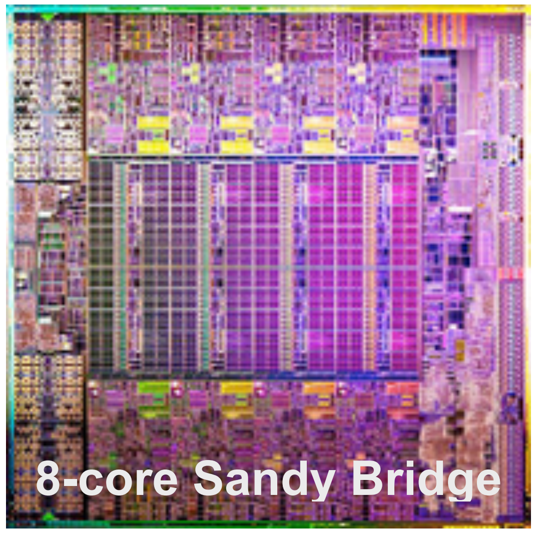
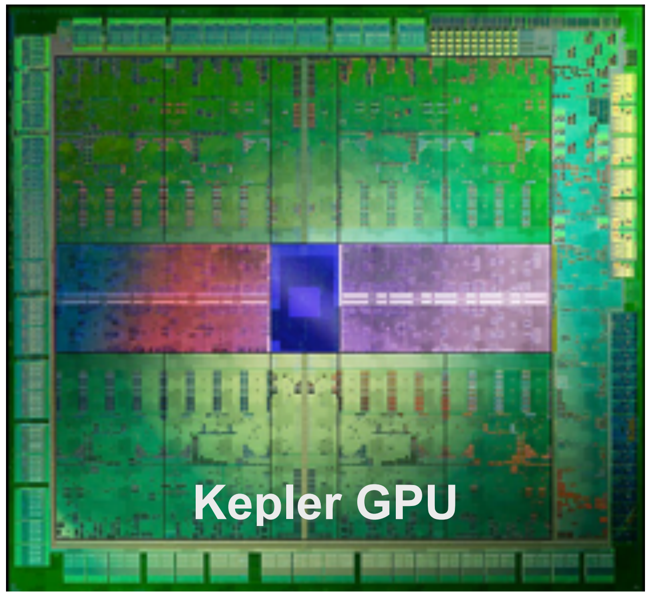
Next-Generation Computing Architectures: Friend or Foe?

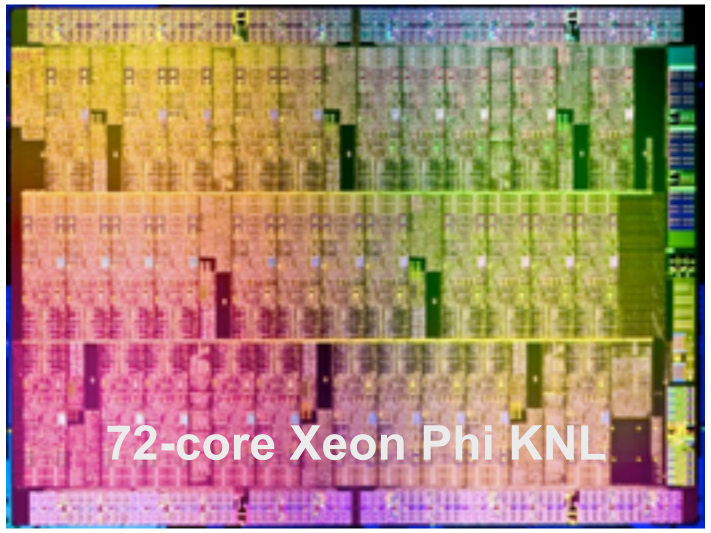
Salman Habib
HEP and MCS Divisions
Argonne National Laboratory

Computation Institute
Argonne National Laboratory
University of Chicago

Kavli Institute for Cosmological Physics
University of Chicago







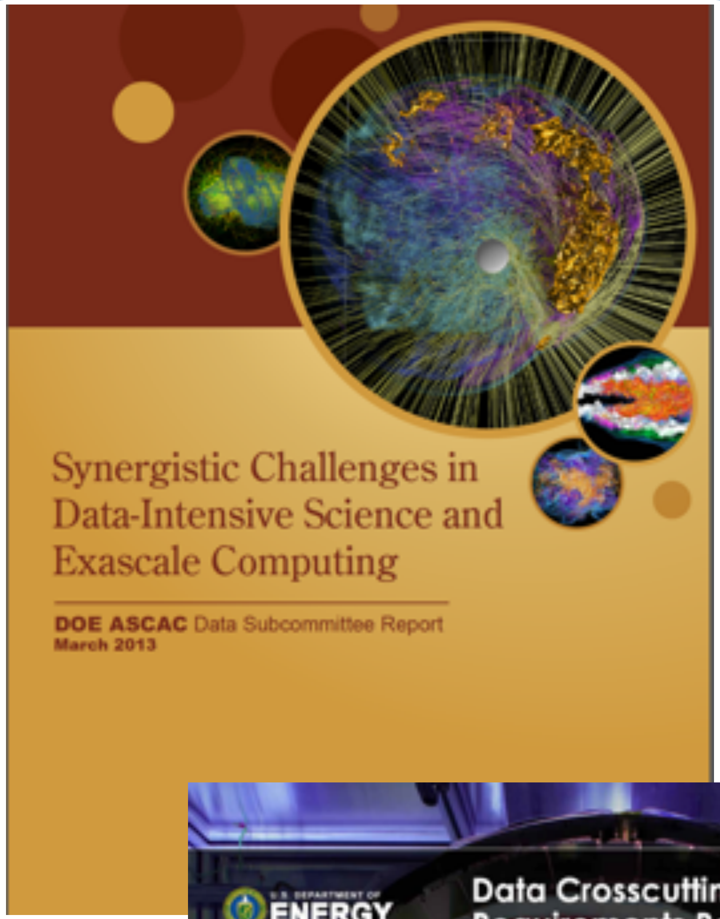
LSST-DESC and Computing

- **LSST-DESC focus is on Level II/III analysis**
  - Estimates of initial computing needs are unclear, ranging from 150-350 TFlops/year
  - Initial storage needs are ~PB, growing linearly
  - Based on this, we would want (at least) the #1 machine in the Top 500 in 2006
  - In 2022 there may be O(1000-10,000) such machines in the US alone!
  - Storage requirement is already 'trivial', LSST is NOT 'Big Data'
- **So what's the problem?**
  - Analyses will be complex (and there will be many)
  - These tasks will expand to fill available computational space
  - Programming models may be very different from those in use today

