Charm++: An Asynchronous Parallel Programming Model with an Intelligent Adaptive Runtime System

Laxmikant (Sanjay) Kale
http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

PARALLEL PROGRAMMING LAB
DEPT. OF COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS

Observations: Exascale applications

- Development of new models must be driven by the needs of exascale applications
  - Multi-resolution
  - Multi-module (multi-physics)
  - Dynamic/adaptive: to handle application variation
  - Adapt to a volatile computational environment
  - Exploit heterogeneous architecture
  - Deal with thermal and energy considerations
- So? Consequences:
  - Must support automated resource management
  - Must support interoperability and parallel composition

Decomposition Challenges

- Current method is to decompose to processors
  - But this has many problems
  - deciding which processor does what work in detail is difficult at large scale
- Decomposition should be independent of number of processors
  - My group's design principle since early 1990's
    - in Charm++ and AMPI

Processors vs. "WUDU"s

- Eliminate "processor" from programmer's vocabulary
  - Well, almost
- Decomposition into:
  - Work-Units and Data Units (WUDUs)
  - Work-units: code, one or more data units
  - Data-units: sections of arrays, meshes, …
  - Amalgams:
    - Objects with associated work-units,
    - Threads with own stack and heap
- Who does decomposition?
  - Programmer, compiler, or both

Different kinds of units

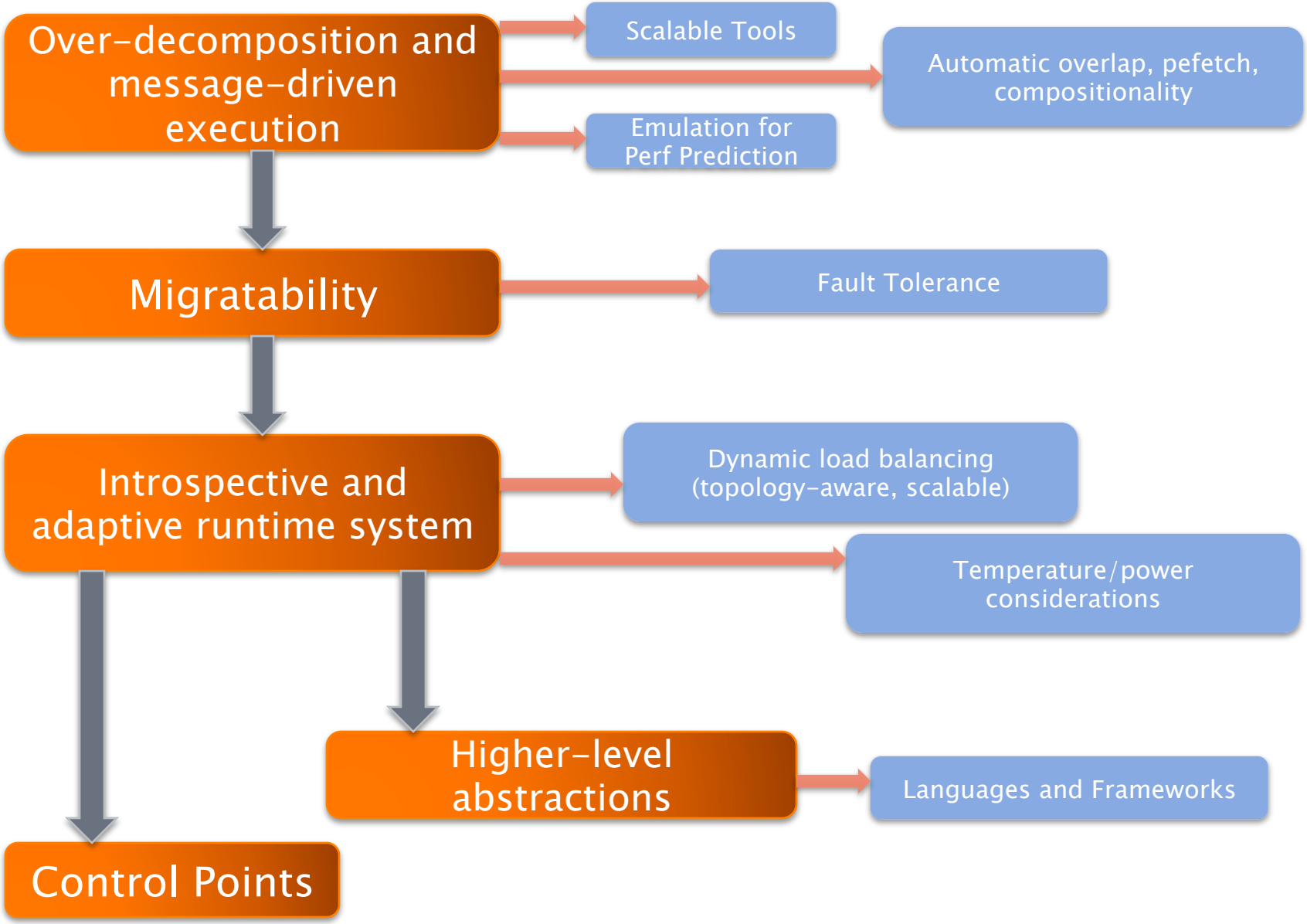
- Migration units:
  - objects, migratable threads (i.e., "processes"), data sections
- DEBs: units of scheduling
  - Dependent Execution Block
  - Begins execution after one or more (potentially) remote dependence is satisfied
- SEBs: units of analysis
  - Sequential Execution Blocks
  - A DEB is partitioned into one or more SEBs
  - Has a "reasonably large" granularity, and uniformity in code structure
  - Loop nests, functions, …

Migratable objects programming model

- Names for this model:
  - Overdecompostion approach
  - Object-based overdecomposition
  - Processor virtualization
  - Migratable-objects programming model

Adaptive Runtime Systems

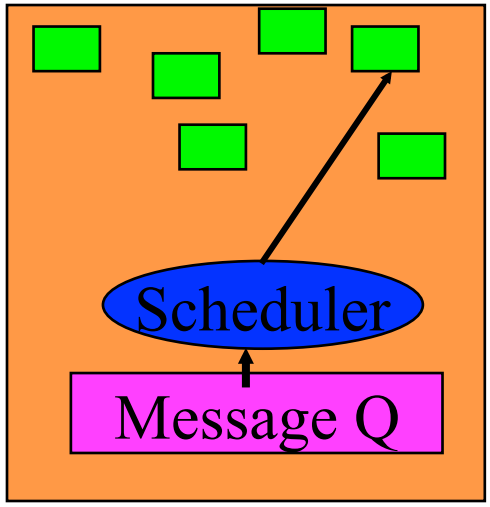
- Decomposing program into a large number of WUDUs empowers the RTS, which can:
  - Migrate WUDUs at will
  - Schedule DEBS at will
  - Instrument computation and communication at the level of these logical units
    - WUDU x communicates y bytes to WUDU z every iteration
    - SEB A has a high cache miss ratio
  - Maintain historical data to track changes in application behavior
    - Historical => previous iterations
    - E.g., to trigger load balancing

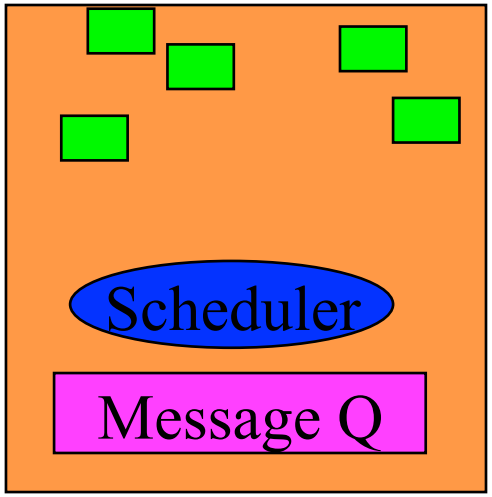

Over-decomposition and message-driven execution
Scalable Tools
Automatic overlap, pefetch, compositionality
Emulation for Perf Prediction
Migratability
Fault Tolerance
Introspective and adaptive runtime system
Dynamic load balancing (topology-aware, scalable)
Temperature/power considerations
Higher-level abstractions
Languages and Frameworks
Control Points

PPL UIUC

Utility for Multi-cores, Many-cores, Accelerators:

- Objects connote and promote locality
- Message-driven execution
  - A strong principle of prediction for data and code use
  - Much stronger than principle of locality
    - Can use to scale memory wall:
    - Prefetching of needed data:
      - into scratch pad memories, for example





Impact on communication

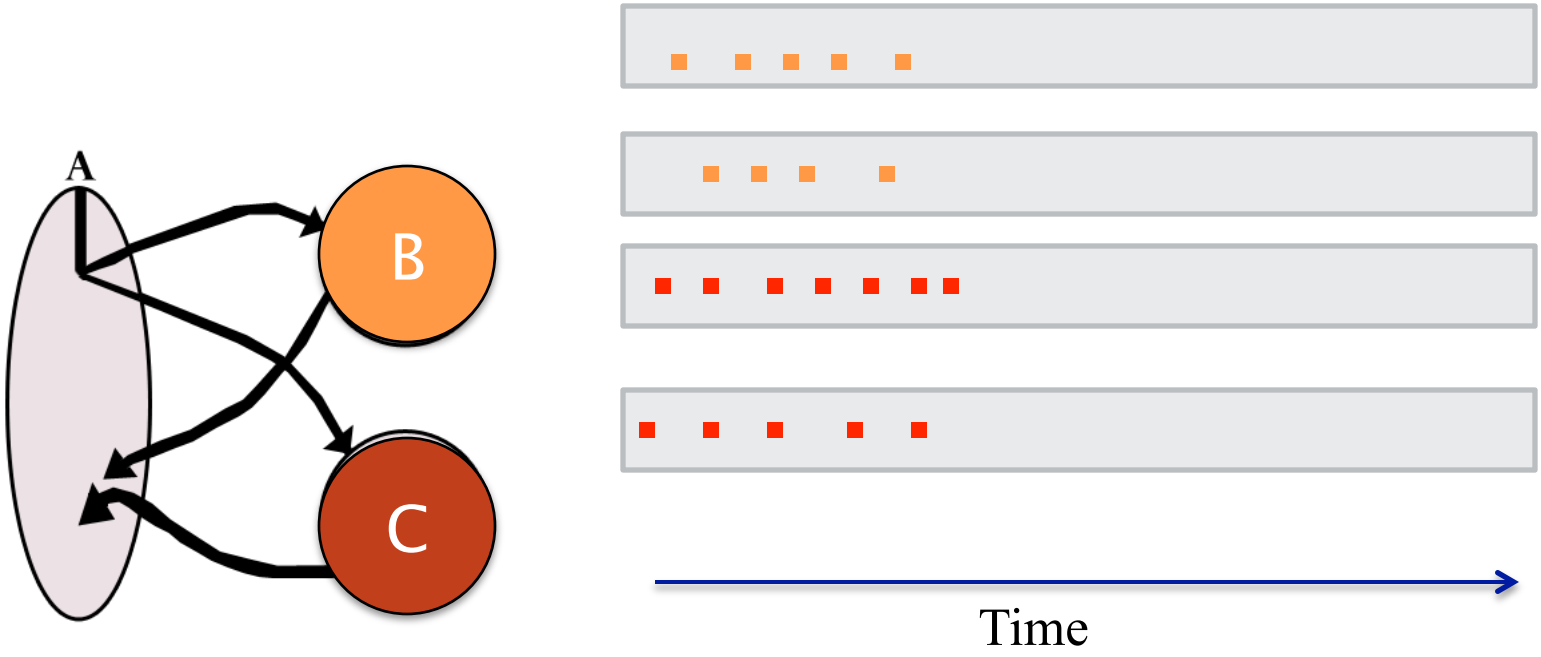
- Current use of communication network:
  - Compute-communicate cycles in typical MPI apps
  - So, the network is used for a fraction of time,
  - and is on the critical path
- So, current *communication networks are over-engineered for by necessity*
- With overdecomposition
  - Communication is spread over an iteration
  - Also, adaptive overlap of communication and computation

Compositionality

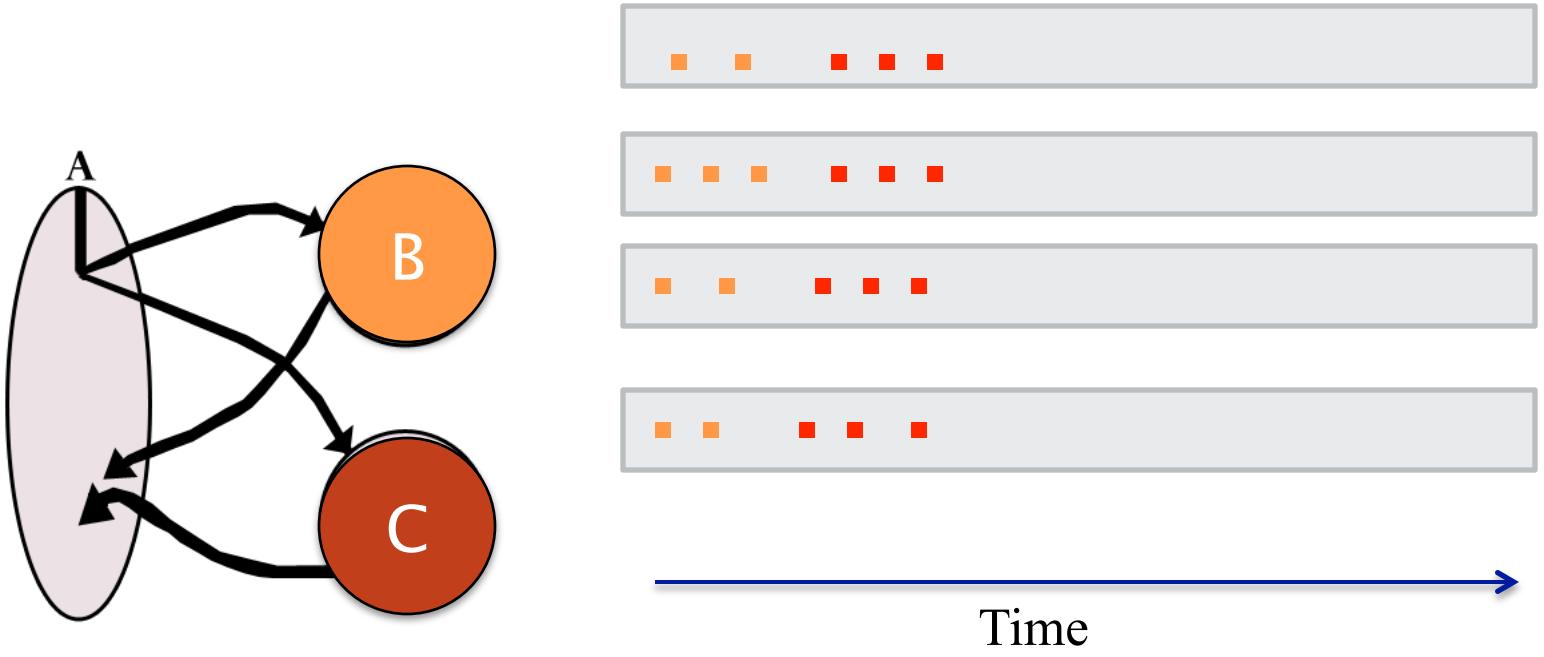
- It is important to support parallel composition
  - For multi-module, multi-physics, multi-paradigm applications…
- What I mean by parallel composition
  - B || C where B, C are independently developed modules
  - B is parallel module by itself, and so is C
  - Programmers who wrote B were unaware of C
  - No dependency between B and C
- This is not supported well by MPI
  - Developers support it by breaking abstraction boundaries
    - E.g., wildcard recvs in module A to process messages for module B
  - Nor by OpenMP implementations:

Without message-driven execution (and virtualization), you get either:

