Summer School

e-Science with Many-core CPU/GPU Processors

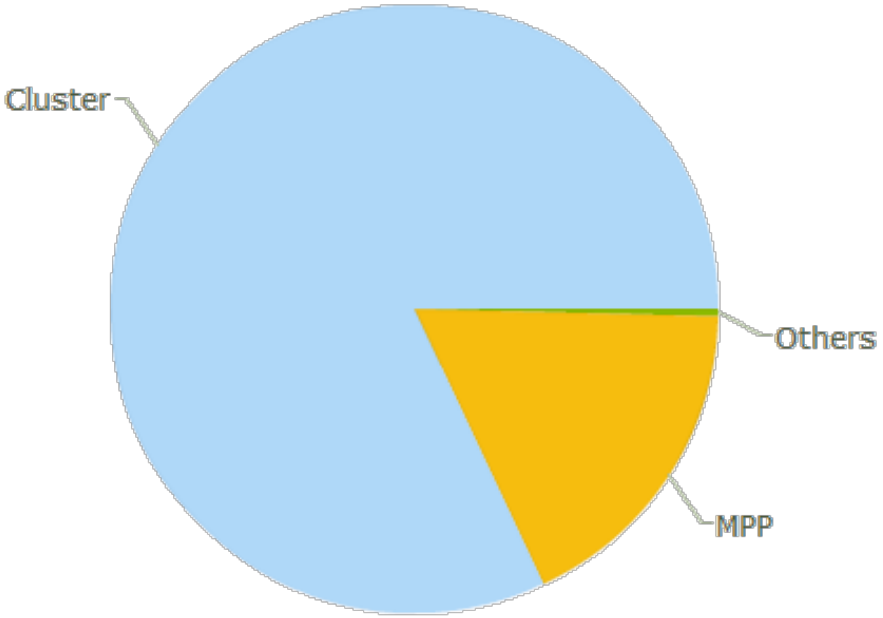
Lecture 5

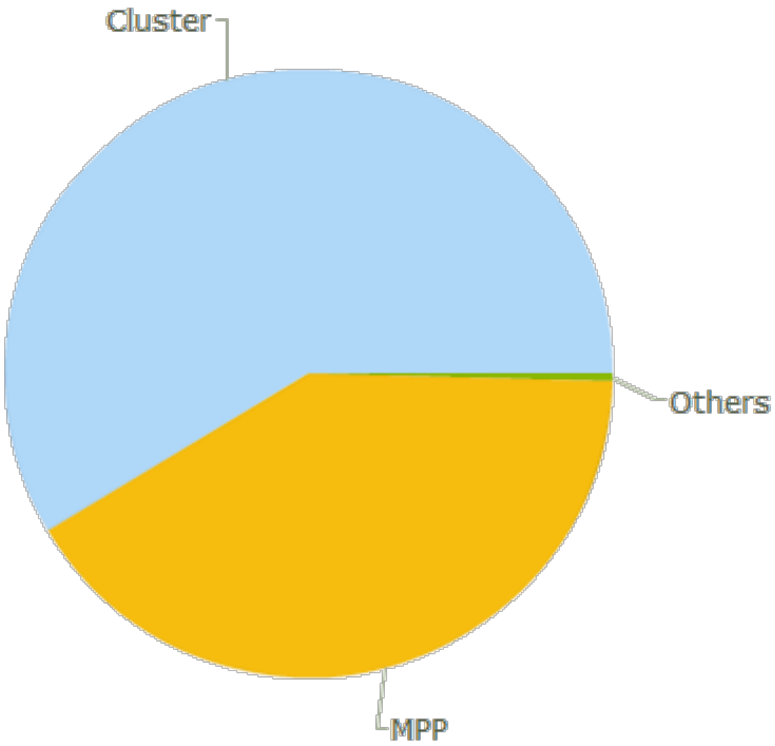
Multiple GPUs in an MPI Cluster

Braga, Portugal, June 14-18, 2010

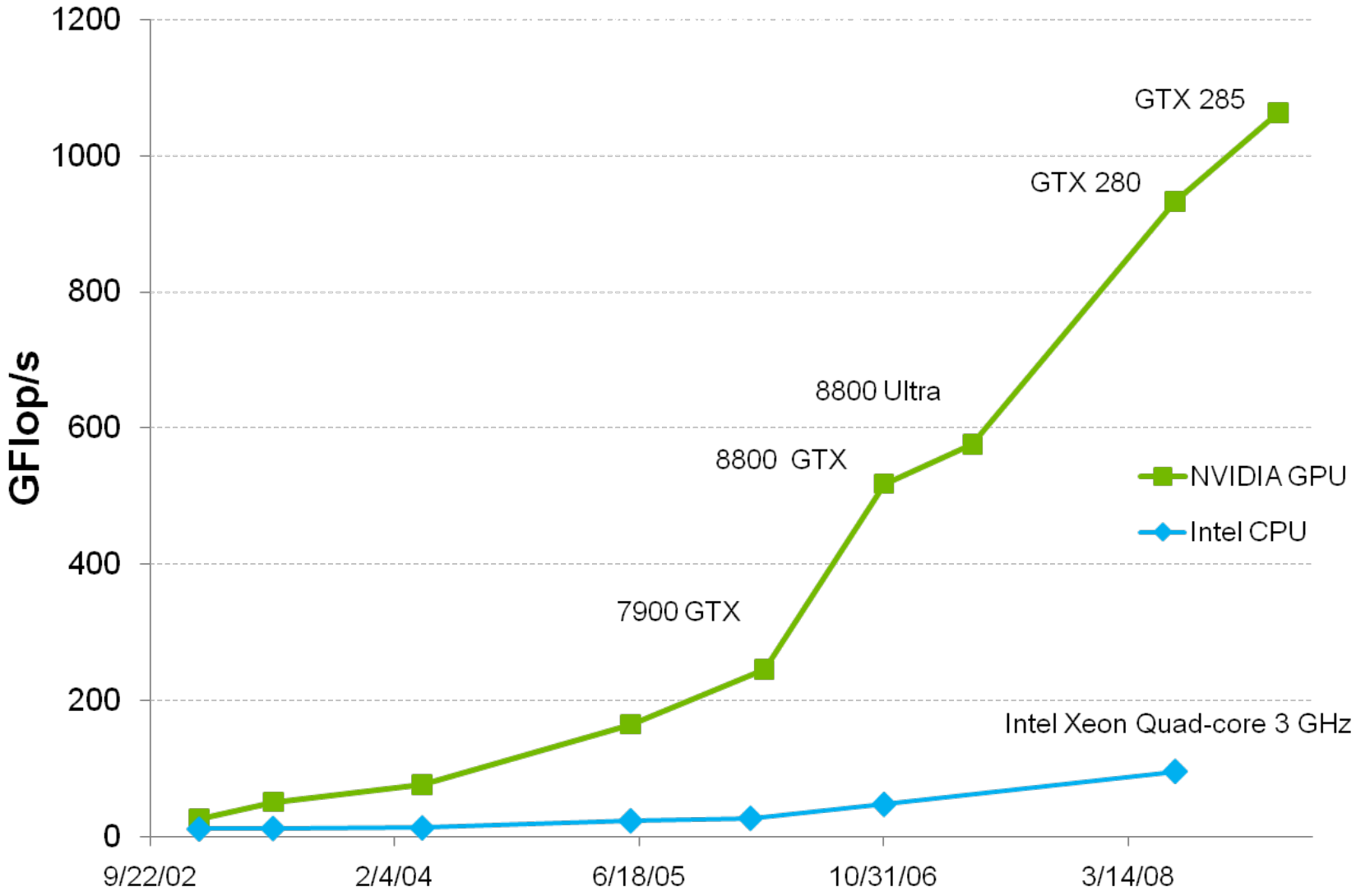
Why Clusters for HPC?

- Clusters are a major workforce in HPC
  - Q: How many systems in top500 are clusters?
  - A: 410 out of 500





Why GPUs in HPC Clusters?



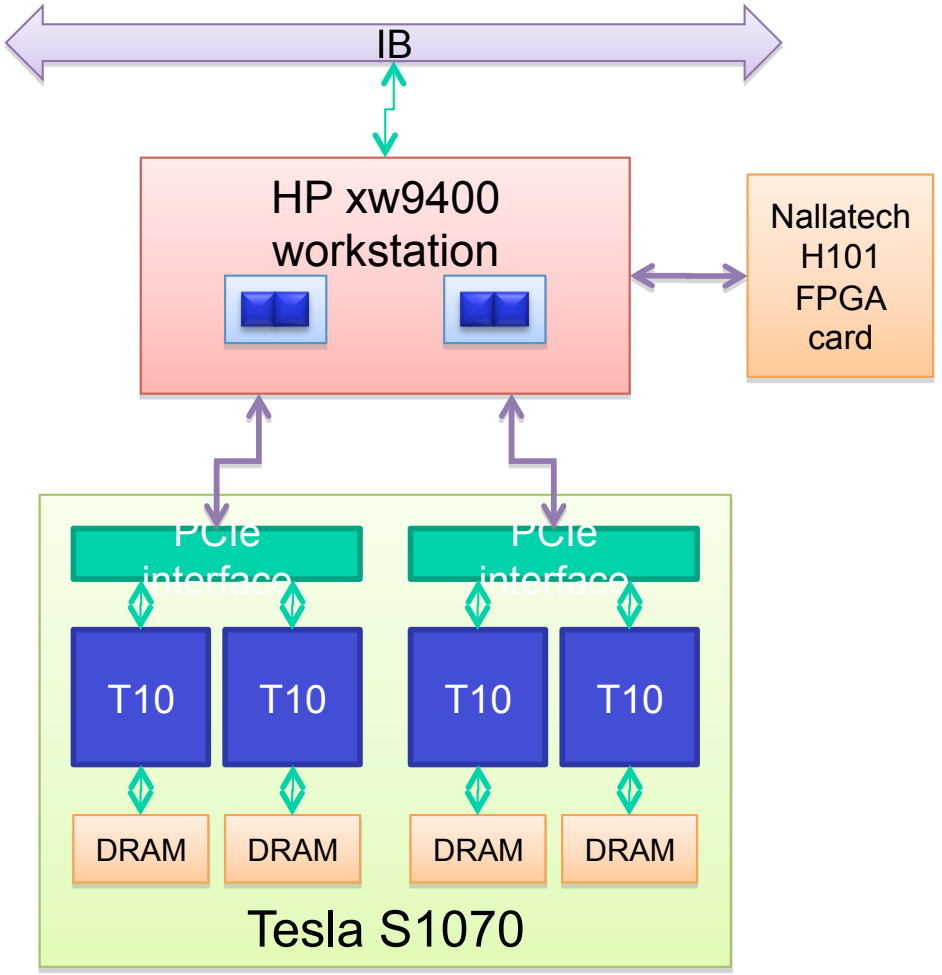
Current GPU Clusters at NCSA

- Lincoln
  - Production system available via the standard NCSA/ TeraGrid HPC allocation

- AC
  - Experimental system available for exploring GPU computing

NCSA Linux Cluster *AC*

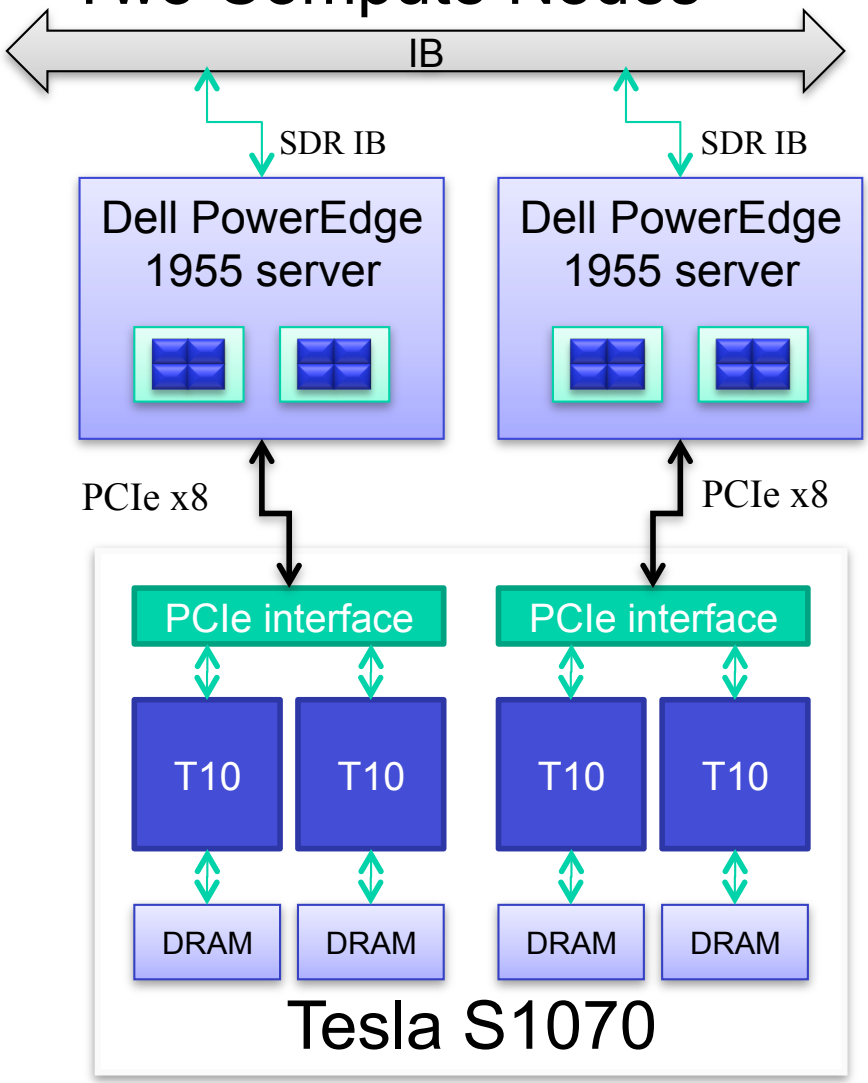
- HP xw9400 workstation
  - 2216 AMD Opteron 2.4 GHz dual socket dual core
  - 8 GB DDR2
  - Infiniband QDR
- Tesla S1070 1U 4-GPU Server
  - 1.3 GHz Tesla T10 processors
  - 4x4 GB GDDR3 SDRAM
- Cluster
  - Servers: 32
  - Accelerator Units: 32 (128 GPUS, 128 TF SP, 10 TF DP)



