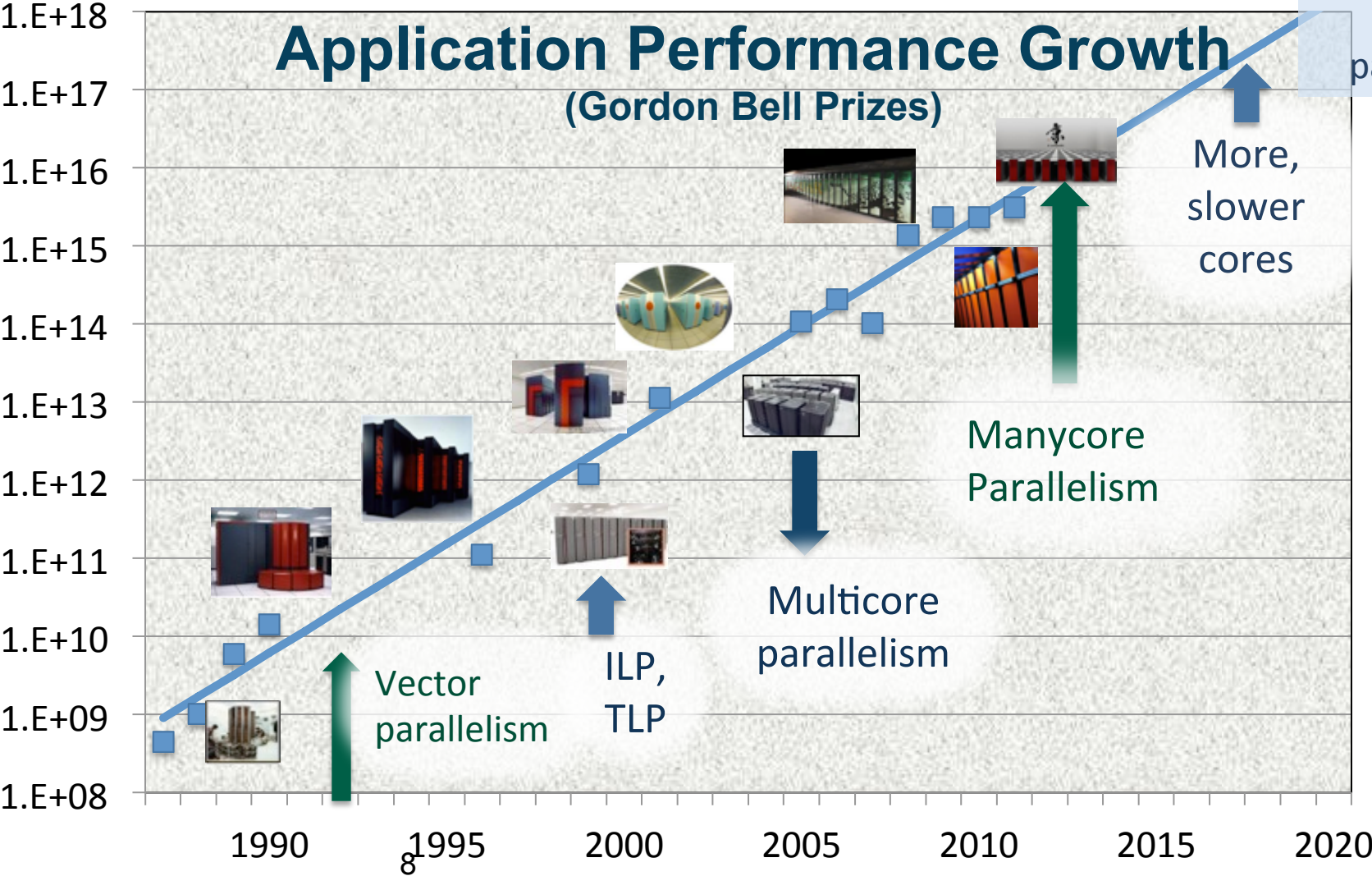
Tunneling FET advantage *only at low clock rates*

More Parallelism at Lower Levels

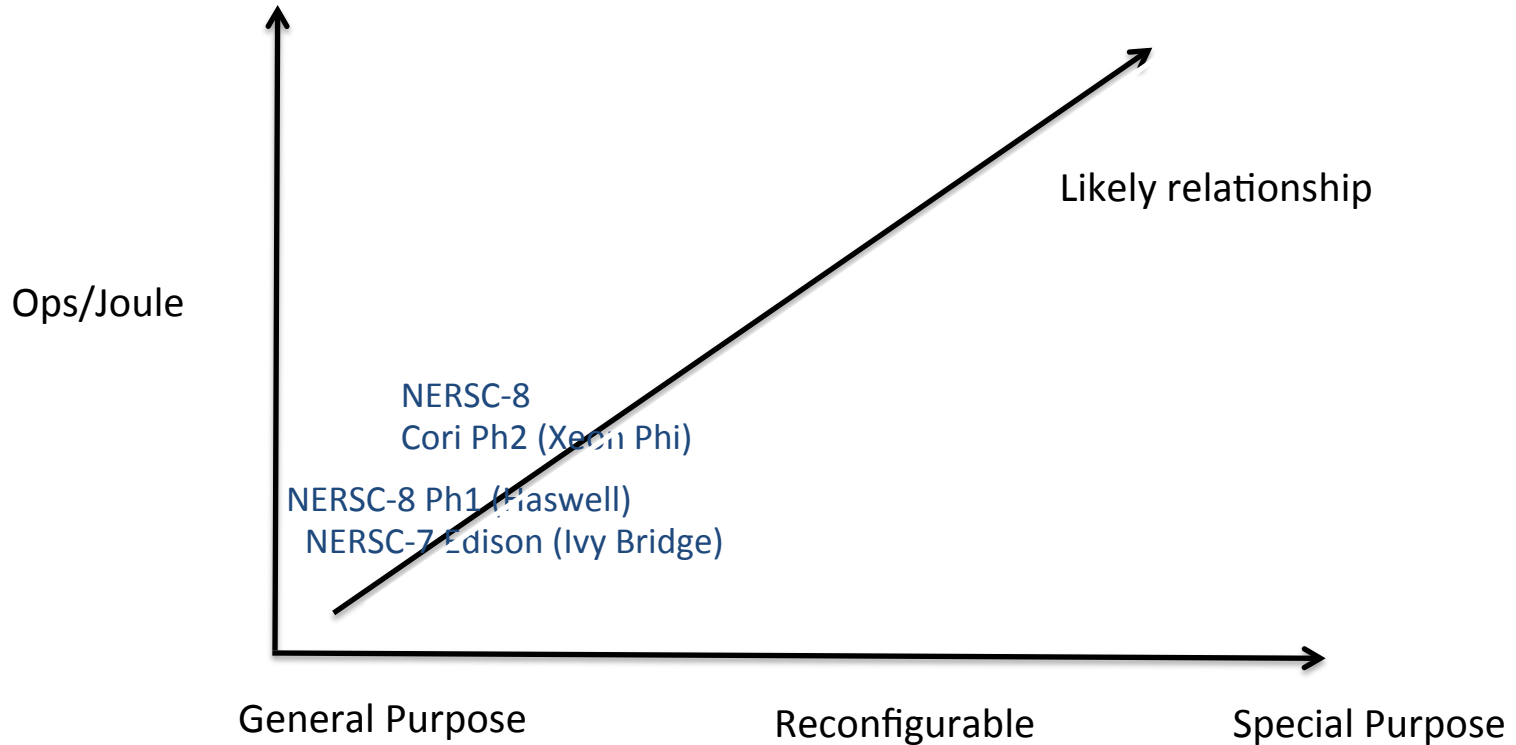
Application Performance Growth

(Gordon Bell Prizes)

More, slower devices; new levels of parallelism



Specialization: End Game for Moore's Law



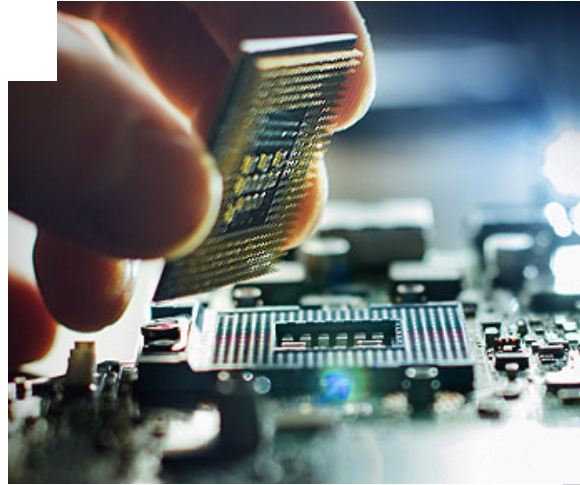
Not just for HPC

Specialization in Deep Learning

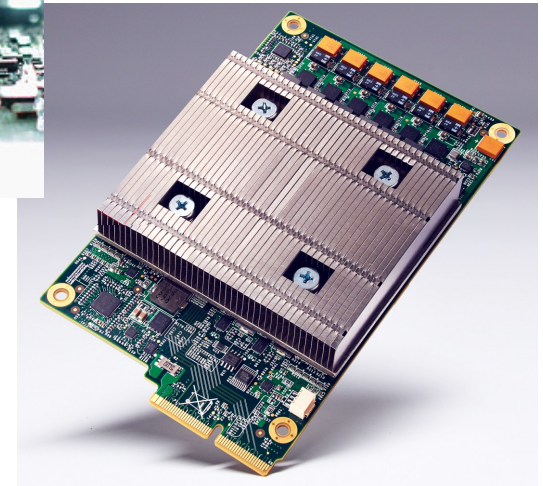


Bandwidth, low precision flops, fixed point,...

NVIDIA builds deep learning appliance with P100 Tesla's



Intel buys deep learning startup, Nervana



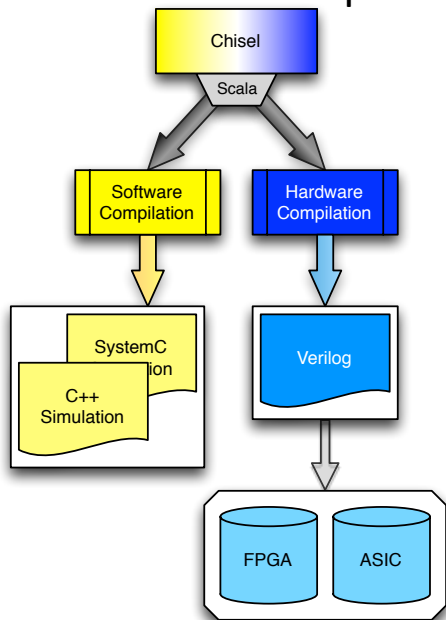
Google designs its own Tensor Processing Unit (TPU)

**Can we use their special purpose systems?
Can we design our own?**

Open Hardware (Synthesis & Simulation)

Chisel

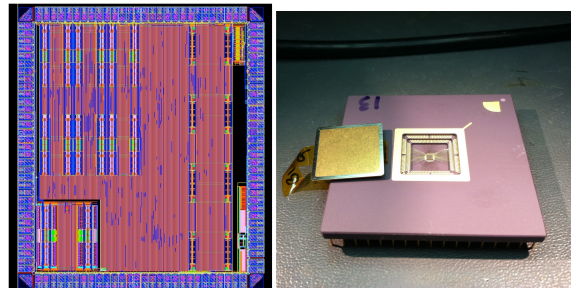
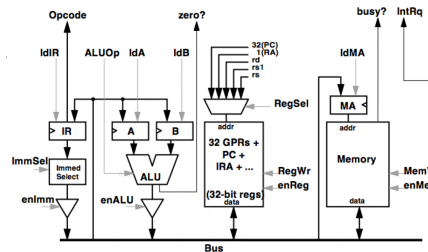
DSL for rapid prototyping of circuits, systems, and arch simulator components



Back-end to synthesize HW with different devices Or new logic families

RISC-V

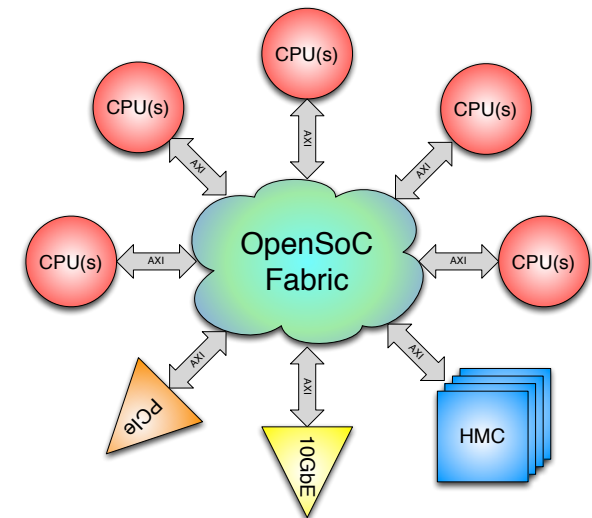
Open Source Extensible ISA/ Cores



Re-implement processor With different devices or Extend w/accelerators

OpenSOC

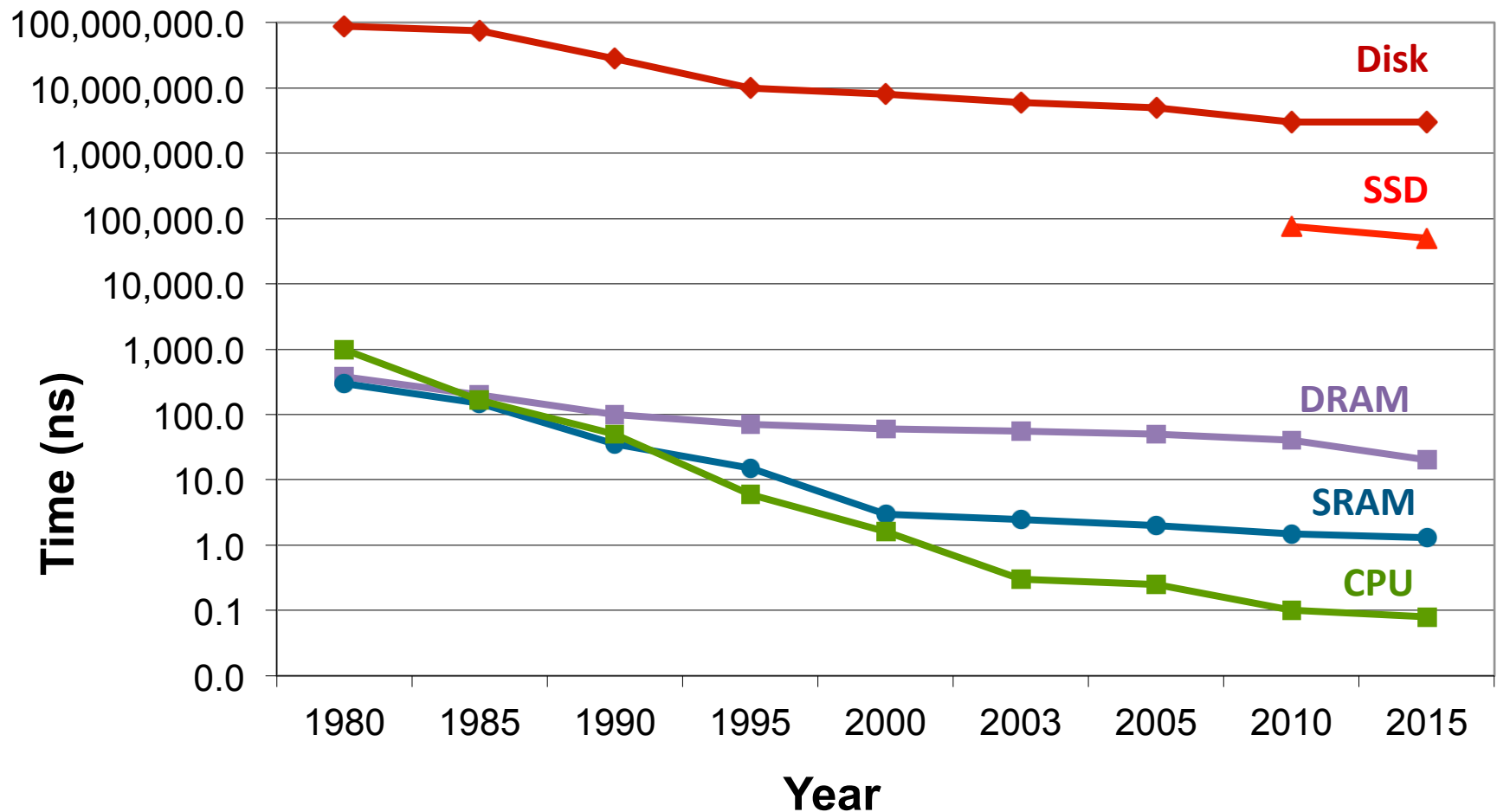
Open Source fabric To integrate accelerators And logic into SOC