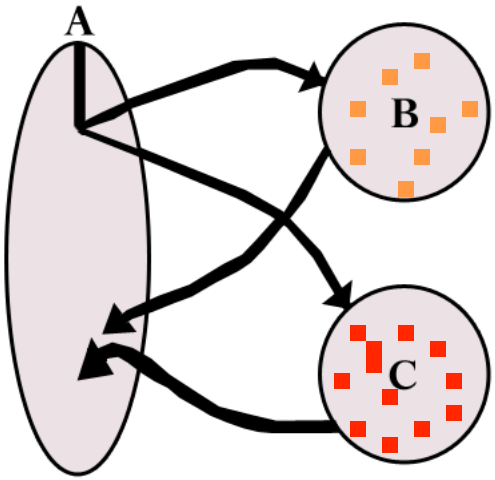
Space-division



OR: Sequentialization



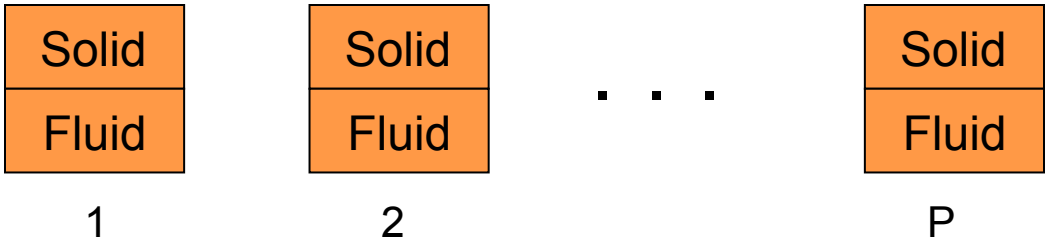
Parallel Composition: A1; (B || C); A2

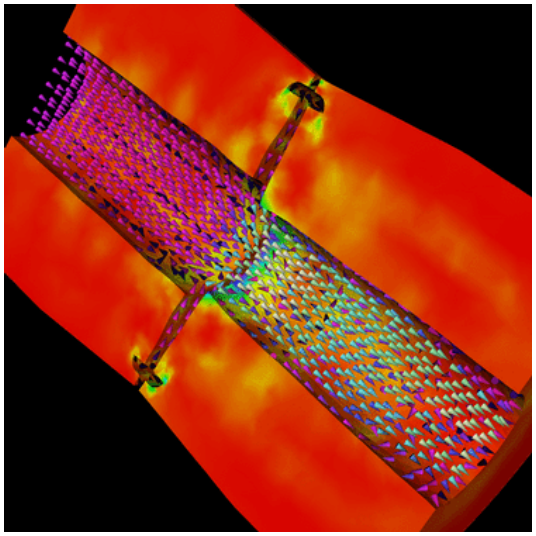


Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly

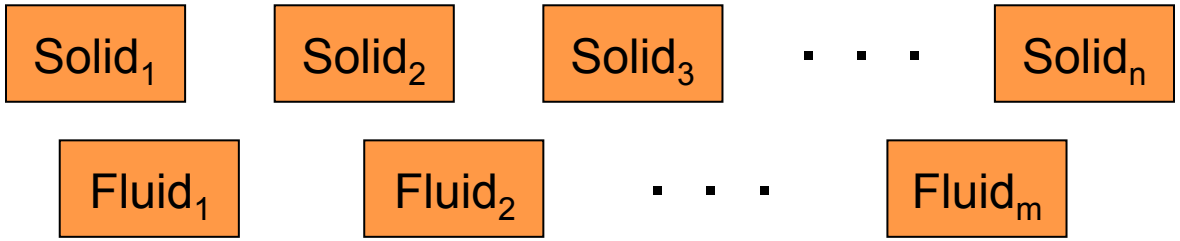
Decomposition Independent of numCores

- Rocket simulation example under traditional MPI



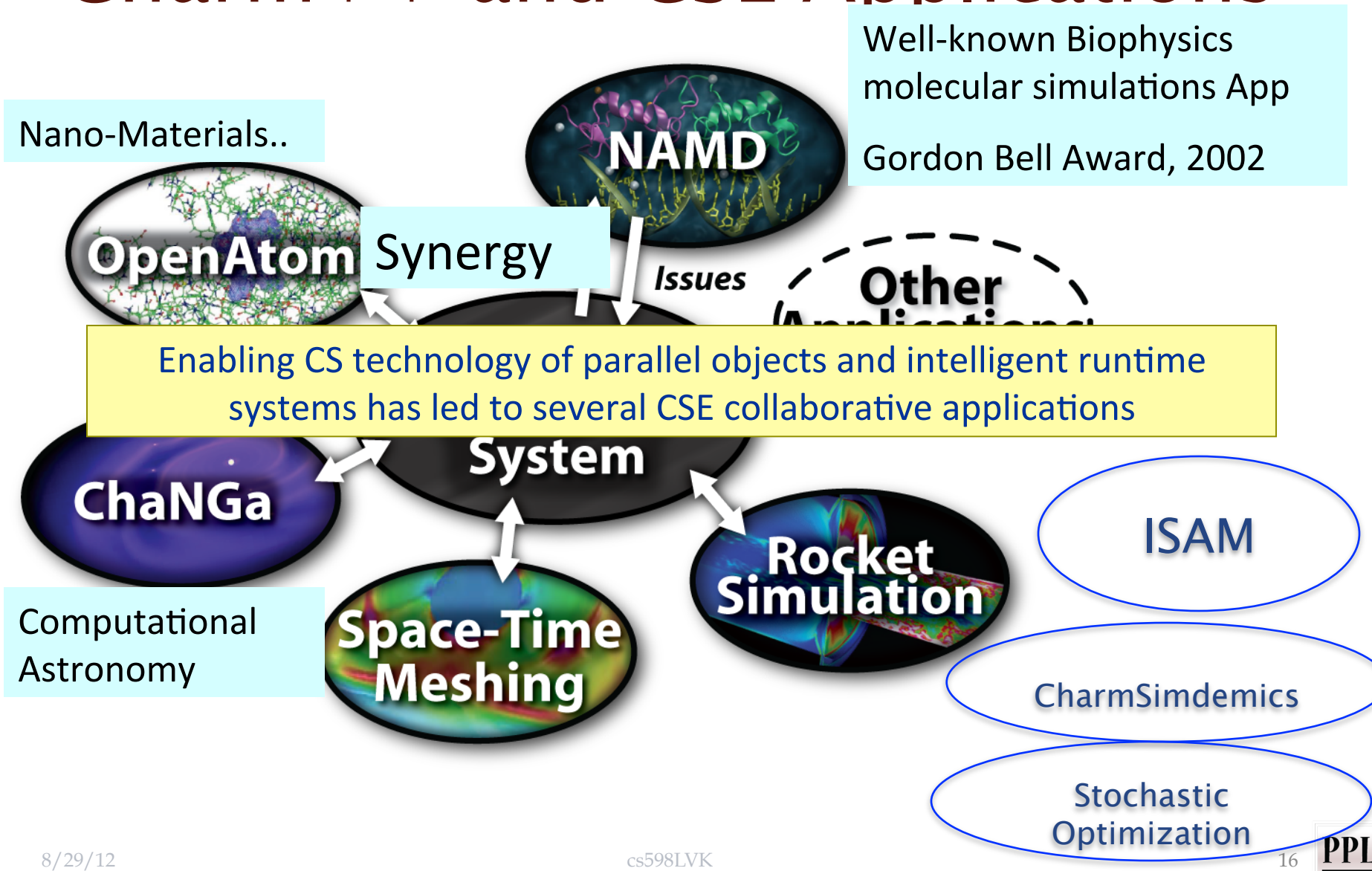


- With migratable-objects:



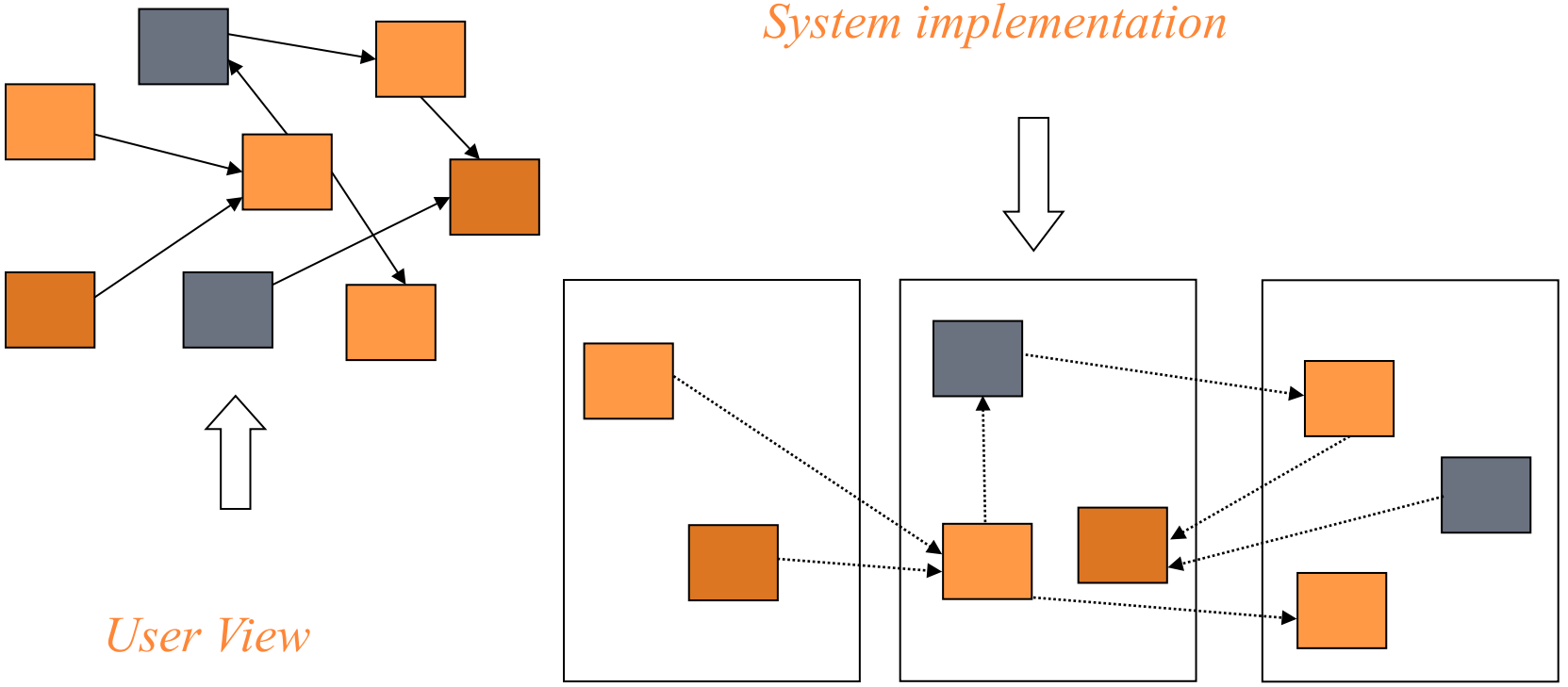
  - Benefit: load balance, communication optimizations, modularity

Charm++ and CSE Applications



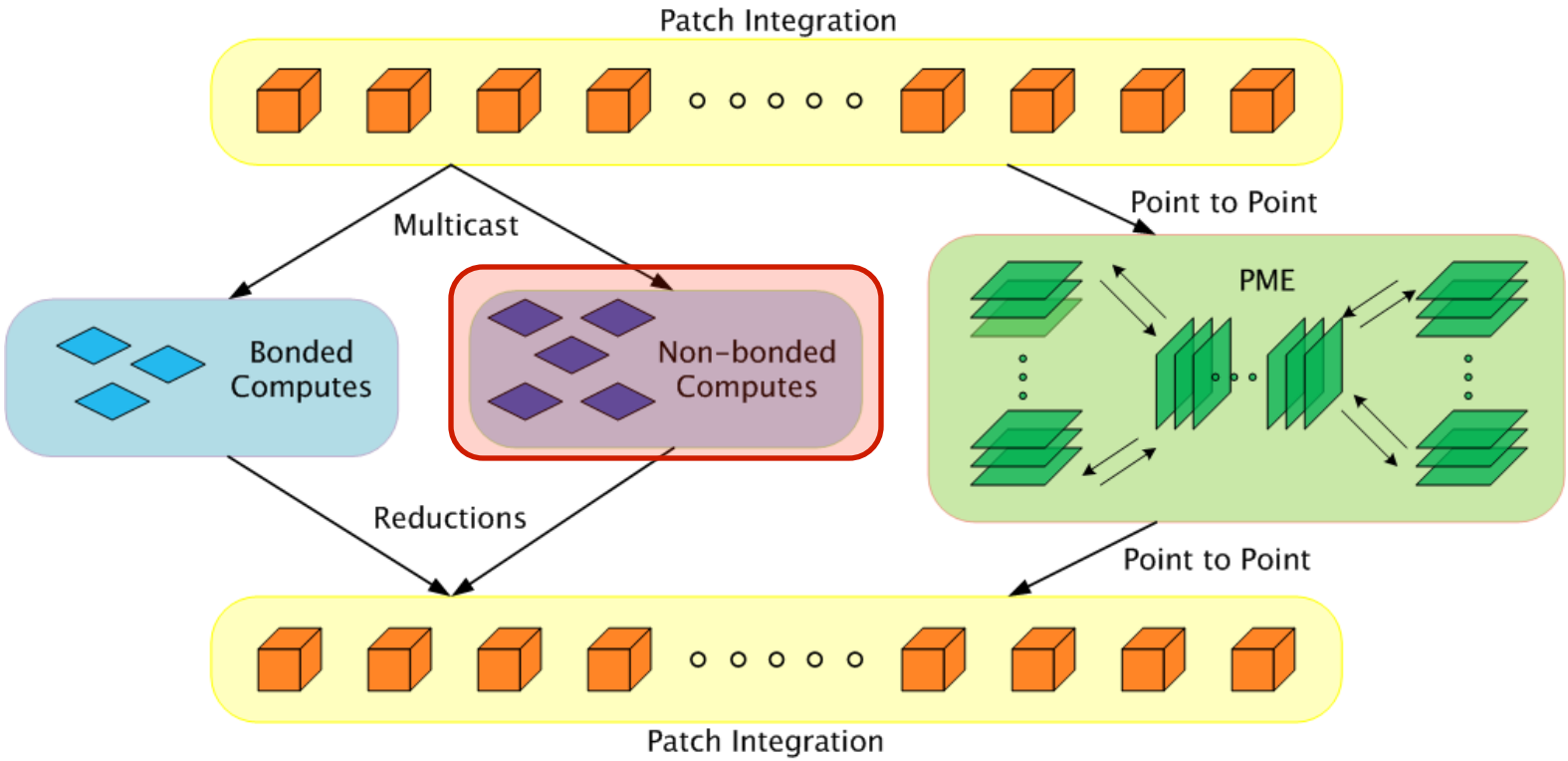
Object Based Over-decomposition: Charm++

- Multiple "indexed collections" of C++ objects
- Indices can be multi-dimensional and/or sparse
- Programmer expresses communication between objects
  - with no reference to processors



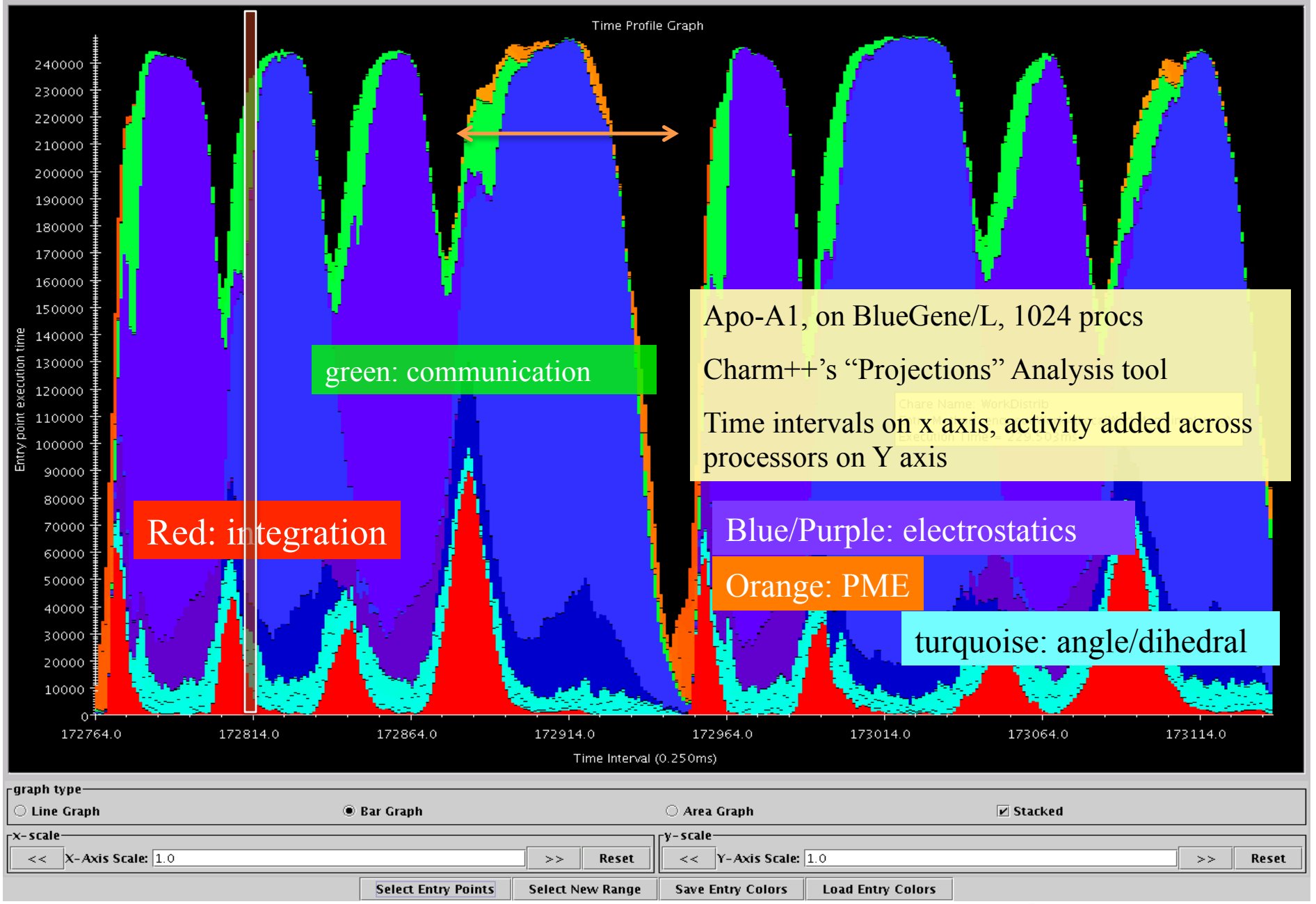
Parallelization Using Charm++

The computation is decomposed into "natural" objects of the application, which are assigned to processors by Charm++ RTS



Bhatele, A., Kumar, S., Mei, C., Phillips, J. C., Zheng, G. & Kale, L. V. 2008 **Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms**. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Miami, FL, USA, April 2008.

