

The Analysis of Impact of Energy Efficiency Requirements on Programming Environments

John Shalf

*Department Head for Computer Science
NERSC CTO
Lawrence Berkeley National Laboratory*

*SC12 Workshop on Energy Efficient HPC
Nov 11, 2012*



Lawrence Berkeley
National Laboratory

Outline

- **Programming Models are a Reflection of the Underlying Machine Architecture**
 - *Express what is **important** for performance*
 - *Hide complexity that is **not** consequential to performance*
- **Programming Models are Increasingly Mismatched with Underlying Hardware Architecture**
 - *Changes in computer architecture trends/costs*
 - *Performance and programmability consequences*
- **The reason for the mismatch is the increasingly power constrained nature of future machine architectures**
 - *Peter Kogge described how data movement is biggest cost factor*
 - *Programming environments and algorithm design is rapidly moving from conserving FLOPs to conserving data movement*
- **Recommendations on Reformulating Programming Environment *together with Hardware Support for Efficiency***
 - *One school of thought says we try to control energy states of HPC*
 - *Alternative is to design to maximize data movement efficiency*

Goal for Programmers at All Levels

(NNSA Exascale Roadmapping Workshop in SF, 2011)

- **Minimize the number of lines of code I have to change when we move to next version of a machine**
 - Evidence that current abstractions are broken are entirely related to effort required to move to each new machine
 - ***Target is the FIRST DERIVATIVE of technology changes!!!***
- **What is changing the fastest (*what do we want to make future pmodels less sensitive to*)**
 - **Insensitive to # cores** (*but unclear if as worried about # of nodes*)
 - **Less sensitive to sources of non-uniformity** (*execution rates and heterogeneous core types*)
 - **Memory capacity/compute ratio** (*strong'ish' - scaling*)
 - **Data Movement Constraints**
 - Increasingly distance-dependent cost of data movement
 - Topological constraints (node-scale & system-wide)
 - Expressed as NUMA domains (within node)

What are durable abstractions (abstract machine model) for HIDING or MITIGATING these design trends?



U.S. DEPARTMENT OF
ENERGY

Office of
Science



What is an Abstract Machine Model?

Definition: An Abstract Machine model represents the machine attributes that will be important to reasoning about code performance

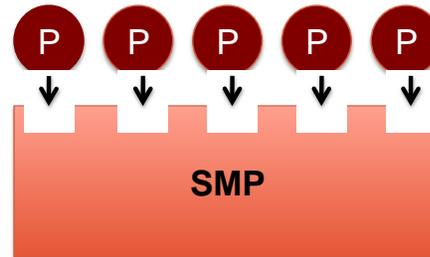
- Enables us to reason about how to map algorithm efficiently onto underlying machine architecture
- Enables us to reason about power/performance trade-offs for different algorithm or execution model choices
- Want model to be as simple as possible, but not neglect any aspects of the machine that are important for performance

Has been relatively consistent in HPC for many years
Pax MPI

The Programming Model is a Reflection of the Underlying *Abstract Machine Model*

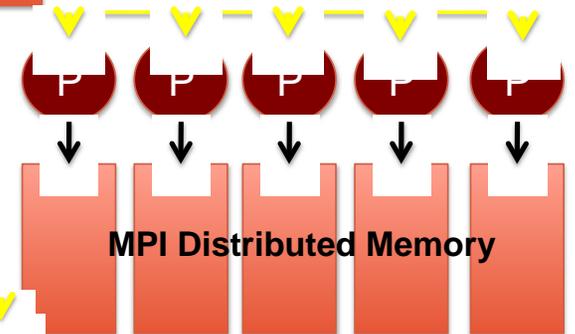
- **Equal cost SMP/PRAM model**

- No notion of non-local access
- `int [nx][ny][nz];`



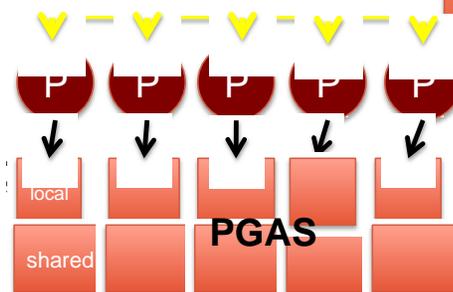
- **Cluster: Distributed memory model**

- No unified memory
- `int [localNX][localNY][localNZ];`



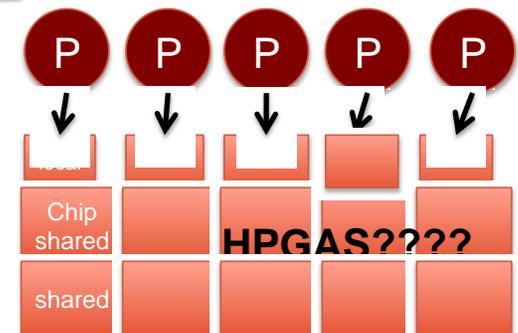
- **PGAS for horizontal locality**

- Data is LOCAL or REMOTE
- `shared [Horizontal] int [nx][ny][nz];`



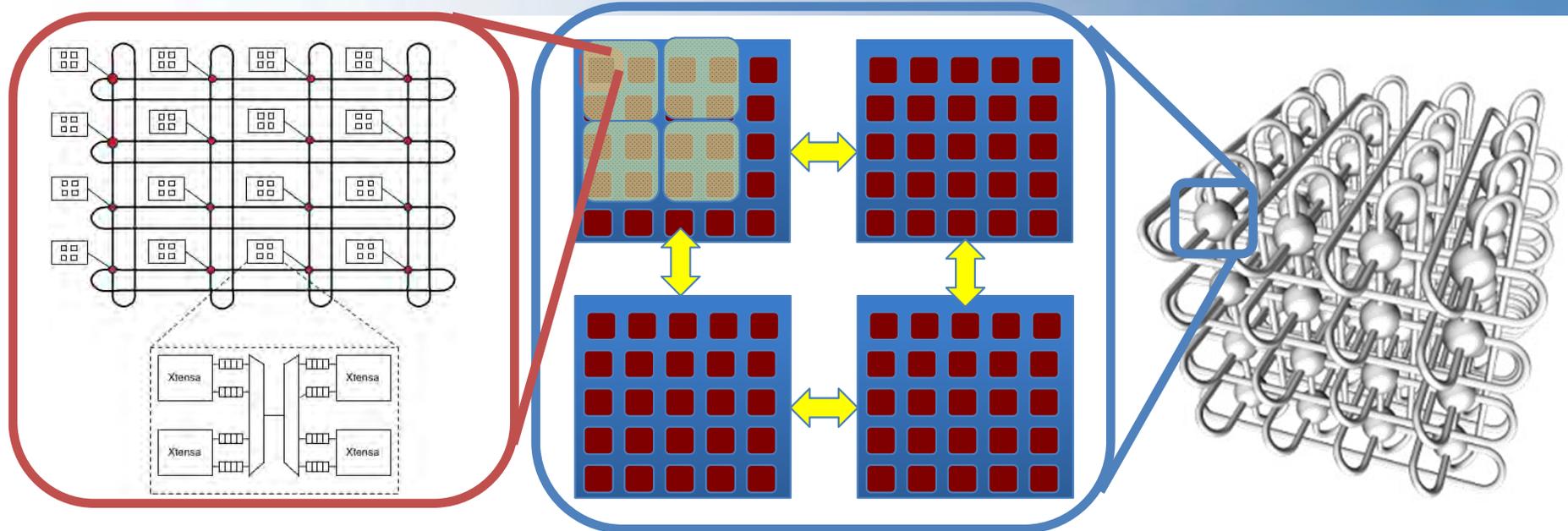
- **HPGAS for vertical data movement**

- Depth of hierarchy also matters now
- `shared [Vertical][Horizontal] int [x][y][z];?`



Parameterized Machine Model

(what do we need to reason about when designing a new code?)



Cores

- How Many
- Heterogeneous
- SIMD Width

Network on Chip (NoC)

- Are they equidistant or
- Constrained Topology (2D)

On-Chip Memory Hierarchy

- Automatic or Scratchpad?
- Memory coherency method?

Node Topology

- NUMA or Flat?
- Topology may be important
- Or perhaps just distance

Memory

- Nonvolatile / multi-tiered?
- Intelligence in memory (or not)

Fault Model for Node

- FIT rates, Kinds of faults
- Granularity of faults/recovery

Interconnect

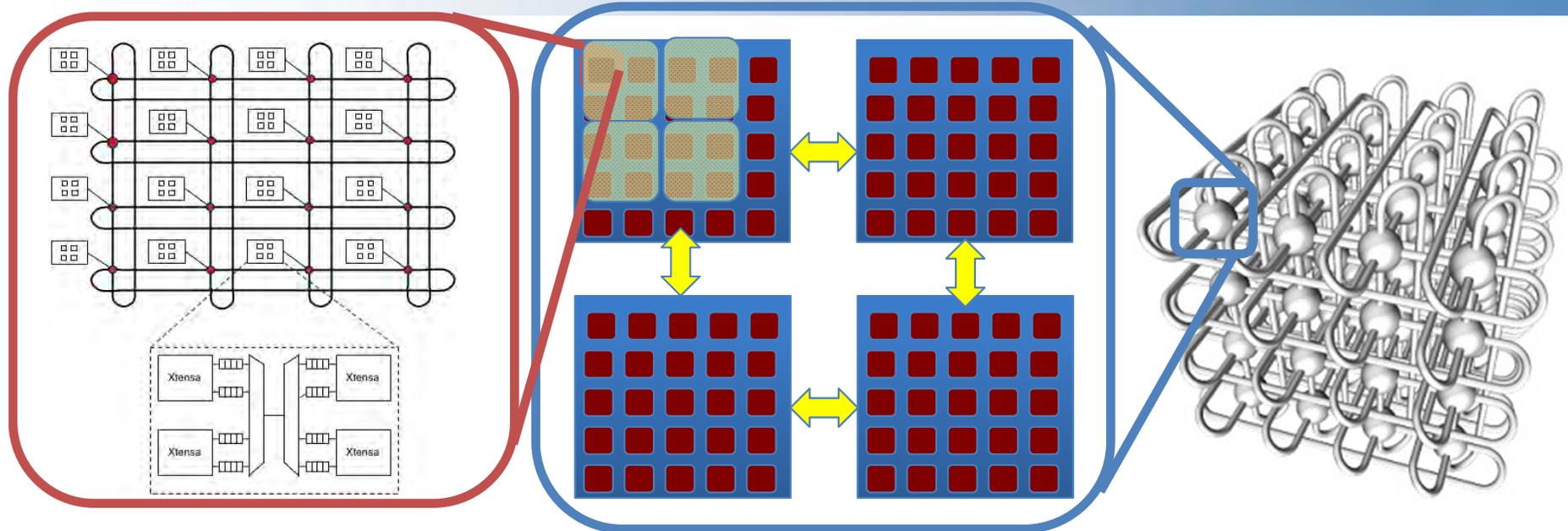
- Bandwidth/Latency/Overhead
- Topology

Primitives for data movement/sync

- Global Address Space or messaging?
- Synchronization primitives/Fences

Parameterized Machine Model

(what do we need to reason about when designing a new code?)



