Programming Model at Extreme Scale

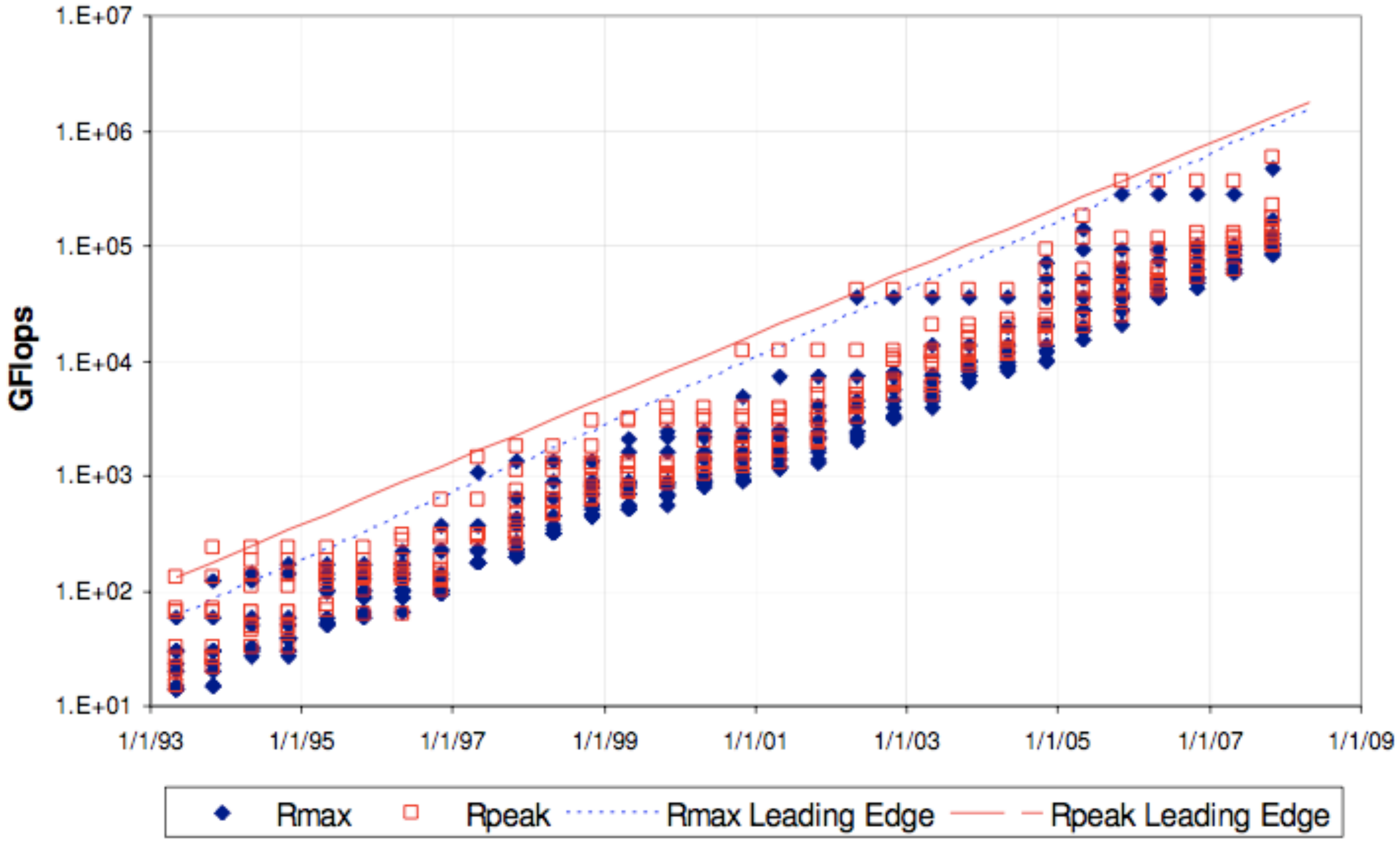
Marc Snir

June 2009

PARALLEL@ILLINOIS