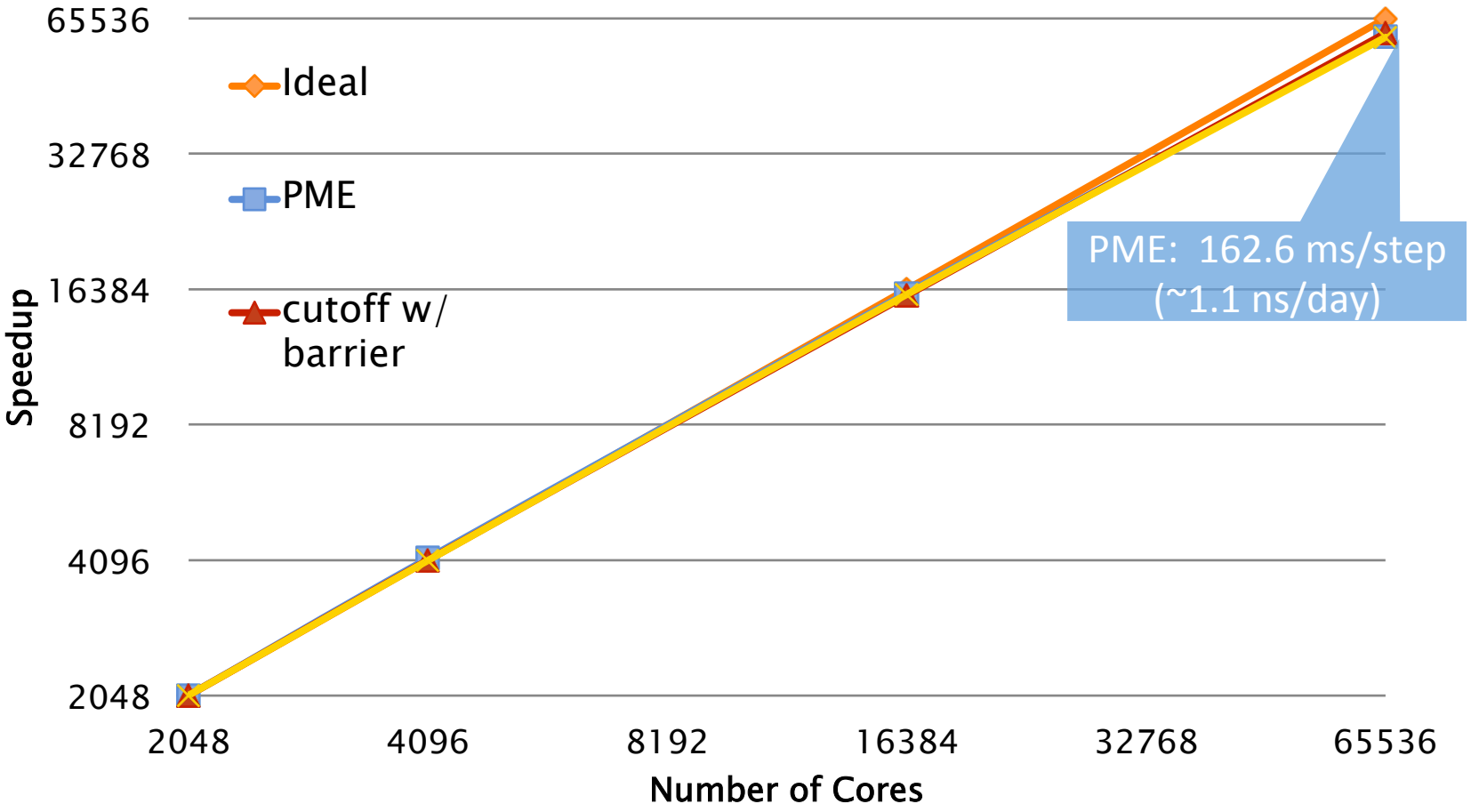
PPL UIUC



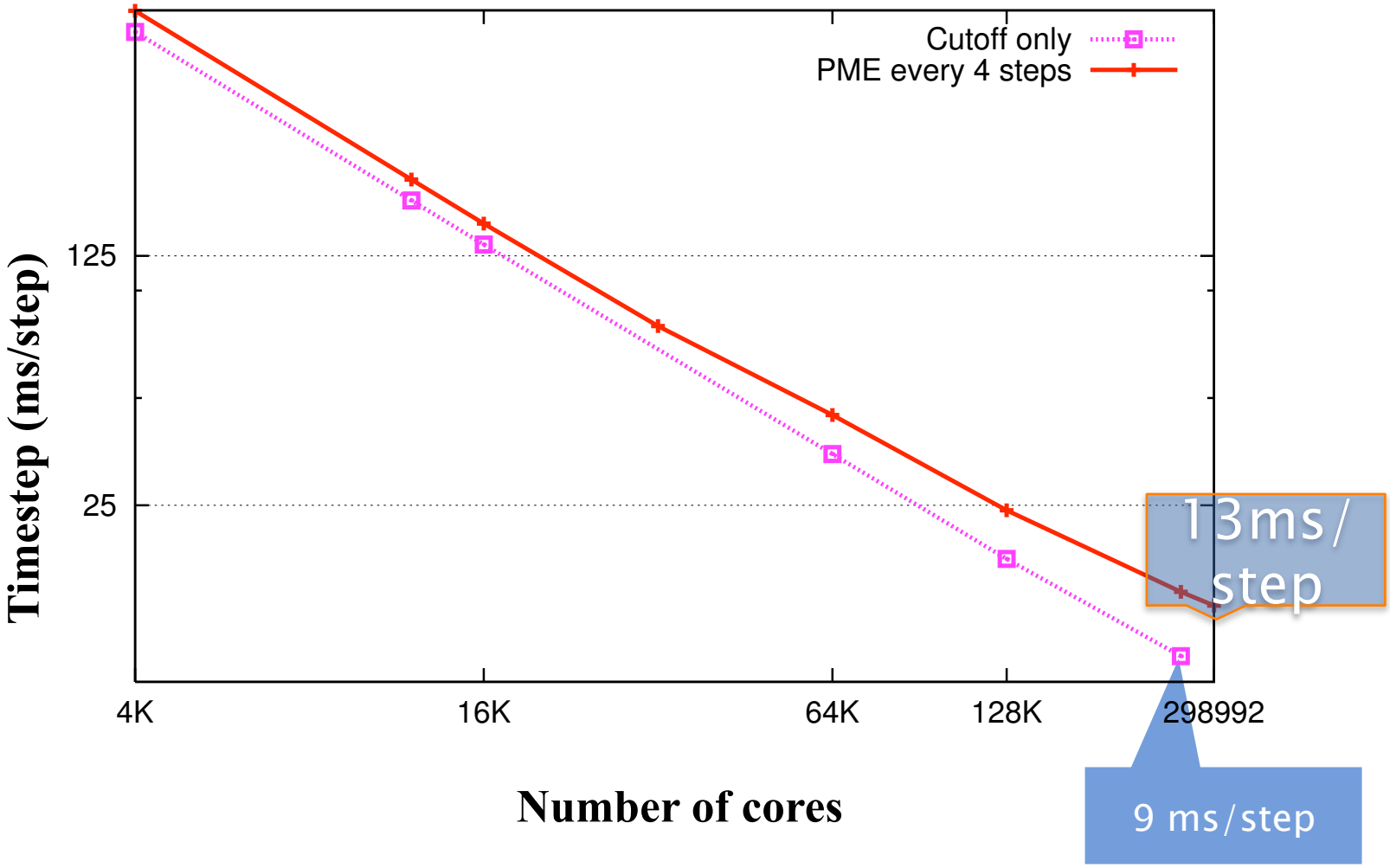
Time

PPL UIUC

Performance on Intrepid (BG/P)

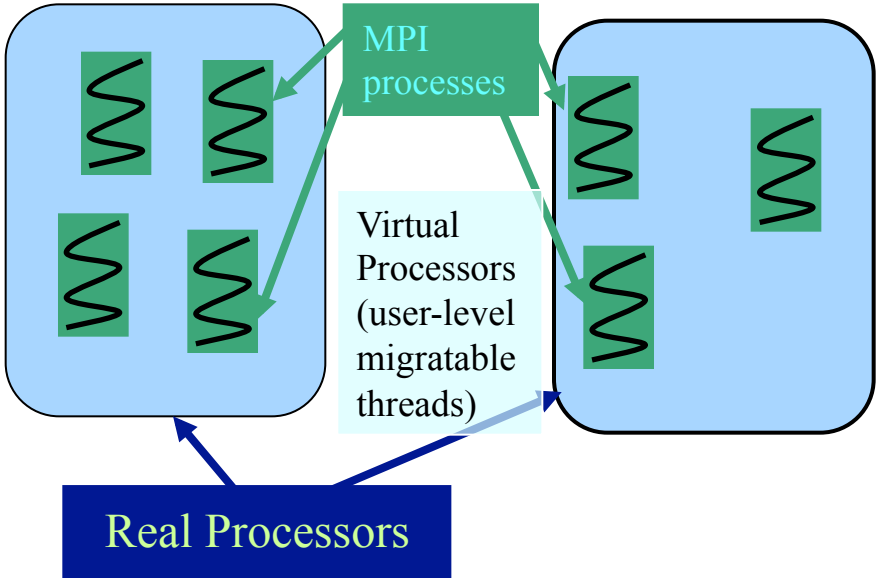


SMP Performance on Titan(Dev)



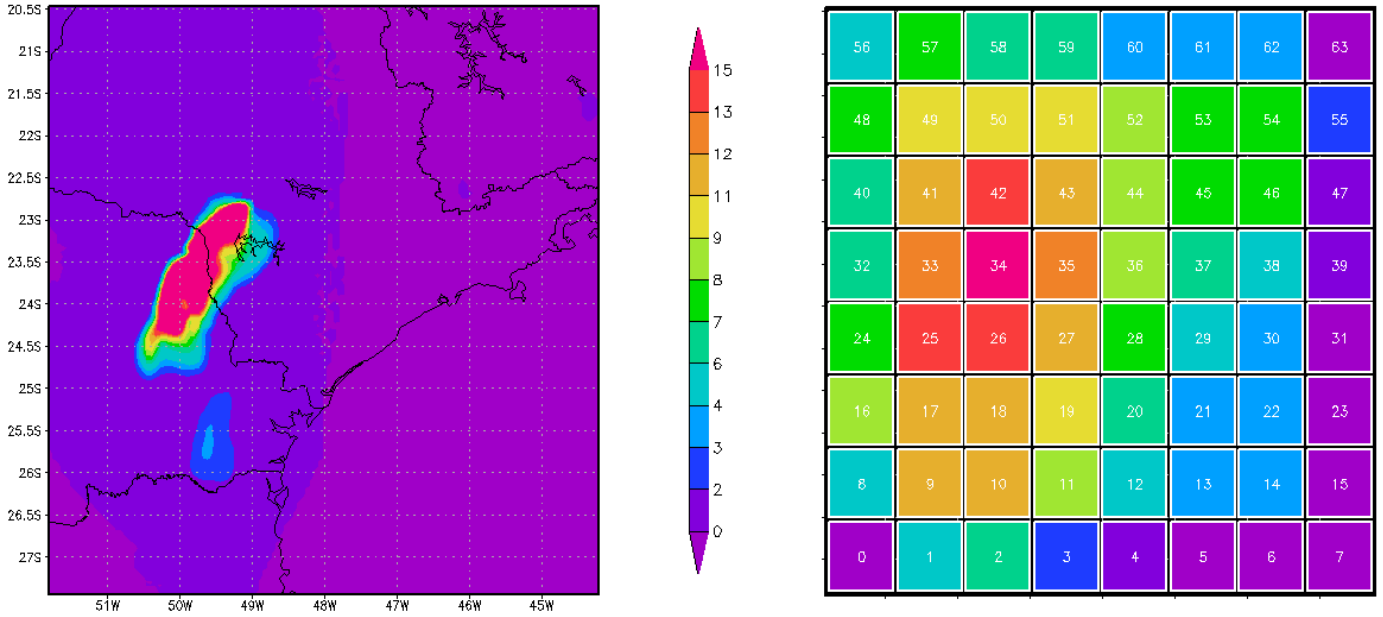
Object Based Over-decomposition: AMPI

- Each MPI process is implemented as a user-level thread
- Threads are light-weight and migratable!
  - <1 microsecond context switch time, potentially >100k threads per core
- Each thread is embedded in a charm++ object (chare)

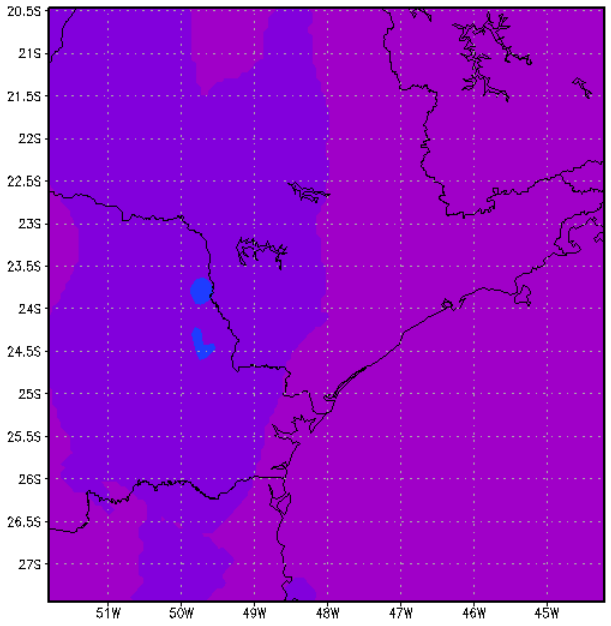


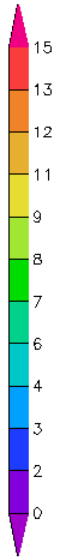
A quick Example: Weather Forecasting in BRAMS

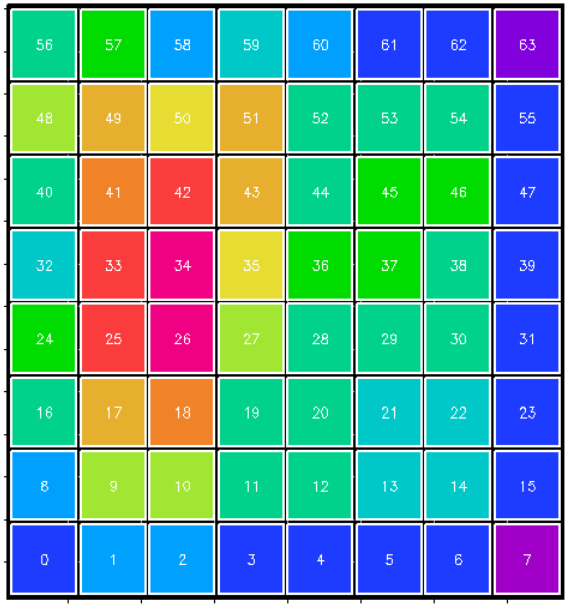
- Brams: Brazillian weather code (based on RAMS)
- AMPI version (Eduardo Rodrigues, with Mendes and J. Panetta)