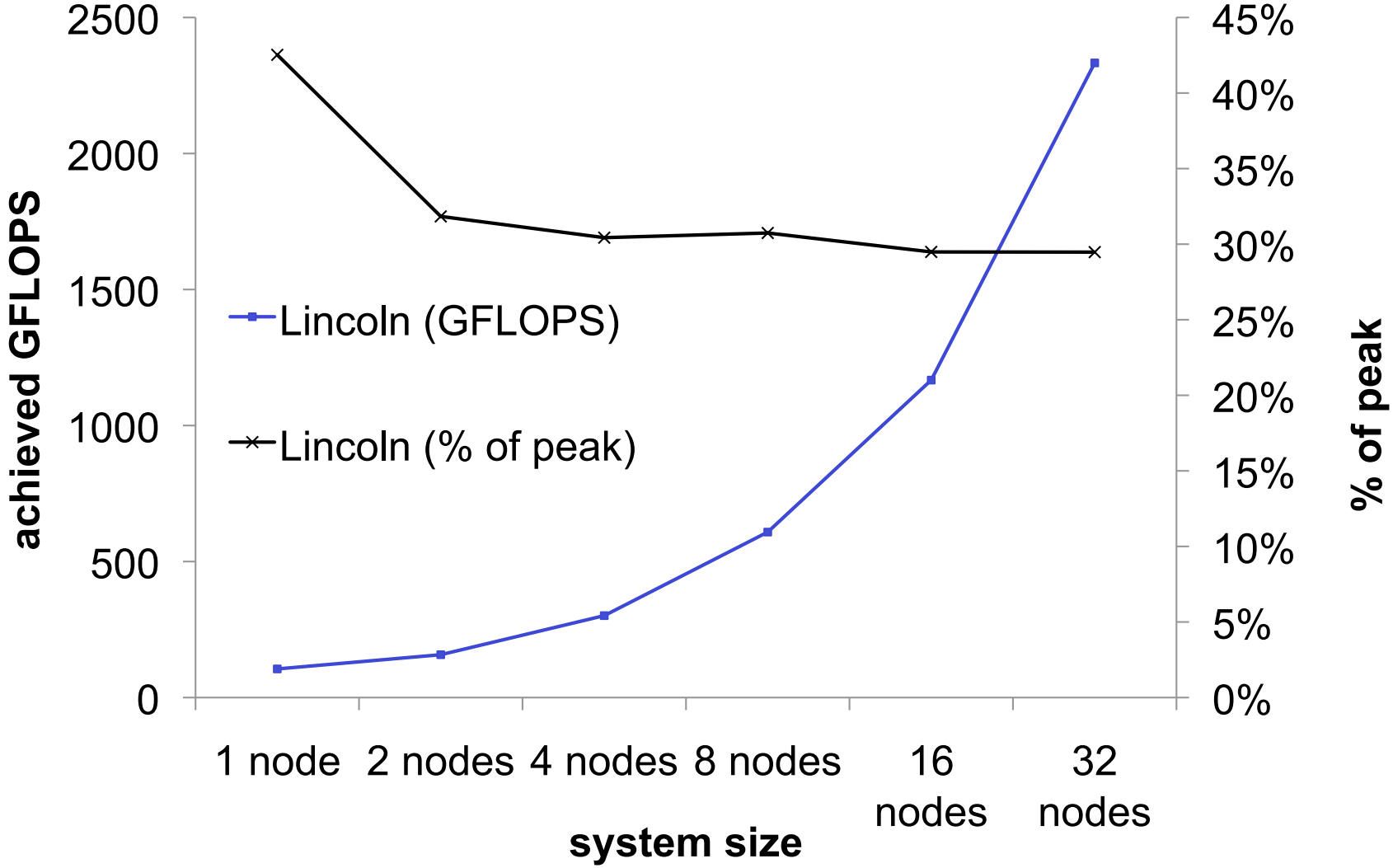
NCSA Linux Cluster *Lincoln*

- Dell PowerEdge 1955 server
  - Intel 64 (Harpertown) 2.33 GHz dual socket quad core
  - 16 GB DDR2
  - Infiniband SDR
- Tesla S1070 1U GPU Computing Server
  - 1.3 GHz Tesla T10 processors
  - 4x4 GB GDDR3 SDRAM
- Cluster
  - Servers: 192
  - Accelerator Units: 96

- Two Compute Nodes



HPL Benchmark for Lincoln



Direct Self-Consistent Field Computations on GPU Clusters

Guochun Shi,
Volodymyr Kindratenko

National Center for Supercomputing Applications

University of Illinois at Urbana-Champaign

Ivan Ufimtsev,
Todd Martinez

Department of Chemistry

Stanford University

Quantum Chemistry

Energy ($H\Psi = E\Psi$):

- Quantifies intra/intermolecular interactions
- Drives chemistry, little interesting happens on flat surface

Geometry optimization ($\nabla_R E = 0$)

- Searches for stable atomic arrangements (molecular shapes)

Molecular dynamics ($\partial^2 R / \partial t^2 = -1/M \nabla_R E$)

- The chemistry itself (at some, sometimes crude, approximation)
- Studies system at atomistic time, and length scales

Exact energy is a hard problem

$$\Psi(\mathbf{r}_i) = ?$$

$$E = ?$$

$$\left\{-\frac{1}{2}\sum_i\left(\frac{\partial^2}{\partial x_i^2}+\frac{\partial^2}{\partial y_i^2}+\frac{\partial^2}{\partial z_i^2}\right)-\sum_{i,A}\frac{Z_A}{|\mathbf{r}_i-\mathbf{R}_A|}+\sum_{i,j}\frac{1}{|\mathbf{r}_i-\mathbf{r}_j|}\right\}\Psi(\mathbf{r}_i)=E\Psi(\mathbf{r}_i)$$

$$\Psi(\ldots\mathbf{r}_i\ldots\mathbf{r}_j\ldots)=-\Psi(\ldots\mathbf{r}_j\ldots\mathbf{r}_i\ldots)$$

Hartree-Fock approximation is one of the simplest

$$\Psi = A\left[\psi_1(r_1)\psi_2(r_2)\cdots\psi_N(r_N)\right]$$

Expand ψ over predefined basis set φ

$$\psi_i(r) = \sum_{j=1}^{k} C_{ij}\varphi_j(r)$$

$$\Psi \Leftrightarrow C_{ij} = ?$$

Hartree-Fock Self Consistent Field (SCF) procedure

$$\mathbf{F}(\mathbf{C})\mathbf{C} = E\mathbf{S}\mathbf{C}$$

$$\mathbf{F}_{k+1}(\mathbf{C}) = \mathbf{F}(\mathbf{C}_k)$$

$$\mathbf{F}_{k+1}\mathbf{C}_{k+1} = E\mathbf{S}\mathbf{C}_{k+1}$$

Repeat until C_{k+1} more or less equals C_k

Hartree-Fock equations

$$\mathbf{F}(\mathbf{C})\mathbf{C} = E\mathbf{S}\mathbf{C}$$

$$F_0(\mathbf{C}) = H_0^{core} + J_0(\mathbf{C}) - \frac{1}{2}K_0(\mathbf{C})$$

$$J_0 = \sum_{kl}[ij \mid kl]P_{kl}(\mathbf{C})$$

$$K_0 = \sum_{kl}[ik \mid jl]P_{kl}(\mathbf{C})$$

$$[ij \mid kl] = \iint \varphi_i(r_1)\varphi_j(r_1)\frac{1}{|r_1 - r_2|}\varphi_k(r_2)\varphi_l(r_2)dr_1 dr_2$$

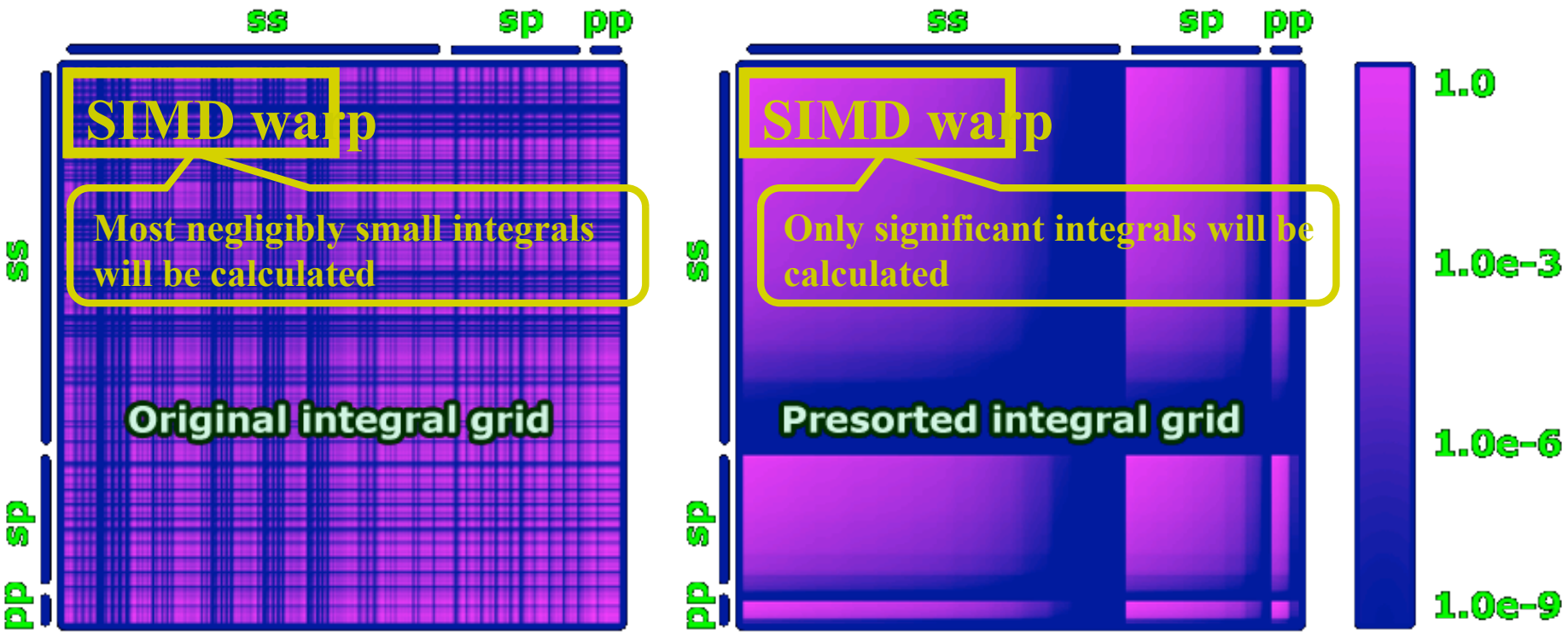
- All matrices are of $N \times N$ size ($N \sim 1{,}000 \ldots 10{,}000$)
- N^3 operations to solve HF equations (need to deal with diagonalization)
- N^4 operations to get **F**

2e integral grid

$$[ij \mid kl] = \iint \varphi_i(r_1)\varphi_j(r_1)\frac{1}{|r_1 - r_2|}\varphi_k(r_2)\varphi_l(r_2)dr_1 dr_2$$

