Charm++ Overview

UCX Hackathon 2019

Nitin Bhat
Software Engineer
Charmworks Inc.

Agenda

1. Introduction to Charm++
2. Code Example (Hello World)
3. Charm++ Messaging Models and Implementation
4. Advanced Features
5. Conclusion/More Information

Introduction to the Charm++ Model

Challenges in Parallel Programming

- Applications are getting more sophisticated
  - Adaptive refinement
  - Multi-scale, multi-module, multi-physics
    - load imbalance emerges as a huge problem for some apps
- Exacerbated by strong scaling needs from apps
  - Strong scaling: run an application with same input data on more processors, and get better speedups
  - Weak scaling: larger datasets on more processors in the same time
- Hardware variability
  - Static/dynamic
  - Heterogeneity: processor types, process variation, etc.
  - Power/Temperature/Energy
  - Component failure

Our View

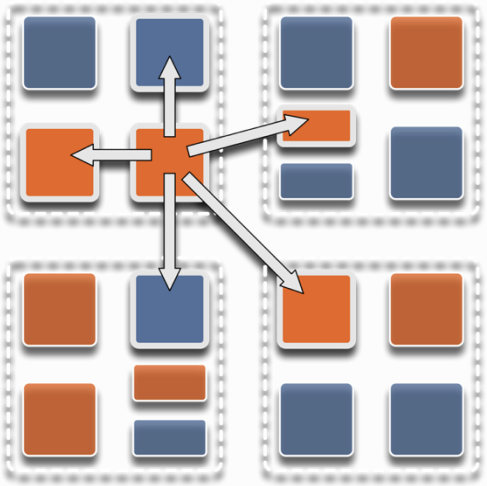
- To deal with these challenges, we must seek:
  - Not full automation
  - Not full burden on app-developers
  - But: a good division of labor between the system and app developers
    - Programmer: what to do in parallel, System: where,when
- Develop language driven by needs of real applications
  - Avoid “platonic” pursuit of “beautiful” ideas
  - Co-developed with NAMD, ChaNGa, OpenAtom,..
- Pragmatic focus
  - Ground-up development, portability,
  - accessibility for a broad user base

What is Charm++?

- Charm++ is a generalized approach to writing parallel programs
  - An alternative to the likes of MPI, UPC, GA etc.
  - But not to sequential languages such as C, C++, and Fortran
- Represents:
  - The style of writing parallel programs
  - The runtime system
  - And the entire ecosystem that surrounds it
- Three design principles:
  - Overdecomposition, Migratability, Asynchrony

Overdecomposition

- Decompose the work units & data units into many more pieces than execution units
  - Cores/Nodes/…
- Not so hard: we do decomposition anyway



Migratability

- Allow these work and data units to be migratable at runtime
  - i.e. the programmer or runtime can move them
- Consequences for the application developer
  - Communication must now be addressed to logical units with global names, not to physical processors
  - But this is a good thing
- Consequences for RTS
  - Must keep track of where each unit is
  - Naming and location management