Platform for experimentation with specialization to extend Moore's Law

Data Movement is Expensive

CPU cycle time vs memory access time

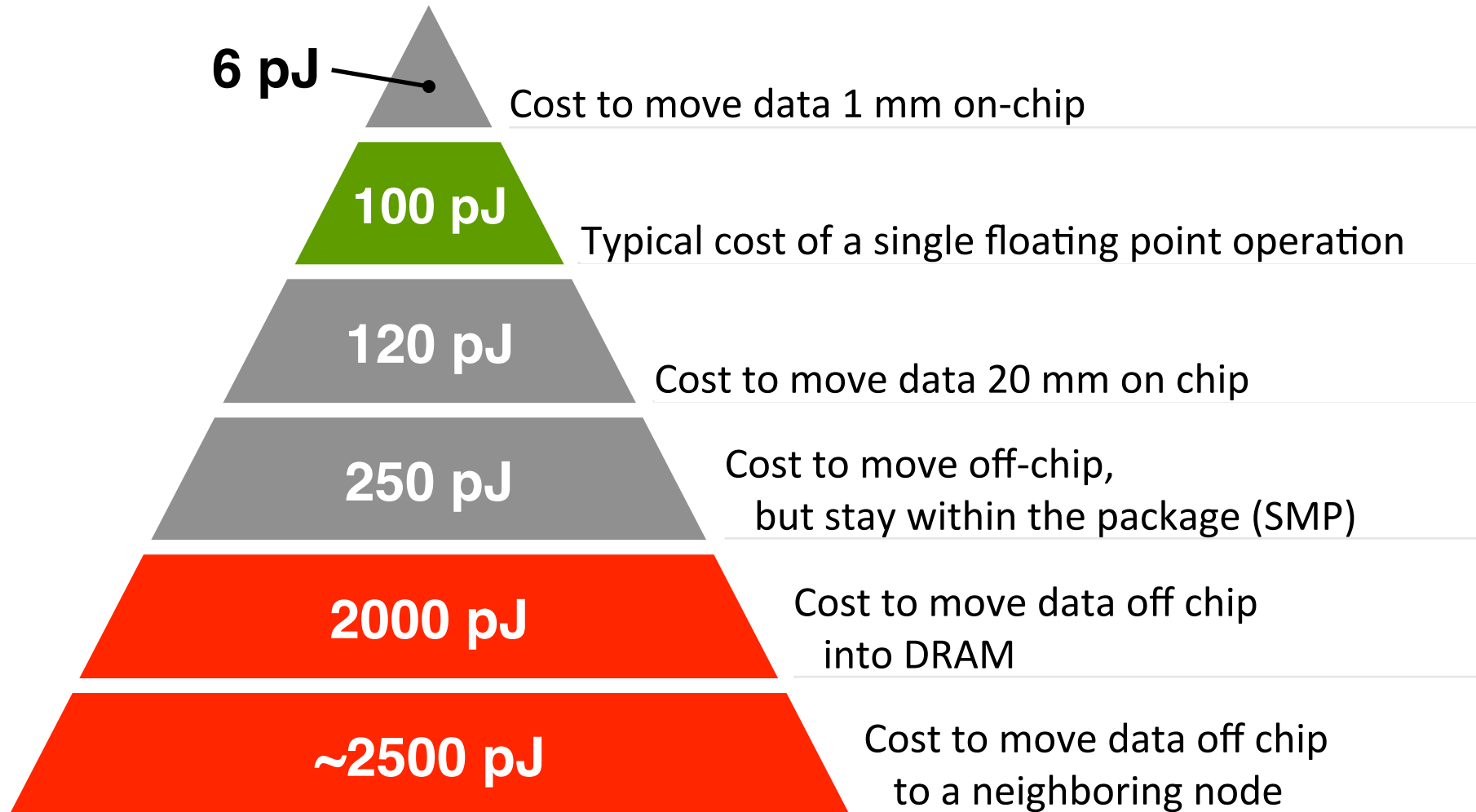


Sources:

<http://csapp.cs.cmu.edu/2e/figures.html>, <http://csapp.cs.cmu.edu/3e/figures.html>

Data Movement is Expensive

Hierarchical power costs.



Summary

- **Even more lower level parallelism**
- **Specialization**
- **Communication even more expensive (relatively)**

The end of Relaxed Programming

THE CHAIR THAT'S TEACHING AMERICA HOW TO

RELAX

What's different about the BarcaLounger reclining chair? It's a chair you'll find in the finest homes in America. But it wasn't just built for looks or style. It's also the chair essentially designed from the inside out for ultimate...

Just sit down, lean back, and let the BarcaLounger's exclusive "Floating Function" take over. Back rest, seat, and leg rest all reclinable to make your body essentially in any position from sitting to reclining.

Nowhere else can you find so much comfort in such a handsome, hand-made chair. It's an achievement in relaxation!

Why the FREE booklet and name of nearest dealer: Barca Manufacturing Company, Dept. G-11, Buffalo, New York.



LOOK UNDER THE FOOTREST FOR THE NAME...

BarcaLounger

Model 1000 available. Also model 1000L, 1000R, 1000S, 1000T, 1000U, 1000V, 1000W, 1000X, 1000Y, 1000Z, 1000AA, 1000AB, 1000AC, 1000AD, 1000AE, 1000AF, 1000AG, 1000AH, 1000AI, 1000AJ, 1000AK, 1000AL, 1000AM, 1000AN, 1000AO, 1000AP, 1000AQ, 1000AR, 1000AS, 1000AT, 1000AU, 1000AV, 1000AW, 1000AX, 1000AY, 1000AZ, 1000BA, 1000BB, 1000BC, 1000BD, 1000BE, 1000BF, 1000BG, 1000BH, 1000BI, 1000BJ, 1000BK, 1000BL, 1000BM, 1000BN, 1000BO, 1000BP, 1000BQ, 1000BR, 1000BS, 1000BT, 1000BU, 1000BV, 1000BW, 1000BX, 1000BY, 1000BZ, 1000CA, 1000CB, 1000CC, 1000CD, 1000CE, 1000CF, 1000CG, 1000CH, 1000CI, 1000CJ, 1000CK, 1000CL, 1000CM, 1000CN, 1000CO, 1000CP, 1000CQ, 1000CR, 1000CS, 1000CT, 1000CU, 1000CV, 1000CW, 1000CX, 1000CY, 1000CZ, 1000DA, 1000DB, 1000DC, 1000DD, 1000DE, 1000DF, 1000DG, 1000DH, 1000DI, 1000DJ, 1000DK, 1000DL, 1000DM, 1000DN, 1000DO, 1000DP, 1000DQ, 1000DR, 1000DS, 1000DT, 1000DU, 1000DV, 1000DW, 1000DX, 1000DY, 1000DZ, 1000EA, 1000EB, 1000EC, 1000ED, 1000EE, 1000EF, 1000EG, 1000EH, 1000EI, 1000EJ, 1000EK, 1000EL, 1000EM, 1000EN, 1000EO, 1000EP, 1000EQ, 1000ER, 1000ES, 1000ET, 1000EU, 1000EV, 1000EW, 1000EX, 1000EY, 1000EZ, 1000FA, 1000FB, 1000FC, 1000FD, 1000FE, 1000FF, 1000FG, 1000FH, 1000FI, 1000FJ, 1000FK, 1000FL, 1000FM, 1000FN, 1000FO, 1000FP, 1000FQ, 1000FR, 1000FS, 1000FT, 1000FU, 1000FV, 1000FW, 1000FX, 1000FY, 1000FZ, 1000GA, 1000GB, 1000GC, 1000GD, 1000GE, 1000GF, 1000GG, 1000GH, 1000GI, 1000GJ, 1000GK, 1000GL, 1000GM, 1000GN, 1000GO, 1000GP, 1000GQ, 1000GR, 1000GS, 1000GT, 1000GU, 1000GV, 1000GW, 1000GX, 1000GY, 1000GZ, 1000HA, 1000HB, 1000HC, 1000HD, 1000HE, 1000HF, 1000HG, 1000HH, 1000HI, 1000HJ, 1000HK, 1000HL, 1000HM, 1000HN, 1000HO, 1000HP, 1000HQ, 1000HR, 1000HS, 1000HT, 1000HU, 1000HV, 1000HW, 1000HX, 1000HY, 1000HZ, 1000IA, 1000IB, 1000IC, 1000ID, 1000IE, 1000IF, 1000IG, 1000IH, 1000II, 1000IJ, 1000IK, 1000IL, 1000IM, 1000IN, 1000IO, 1000IP, 1000IQ, 1000IR, 1000IS, 1000IT, 1000IU, 1000IV, 1000IW, 1000IX, 1000IY, 1000IZ, 1000JA, 1000JB, 1000JC, 1000JD, 1000JE, 1000JF, 1000JG, 1000JH, 1000JI, 1000JJ, 1000JK, 1000JL, 1000JM, 1000JN, 1000JO, 1000JP, 1000JQ, 1000JR, 1000JS, 1000JT, 1000JU, 1000JV, 1000JW, 1000JX, 1000JY, 1000JZ, 1000KA, 1000KB, 1000KC, 1000KD, 1000KE, 1000KF, 1000KG, 1000KH, 1000KI, 1000KJ, 1000KL, 1000KM, 1000KN, 1000KO, 1000KP, 1000KQ, 1000KR, 1000KS, 1000KT, 1000KU, 1000KV, 1000KW, 1000KX, 1000KY, 1000KZ, 1000LA, 1000LB, 1000LC, 1000LD, 1000LE, 1000LF, 1000LG, 1000LH, 1000LI, 1000LJ, 1000LK, 1000LL, 1000LM, 1000LN, 1000LO, 1000LP, 1000LQ, 1000LR, 1000LS, 1000LT, 1000LU, 1000LV, 1000LW, 1000LX, 1000LY, 1000LZ, 1000MA, 1000MB, 1000MC, 1000MD, 1000ME, 1000MF, 1000MG, 1000MH, 1000MI, 1000MJ, 1000MK, 1000ML, 1000MM, 1000MN, 1000MO, 1000MP, 1000MQ, 1000MR, 1000MS, 1000MT, 1000MU, 1000MV, 1000MW, 1000MX, 1000MY, 1000MZ, 1000NA, 1000NB, 1000NC, 1000ND, 1000NE, 1000NF, 1000NG, 1000NH, 1000NI, 1000NJ, 1000NK, 1000NL, 1000NM, 1000NN, 1000NO, 1000NP, 1000NQ, 1000NR, 1000NS, 1000NT, 1000NU, 1000NV, 1000NW, 1000NX, 1000NY, 1000NZ, 1000OA, 1000OB, 1000OC, 1000OD, 1000OE, 1000OF, 1000OG, 1000OH, 1000OI, 1000OJ, 1000OK, 1000OL, 1000OM, 1000ON, 1000OO, 1000OP, 1000OQ, 1000OR, 1000OS, 1000OT, 1000OU, 1000OV, 1000OW, 1000OX, 1000OY, 1000OZ, 1000PA, 1000PB, 1000PC, 1000PD, 1000PE, 1000PF, 1000PG, 1000PH, 1000PI, 1000PJ, 1000PK, 1000PL, 1000PM, 1000PN, 1000PO, 1000PP, 1000PQ, 1000PR, 1000PS, 1000PT, 1000PU, 1000PV, 1000PW, 1000PX, 1000PY, 1000PZ, 1000QA, 1000QB, 1000QC, 1000QD, 1000QE, 1000QF, 1000QG, 1000QH, 1000QI, 1000QJ, 1000QK, 1000QL, 1000QM, 1000QN, 1000QO, 1000QP, 1000QQ, 1000QR, 1000QS, 1000QT, 1000QU, 1000QV, 1000QW, 1000QX, 1000QY, 1000QZ, 1000RA, 1000RB, 1000RC, 1000RD, 1000RE, 1000RF, 1000RG, 1000RH, 1000RI, 1000RJ, 1000RK, 1000RL, 1000RM, 1000RN, 1000RO, 1000RP, 1000RQ, 1000RR, 1000RS, 1000RT, 1000RU, 1000RV, 1000RW, 1000RX, 1000RY, 1000RZ, 1000SA, 1000SB, 1000SC, 1000SD, 1000SE, 1000SF, 1000SG, 1000SH, 1000SI, 1000SJ, 1000SK, 1000SL, 1000SM, 1000SN, 1000SO, 1000SP, 1000SQ, 1000SR, 1000SS, 1000ST, 1000SU, 1000SV, 1000SW, 1000SX, 1000SY, 1000SZ, 1000TA, 1000TB, 1000TC, 1000TD, 1000TE, 1000TF, 1000TG, 1000TH, 1000TI, 1000TJ, 1000TK, 1000TL, 1000TM, 1000TN, 1000TO, 1000TP, 1000TQ, 1000TR, 1000TS, 1000TT, 1000TU, 1000TV, 1000TW, 1000TX, 1000TY, 1000TZ, 1000UA, 1000UB, 1000UC, 1000UD, 1000UE, 1000UF, 1000UG, 1000UH, 1000UI, 1000UJ, 1000UK, 1000UL, 1000UM, 1000UN, 1000UO, 1000UP, 1000UQ, 1000UR, 1000US, 1000UT, 1000UU, 1000UV, 1000UW, 1000UX, 1000UY, 1000UZ, 1000VA, 1000VB, 1000VC, 1000VD, 1000VE, 1000VF, 1000VG, 1000VH, 1000VI, 1000VJ, 1000VK, 1000VL, 1000VM, 1000VN, 1000VO, 1000VP, 1000VQ, 1000VR, 1000VS, 1000VT, 1000VU, 1000VV, 1000VW, 1000VX, 1000VY, 1000VZ, 1000WA, 1000WB, 1000WC, 1000WD, 1000WE, 1000WF, 1000WG, 1000WH, 1000WI, 1000WJ, 1000WK, 1000WL, 1000WM, 1000WN, 1000WO, 1000WP, 1000WQ, 1000WR, 1000WS, 1000WT, 1000WU, 1000WV, 1000WW, 1000WX, 1000WY, 1000WZ, 1000XA, 1000XB, 1000XC, 1000XD, 1000XE, 1000XF, 1000XG, 1000XH, 1000XI, 1000XJ, 1000XK, 1000XL, 1000XM, 1000XN, 1000XO, 1000XP, 1000XQ, 1000XR, 1000XS, 1000XT, 1000XU, 1000XV, 1000XW, 1000XX, 1000XY, 1000XZ, 1000YA, 1000YB, 1000YC, 1000YD, 1000YE, 1000YF, 1000YG, 1000YH, 1000YI, 1000YJ, 1000YK, 1000YL, 1000YM, 1000YN, 1000YO, 1000YP, 1000YQ, 1000YR, 1000YS, 1000YT, 1000YU, 1000YV, 1000YW, 1000YX, 1000YY, 1000YZ, 1000ZA, 1000ZB, 1000ZC, 1000ZD, 1000ZE, 1000ZF, 1000ZG, 1000ZH, 1000ZI, 1000ZJ, 1000ZK, 1000ZL, 1000ZM, 1000ZN, 1000ZO, 1000ZP, 1000ZQ, 1000ZR, 1000ZS, 1000ZT, 1000ZU, 1000ZV, 1000ZW, 1000ZX, 1000ZY, 1000ZZ

- Moore: The Law that taught performance programmers to relax

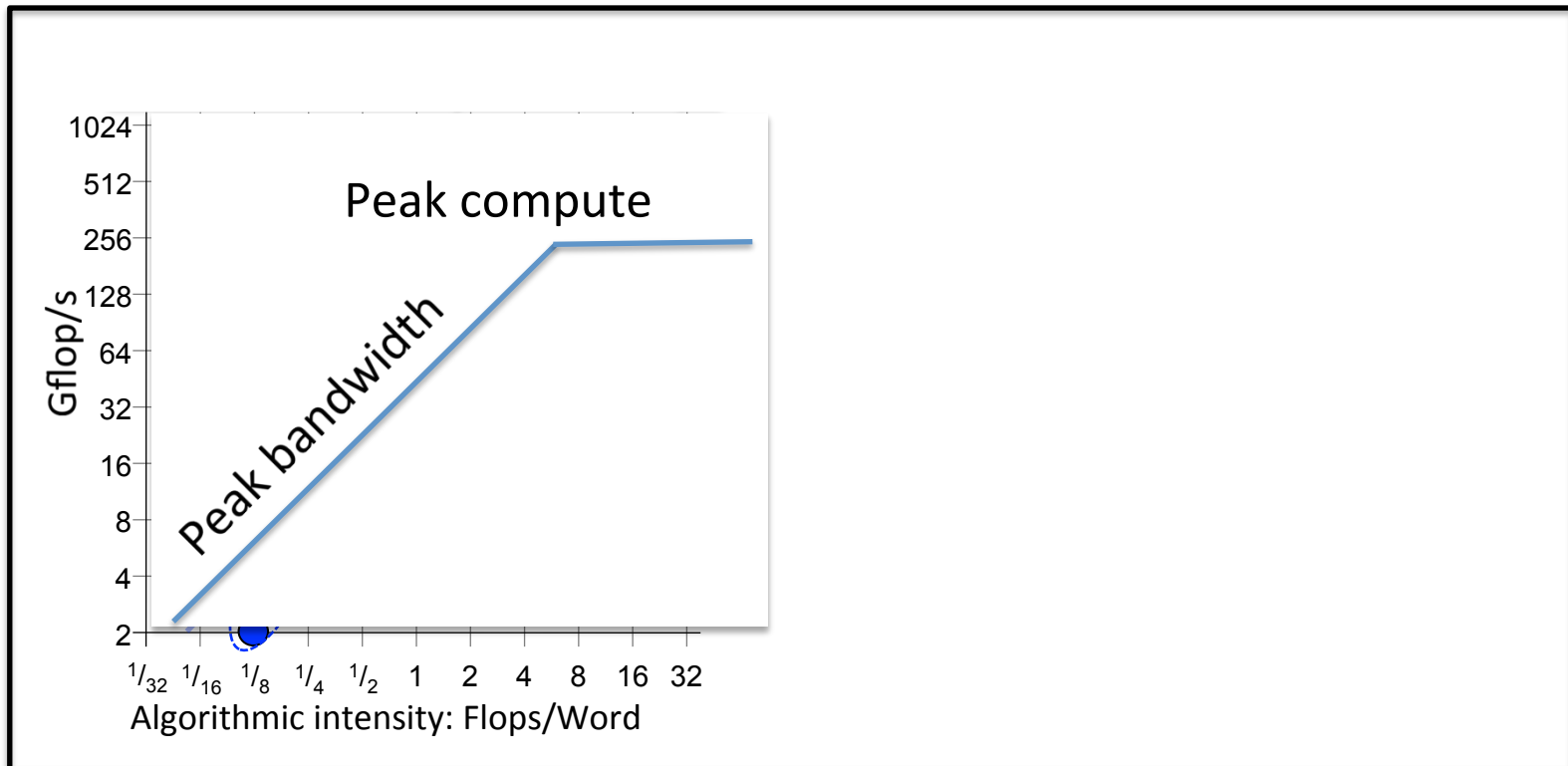
Don't Fear the Compiler

Who needs compilers?