For each parameterized machine attribute, can

- **Ignore it:** If ignoring it has no serious power/performance consequences
- **Abstract it (*virtualize*):** If it is well enough understood to support an automated mechanism to optimize layout or schedule
 - This makes programmers life easier (one less thing to worry about)
- **Expose it (*unvirtualize*):** If there is not a clear automated way of make decisions
 - Must involve the human/programmer in the process (*make pmodel more expressive*)
 - Directives to control data movement or layout (for example)

Want model to be as simple as possible, but not neglect any aspects of the machine that are important for performance

Data Movement

The problem with Wires:

Energy to move data proportional to distance

- **Cost to move a bit on copper wire:**

- **Power = bitrate * Length / cross-section-area**

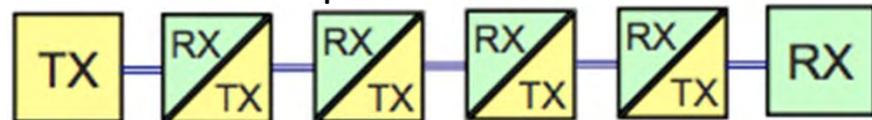


- **Wire data capacity constant as feature size shrinks**
- ***Cost to move bit proportional to distance***
- ***~1-5TByte/sec max feasible off-chip BW (10-20GHz/pin)***
- ***Photonics is a wildcard***

Photonics requires no redrive
and passive switch little power



Copper requires to signal amplification
even for on-chip connections



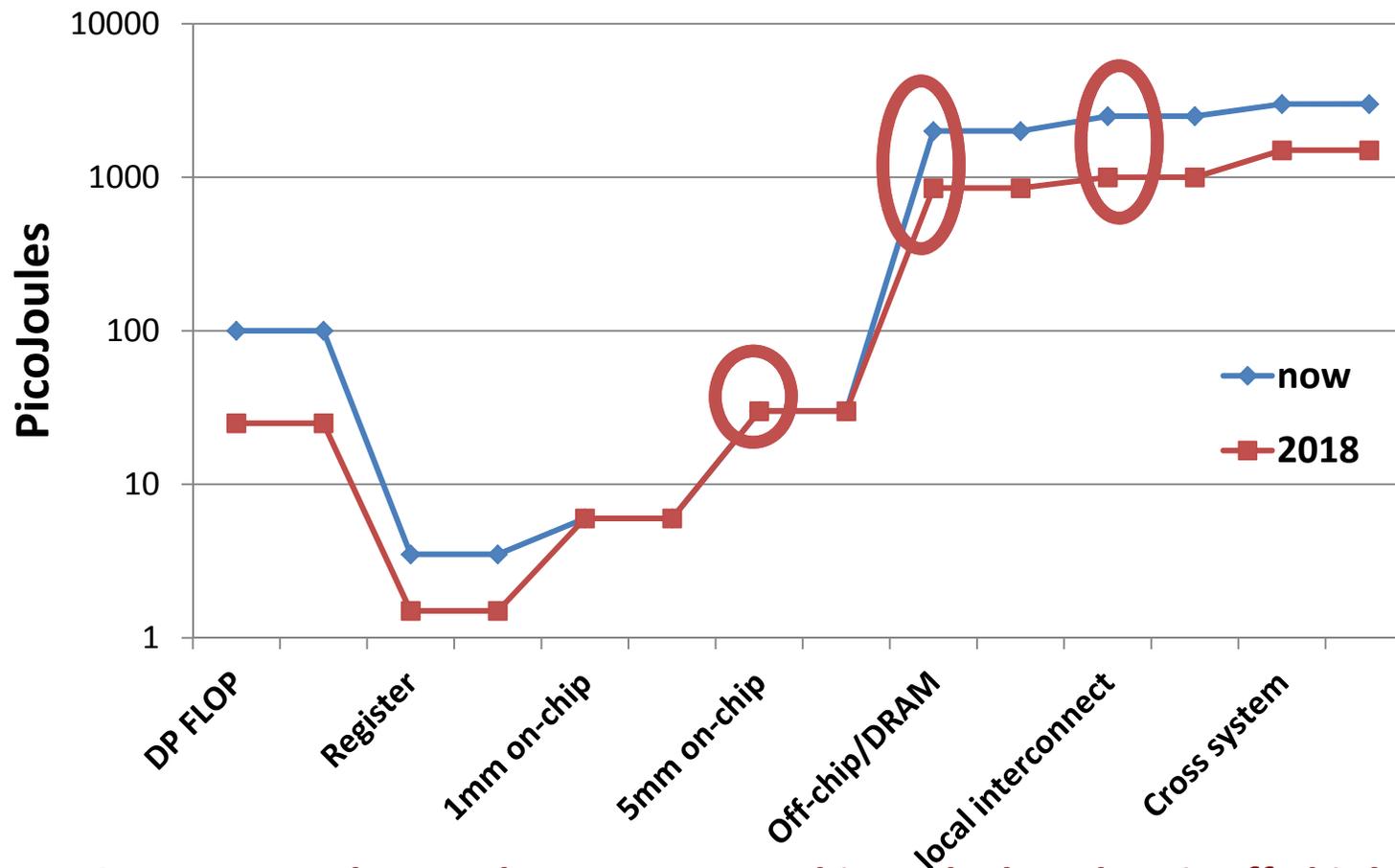
U.S. DEPARTMENT OF
ENERGY

Office of
Science



Data Movement Costs

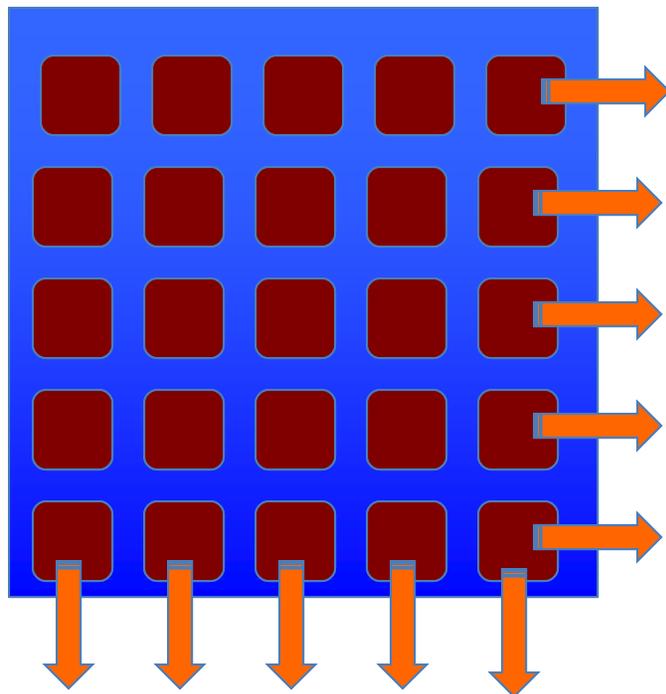
Energy Efficiency will require careful management of data locality



Important to know when you are on-chip and when data is off-chip!

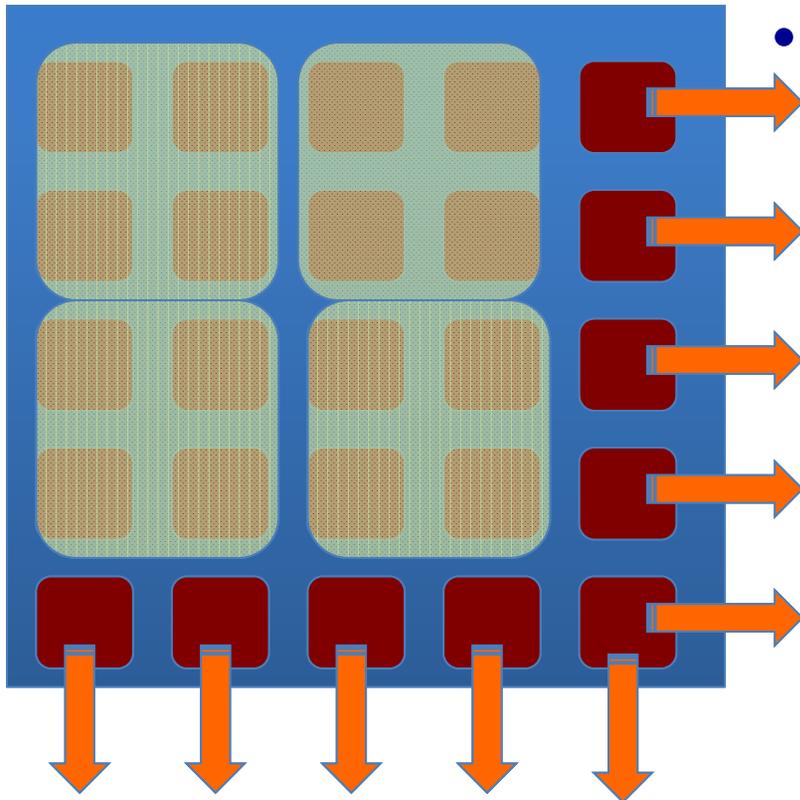
Future of On-Chip Architecture

(Nov 2009 DOE arch workshop)



- **~1000-10k simple cores**
- **4-8 wide SIMD or VLIW bundles**
- **Either 4 or 50+ HW threads**
- **On-chip communication Fabric**
 - Low-degree topology for on-chip communication (torus or mesh)
 - *Can we scale cache-coherence?*
 - HW msg. passing
 - Global (possibly nonCC memory)
 - Shared register file (clusters)
- **Off-chip communication fabric**
 - Integrated directly on an SoC
 - Reduced component counts
 - Coherent with TLB (no pinning)

Cost of Data Movement



- **Cost of moving long-distances on chip motivates clustering on-chip**

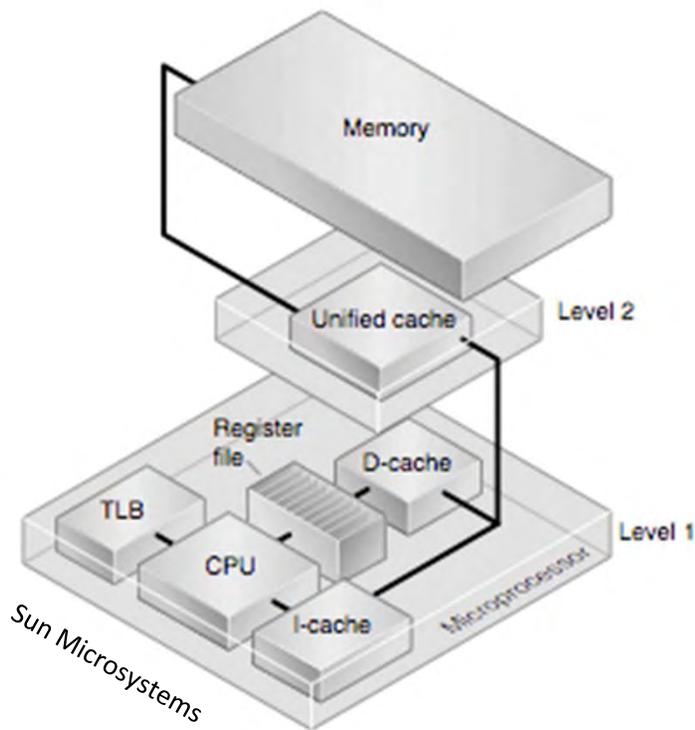
- 1mm costs ~6pj (today & 2018)
- 20mm costs ~120 pj (today & 2018)
- FLOP costs ~100pj today
- FLOP costs ~25pj in 2018

Different Architectural Directions

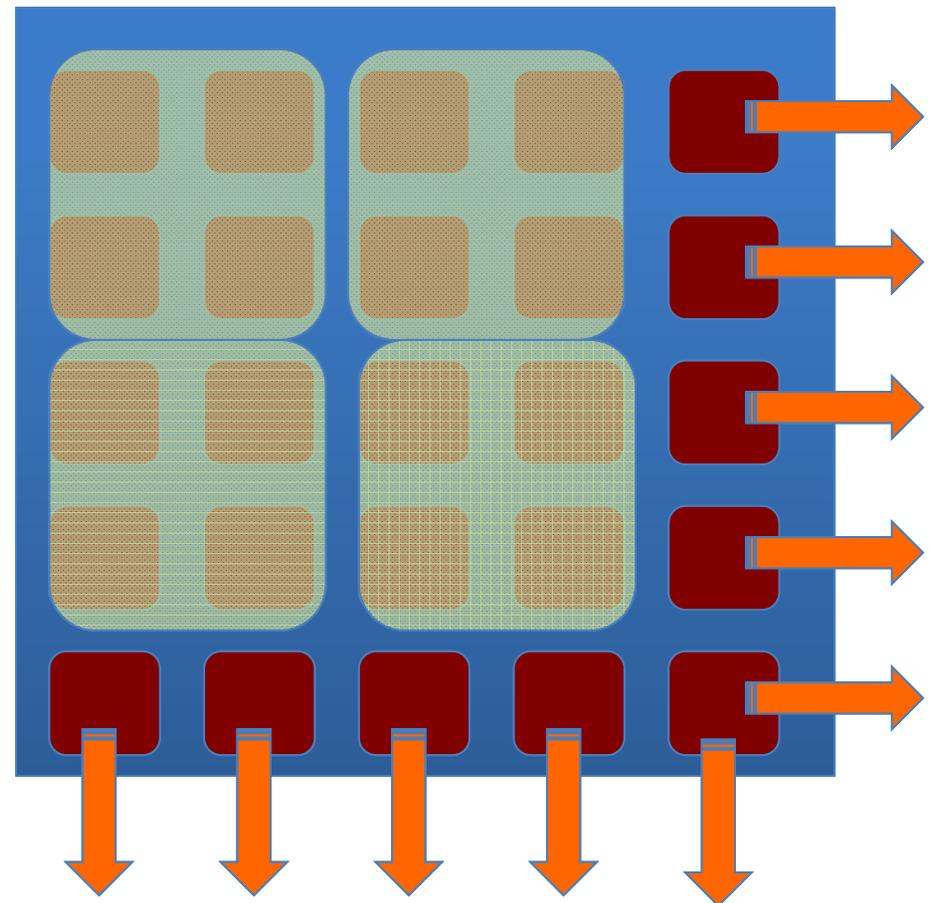
- GPU: WARPs of hardware threads clustered around shared register file
- CMP: limited area cache-coherence
- CMT: hardware multithreading clusters

Data Locality Management

Vertical Locality Management (spatio-temporal optimization)



Horizontal Locality Management (topology optimization)

