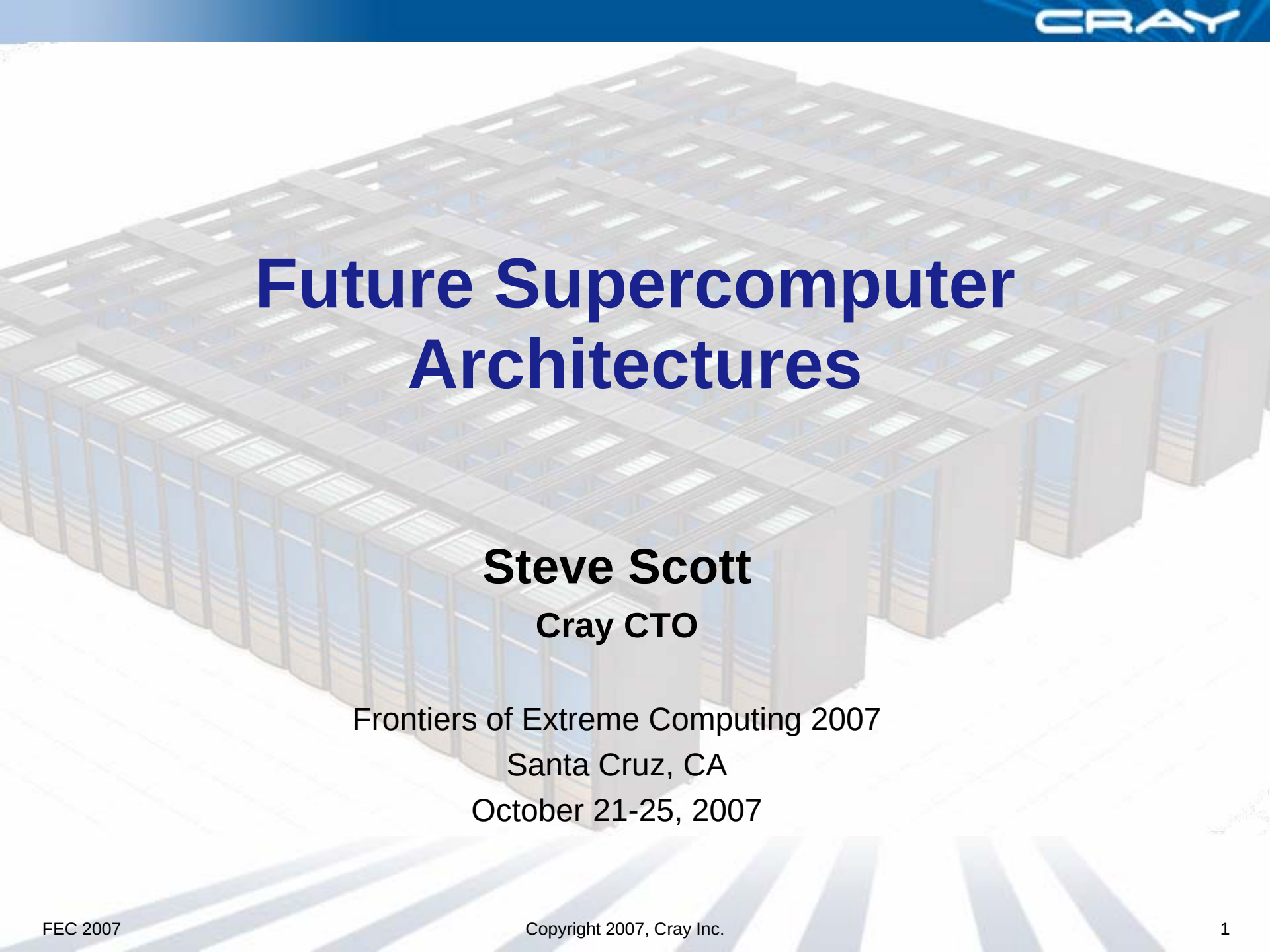


# Future Supercomputer Architectures

**Steve Scott**

**Cray CTO**

Frontiers of Extreme Computing 2007

Santa Cruz, CA

October 21-25, 2007

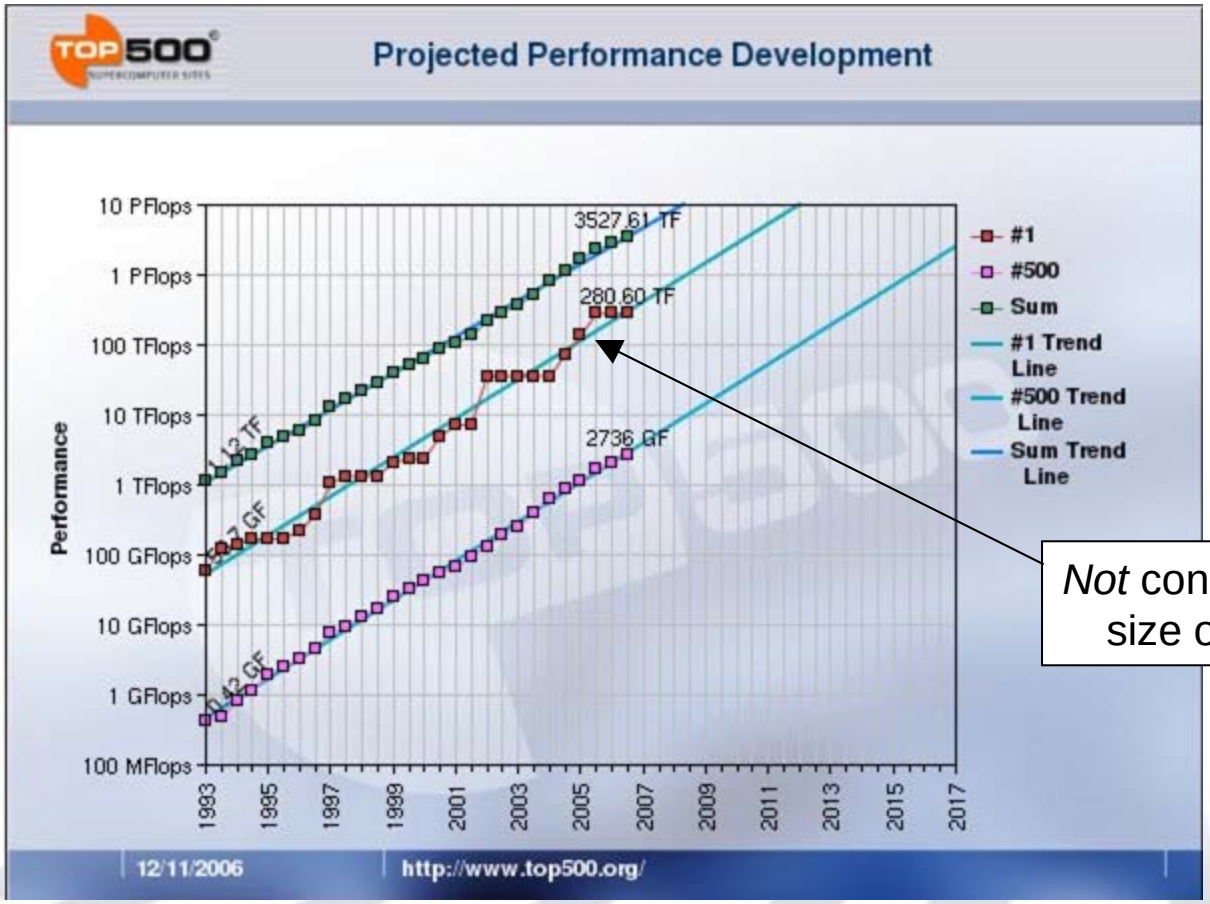
# Where Are We Today?

- O(100 TF) systems
- Two designs at the top end:
  - IBM Blue Gene: custom 3D torus, SOC design w/ embedded PowerPC processors, relatively slower/lower power processors and smaller node memories, COTS memory chips
  - Cray XT4: custom 3D torus, COTS x86-64 processor, COTS memory DIMMS, higher W per peak flop, but similar W/delivered flop
- Multi-core scalar CMOS processors
- Programming model is C and Fortran with MPI (some OpenMP)
- GPFS and Lustre parallel file systems

- Petaflop systems will be straight-forward extrapolations:
  - O(20K x86 sockets), O(100K BlueGene sockets)
  - 4-8 cores per socket
- Key issues:
  - Application scaling and tuning
  - System software tuning and maturing

# March to a Lottaflop

- Milestones: Peak, Linpack, Application, Broad set of apps
- Easy: just wait till 2009 for a Linpack petaflop, 2019 for an exaflop, 2030 for a zettaflop  (right?)




*Not* constant cost, size or power

# Major Processor Inflection Point

- Have pushed pipelining about as far as practical
- Have mined most of the available instruction level parallelism
  - ⇒ Benefit of more complex processors has significantly diminished
- Power becoming *the* issue (both heat density and electricity cost)
  - ⇒ Must moderate frequencies
  - ⇒ Focus on parallel performance vs. fast thread performance
- Flops getting increasingly cheap relative to bandwidth
  - Energy cost of moving data (even across chip) dwarfs that of computation
  - ⇒ Okay to overprovision flops to maximum utilization of memory bandwidth (but even flops are too power hungry at the exascale)
- Wire delay starting to dominate over transistor delay
  - ⇒ Can't treat whole chip as single processor; must exploit locality