- **Scientific computing relies heavily on libraries**
 - E.g., LAPACK and FFTW are widely used
- **Languages and compilers are still useful**
 - Higher level syntax is needed for productivity
 - We need a language
 - Static analysis is helps with correctness
 - We need a compiler (front-end)
 - Optimizations are needed to get performance
 - We need a compiler (back-end)

Autotuning: Write Code Generators

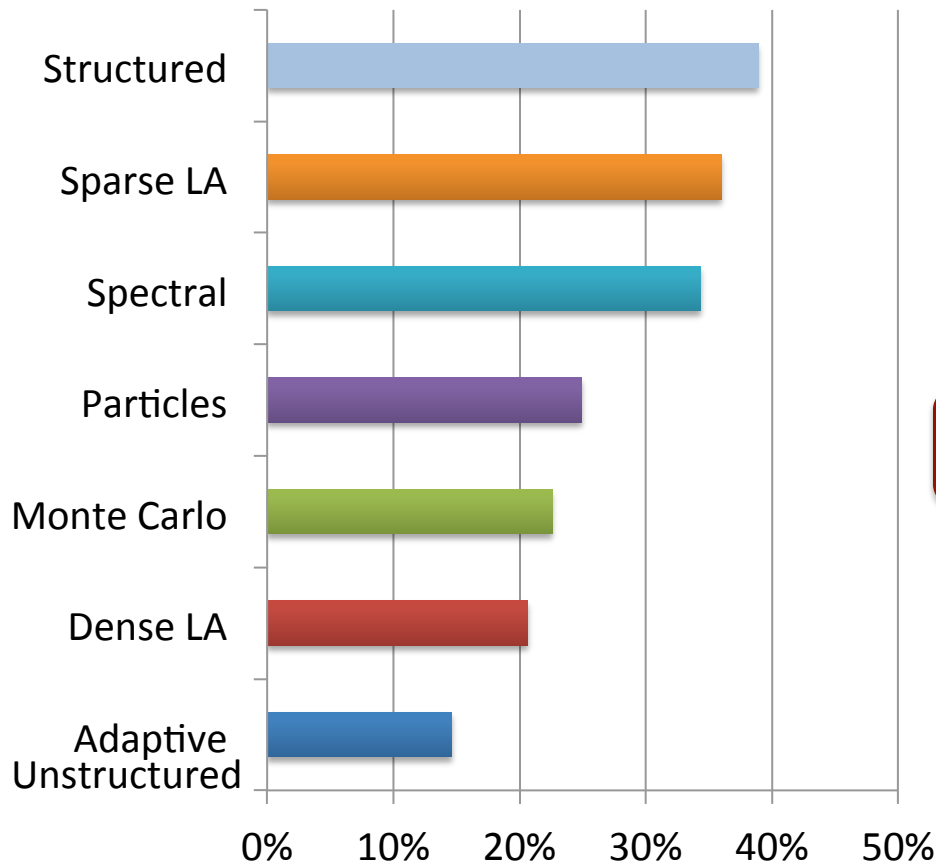
- Two “unsolved” compiler problems:
 - dependence analysis and
 - accurate performance models
- Autotuners are code generators plus search



Work by Williams, Oliner, Shalf, Madduri, Kamil, Im, Ethier, Moore's Law End Game

What we have and what we need

NERSC survey: what motifs do they use?

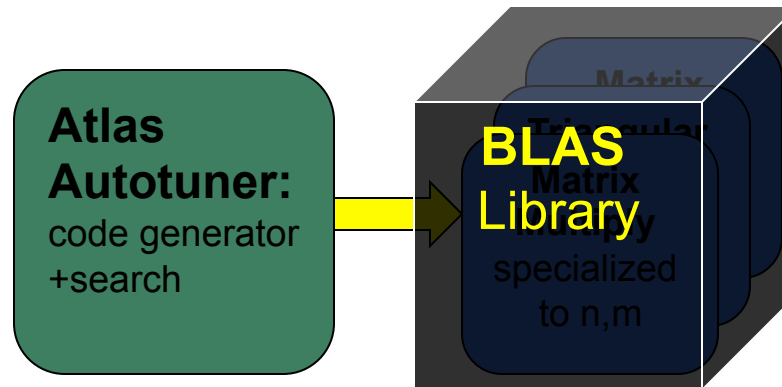


What code generators do we have?

Dense Linear Algebra	Atlas
Spectral Algorithms	FFTW, Spiral
Sparse Linear Algebra	OSKI
Structured Grids	TBD
Unstructured Grids	
Particle Methods	
Monte Carlo	

Stencils are both the most important motifs and a gap in our tools

Approaches to Autotuning



How do we produce all of these (correct) versions?

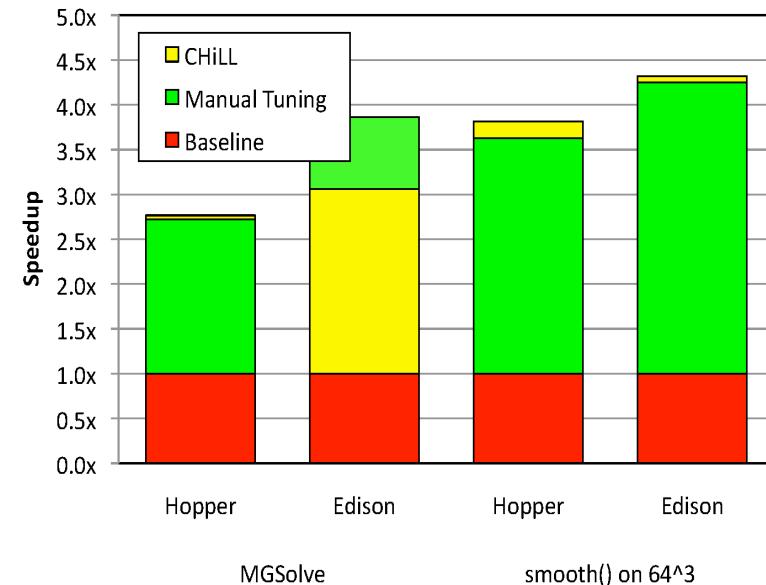
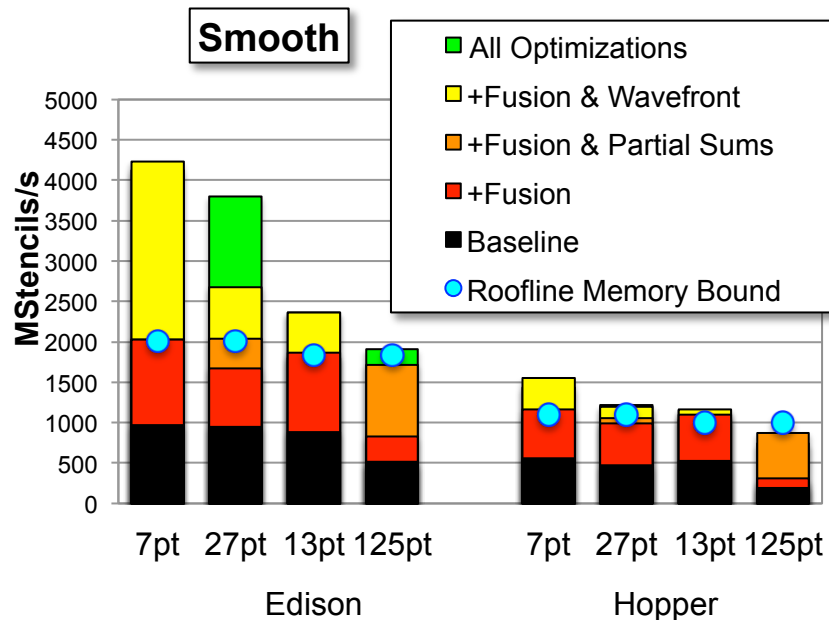
- Using scripts (Python, perl, ML, C,..)
- Compiling annotated general-purpose language (X-Tune,...)
- Use preprocessor to generator code (Raja, Kokkos, TiDA)
- Compile a domain-specific language (D-TEC, Halide)
- Domain-specific compiler for domain-specific language (SEJITS)

Approximate categorization!

Several Projects and Pls: Sam Williams, Mary Hall, Dan Quinlan, Armando Fox, Saman Amarsinghe, Armando Solar-Lezama, Jack Dongarra, Moore's Law End Game

Approach #1: Compiler-Directed Autotuning

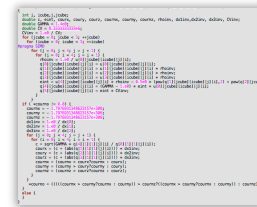
- **Two hard compiler problems**
 - Analyzing the code to determine legal transformations
 - Selecting the best (or close) optimized version
- **Approach #1: General-purpose compilers (+ annotations)**
 - Use *communication-avoiding optimizations* to reduce memory bandwidth
 - Apply **CHILL compiler** technology with general polyhedral optimizations
 - Use autotuning to select optimized version



Results on Geometric Multigrid (miniGMG Smoother)

Approach #2: DSLs with General Purpose Compiler

- **Generation of Complex Code for 10 Levels of Memory Hierarchy with SW managed cache**
 - 4th order stencil computation from CNS Co-Design Proxy-App
 - Same DSL code can generate to 2, 3, 4, ... levels too
 - Code size of autogenerated code



Memory Hierarchy	2 Level	3 Level	4 Level	...	10 level
DSL Code	20				
Auto Generated Code	446	500	553		819

Use of Rose/PolyOpt to apply DSLs to large applications and collaboration on AMR

Approach #3: Domain-Specific (but not too specific)

Languages used by other markets

Developed for Image Processing



- 10+ FTEs developing Halide
- 50+ FTEs use it; > 20 kLOC

HPGMG (Multigrid on Halide)

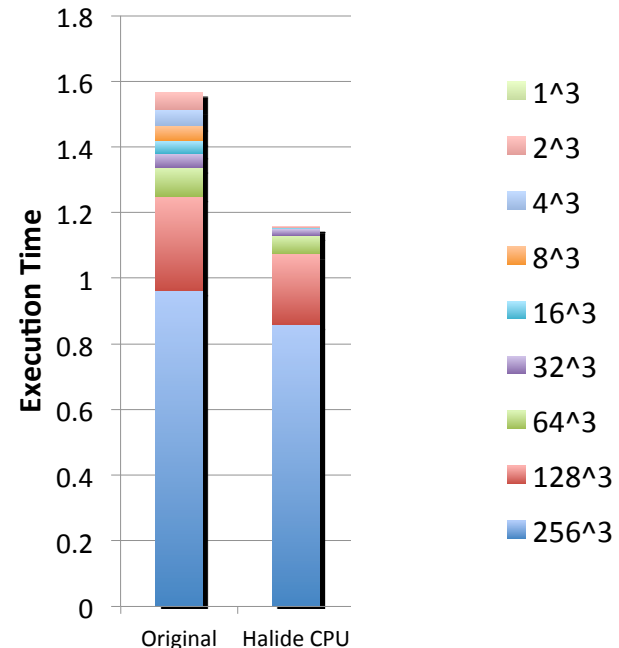
- Halide Algorithm by domain expert

```
Func Ax_n("Ax_n", lambda("lambda"), chebyshev("chebyshev"));
Var i("i"), j("j"), k("k");
Ax_n(i,j,k) = a*alpha(i,j,k)*x_n(i,j,k) - b*2inv*(
  beta_i(i,j,k) *(valid(i-1,j,k)*(x_n(i,j,k) + x_n(i-1,j,k)) - 2.0f*x_n(i,j,k))
  + beta_j(i,j,k) *(valid(i,j-1,k)*(x_n(i,j,k) + x_n(i,j-1,k)) - 2.0f*x_n(i,j,k))
  + beta_k(i,j,k) *(valid(i,j,k-1)*(x_n(i,j,k) + x_n(i,j,k-1)) - 2.0f*x_n(i,j,k))
  + beta_i(i+1,j,k) *(valid(i+1,j,k)*(x_n(i,j,k) + x_n(i+1,j,k)) - 2.0f*x_n(i,j,k))
  + beta_j(i,j+1,k) *(valid(i,j+1,k)*(x_n(i,j,k) + x_n(i,j+1,k)) - 2.0f*x_n(i,j,k))
  + beta_k(i,j,k+1) *(valid(i,j,k+1)*(x_n(i,j,k) + x_n(i,j,k+1)) - 2.0f*x_n(i,j,k)));
lambda(i,j,k) = 1.0f / (a*alpha(i,j,k) - b*2inv*(
  beta_i(i,j,k) *(valid(i-1,j,k) - 2.0f)
  + beta_j(i,j,k) *(valid(i,j-1,k) - 2.0f)
  + beta_k(i,j,k) *(valid(i,j,k-1) - 2.0f)
  + beta_i(i+1,j,k) *(valid(i+1,j,k) - 2.0f)
  + beta_j(i,j+1,k) *(valid(i,j+1,k) - 2.0f)
  + beta_k(i,j,k+1) *(valid(i,j,k+1) - 2.0f)));
chebyshev(i,j,k) = x_n(i,j,k) + c1*(x_n(i,j,k)-x_nm1(i,j,k))+
  c2*lambda(i,j,k)*(rhs(i,j,k)-Ax_n(i,j,k));
```

- Halide Schedule either
 - Auto-generated by autotuning with opentuner
 - Or hand created by an optimization expert

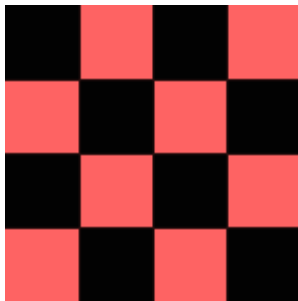
Halide performance

- Autogenerated schedule for CPU
- Hand created schedule for GPU
- No change to the algorithm

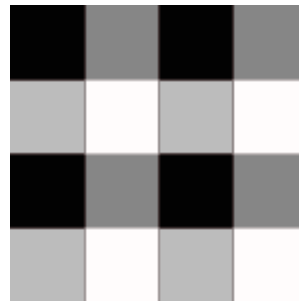


Approach #4: Small Compiler for Small Language

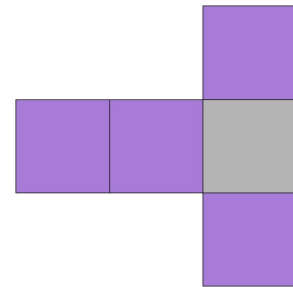
- **Snowflake: A DSL for Science Stencils**
 - Domain calculus inspired by Titanium, UPC++, and AMR in general