$$[ij \mid kl] \le \sqrt{[ij \mid ij]}\sqrt{[kl \mid kl]} \le 10^{-11}$$

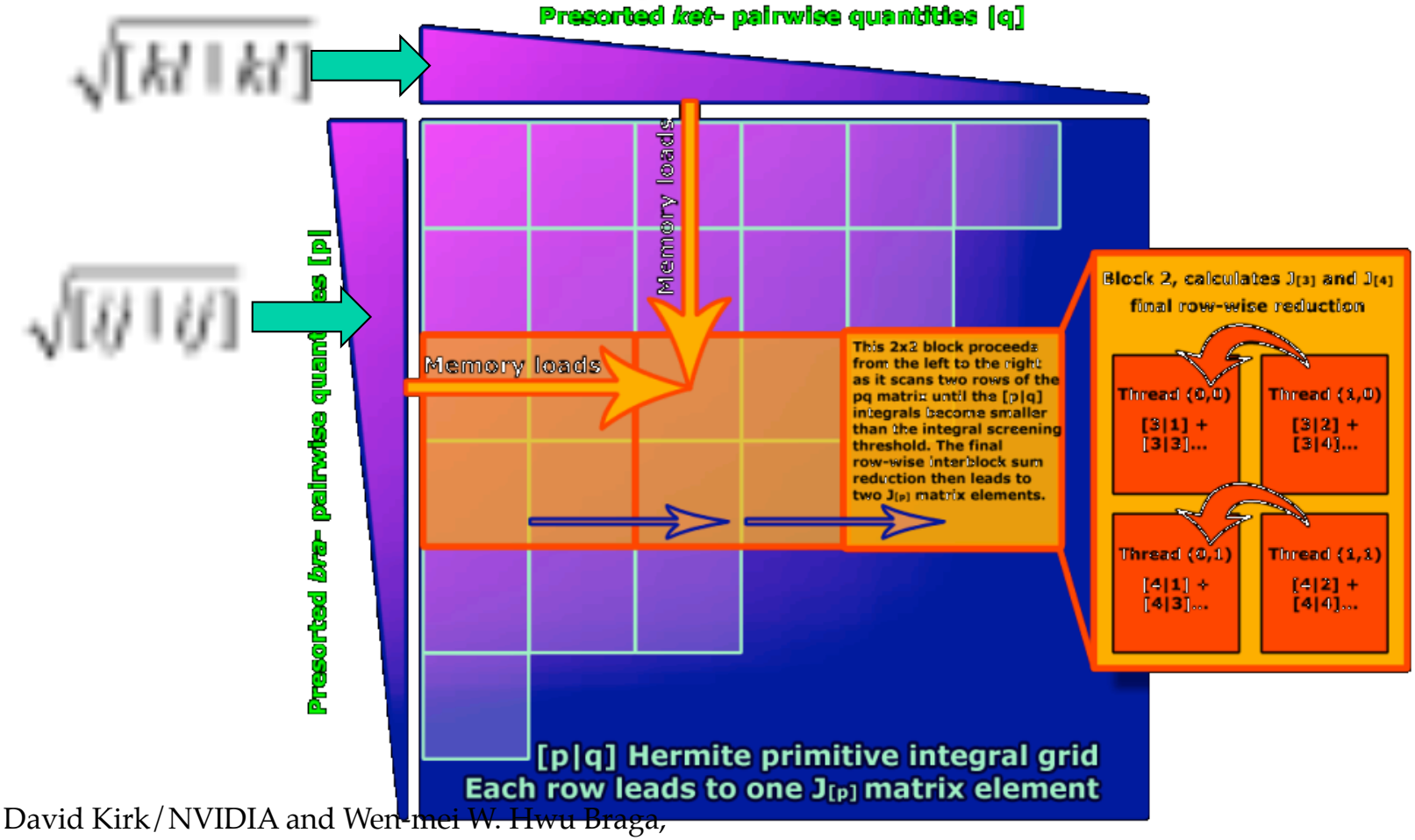
leaves only N^2 out of N^4 integrals

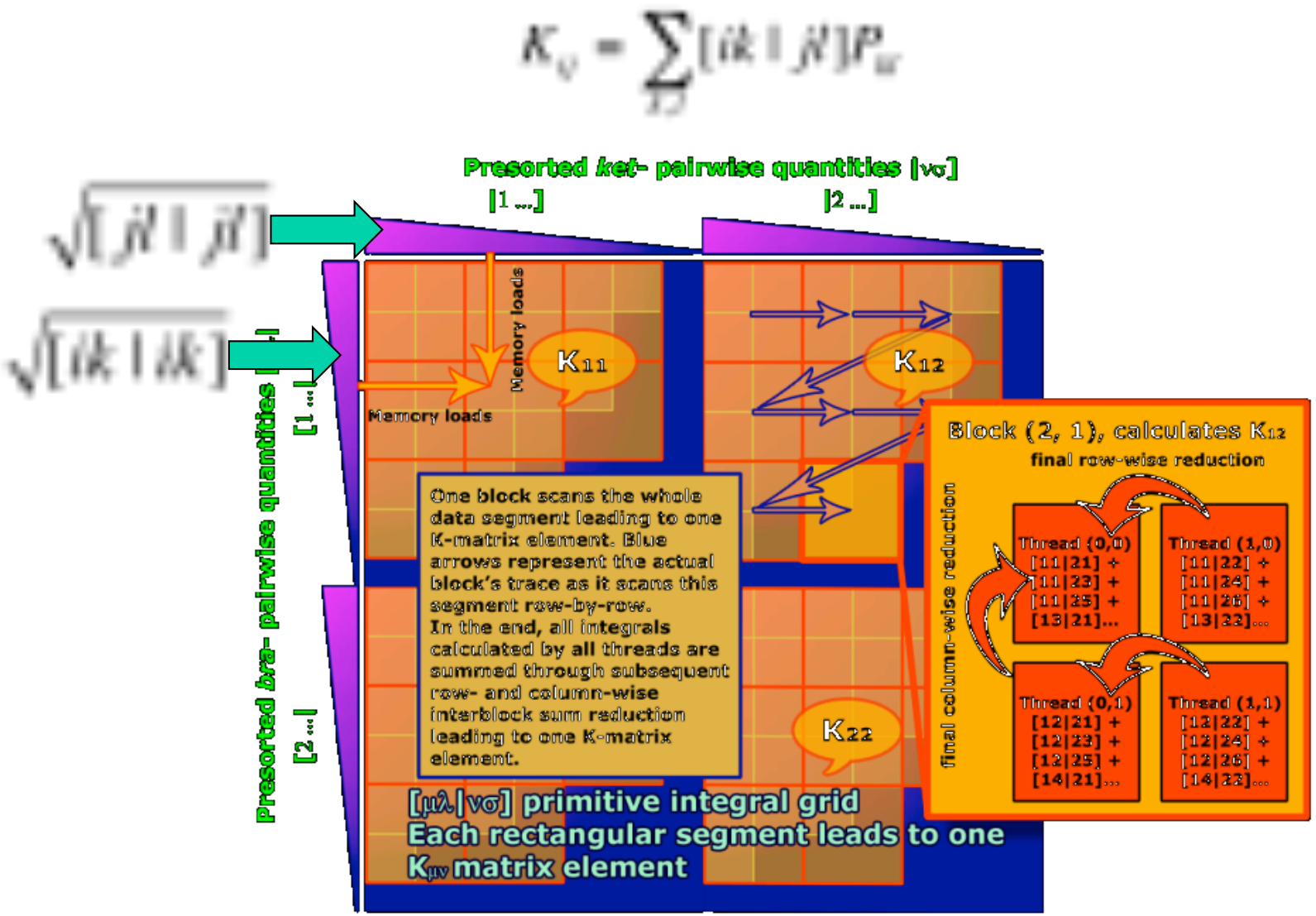


J-matrix implementation

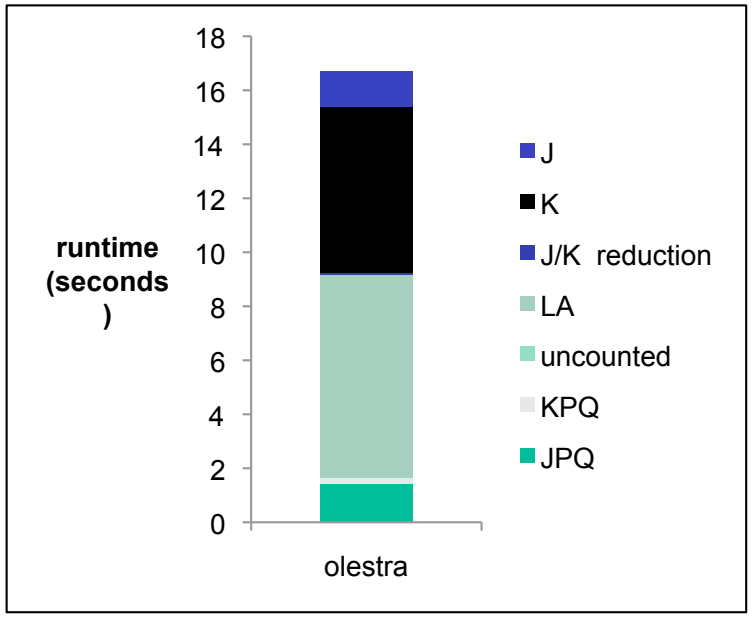
$$J_{ij} = \sum_{kl} [ij \mid kl] P_{kl}$$

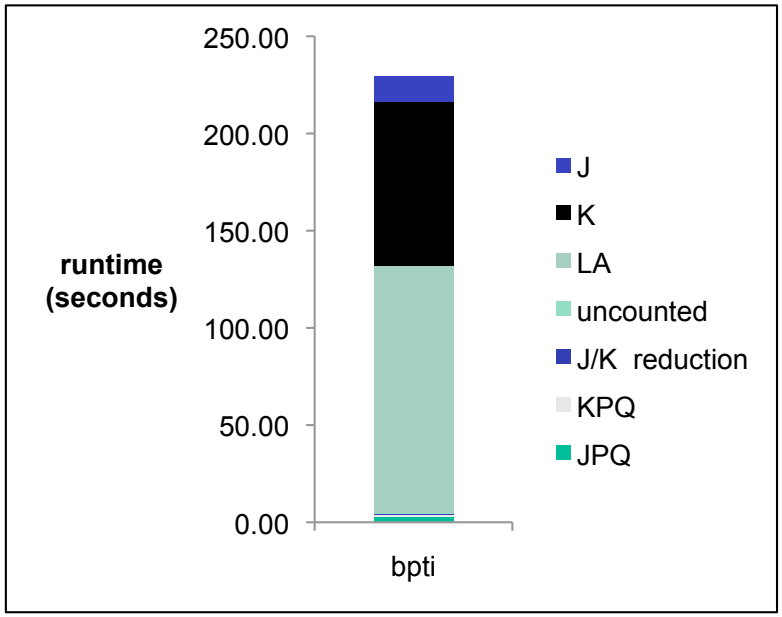
$$[ij \mid kl] \le \sqrt{[ij \mid ij]}\sqrt{[kl \mid kl]}$$





Node execution time breakdown





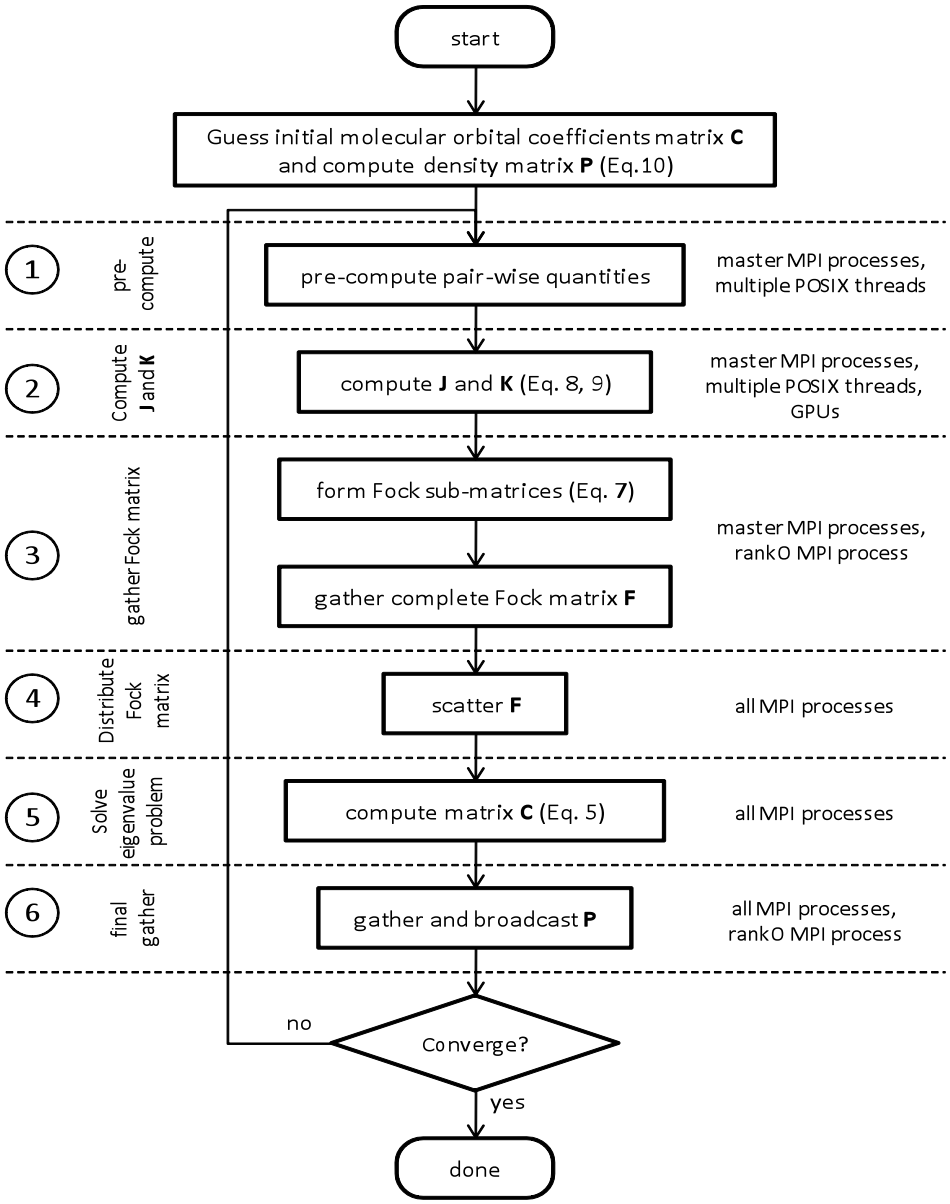
- Each node contains 8 CPU cores and 2 GPUs.
- The J and K matrices computation and Linear Algebra (LA) computation dominate the overall execution time
- Pair quantity computations can be significant

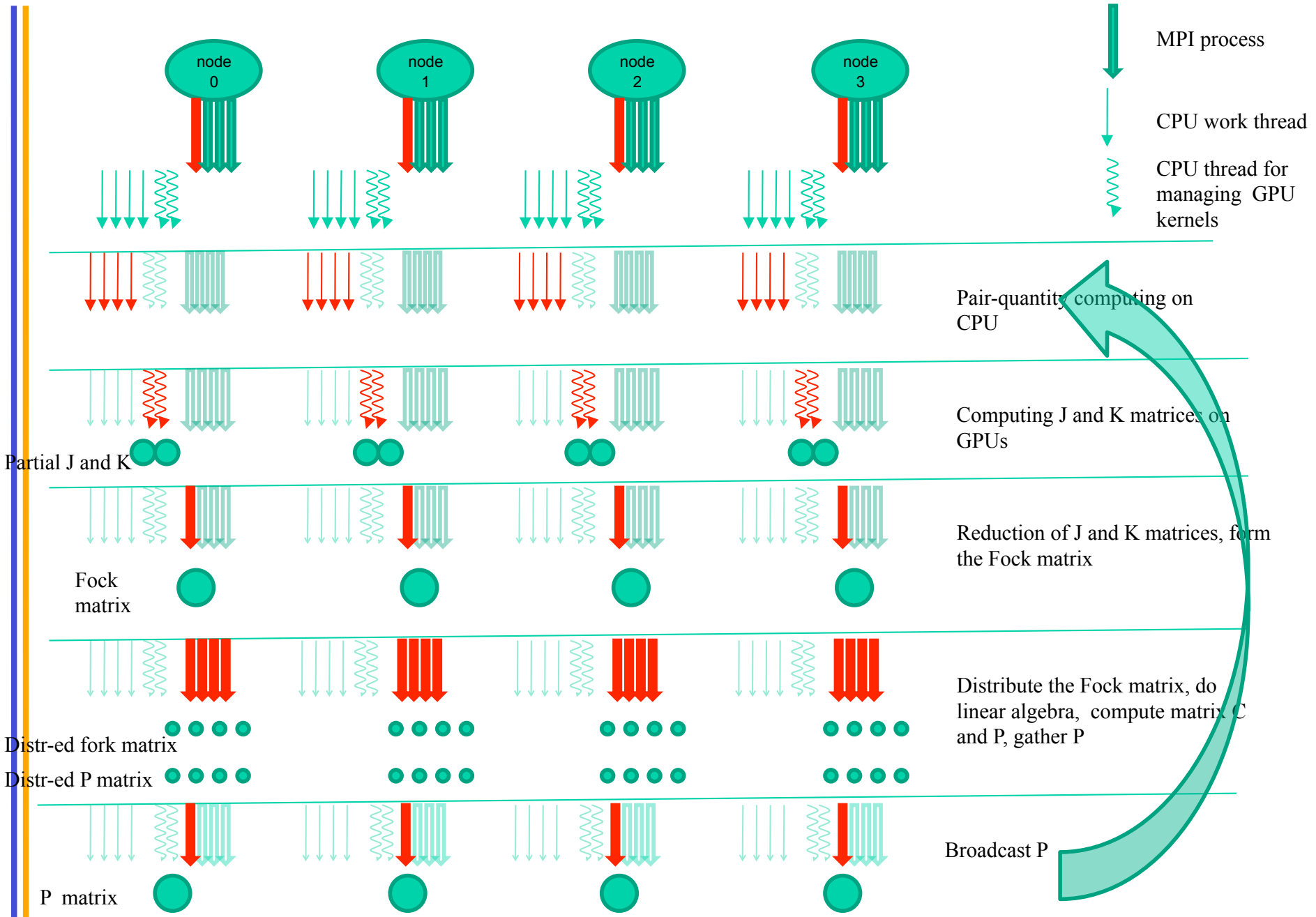
GPU cluster parallelization strategy

- Each GPU has a global id
  - nodeid * num_gpu_per_node + local_gpu_index
- J matrix work distribution (diagram)
- K matrix work distribution (diagram)
- LA using SCALAPACK

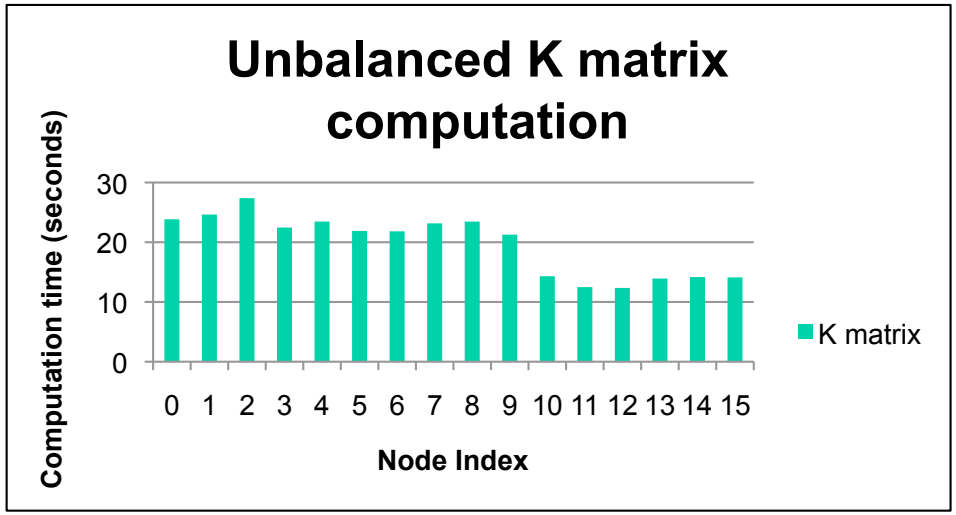
Parallelization strategy (II)

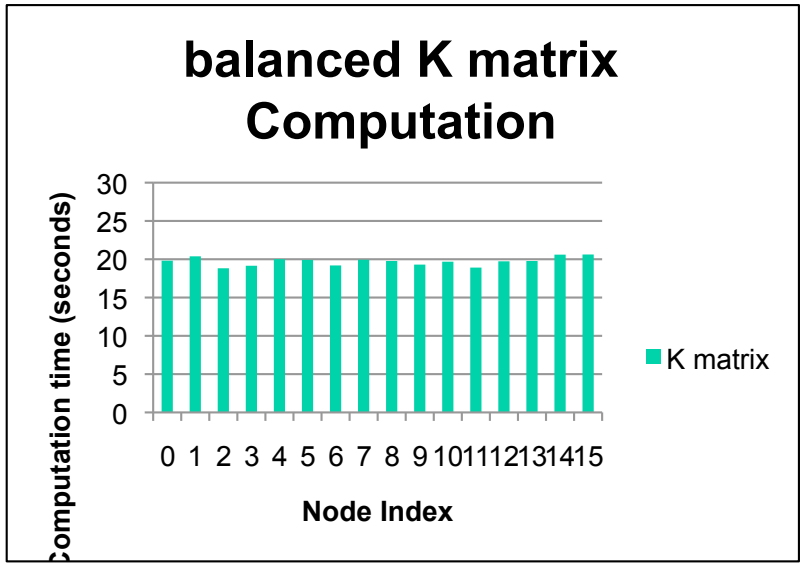
- Start as MPI program, each node has as many MPI processes as CPU cores
- One MPI process per node is designated as “master”
- The master MPI processes create threads for controlling GPUs as well as CPU work threads