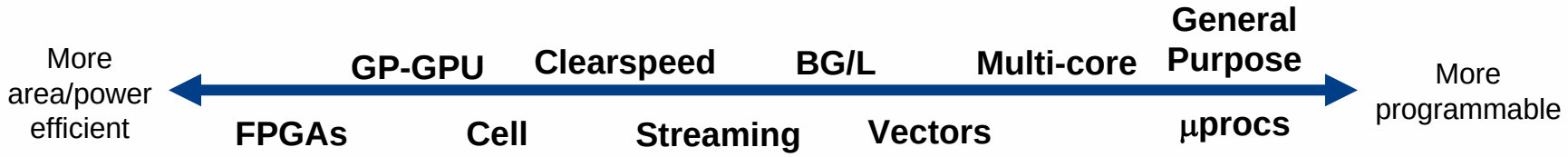
- Commercial response has been to go multi-core
  - Helps alleviate many of these problems
    - Keeps processors simpler and smaller
  - Will work likely work well on many applications
    - Transaction processing, web serving
    - Highly scalable, dense, regular, blockable, well load-balanced, HPC applications

# Concerns With Multicore

- Rapidly increasing number of processors per system
  - Million-thread MPI applications, anyone?
- Contention for bandwidth off chip
  - Ratios are continuing to worsen
- Synchronization, load balancing and managing parallelism across cores
  - Synchronization is very expensive/heavyweight in traditional processors
  - Dynamic problems create severe load imbalances, which require sophisticated runtime software, synchronization and bandwidth to address
- Memory latency continues to grow and is becoming more variable
  - Traditional processors are not latency tolerant
- Still a lot of control overhead in conventional processors
  - Complex pipelines, extensive prediction schemes, highly associative cache hierarchies, replay logic, high instruction processing bandwidth, etc.
  - Flies in the face of efficient *parallel* performance
- Lots of experimentation with alternative microarchitectures and accelerators

# So, Can We Just Pack Chips with Flops?

- Key is making the system easily programmable
- Must balance peak computational power with generality
  - How easy is it to map high level code onto the machine?
  - How easy is it for computation units to access global data?
  - Machine-specific tuning or code mods required?
- Some examples:
  - FPGAs, Clearspeed CSX600, IBM Cell, GP-GPUs
  - All require significant changes to the source code
  - None have a whole program compiler
- Flop efficiency vs. generality/programmability spectrum:
  - Qualitative only
  - Also influenced by memory architecture, memory system and network



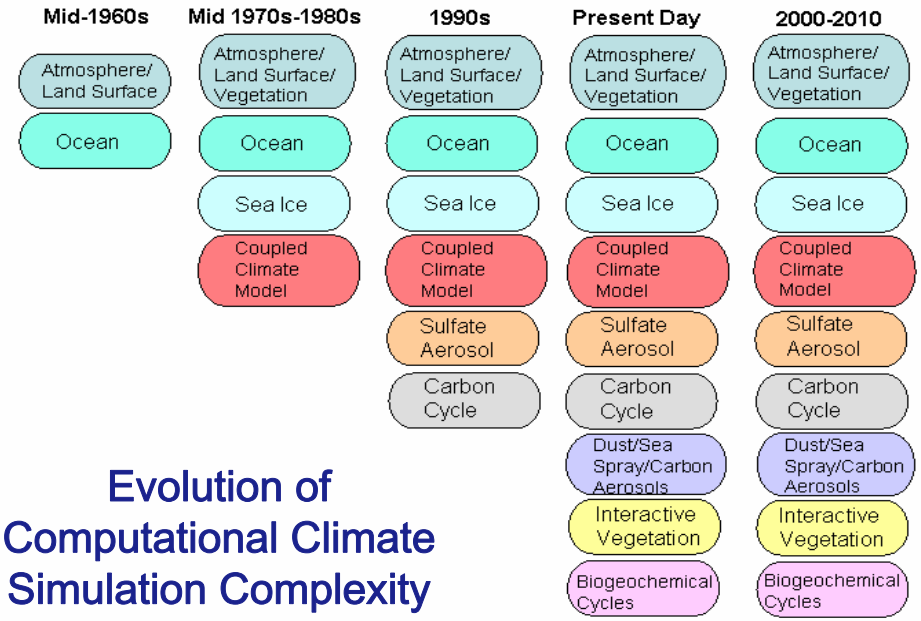

*Resolving this tension is the primary challenge of computer architecture over the next decade.*

# Opportunities to Exploit Heterogeneity

- Applications vary considerably in their demands
- Any HPC application contains some form of parallelism
  - Many HPC apps have rich, SIMD-style *data-level parallelism*
    - Can significantly accelerate via vectorization
  - Those that don't generally have rich *thread-level parallelism*
    - Can significantly accelerate via multithreading
  - Some parts of applications are not parallel at all
    - Need fast serial scalar execution speed (Amdahl's Law)
- Applications also vary in their communications needs
  - Required memory bandwidth and granularity
    - Some work well out of cache, some don't
  - Required network bandwidth and granularity
    - Some ok with message passing, some need shared memory
- No one processor/system design is best for all apps