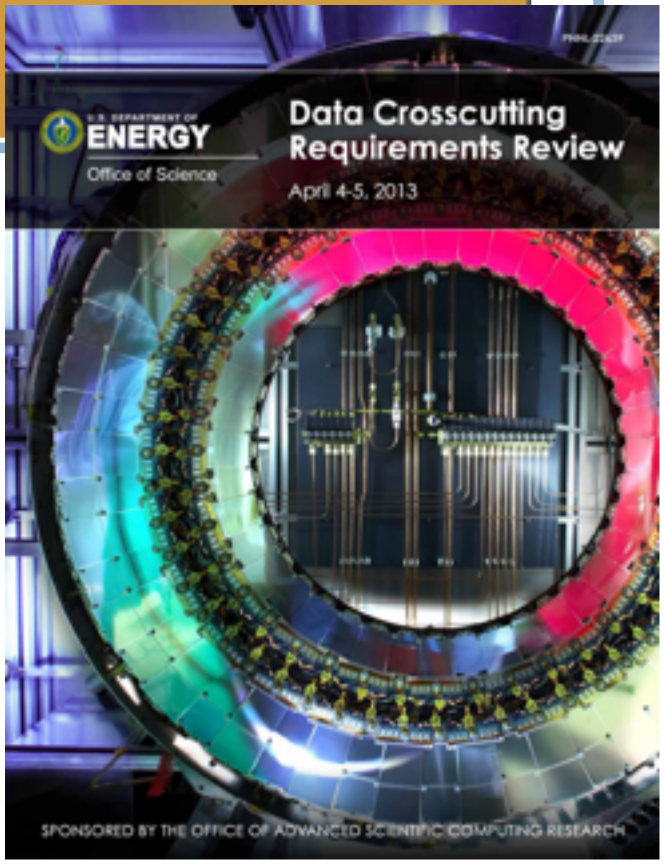
300 TFlops/10PB, 10kW in 2020 (Projection)

Different Flavors of Computing

- **High Performance Computing ('PDEs')**
  - Parallel systems with a fast network
  - Designed to run tightly coupled jobs
  - High performance parallel file system
  - Batch processing
- **Data-Intensive Computing ('Interactive Analytics')**
  - Parallel systems with balanced I/O
  - Designed for data analytics
  - System level storage model
  - Interactive processing
- **High Throughput Computing ('Events'/'Workflows')**
  - Distributed systems with 'slow' networks
  - Designed to run loosely coupled jobs
  - System level/Distributed data model
  - Batch processing