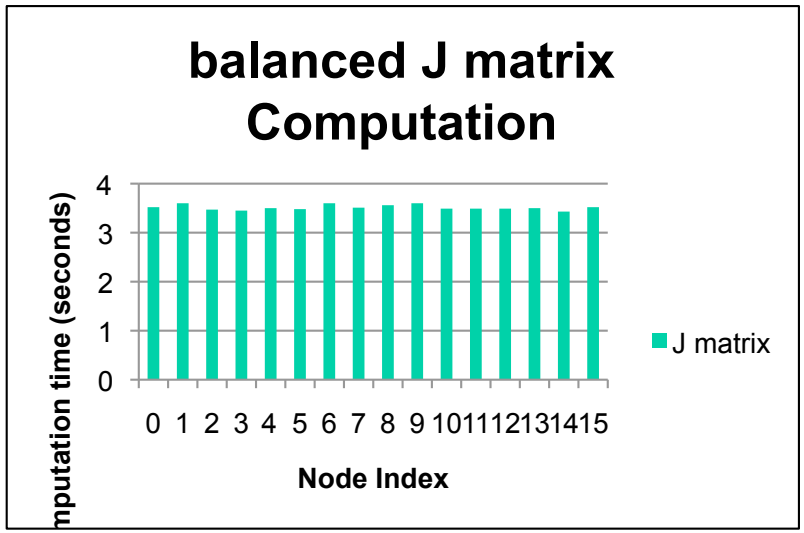




Performance: load balancing

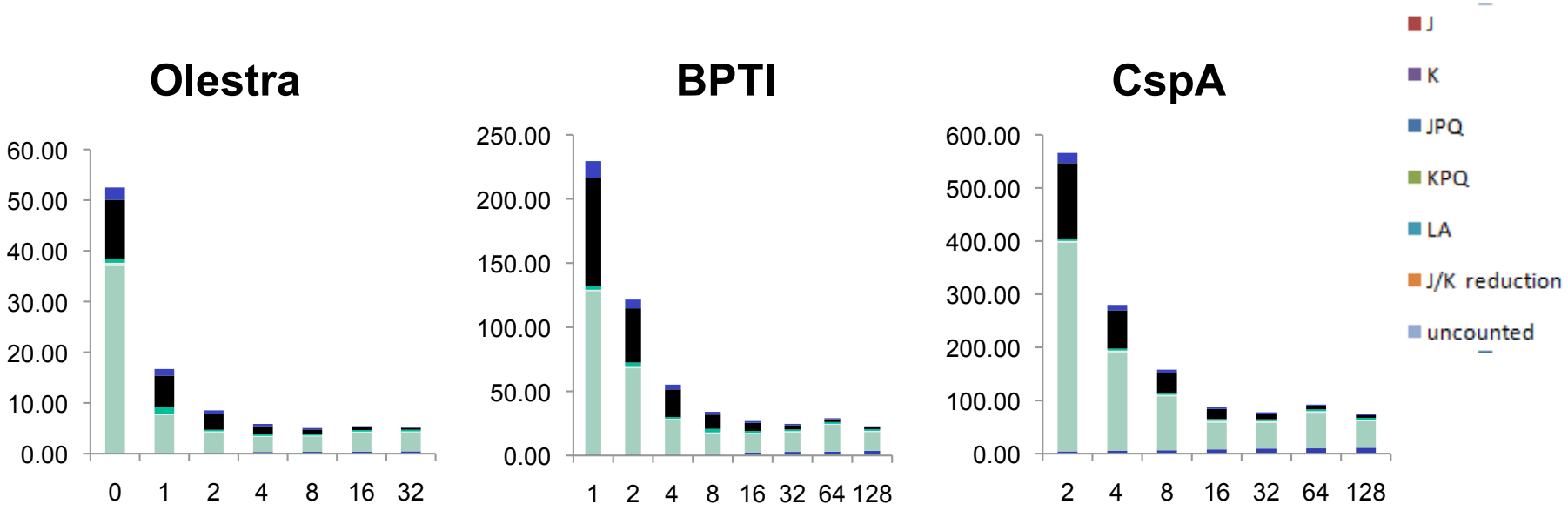




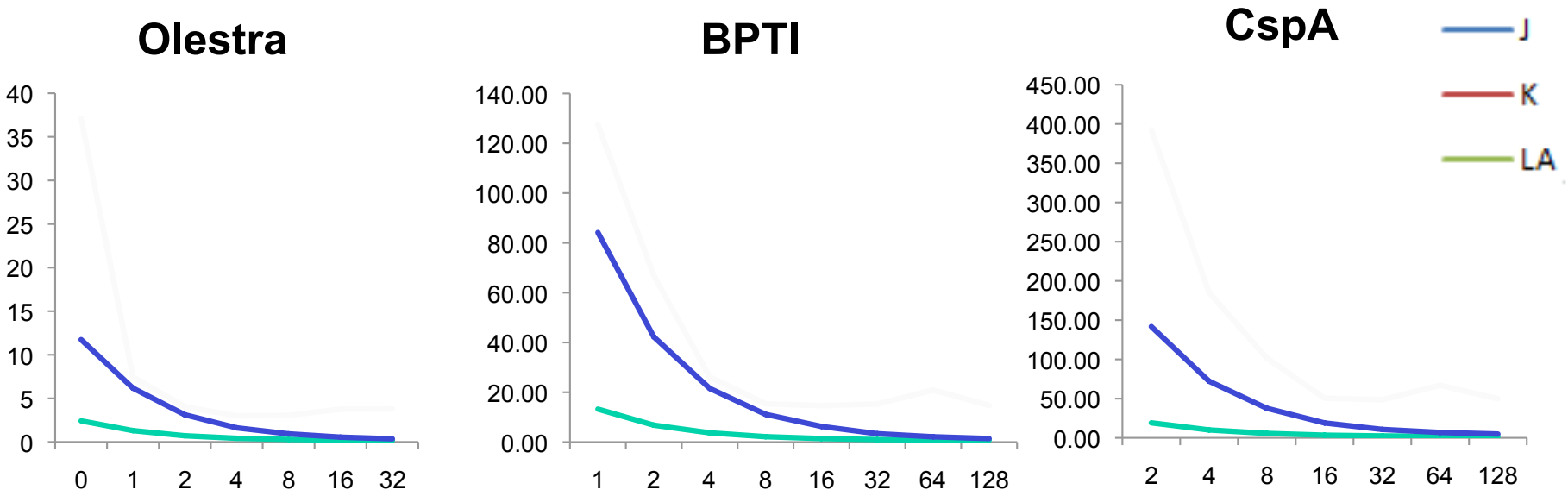


Performance

	Atoms	Electrons	Orbitals	S shells	P shells
Olestra	453	1366	2131	1081	350
BPTI	875	3400	4893	2202	897
CspA	1732	6290	8753	4220	1511

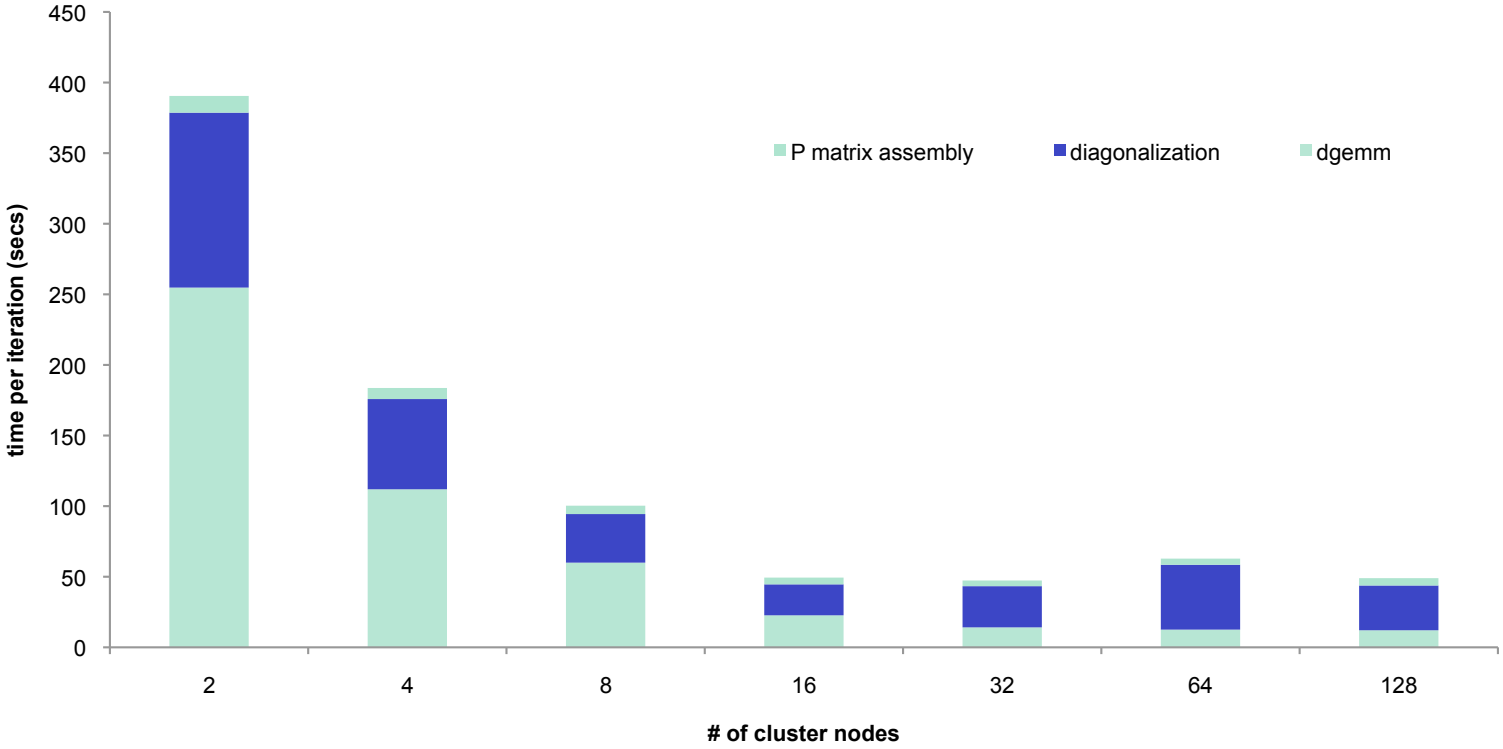


Scalabilty of J, K andi LA



- J and K matrices computation can scale well to 128 nodes
- Linear Algebra scales only up to 16 nodes even for CsPA molecule

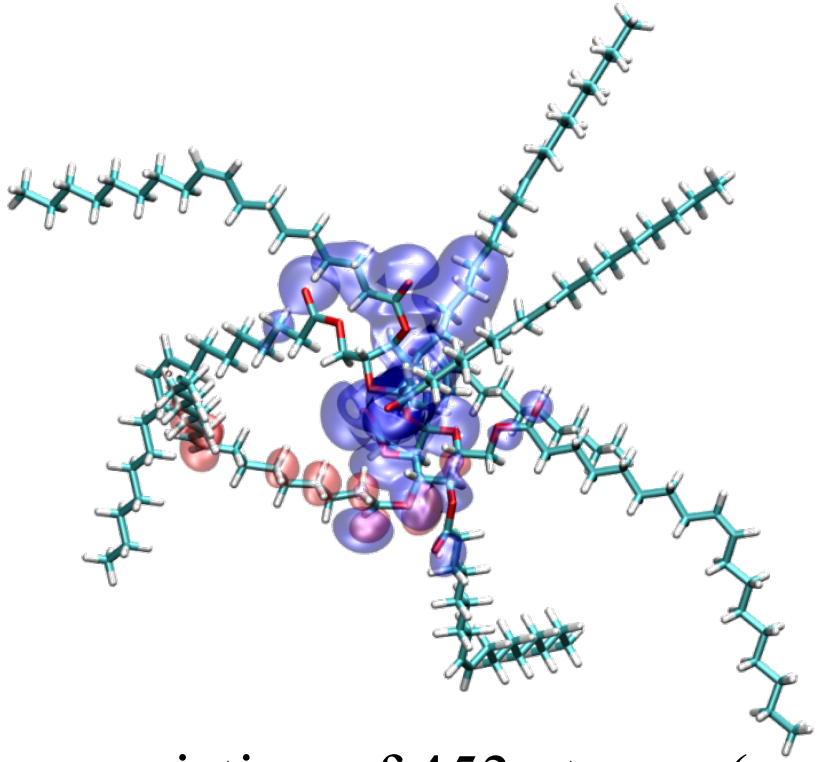
Performance: Linear Algebra breakdown



- Diagonization scales the worst, dgemm is also important

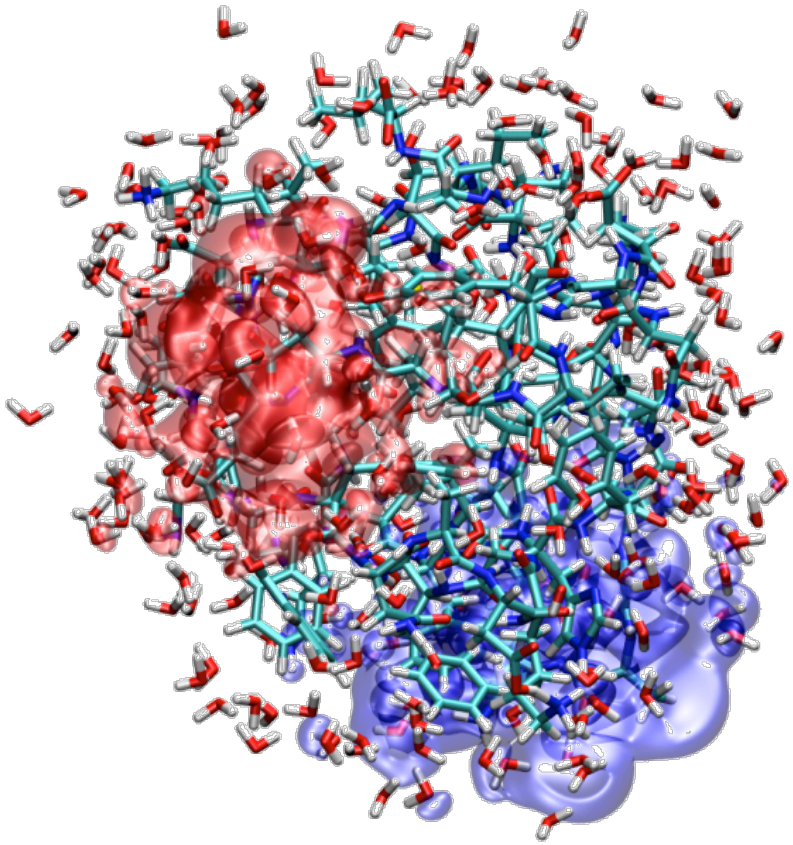
- A fast, scalable GPU based SCALAPACK is needed
  - Magma from UTK?
  - Cula?