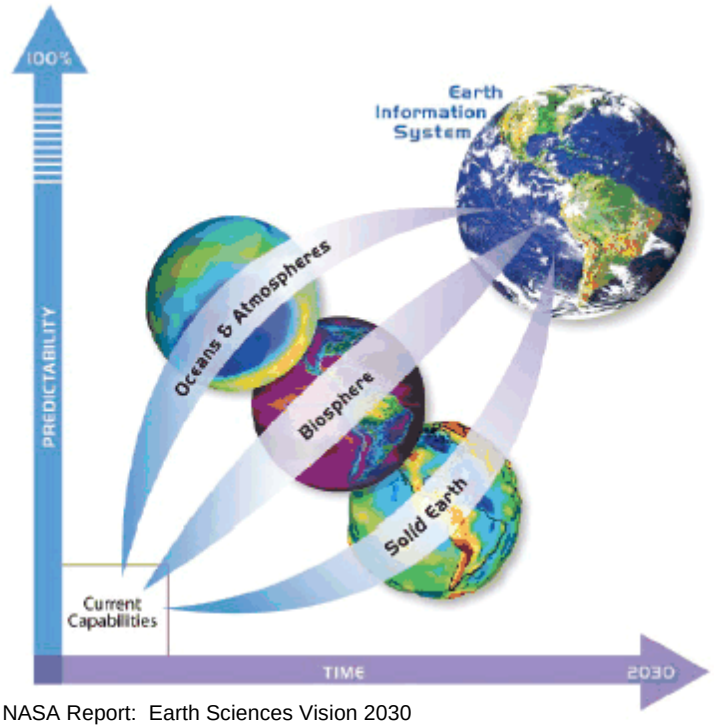
# Increasingly Complex Application Requirements

## Earth Sciences Example




Evolution of Computational Climate Simulation Complexity

International Intergovernmental Panel on Climate Change, 2004, as updated by Washington, NCAR, 2005




NASA Report: Earth Sciences Vision 2030

*Increased complexity and number of components lends itself well to a variety of processing technologies*

- Similar trends in astrophysics, nuclear engineering, CAE, etc.
  - Higher resolution, multi-scale, multi-science

# Cascade Approach to Higher Productivity

## Design an **adaptive, configurable** machine

- Serial (single thread, latency-driven) performance
- SIMD data level parallelism (vectorizable)
- Fine grained MIMD parallelism (threadable)
- Regular and sparse bandwidth of varying intensities

⇒ Increases performance

⇒ Significantly eases programming

⇒ Makes the machine much more broadly applicable

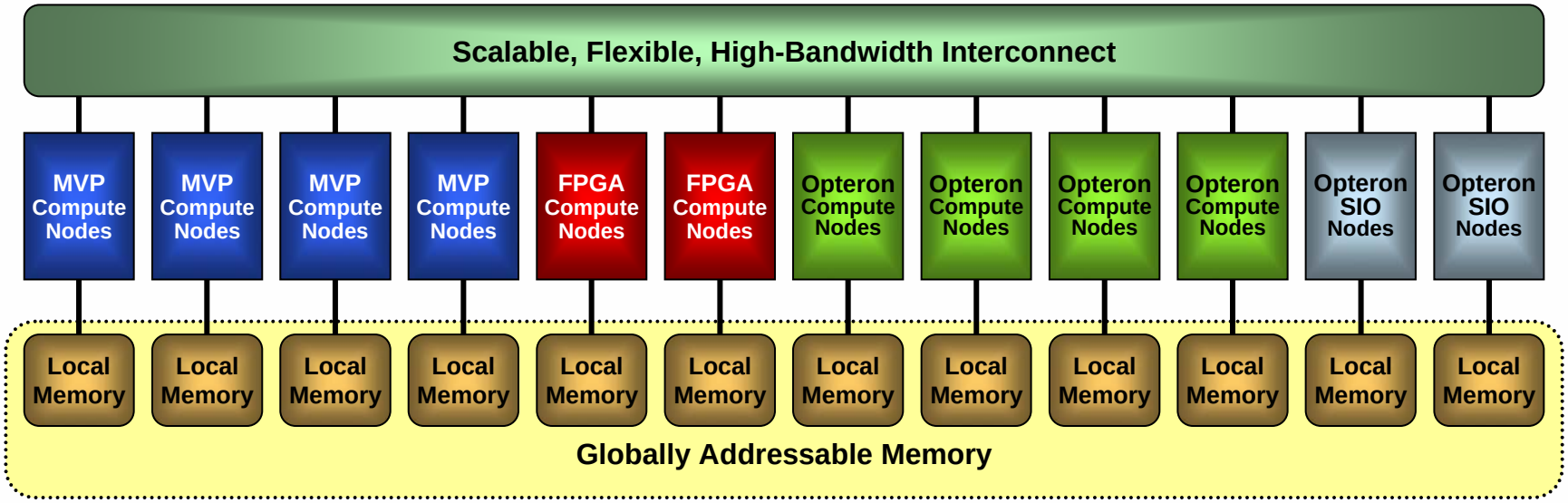
## Ease the development of parallel codes