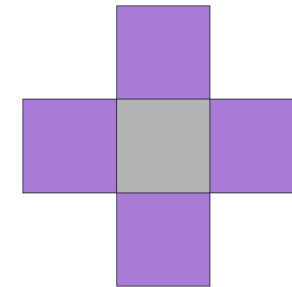
(a) Red-Black tiling



(b) 4-color tiling



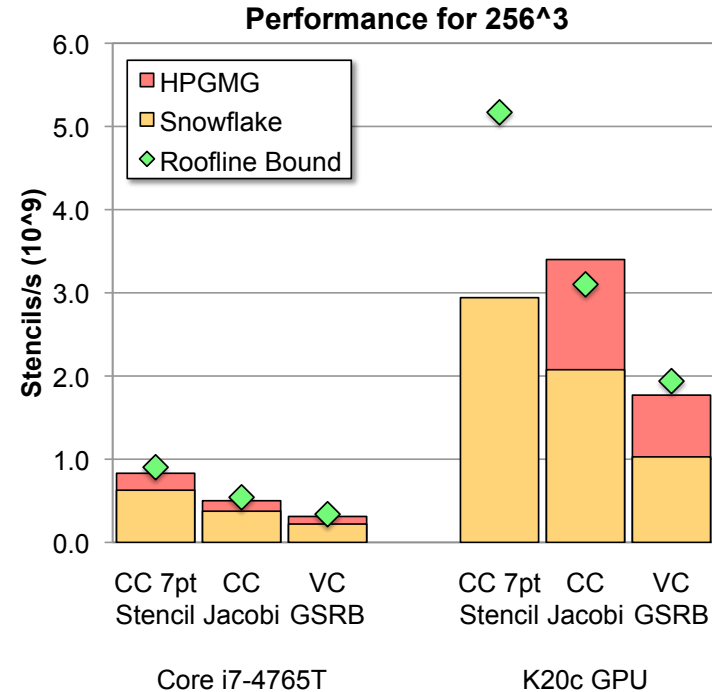
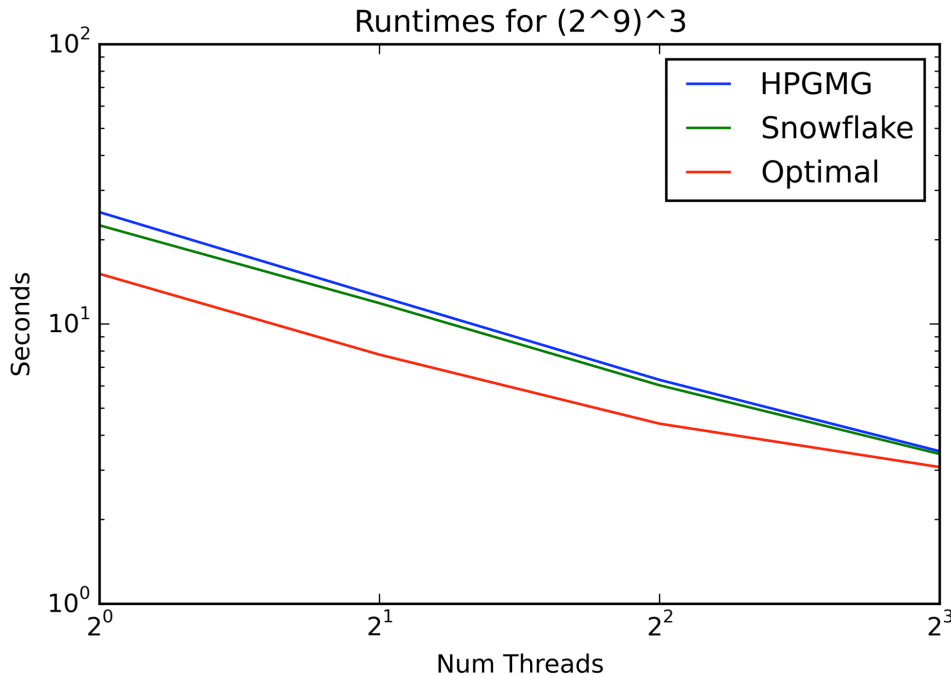
(c) Asymmetric stencil used near mesh boundary



(d) 5-point Jacobi stencil

- **Complex stencils: red/black, asymmetric**
- **Update-in-place while preserving provable parallelism**
- **Complex boundary conditions**

Snowflake Performance



- Performance on the HPGMG application benchmark using all the features of Snowflake
- Competitive with hand-optimized performance
- Within 2x of optimal roofline

Algorithms for the Hardware

Beyond Domain Decomposition

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

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

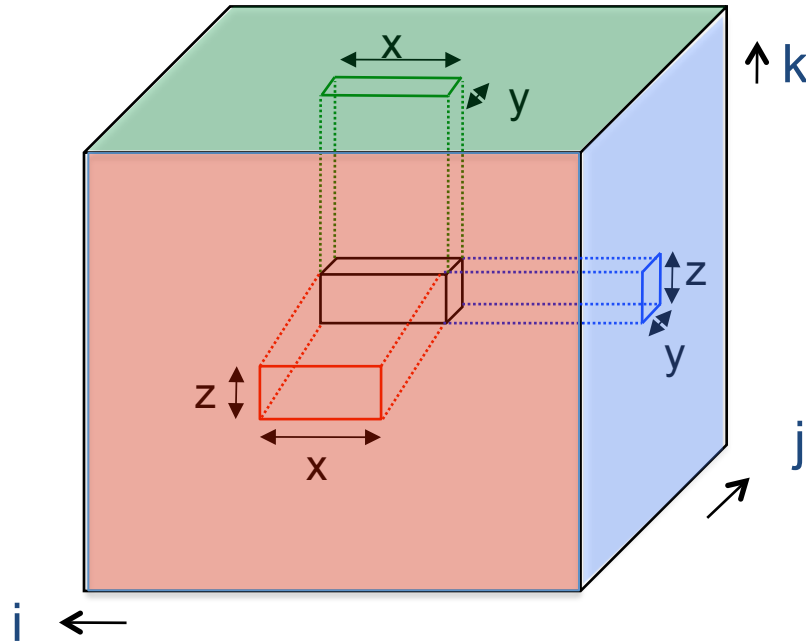
Lesson: Never waste fast memory

And don't get hung up on the owner computes rule

Can we generalize for compiler writers?

Deconstructing 2.5D Matrix Multiply

Solomonick & Demmel



- Tiling the iteration space
- 2D algorithm: never chop k dim
- 2.5 or 3D: Assume + is associative; chop k, which is \rightarrow replication of C matrix

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

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

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*
$$\#words_moved = \Omega(\#iterations/M^e)$$
for some e we can determine
- **Thm (C/D/K/S/Y):** Under some assumptions, we can determine the optimal tiles sizes
 - E.g., index expressions are just subsets of indices
- **Long term goal:** All compilers should generate communication optimal code from nested loops

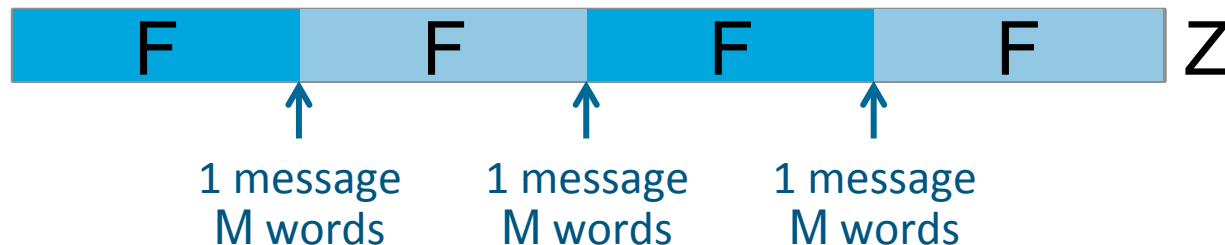
Communication Lower Bounds

- For loop nests with arrays
 - **M** words of data in fast memory, i.e., n/p .

$$\#msgs = \frac{\#flops \text{ each processor has to do } (Z)}{\text{max \#useful flops with } \mathbf{M} \text{ words } (F)}$$

$$\#words = \#msgs \cdot \mathbf{M}$$

M^2 for N-body $M^{3/2}$ for matmul



Implications for Compilers

- **Much of the work on compilers is based on owner-computes**
 - For MM: Divide C into chunks, schedule movement of A/B
 - Data-driven domain decomposition partitions data; but we can partition work instead
- **Ways to compute C “pencil”**
 1. Serially
 2. Parallel reduction
 3. Parallel asynchronous (atomic) updates
 4. Or any hybrid of these *Standard vectorization trick*
- **For what types / operators does this work?**
 - “+” is associative for 1,2 rest of RHS is “simple”
 - and commutative for 3

x += ...

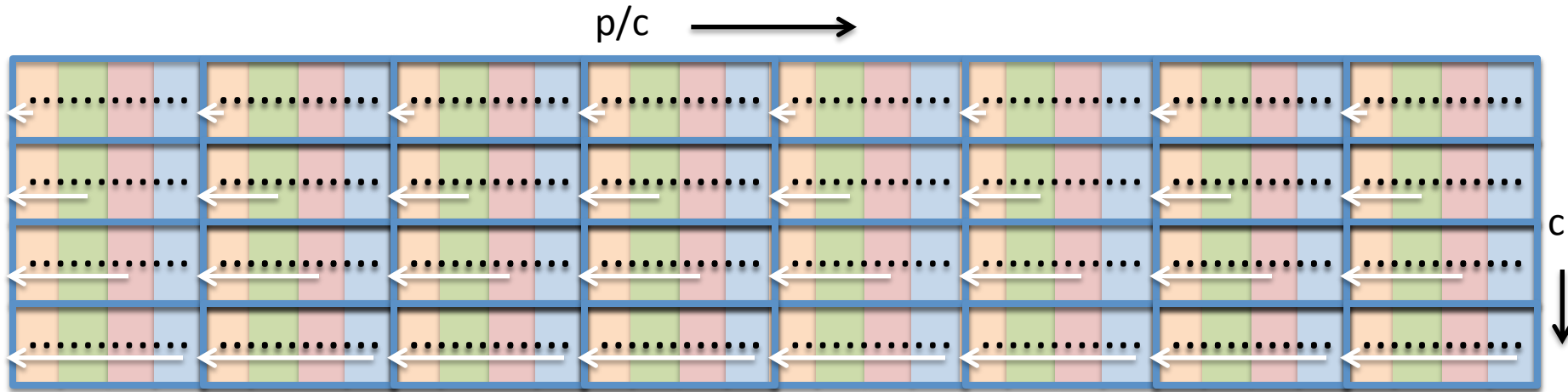
x += ...

x += ...

x += ...

Using x for C[i,j] here

Communication Avoiding Version (using a “1.5D” decomposition)



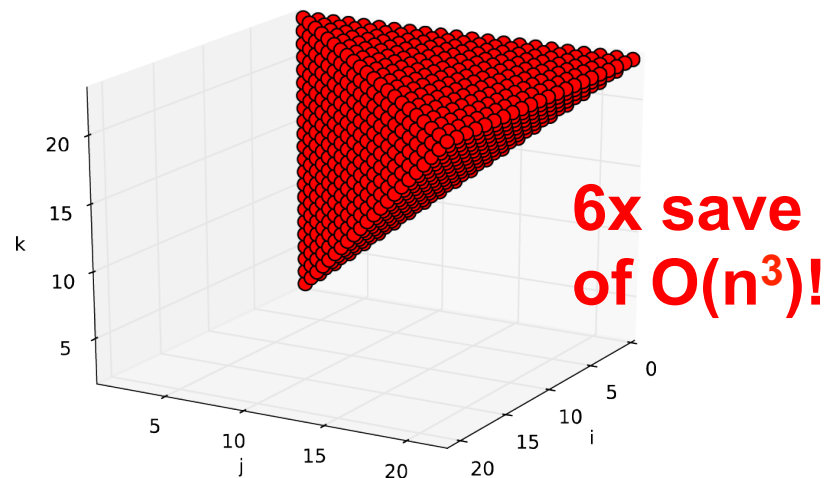
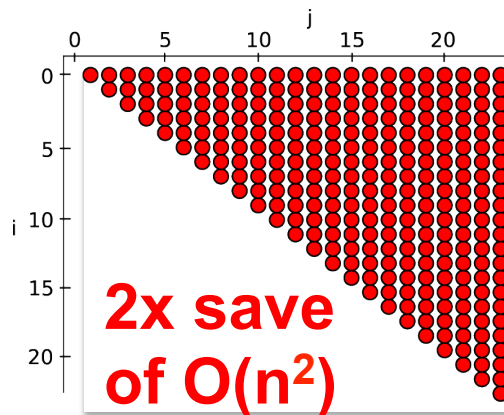
- **Divide p into c groups. Replicate particles within group.**
 - First row responsible for updating all by orange, second all by green,...
- **Algorithm: shift copy of $n/(p*c)$ particles to the left**
 - Combine with previous data before passing further level (log steps)
- **Reduce across c to produce final value for each particle**
- Total Computation: $O(n^2/p)$;
- Total Communication: $O(\log(p/c) + \log c)$ messages,

$$\text{Limit: } c \leq p^{1/2}$$

$$O(n*(c/p+1/c)) \text{ words}$$

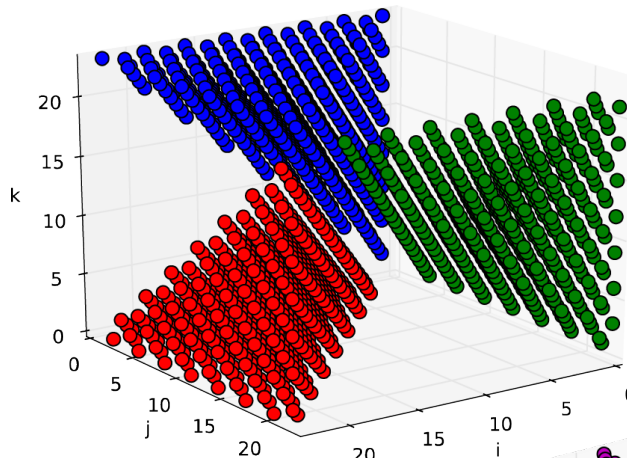
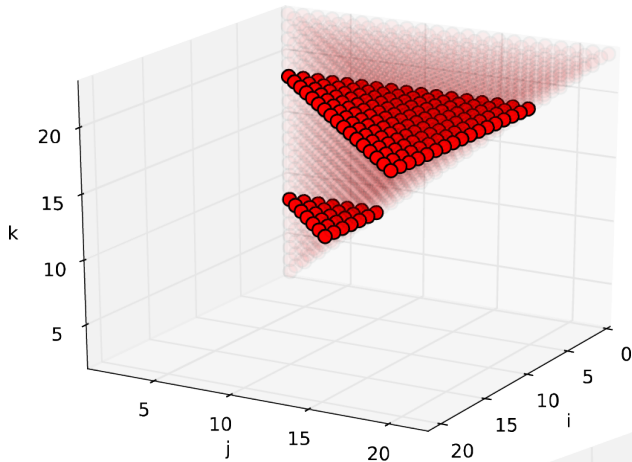
Challenge: Symmetry & Load Balance

- Force symmetry ($f_{ij} = -f_{ji}$) saves computation
- 2-body force matrix vs 3-body force cube



- How to divide work equally?

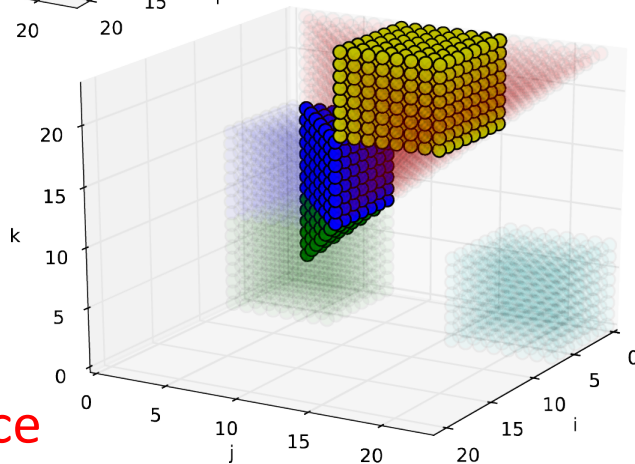
All-triplets 3-body: Challenges



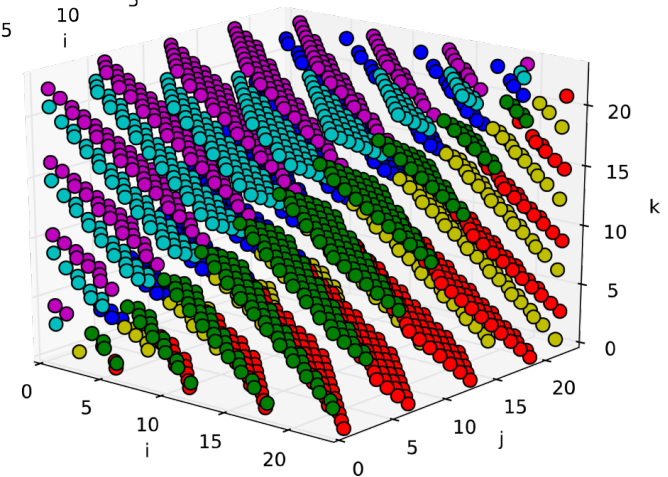
[Sumanth et al. 2007]

Symmetry
Load balance
Communication?