Results: Olestra molecule



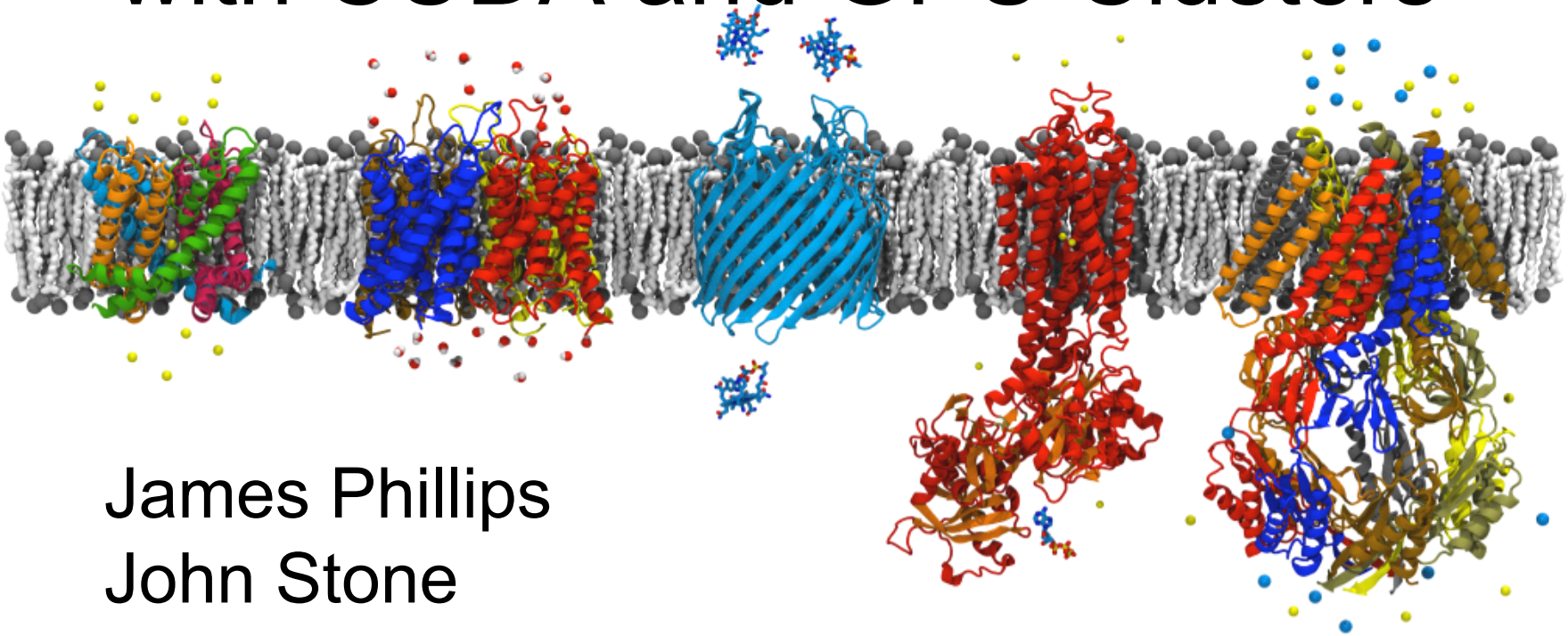
Olestra molecule consisting of 453 atoms (a small example model used of testing the developed software) can be computed by the state-of-the-art quantum chemistry software package GAMESS running on an Intel Pentium D 3 GHz processor in over 12,408 seconds whereas our 8-node GPU cluster implementation performs the same computation in just over 5 seconds, a 2,452× speedup.

Example: CspA molecule



For larger models, one SCF iteration for Cold shock protein A (CspA) molecule consisting of 1,732 atoms can be done in 88 seconds on a 16 node GPU cluster.

Accelerating Biomolecular Modeling with CUDA and GPU Clusters

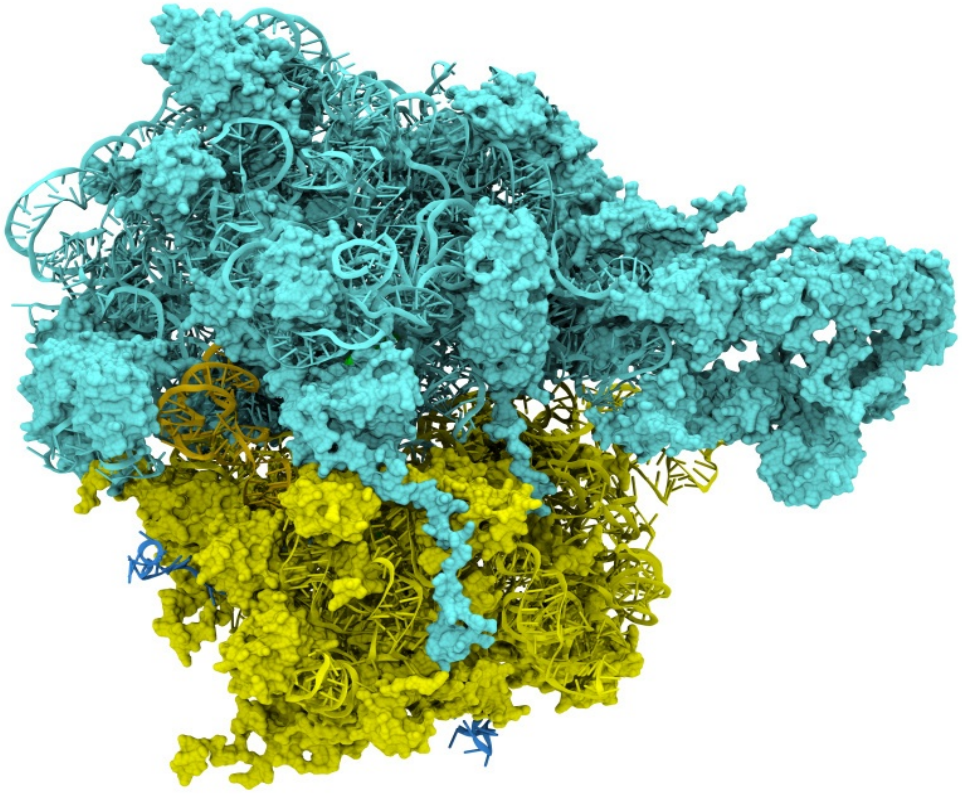


James Phillips
John Stone
Klaus Schulten

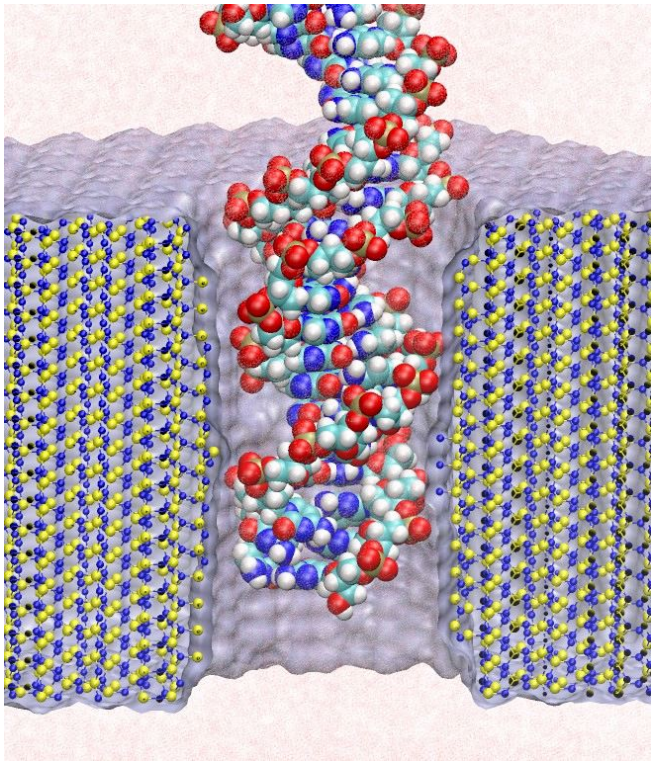
http://www.ks.uiuc.edu/Research/gpu/

Computational Microscopy

Ribosome: synthesizes proteins from genetic information, target for antibiotics



Silicon nanopore: bionanodevice for sequencing DNA efficiently



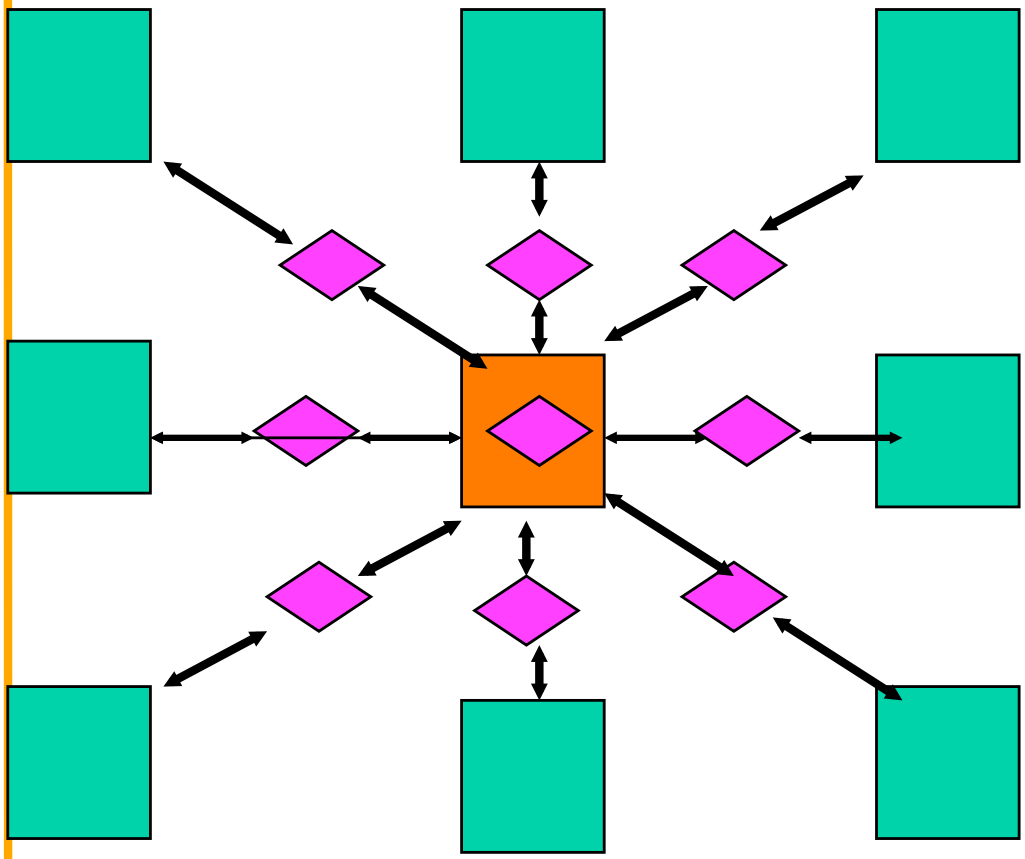
NAMD: Practical Supercomputing

- 35,000 users can't all be computer expe
  - 18% are NIH-funded; many in other countries
  - 8200 have downloaded more than one versio
- User experience is the same on all platforms.
  - No change in input, output, or configuration files.
  - Run any simulation on **any number of processors**
  - Precompiled binaries available when possible.
- Desktops and laptops – setup and testing
  - x86 and x86-64 Windows, and Macintosh
  - Allow both shared-memory and network-based parallelis
- Linux clusters – affordable workhorses
  - x86, x86-64, and Itanium processors
  - Gigabit ethernet, Myrinet, InfiniBand, Quadrics, Altix, etc

Our Goal: Practical Acceleration

- Broadly applicable to scientific computing
  - Programmable by domain scientists
  - Scalable from small to large machines
- Broadly available to researchers
  - Price driven by commodity market
  - Low burden on system administration
- Sustainable performance advantage
  - Performance driven by Moore's law
  - Stable market and supply chain

NAMD Hybrid Decomposition



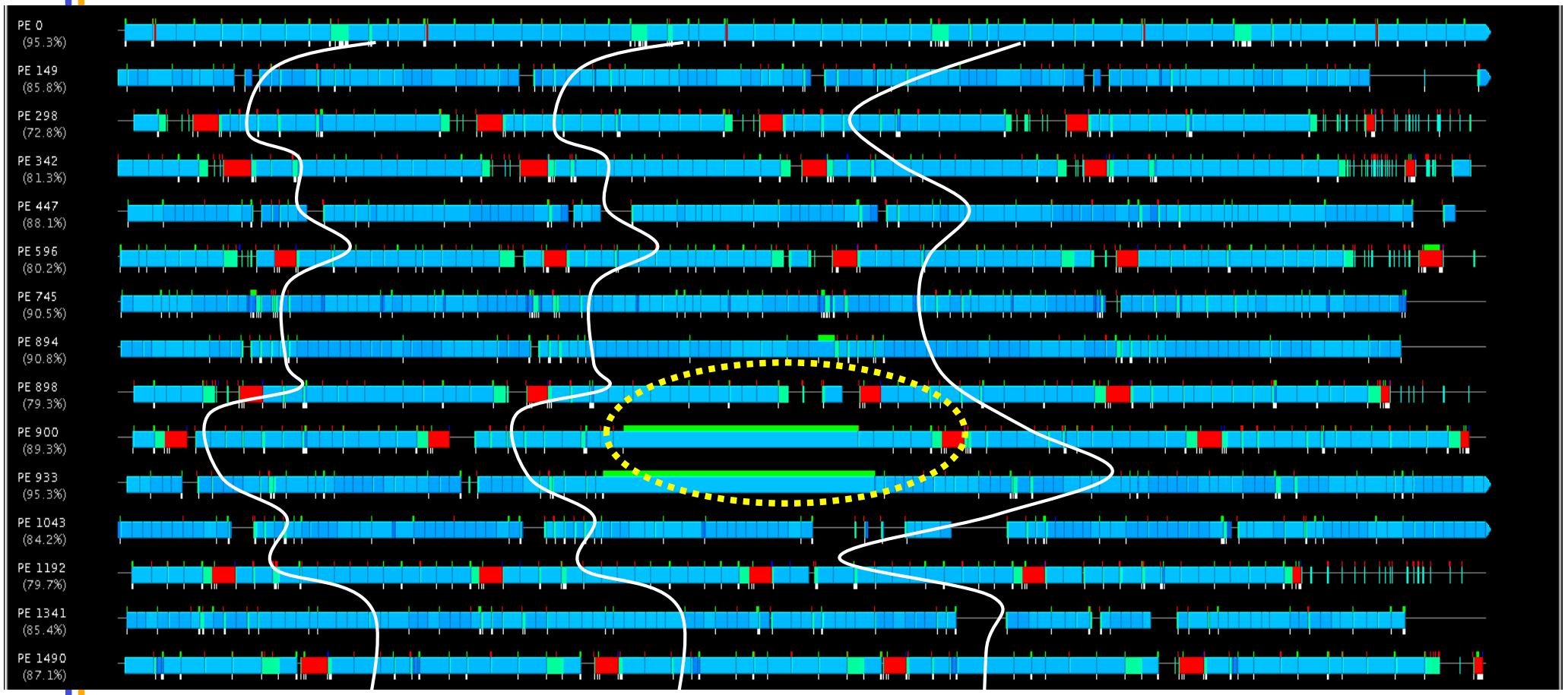
- Spatially decompose data and communication.
- Separate but related work decomposition.
- “Compute objects” facilitate iterative, measurement-based load balancing system.