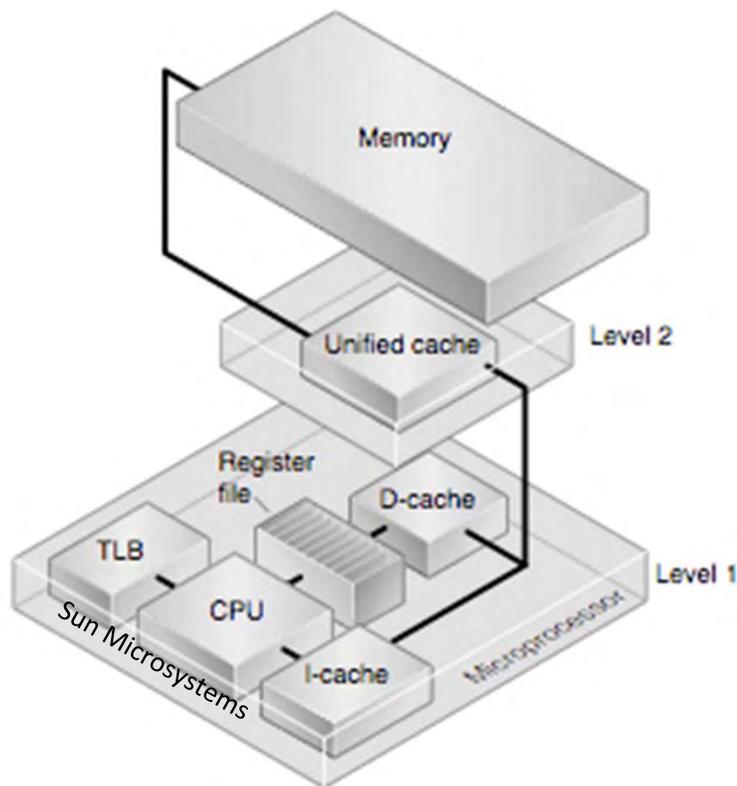


Shifting our Programming Paradigm to Reflect Emerging Design Constraints

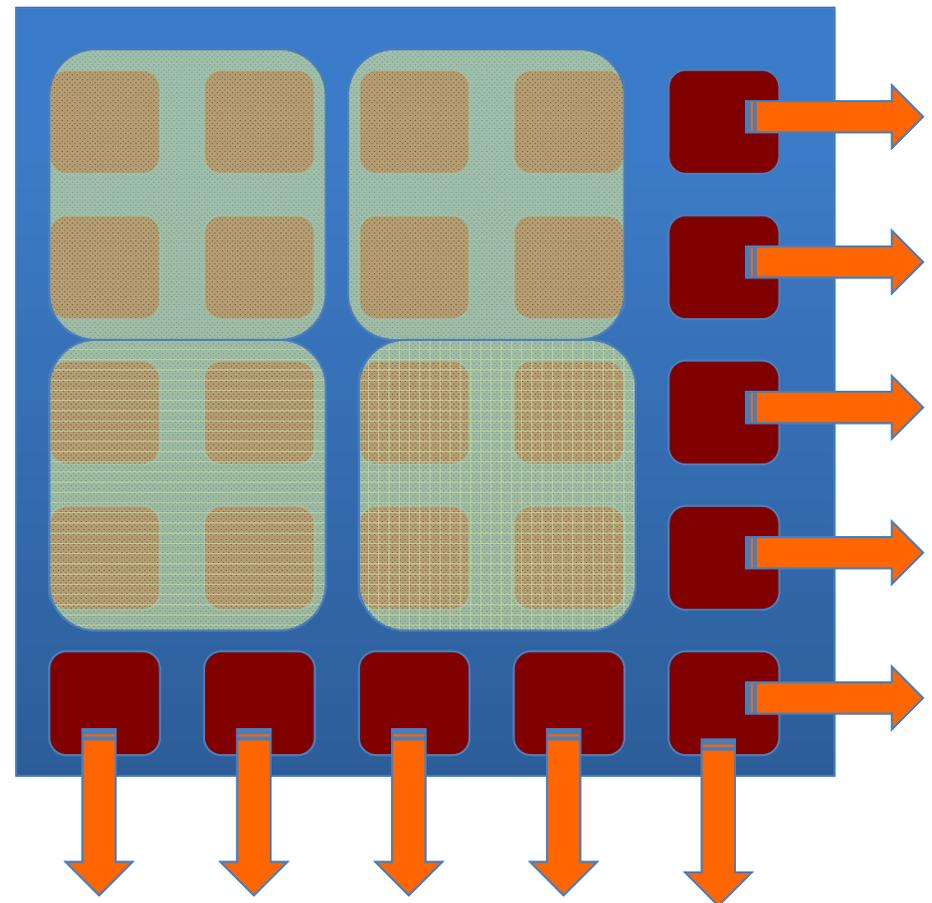
- **Old Mental Model -- *Reduce FLOPs***
 - FLOPS used to be the most expensive (conserve what is expensive)
 - Concern about sustained-to-peak performance (% of peak flop rate)
- **Technology Trends (*are mismatched with current pmodel*)**
 - Cost of data movement rising faster than cost of a flop. (*IKEA FLOPs*)
 - *New costs center around vertical and horizontal data movement*
- **New Approaches need *CoDesign* of Hardware/Software mechanisms for a complete programming environment**
 - **Communication Avoiding Algorithms and Average Communication Distance Model**
 - **More expressive type-systems to express data layouts**
 - Enables compilers and runtimes info to reason about data layout
 - **Functional Semantics to simplify automated data movement**
 - Make data volume and movement trivial to identify and compute
 - Make tedious `CUDA_copy` and `ACC` data movement directives go away

Data Locality Management

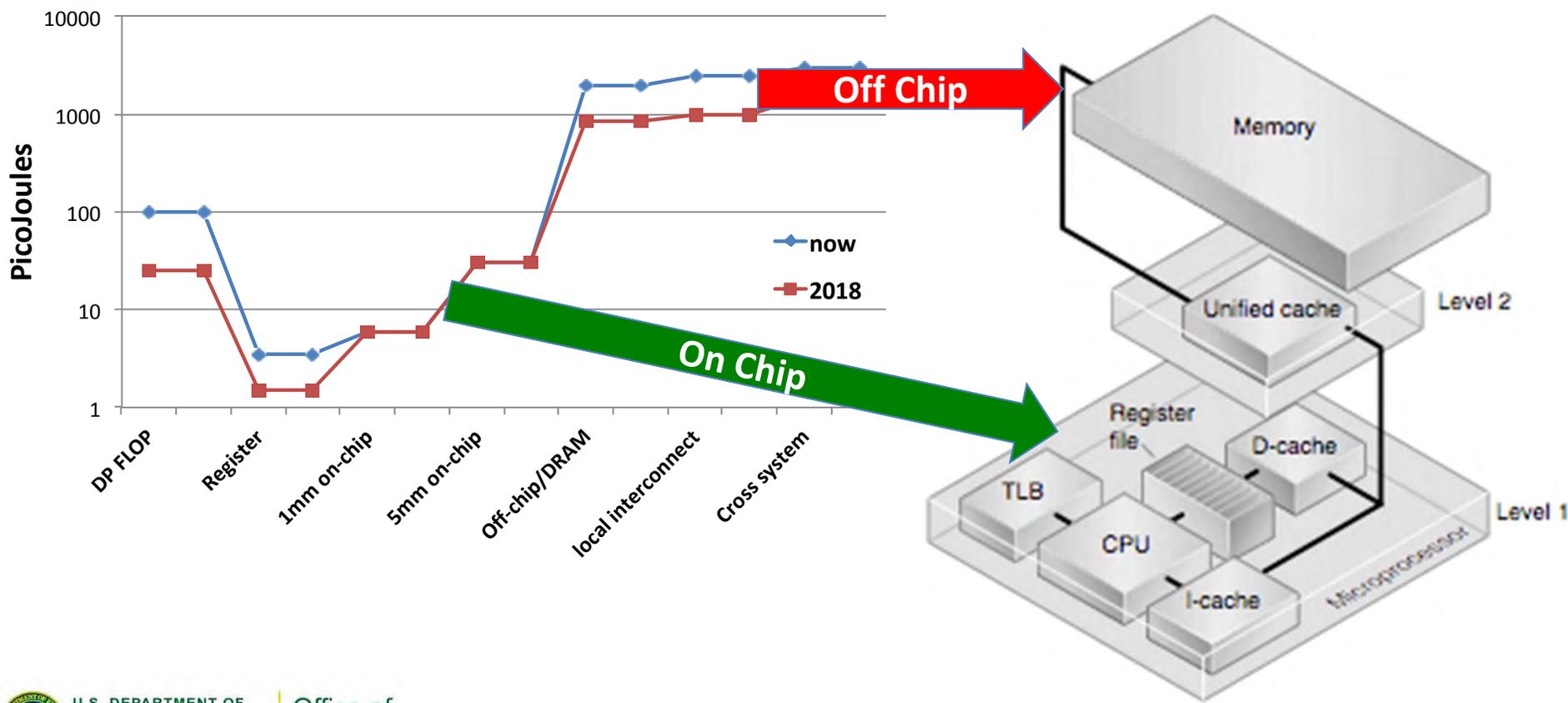
Vertical Locality Management (spatio-temporal optimization)



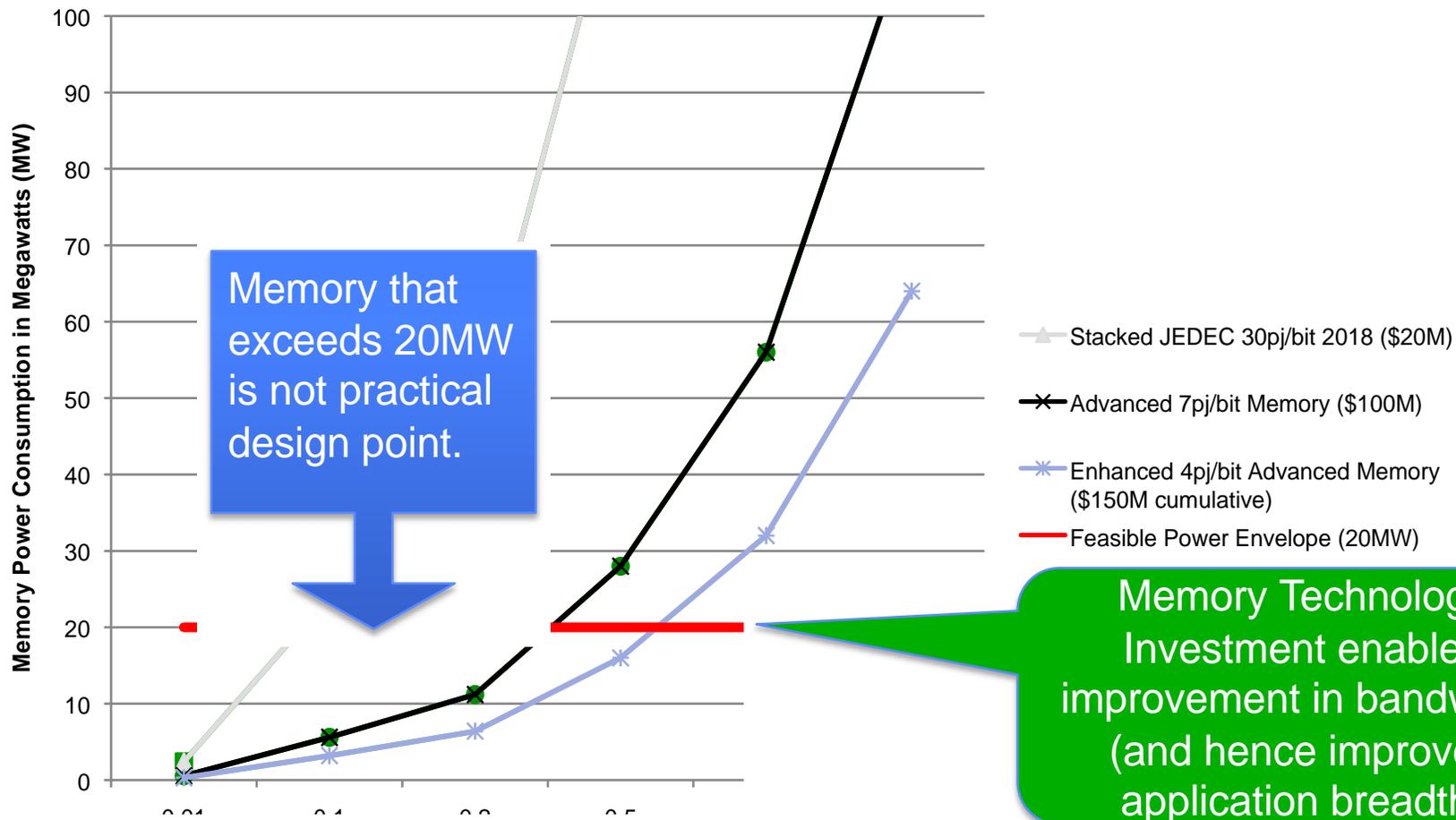
Horizontal Locality Management (topology optimization)