Synergistic Challenges in Data-Intensive Science and Exascale Computing

DOE ASCAC Data Subcommittee Report
March 2013


PNNL-22639

U.S. DEPARTMENT OF ENERGY
Office of Science

Data Crosscutting Requirements Review
April 4-5, 2013

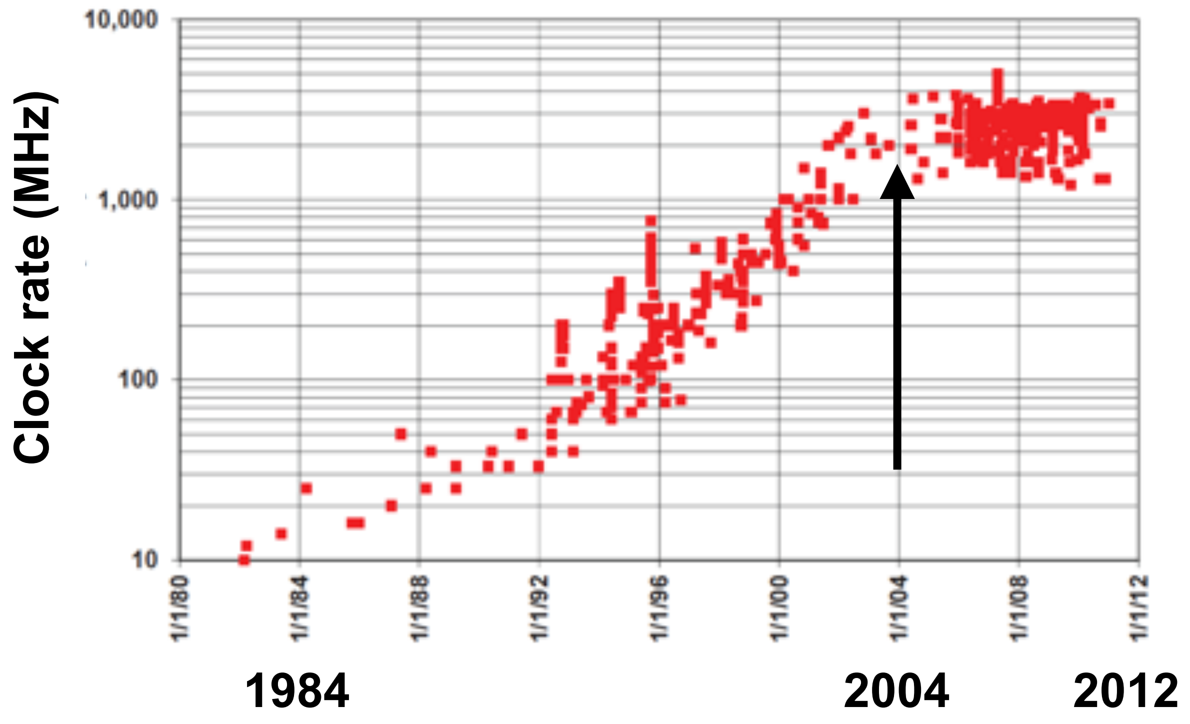
SPONSORED BY THE OFFICE OF ADVANCED SCIENTIFIC COMPUTING RESEARCH

Interactive Exercise I

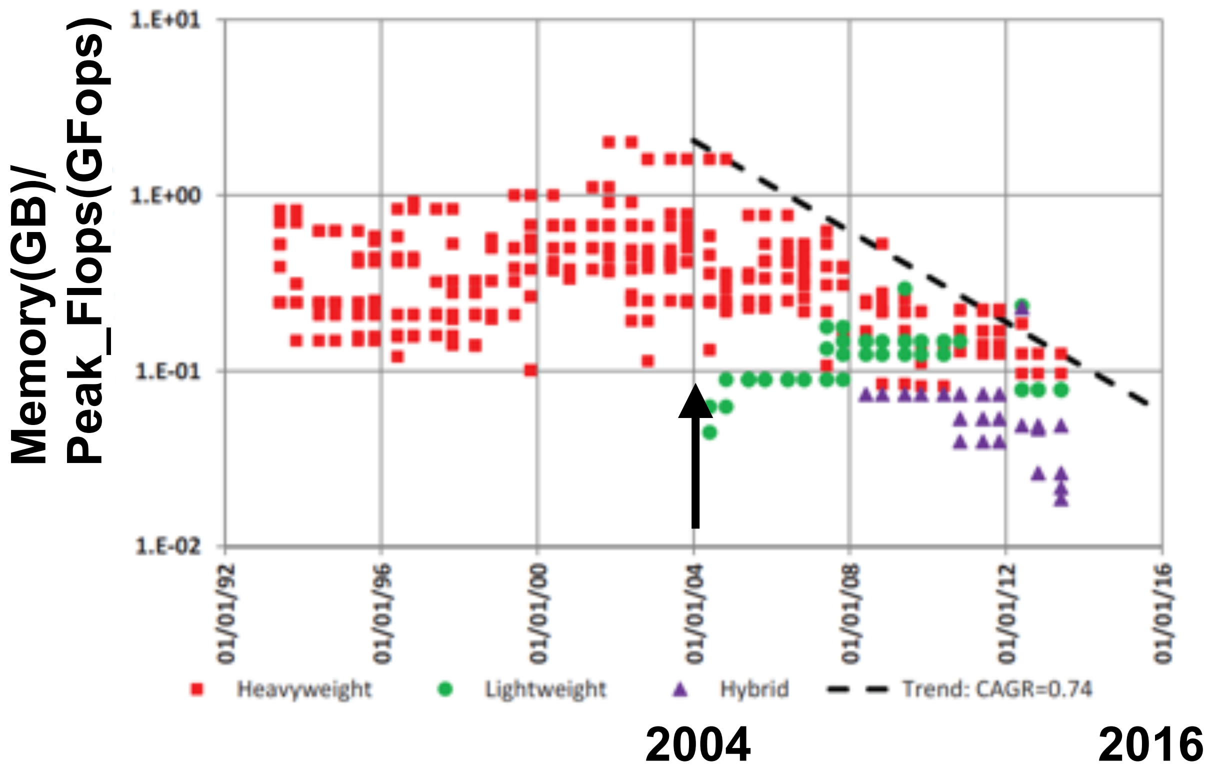
- **Do you care about computational architectures (what's in the processor/system) in your current work?**
  - If so, why?
  - If not, why not?
- **Types of computing**
  - Do you exploit parallelism in your codes?
  - If so, are your use cases HPC, DISC, or HTC?
  - If you don't exploit parallelism, why not?
  - I don't have any codes
- **Programming expertise**
  - Do you consider yourself to be primarily a code 'consumer', i.e., you primarily stitch together programs/workflows based on functional units written by others?
  - Or do you consider yourself to be an application developer, i.e., you actually design/understand/implement the key algorithms?
  - None of the above (if so, explain what you do)

The Problem: Hardware Evolution

- **Power is the main constraint**
  - Target: 30X performance gain by 2020
  - ~10-20MW per large system
  - Power/Socket roughly const.
- **Only way out: more cores**
  - Several design choices (e.g., cache vs. compute vs. interconnect)
  - All lead to more complexity
- **Micro-architecture gains sacrificed**
  - Accelerate specific tasks
  - Restrict memory access structure (SIMD/SIMT)
- **Machine balance sacrifice**
  - Memory/Flops; comm BW/Flops — all go in the "wrong" direction

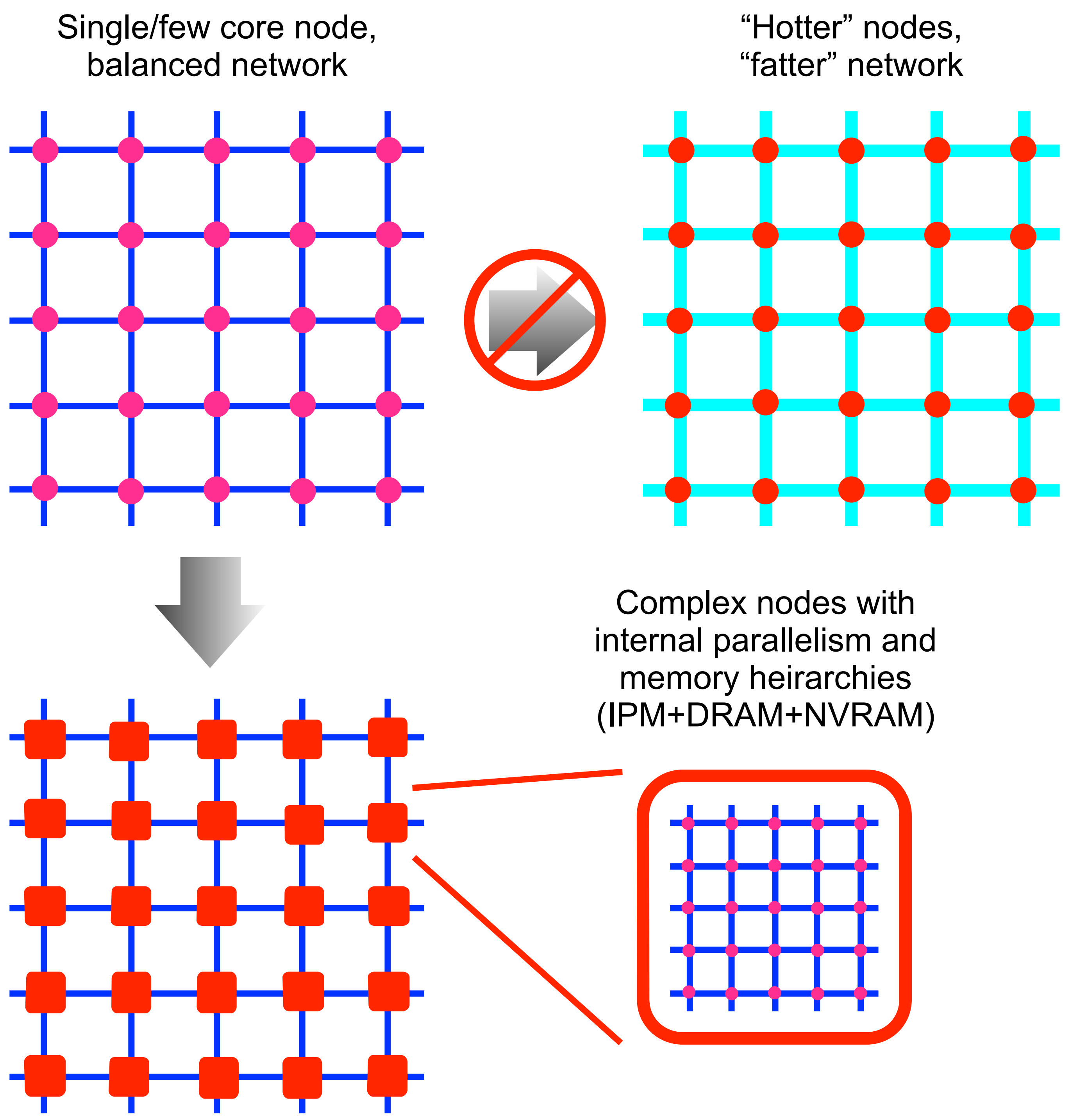


Kogge and Resnick (2013)