NAMD Code is Message-Driven

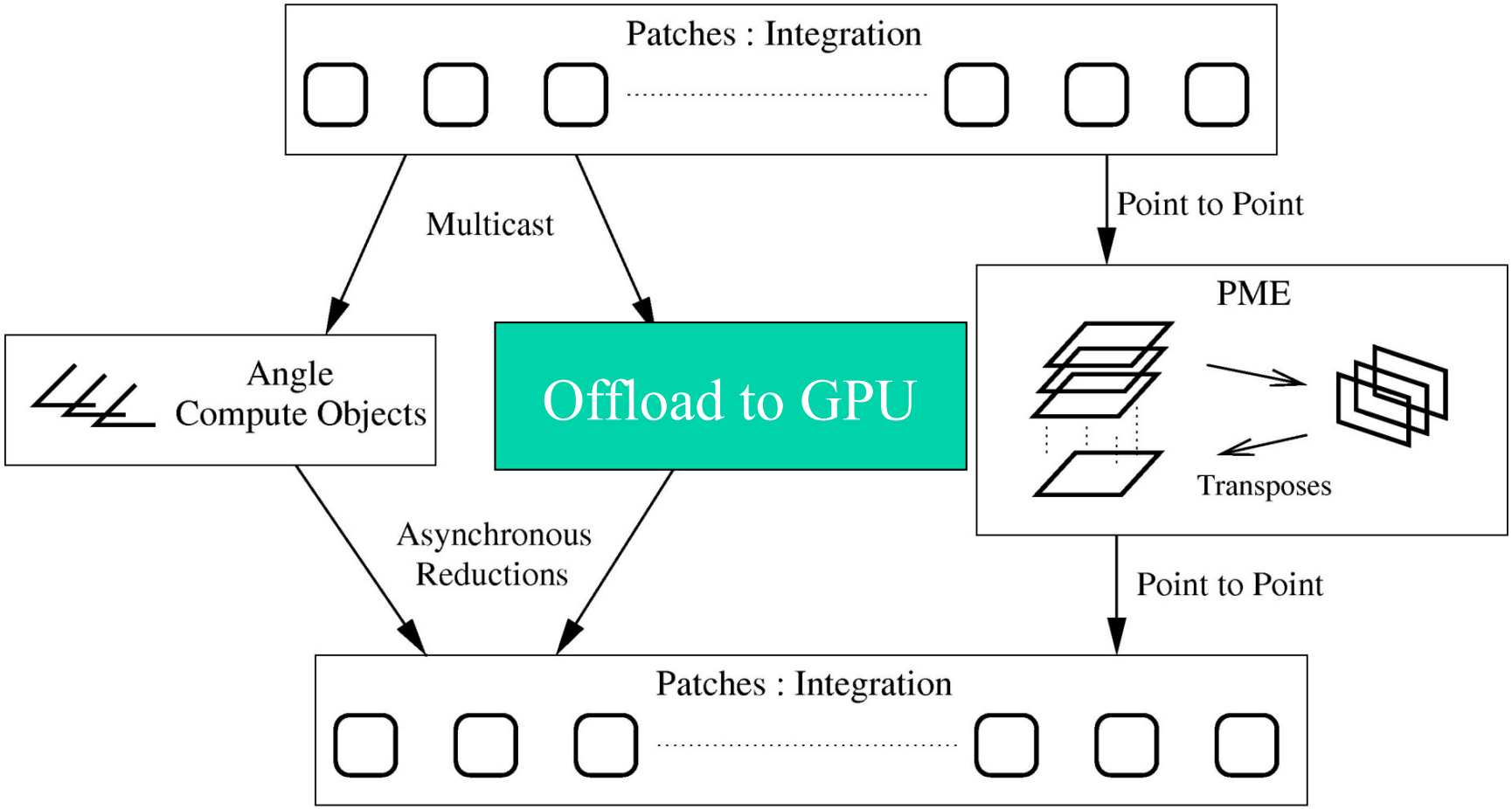
- No receive calls as in “message passing”
- Messages sent to object “entry points”
- Incoming messages placed in queue
  - Priorities are necessary for performance
- Execution generates new messages
- Implemented in Charm++ on top of MPI
  - Can be emulated in MPI alone
  - Charm++ provides tools and idioms
  - Parallel Programming Lab: http://charm.cs.uiuc.edu/

System Noise Example

Timeline from Charm++ tool “Projections” http://charm.cs.uiuc.edu/



NAMD Overlapping Execution



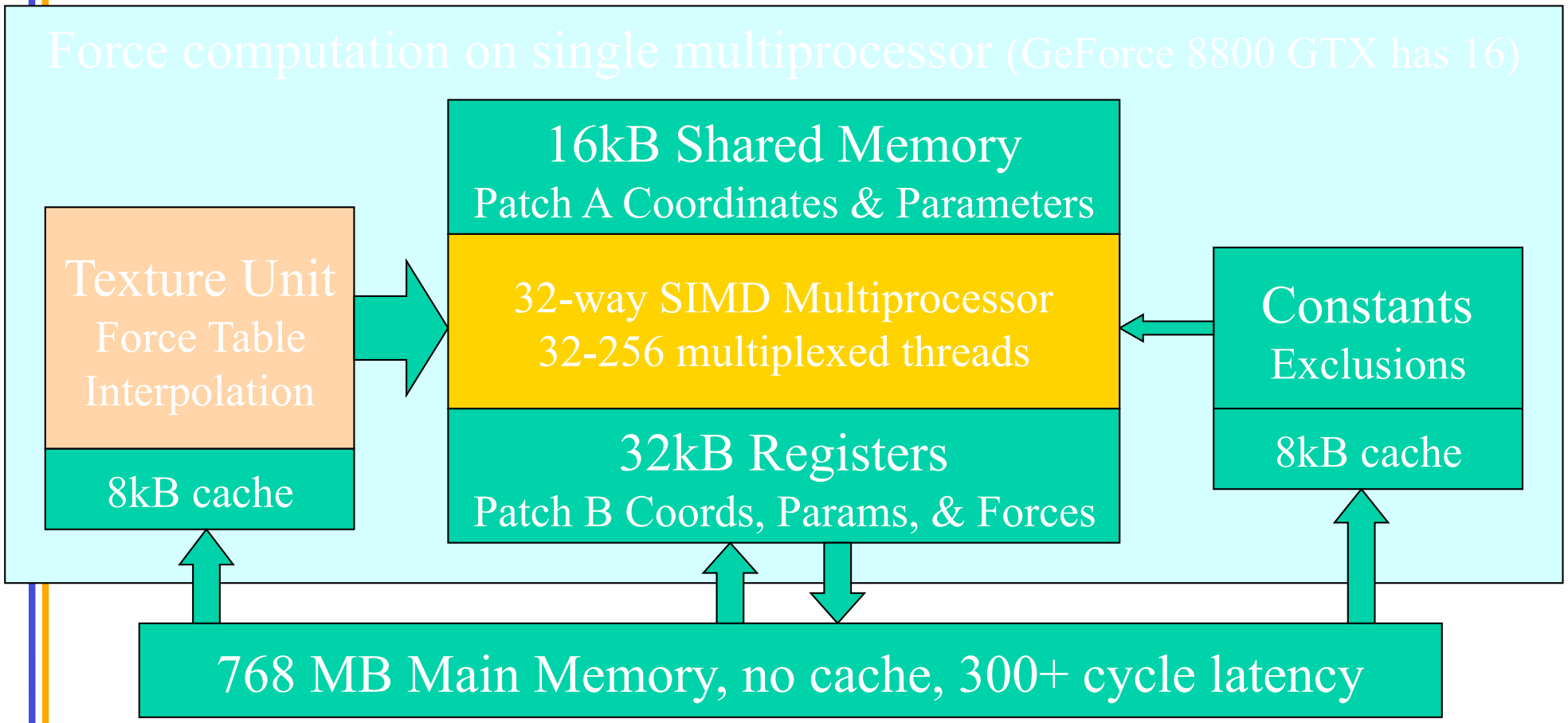
Objects are assigned to processors and queued as data arrives.

MPI Message-Driven CUDA Kernels?

- No, CUDA Kernels are too coarse-grained.
  - CPU needs fine-grained work to interleave and pipeline.
  - GPU needs large numbers of tasks submitted all at once.
- No, CUDA lacks priorities.
  - FIFO isn't enough.
- Perhaps in a future interface:
  - Stream data to GPU.
  - Append blocks to a running kernel invocation.
  - Stream data out as blocks complete.
- Fermi looks very promising!

Nonbonded Forces on CUDA GPU

- Start with most expensive calculation: direct nonbonded interactions.
- Decompose work into pairs of patches, identical to NAMD structure.
- GPU hardware assigns patch-pairs to multiprocessors dynamically.



```
texture<float4> force_table;
__constant__ unsigned int exclusions[];
__shared__ atom jatom[];
atom iatom;      // per-thread atom, stored in registers
float4 iforce;   // per-thread force, stored in registers
for ( int j = 0; j < jatom_count; ++j ) {
  float dx = jatom[j].x - iatom.x;   float dy = jatom[j].y - iatom.y;  float dz = jatom[j].z - iatom.z;
  float r2 = dx*dx + dy*dy + dz*dz;
  if ( r2 < cutoff2 ) {
    float4 ft = texfetch(force_table, 1.f/sqrt(r2));
    bool excluded = false;
    int indexdiff = iatom.index - jatom[j].index;
    if ( abs(indexdiff) <= (int) jatom[j].excl_maxdiff ) {
      indexdiff += jatom[j].excl_index;
      excluded = ((exclusions[indexdiff>>5] & (1<<(indexdiff&31))) != 0);
    }
    float f = iatom.half_sigma + jatom[j].half_sigma;  // sigma
    f *= f*f;  // sigma^3
    f *= f;  // sigma^6
    f *= ( f * ft.x + ft.y );  // sigma^12 * fi.x - sigma^6 * fi.y
    f *= iatom.sqrt_epsilon * jatom[j].sqrt_epsilon;
    float qq = iatom.charge * jatom[j].charge;
    if ( excluded ) { f = qq * ft.w; }  // PME correction
    else { f += qq * ft.z; }  // Coulomb
    iforce.x += dx * f;   iforce.y += dy * f;    iforce.z += dz * f;
    iforce.w += 1.f;  // interaction count or energy
  }
}
```