PPL UIUC

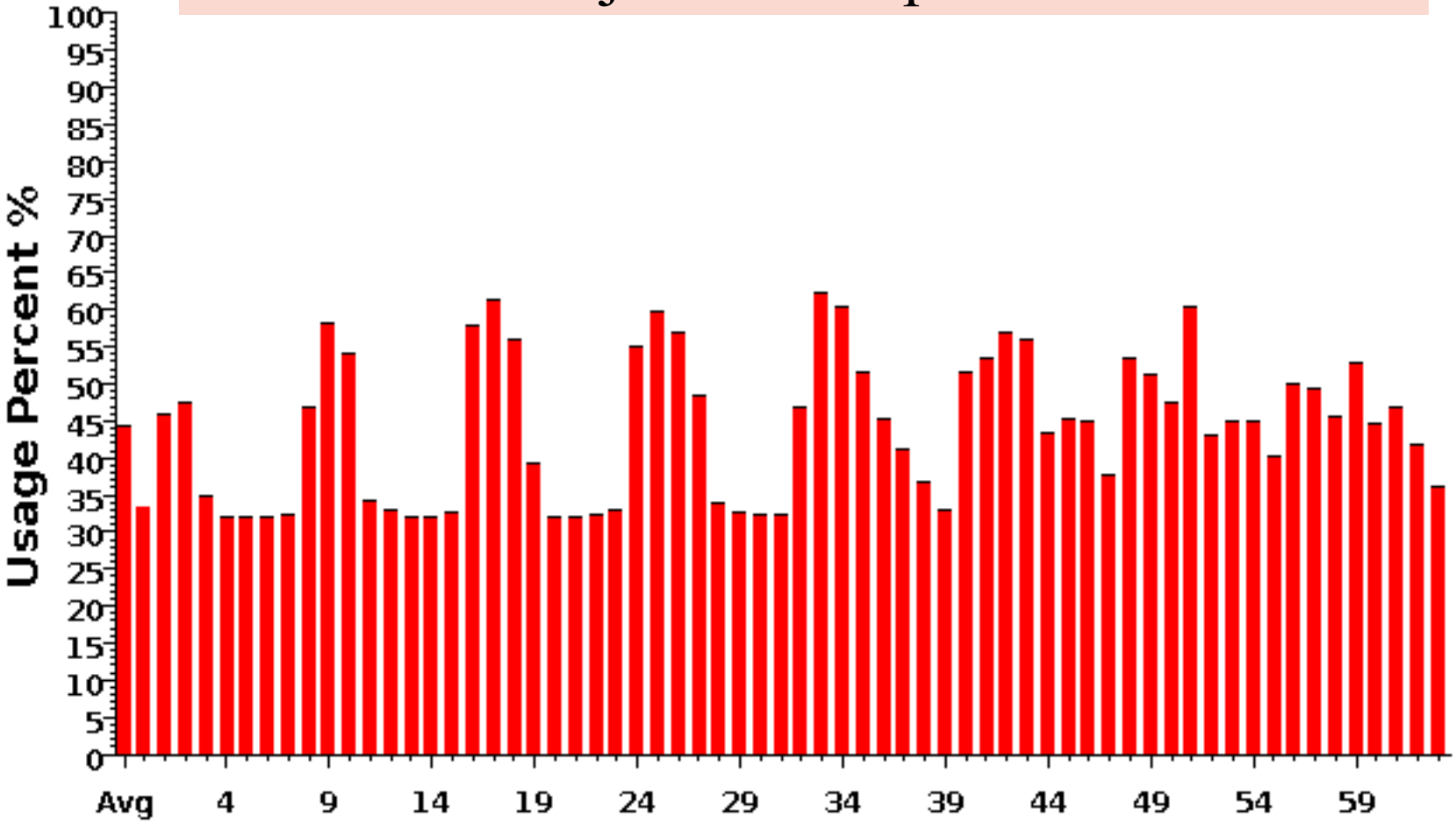




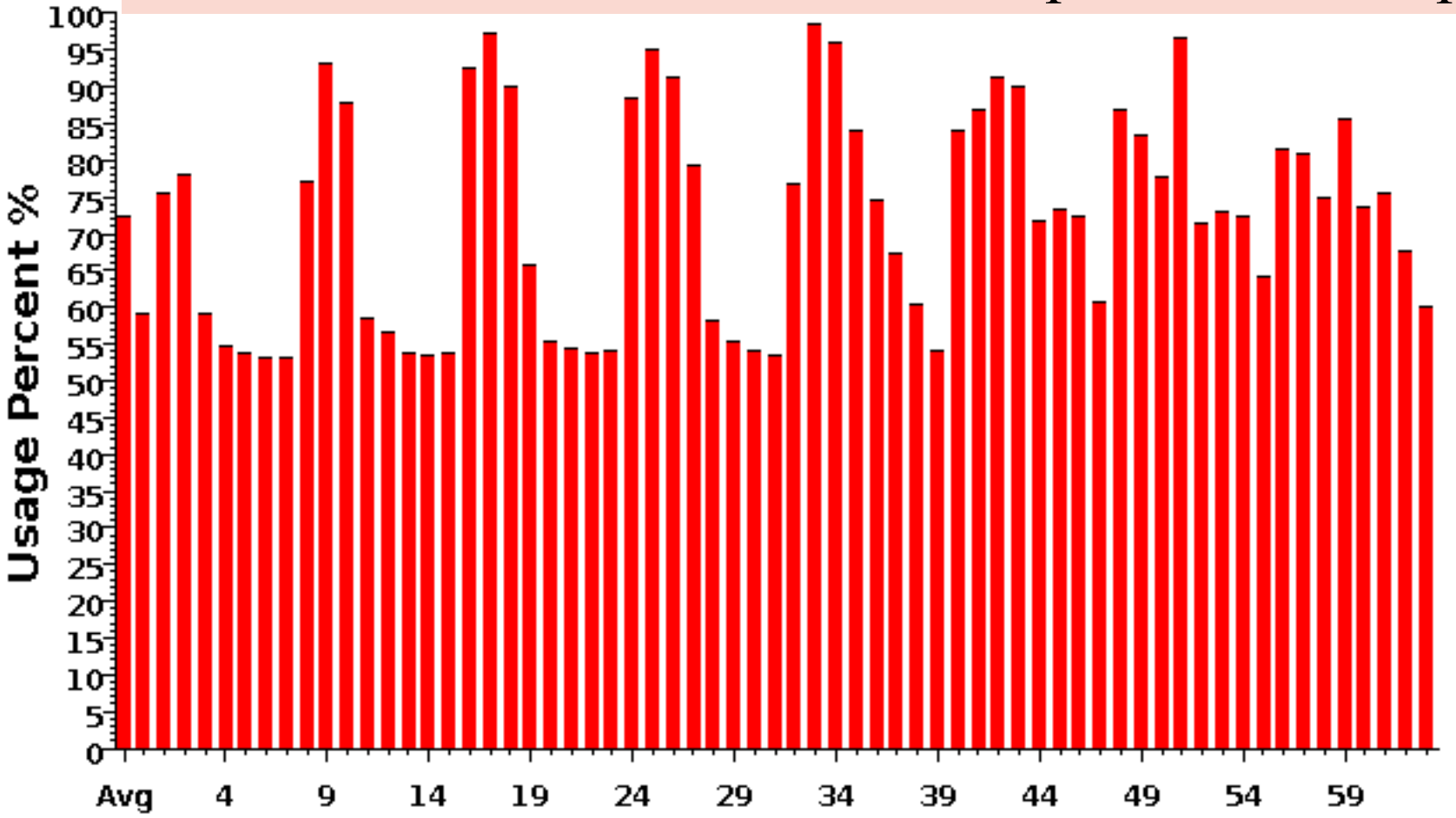


GrADS: COLA/IGES 2010-02-18-09:46 GrADS: COLA/IGES 2010-02-18-10:00

Baseline: 64 objects on 64 processors

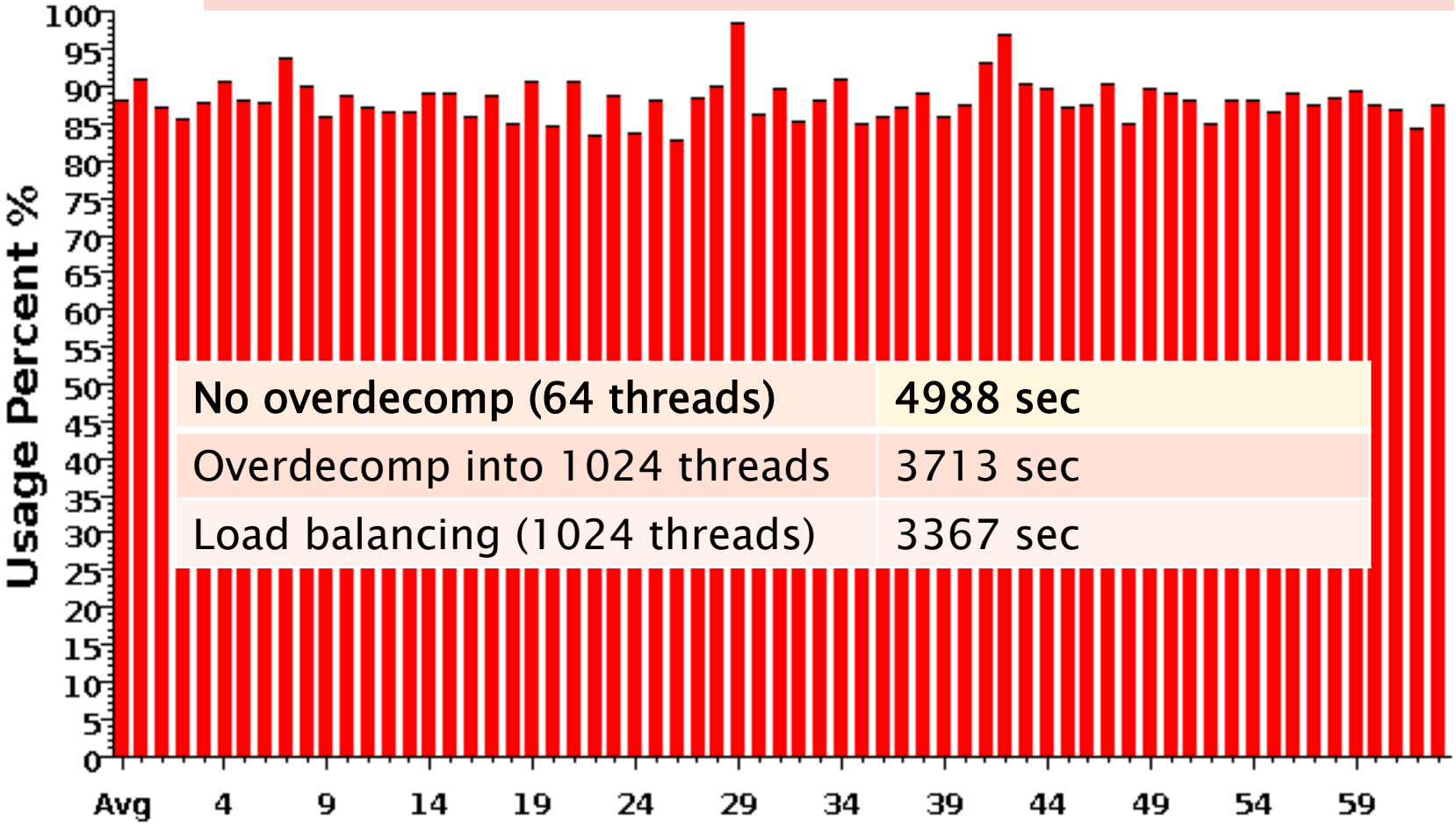


Over-decomposition: 1024 objects on 64 processors: Benefits from communication/computation overlap



PPL UIUC

With Load Balancing:
1024 objects on 64 processors



No overdecomp (64 threads)	4988 sec
Overdecomp into 1024 threads	3713 sec
Load balancing (1024 threads)	3367 sec

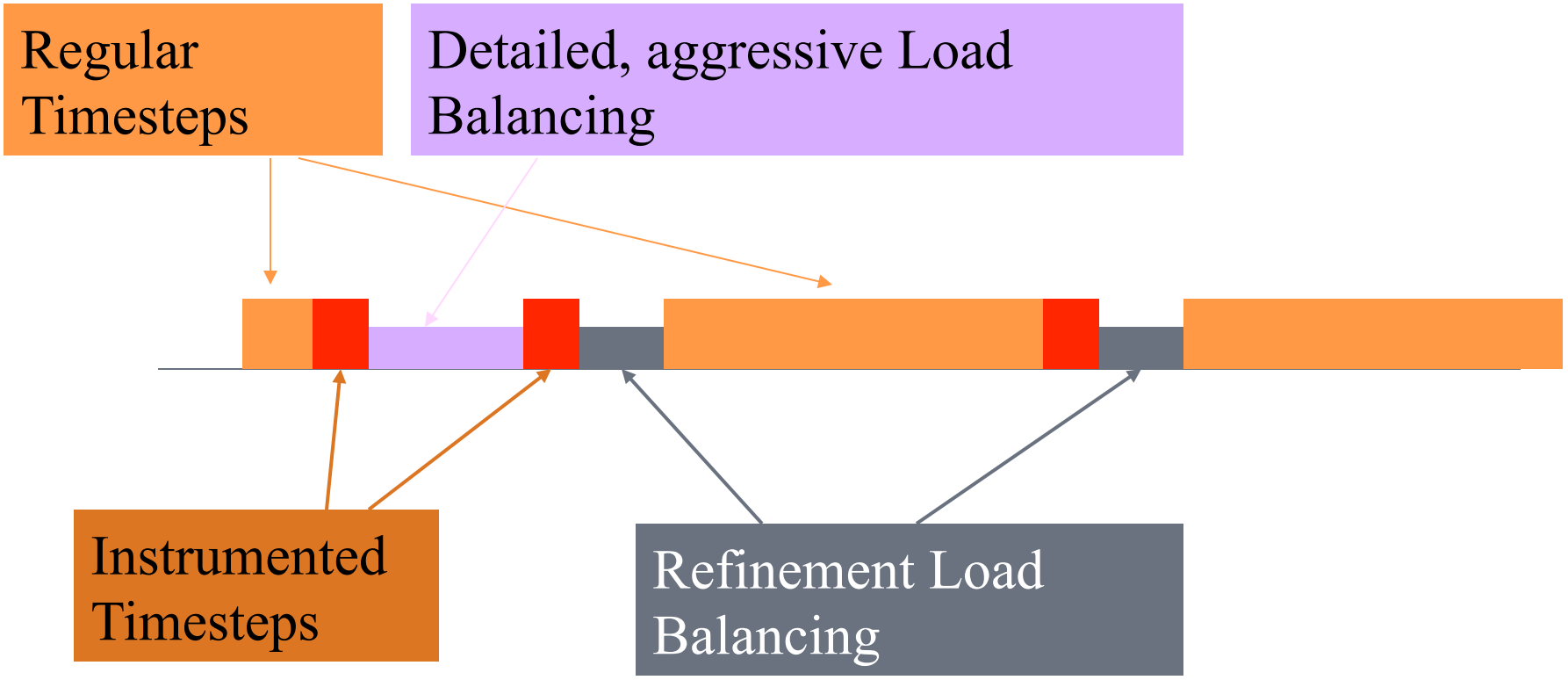
PPL UIUC

Principle of Persistence

- Once the computation is expressed in terms of its natural (migratable) objects
- *Computational loads and communication patterns tend to persist, even in dynamic computations*
- So, recent past is a good predictor of near future

In spite of increase in irregularity and adaptivity, this principle still applies at exascale, and is our main friend.

Measurement-based Load Balancing



ChaNGa: Parallel Gravity

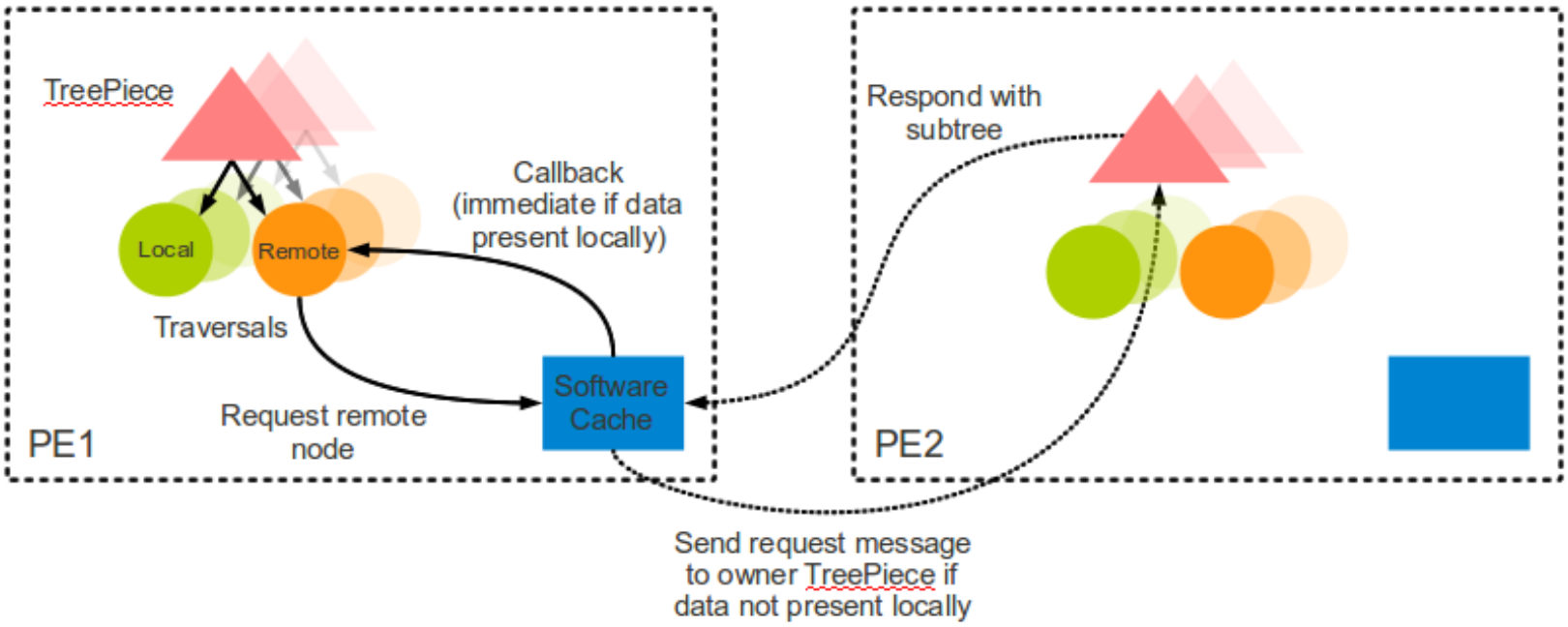
- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomp
  - Geometry has better aspect ratios, so you "open" up fewer nodes
  - But is not used because it leads to bad load balance
  - Assumption: one-to-one map between sub-trees and PEs
  - Binary trees are considered better load balanced

Evolution of Universe and Galaxy Formation

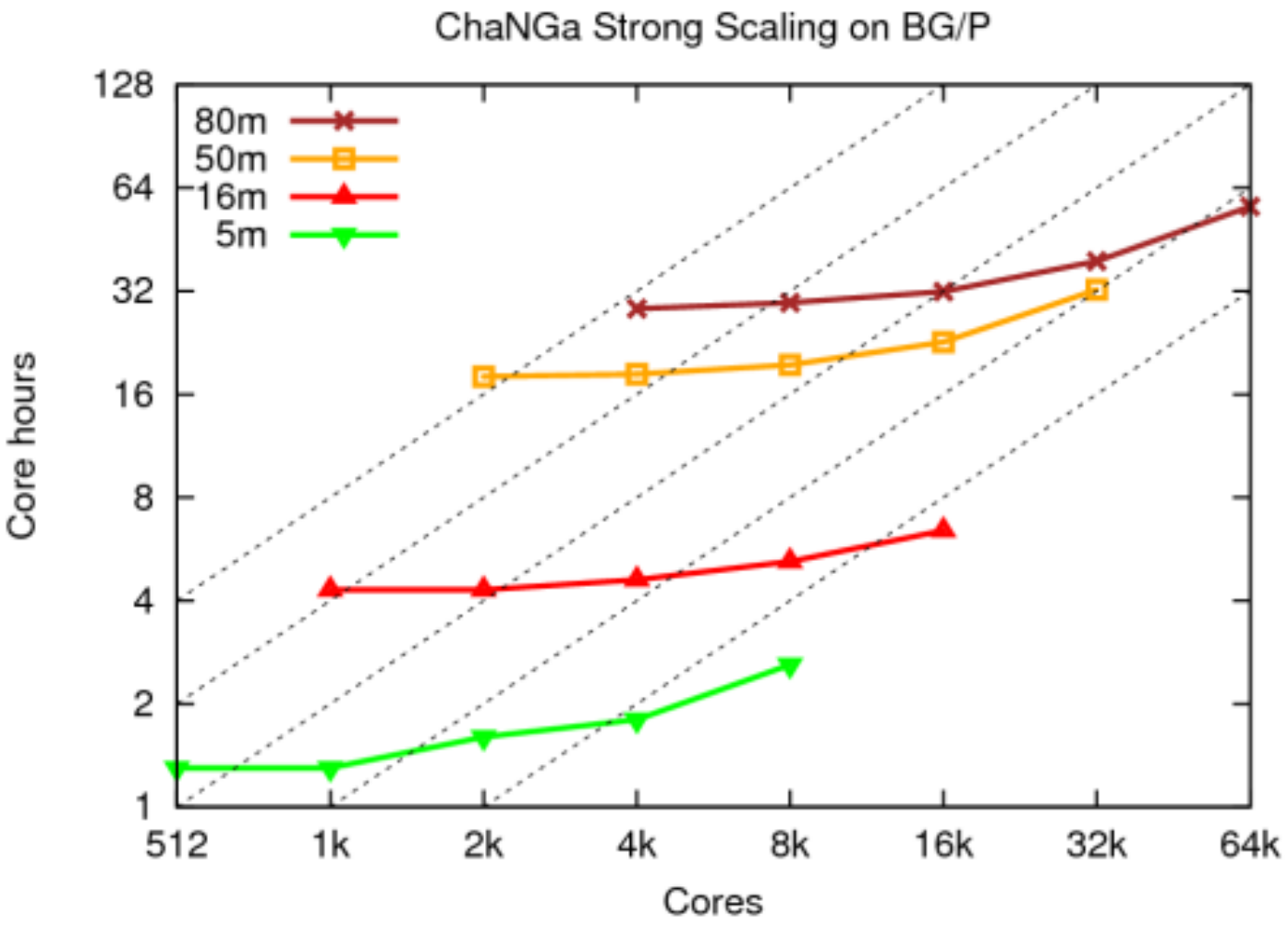
With Charm++: Use Oct-Tree, and let Charm++ map subtrees to processors

PPL UIUC

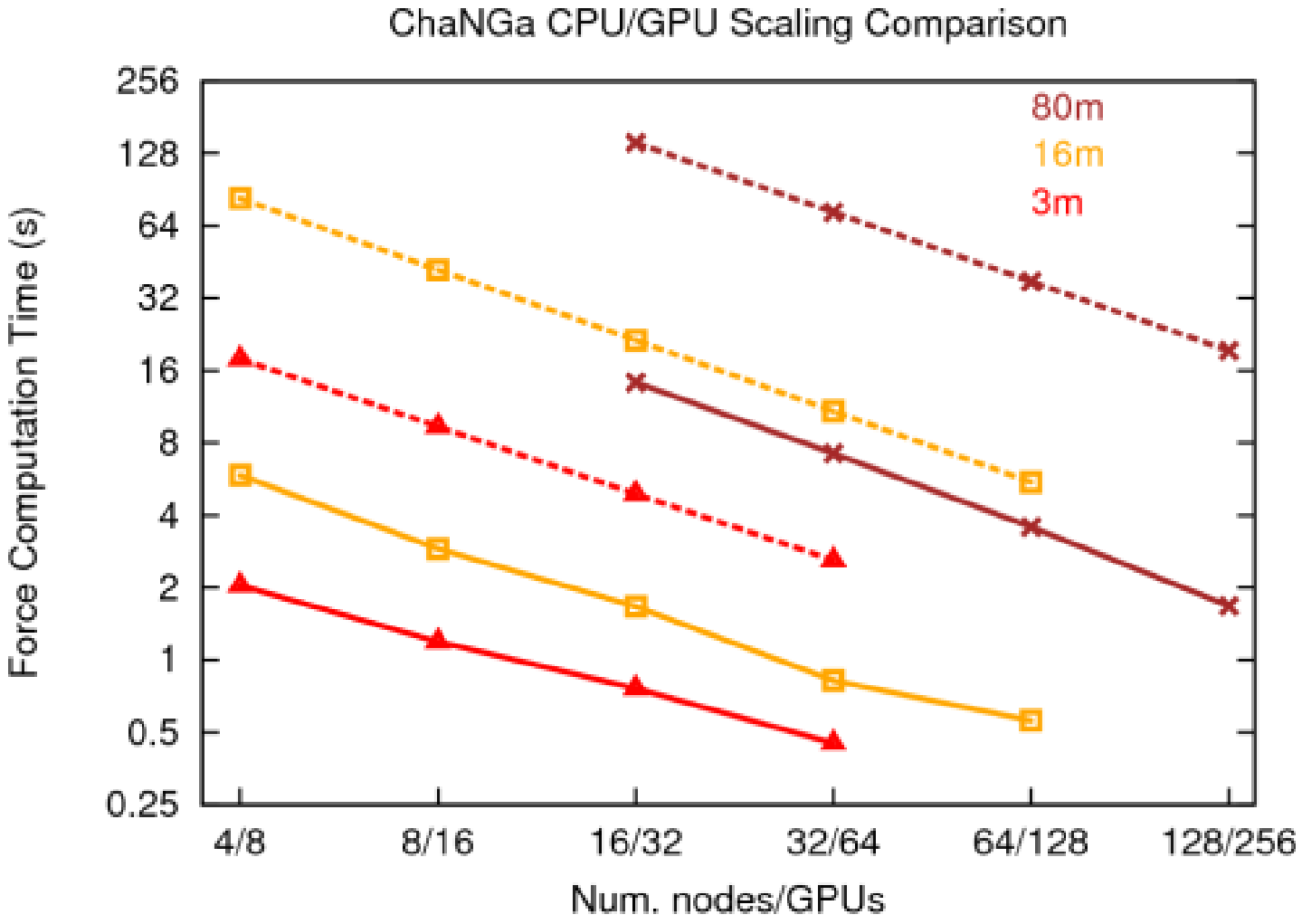
Control flow



CPU Performance



GPU Performance



Load Balancing for Large Machines: I

- Centralized balancers achieve best balance
  - Collect object-communication graph on one processor
  - But won't scale beyond tens of thousands of nodes
- Fully distributed load balancers
  - Avoid bottleneck but… Achieve poor load balance
  - Not adequately agile
- Hierarchical load balancers
  - Careful control of what information goes up and down the hierarchy can lead to fast, high-quality balancers
- Need for a universal balancer that works for all applications