Hardware Evolution: Implications

- **Programming view (current)**
  - Processor properties hidden
  - Simple control flow
  - Single memory image
  - (Mostly) portable code
- **How to handle complex nodes?**
  - Magic compiler unlikely, problem is too complex
  - “Problem + Task Decompostion + Hardware” — single problem
  - Portability is a serious issue
- **Possible approaches**
  - Rely on someone else (e.g. NumPy)
  - Bite the bullet

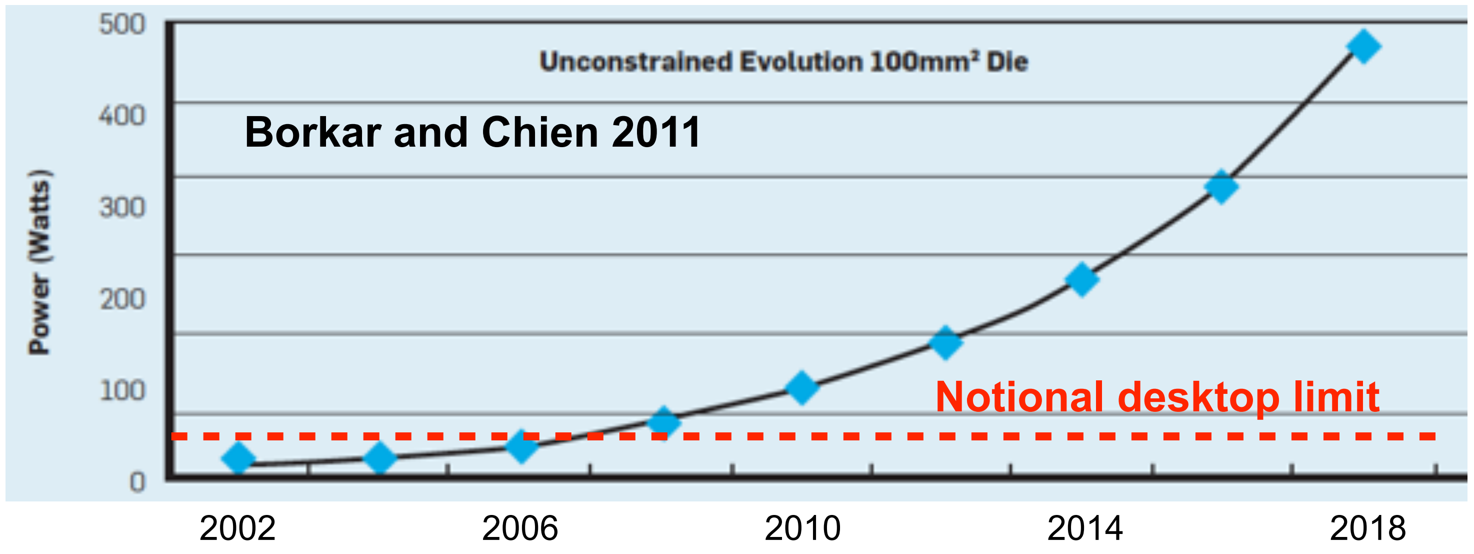


Single/few core node, balanced network
“Hotter” nodes, “fatter” network
Complex nodes with internal parallelism and memory heirarchies (IPM+DRAM+NVRAM)

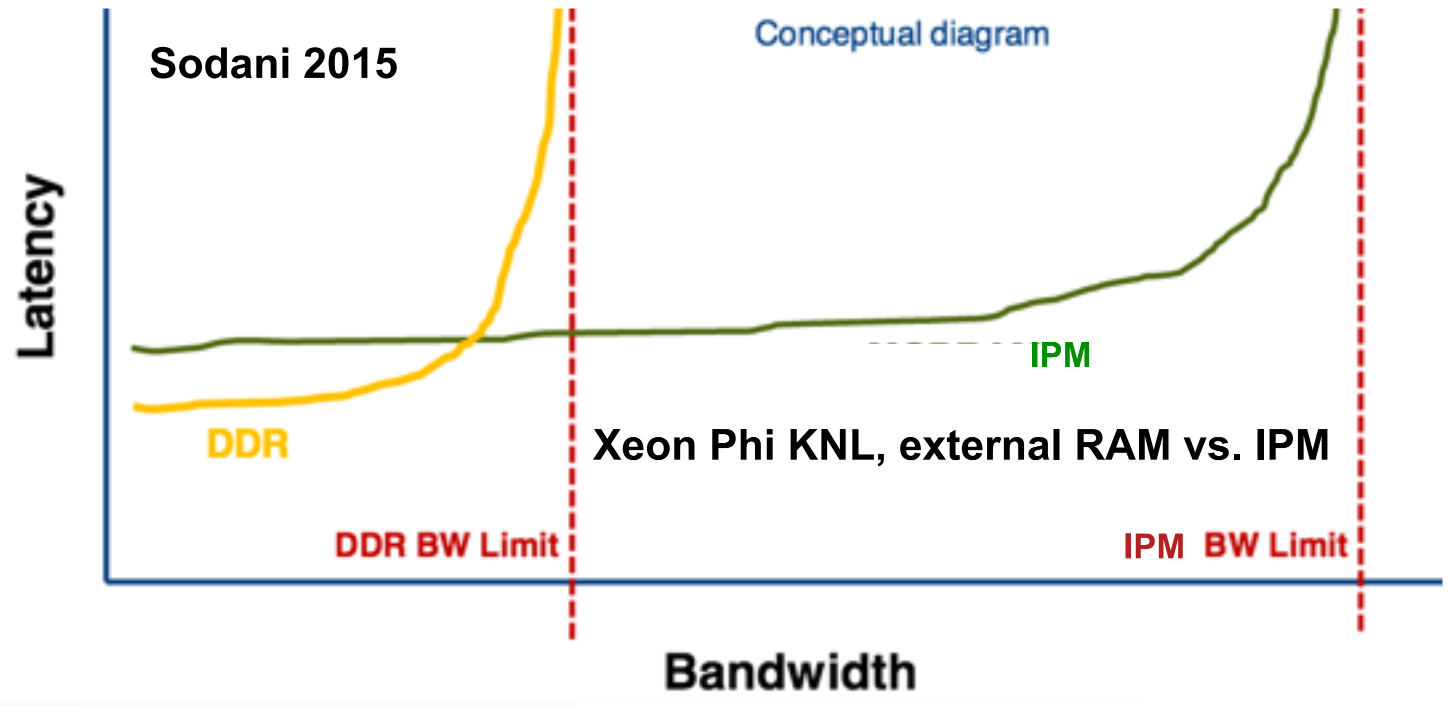
Evolution of HPC systems

Processor Complexity

- **Example: Xeon Phi KNL**
  - 72 out-of-order cores
  - 512 bit vector registers (X2)
  - 4 threads/core
  - 16GB IPM at 400GB/s
  - 3+ TFlops in DP
  - Multiple memory modes (flat, cache, hybrid)
  - In principle, what runs on a standard Xeon will run on KNL, but extracting performance will require work
- **CPU/GPU**
  - Also possible but not covered here, likely to have only a niche application in the current context