www.parallel.illinois.edu

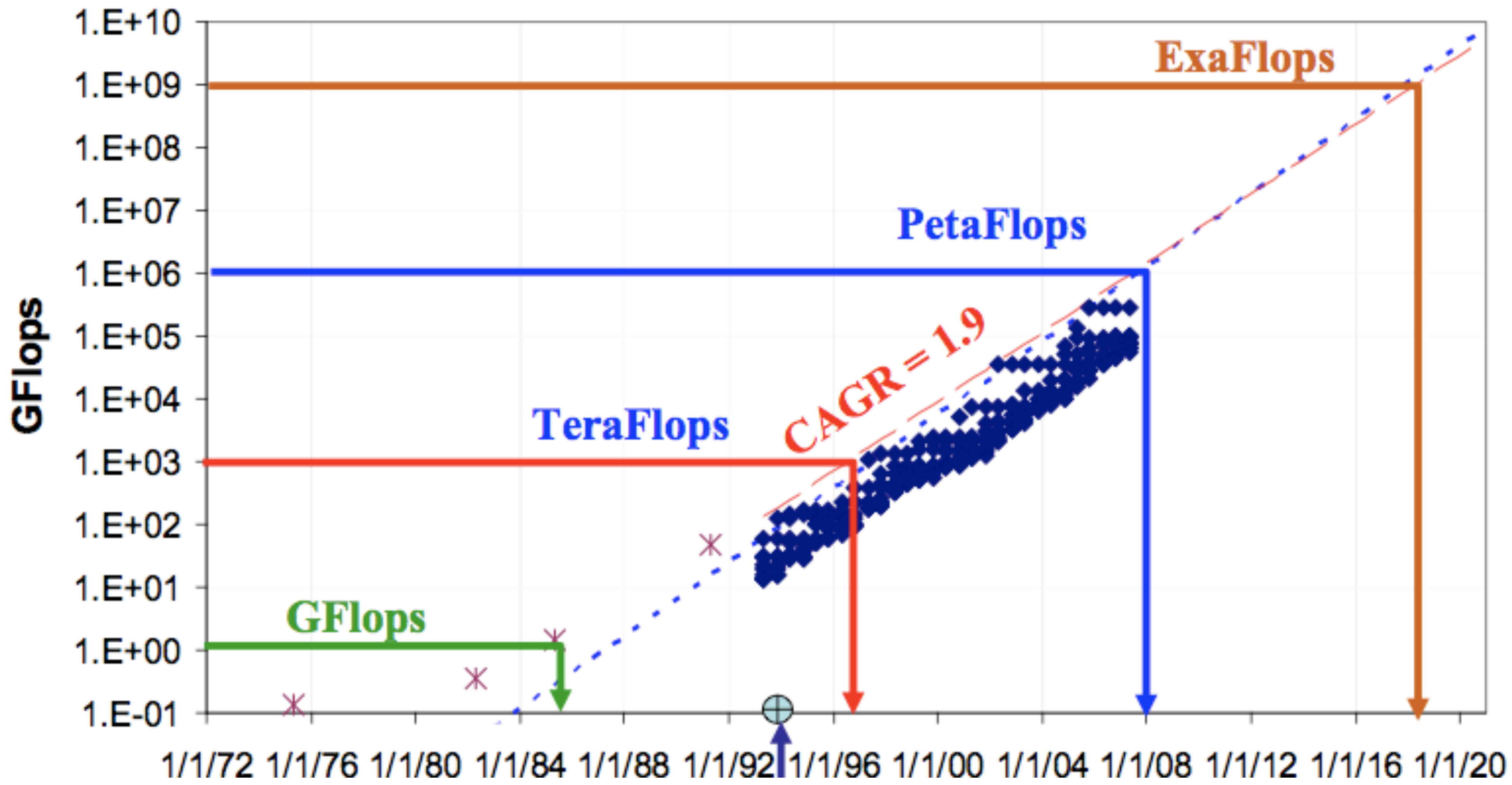
Supercomputer Performance Evolution



(Kogge et al.)