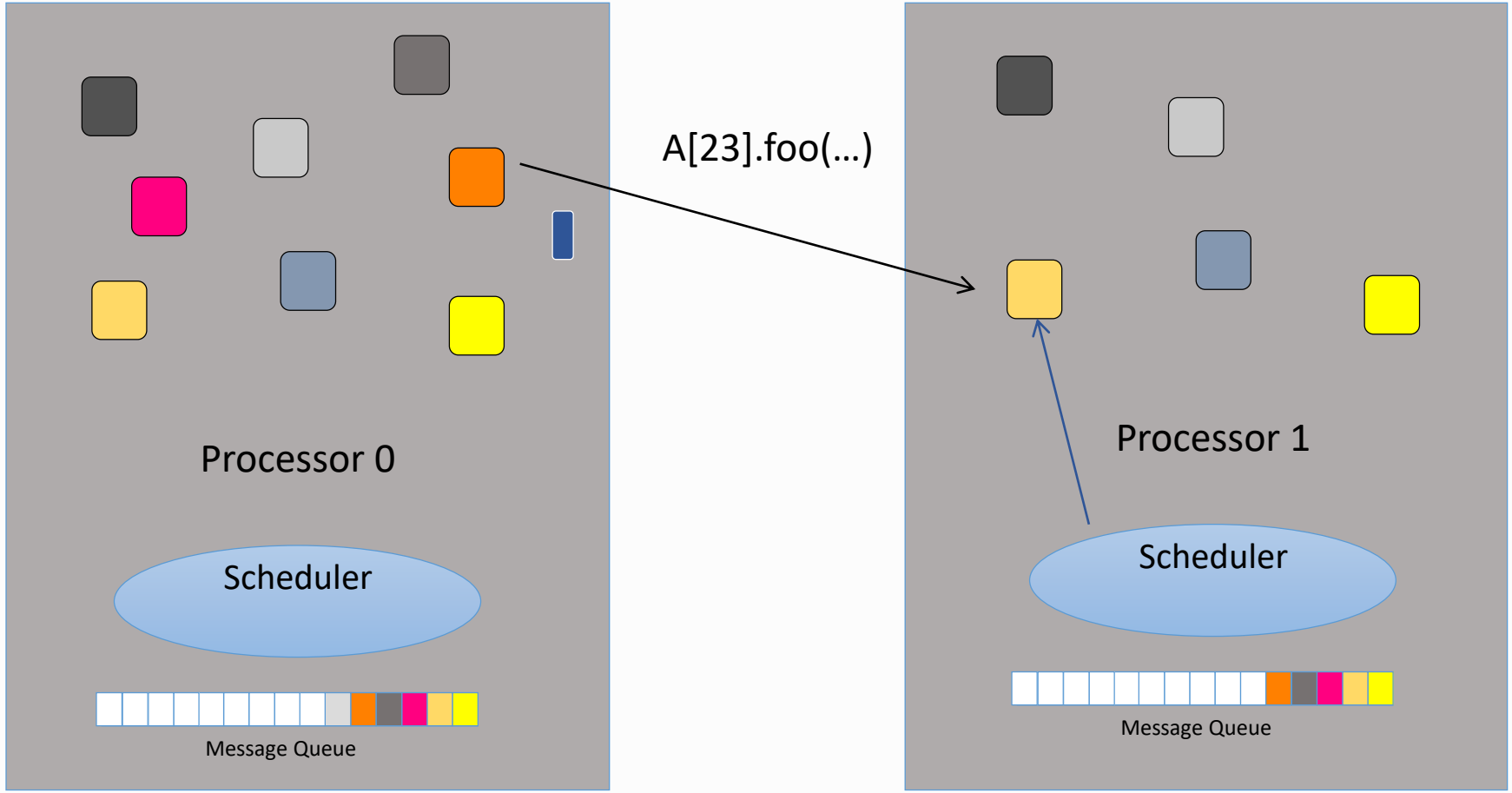
Asynchrony: Message-Driven Execution

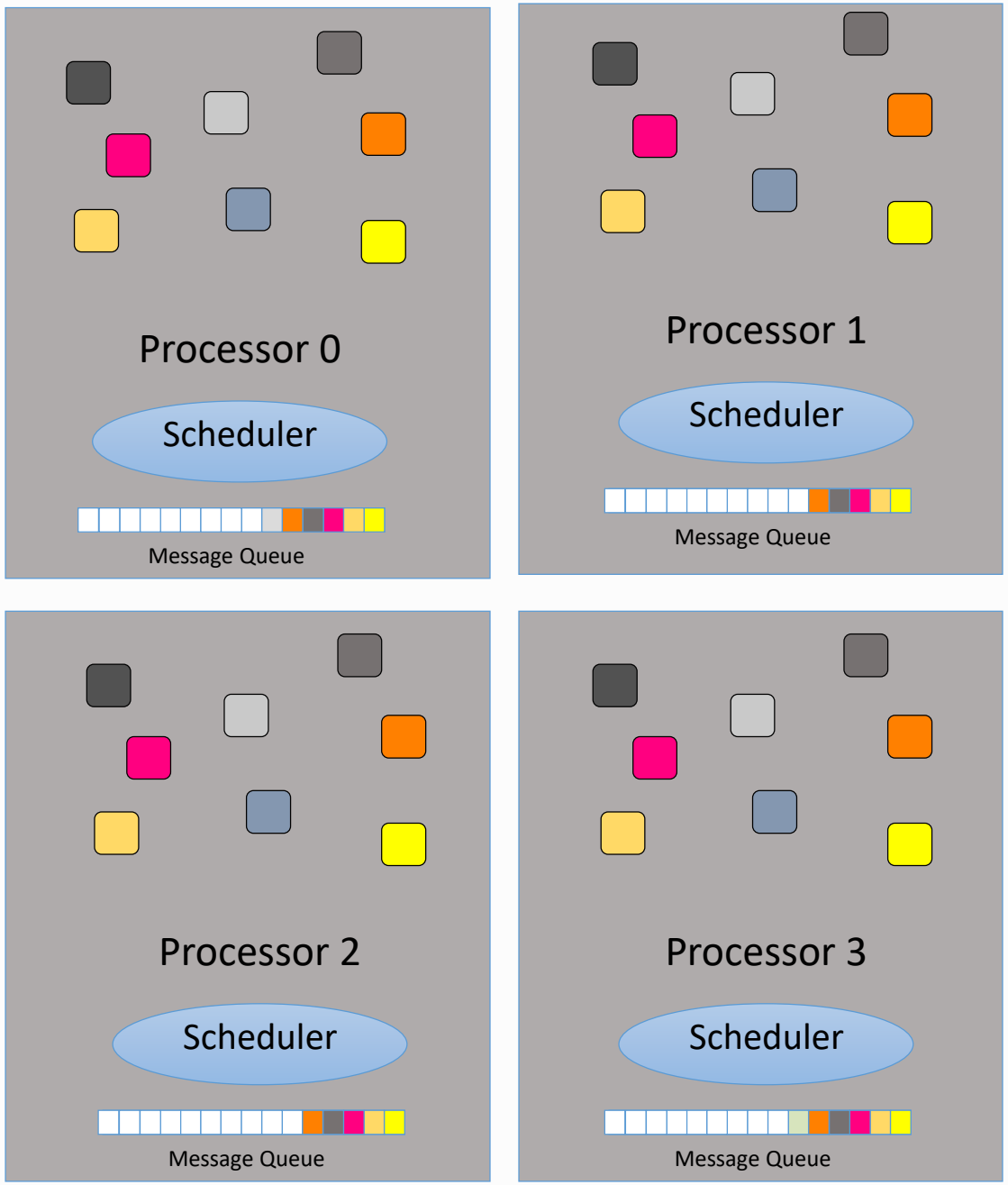
- With over-decomposition and migratability:
  - You have multiple units on each processor
  - They address each other via logical names
- Need for scheduling:
  - What sequence should the work units execute in?
  - One answer: let the programmer sequence them
    - Seen in current codes, e.g. some AMR frameworks
  - Message-driven execution:
    - Let the work-unit that happens to have data (“message”) available for it execute next
    - Let the RTS select among ready work units
    - Programmer should not specify what executes next, but can influence it via priorities