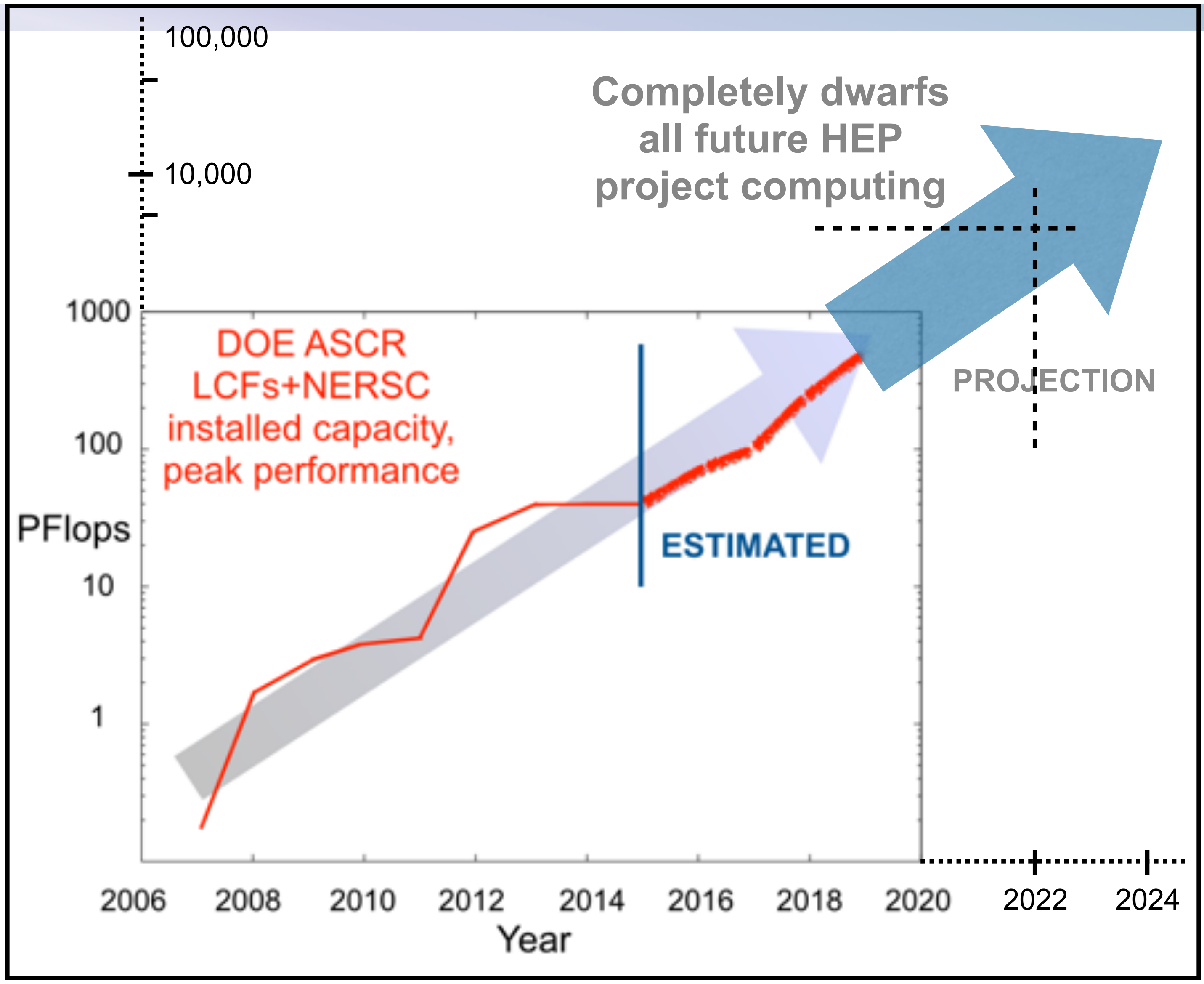


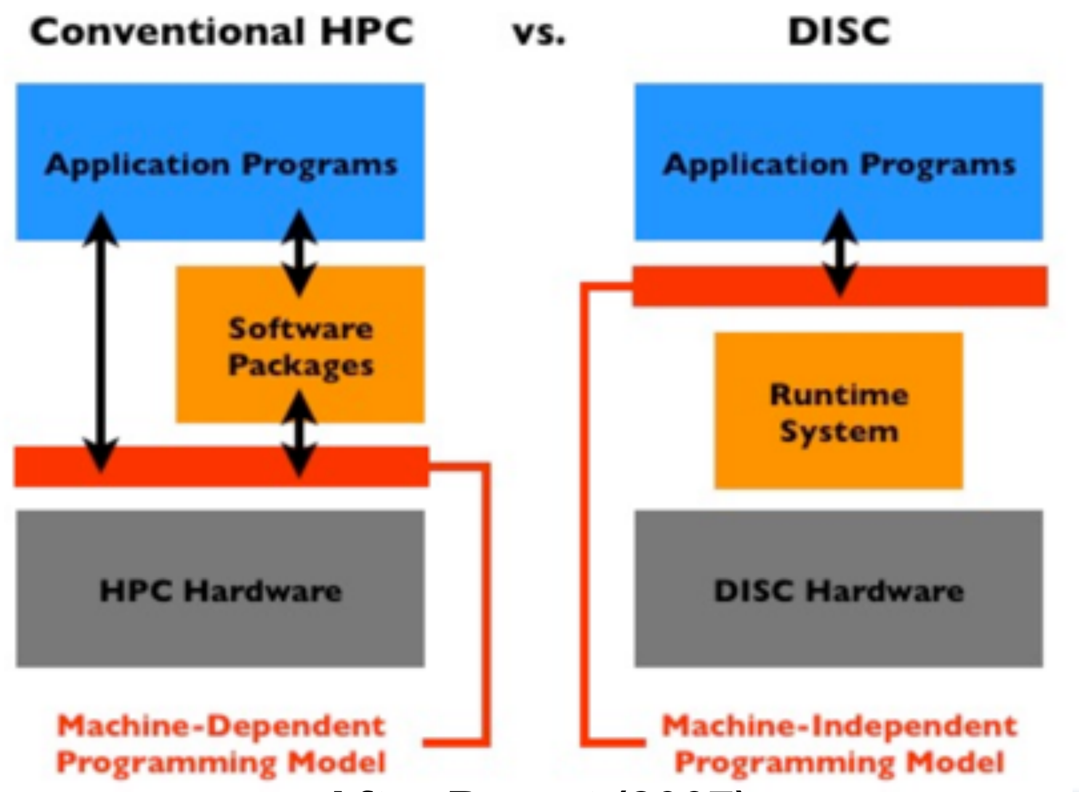
Performance requires many cores but each core cannot run at high clock frequencies (power limit), therefore performance requires vectorization; keeping the cores fed requires a fast on-chip communication fabric; to avoid the "memory wall", need fast on-chip storage (IPM) — to maximize performance code must be able to exploit parallelism (288-way) and be highly vectorized