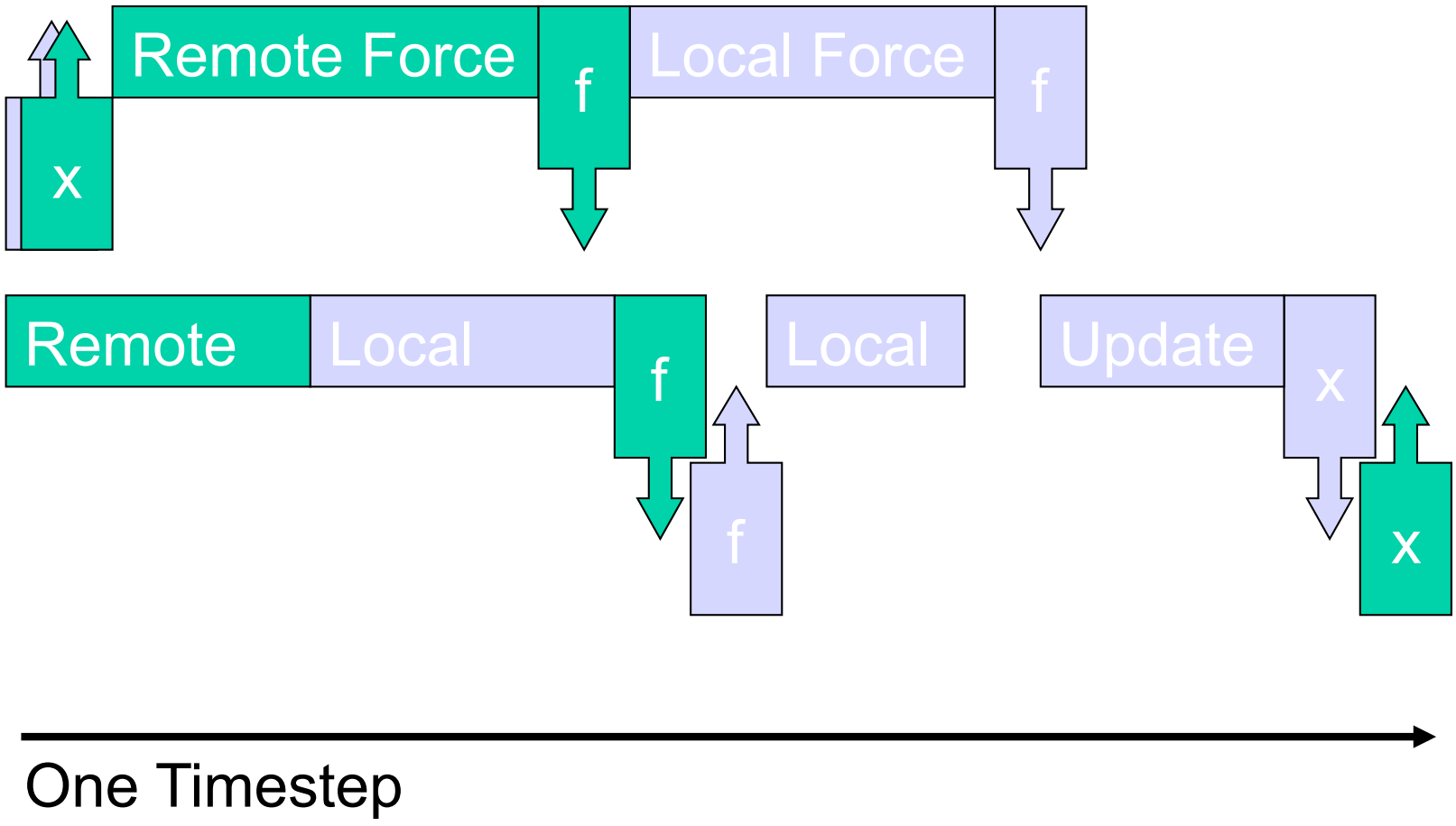
Force Interpolation

Exclusions

Parameters

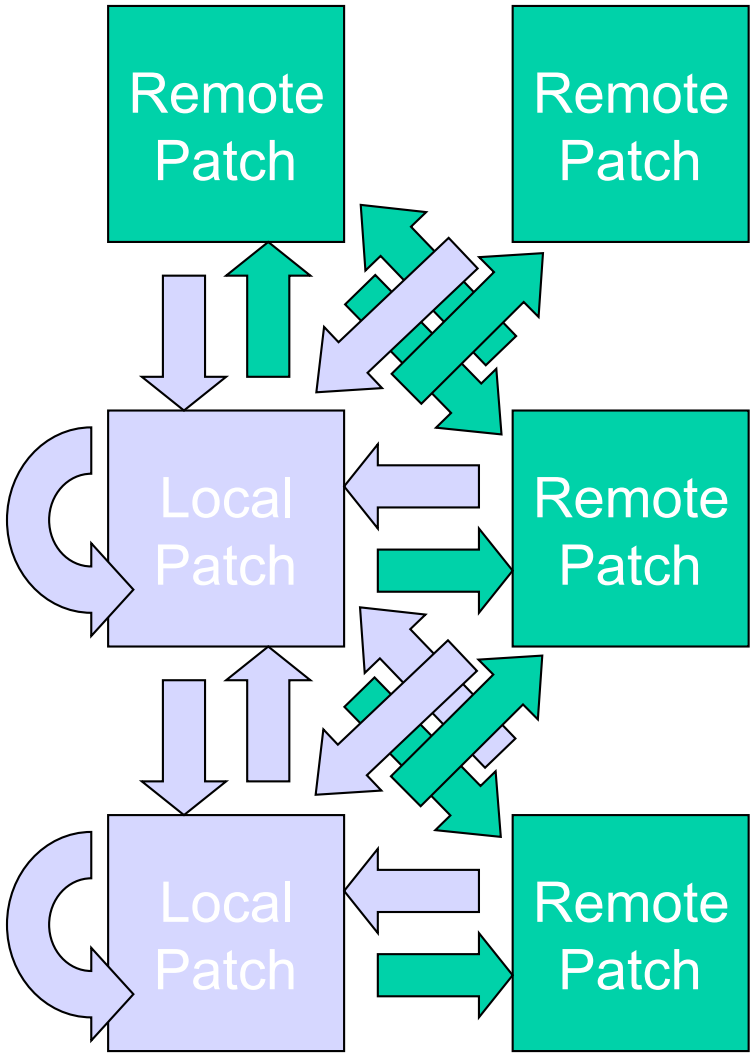
Accumulation

Overlapping GPU and CPU with Communication



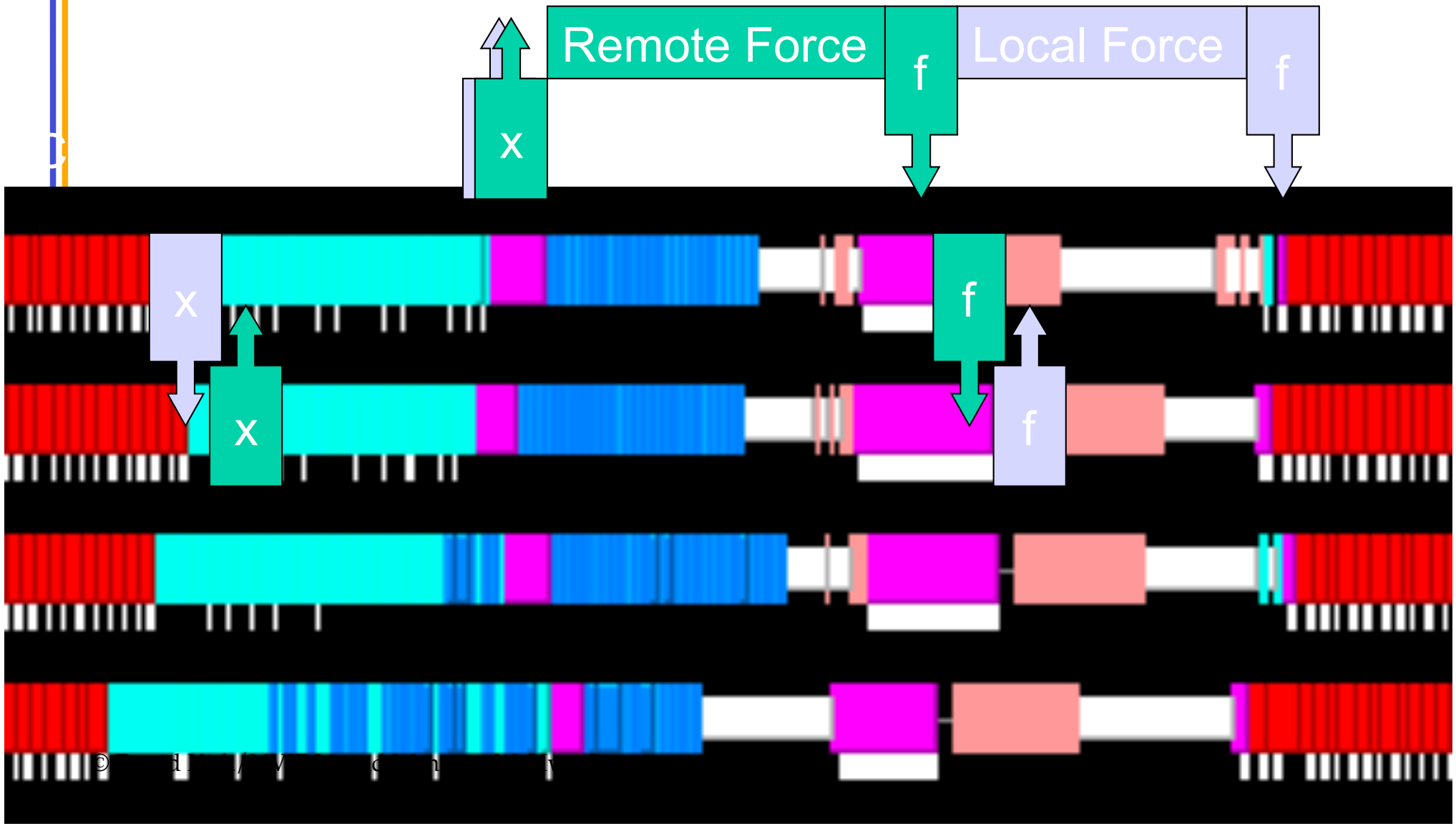
“Remote Forces”

- Forces on atoms in a local patch are “local”
- Forces on atoms in a remote patch are “remote”
- Calculate remote forces first to overlap force communication with local force calculation
- Not enough work to overlap with position communication

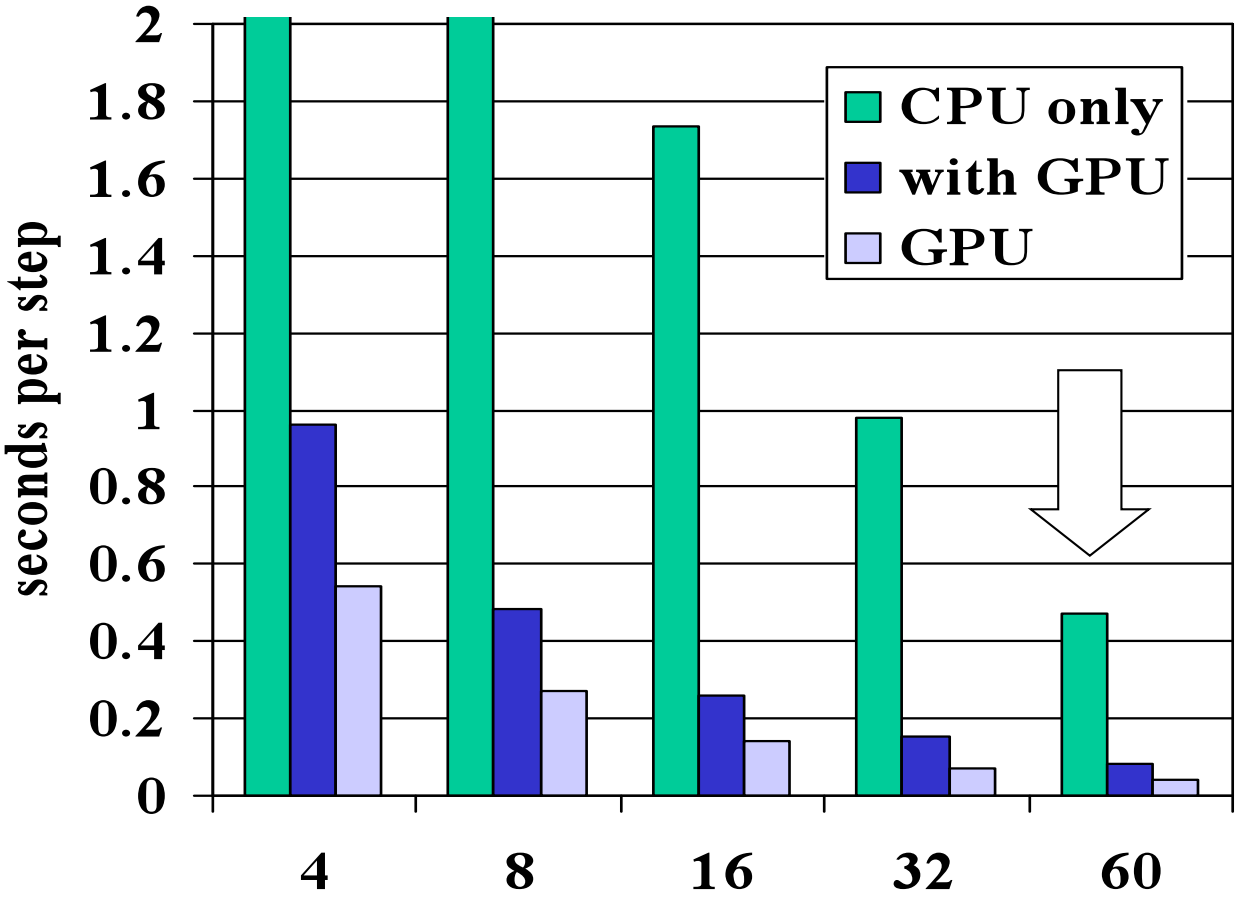


Actual Timelines from NAMD

Generated using Charm++ tool “Projections” http://charm.cs.uiuc.edu/



NCSA “4+4” QuadroPlex Cluster



CUDA/OpenCL Wrapper Library

- Basic operation principle:
  - Use /etc/ld.so.preload to overload (intercept) a subset of CUDA/OpenCL functions, e.g. {cu|cuda}{Get|Set}Device, clGetDeviceIDs, etc
- Purpose:
  - Enables controlled GPU device visibility and access, extending resource allocation to the workload manager
  - Prove or disprove feature usefulness, with the hope of eventual uptake or reimplementation of proven features by the vendor
  - Provides a platform for rapid implementation and testing of HPC relevant features not available in NVIDIA APIs
- Features:
  - NUMA Affinity mapping
    - Sets thread affinity to CPU core nearest the gpu device
  - Shared host, multi-gpu device fencing
    - Only GPUs allocated by scheduler are visible or accessible to user
    - GPU device numbers are virtualized, with a fixed mapping to a physical device per user environment
    - User always sees allocated GPU devices indexed from 0

CUDA/OpenCL Wrapper Library

- Features (cont'd):
  - Device Rotation (deprecated)
    - Virtual to Physical device mapping rotated for each process accessing a GPU device
    - Allowed for common execution parameters (e.g. Target gpu0 with 4 processes, each one gets separate gpu, assuming 4 gpus available)
    - CUDA 2.2 introduced compute-exclusive device mode, which includes fallback to next device. Device rotation feature may no longer needed
  - Memory Scrubber
    - Independent utility from wrapper, but packaged with it
    - Linux kernel does no management of GPU device memory
    - Must run between user jobs to ensure security between users
- Availability
  - NCSA/UofI Open Source License
  - https://sourceforge.net/projects/cudawrapper/

CUDA Memtest

- 4GB of Tesla GPU memory is not ECC protected
- Hunt for “soft error”
- Features
  - Full re-implementation of every test included in memtest86
  - Random and fixed test patterns, error reports, error addresses, test specification
  - Email notification
  - Includes additional stress test for software and hardware errors
- Usage scenarios
  - Hardware test for defective GPU memory chips
  - CUDA API/driver software bugs detection
  - Hardware test for detecting soft errors due to non-ECC memory
- No soft error detected in 2 years x 4 gig of cumulative runtime
- Availability
  - NCSA/UofI Open Source License
  - https://sourceforge.net/projects/cudagpumemtest/

GPU Node Pre/Post Allocation Sequence

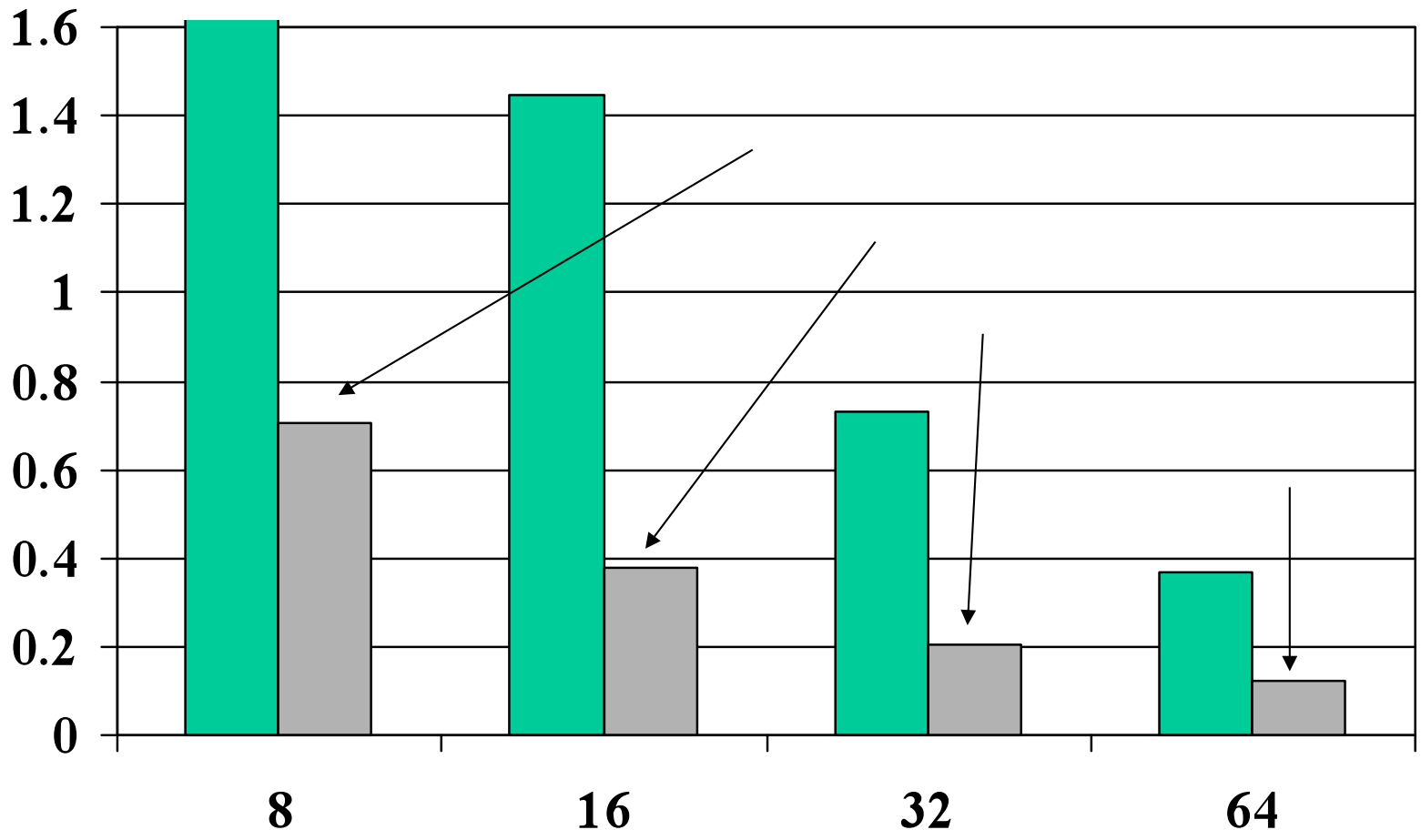
- Pre-Job (minimized for rapid device acquisition)
  - Assemble detected device file unless it exists
  - Sanity check results
  - Checkout requested GPU devices from that file
  - Initialize CUDA wrapper shared memory segment with unique key for user (allows user to ssh to node outside of job environment and have same gpu devices visible)
- Post-Job
  - Use quick memtest run to verify healthy GPU state
  - If bad state detected, mark node offline if other jobs present on node
  - If no other jobs, reload kernel module to “heal” node (for CUDA 2.2 driver bug)
  - Run memscrubber utility to clear gpu device memory
  - Notify of any failure events with job details
  - Terminate wrapper shared memory segment
  - Check-in GPUs back to global file of detected devices

NCSA “8+2” Lincoln Cluster

- How to share a GPU among 4 CPU cores?
  - Send all GPU work to one process?
  - Coordinate via messages to avoid conflict?
  - Or just hope for the best?