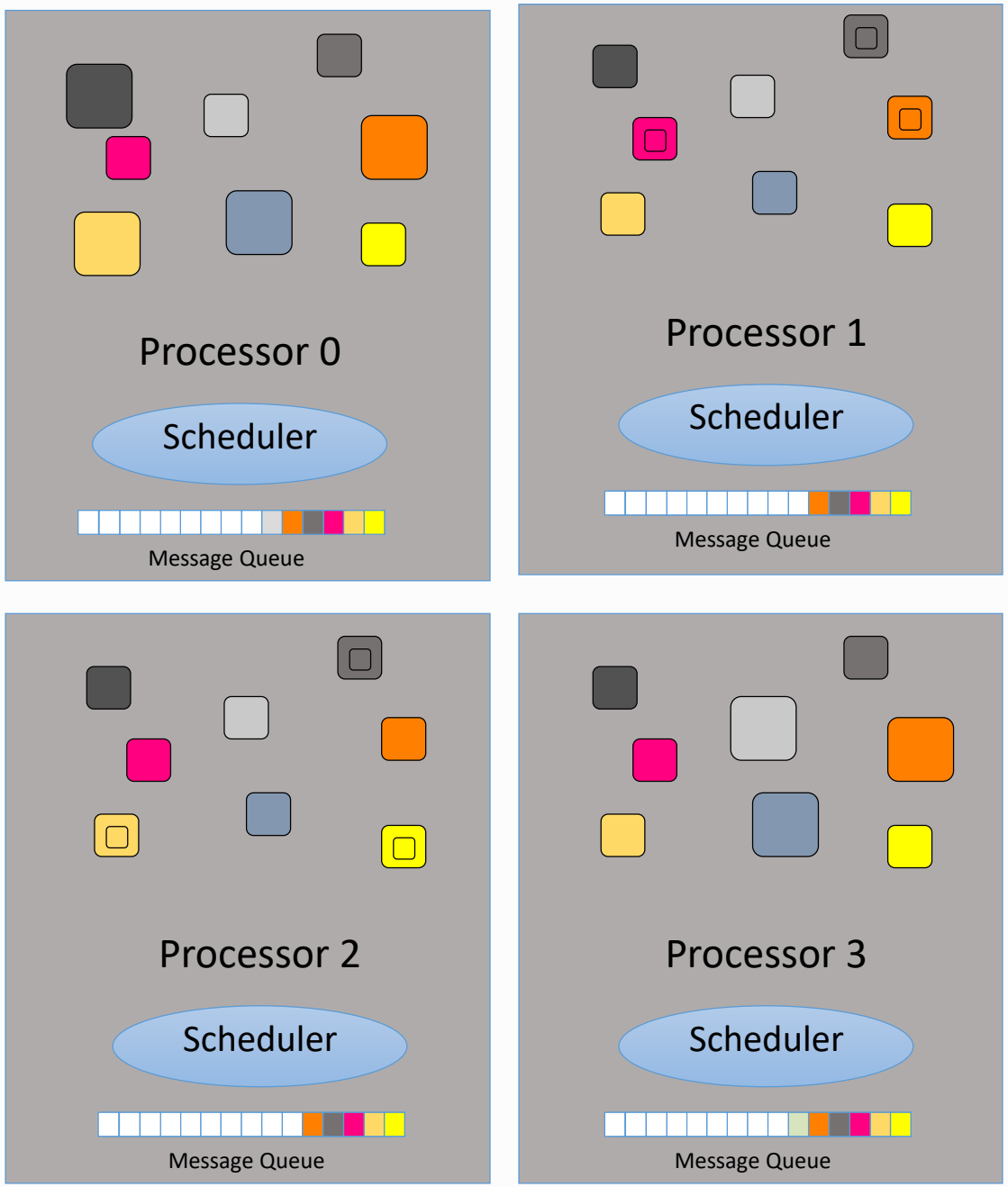
Realization of This Model in Charm++

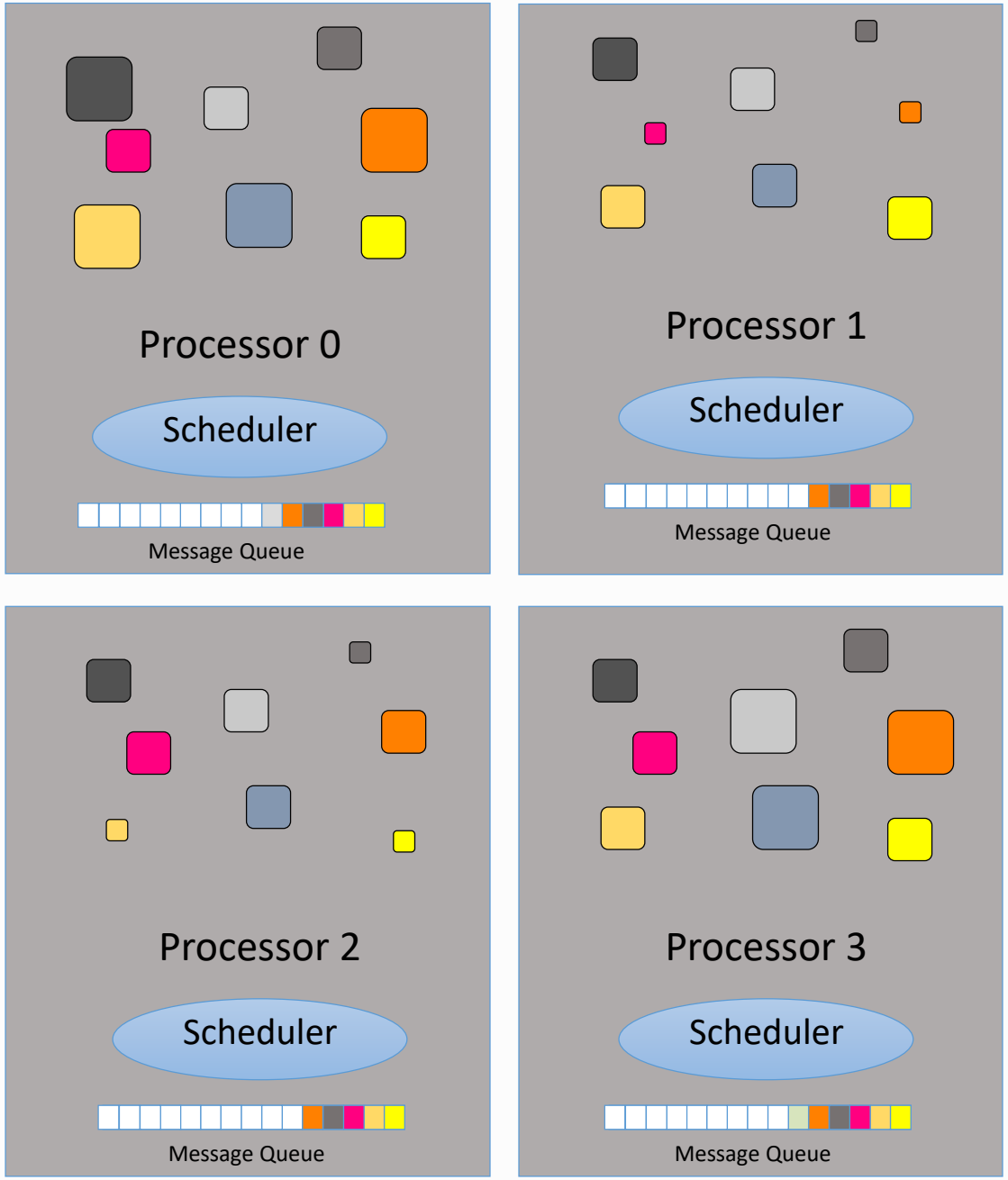
- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as “entry” methods
    - Which can be invoked asynchronously by remote chares
  - Chares are organized into indexed collections (chare arrays)
    - Each collection may have its own indexing scheme
      - 1D, ..., 6D
      - Sparse
      - Bitvector or string as an index
  - Chares communicate via asynchronous method invocations
    - `A[i].foo(…);`
      - A is the name of a collection, `i` is the index of the particular chare.