- Legacy programming models: MPI, OpenMP
- Improved variants: SHMEM, UPC and CoArray Fortran (CAF)
- New alternative: Global View (GMA, Chapel)

## Provide programming tools to ease debugging and tuning at scale

- Automatic performance analysis; comparative debugging

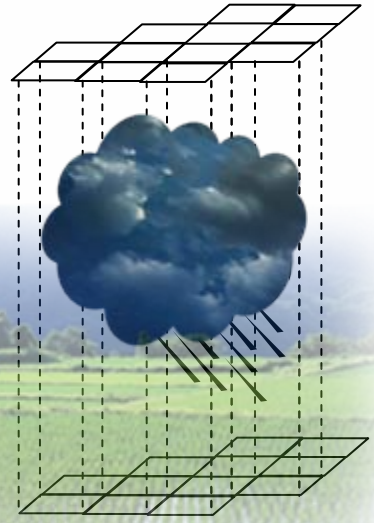
# Cascade System Architecture




- Globally addressable memory with unified addressing architecture
- Configurable network, memory, processing and I/O
- Heterogeneous processing across node types, and within MVP nodes
- Can adapt at configuration time, compile time, run time

# Example Application:
## Weather Research & Forecasting (WRF) Model

- Mesoscale numerical weather prediction system
  - Regional forecast model (meters to thousands of kilometers)
- Operational forecasting, environmental modeling, & atmospheric research
  - Key application for Cray (both vector & scalar MPP systems)
- Accelerating WRF performance:
  - Part of the code is serial:
    - Runs on Opteron for best-of-class serial performance
  - Most of the code vectorizes really well
    - Dynamics and radiation physics
    - Runs on Granite accelerator in vector mode
  - Cloud physics doesn't vectorize
    - Little FP, lots of branching and conditionals
    - Degrades performance on vector systems
    - Vertical columns above grid points are all independent
    - Runs on Granite accelerator in multithreaded mode



# Key Challenges to Get to the Exascale (1)

- Okay, all we need is a system with 5-10 TF/chip and 100-200K chips…

- Power
  - If 2014 FPU is ~10-20 pJ/flop, then 1 EF = 10-20MW for flops alone!
  - Memory bandwidth:
    - Eref/sec * 10% miss * 100 bits * 3pJ/bit = 30 MW for memory bandwidth
  - These are aggressive numbers and leave much of the power out
  - Houston, we have a problem!

- Processor microarchitecture to exploit locality
  - Resolving the tension between programmability and efficiency
  - Need a new microarchitecture and execution model
    - Much lower control overhead relative to computation
    - Much more aggressive exploitation of locality (explicit control of data movement)
  - Co-design hardware with compiler/runtime software
  - Do *not* burden the programmer with this!
    - Need to be able to write in a portable HLL and compile to the microarchitecture
    - Best if the compiler took care of the parallelism at the node level (so user sees fewer, more capable nodes)