Load Balancing for Large Machines: II

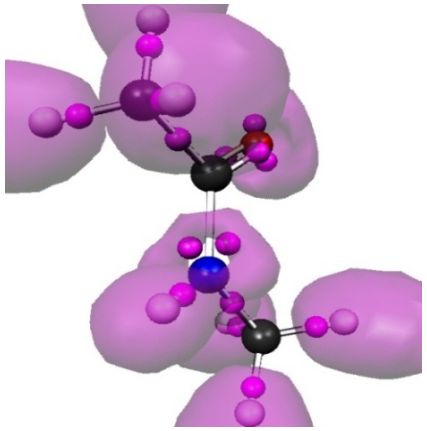
- Interconnection topology starts to matter again
  - Was hidden due to wormhole routing etc.
  - Latency variation is still small
  - But bandwidth occupancy is a problem
- Topology aware load balancers
  - Some general heuristic have shown good performance
    - But may require too much compute power
  - Also, special-purpose heuristic work fine when applicable
  - Still, many open challenges

OpenAtom

Car-Parinello Molecular Dynamics

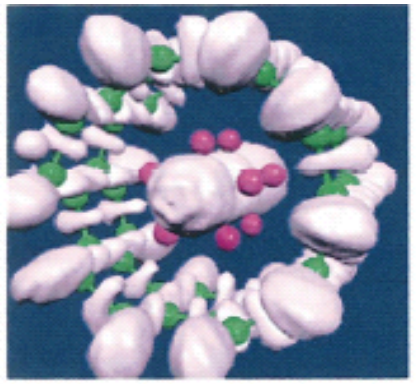
NSF ITR 2001-2007, IBM, DOE

Molecular Clusters :

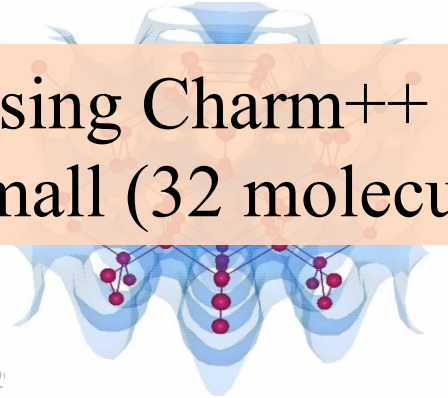


G. Martyna (IBM)
M. Tuckerman (NYU)
L. Kale (UIUC)
J. Dongarra

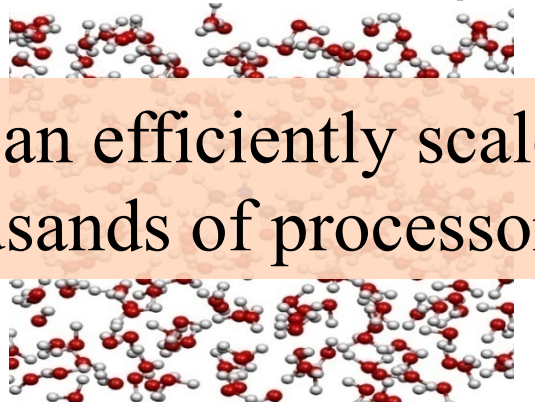
Nanowires:



Semiconductor Surfaces:



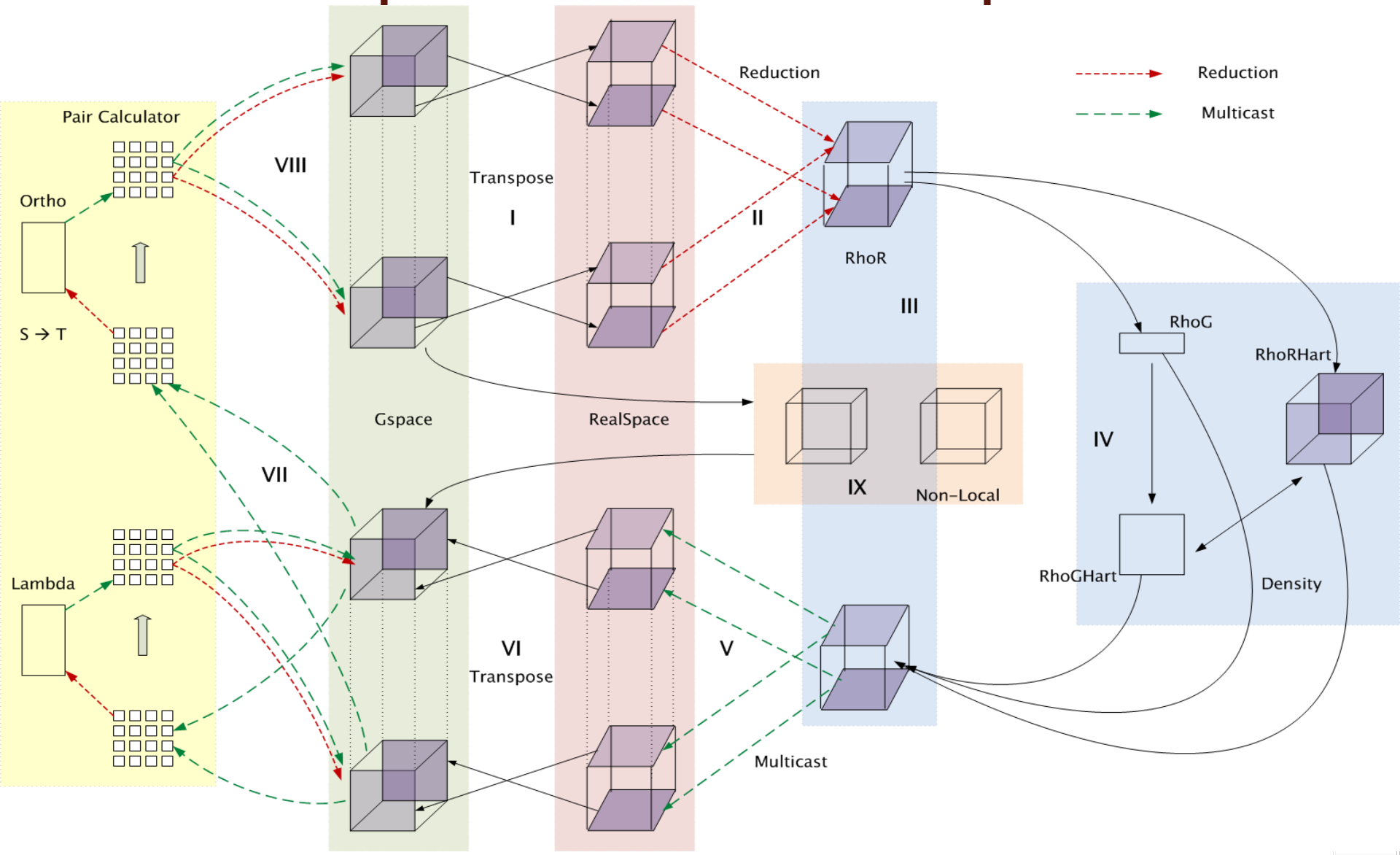
3D-Solids/Liquids:



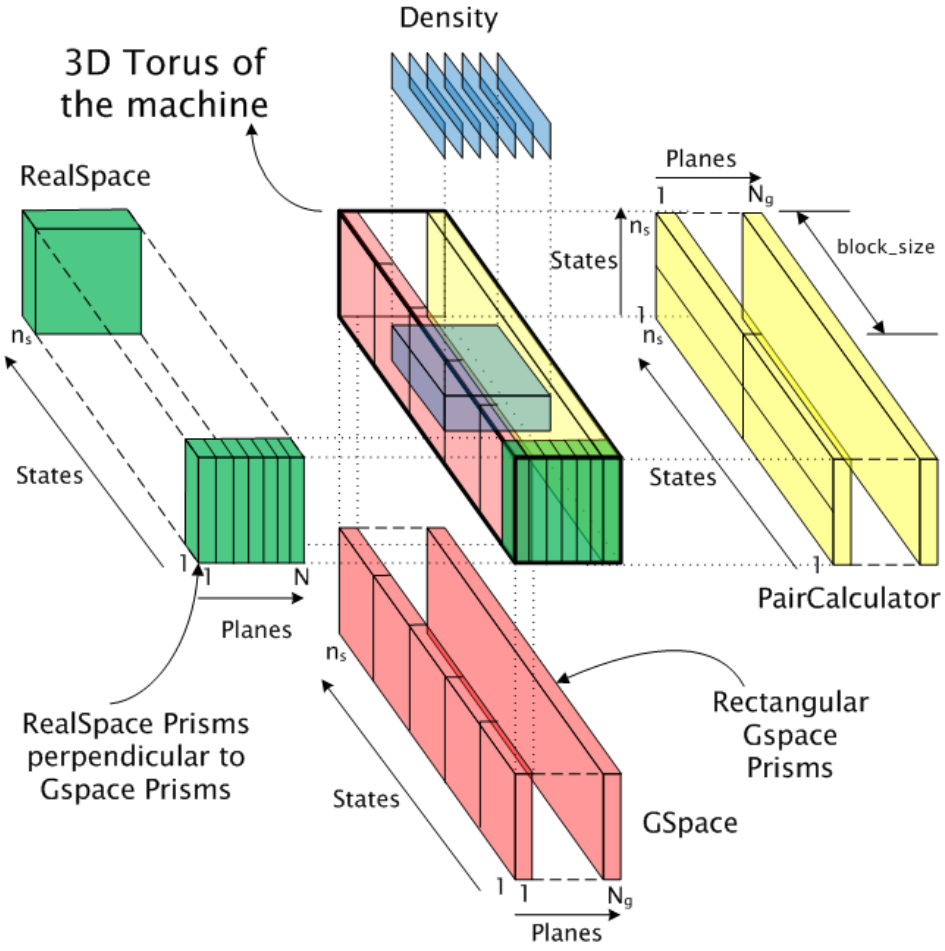
Using Charm++ virtualization, we can efficiently scale small (32 molecule) systems to thousands of processors

PPL UIUC

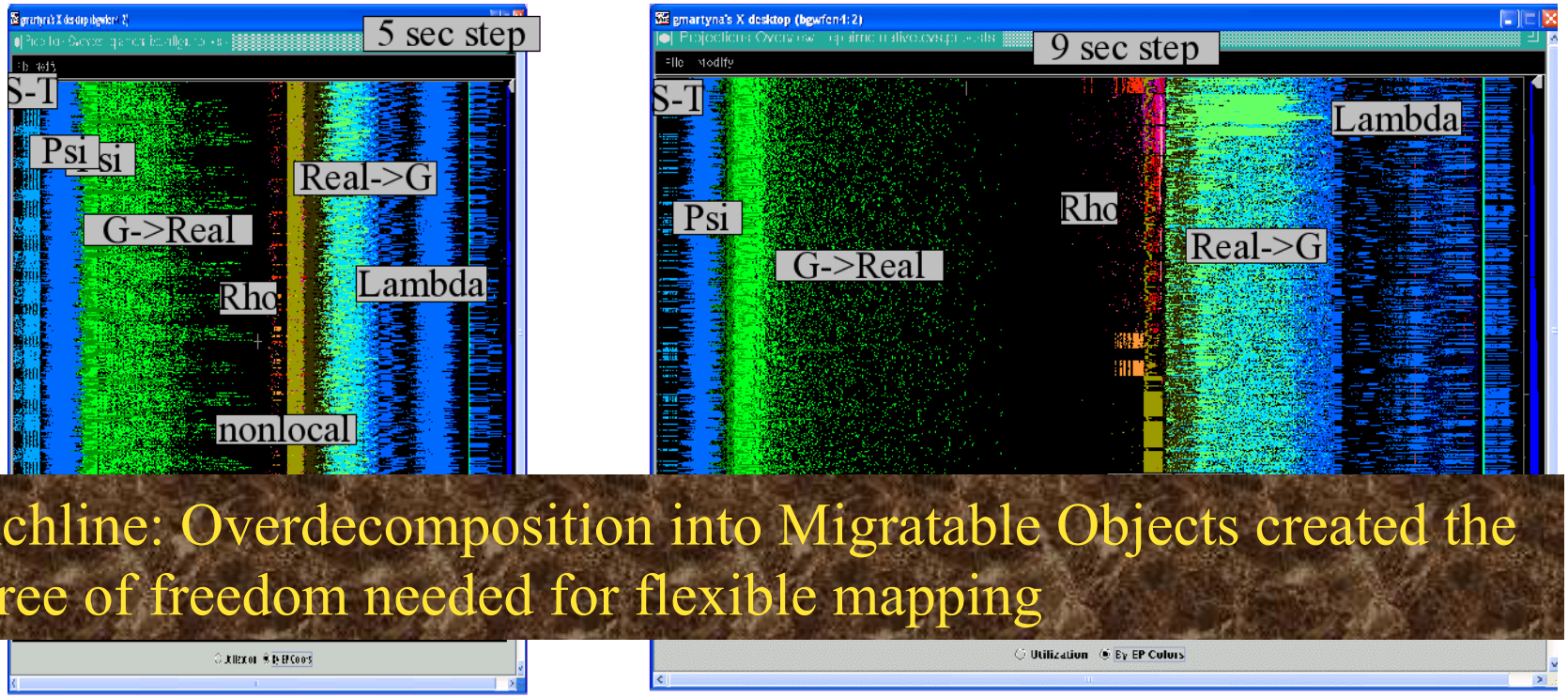
Decomposition and Computation



Topology Aware Mapping of Objects



Improvements by topological aware mapping of computation to processors



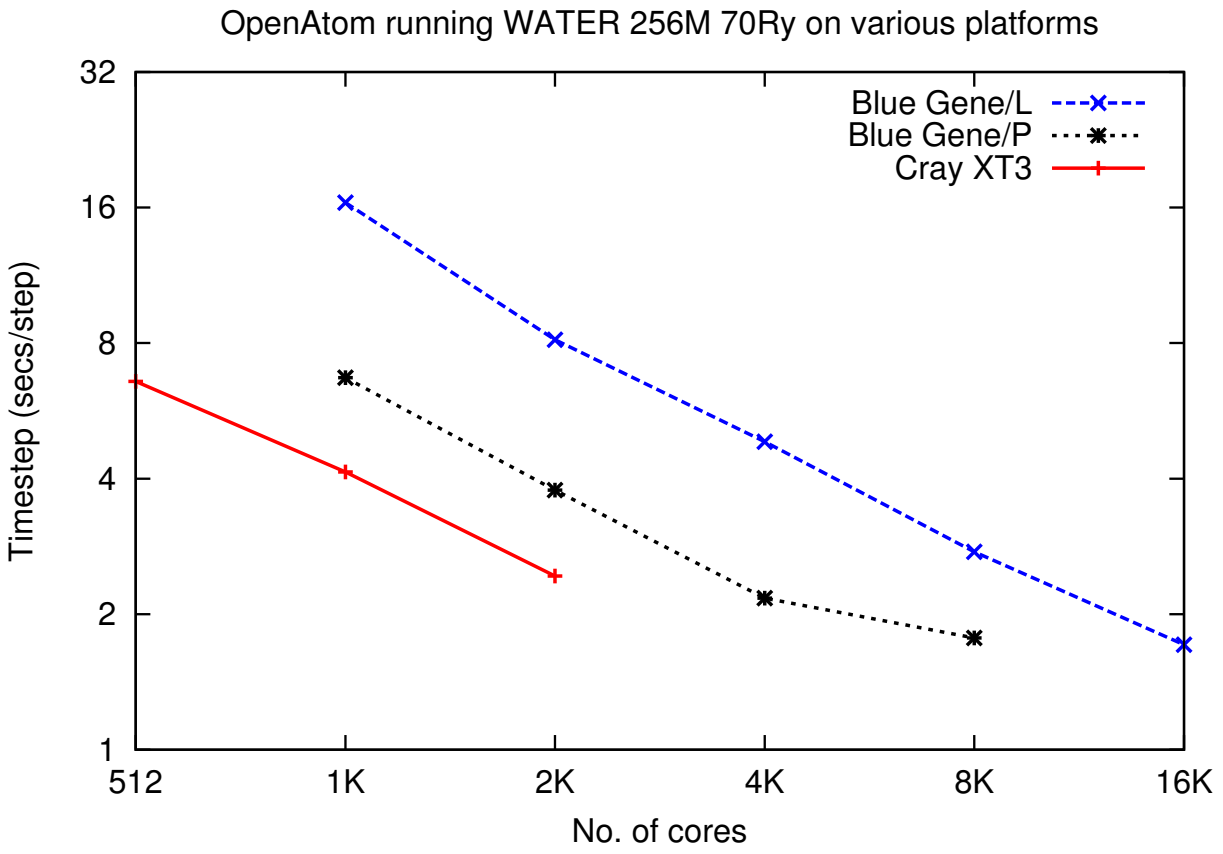
Punchline: Overdecomposition into Migratable Objects created the degree of freedom needed for flexible mapping

The simulation of the right panel, maps computational work to processors taking the network connectivity into account while the left panel simulation does not. The "black" or idle time processors spent waiting for computational work to arrive on processors is significantly reduced at left. (256waters, 70R, on BG/L 4096 cores)

PPL UIUC

OpenAtom Performance Sampler

Ongoing work on:
K-points



Saving Cooling Energy

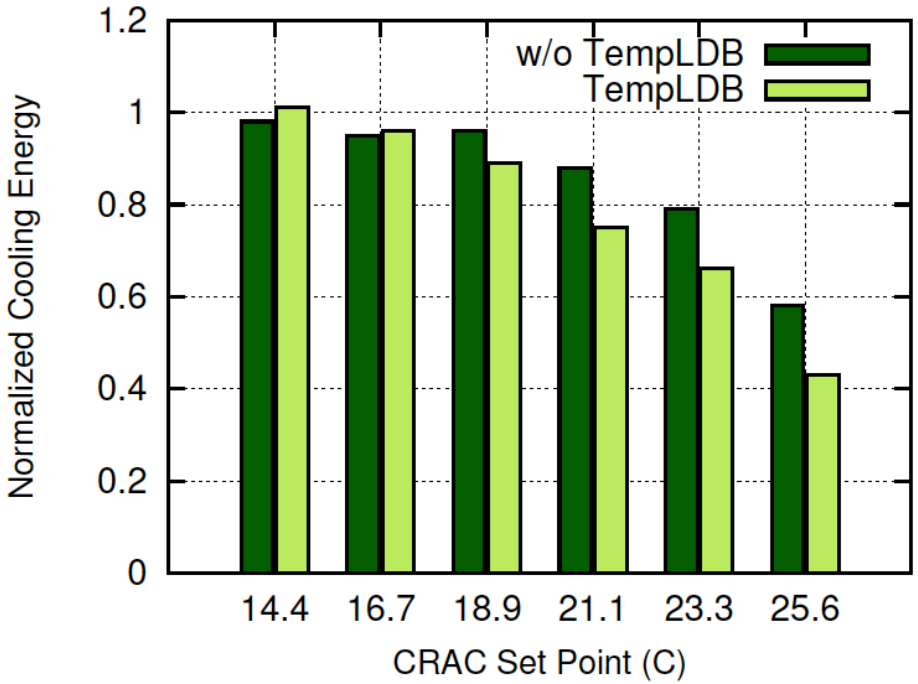
- Some cores/chips might get too hot
  - We want to avoid
    - Running everyone at lower speed,
    - Conservative (expensive) cooling
- Reduce frequency (DVFS) of the hot cores?
  - Works fine for sequential computing
  - In parallel:
    - There are dependences/barriers
    - Slowing one core down by 40% slows the whole computation by 40%!
      - Big loss when the #processors is large

Migratable Objects to the rescue!