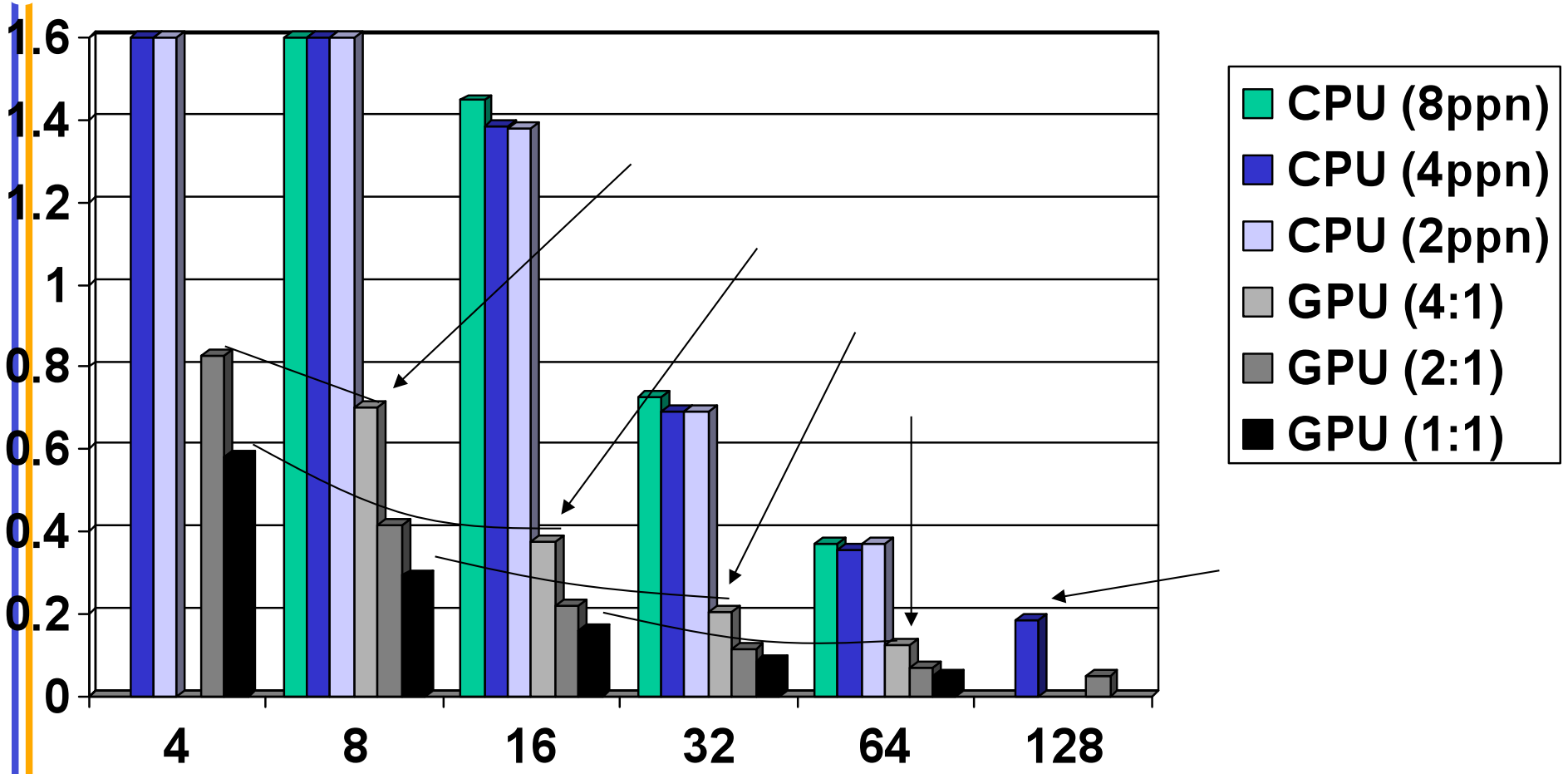
NCSA Lincoln Cluster Performance

(8 Intel cores and 2 NVIDIA Telsa GPUs per node)

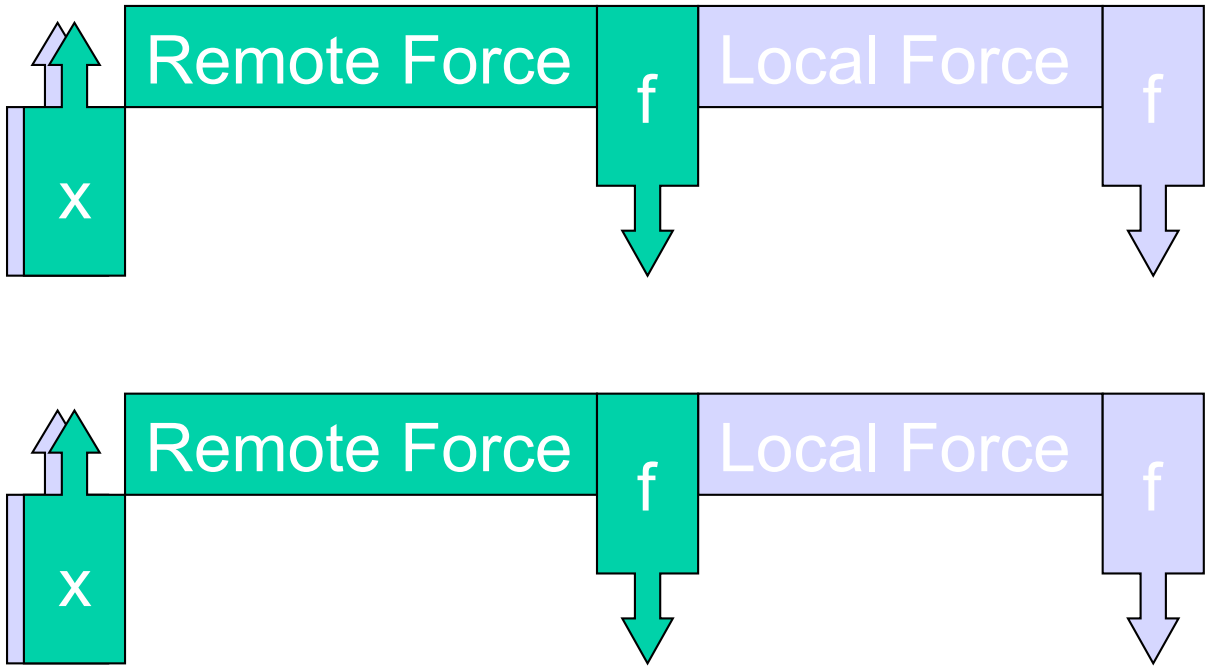


NCSA Lincoln Cluster Performance

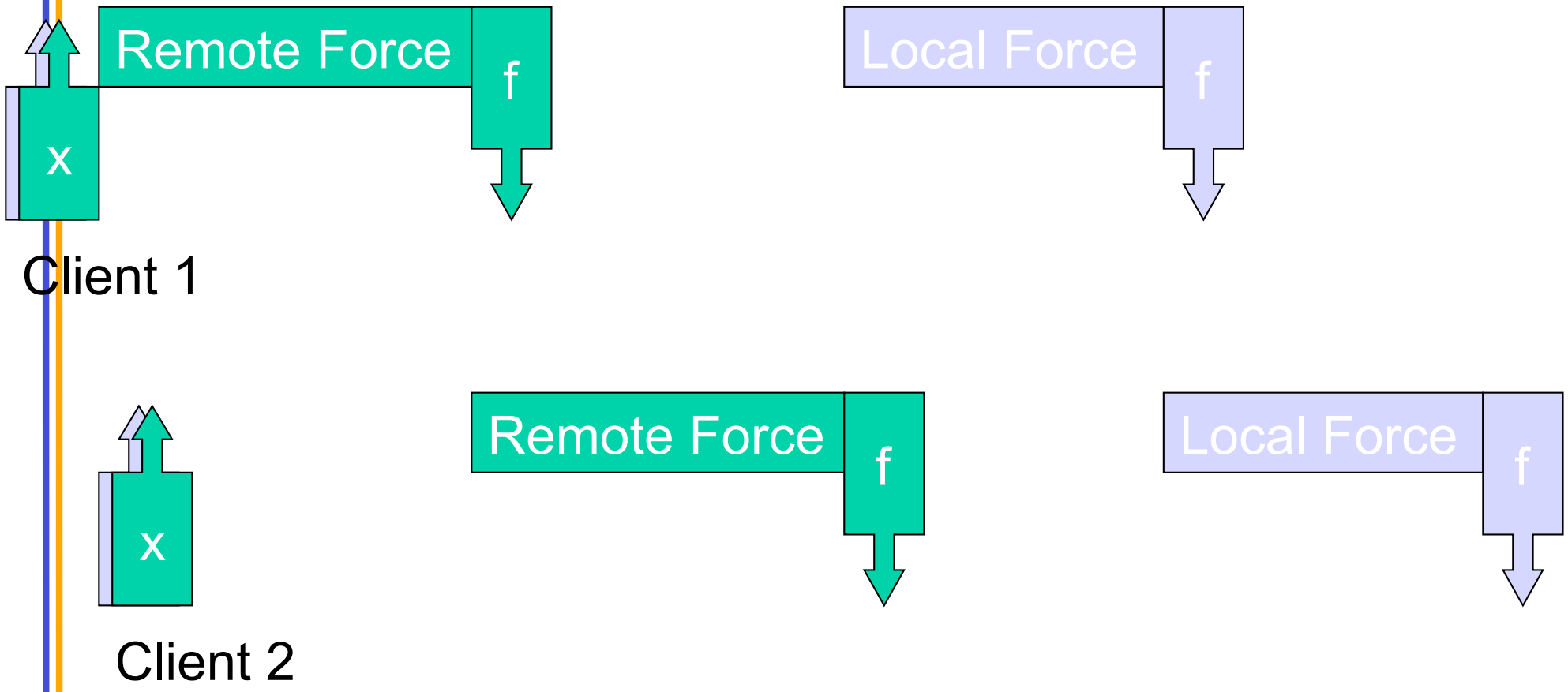
(8 cores and 2 GPUs per node)



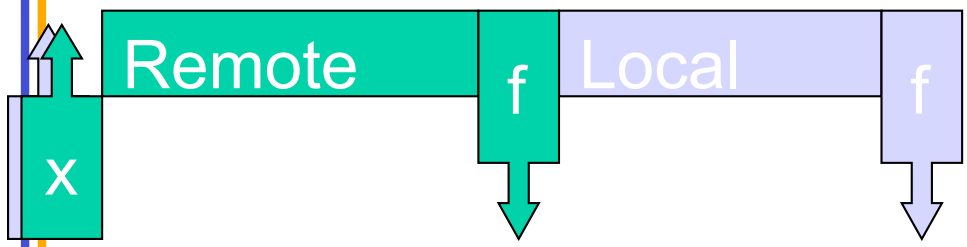
No GPU Sharing (Ideal World)



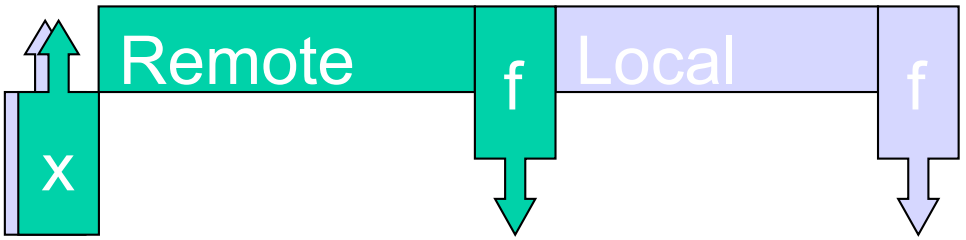
GPU Sharing (Desired)



GPU Sharing (Feared)

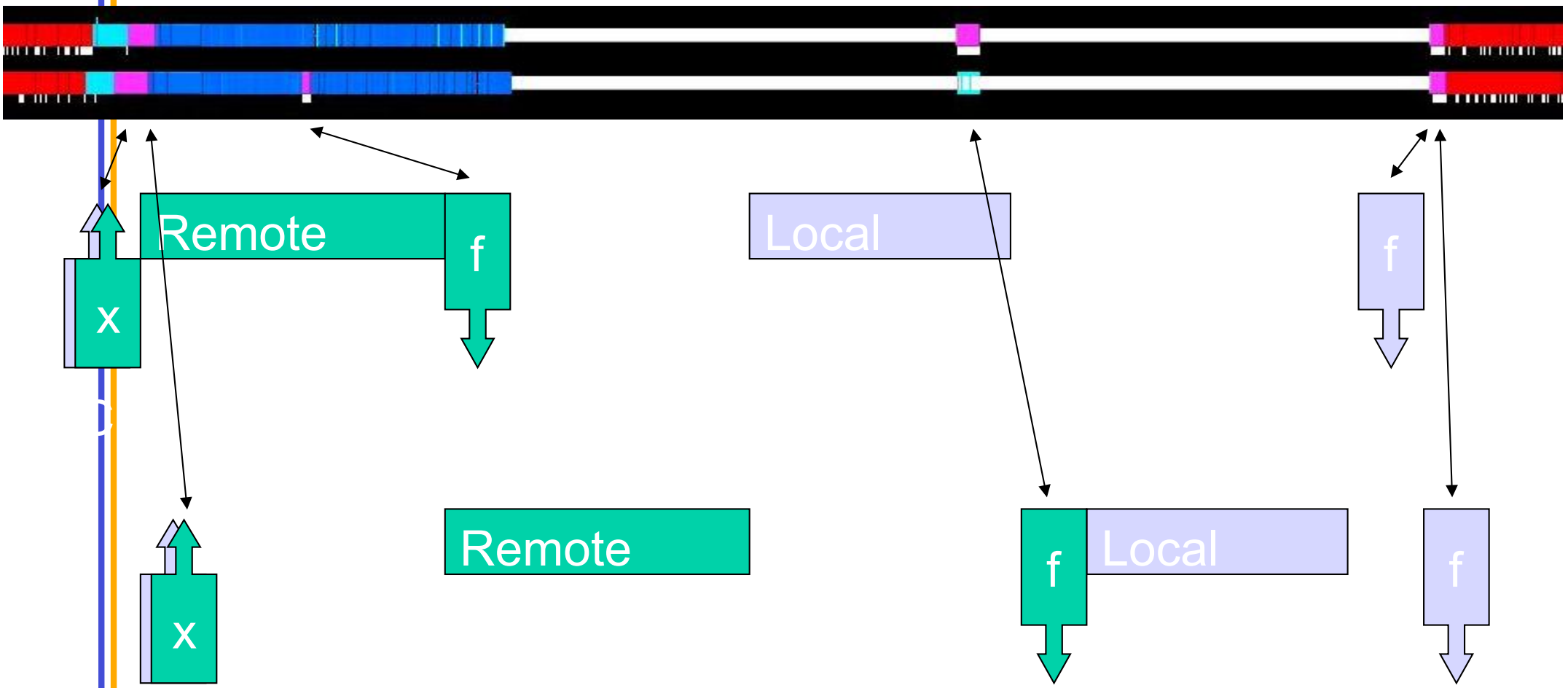


Client 1



Client 2

GPU Sharing (Observed)



GPU Sharing (Explained)

- CUDA is behaving reasonably, but
- Force calculation is actually two kernels
  - Longer kernel writes to multiple arrays
  - Shorter kernel combines output
- Possible solutions:
  - Modify CUDA to be less “fair” (please!)
  - Use locks (atomics) to merge kernels (not G80)
  - Explicit inter-client coordination

Inter-client Communication

- First identify which processes share a GPU
  - Need to know physical node for each process
  - GPU-assignment must reveal real device ID
  - Threads don't eliminate the problem
  - Production code can't make assumptions
- Token-passing is simple and predictable
  - Rotate clients in fixed order
  - High-priority, yield, low-priority, yield, …