PARALLEL@ILLINOIS

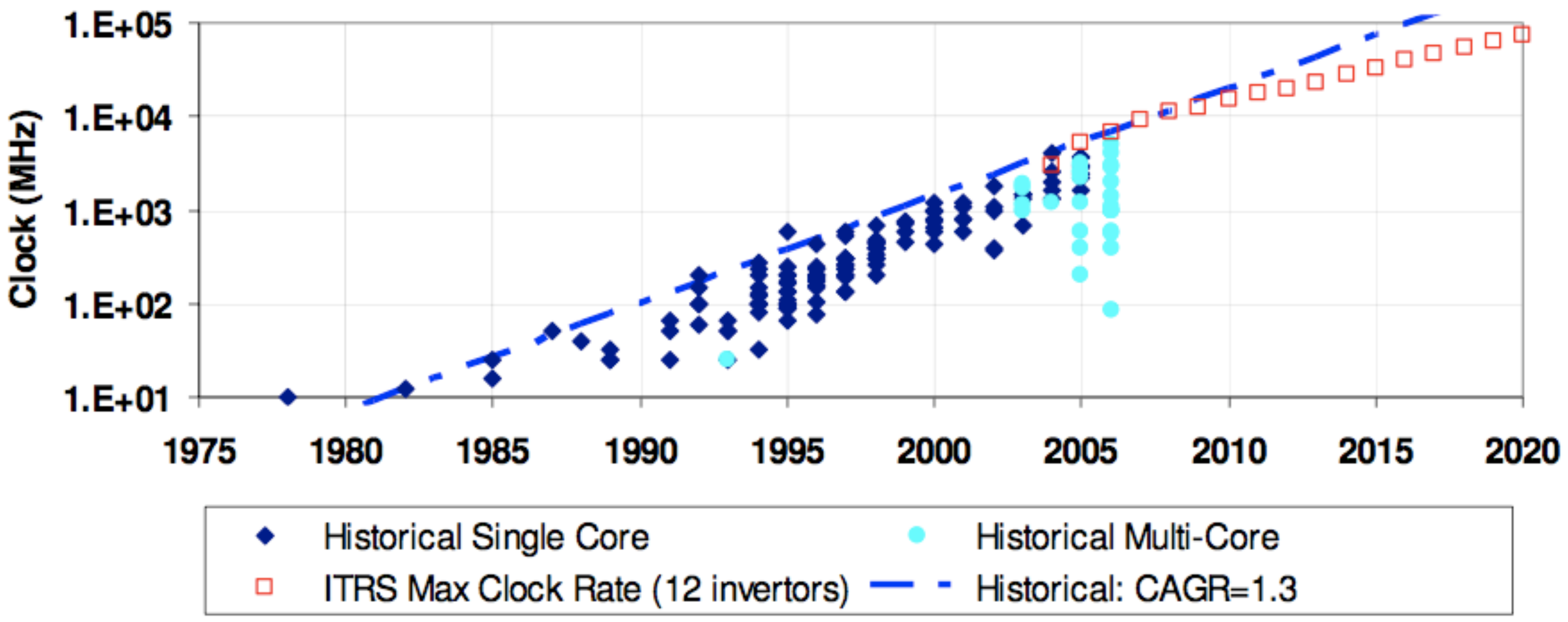
Performance growths 1,000-fold every 11 years