Message-driven Execution

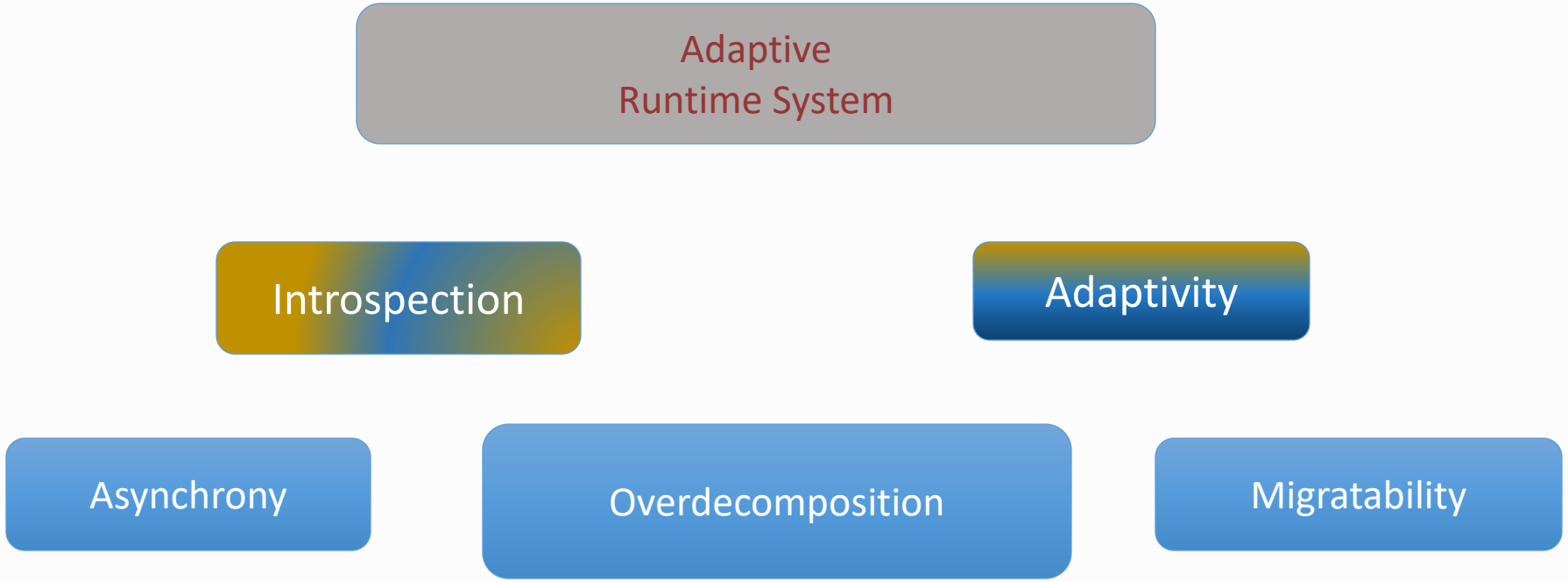



Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue


Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue


Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue

Empowering the RTS



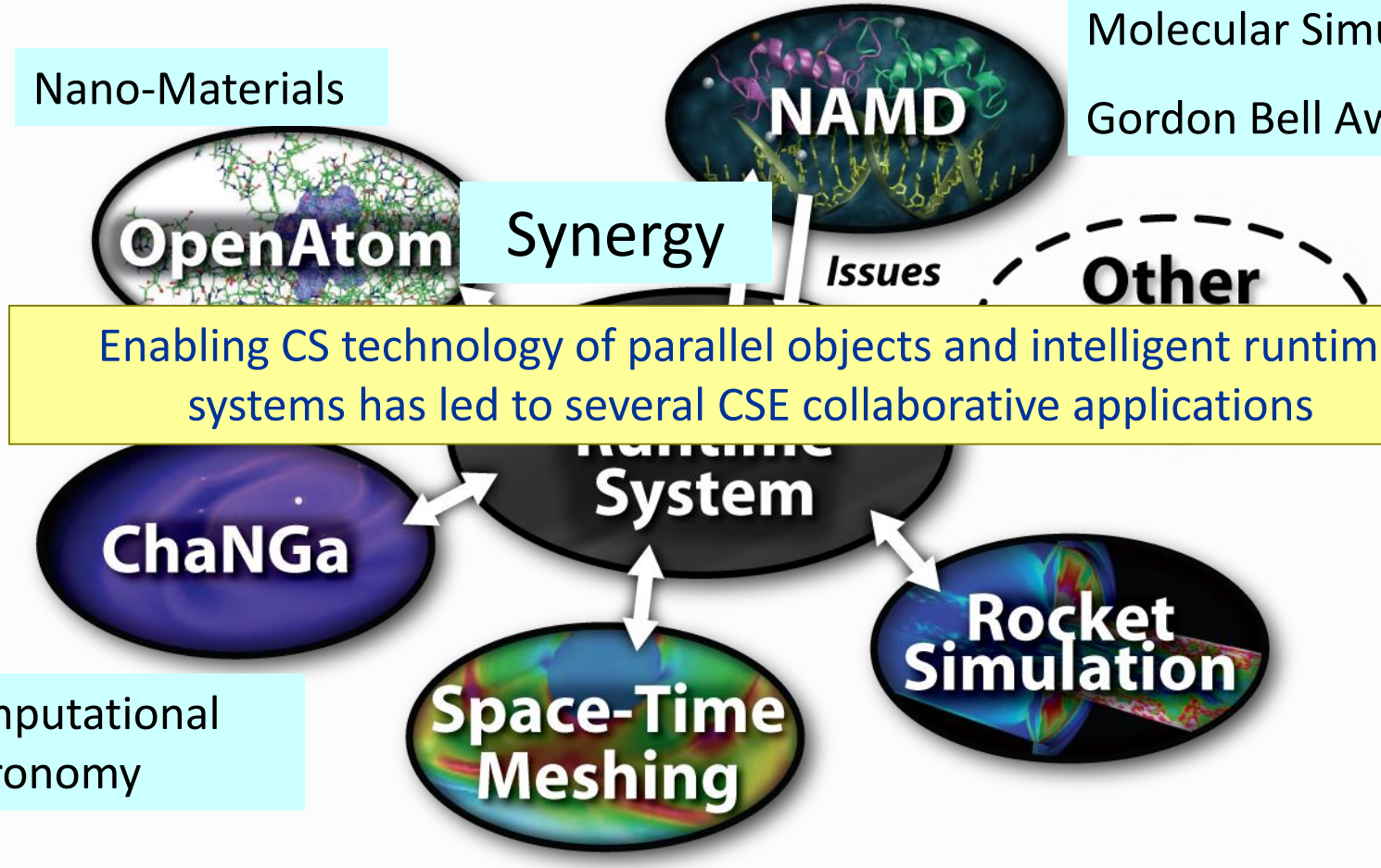
- The Adaptive RTS can:
  - Dynamically balance loads
  - Optimize communication:
    - Spread over time, async collectives
  - Automatic latency tolerance
  - Prefetch data with almost perfect predictability