Hardware/Software for Managing Vertical Data Locality



Memory Bandwidth

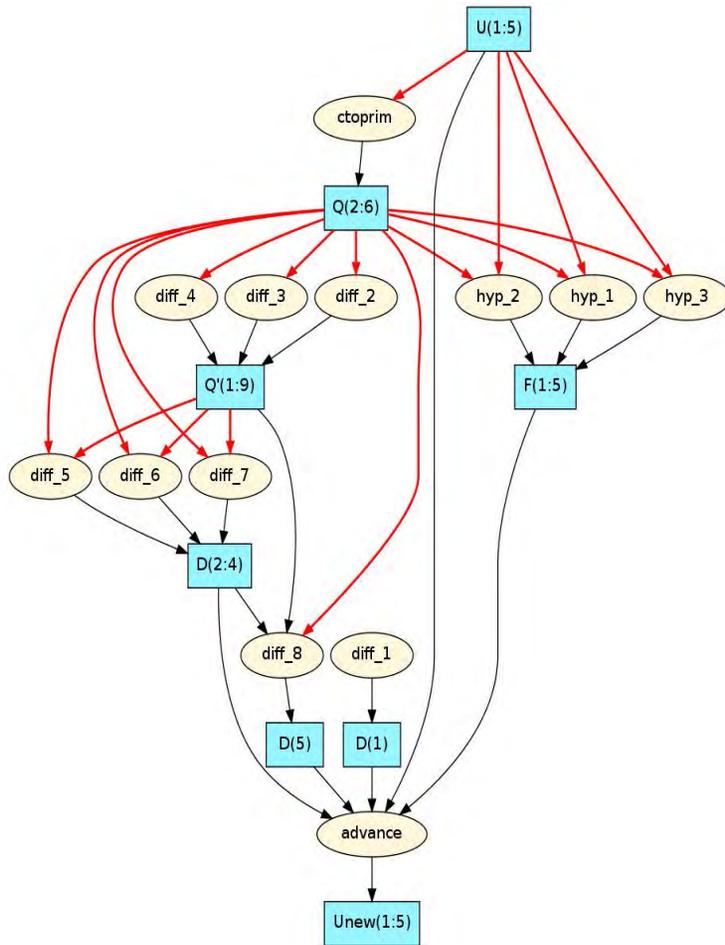


Application performance and breadth pushes us to higher

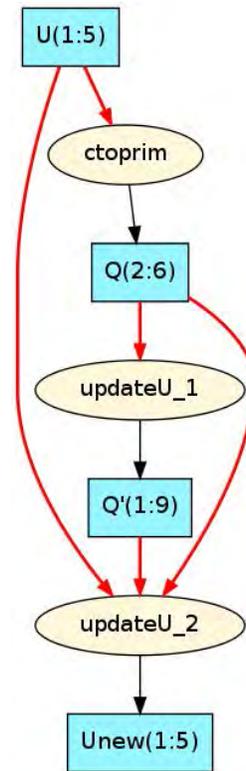
Power pushes us to lower bandwidth

Loop Fusion To Reduce Memory Bandwidth

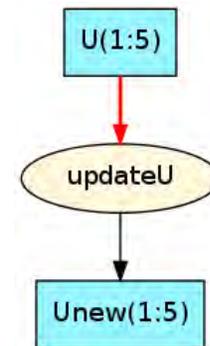
“use cache as bandwidth filter”



Baseline
2.9 GB/sweep
1.78 Bytes/Flop



Simple Fusion
1.6 GB/sweep (-46%)
0.96 Bytes/Flop



Aggressive Fusion
0.48 GB/sweep (-84%)
0.29 Bytes/Flop

Note: This is not traditional fusion. Current compilers models are not up to this task.

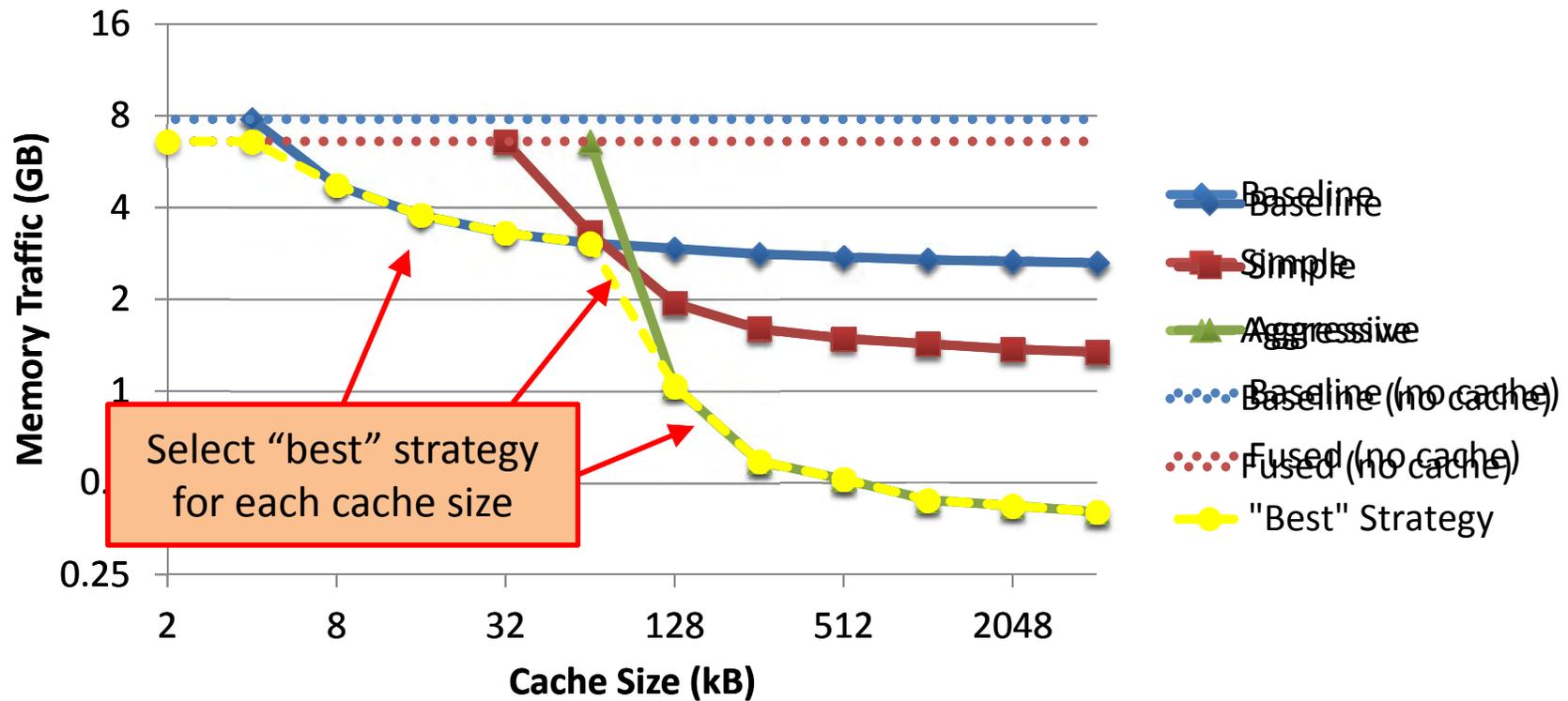
But how much is it worth to fix them?

Codesign Question: *How Much Cache Should I have?*

More Cache reduces memory bandwidth requirements

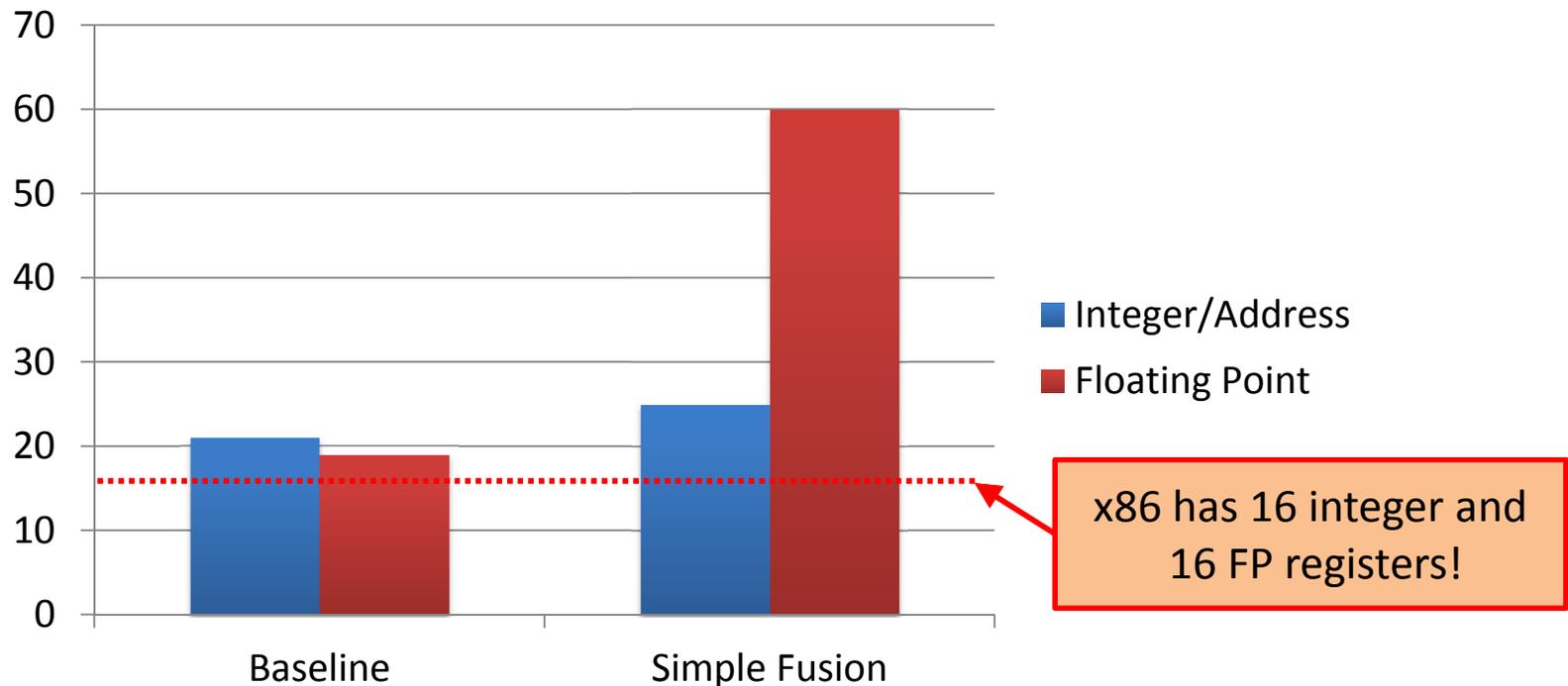
But consumes surface area, so need to give up some processor cores or other services

Memory Traffic vs Cache Size for Loop Fusion Scenarios ("best" block size)



Codesign Question:

How many registers should I have?

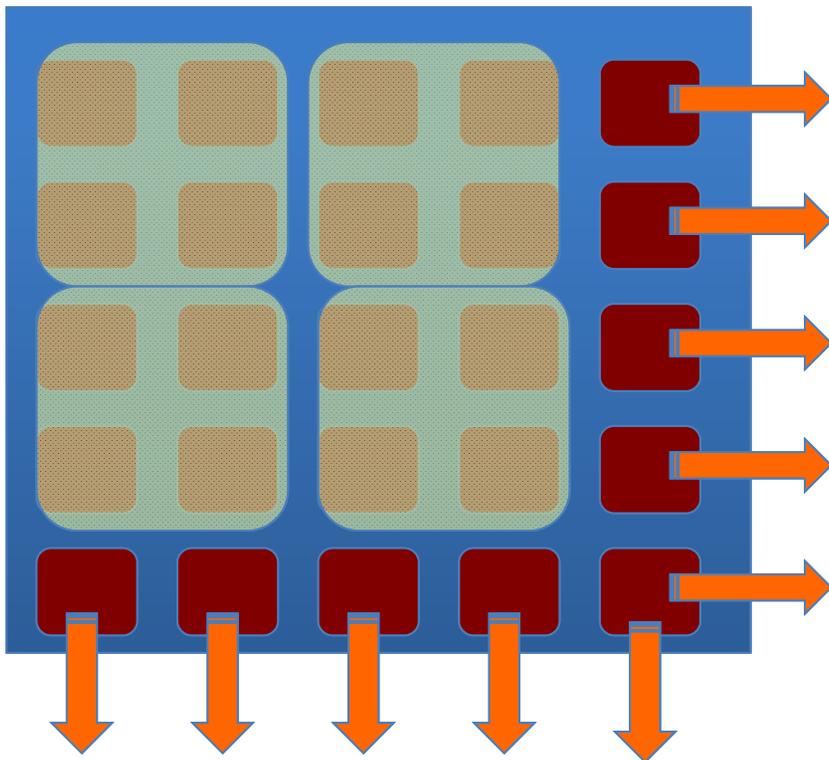


- If not enough registers available to hold state, registers spill into the L1 cache, increasing cache traffic and possibly affecting performance

