Composable and modular Exascale Programming Models with intelligent runtime systems: To Virtualize or Not?! Of course, virtualize

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
PPL UIUC

Observations: exascale machines

- Just restating, with a bit of my take added
- Many (1000+) cores in a “node”
- Heterogeneous cores:
  - specialization saves energy
  - Possibly reconfigurable hardware
- Main reason for accelerators:
  - “cache” idea had outlived its utility
  - So: explicit control over data movement
    - Scratchpad memories a la Cell, GPGPU, ..
  - Hardware context switches for tolerating latency
- Communication challenges: variable speeds?

Application Segmentation

- We may have to specialize architectures to classes of applications
  - Two dimensions: memory-per-core, bisection bandwidth
  - Of the 4 quadrants formed, more than 1 are populated by real apps, I think
  - We can design *very* different machines for each class
    - E.g. For many apps we may need to go to a machine design with (say) no external DRAM. Use all the pins for communication.., and say use a simple grid network.
- We need a serious study of applications
  - Emphasizing exascale problem instances
  - Use something like BigSim to do parametric studies to quantify needs of application

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
- 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
  - Our 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, ..

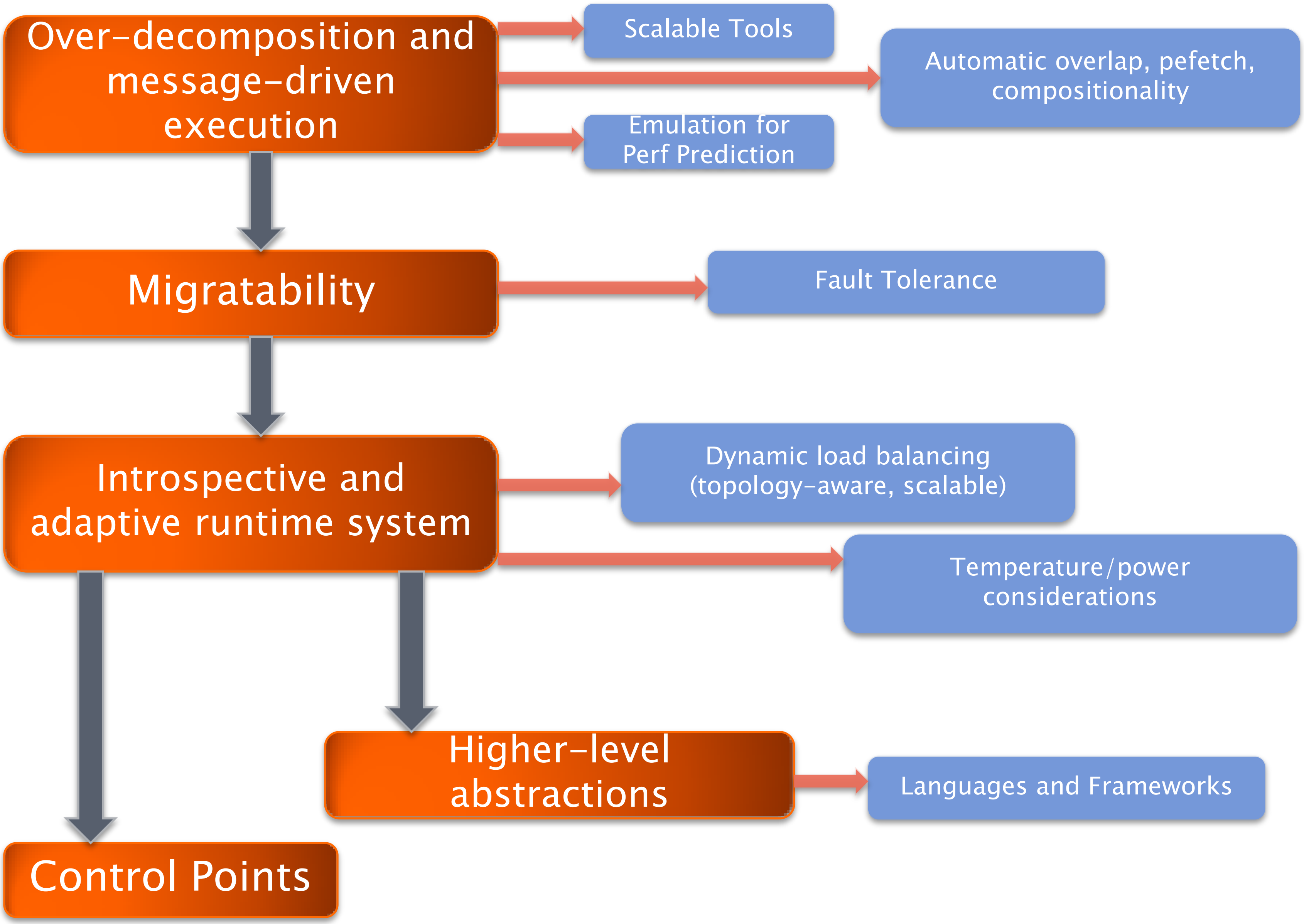
Migratable objects programming model

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

Empower Adaptive Runtime System

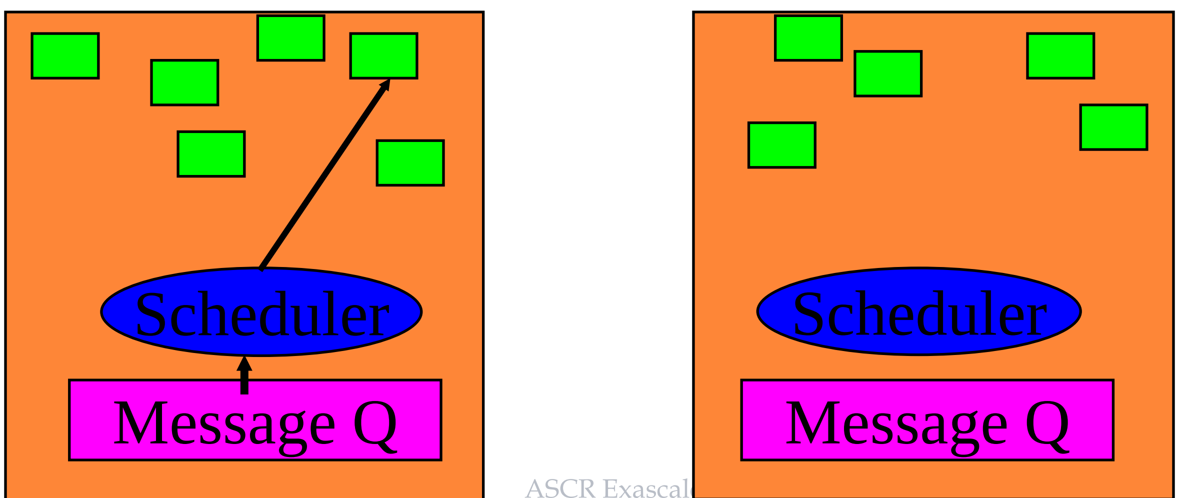
- Decomposing program into a large number of WUDUs empowers the RTS, which can:
  - Migrate WUDUs at will
  - Schedule DEBS at will
  - Instrument computation and communication at the level of these logical units
    - WUDU x communicates y bytes to WUDU z every iteration
    - SEB A has a high cache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