# Key Challenges to Get to the Exascale (2)

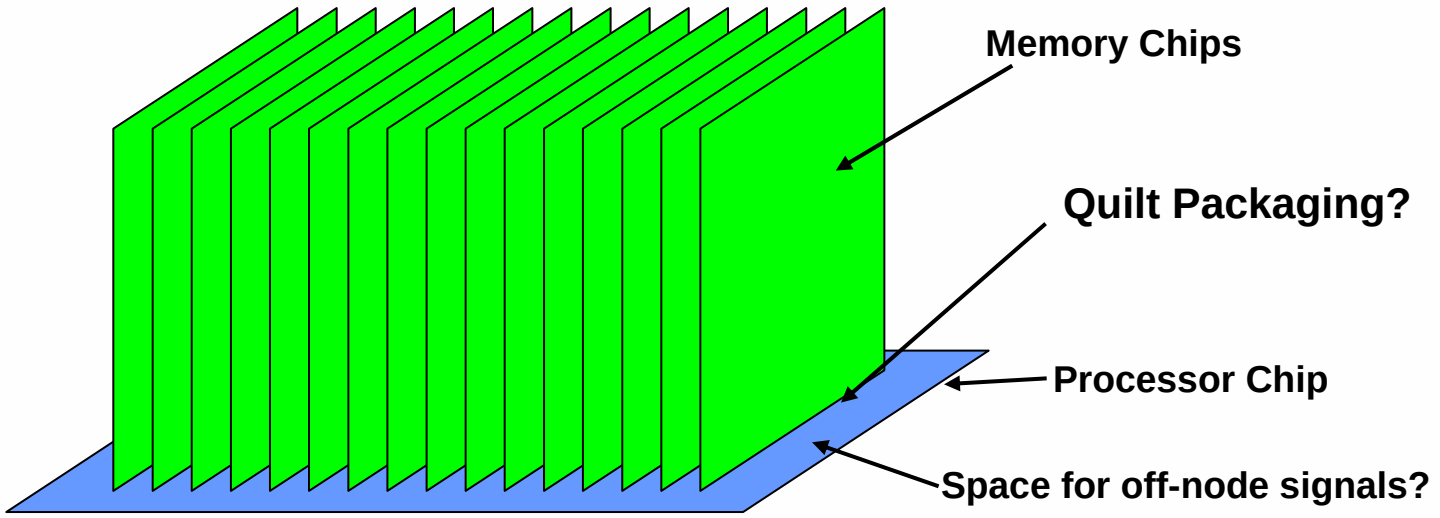
- System and application resiliency
  - Memory alone will require O(100M) chips
    - Assume N^3/4 scaling starting from 1B/flop at the Tflop ⇒ need 32 PB
    - If 16 Gbit chips, w/ ECC, need 18M chips (biggest sys. today < 2M)
  - Forget trying to make hardware MTBF acceptable (*deal* with failures)
  - *System* must be resilient to almost any single point of failure (and more)
    - Can be done with good system software design and appropriate hardware hooks
  - Completing *applications* much harder than keeping system up
    - Could bite the bullet (and watt!) for redundant logic everywhere
    - Checkpoint/restart likely doesn't scale due to I/O bandwidth
      - but consider adding SSD layer
    - Frameworks for resilient apps (think NWChem and beyond)
    - May be able to be automated (STM/runtime/compiler)

- Local memory bandwidth technologies
  - Processors + DIMMs has to go (too big of a bottleneck!)
  - 3D chip stacking, direct optical memory connections
  - Nano-structure memory; phase-change memory; ….
    - help me here, technologists!
    - (I like that Tbit/cm^2 crosspoint memory Stan!)

# A Suggestion for Packaging Granularity

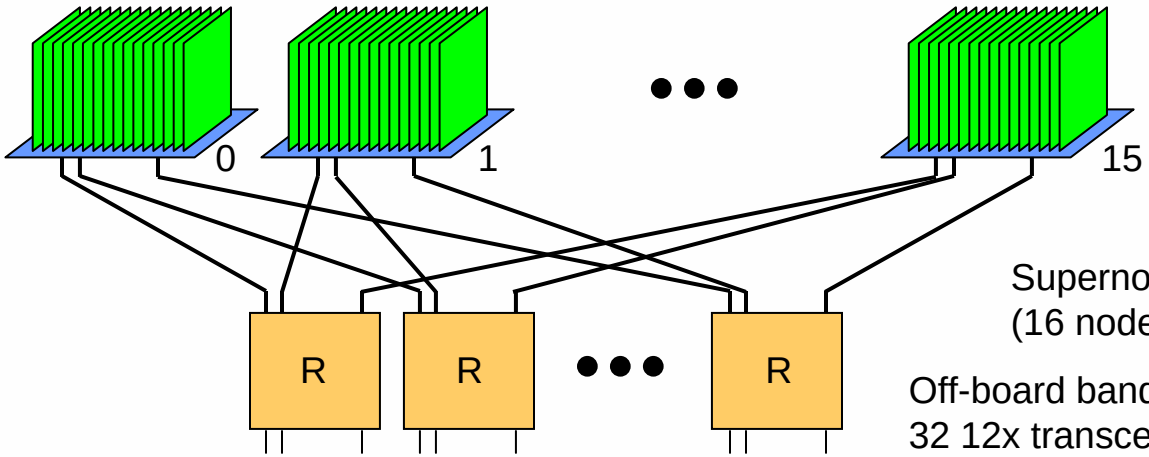
- Put as much memory as you can attached directly to the processor
  - Call this a node
  - It's fairly small
- Build as good a network as you can between nodes
  - Optimize the network around packaging constraints (lots more bandwidth on the board than off it)
  - Provide support for board-level domain of "flat" shared memory
  - Distributed memory model beyond the board
- Layer on a good system architecture
  - Globally addressable memory
  - Scalable address translation and synchronization
  - Latency tolerant processor architecture