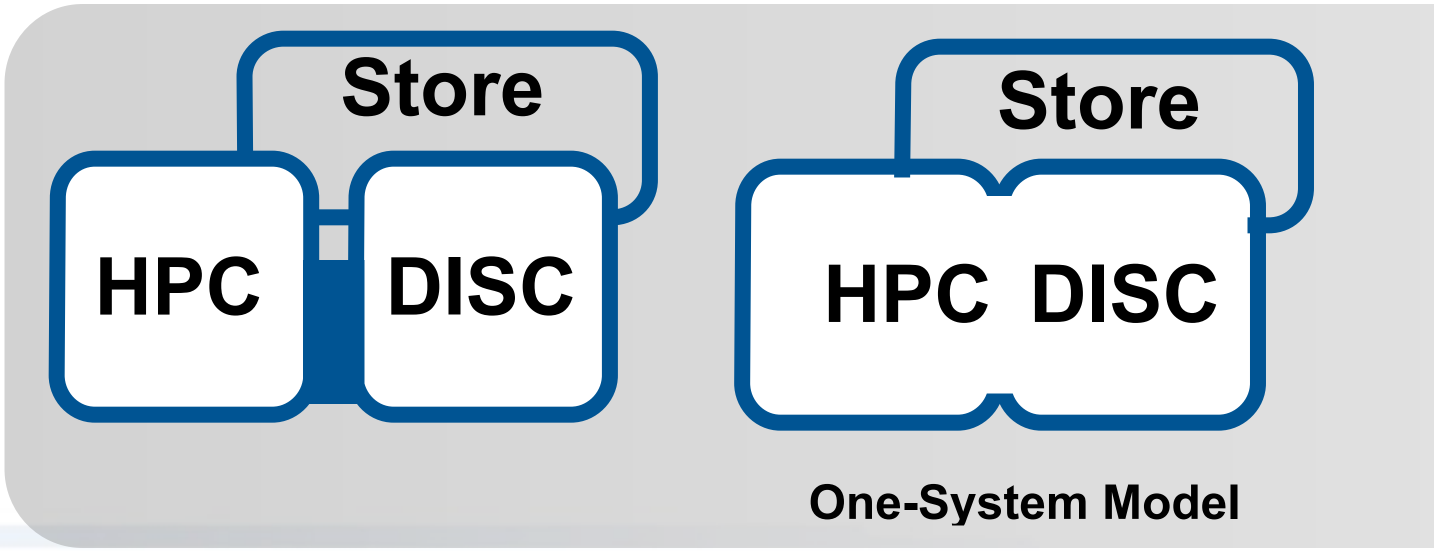
Hardware Co-Evolution: HPC and DISC

- **HPC systems on "Two Swim-Lane Track" (CPU-GPU and Manycore)**
  - 30X performance gain by 2020
  - ~10-20MW per large system
  - power/socket roughly const.
- **HPC and DISC convergence**
  - Not a matter of hardware — convergence at the node level
  - Configurational flexibility





After Bryant (2007)

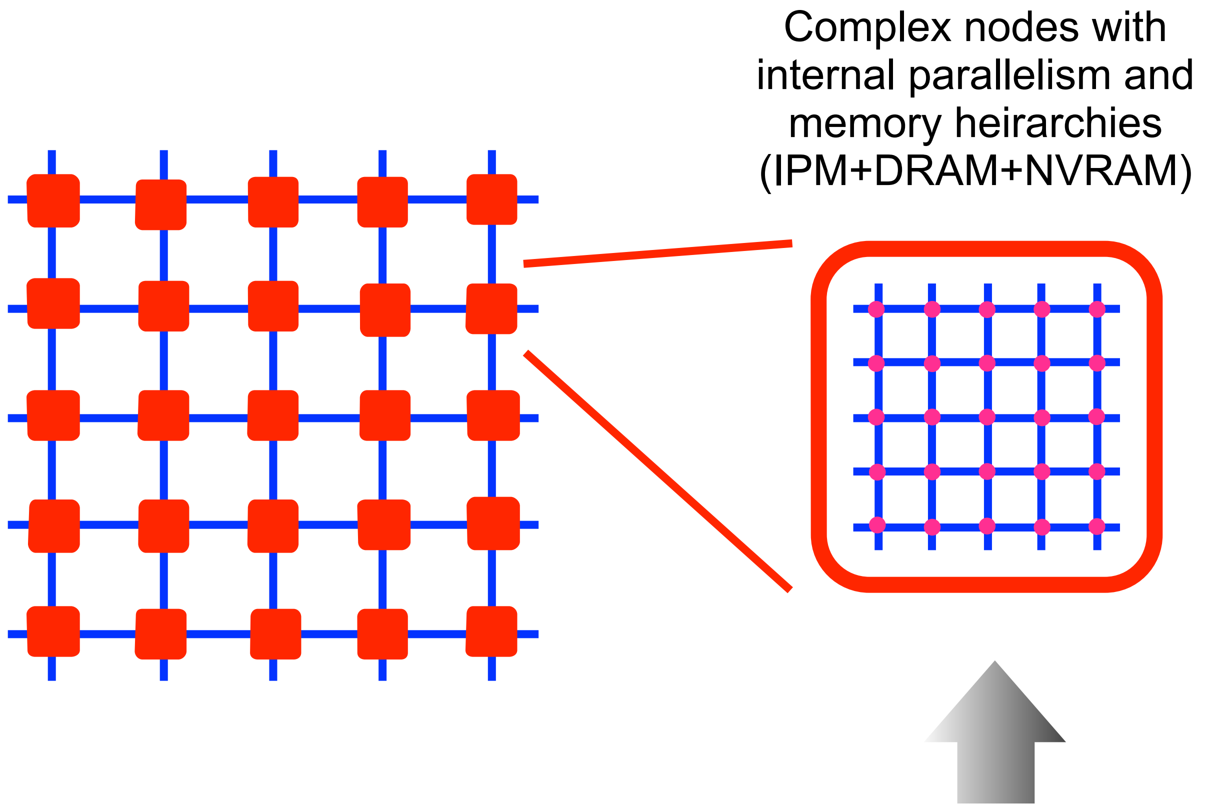


Interactive Exercise II

- **Do you NOW care about computational architectures in your future work?**
  - If so, why?
  - If not, why not?
- **Types of computing**
  - Would you exploit parallelism in your codes in the future?
  - If so, are your use cases HPC, DISC, or HTC?
  - If you won't exploit parallelism, why not?
  - I don't have any codes, so —
- **Programming expertise**
  - Do you wish to learn more about computational architectures?
  - Or have you seen enough — someone else should deal with this?