PPL UIUC


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

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



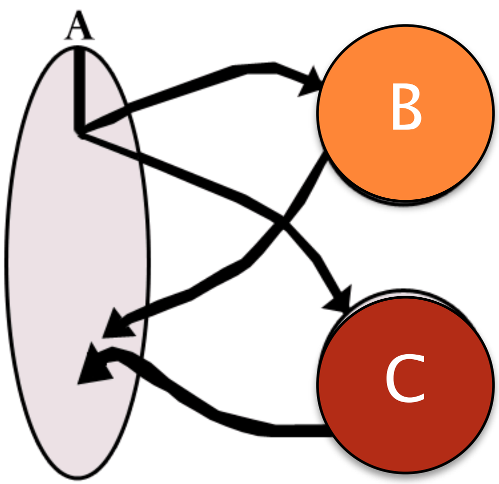
Impact on communication

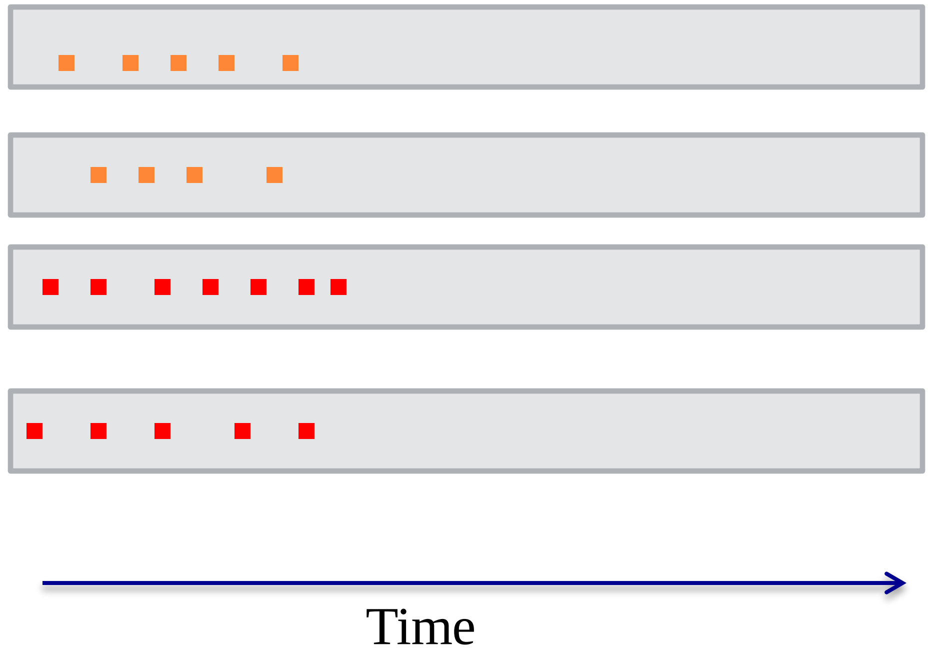
- Current machines are over-engineered for communication by necessity:
  - Compute-communicate cycles in typical MPI apps
  - So, the network is used for a fraction of time,
  - and is on the critical path
- With overdecomposition (virtualization)
  - Communication is spread over an iteration
  - Also,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

Compositionality

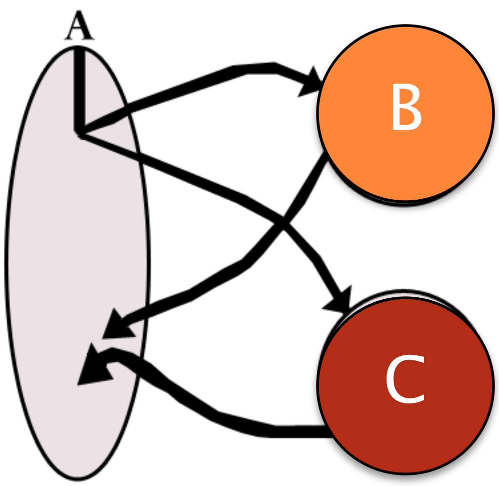
- It is important to support parallel composition
  - For multi-module, multi-physics, multi-paradigm applications..
- What I mean by parallel composition
  - B || C where B and C are independently developed modules
  - B is parallel module by itself, and so is C
  - Programmers who wrote B were unaware of 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 :

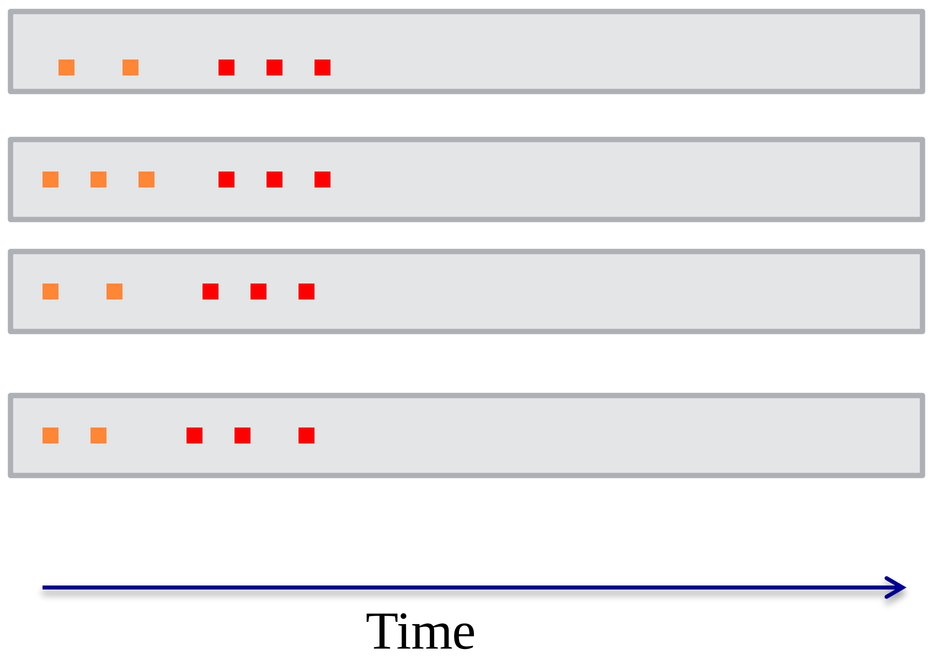
Without message-driven execution (and virtualization), you get either:
Space-division