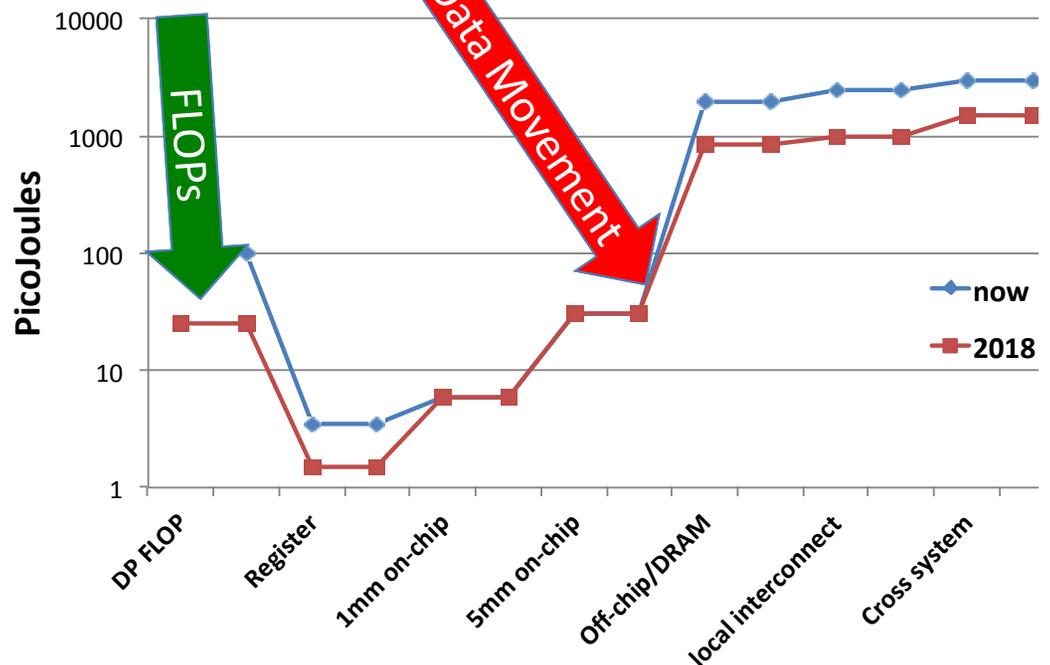
Conclusions on Vertical Locality Management

- **Aggressive Fusion is essential to lower memory bandwidth requirements**
 - But to get the advantage need large L1 cache (would need to be scratchpad to be feasible)
 - Also requires larger register file
 - And requires new programming paradigm to enable aggressive fusion (functional semantics or other hints to facilitate compiler analysis)
- **Benchmarking on current architectures would have missed this opportunity**
 - Requires predictive modeling and architectural simulation
 - This is the center of codesign
- **Many of the most valuable hardware opportunities identified by codesign will have major impact on our programming paradigm!**
 - Its not just about transforming code and algorithms
 - Choices affect our entire paradigm for programming these systems!
 - *Must think deeper about ramifications to programming ecosystem (just as we did in the transition from vec to MPI)*

Software/Hardware Mechanisms for Managing Horizontal Data Locality



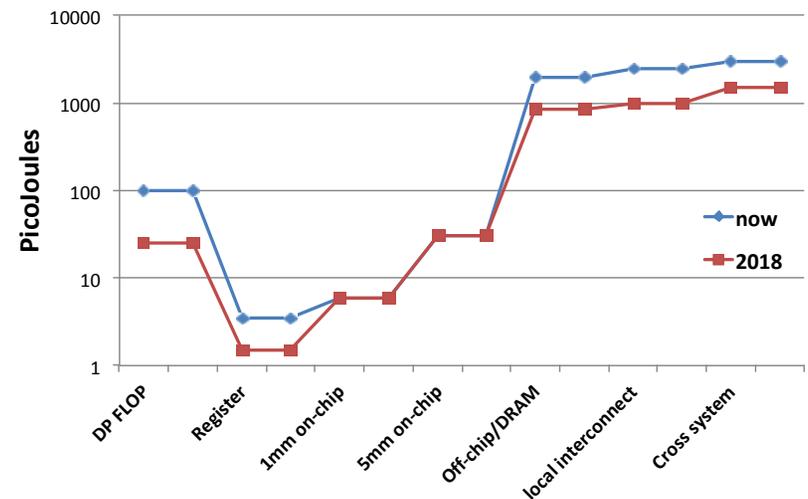
FLOPs cost more than on-chip data movement!
(NUMA)



Problems with Existing Abstractions for Expressing Locality

- Our current programming models assume all communicating elements are equidistant (PRAM)
 - OpenMP, and MPI each assume flat machine at their level of parallelism

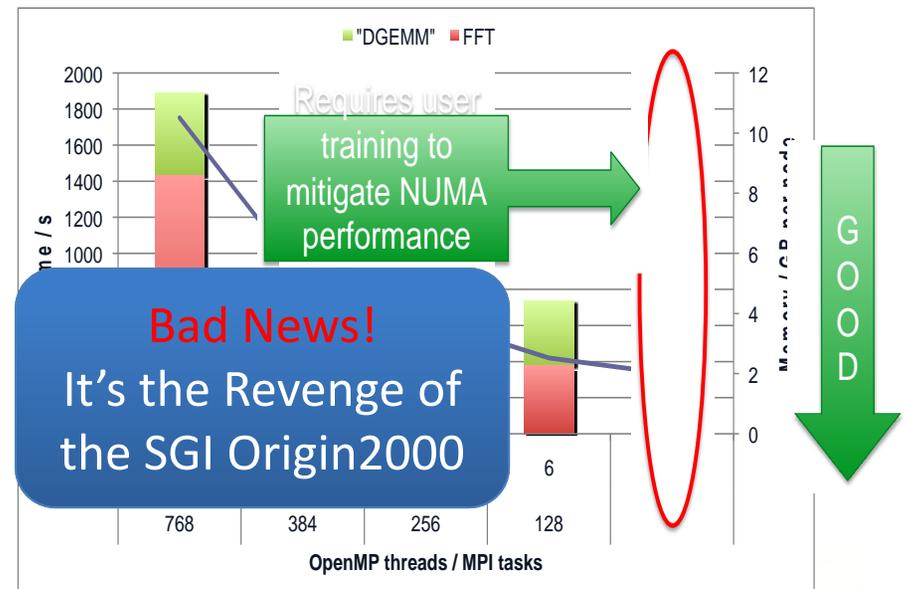
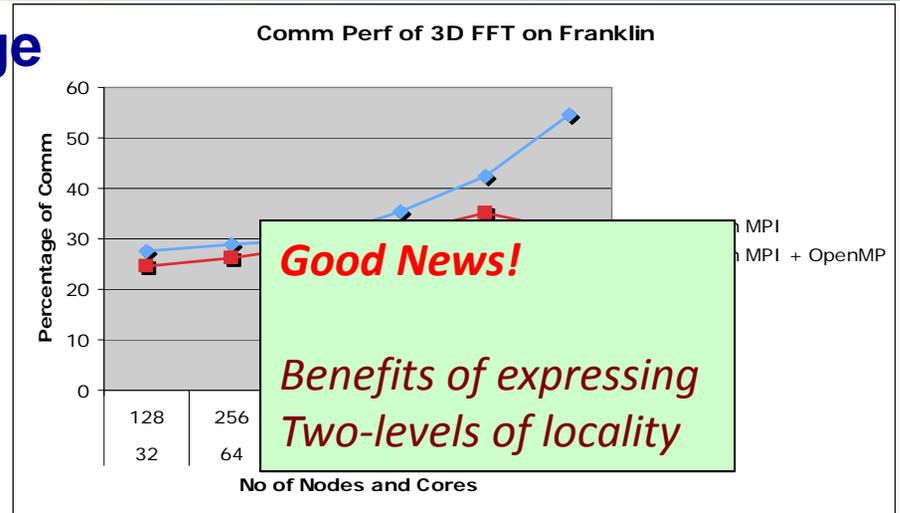
- But the machine is not flat!!!
 - Lose performance because expectation and reality are mismatched
 - *Pmodel does not match underlying machine model!!!*



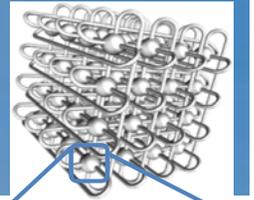
- What is wrong with Flat MPI?
 - 10x higher bandwidth between cores on chip
 - 10x lower latency between cores on chip
 - If you pretend that every core is a peer (each is just a generic MPI rank) you are leaving a lot of performance on the table
 - You cannot domain-decompose things forever

Current Practices (MPI+X)

- **MPI+OMP Hybrid recognizes huge cost for going off-chip**
- **Hybrid Model improves 3D FFT communication performance**
 - Enables node to send larger messages between nodes
 - Substantial improvements in communications efficiency
- **But OMP offers no management of data locality**
 - Huge performance penalty for ignoring NUMA effects
 - *Then programmer responsible for matching up computation with data layout!! (UGH!)*
 - *Makes library writing difficult and **Makes AMR nearly impossible!***

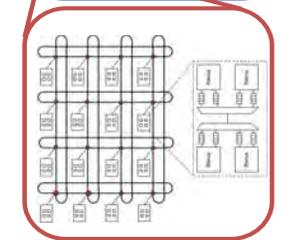
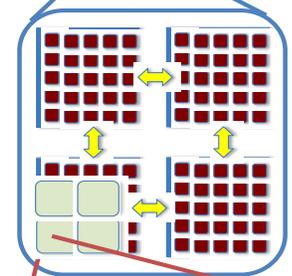


Expressing Hierarchical Layout



- **Hierarchical layout statements**

- Express mapping of “natural” enumeration of an array to the unnatural system memory hierarchy
- Maintain unified “global” index space for arrays ($A[x][y][z]$)
- Support mapping to complex address spaces
- Convenient for programmers



- **Iteration expressions more powerful when they bind to *data locality* instead of *threadnumber***

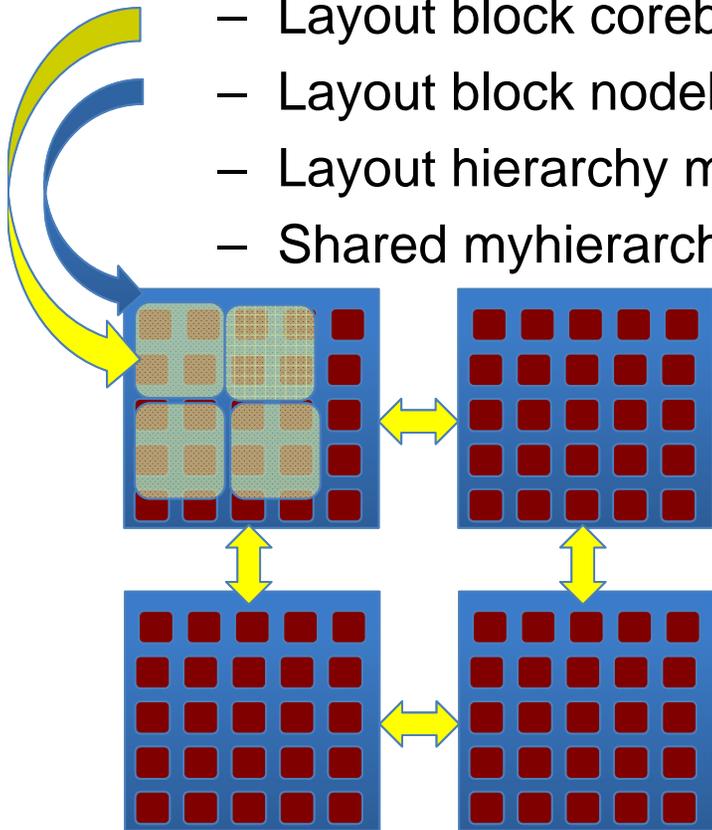
- instead of `upc_forall(;;;<threadnumber>)`
- Use `upc_forall(;;;<implicitly where Array A is local>)`

```
upc_forall(i=0;i<NX;i++;A)
  C[j]+=A[j]*B[i][j]);
```

Hierarchical Layout Statements

- **Building up a hierarchical layout**

- Layout block `coreblk {blockx,blocky};`
- Layout block `nodeblk {nnx,nnny,nnnz};`
- Layout hierarchy `myheirarchy {coreblk,nodeblk};`
- Shared `myhierarchy double a[nx][ny][nz];`