# 3D Node: Processor + Orthogonal Memory Chips




- Capacity
  - 8-32 memory chips @ 1 GB each = 8-32 GB per node
- Bandwidth
  - 5 μm pitch wires (10 μm per diff signal), 15mm edge ⇒ 1500 signals per memory chip
  - Need to keep signaling rates to < 10 Gbps with memory periphery transistors
  - Assume 512 bits/dir @ 8.25 Gbps, packetized protocol, 80% read efficiency
    - ⇒ 320 GB/s read bandwidth per memory chip (1.28W at 0.5 pJ/bit)
    - ⇒ 2.5-10 TB/s read bandwidth per node with 8-32 memory chips
  - Could nicely feed a 5-10 TF node
  - Probably still too much power in memory chips to support this…

# Example Board Architecture




*Shown as fat-tree. Could consider flattened butterfly or other topology for on-board or off-board links.*

Aggregate node bandwidth:
(16 nodes)*(4 TB/s/node) = 80 TB/s

Supernode bandwidth:
(16 nodes)*(64 sigs/node)*(25 Gbps) = 6.4 TB/s

Off-board bandwidth:
32 12x transceivers @ 16 Gbps = 768 GB/s

- Can treat as 16 nodes for highest aggregate memory bandwidth
- Could combine into 2, 4, 8 or 16-node “super-nodes”
  - Flat addressing, latency and bandwidth
  - Hashed to avoid bank conflicts
  - Would still want compiler to exploit locality within a single node
    - Either via explicit local segments or via caching (possibly in main memory)
- Inter-node signaling shown using conservative technology extrapolations
  - Could also consider high-bandwidth on-board technologies (quilting, capacitive coupling, optics?, etc.) to boost super-node bandwidth even further

# Key Challenges to Get to the Exascale (3)

- Programming difficulty
  - MPI is a low-productivity programming model
    - What's more, it's not *really* portable, in that it is a wholly unsuitable programming model for machines with global memory, advanced latency hiding mechanisms, and low overhead synchronization
  - Debuggers don't scale
  - Performance tools don't scale
  - Time is right for a high productivity language

# Chapel

## A new parallel language developed by Cray for HPCS

### Themes

- Raise level of abstraction, generality compared to SPMD approaches
- Support prototyping of parallel codes + evolution to production-grade
- Narrow gap between parallel and mainstream languages

### Chapel's Productivity Goals

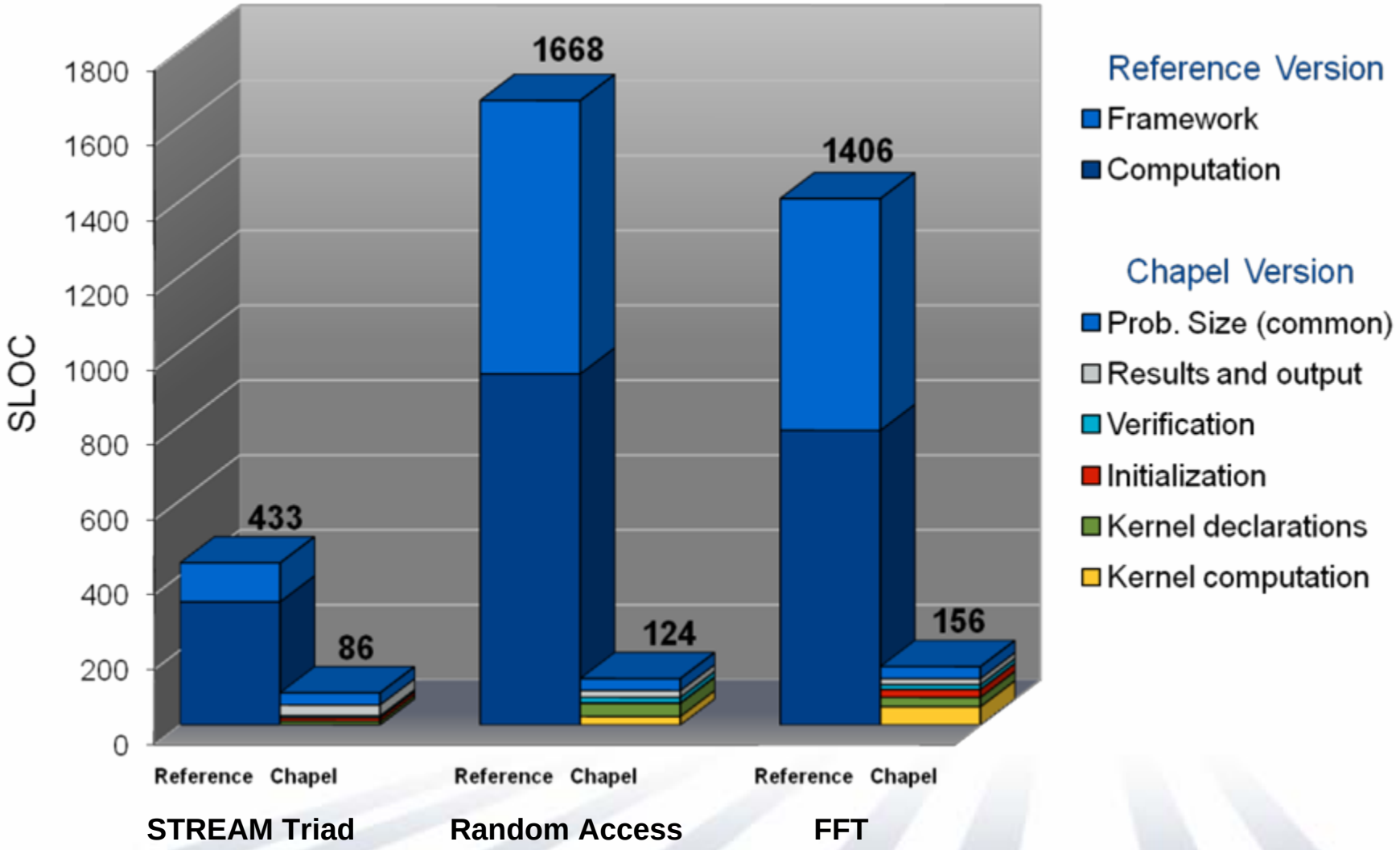
- Vastly improve **programmability** over current languages/models
- Support **performance** that matches or beats MPI
- Improve **portability** over current languages/models (actually better than MPI)
- Improve code **robustness** via better abstractions and semantics

```
...
coforall block in UpdateSpace.subBlocks do
  for r in RAStream( block ) do
    T( r & indexMask ) ^= r;
```