PARALLEL@ILLINOIS

Factors of Performance Growth

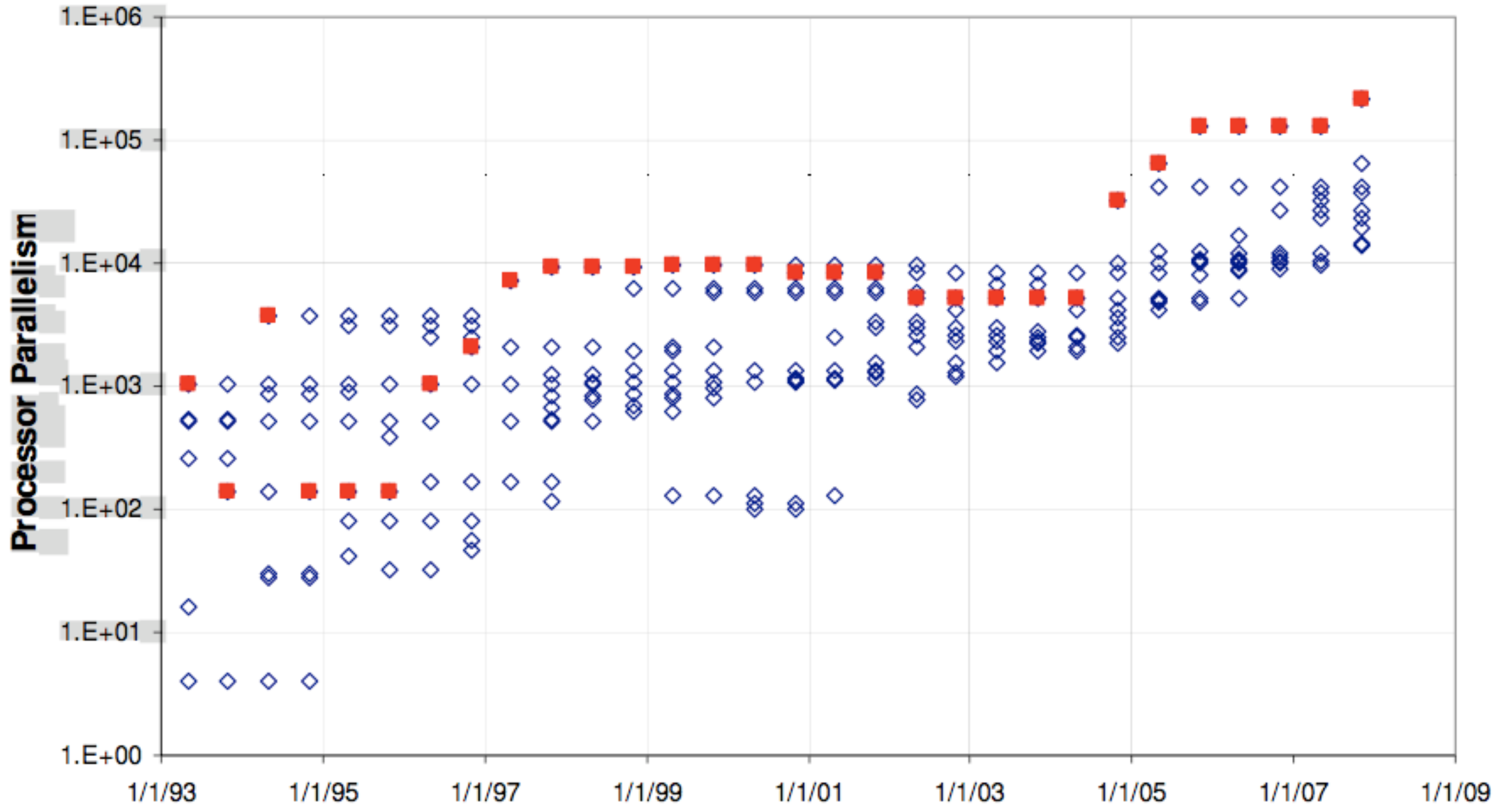
- Growth in clock rate – now slowing



PARALLEL@ILLINOIS

Factors of Performance Growth

- Growth in number of processors and (SIMD) IPC – now accelerating



PARALLEL@ILLINOIS

Toward Exascale

- Transistor density continues to increase (Moore's law)
  - up to 2020 – 8 nm; not clear what happens beyond
- Clock frequency does not increase
  - Power barrier
- Increased performance comes only from increased number of cores per chip, increased use of SIMD instructions and increased number of chips

Exascale in 2020

- Extrapolation of current technology
  - 100M- 1B threads
  - 100-500 MWatts
- Energy consumption might be reduced one order of magnitude with aggressive technology and architecture change
  - Low power cores (more cores)
  - Aggressive voltage scaling (more errors)
  - Aggressive DRAM redesign (less bandwidth)

Main Issues

- Increased parallelism
- Resiliency
- Variability
- Virtualization
- Hybrid HW

Managing with 1M-1B Threads

- Increased parallelism
- Resiliency
- Variability
- Virtualization
- Hybrid HW

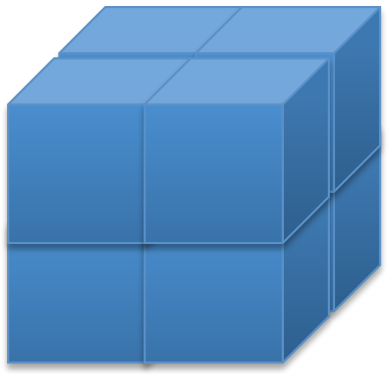
PARALLEL@ILLINOIS

Scaling Applications

- Weak scaling: use more powerful machine to solve larger problem
  - increase application size and keep running time constant; e.g., refine grid
  - Larger problem may not be of interest
  - May want to scale time, not space (molecular dynamics)
  - Cannot scale space without scaling time (iterative methods): granularity decreases and communication increases