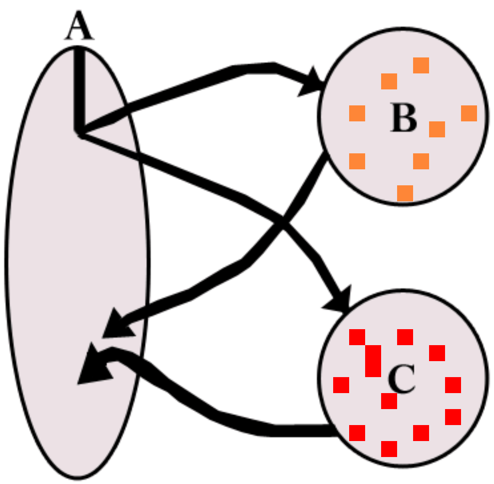


OR: Sequentialization





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

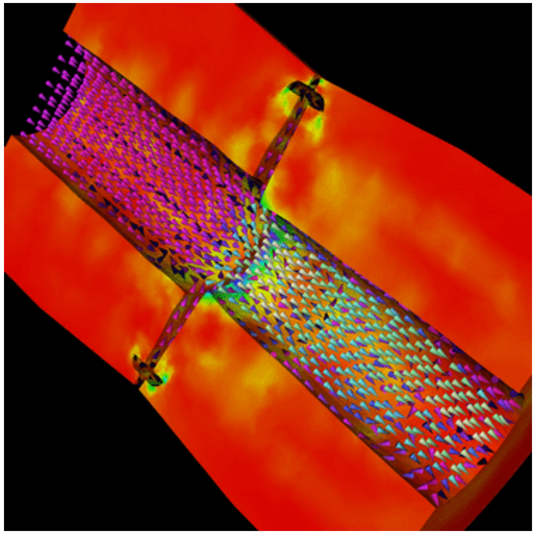


Recall: Different modules, written in different languages/paradigms, can overlap in time and on processors, without programmer having to worry about this explicitly

Decomposition Independent of numCores

- Rocket simulation example under traditional MPI

1	2	. . .	P
Solid	Solid		Solid
Fluid	Fluid		Fluid



- With migratable-objects:

Solid_1 Solid_2 Solid_3 . . . Solid_n

Fluid_1 Fluid_2 . . . Fluid_m

  – Benefit: load balance, communication optimizations, modularity

Load Balancing

- Static
  - Irregular applications
  - Programmer shouldn't have to figure out ideal mapping
- Dynamic:
  - Applications are increasingly using adaptive strategies
  - Abrupt refinements
  - Continuous migration of work: e.g. particles in MD
- Challenges:
  - Performance limited by most overloaded processor
  - The chance that one processor is severely overloaded gets higher as #processors increases

Migratable Objects Empower Automated Load Balancing!

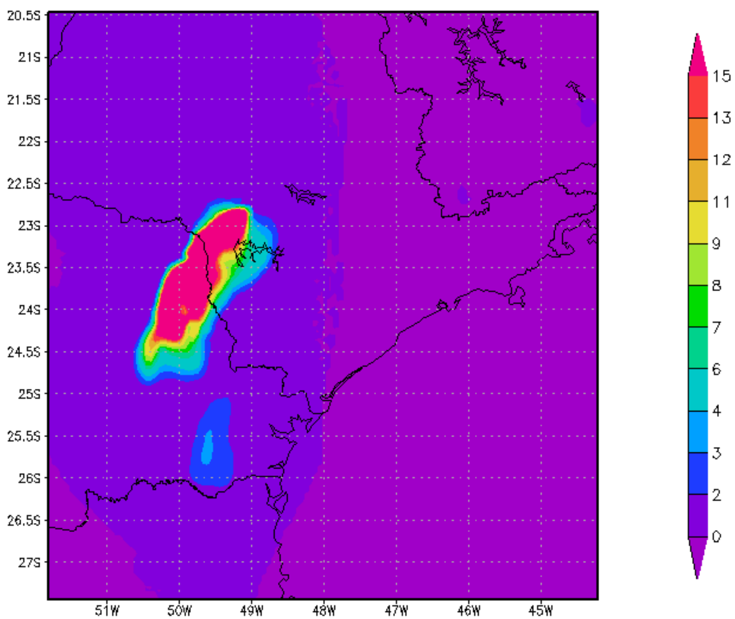
Principle of Persistence

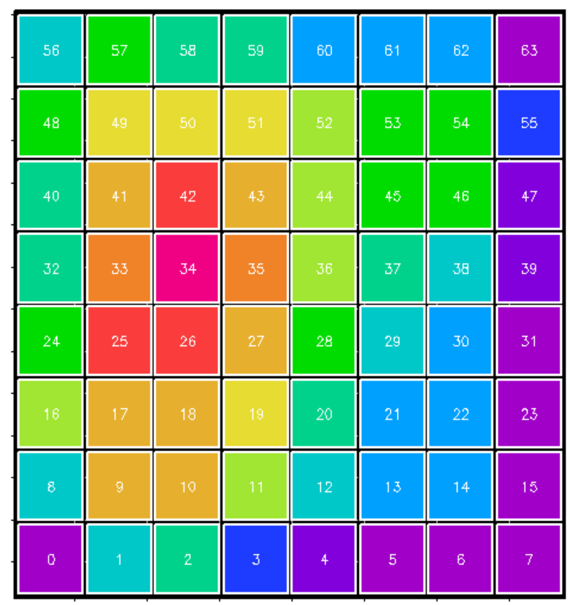
- Once the computation is expressed in terms of its natural (migratable) objects
- *Computational loads and communication patterns tend to persist, even in dynamic computations*
- So, recent past is a good predictor of near future