[Li et al. 2006]
[Li et al. 2008]

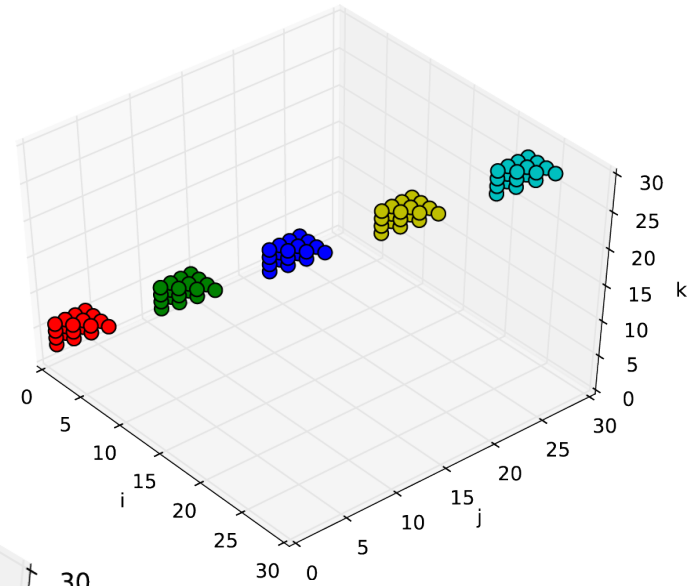
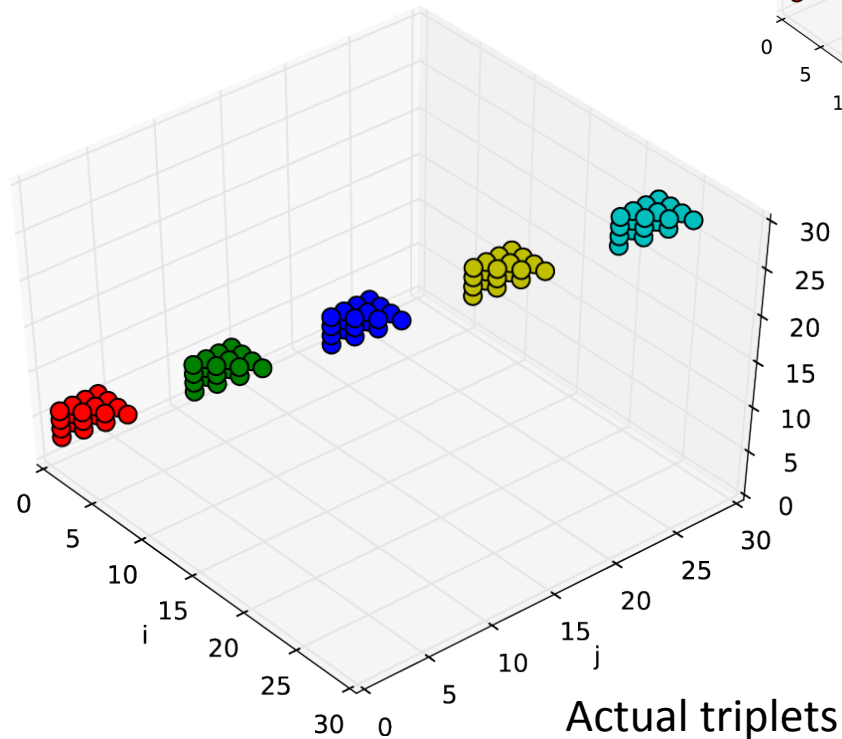


Symmetry
Load balance
Communication?



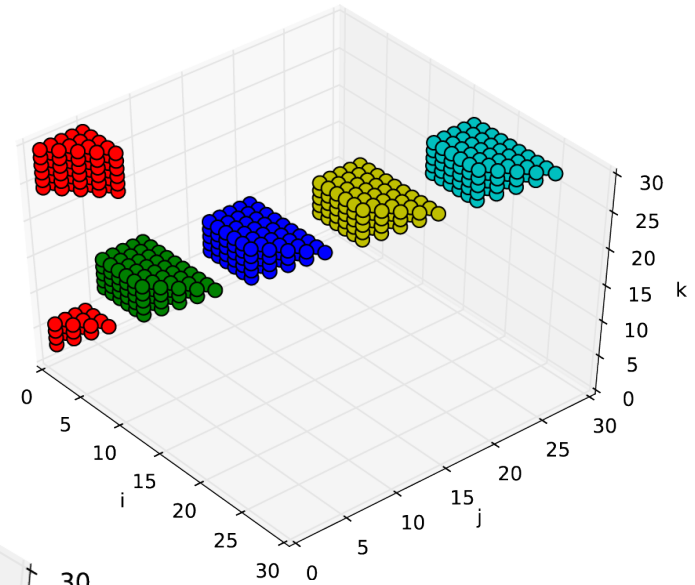
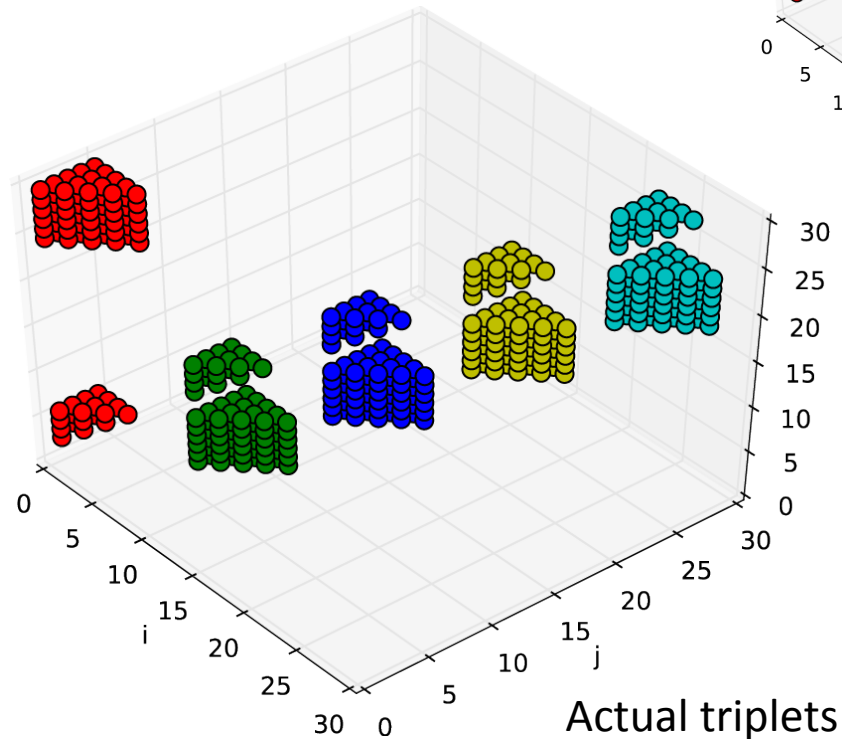
* Colors have no special meaning --
for illustration purpose only.

- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes

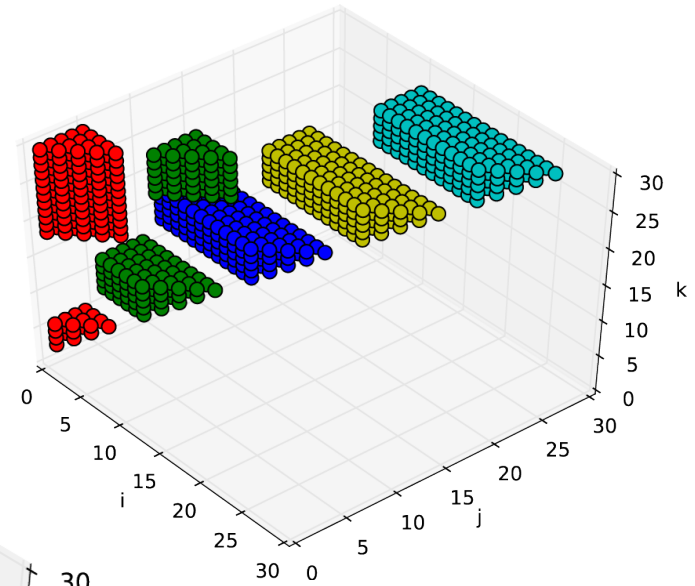
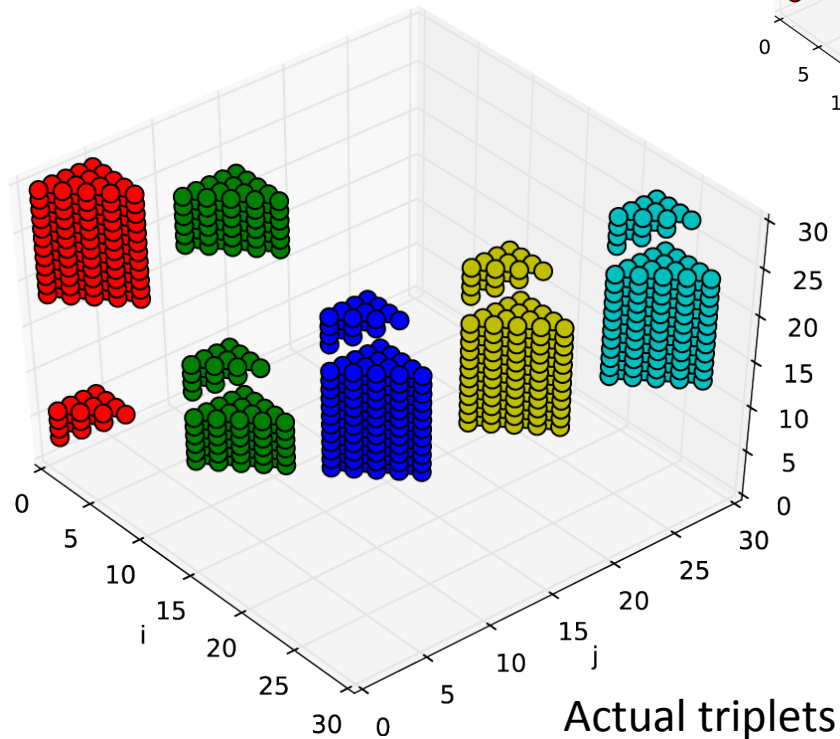


Equivalent triplets in
the big tetrahedron

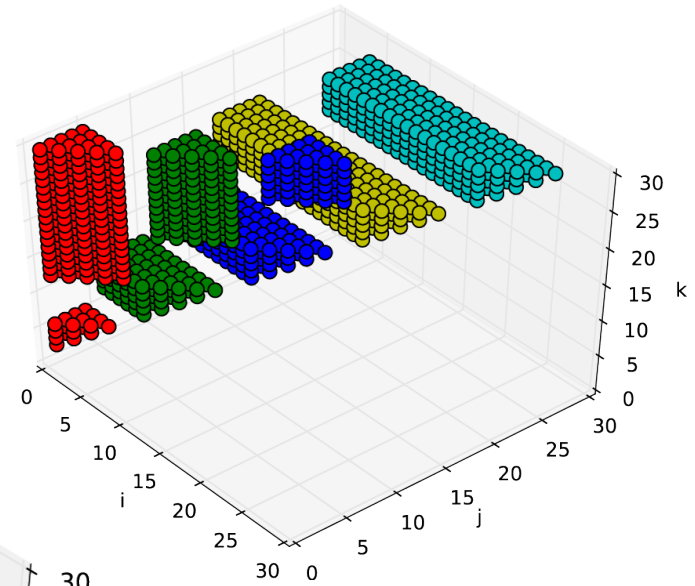
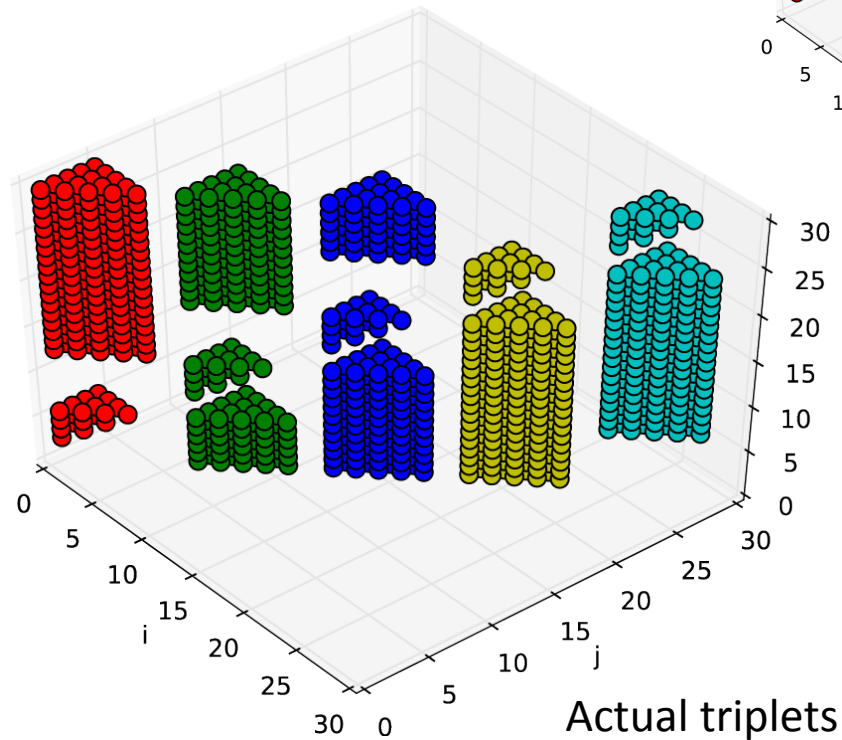
- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes



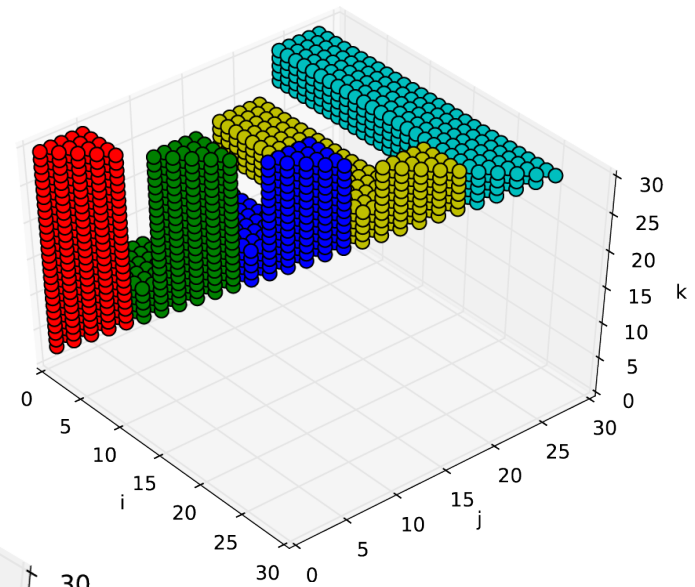
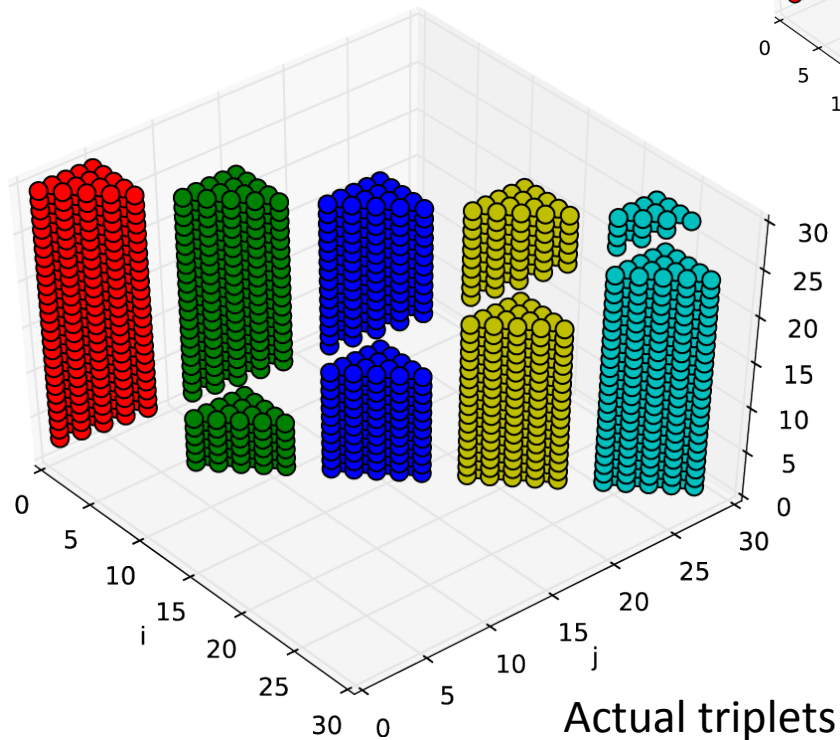
- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes



- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes

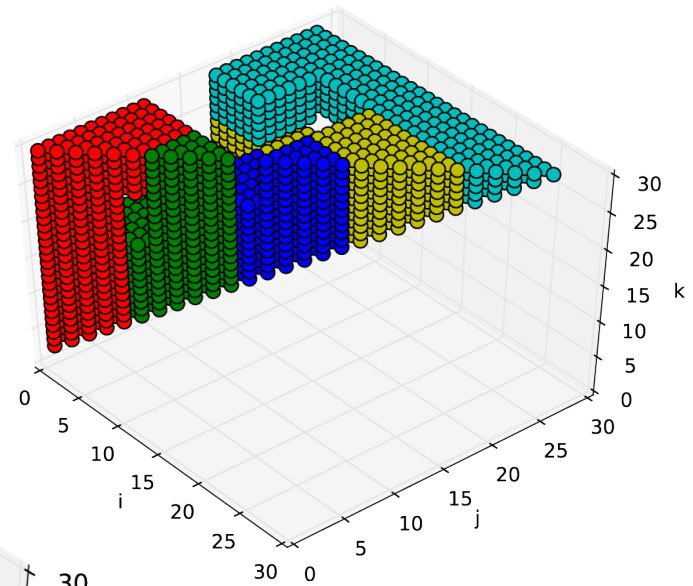
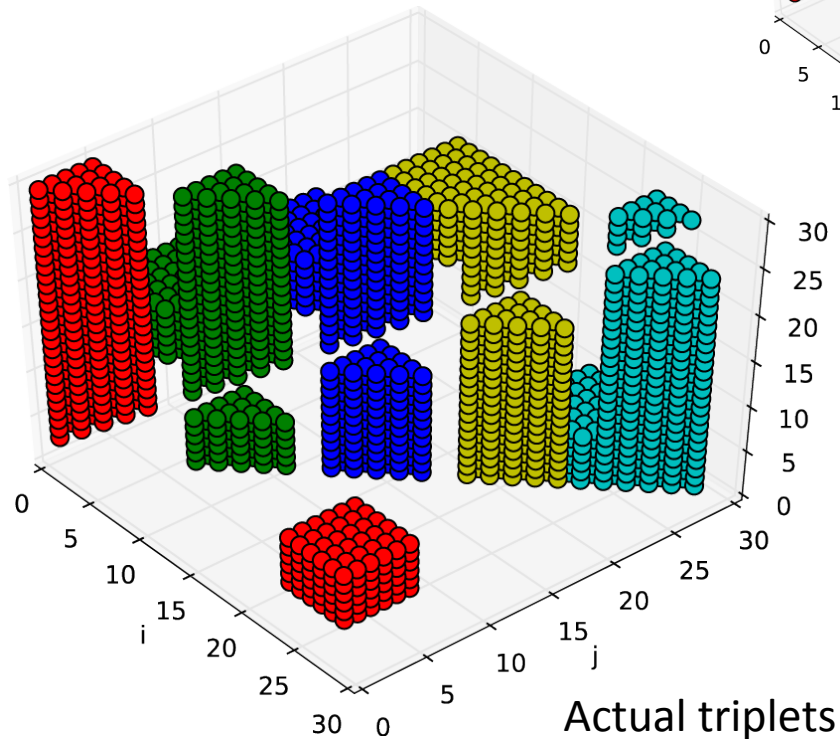


- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes



Equivalent triplets in
the big tetrahedron

- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes

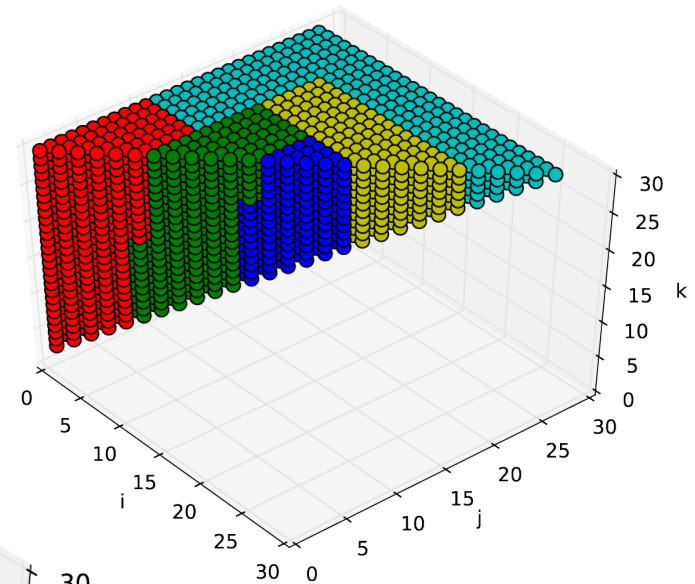
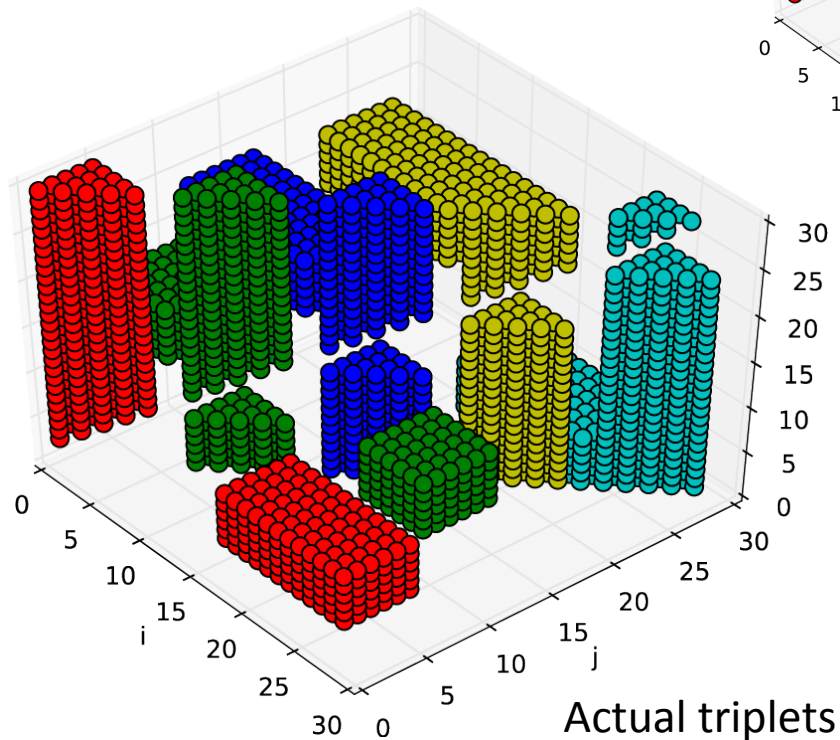


Equivalent triplets in
the big tetrahedron

CA 3-body

[Koanantakool and Yelick 2014]