Likely Exploits

- **Most use cases likely to be DISC/HTC**
  - Note HPC systems can easily handle these in the very near future
  - Will possibly fall into two classes — **1)** many runs of a simple, not highly optimizable code, **2)** smaller, but still sizable number of runs of a potentially optimizable code
- **'Single node' application span**
  - Nodes are big enough: >100GB RAM + NVRAM
  - Main parallelism exploit will be at the node level
- **Exceptions**
  - Large-scale spatio-temporal statistics (will need system level parallelism — HPC application)


Complex nodes with internal parallelism and memory heirarchies (IPM+DRAM+NVRAM)

Focus on node-level parallelism: Quasi-independent tasks run on individual nodes; intermittent communication (if at all) across nodes

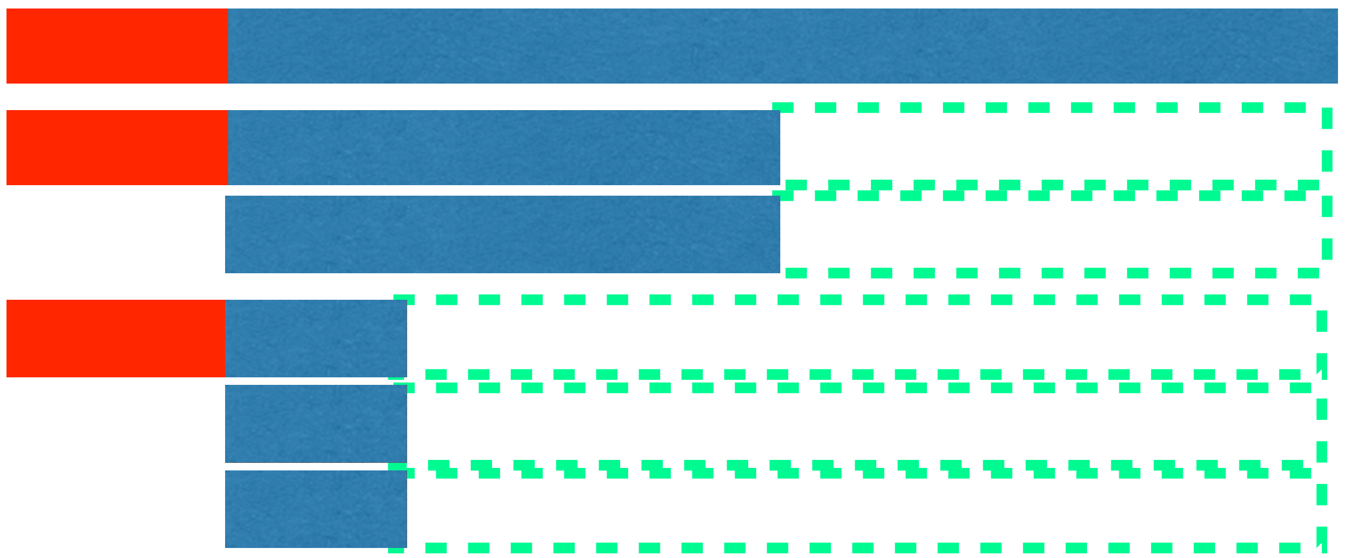
Main themes:

1. Locality, locality, locality, —

2. Threading

Serial and Parallel Workloads

$T_1 = t_{ser} + t_{par}$



$T_N \geq t_{ser} + t_{par}/N$

But Amdahl's Law is most often not directly relevant for a number of reasons:

1. **As problem size increases (possible as N increases), the amount of parallel work increases relative to the serial work and the bound can be exceeded (Gustafson's law)**
2. **In reality, there is a complex question of dependencies, overheads, communication, memory access, memory limitations, that can make life complicated (results could be worse than Amdahl's prediction)**

- **Amdahl's Law**
  - Suppose N+1 computational units are available and all serial work is done by one of them (N=1), then the speedup is bounded:

$$S = T_1/T_N \leq \frac{t_{ser} + t_{par}}{t_{ser} + t_{par}/N}$$

$$t_{ser} = fT_1$$
$$t_{par} = (1-f)T_1$$

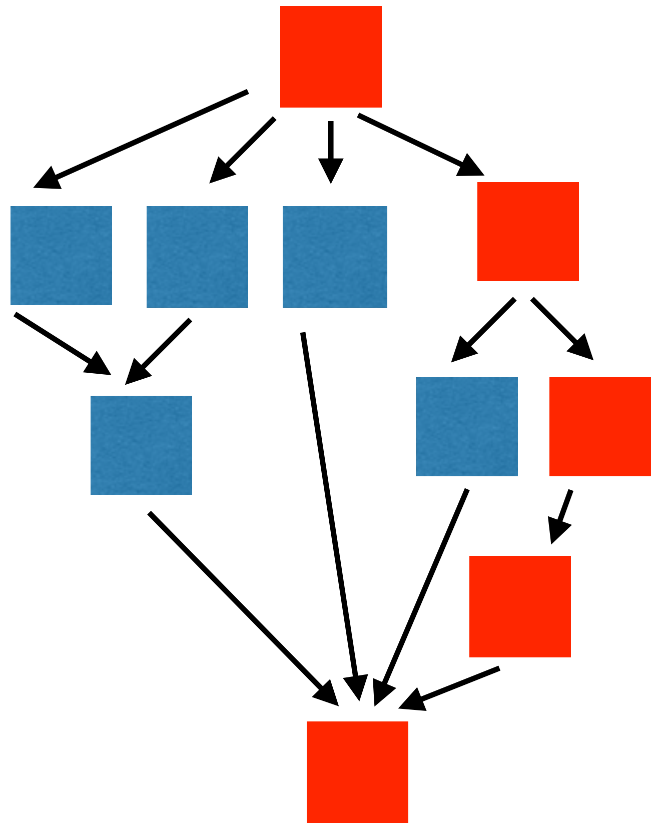
$$S \leq \frac{1}{f + (1-f)/N}$$

  - Note that even with infinite N, the speedup is finite
  - Assignment — plot S as a function of N at fixed values of f

Work-Span Model

- **Work and Span**
  - Suppose all computations can be broken up into unit work chunks, the tasks form a directed acyclic graph (DAG), and a task runs when all its precursors have been completed
  - Example — we have 10 tasks so the **work** = 10; the **span** is the time taken on an ideal parallel machine; there is a 5-path chain (critical path) here that cannot be parallelized further because of dependencies, so **span**=5
  - The speedup S is always less than work/span, in our case, 2, but usually we would like to be in a situation where work >> span
  - In this case, a good practical estimate for the run time is

$$T_N \sim \overset{\downarrow}{T_1}/N + \overset{\downarrow}{T_\infty}$$



For more details, see McCool, Robison, Reinders, Structured Parallel Programming (2012)