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

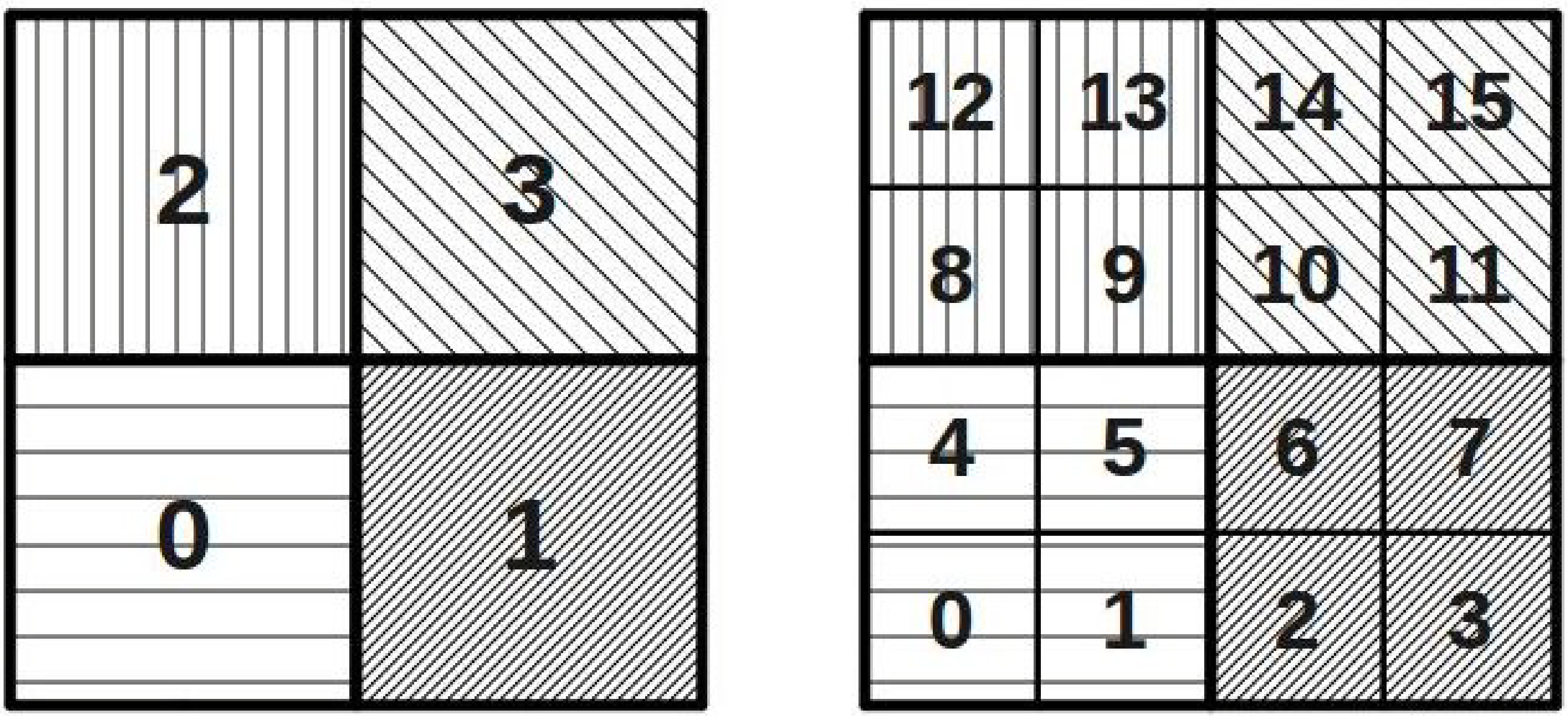
A quick Example: Weather Forecasting in BRAMS

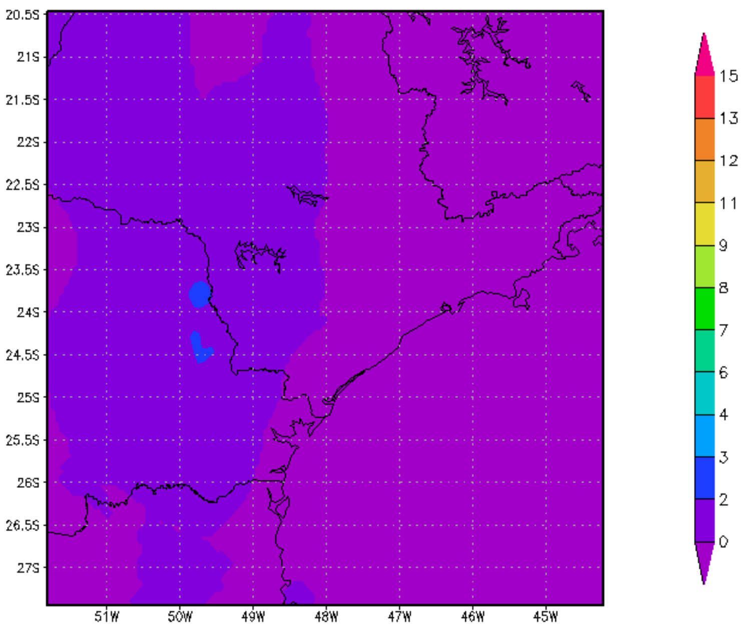
- Brams: Brazillian weather code (based on RAMS)
- AMPI version (Eduardo Rodrigues, with Mendes and J. Panetta)