Scaling Iterative Methods

- Assume that number of cores (and compute power) are increased by factor of k^4
- Space and time scales are refined by factor of k
- Mesh size increased by factor of $k \times k \times k$
- Local cell dimension decreases by factor of $k^{1/4}$
- Cell volume decreases by factor of $k^{3/4}$ while area decreases by factor of $k^{2/4}$; area to volume ratio (communication to computation ratio) increases by factor of $k^{3/2}$.



Debugging and Tuning: Observing 1B Threads

- Scalable infrastructure to control and instrument 1B threads
- Parallel information compression to identify “anomalies”
- Need to ability to express “normality” (global correctness and performance assertions)

PARALLEL@ILLINOIS

Main Issues

- Increased parallelism
- Resiliency
- Variability
- Virtualization
- Hybrid HW

Decreasing Mean Time to Failure

- Problem:
  - More transistors
  - Smaller transistors
  - Lower voltage
  - More manufacturing variance
- Current technology: global, synchronized checkpoint; HW error detection
- Future technology:
  - Continued HW error detection with Increased HW redundancy & error checking
  - More efficient checkpoint (OS?, compiler? **application**?)
    - OK if number of components stays constant
  - Integrated HW/SW/application approach for fault detection, isolation and local recovery
    - May be needed later if number of components per system increases