Temperature-aware Load Balancing

- Reduce frequency if temperature is high
  - Independently for each core or chip
- Migrate objects away from the slowed-down processors
  - Balance load using an existing strategy
  - Strategies take speed of processors into account
- Recently implemented in experimental version
  - SC 2011 paper

Cooling Energy Consumption

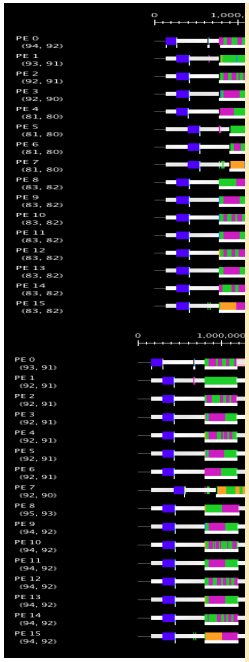


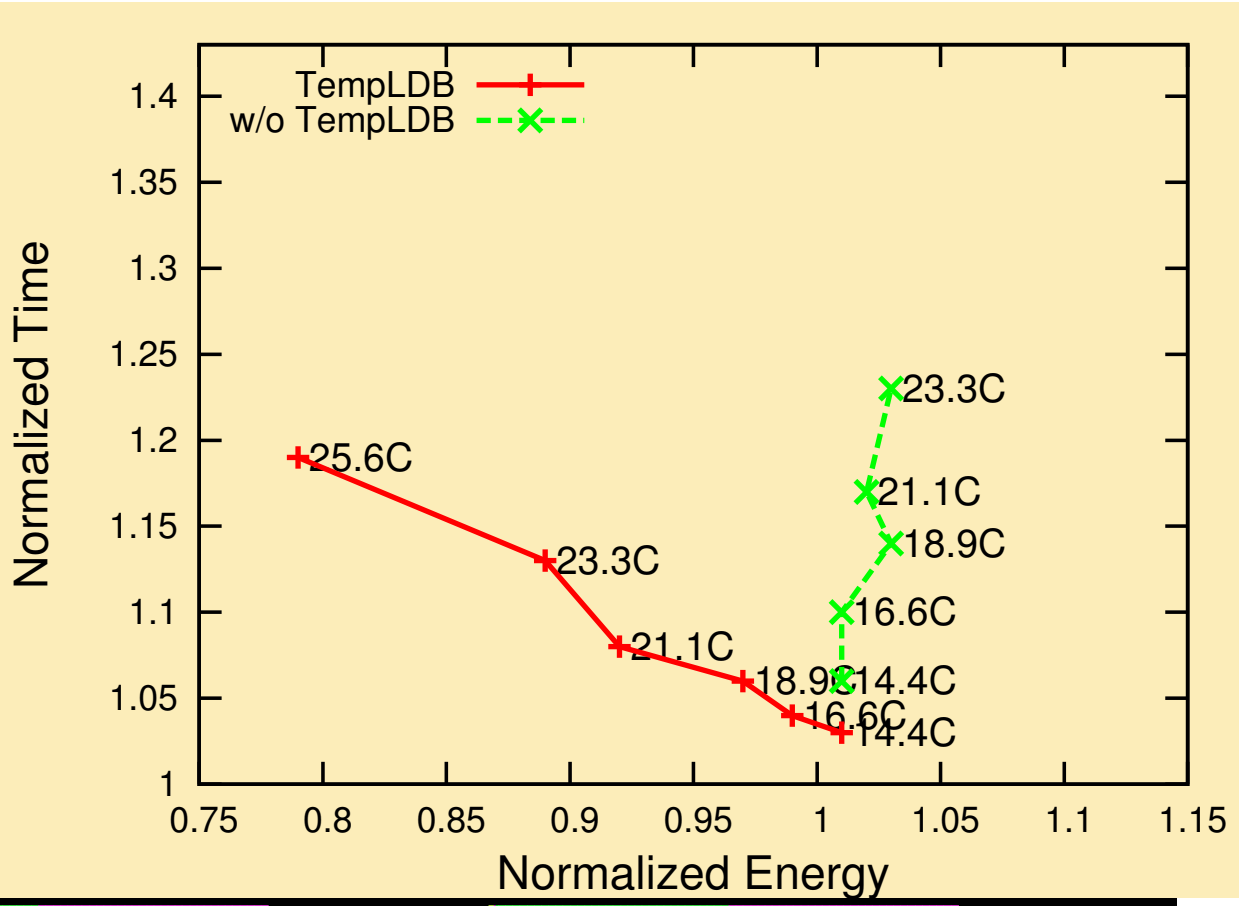
$$c_{norm} = \frac{T_{hot}^{LB} - T_{ac}^{LB}}{T_{hot}^{base} - T_{ac}^{base}} * t_{norm}^{LB}$$

Jacobi2D on 128 Cores

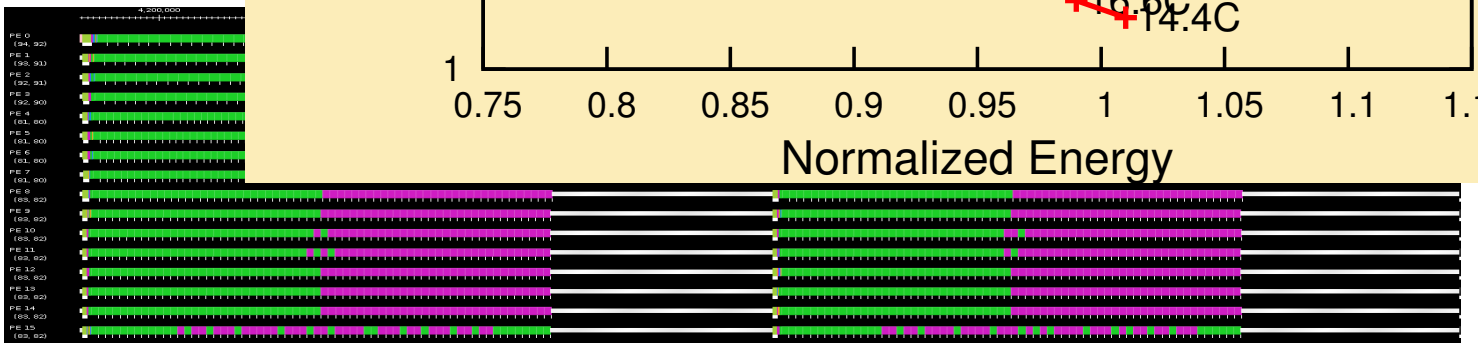
- Both schemes save energy as cooling energy consumption depends on CRAC set-point (TempLDB better)
- Our scheme saves up to 57% (better than w/o TempLDB) mainly due to smaller timing penalty

Benefits of Temperature Aware LB



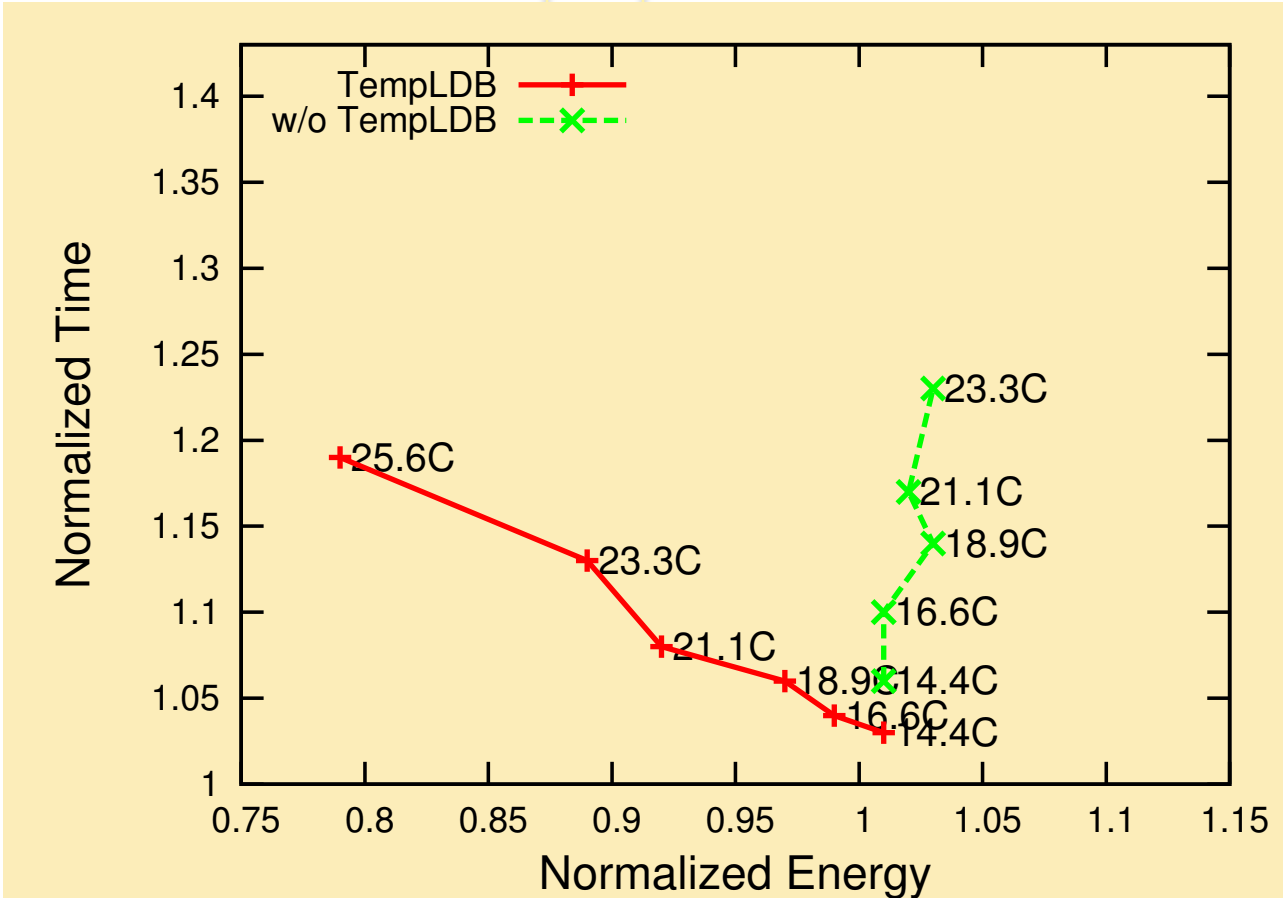


Proje



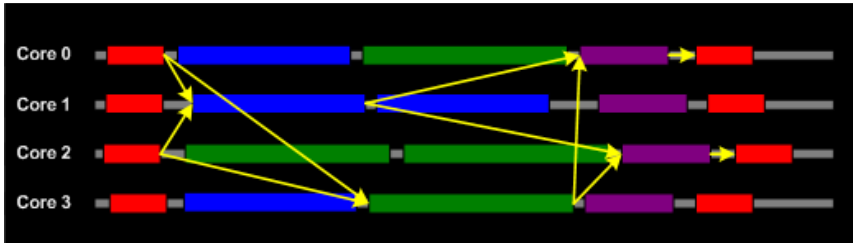
Zoomed projection timeline for two iterations without temperature aware LB

Benefits of Temperature Aware LB



Other Power-related Optimizations

- Other optimizations are in progress:
  - Staying within given energy budget, or power budget
    - Selectively change frequencies so as to minimize impact on finish time
  - Reducing power consumed with low impact on finish time
    - Identify code segments (methods) with high miss-rates
      - Using measurements (principle of persistence)
    - Reduce frequencies for those,
    - and balance load with that assumption
  - Use critical paths analysis:
    - Slow down methods not on critical paths
    - Aggressive: migrate critical-path objects to faster cores