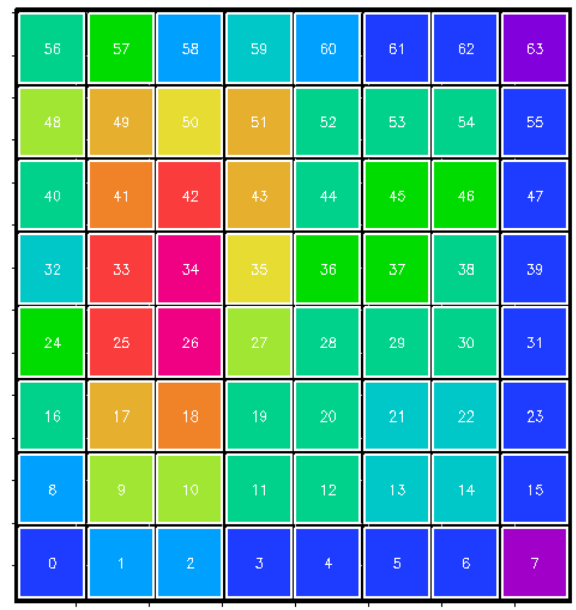




Basic Virtualzation of BRAMS

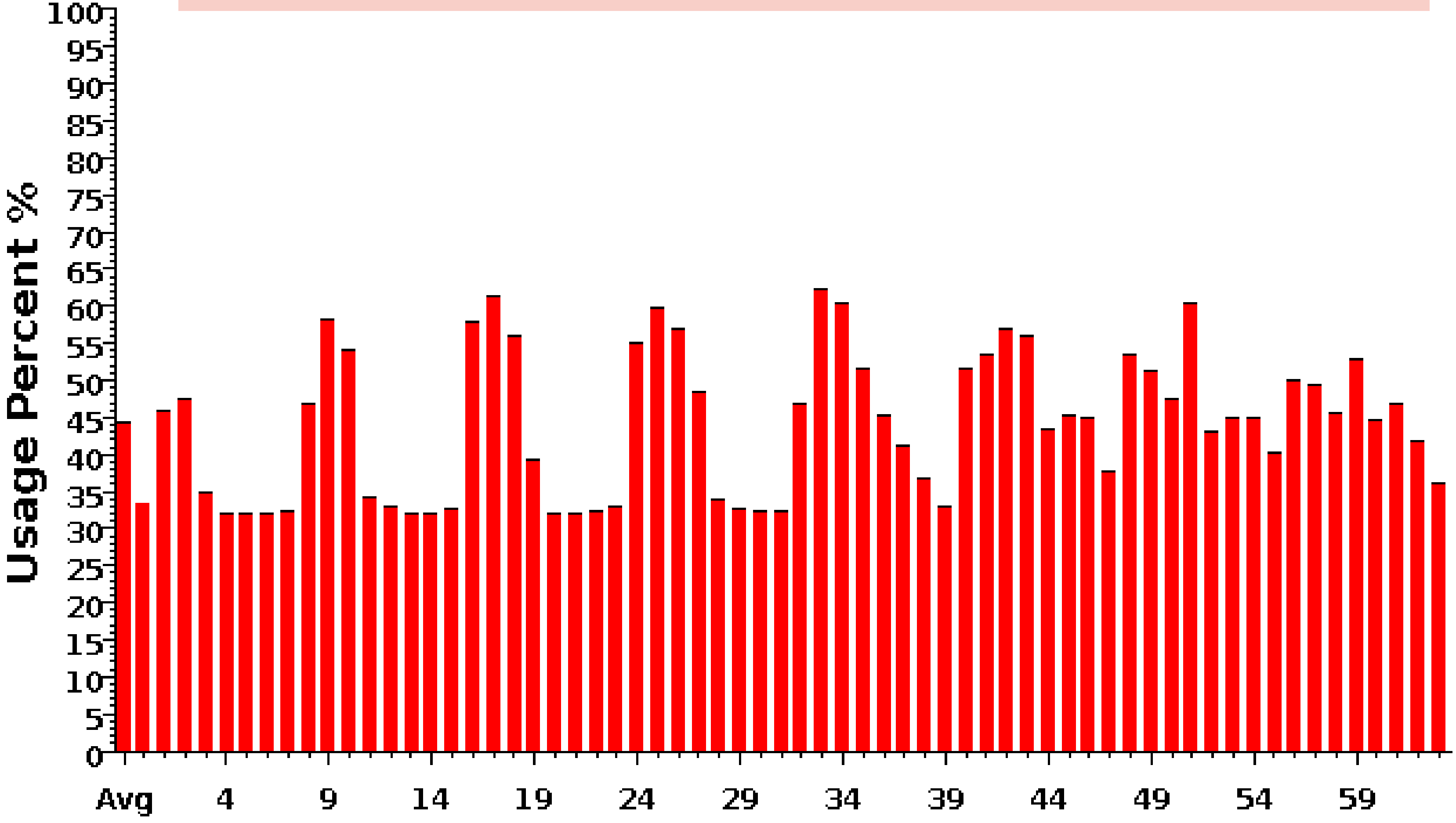




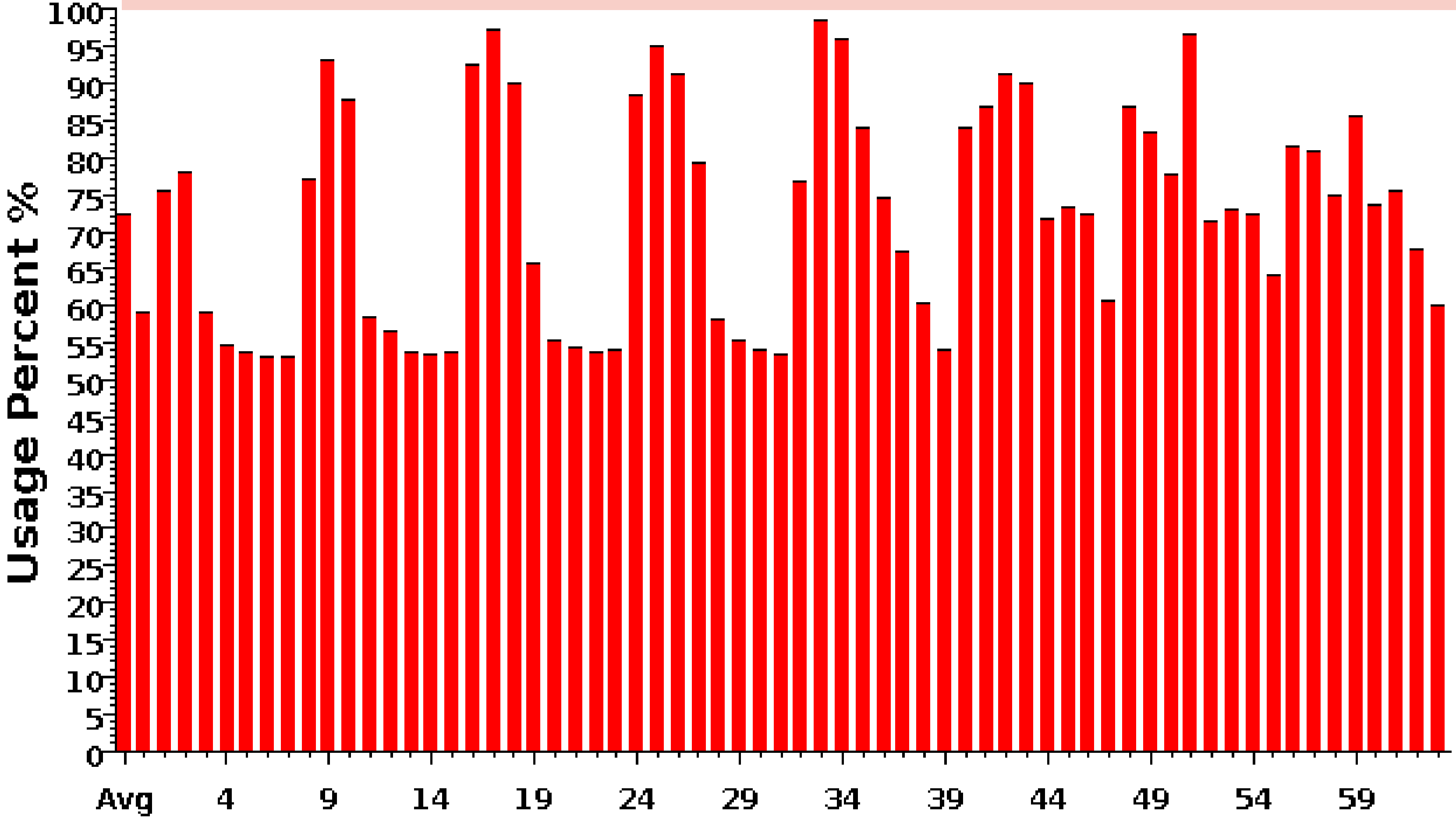


GrADS: COLA/IGES 2010-02-18-09:46 GrADS: COLA/IGES 2010-02-18-10:00

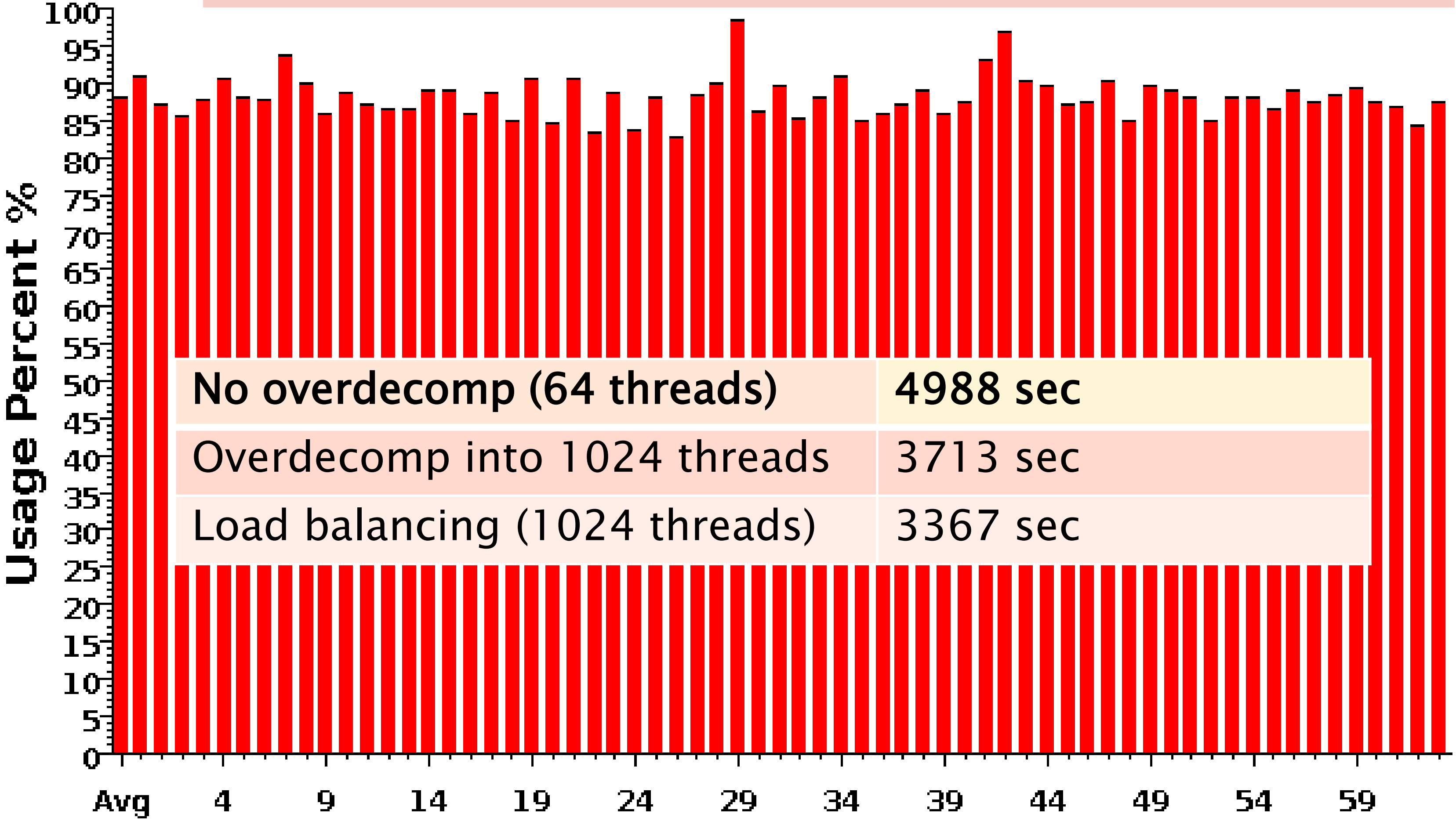
Baseline: 64 objects on 64 processors