- **Then use data-localized parallel loop**

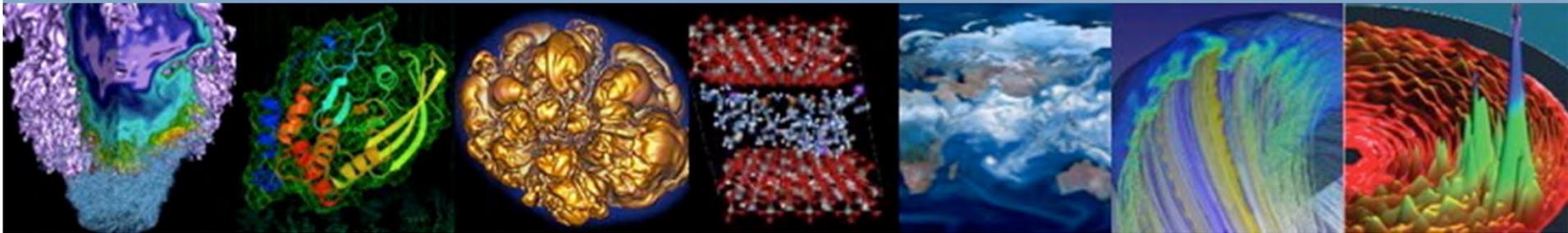
```
doall_at(i=0;i<nx;i++;a){  
  doall_at(j=0;j<ny;j++;a){  
    doall_at(k=0;k<nz;k++;a){  
      a[i][j][k]=C*a[i+1]...>
```

- *And if layout changes, this loop remains the same*

Satisfies the request of the application developers
(minimize the amount of code that changes)

Conclusions on Data Layout

- **Failure to express data locality has substantial cost in application performance**
 - Compiler and runtime cannot figure this out on its own given limited information current languages and programming models provide
- **Hierarchical data layout statements offer *better expressiveness***
 - Must be hierarchical
 - Must be multidimensional
 - Support composable build-up of layout description
- **Data-centric parallel expressions offer *better virtualization of # processors/threads***
 - Don't execute based on "thread number"
 - Parallelize & execute based on data locality
 - Enables layout to be specified in machine-dependent manner without changing execution



Interconnects

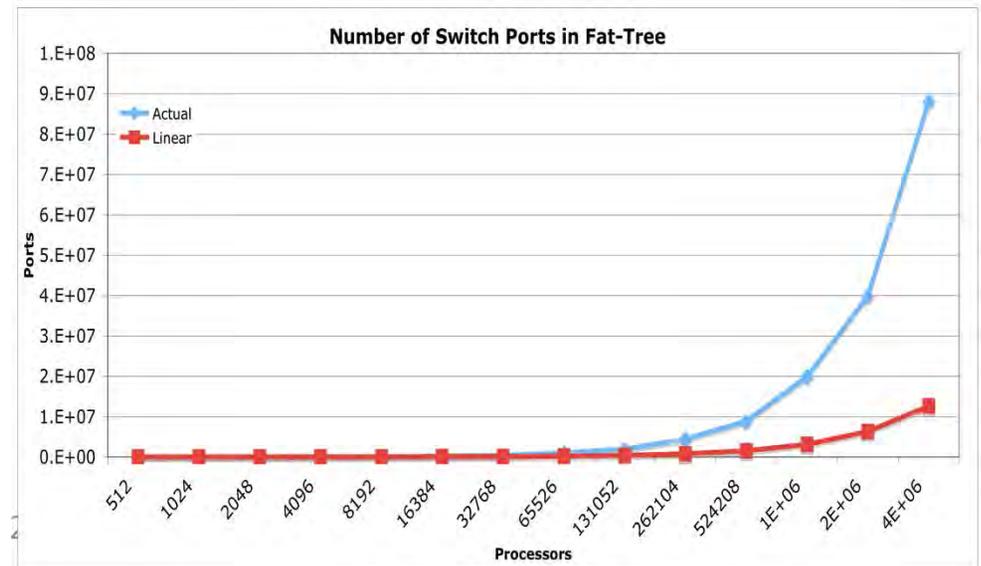
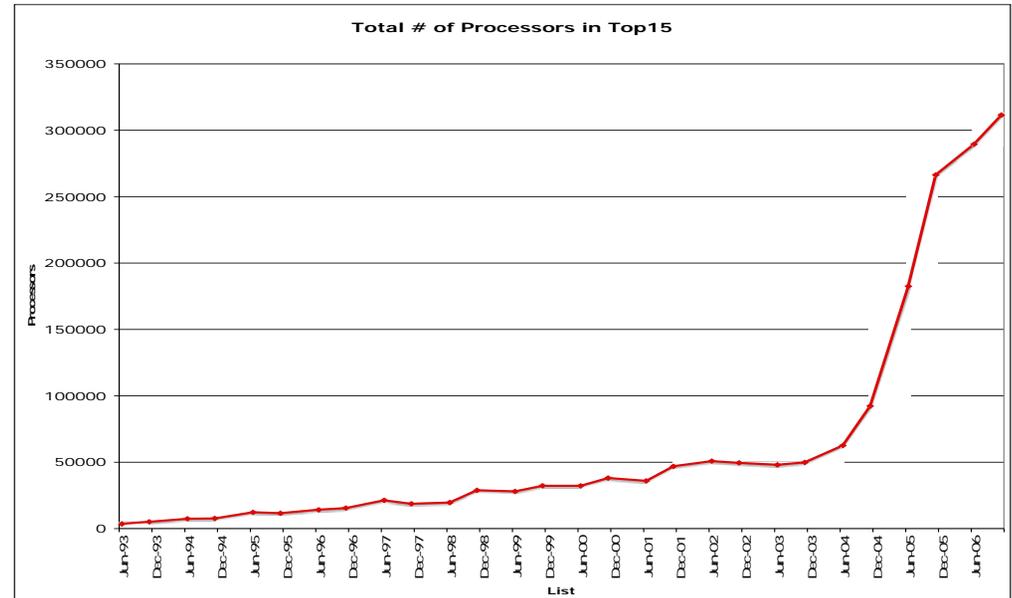
Technology Trends and Effects on
Application Performance



Lawrence Berkeley
National Laboratory

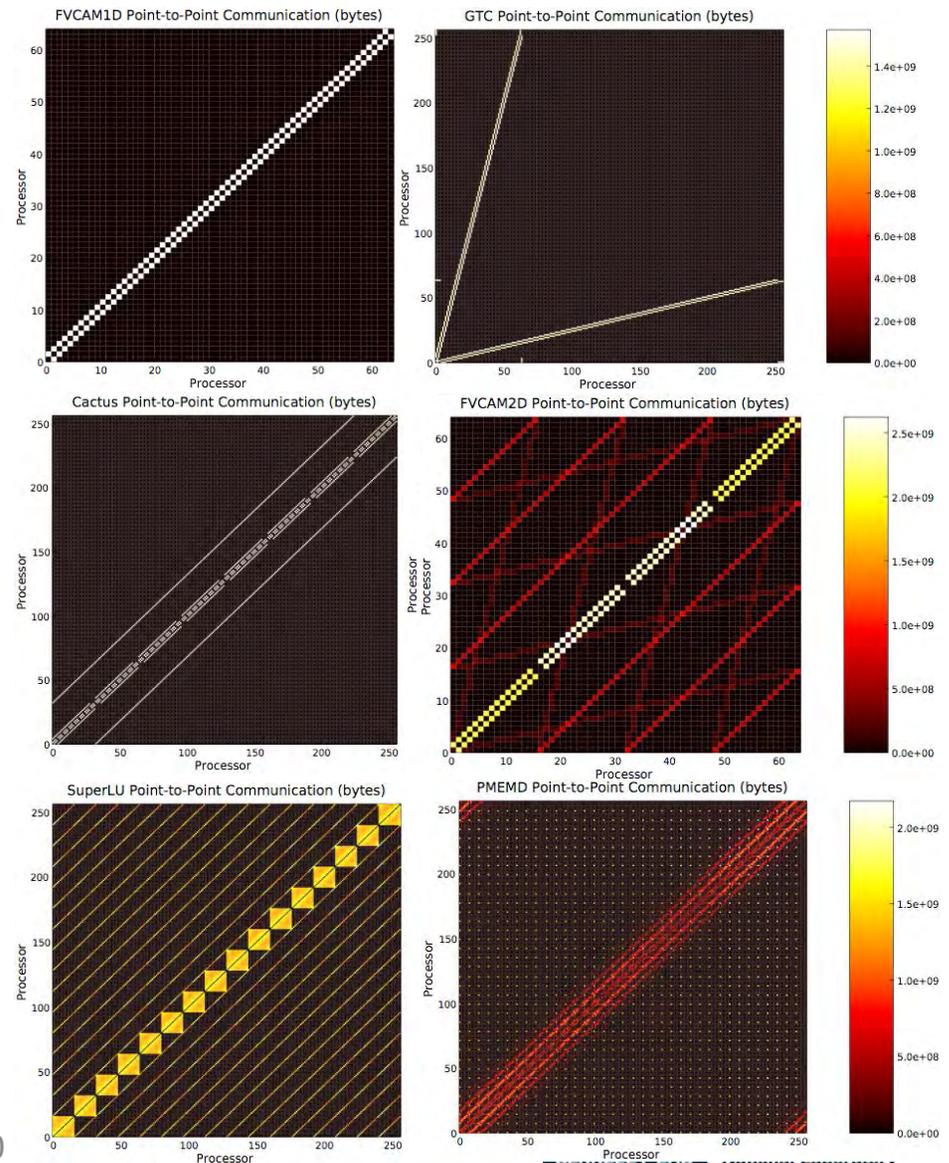
Scalable Interconnects

- Can't afford to continue with Fat-trees or other Fully-Connected Networks (FCNs)
- But will Ultrascale applications perform well on lower degree networks like meshes, hypercubes or torii. Or high-radix routers/tapered dragonfly?
- How can we wire up a custom interconnect topology for each application?



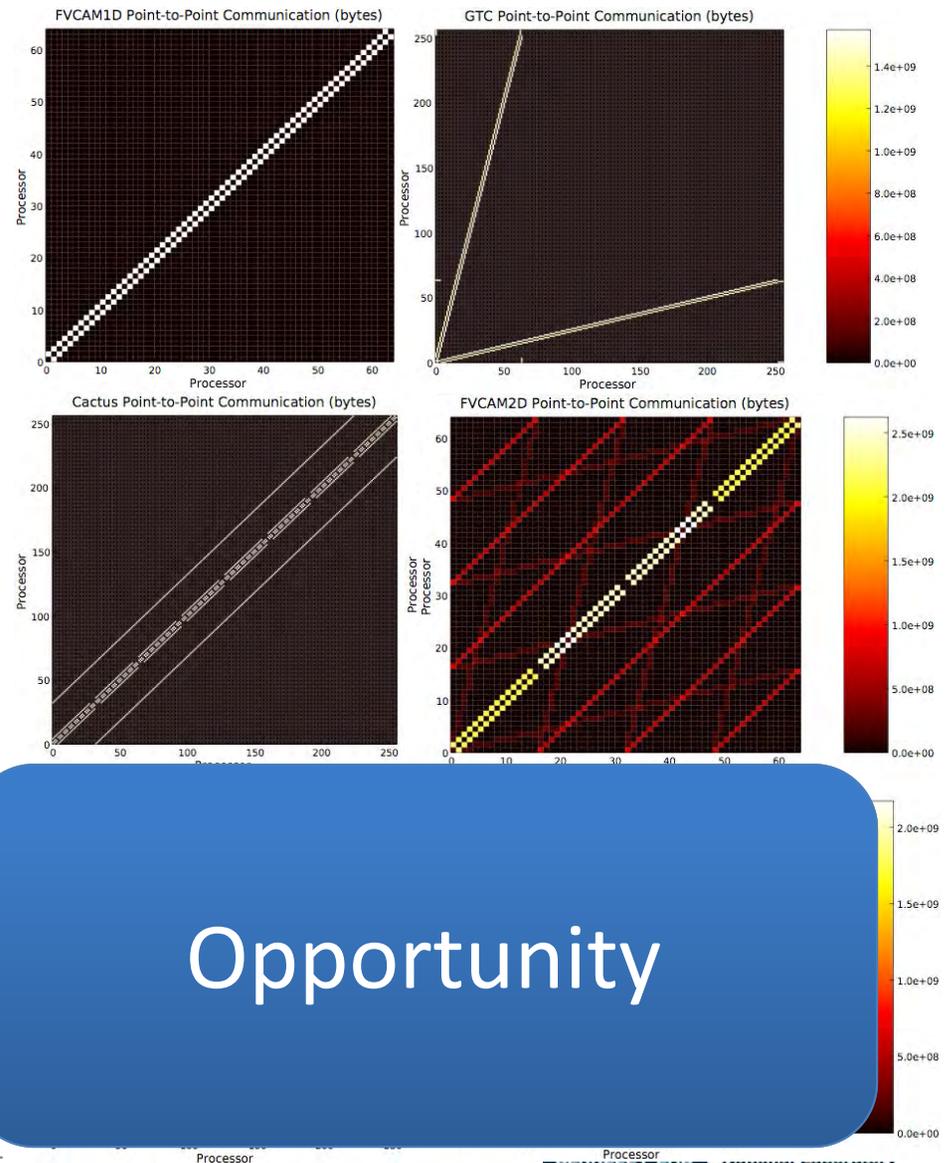
Interconnect Design Considerations for Message Passing Applications

- **Application studies provide insight to requirements for Interconnects (both on-chip and off-chip)**
 - On-chip interconnect is 2D planar (crossbar won't scale!)
 - Sparse connectivity for most apps.; crossbar is overkill
 - No single best topology
 - Most point-to-point message exhibit sparse topology + often bandwidth bound
 - Collectives tiny and primarily latency bound
- **Ultimately, need to be aware of the on-chip interconnect topology in addition to the off-chip topology**
 - Adaptive topology interconnects (HFAST)
 - Intelligent task migration?