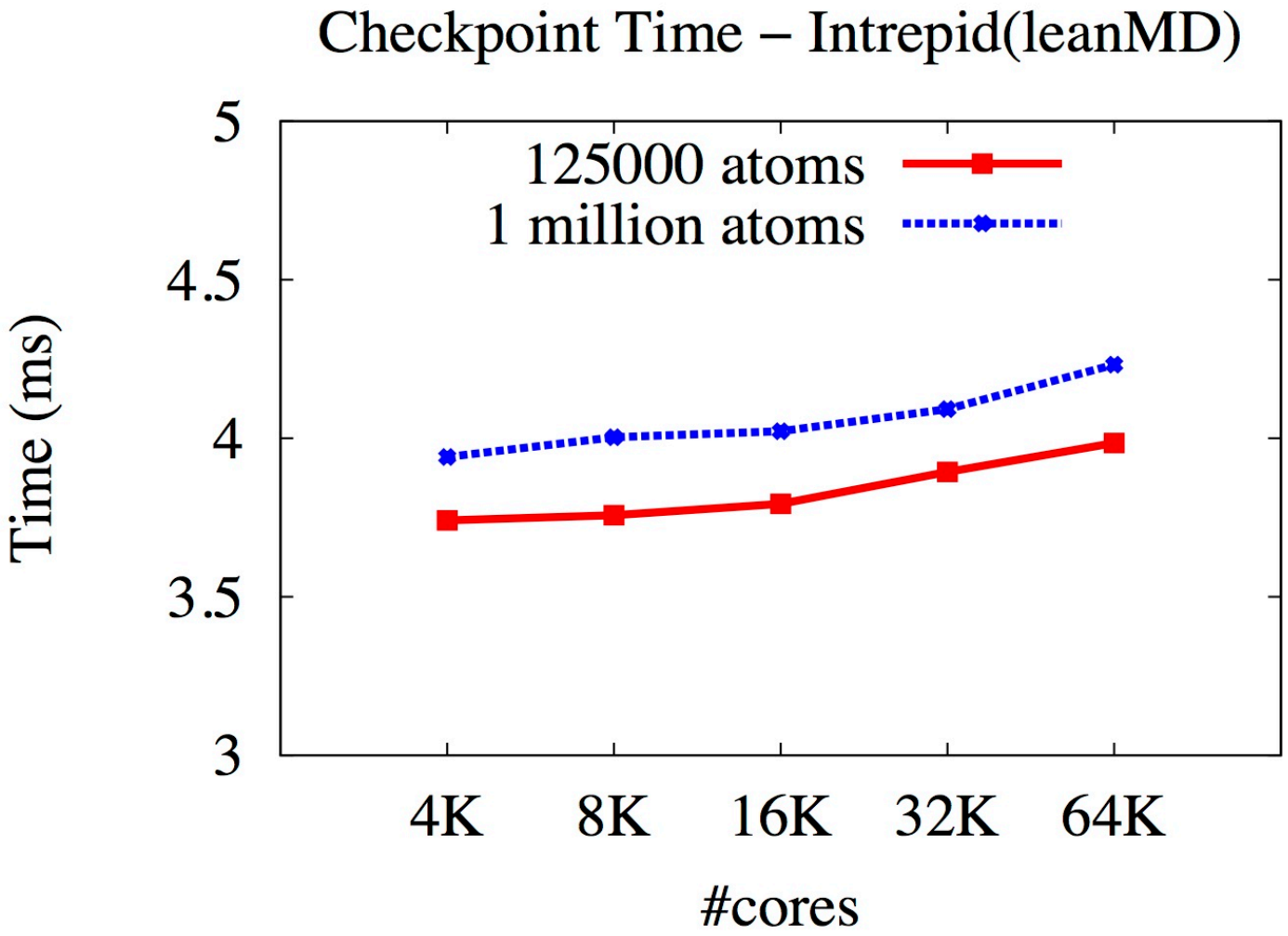
PPL UIUC

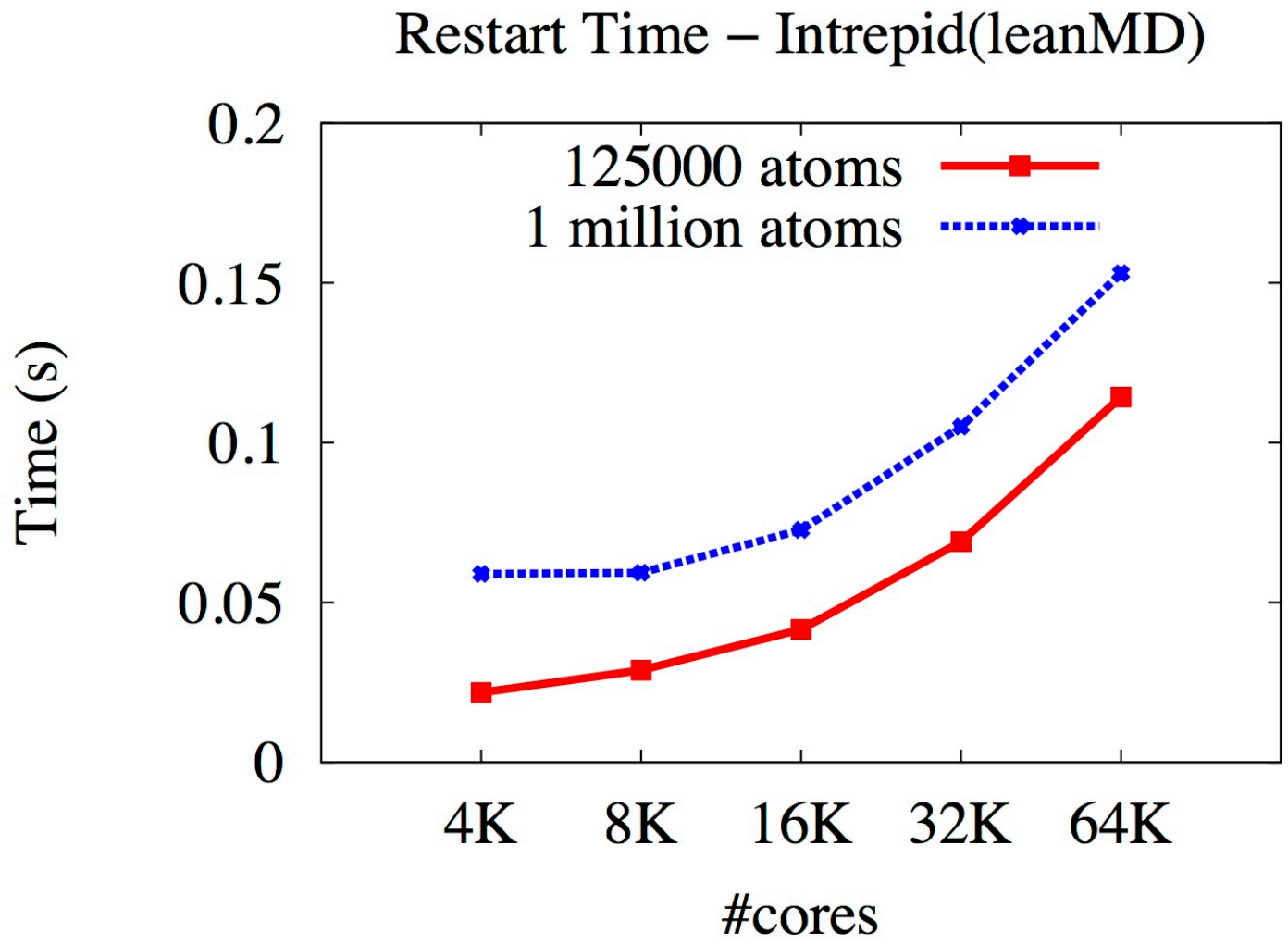
Fault Tolerance in Charm++/AMPI

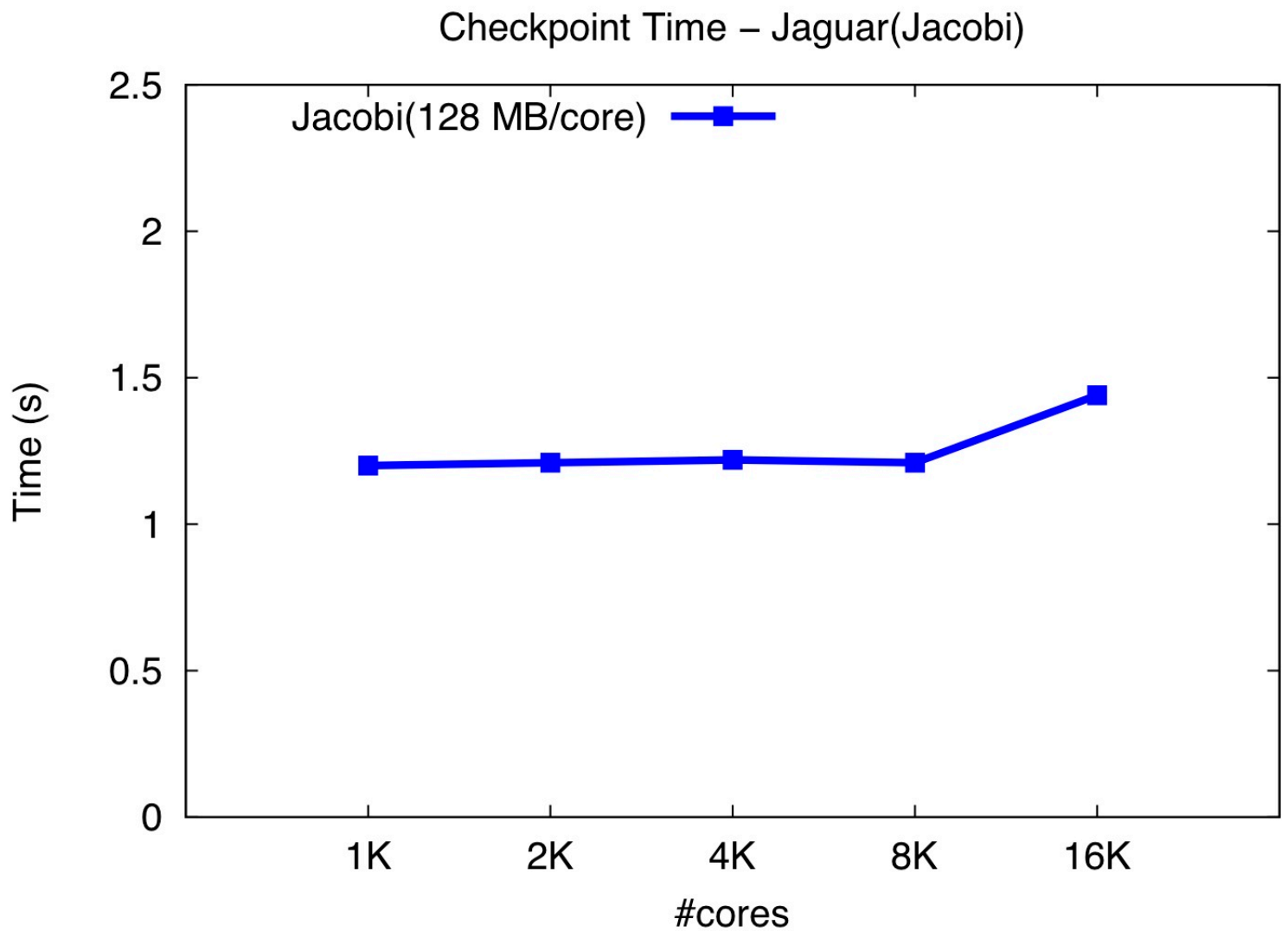
- Four Approaches:
  - Disk-based checkpoint/restart
  - In-memory double checkpoint/restart
  - Proactive object migration
  - Message-logging: scalable fault tolerance
- Common Features:
  - Easy checkpoint:
    - migrate-to-disk leverages object-migration capabilities
  - Based on dynamic runtime capabilities
  - Can be used in concert with load-balancing schemes

In-memory checkpointing

- Is practical for many apps
  - Relatively small footprint at checkpoint time
- Very fast times…
- Demonstration challenge:
  - Works fine for clusters
  - For MPI-based implementations running at centers:
    - Scheduler does not allow job to continue on failure
    - Communication layers not fault tolerant
  - Fault injection: dieNow(),
  - Spare processors


Checkpoint Time – Intrepid(leanMD)
125000 atoms
1 million atoms
Time (ms)
5
4.5
4
3.5
3
4K
8K
16K
32K
64K
#cores


Restart Time – Intrepid(leanMD)
125000 atoms
1 million atoms
Time (s)
0.2
0.15
0.1
0.05
0
4K
8K
16K
32K
64K
#cores


Checkpoint Time – Jaguar(Jacobi)
Jacobi(128 MB/core)
Time (s)
2.5
2
1.5
1
0.5
0
1K
2K
4K
8K
16K
#cores

Scalable Fault tolerance

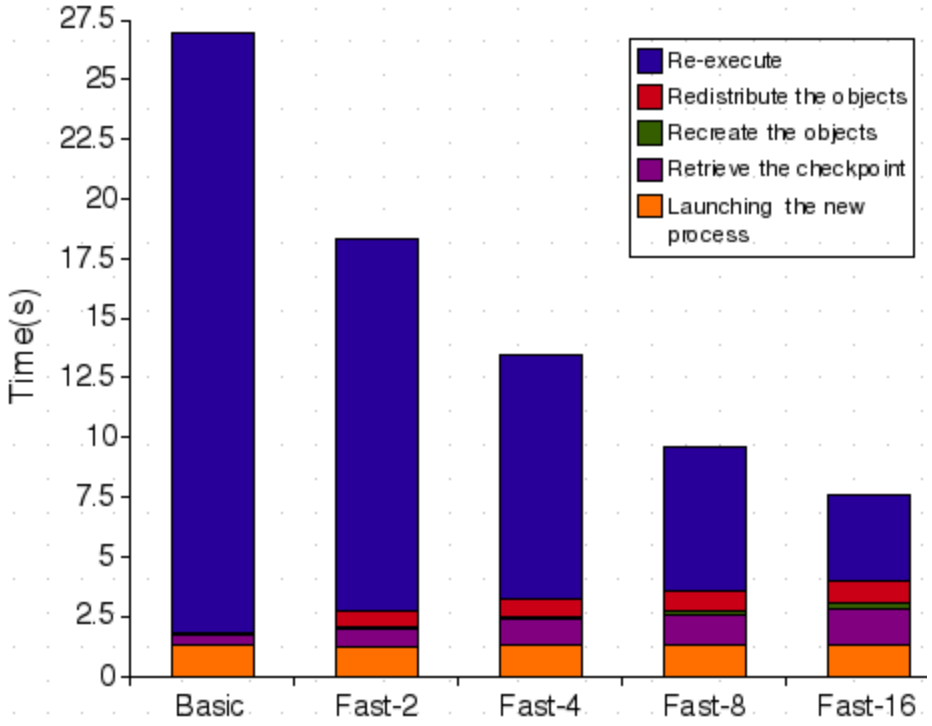
- Faults will be common at exascale
  - Failstop, and soft failures are both important
- Checkpoint-restart may not scale
  - Requires all nodes to roll back even when just one fails
    - Inefficient: computation and power
  - As MTBF goes lower, it becomes infeasible

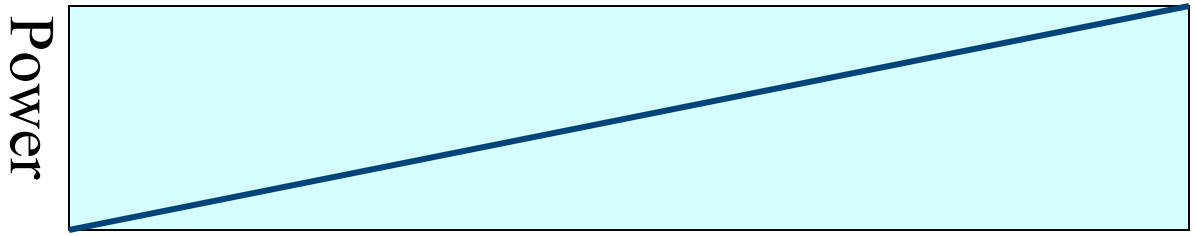
Message-Logging

- Basic Idea:
  - Messages are stored by sender during execution
  - Periodic checkpoints still maintained
  - After a crash, reprocess "resent" messages to regain state
- Does it help at exascale?
  - Not really, or only a bit: Same time for recovery!
- With virtualization,
  - work in one processor is divided across multiple virtual processors; thus, *restart can be parallelized*
  - Virtualization helps fault-free case as well

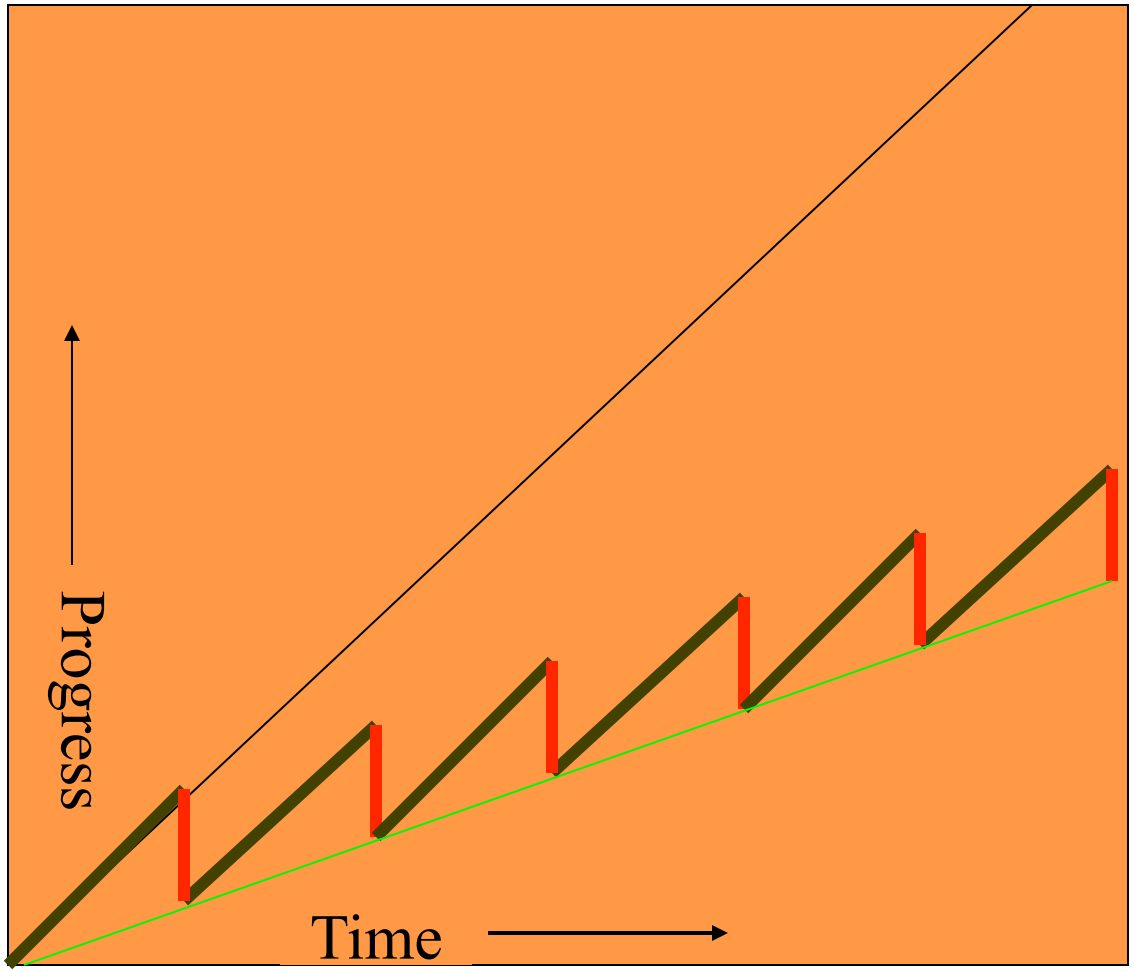
Message-Logging (cont.)

- Fast Parallel restart performance:
  - Test: 7-point 3D-stencil in MPI, P=32, 2 ≤ VP ≤ 16
  - Checkpoint taken every 30s, failure inserted at t=27s