### Status

- Draft language specification available
- Portable prototype implementation underway
- Performing application kernel studies to exercise Chapel
- Working with HPCS mission partners & parallel community to evaluate Chapel
- Initial evaluation releases made available December 2006, June 2007

# Chapel Code Size Comparison
## For HPC Challenge Benchmarks



# One Last Exascale Challenge (4)

- Need to build systems for *tomorrow's* applications
  - Irregular, dynamic, sparse, heterogeneous….
  - Codes that don't exhibit locality, or that have limited per-thread concurrency
  - Need to start, stop, move and synchronize computation efficiently
  - Let's not solve the scaling problem for the easy apps and declare success

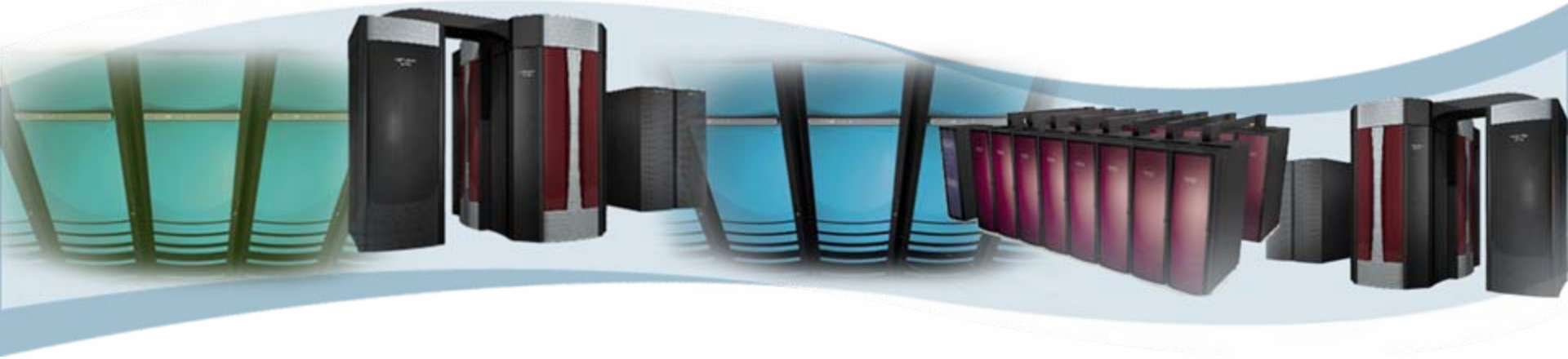
  - "Leave no application behind"

# Key Challenges to Get to the Zettascale

- I accept that CMOS won't get there due to power and other reasons.
- New computing technologies will likely require new architectures, new execution models and new programming models
  - Exploitation of locality will be key
  - Very likely to involve massive threading and lightweight thread migration
- Architects need to understand the technological sandbox within the next dozen years or so…
- Absolutely must have better programming models where humans don't have to coordinate all the data distribution and communication
  - Would be nice if those were the same programming models used at Exascale
- Need to have much more sophisticated and automated tools for performance and correctness analysis
  - Presumably involving pervasive introspection

- I am an optimist. I think we *will* get to zettaflop computing using some interesting post-CMOS technology by ~2030. It will look different than any of us imagine today. Good occasion to retire.

# Thank you.



# Questions?

sscott@cray.com