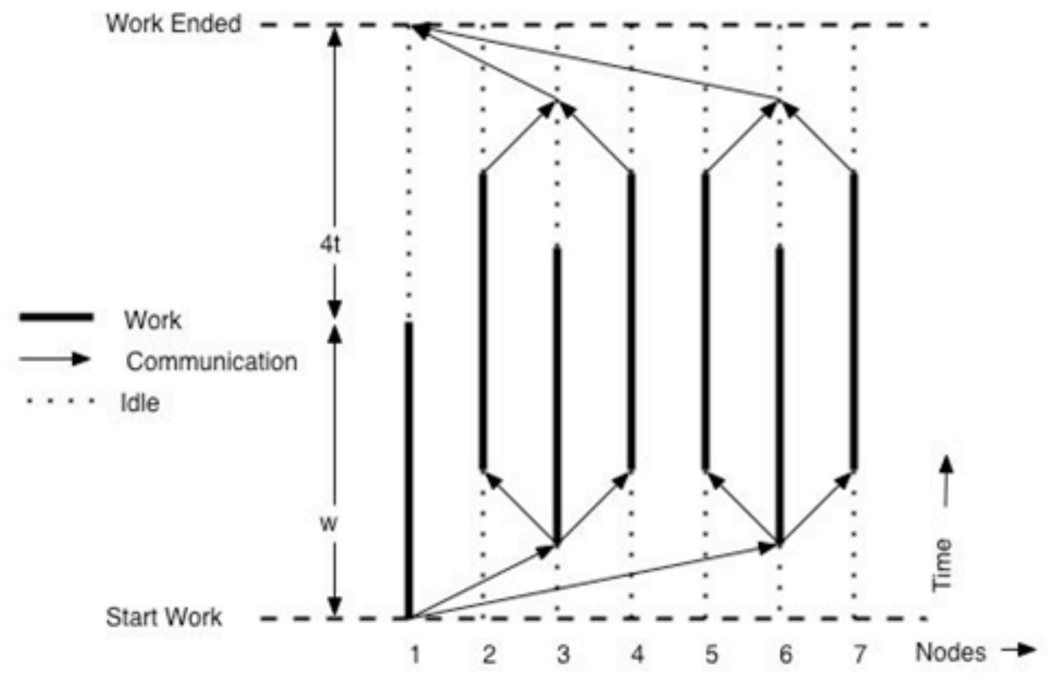
Main Issues

- Increased parallelism
- Resiliency
- Variability
- Virtualization
- Hybrid HW

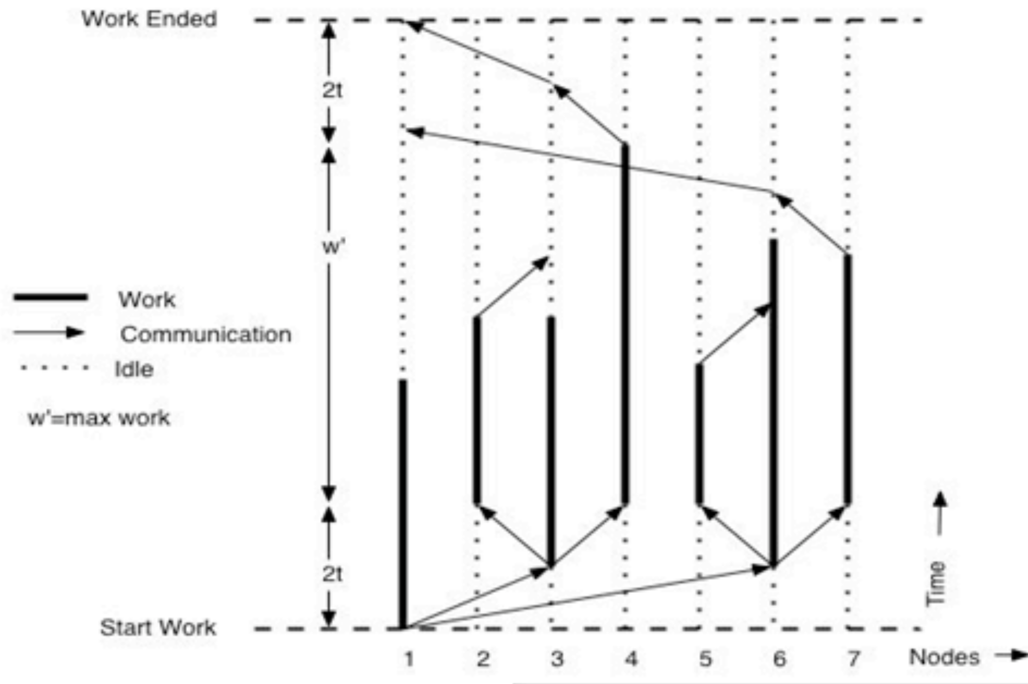
Bulk Synchronous

- Many parallel applications are written in a “bulk-synchronous style”: alternating stages of local computation and global communication
- Models implicitly assumes that all processes advance at the same compute speed
- Assumptions breaks down for an increasingly large number of reasons
  - Black Swans
  - OS jitter
  - Application jitter
  - HW jitter

Jitter Illustrated



Collectives in a cluster with no OS Jitter



Collectives in a cluster with OS Jitter

OS jitter has been empirically measured to slow down computations by a factor of 2 or more

(IBM)

Jitter Causes

- Black Swans
  - If each thread is unavailable (busy) for 1 msec once a month, than most collective communications involving 1B threads take > 1 msec (the black swan effect)
- OS jitter
  - Background OS activities (daemons, heartbeats…)
- HW jitter
  - Background error recovery activities (e.g., memory scrubbing & error handling); power management; management of manufacturing variability; degraded operation modes
- Application jitter
  - Input-dependent variability in computation intensity
- **Need to move away from bulk model**

Possible Approaches

- User helped source code optimization
  - Replace blocking communication (including collective communication) with non blocking communication
  - Refactoring tools help user make changes correctly

```
MPI_Barrier                    MPI_Barrier_start
                               ....
                               MPI_Barrier_end
```

  - Code between start—end should not conflict with code at other processes not separated by full barrier

Possible Approaches (2)

- Compiler optimizations (no change in source code)
  - execute “sends” as early as possible; execute “receives” as late as possible
  - tradeoff with communication aggregation
- Run-time optimization: virtualization