Interconnect Design Considerations for Message Passing Applications

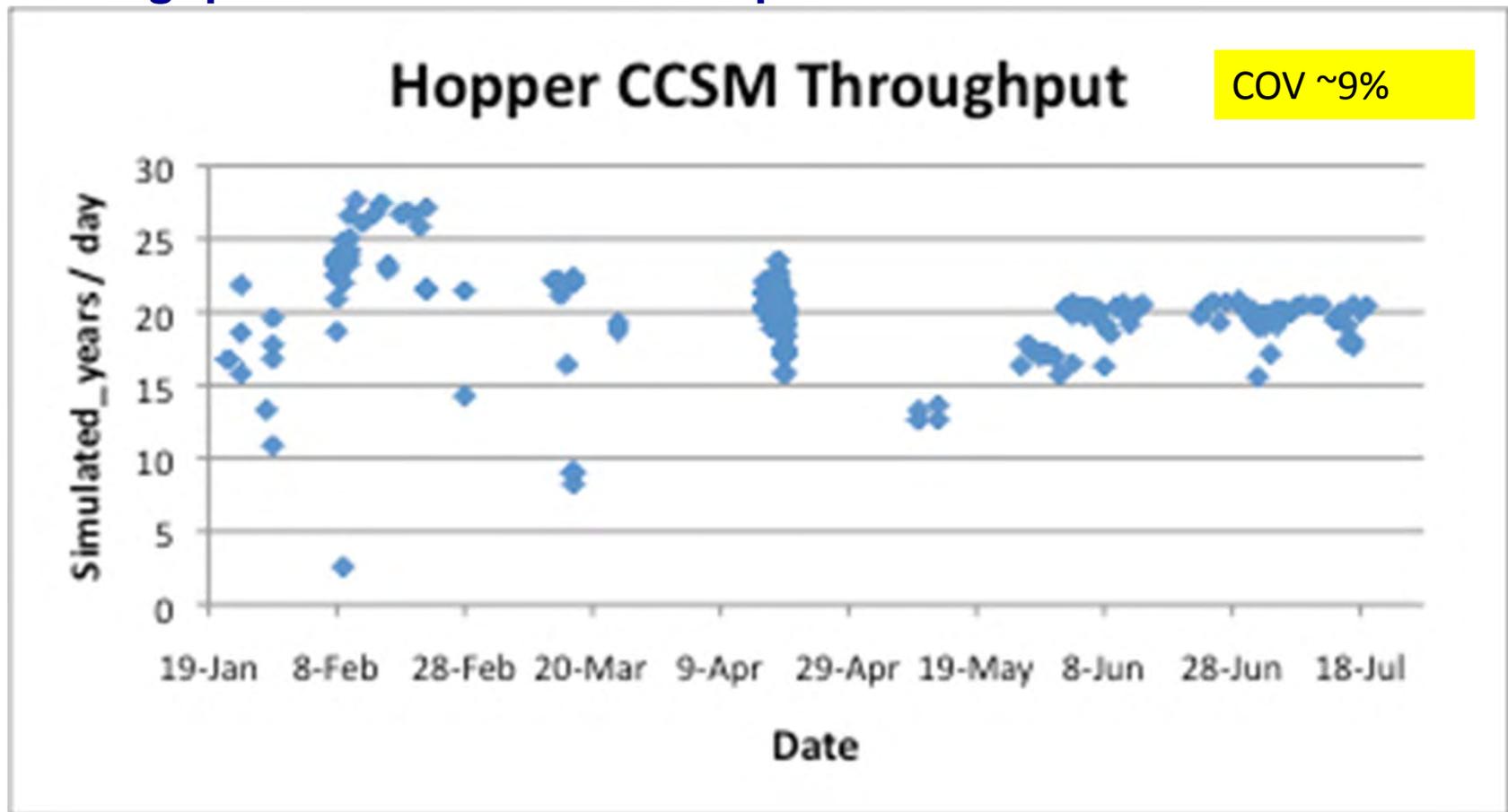
- **Application studies provide insight to requirements for Interconnects (both on-chip and off-chip)**
 - On-chip interconnect is 2D planar (crossbar won't scale!)
 - Sparse connectivity for most apps.; crossbar is overkill
 - No single best topology
 - Most point-to-point message exhibit sparse topology + often bandwidth bound
 - Collectives tiny and primarily latency bound
- **Ultimately, need to be aware of the on chip interconnect topology in addition to the off-chip topology**
 - Adaptive topology interconnects (HFAST)
 - Intelligent task migration?



CCSM Performance Variability

(trials of embedding communication topologies)

- Result of 311 runs of the coupled climate model showing model throughput as a function of completion date.