- $p=5$ (in colors)
- 6 particles per processor
- 5x5 subcubes

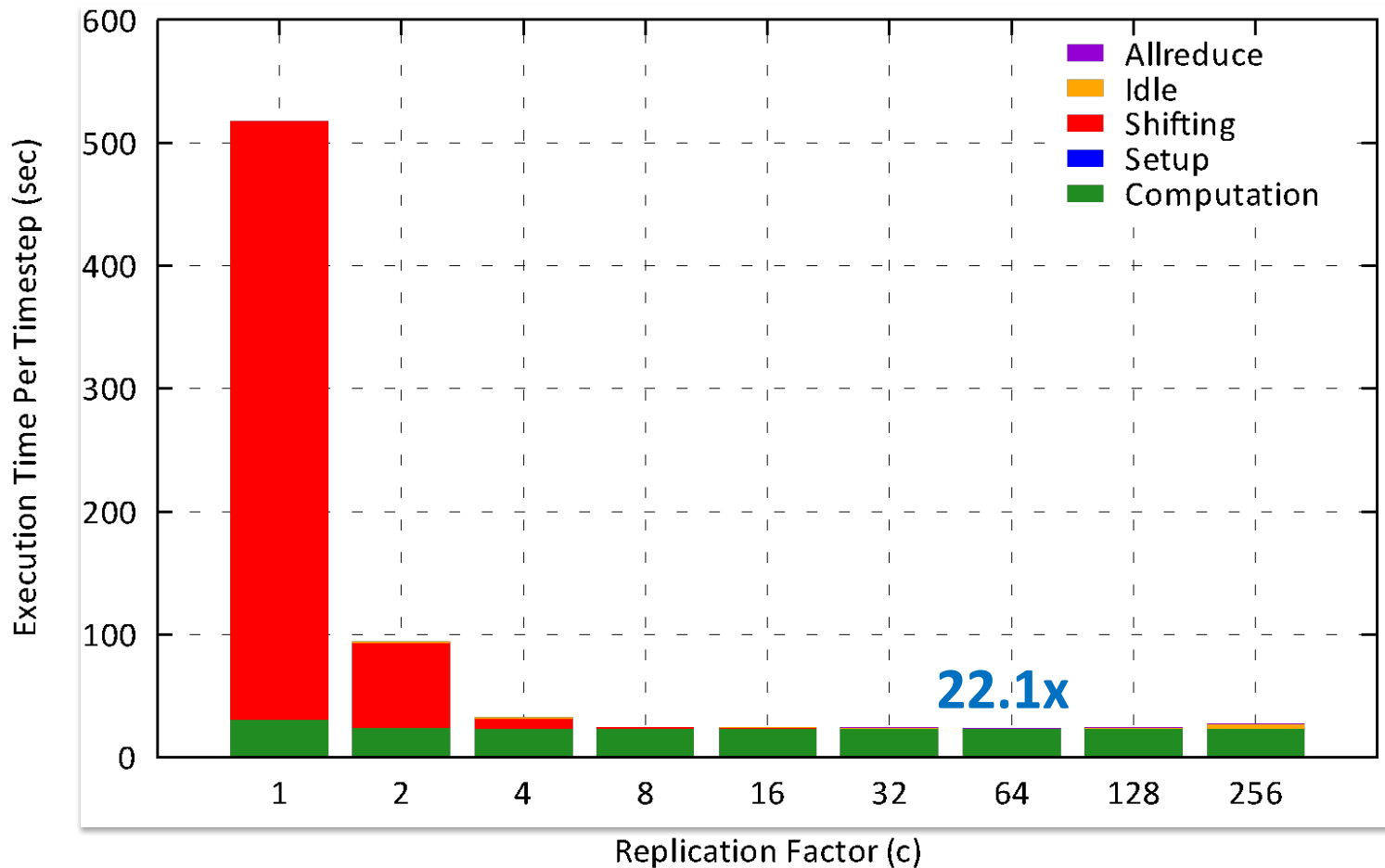


Equivalent triplets in the big tetrahedron

**Communication optimal.
Replication decreases
#msgs and #words by
factors of c^3 and c^2 .**

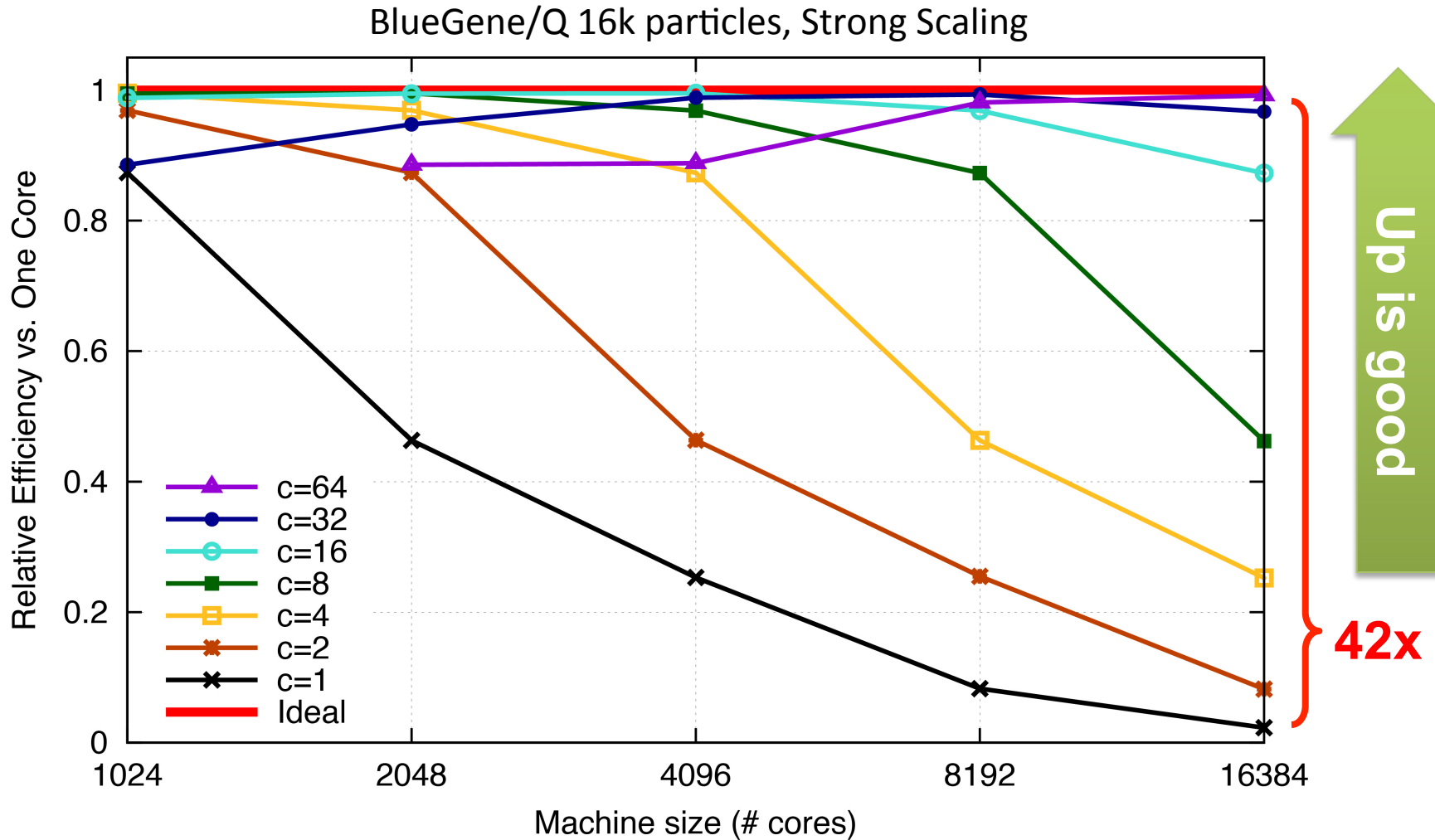
3-Way N-Body Speedup

- Cray XC30, 24k cores, 24k particles

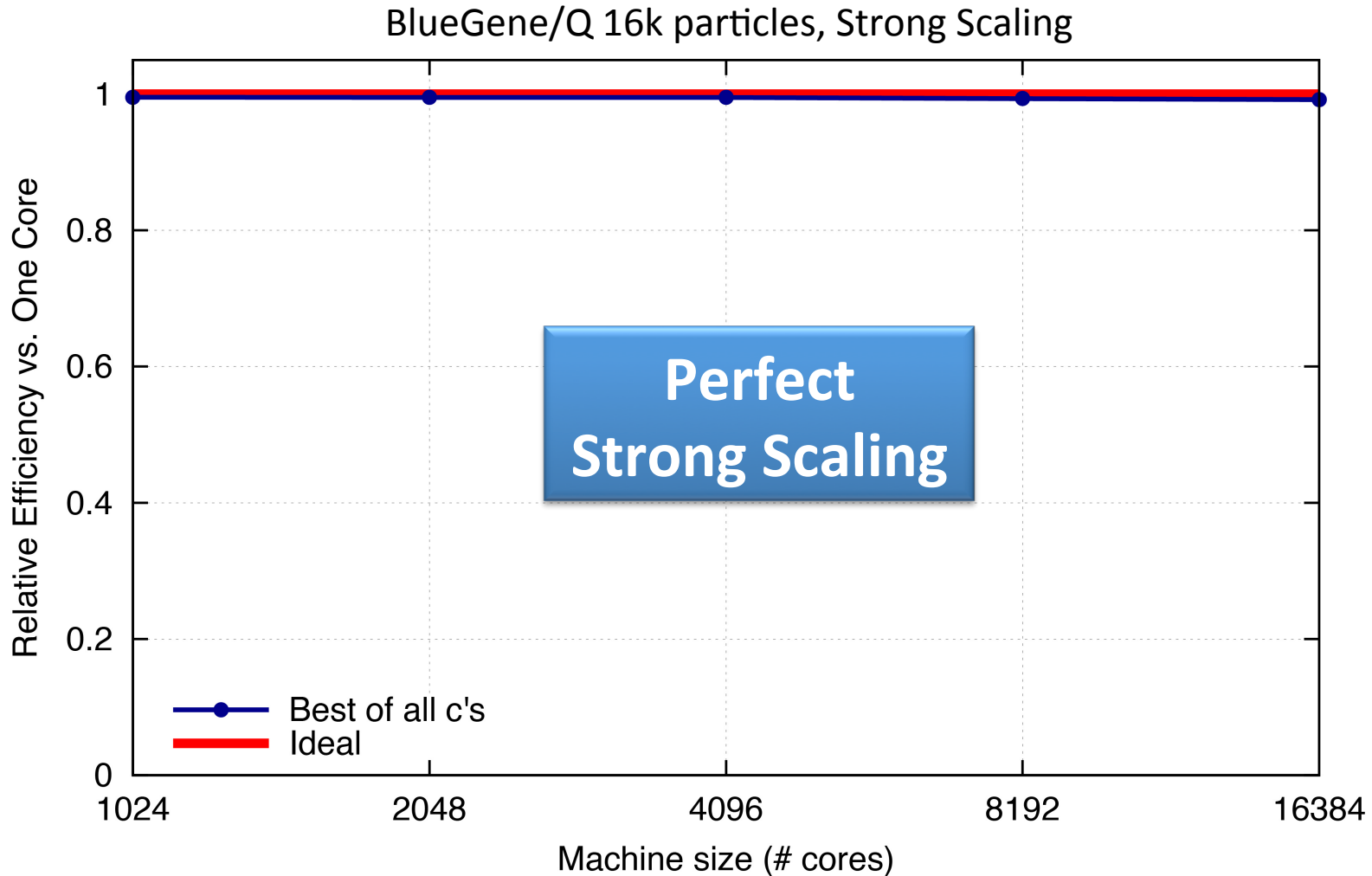


Koanantakool & Yelick

Perfect Strong Scaling



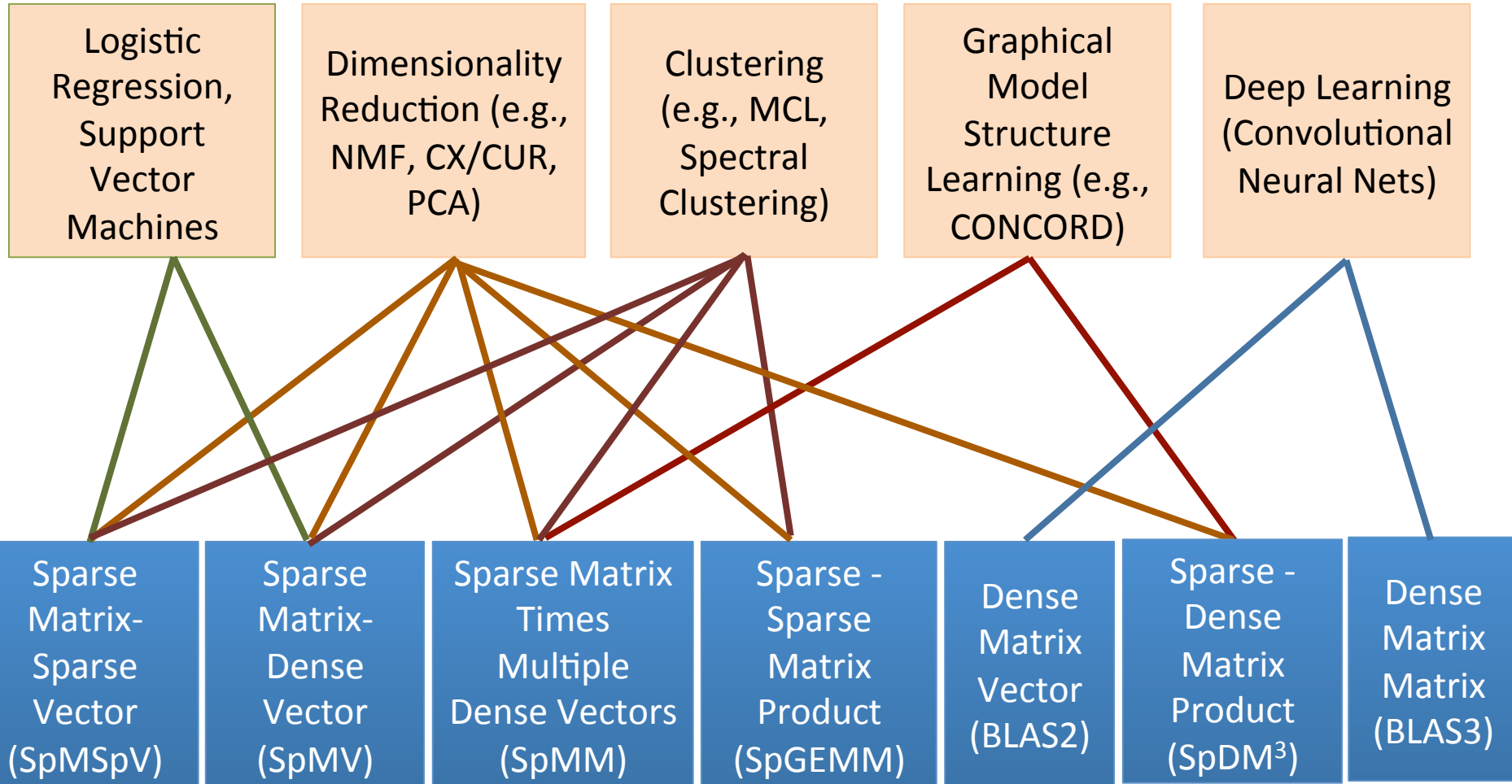
Perfect Strong Scaling



Analytics vs. Simulation Kernels:

7 Giants of Data	7 Dwarfs of Simulation
Basic statistics	Monte Carlo methods
Generalized N-Body	Particle methods
Graph-theory	Unstructured meshes
Linear algebra	Dense Linear Algebra Sparse Linear Algebra
Optimizations	
Integrations	Spectral methods
Alignment	Structured Meshes

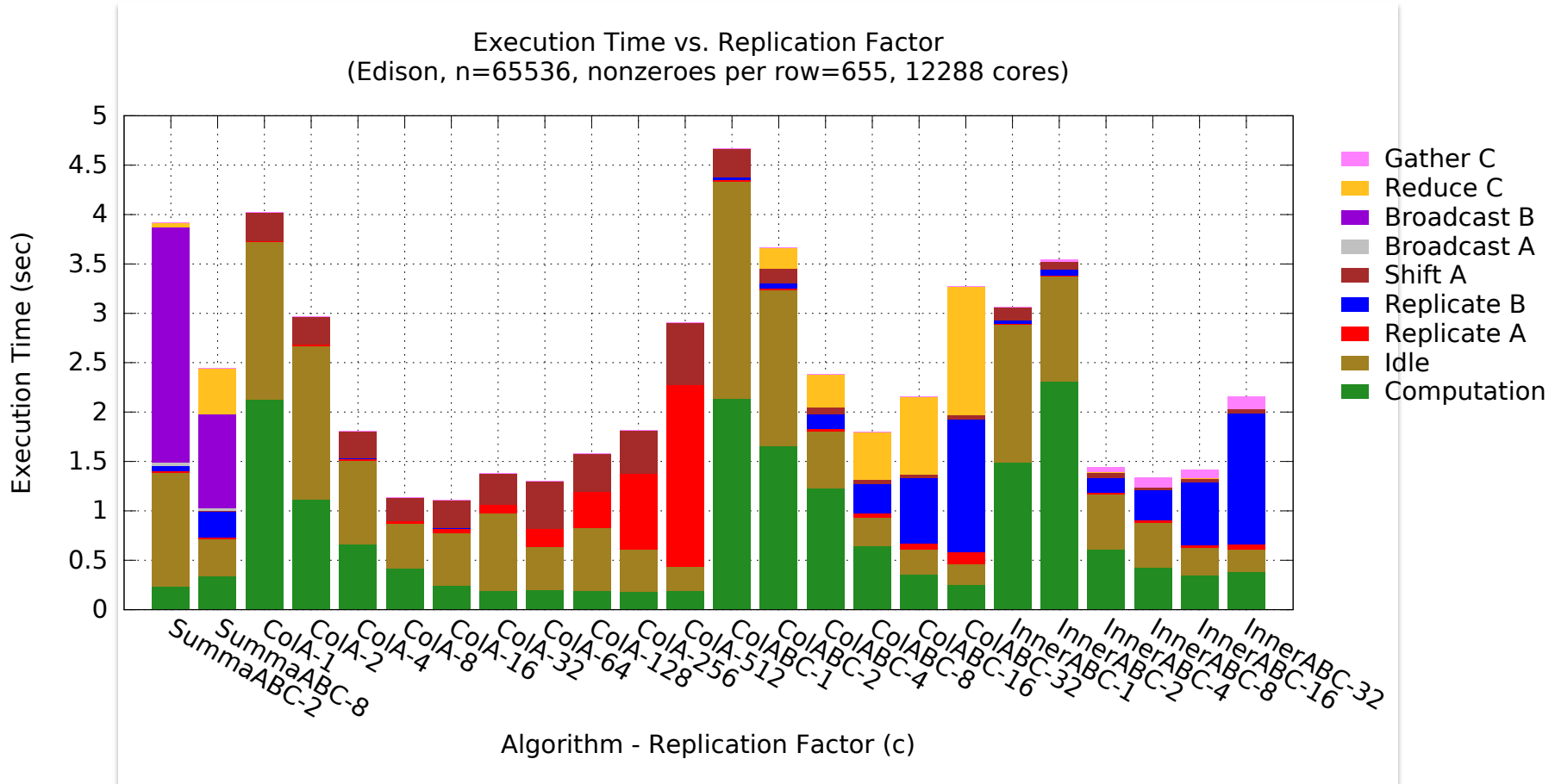
Machine Learning Mapping to Linear Algebra



Aydin Buluc

Increasing arithmetic intensity

Sparse-Dense Matrix Multiply Too!

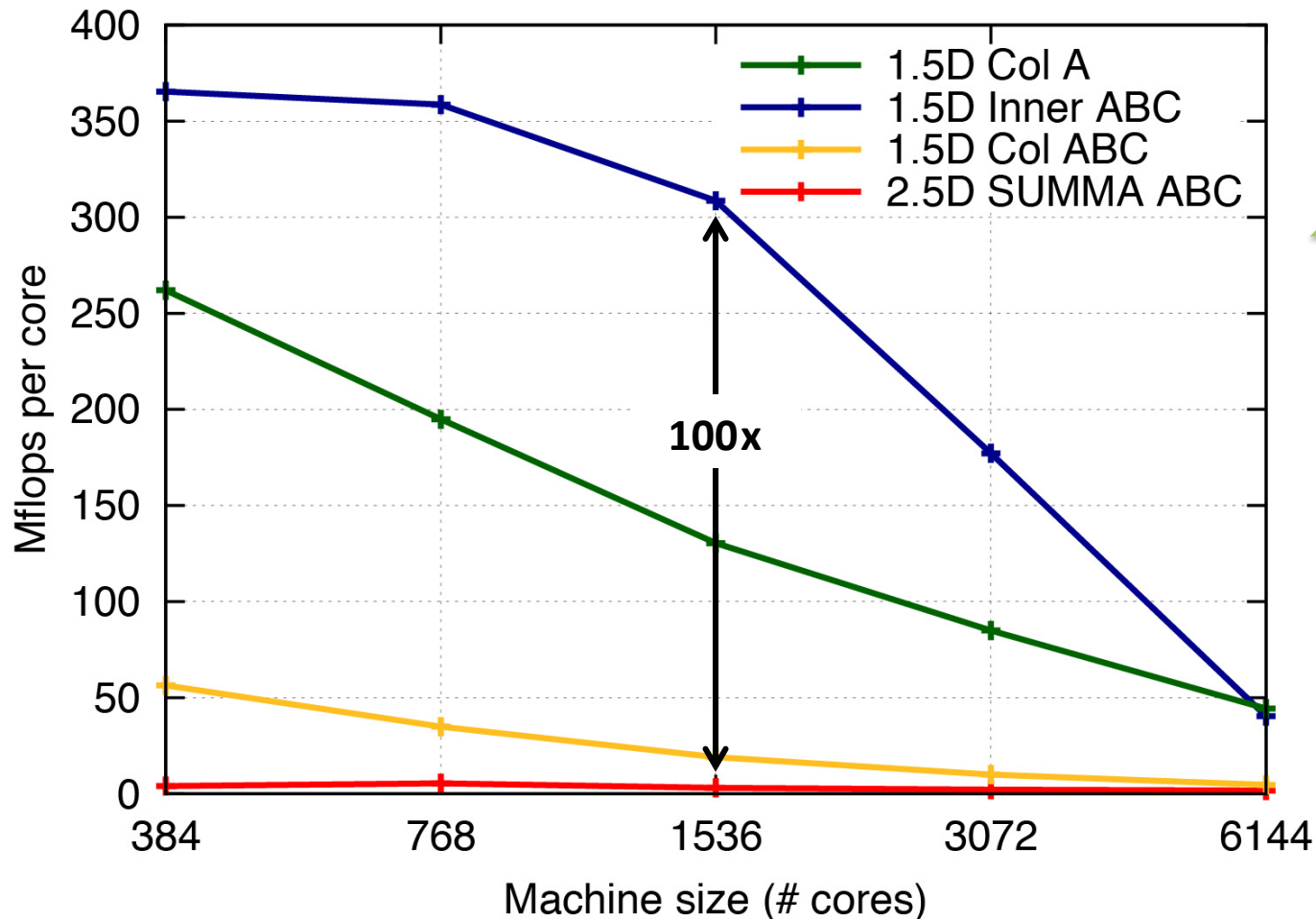


- **Variety of algorithms that divide in or 2 dimensions**

Koanantakool & Yelick

100x Improvement

- $A^{66k \times 172k}$, $B^{172k \times 66k}$, 0.0038% nnz, Cray XC30



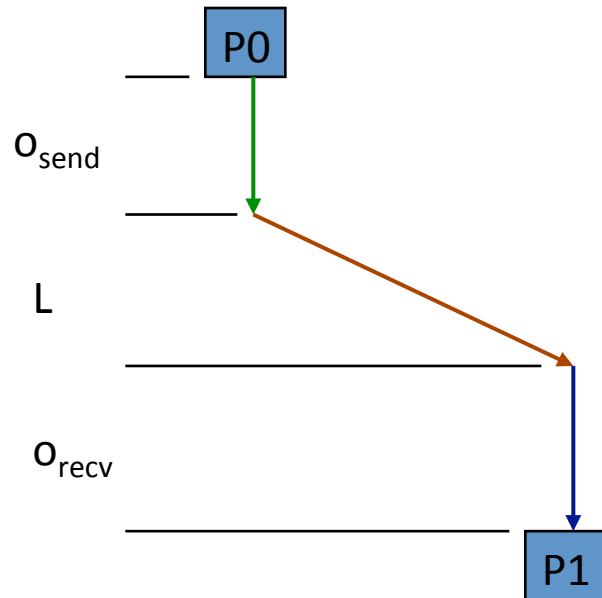
Communication-Avoiding Algorithm Sample Speedups

- Up to 11.8x faster for direct N-body on 32K core IBM BG/P
- Up to 100x faster for sparse-dense matmul on Cray XC30
- Up to 12x faster for 2.5D matmul on 64K core IBM BG/P
- Up to 3x faster for tensor contractions on 2K core Cray XE/6
- Up to 6.2x faster for APSP on 24K core Cray CE6
- Up to 2.1x faster for 2.5D LU on 64K core IBM BG/P
- Up to 13x faster for TSQR on Tesla C2050 Fermi NVIDIA GPU
- Up to 6.7x faster for symeig (band A) on 10 core Intel Westmere
- Up to 2x faster for 2.5D Strassen on 38K core Cray XT4
- Up to 4.2x faster for MiniGMG benchmark bottom solver, using CA-BiCGStab (2.5x for overall solve)

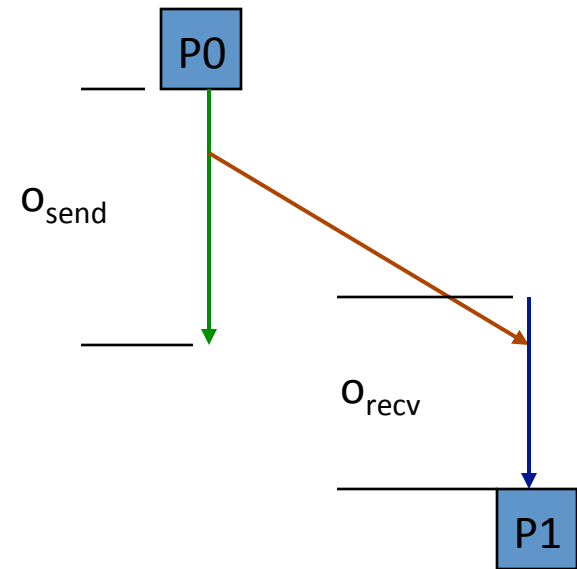
Overhead Can't be Tolerated

Modified LogGP Model

- **LogGP: no overlap**

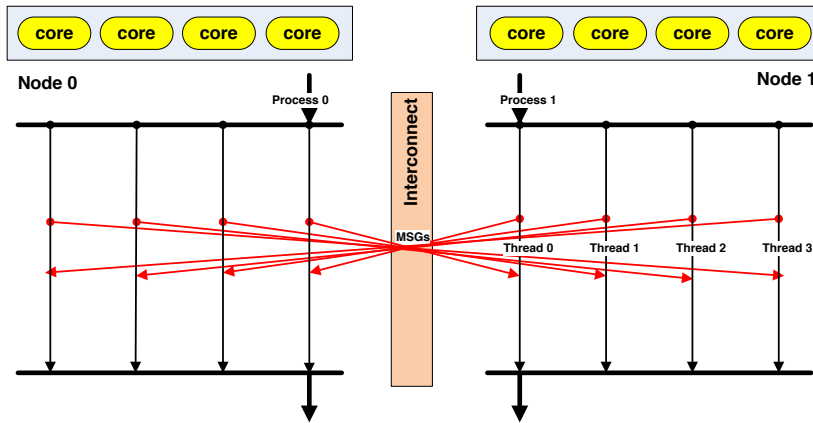


- **Observed: overheads can overlap: L can be negative**

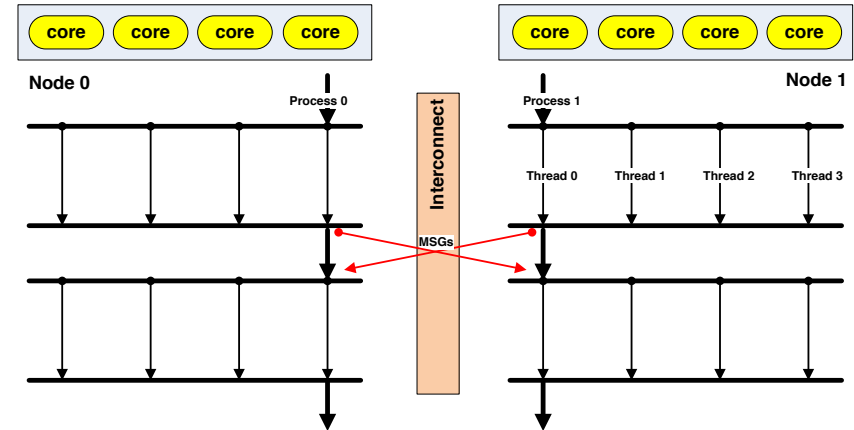


EEL: end to end latency (instead of transport latency L)
g: minimum time between small message sends
G: additional gap per byte for larger messages

Communication and Manycore: the problem is the “+”