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



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

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

Dealing with Thermal Variation

- 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

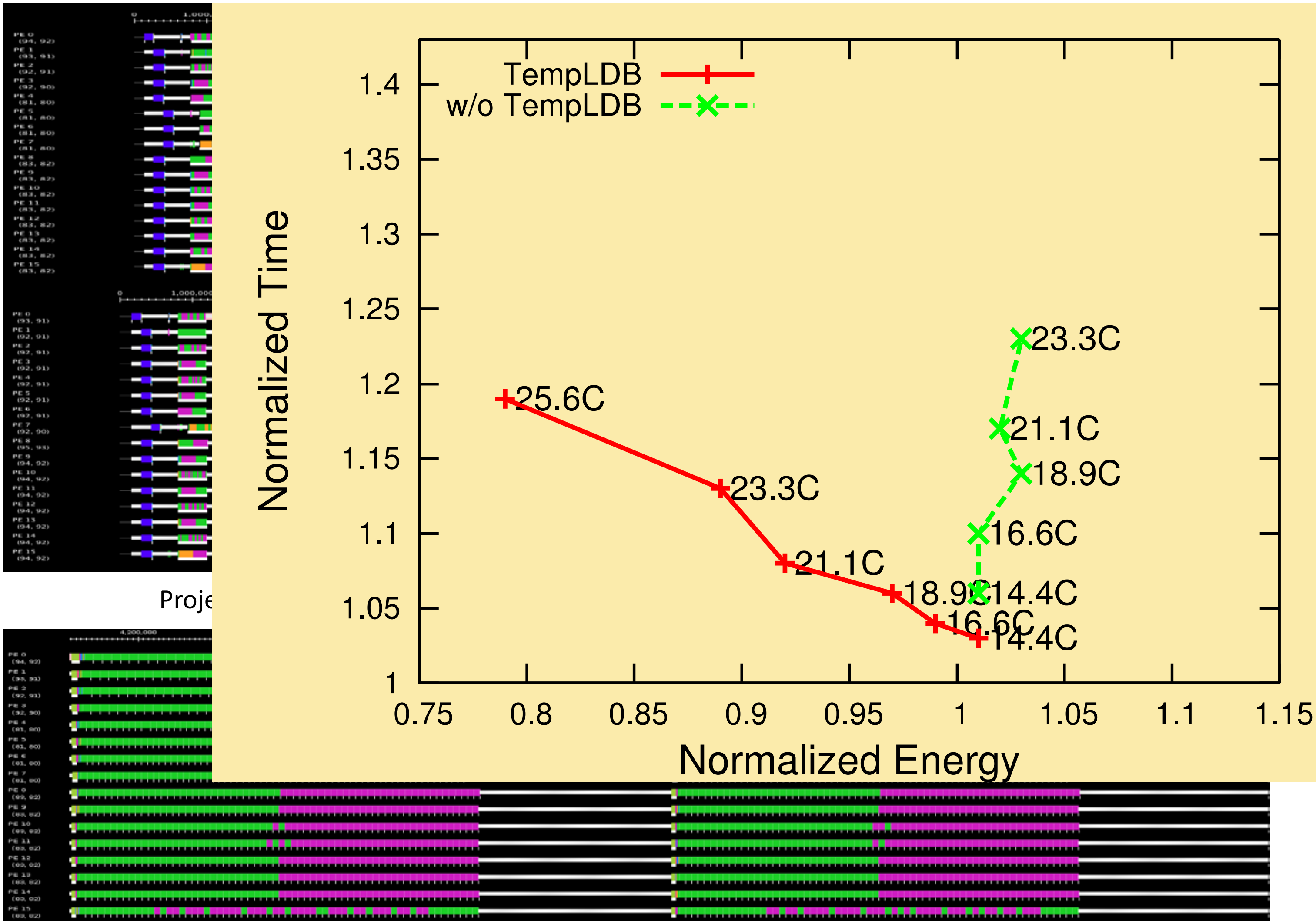
Migratable Objects to the rescue!