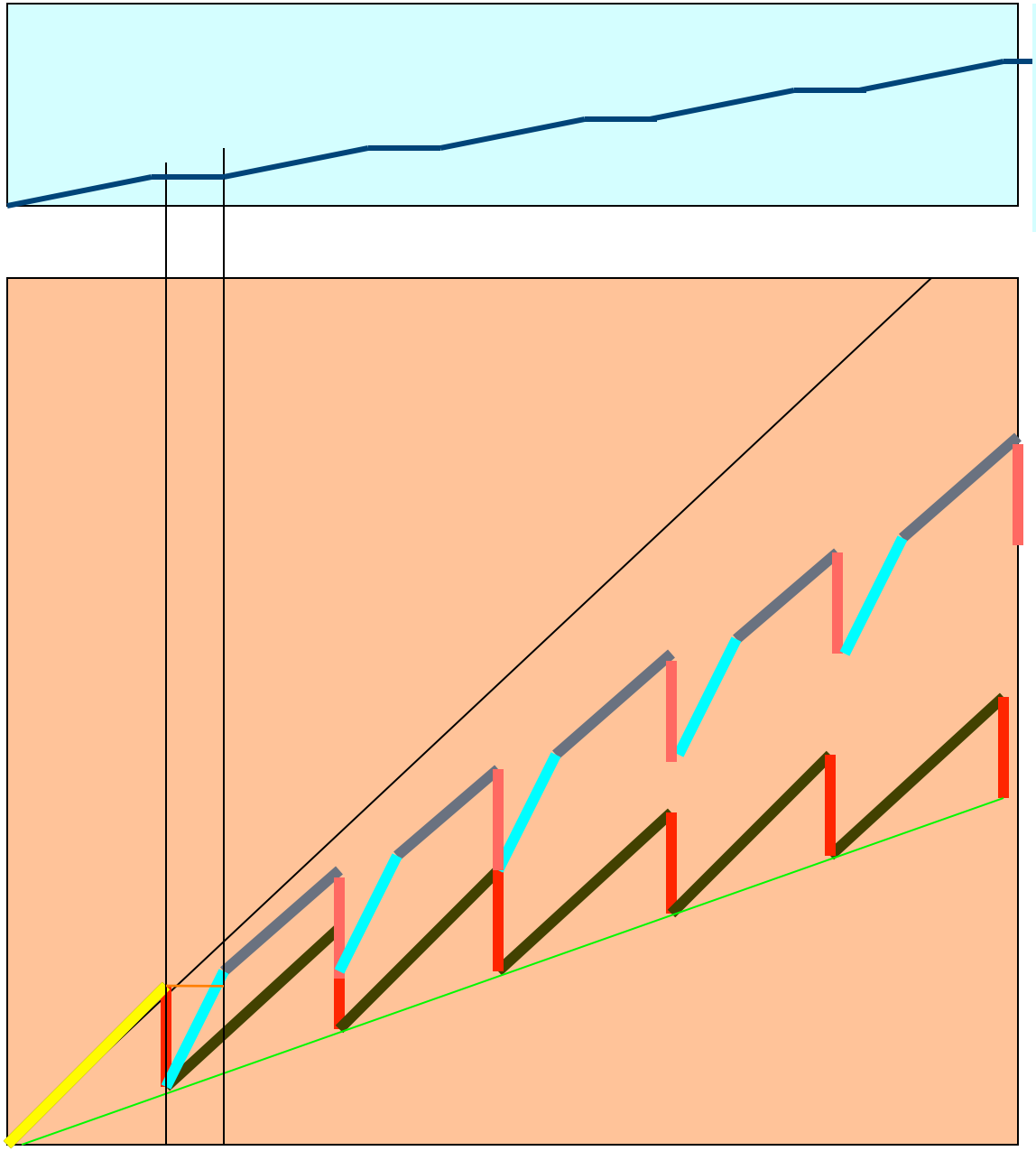


Power consumption is continuous



Normal Checkpoint-Resart method

Progress is slowed down with failures



Power consumption is lower during recovery

Message logging + Object-based virtualization

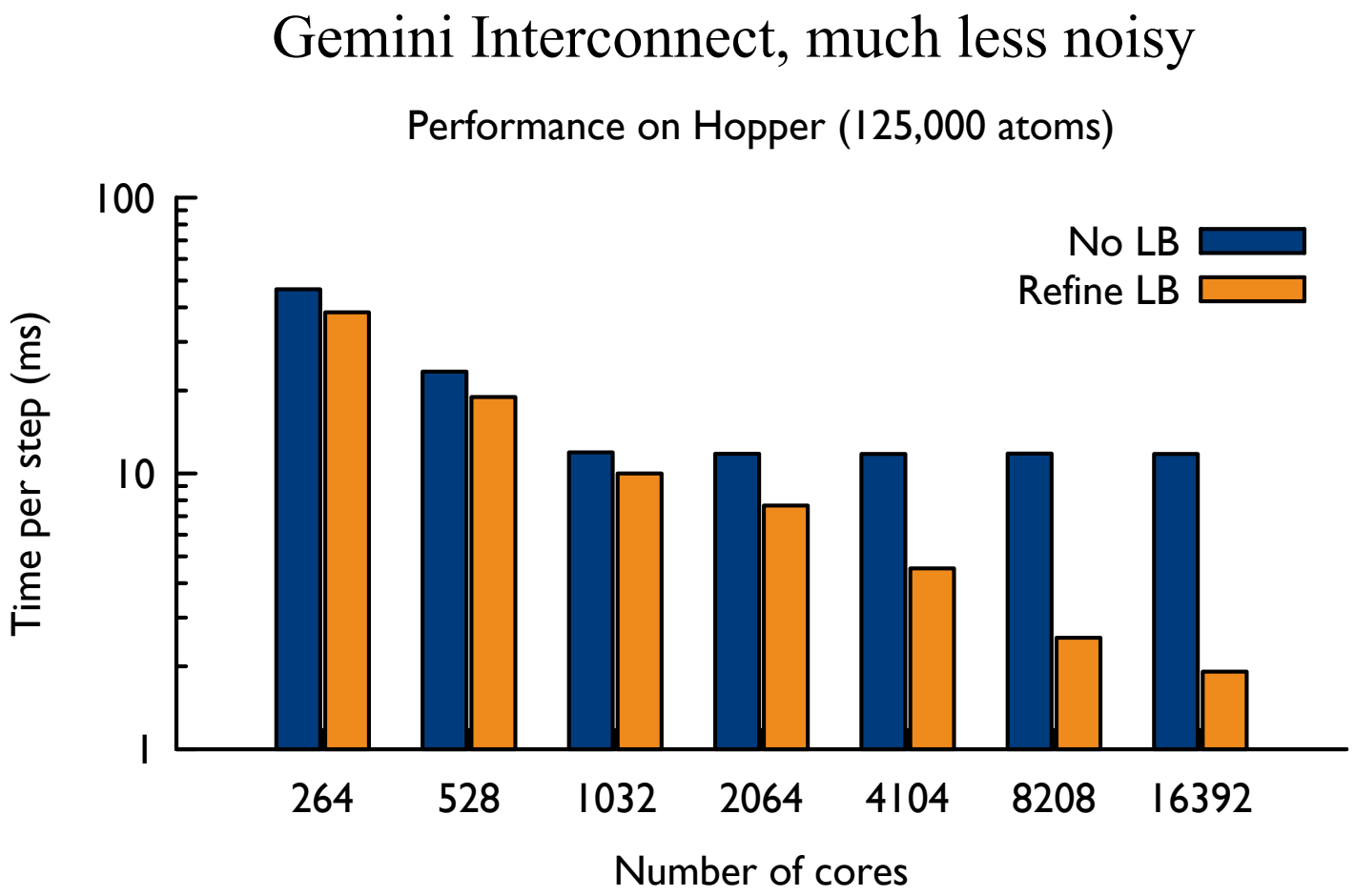
Progress is faster with failures

PPL UIUC

HPC Challenge Competition

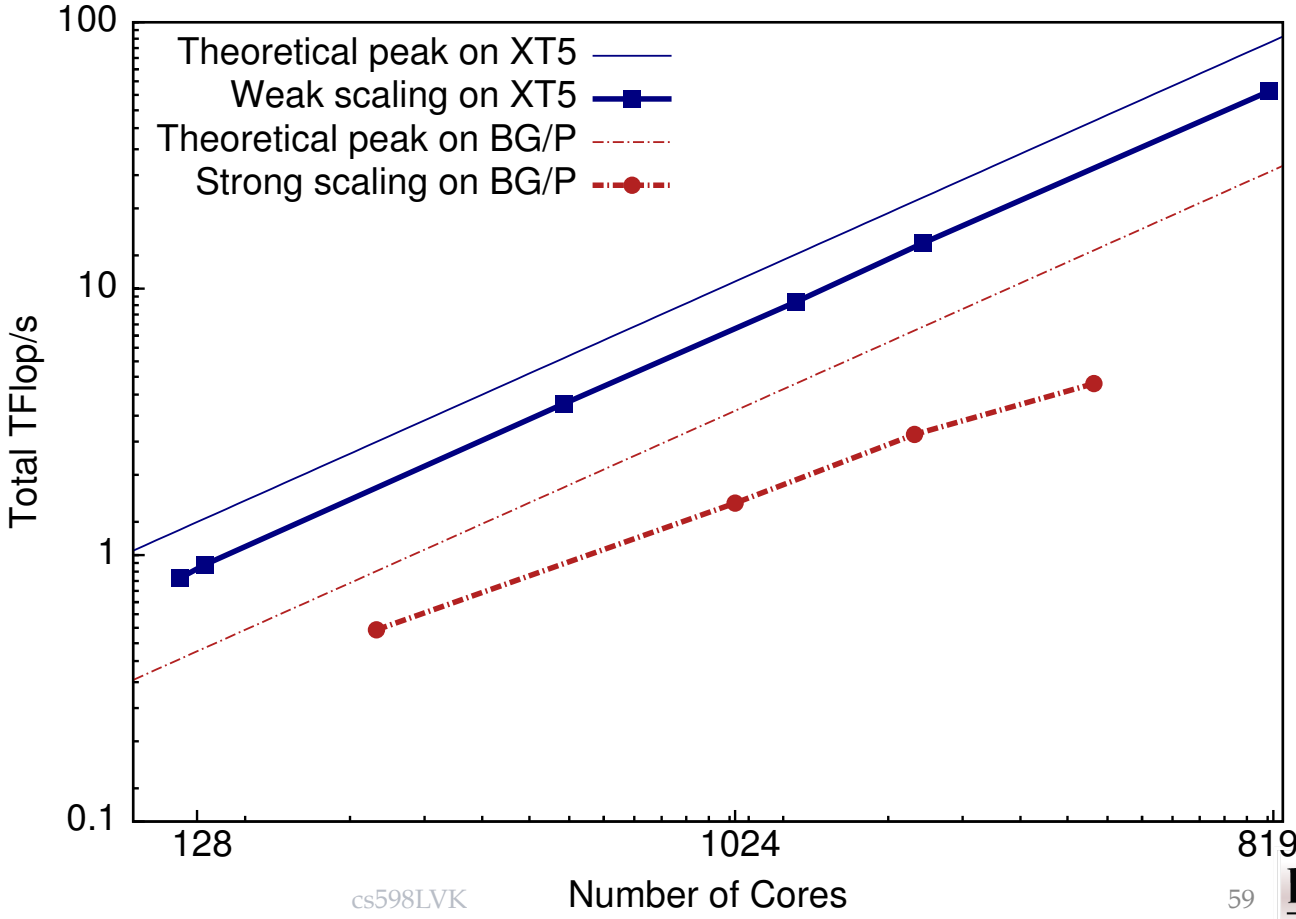
- Conducted at Supercomputing
- 2 parts:
  - Class I: machine performance
  - Class II: programming model productivity
    - Has been typically split in two sub-awards
  - We implemented in Charm++
    - LU decomposition
    - RandomAccess
    - LeanMD
    - Barnes-Hut
- Main competitors this year:
  - Chapel (Cray), CAF (Rice), and Charm++ (UIUC)

Strong Scaling on Hopper for LeanMD



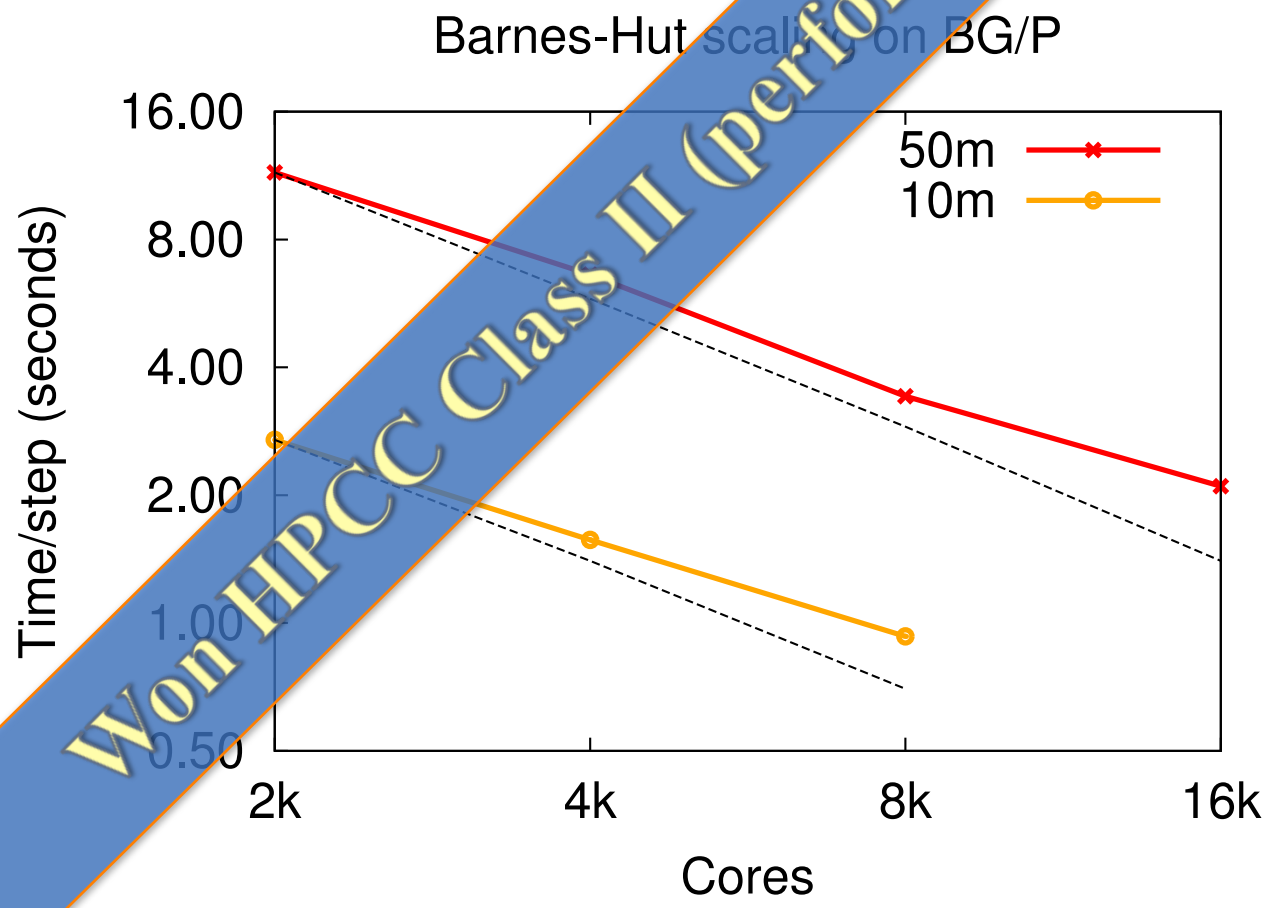
CharmLU: productivity and performance

- 1650 lines of source
- 67% of peak on Jaguar



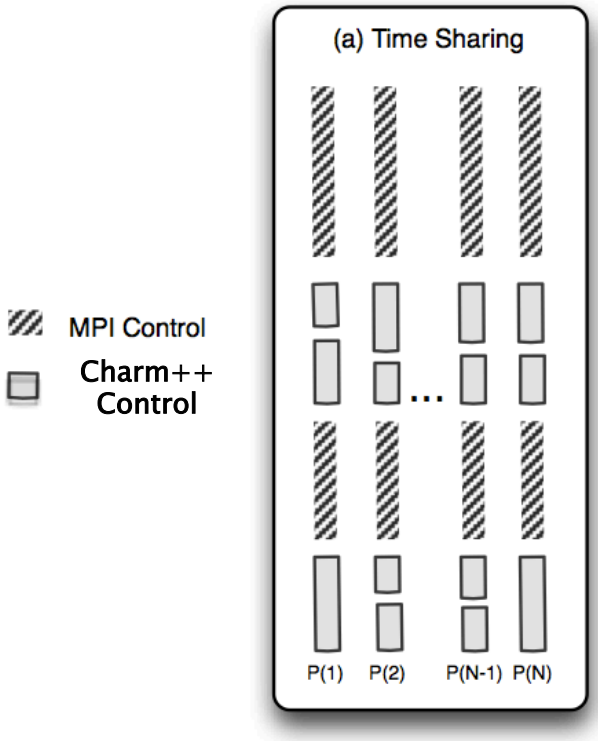
Barnes-Hut

High Density Variation with a *Plummer* distribution of particles

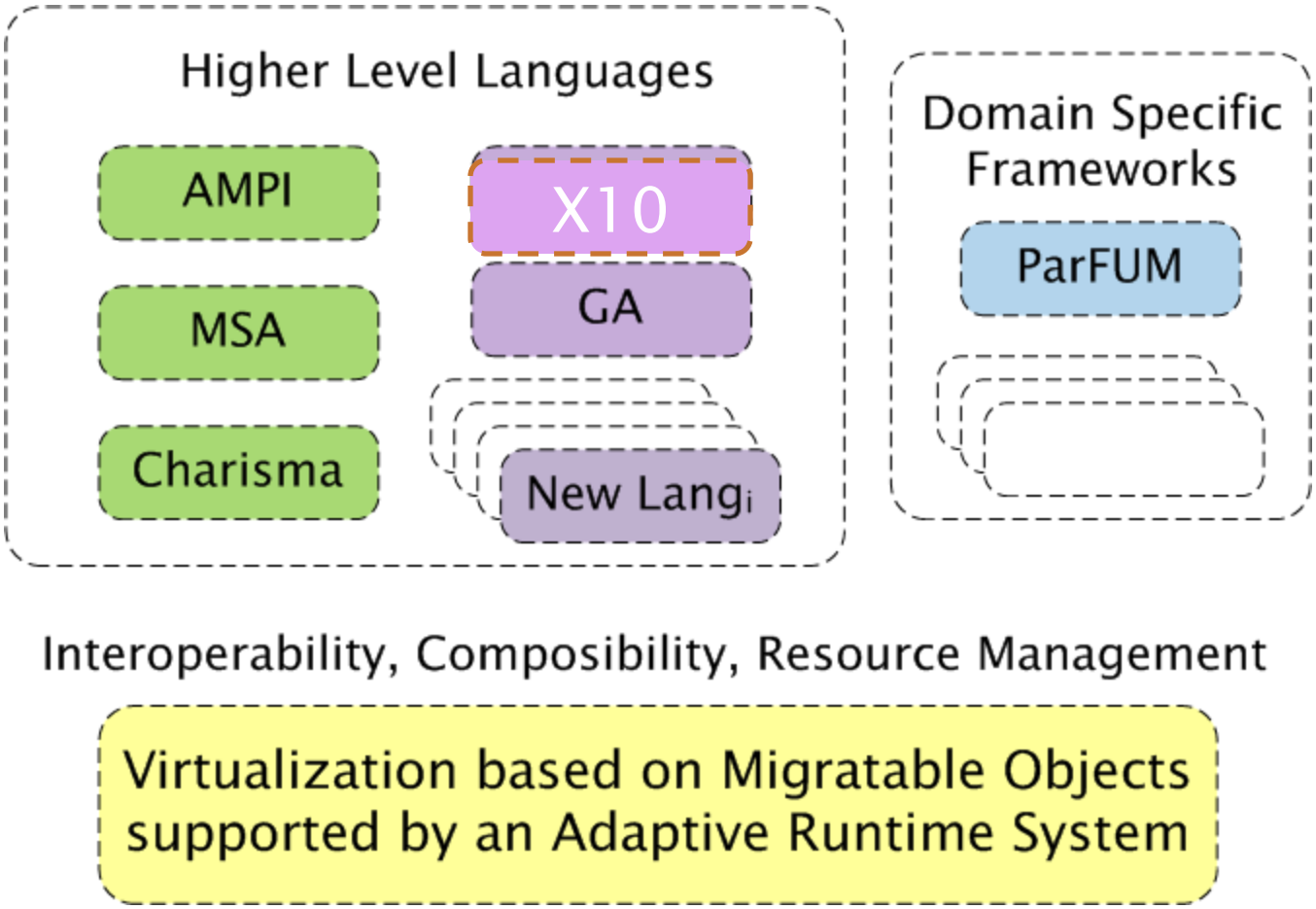


Won HPCC Class II (performance)

Charm++ interoperates with MPI



A View of an Interoperable Future



Interoperability allows faster evolution of programming models

Evolution doesn't lead to a single winner species, but to a stable and effective ecosystem.

Similarly, we will get to a collection of viable programming models that co-exists well together.

Summary

- Do away with the notion of processors
  - Adaptive Runtimes, enabled by migratable-objects programming model
    - Are necessary at extreme scale
    - Need to become more intelligent and introspective
    - Help manage accelerators, balance load, tolerate faults,
- Interoperability, concurrent composition become even more important
  - Supported by Migratable Objects and message-driven execution
- Charm++ is production-quality and ready for your application!
  - You can interoperate with Charm++, AMPI, MPI and OpenMP modules
- New programming models and frameworks
  - Create an ecosystem/toolbox of programming paradigms rather than one "super" language

More Info: http://charm.cs.illinois.edu/