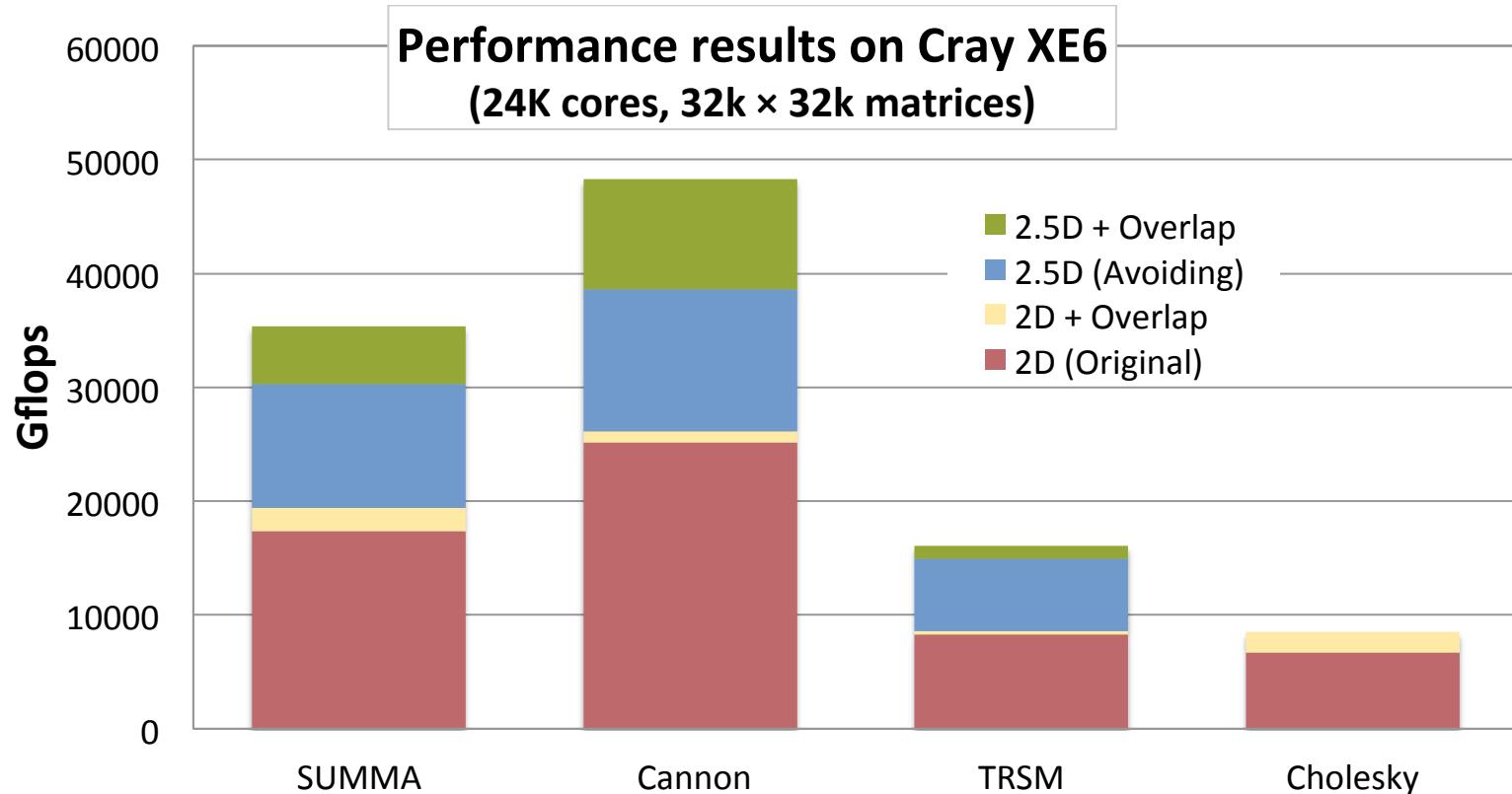
Ideal hybrid programming



Default hybrid programming

- **MPI + X today:**
 - Communicate on one lightweight core
 - Reverse offload to heavyweight core
- **MPI stack may not run well on lightweight cores**
- **Issues preventing efficient interoperability:**
 - Addressability: can't name remote threads?
 - Separability: How to manage communication resources for independent paths
- **More feasible for 1-sided than 2-sided**

Communication Overlap Complements Avoidance



Even with communication-optimal algorithms (minimized bandwidth) there are still benefits to overlap and other things that speed up networks

SC'12 paper (Georganas, González-Domínguez, Solomonik, Zheng, Touriño, Yelick)

Avoid Unnecessary Synchronization

Sources of Unnecessary Synchronization

Loop Parallelism

```
!$OMP PARALLEL DO
  DO I=2,N
    B(I) = (A(I) + A(I-1)) / 2.0
  ENDDO
!$OMP END PARALLEL DO
```

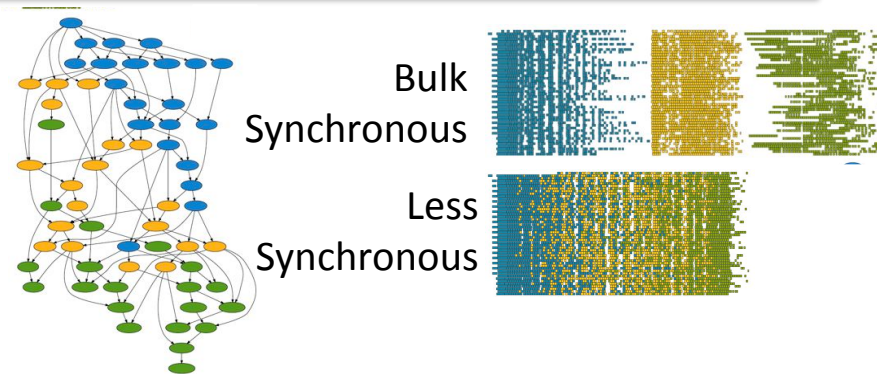
“Simple” OpenMP parallelism implicitly synchronized between loops

Libraries

Analysis	% barriers	Speedup
Auto	42%	13%
Guided	63%	14%

NWChem: most of barriers are unnecessary (Corvette)

Abstraction



LAPACK: removing barriers ~2x faster (PLASMA)

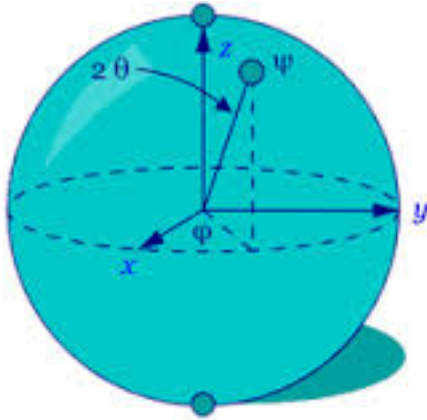
Accelerator Offload

```
!$acc data copyin(cix,ci1,ci2,ci3,ci4,ci5,ci6,ci7,ci8,ci9,ci10,ci11,&
!$acc& ci12,ci13,ci14,r,b,uxyz,cell,rho,grad,index_max,index,&
!$acc& ciy,ciz,wet,np,streaming_sbuf1, &
!$acc& streaming_sbuf1,streaming_sbuf2,streaming_sbuf4,streaming_sbuf5,&
!$acc& streaming_sbuf7s,streaming_sbuf8s,streaming_sbuf9n,streaming_sbuf10s,&
!$acc& streaming_sbuf11n,streaming_sbuf12n,streaming_sbuf13s,streaming_sbuf14n,&
!$acc& streaming_sbuf7e,streaming_sbuf8w,streaming_sbuf9e,streaming_sbuf10e,&
!$acc& streaming_sbuf11w,streaming_sbuf12e,streaming_sbuf13w,streaming_sbuf14w, &
!$acc& streaming_rbuf1,streaming_rbuf2,streaming_rbuf4,streaming_rbuf5, &
!$acc& streaming_rbuf7n,streaming_rbuf8n,streaming_rbuf9s,streaming_rbuf10n, &
!$acc& streaming_rbuf11s,streaming_rbuf12s,streaming_rbuf13n,streaming_rbuf14s,&
!$acc& streaming_rbuf7w,streaming_rbuf8e,streaming_rbuf9w,streaming_rbuf10w, &
!$acc& streaming_rbuf11e,streaming_rbuf12w,streaming_rbuf13e,streaming_rbuf14e, &
!$acc& send_e,send_w,send_n,send_s,recv_e,recv_w,recv_n,recv_s)
```

The transfer between host and GPU can be slow and cumbersome, and may (if not careful) get synchronized

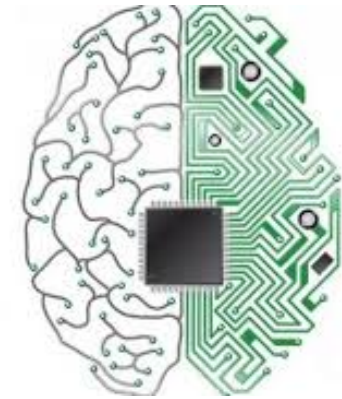
Beyond Moore

Beyond Digital Computing Law



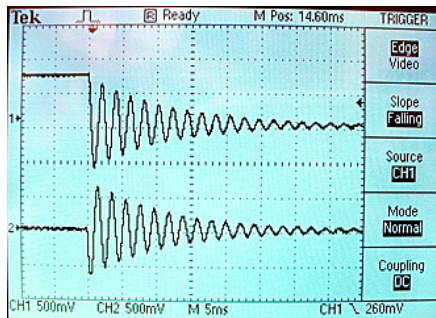
Quantum

Is there a new model of computing that is useful for science?



Neuromorphic

Are there ways of storing, transferring and computing on information that significantly reduce power?

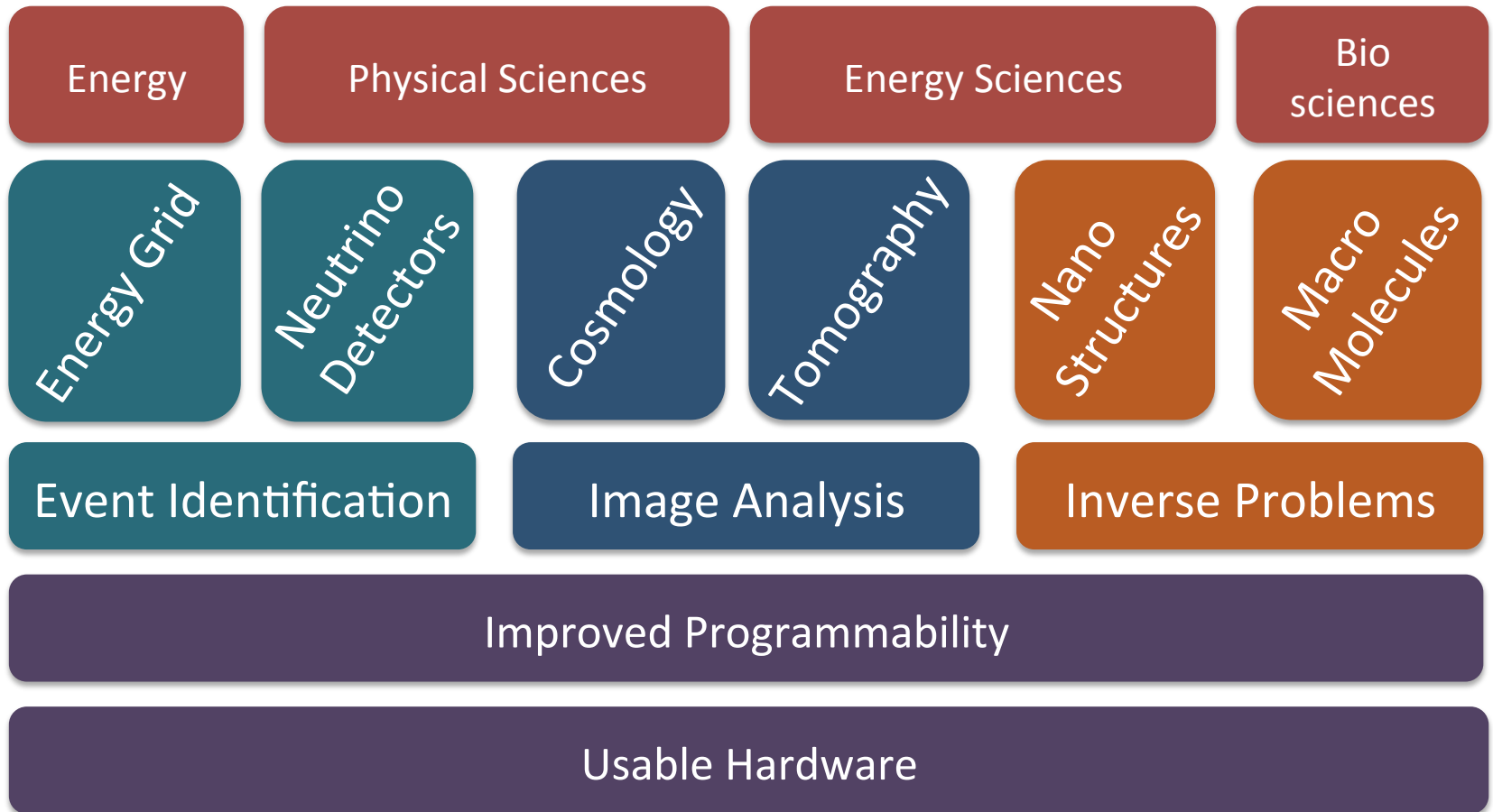


Analog



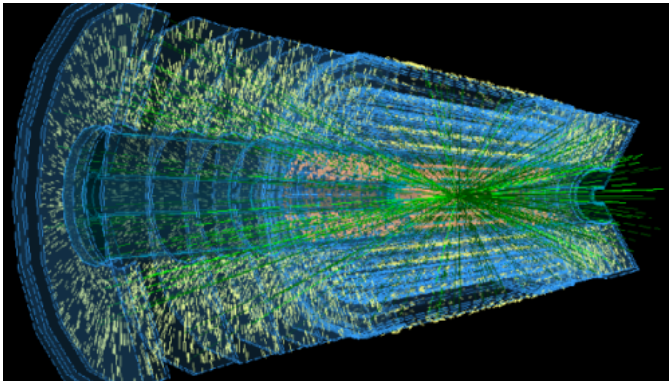
FPGAs

Science Applications of Neuromorphic Computing



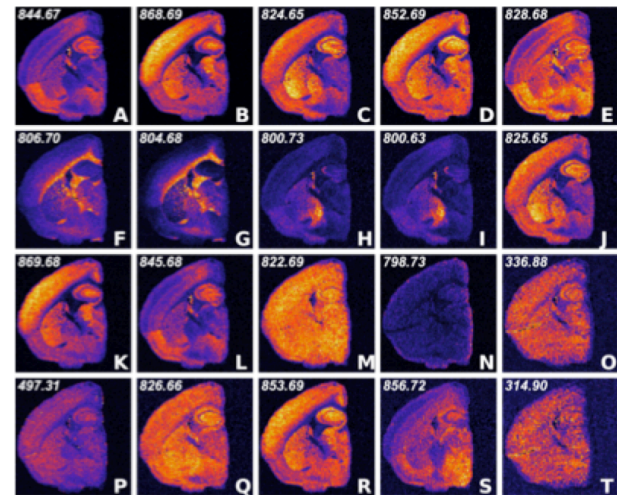
Data processing with special purpose hardware

- General trend towards specialization for continued performance growth
- Data processing (on raw data) will be first in science



Particle Tracking with Neuromorphic chips

Computing in Detectors



Deep learning processors for image analysis

FPGAS for genome analysis

And can we also use these for simulation?

QUANTUM INFORMATION SCIENCE (QIS): AN APPLICATIONS ORIENTED VIEW

QIS

Computing (~100-100,000)

- Gate Based: Shor, Grover,...
- Adiabatic Quantum Computing
- Quantum Annealing

Simulations (~1-100)

- Chemical & Materials Science
- Theoretical Physics: Cosmology,...

Communication (~1, flying)

- Quantum Key Distribution
- Quantum Commitment

Metrology (~1-10)

- Precision measurements, squeezing
- Sensors (Magnetic, Charge, Light)

Electronic Structure Methods for Chemistry

Method	Scaling for N electrons
DFT	$O(N^2)$ - $O(N^3)$
HF	$O(N^2)$ - $O(N^4)$
MP2	$O(N^5)$
CISD, CCSD	$O(N^6)$
CCSD(T)	$O(N^7)$
CCSDT	$O(N^8)$
FCI	$O(\exp(N))$

Improving Quantum Algorithms for Quantum Chemistry

M. B. Hastings,^{1,2} Dave Wecker,² Bela Bauer,¹ and Matthias Troyer³

¹Station Q, Microsoft Research, Santa Barbara, CA 93106-6105, USA
²Quantum Architectures and Computation Group, Microsoft Research, Redmond, WA 98052,
³Theoretische Physik, ETH Zurich, 8093 Zurich, Switzerland

We present several improvements to the standard Trotter-Suzuki based algorithms used in simulation of quantum chemistry on a quantum computer. First, we modify how Jordan-Wigner transformations are implemented to reduce their cost from linear or logarithmic in the number of orbitals to a constant. Our modification does not require additional ancilla qubits. Then we demonstrate how many operations can be performed in parallel, reducing the parallel depth of the circuit, at the cost of increasing the number of qubits required. Thirdly, we modify the Trotter-Suzuki decomposition to reduce errors introduced by the approximation, and validate using numerical simulation.

Scaling for N electrons on Quantum Device

FC #1	$O(N^9)$
FC #2	$O(N^7)$
FC #3	$O(N^{5.5})$

Full configuration interaction

Quantum Simulation:

“What quantum computers do in their sleep” [Scott Aaronson]

Open questions in theory and practice

End Game for Moore's Law

More parallelism

More specialization

(hardware and programming models)

Less communication

Understand your applications!

Thank you!