U.S. DEPARTMENT OF
ENERGY

Office of
Science

Data from Harvey Wasserman

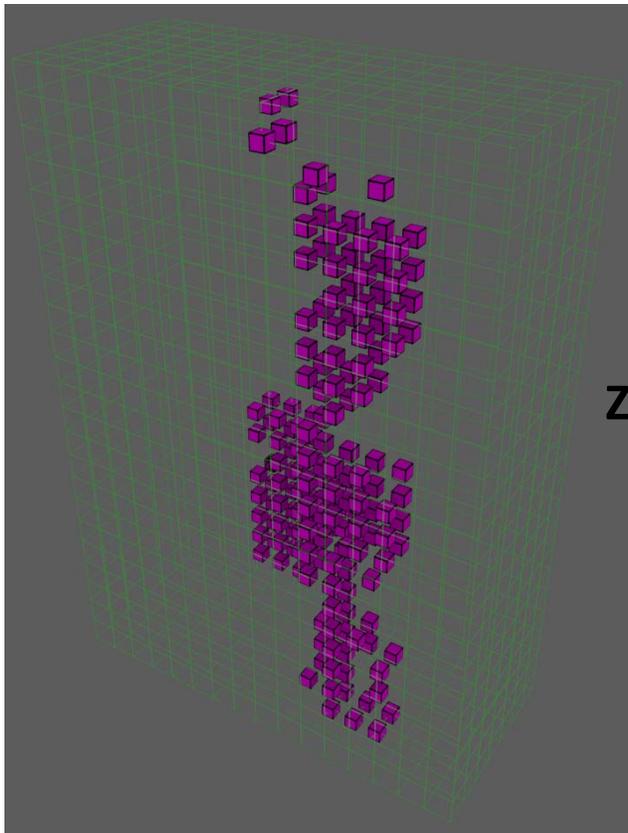
32



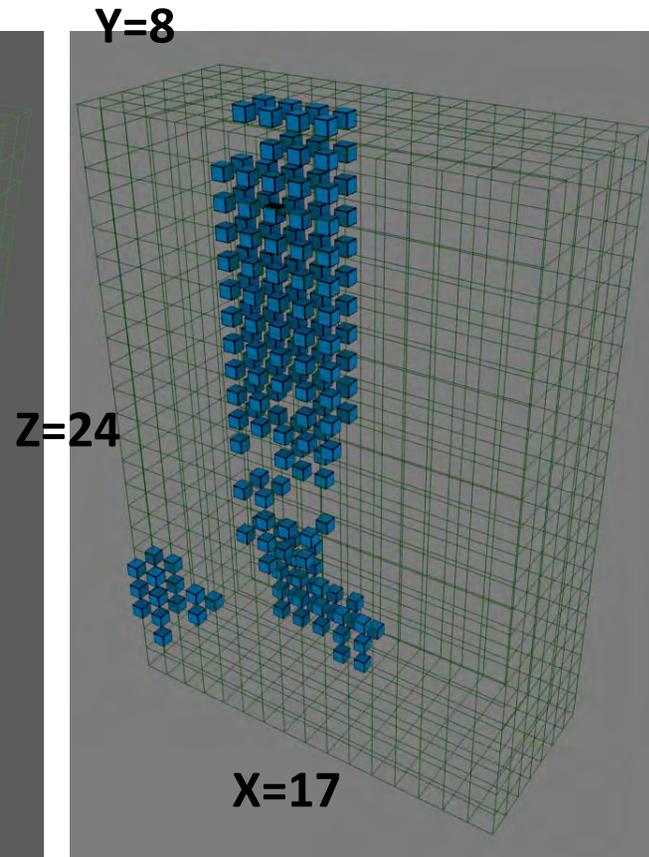
Lawrence Berkeley
National Laboratory

Node placement of a fast, average and slow run

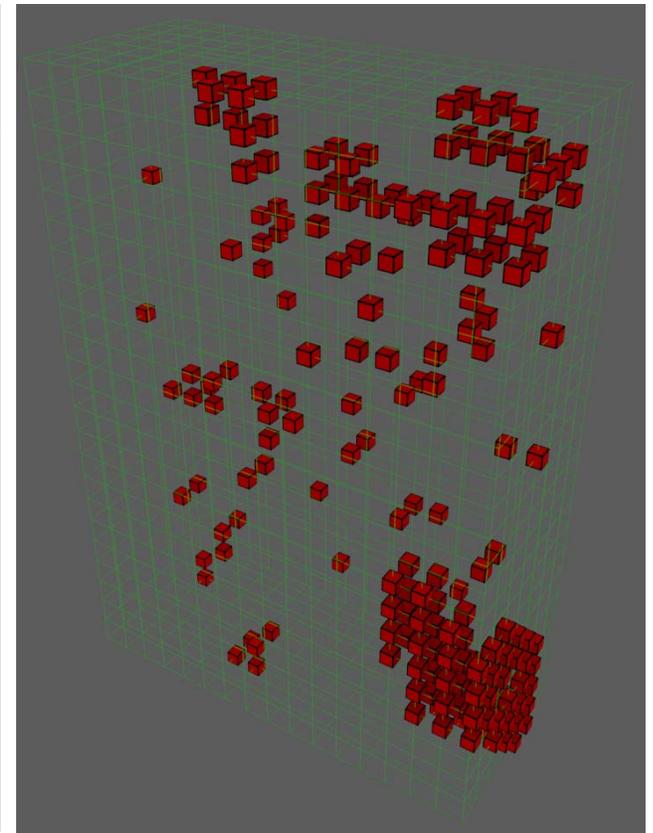
from Katie Antypas



Fast run: 940 seconds



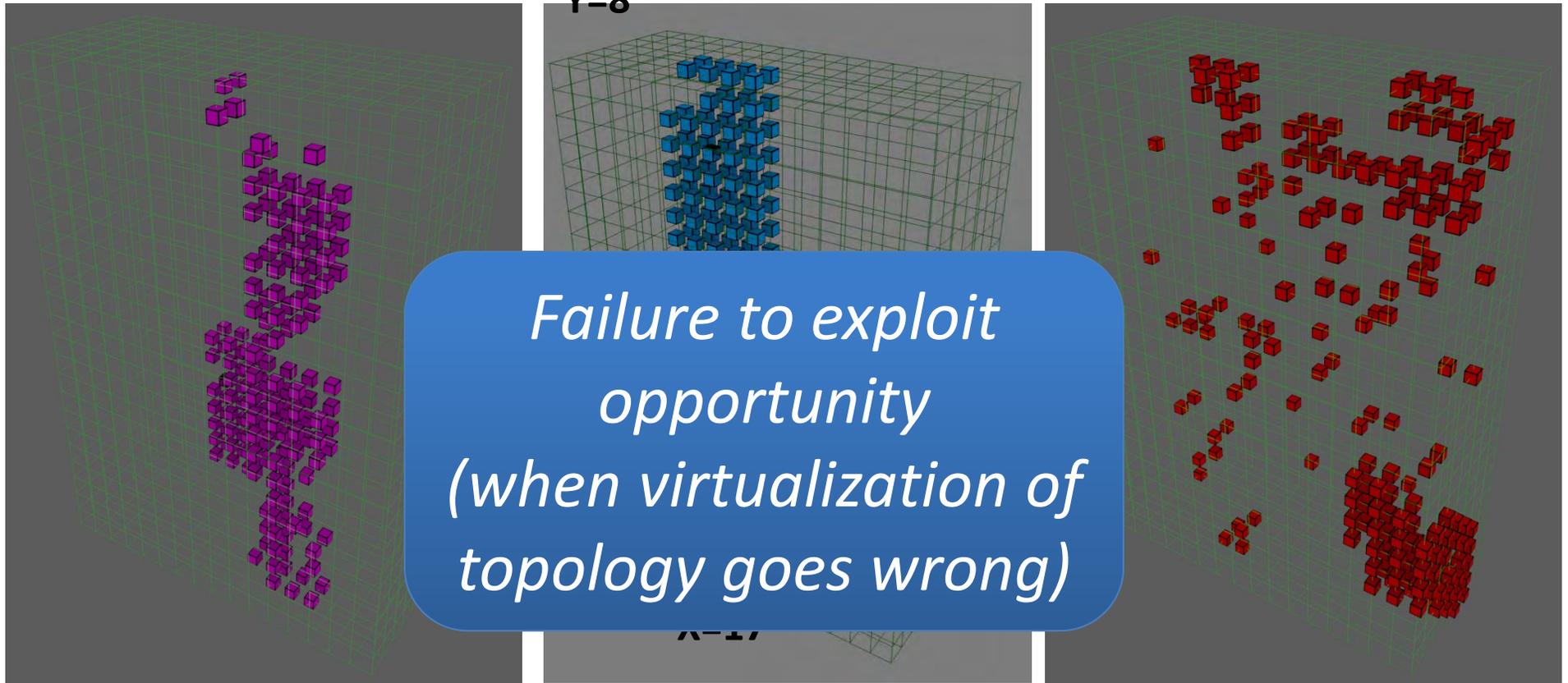
Average run: 1100 seconds



Slow run: 2462 seconds

Node placement of a fast, average and slow run

from Katie Antypas



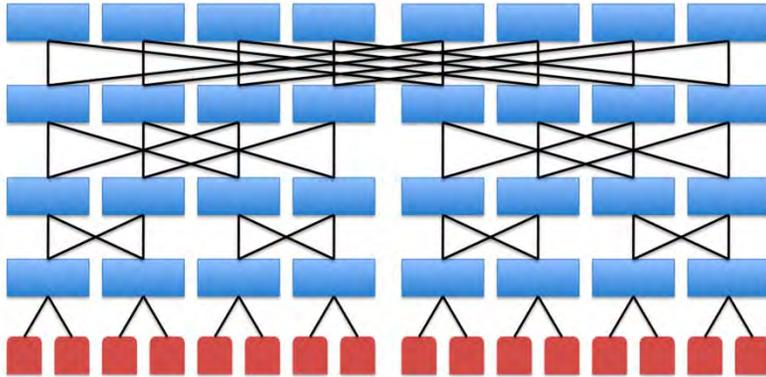
Fast run: 940 seconds

Average run: 1100 seconds

Slow run: 2462 seconds

Topology Optimization

(turning *Fat-trees* into *Fit-trees*)



A 2-ary 4-tree with 16 nodes.

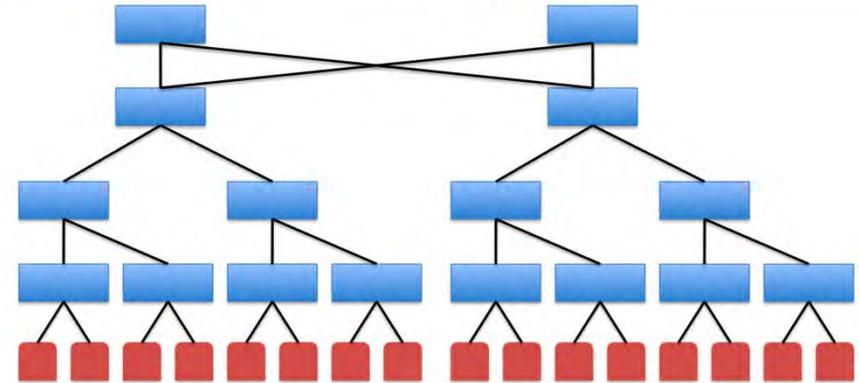
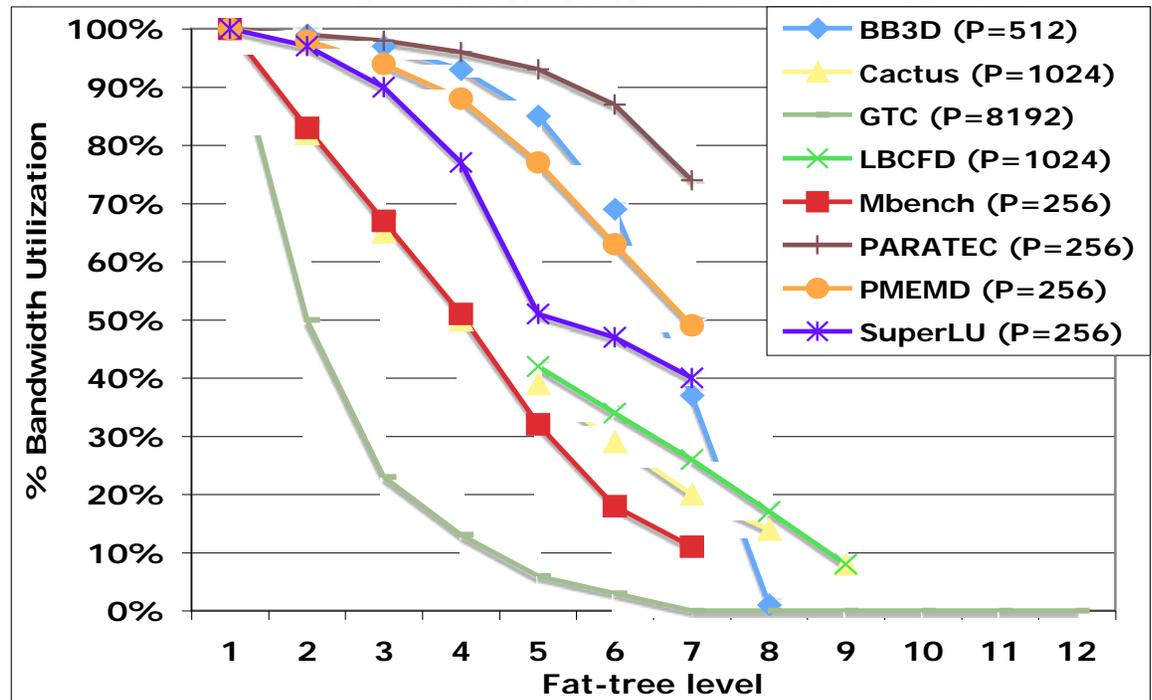


Figure 2: A (2, 2, 4)-TL fit-tree with 16 nodes.

- A Fit-tree uses OCS to prune unused (or infrequently used) connections in a Fat-Tree
- Tailor the interconnect bandwidth taper to match application data flows



Conclusions on Interconnect

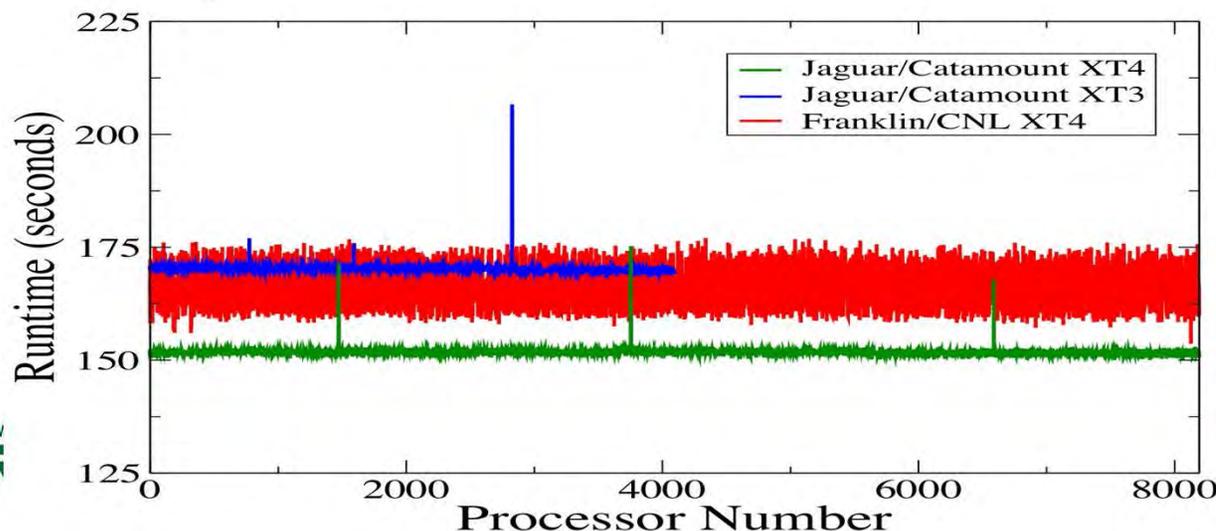
- **Huge opportunity for communication topology optimization to improve performance**
 - Runtime information gathering for active task migration, circuit switching
 - Use intelligent runtime to remap for locality or to use circuit switching to optimize switch topology
- **Current Programming Models do not provide facility to express topology**
 - OpenMP topology un-aware
 - MPI has topology directives (tedious, and rarely implemented or used)
 - **Results in substantial measurable losses in performance (*within node/OpenMP and inter-node/MPI*)**

Need to provide the compiler, runtime & resource manager more information about topology

Heterogeneity / Inhomogeneity async pmodels?

Assumptions of Uniformity is Breaking (many new sources of heterogeneity)

- Heterogeneous compute engines (hybrid/GPU computing)
- Irregular algorithms
- Fine grained power mgmt. makes homogeneous cores look heterogeneous
 - thermal throttling on Sandybridge – no longer guarantee deterministic clock rate
- Nonuniformities in process technology creates non-uniform operating characteristics for cores on a CMP
- Fault resilience introduces inhomogeneity in execution rates
 - error correction is not instantaneous
 - And this will get WAY worse if we move towards software-based resilience



Conclusions on Heterogeneity

- **Sources of performance heterogeneity increasing (especially as we try to extract more energy efficiency)**
 - Heterogeneous architectures (accelerator)
 - Thermal throttling
 - Near Threshold: increased heterogeneity for clock rates
 - Performance heterogeneity due to transient error recovery
- **Current Bulk Synchronous Model not up to task**
 - Current focus is on removing sources of performance variation (jitter), is increasingly impractical
 - Huge costs in power/complexity/performance to extend the life of a purely bulk synchronous model

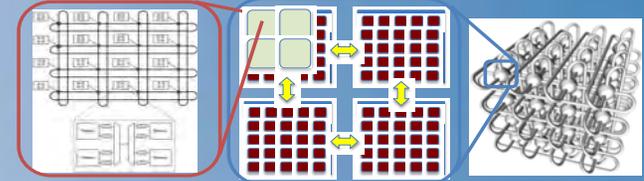
Embrace performance heterogeneity: Study use of asynchronous computational models (e.g. SWARM, HPX, and other concepts from 1980s)

Why Wait for Exascale everything is breaking NOW!

Conclusions

- **Emerging hardware constraints are increasingly mismatched with our current programming paradigm**
 - Current emphasis is on preserving FLOPs
 - The real costs now are not FLOPs... it is data movement
 - Requires shift to a data-locality centric programming paradigm and hardware features to support it
- **Codesign is *NOT* just design optimization**
 - The programming environment and associated “abstract machine model” is a reflection of the underlying machine architecture
 - Therefore, design decisions can have deep effect your entire programming paradigm
 - Hardware/Software Codesign **MUST** consider ergonomic decisions about your programming environment together with performance
- **Performance Portability Should be Top-Tier Metric for CoDesign process**
 - Know what to **IGNORE**, what to **ABSTRACT**, and what to make more **EXPRESSIVE**

Remember the Abstract Machine Model



Programming model IS, and SHOULD BE a proper reflection of the underlying machine architecture

Machine attributes are parameterized

- Changes with each generation of machine and between different vendor implementations
- Pmodel should target the parameterized attributes

For each parameterized machine attribute

- **Ignore it:** If ignoring it has no serious power/performance consequences
- **Abstract it (*virtualize*):** If it is well enough understood to support an automated mechanism to optimize layout or schedule
- **Expose it (*unvirtualize*):** If there is not a clear automated way of make decisions

Recommendations

- **Data layout (currently, make it more expressive)**
 - Need to support hierarchical data layout that closer matches architecture
 - Automated method to select optimal layout is elusive, but type-system can support minimally invasive user selection of layout
- **Horizontal locality management (virtualize)**
 - Flexibly use message queues and global address space
 - Give intelligent runtime tools to dynamically compute cost of data movement
- **Vertical data locality management (make more expressive)**
 - Need good abstraction for software managed memory
 - Malleable memories (allow us to sit on fence while awaiting good abstraction)
- **Heterogeneity (virtualize)**
 - Its going to be there whether you want it or not
 - Pushes us towards async model for computation (post-SPMD)
- **Parallelism (virtualize)**
 - Need abstraction to virtualize # processors (but must be cognizant of layout)
 - For synchronous model (or sections of code) locality-aware iterators or loops enable implicit binding of work to local data.
 - For async codes, need to go to functional model to get implicit parallelism
 - Helps with scheduling
 - Does not solve data layout problem