PPL UIUC

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

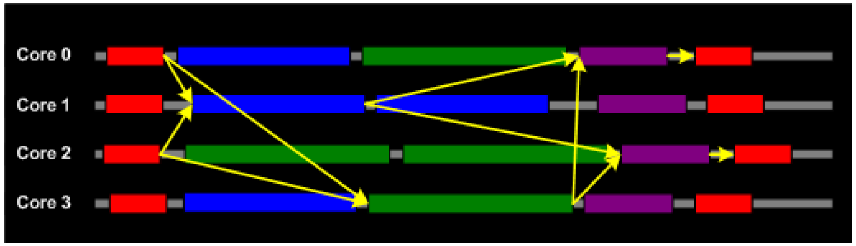
Benefits of Temperature Aware LB



Zoomed projection timeline for two iterations without 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



Scalable Fault tolerance

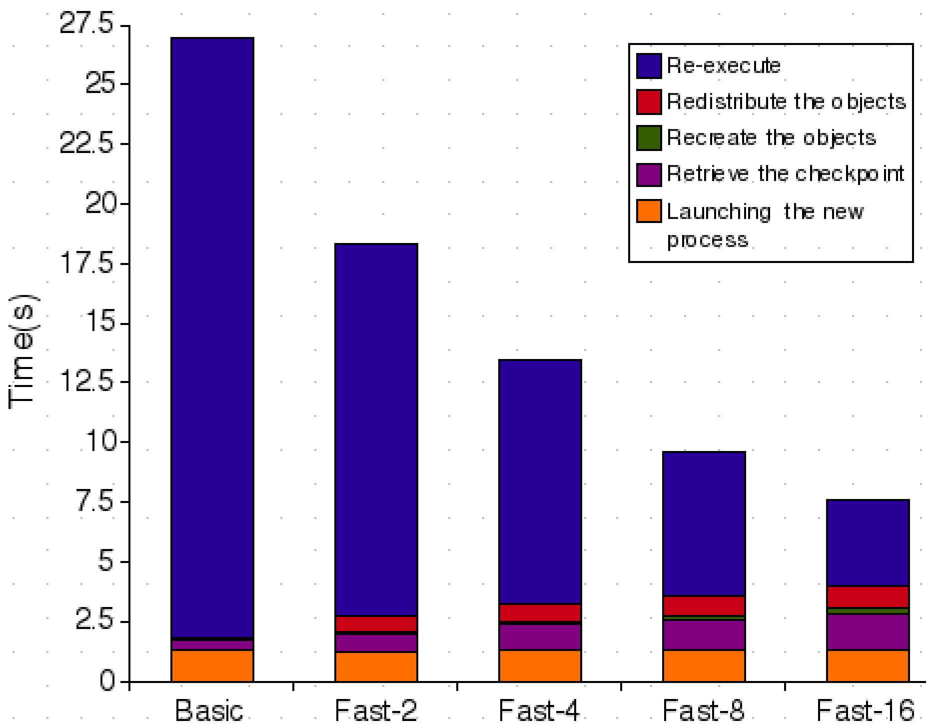
- Faults will be common at exascale
  - Failstop, and soft failures are both important
- Checkpoint-restart will not scale
  - Requires all nodes to roll back even when just one fails
    - Inefficient: computation and power
  - As MTBF goes lower, it becomes infeasible

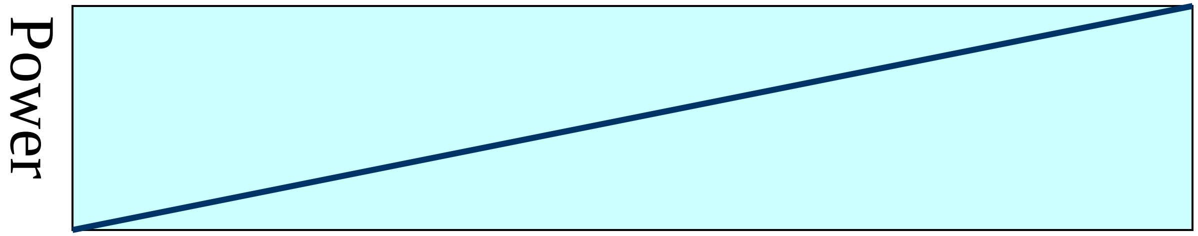
Message-Logging

- Basic Idea:
  - Messages are stored by sender during execution
  - Periodic checkpoints still maintained
  - After a crash, reprocess “recent” 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

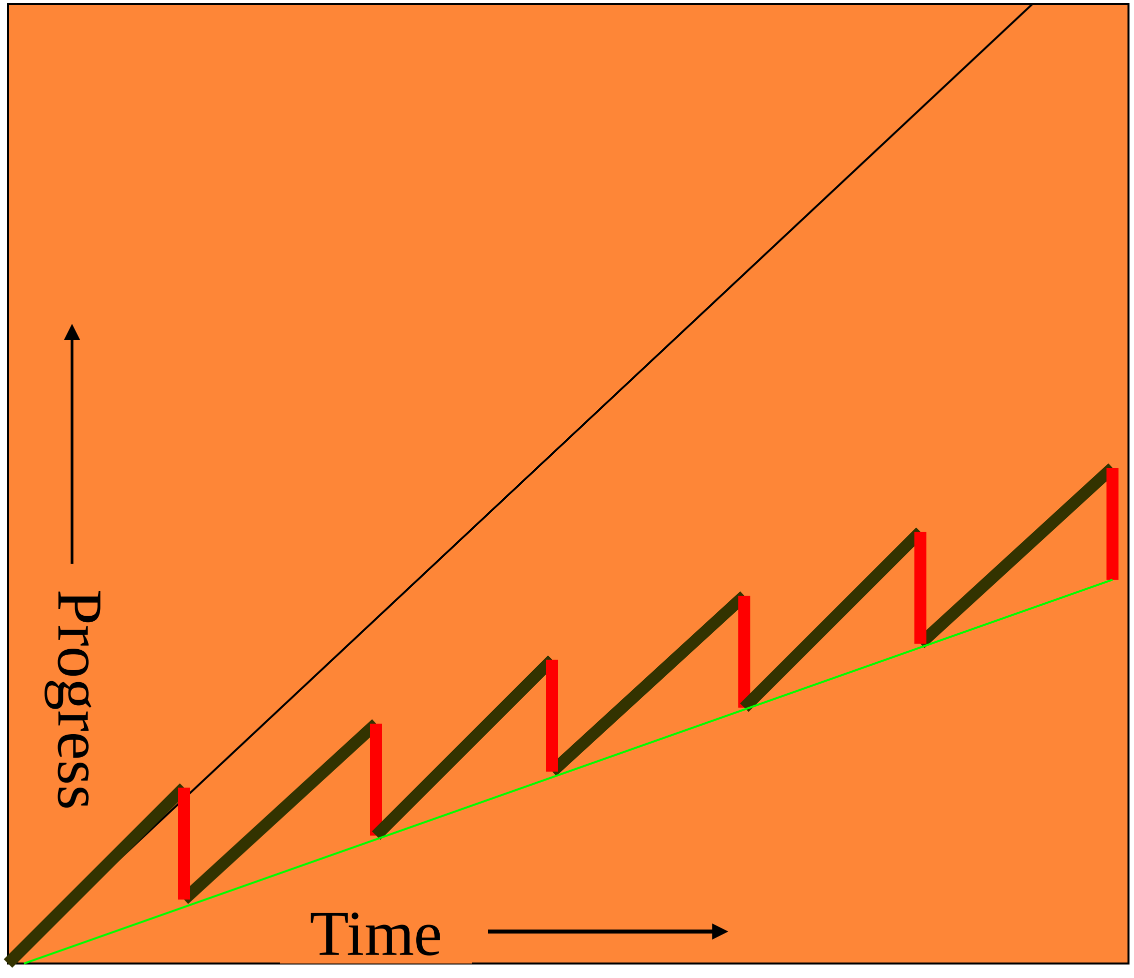
Message-Logging (cont.)