Communication Avoidance

- **But “compute is free”? (Locality, once again)**
  - One of the key problems in modern systems is not that we are compute bound, but that we are often communication-bound (slow networks and memory access limitations); note that the work-span model assumed communication costs were zero!
  - Model: words are packed into a contiguous block of memory, i.e., the message, which is then sent to the destination processor; the packing induces a **latency cost** and the size of the message induces a **bandwidth cost**, these have to be added to the **computational cost**

$$\overset{\text{latency cost}}{\alpha}\underset{\text{\# of messages}}{S} + \overset{\text{bandwidth cost}}{\beta}\underset{\text{total words in message}}{W} + \overset{\text{computational cost}}{\gamma}\underset{\text{arithmetic operations}}{F}$$

$$\alpha >> \beta >> \gamma$$

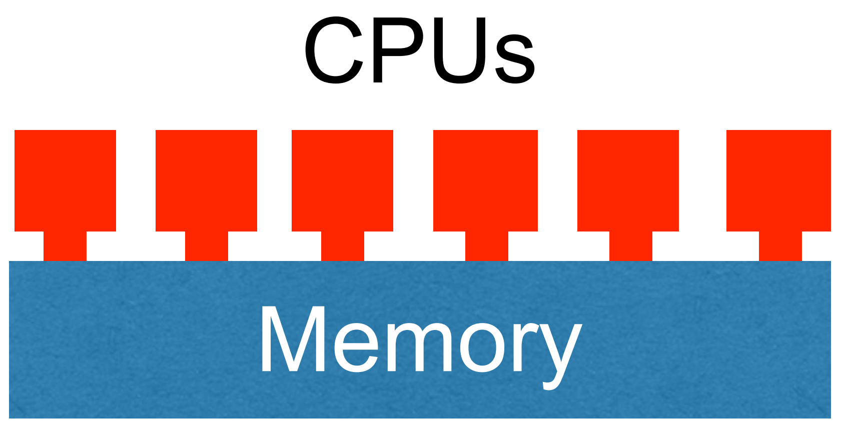
For lower bounds this may be sufficient, for upper bounds, one would need to sum along the critical path; for more on this see Ballard et al. Acta Numerica 2014

Lessons so Far

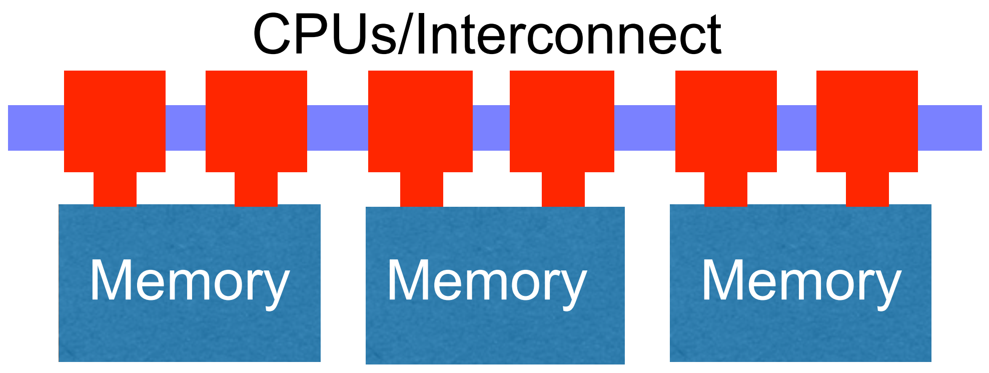
- **Life is complicated**
  - In case real performance gains are required, simple “code ports” may not work
  - Rethinking of algorithms in ways that align with modern architectures will be likely needed; in many cases, this is an art, yet to become a science
  - But we don’t all have to become performance experts if extreme performance is not required; fortunately this is rarely the case
  - Embarrassingly parallel applications with low compute intensity should be easy to implement on all platforms
  - Portability on multi/manycore systems is important (likely to be primarily via the use of OpenMP)

Parallel Programming: MPI and OpenMP

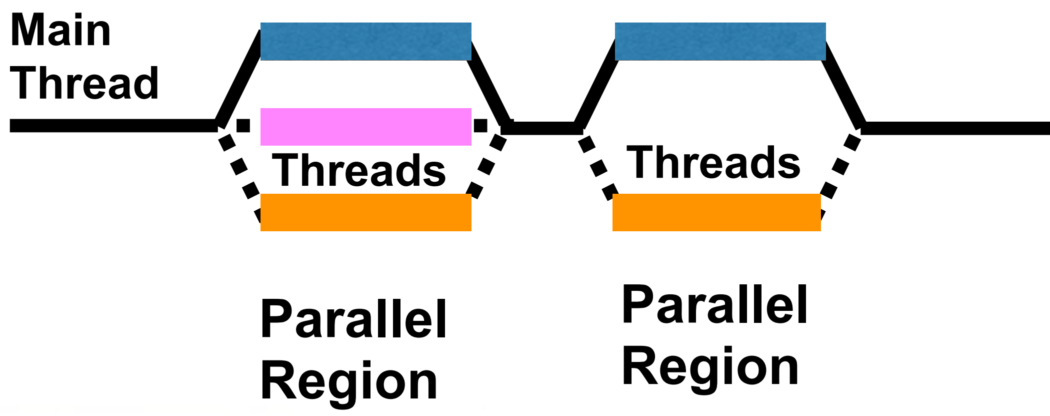
- **MPI (Message Passing Interface)**
  - We won't cover it today; MPI is essentially a distributed memory model approach independent of the actual architecture
- **OpenMP (Open Multi-Processing)**
  - Designed for shared memory systems
  - Explicit programming model (parallelization is not automatic)
  - Parallelism expressed solely through threads ('atom' of processing)
  - Parallel execution model is **fork-join**
  - Mostly works through compiler directives (C/C++, Cython, Fortran, —)
  - **Adrian Pope** will take you through some simple examples; you can run them on your NERSC accounts



Uniform Memory Access



Non-Uniform Memory Access (NUMA)



Where to Run?

- **Options**
  - LSST repo at NERSC will probably allow people to try things out, but allocations may not be enough for serious production (however, no such need appears to exist currently)
  - A recently started project across Argonne and LBNL will have LSST software targets; NCSA will likely partner (software containers to allow data-intensive tasks to run on HPC resources, to stay in touch send email to Salman Habib or Peter Nugent)
  - Portability on multi/manycore systems is important (likely to be primarily via the use of OpenMP)
  - Cloud as a resource will work for many use cases that don't require major compute intensity