Main Issues

- Increased parallelism
- Resiliency
- Variability
- Virtualization
- Hybrid HW

Task Virtualization

- Multiple logical tasks are scheduled on each physical core; tasks are scheduled nonpreemptively; task migration is supported
  - Hides variance and communication latency
  - Helps with scalability (decouples # tasks from # cores)
  - Helps with resiliency
  - *Needed for modularity* (multiphysics/multiscale codes – handling parallel coupling of modules)
  - *Improves performance* (better locality)
  - *Scales* (Charm++/AMPI)
  - Can be implemented below MPI or PGAS languages

Task Virtualization Styles

- Varying, user controlled number of tasks (AMPI)
  - Locality achieved by load balancer
- Recursive (hierarchical) range splitter (TBB)
  - Method to split (recursively) problem in two sub-problems
  - Method to combine two sub-solutions
  - Method to decide when sub-problem is small enough to be solved sequentially
  - Method to solve sub-problem sequentially

  - Locality is achieved implicitly

Main Issues

- Increased parallelism
- Resiliency
- Variability
- Virtualization
- Hybrid HW

Hybrid Communication

- Multiple levels of caches and of cache sharing
- Different communication models intra and inter node
  - Coherent shared memory inside chip (node)
  - rDMA (put/get/update) across nodes
- Hybrid features change every HW generation
- Need to be able to easily adjust number of cores & replace inter-node communication with intra-node communication
- Easy to "downgrade" (use shared memory for message passing); hard to "upgrade"; hence tend to use lowest commonality (message passing)
- No good interoperability between shared memory (e.g., OpenMP) and message passing (MPI)

Possible Directions

- Express cache oblivious algorithms using recursive range splitting
  - May provide 3 methods:
    - Distributed memory splitting/merging
    - Shared memory splitting/merging
    - Sequential
- (Low hanging fruit) Enable shared memory communication across MPI tasks

PARALLEL@ILLINOIS

Hybrid Computation

- Vector instructions
- Different core types
- Accelerators

- Can significantly reduce energy per flop
- Require (now) different source code
- Easy to compile CUDA to multicore; hard to compile OpenMP to GPU

Possible Approaches

- Use library auto-tuning
- Reduce semantic gap at architecture level; use (static or dynamic) compilation

Shared Memory Programming

- Explicit parallel control
- Global name space: any thread can access any variable by same name
- Caching, not copying: local copies are needed for performance, but are accessed using same global name
  - copying can be implicit (HW caches) or explicit (check-in/check-out, ownership transfer)
- Enforced determinism: conflicting accesses are always ordered
  - relaxed ordering for associative operations is OK

Thread Parallelism: OpenMP

- Explicit parallel control, global name space, implicit caching
- No information on communication patterns => Hard to optimize communication in large systems with no HW support to cache coherence
- Does not prevent memory races and suffer from hard to find synchronization bugs
- Encourages programming style with low granularity and low locality

PGAS Languages: UPC & CAF

- Simple parallel control (as MPI): fixed number of communicating tasks
- Global name space, but no caching
- No information on communication patterns => hard to optimize communication
- Does not prevent memory races
- Main advantages:
  - Easy to port from MPI
  - Communication is compiled and can be optimized

Possible Alternative

- Global name space, explicit control (similar to OpenMP)
- Shared data structures are partitioned by the program into disjoint regions
  - declaratively (type annotations, directives)
  - imperatively
- Protocol is defined *and enforced* to ensure that concurrent accesses to a region are non-interfering
  - Read/Write: only one writer per region at any time
  - Read/Write/Accumulate …
- Informally: programmer defines "logical cache lines"; cache line state can only change at synchronization points

Two Examples

- Multiphase Shared Arrays (Kale)
  - Arrays are partitioned, with one “owner thread” per partition
  - Each array can be in one of three modes: shared/exclusive/ accumulate
  - Protocol is enforced by run-time (assume simple partitions)
- Deterministic Parallel Java (V Adve)
  - Type annotations are used to partition the heap
  - Effect annotations are used to specify which region can be affected (read/written) by each method or code block
  - Protocol is enforced by compiler

Research Issues

- Simple, static partitions vs. complex, dynamic partitions
  - Note: DPJ regions are defined declaratively, but regions can be execution-dependent (nested types, parametric types)
- Compile time enforcement (preferred, but possibly restrictive) vs. run-time enforcement (less restrictive, but possibly expensive, especially for irregular partitions)
- Expressiveness and ease of programming

?