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



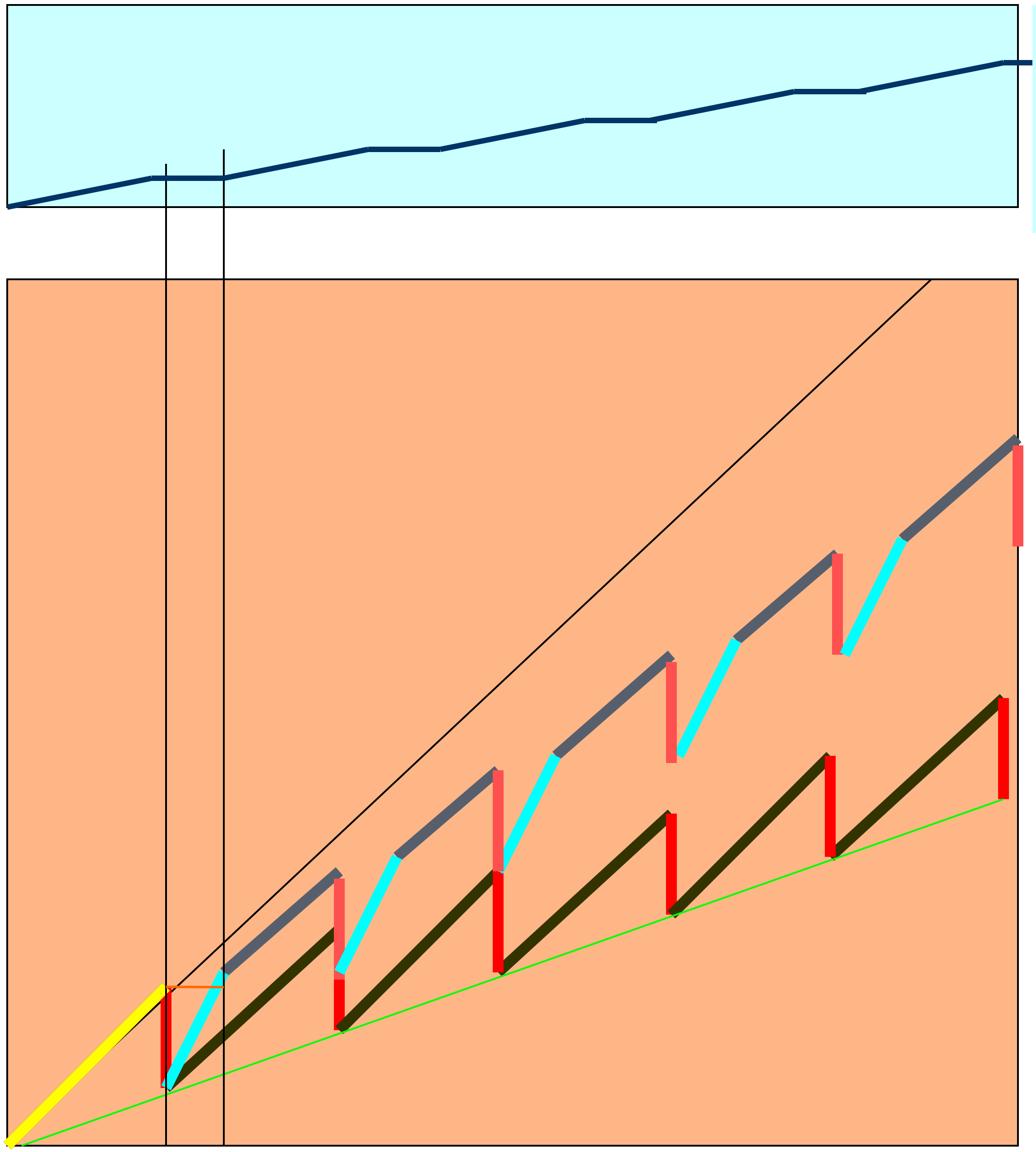


Power consumption is continuous



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

Virtualization: Pros, Cons, and Remedies

- We examined the "Pro"s so far.
- Cons and remedies:
  - Memory in ghost layer increases
    - Fuse local regions with compiler support
    - Fetch one ghost layer at a time
    - Hybridize (pthreads/openMP inside objects/DEBs)
  - Less control over scheduling?
    - i.e. too much asynchrony?
    - But can be controlled in various ways by an observant RTS
  - Too radical and new?
    - Well, its working well for the past 10-15 years in multiple applications, via Charm++ and AMPI
  - Too old?
    - What can I say. May be we can invent a new name

PPL UIUC

New Programming Models

- Simplify parallel programming, improve productivity
- Two broad themes:
- **Frameworks**
  - Encapsulate common data-structure specific code
  - Or domain specific code
  - Avoids duplication/promotes reuse of expensive parallel software
- **Simpler but incomplete languages:**
  - *Restricting* modes of interactions among parallel entities leads to simpler languages
  - Each language may be incomplete but:
    - Addresses important subclasses of algorithms
    - Together with other models, lead to a complete toolkit

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.

Compiler Support

- Needed, but in a low-brow way
  - Not for auto-parallelization
- A basic compiler infrastructure
  - Easy to extend
  - Allows code restructuring
  - Supports syntax that improves productivity
  - Basic, well-understood analyses
    - E.g. live-variables analysis for checkpointing
  - **Inserting Control-points to provide knobs to RTS**
- Rose?

Less-technical points

- Where are the youngsters??
  - We have a big problem for the field if young computer scientists are not joining this field
- Need for dialogue:
  - friendly, no-holds-barred, and extensive discussion among the 20 or so leading researchers in the field
  - Feasible now, because most of us are senior (well ☺) researchers, in no need for jockeying, and facing the largest challenge of our times for this field

Summary

- Do away with the notion of processors
  - Adaptive Runtimes, enabled by migratable-objects programming model (aka virtualization)
    - Are necessary at exascale
    - Need to become more intelligent and introspective
    - Help manage accelerators, balance load, tolerate faults,
- Interoperability, concurrent composition become even more important
  - Supported by virtualization
- New programming models and frameworks
  - Create an ecosystem/toolbox of programming paradigms rather than one "super" language
  - Avoid premature standardization