Charm++ and CSE Applications

Well-known Biophysics Molecular Simulation App

Gordon Bell Award, 2002

Nano-Materials



Enabling CS technology of parallel objects and intelligent runtime systems has led to several CSE collaborative applications

Computational Astronomy

Summary: What is Charm++?

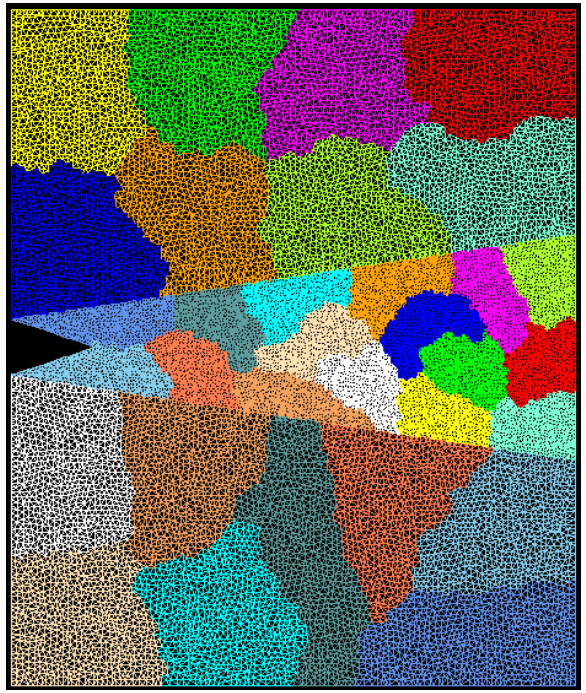
- Charm++ is a way of parallel programming
- It is based on:
  - Objects
  - Overdecomposition
  - Asynchrony
    - Asynchronous method invocations
  - Migratability
  - Adaptive runtime system
- It has been co-developed synergistically with multiple CSE applications

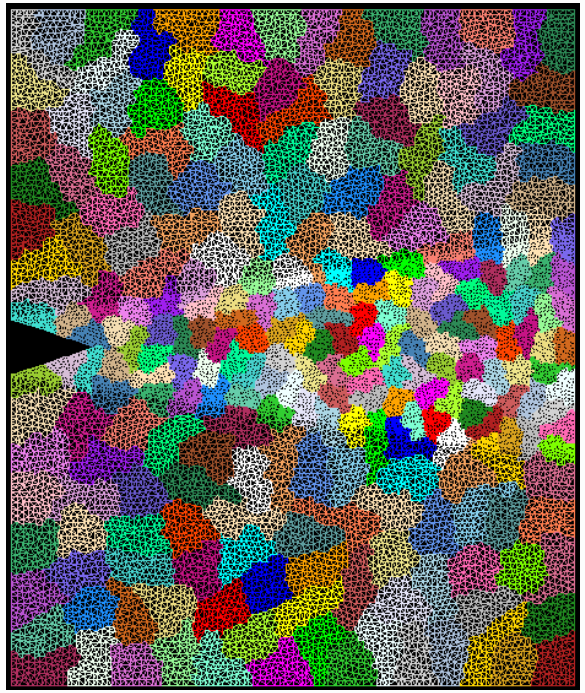
Grainsize

- Charm++ philosophy:
  - Let the programmer decompose their work and data into coarse-grained entities
- It is important to understand what I mean by coarse-grained entities
  - You don't write sequential programs that some system will auto-decompose
  - You don't write programs when there is one object for each *float*
  - You consciously choose a grainsize, **but** choose it independently of the number of processors
    - Or parameterize it, so you can tune later

Crack Propagation

This is 2D, circa 2002...
but shows overdecomposition for unstructured meshes

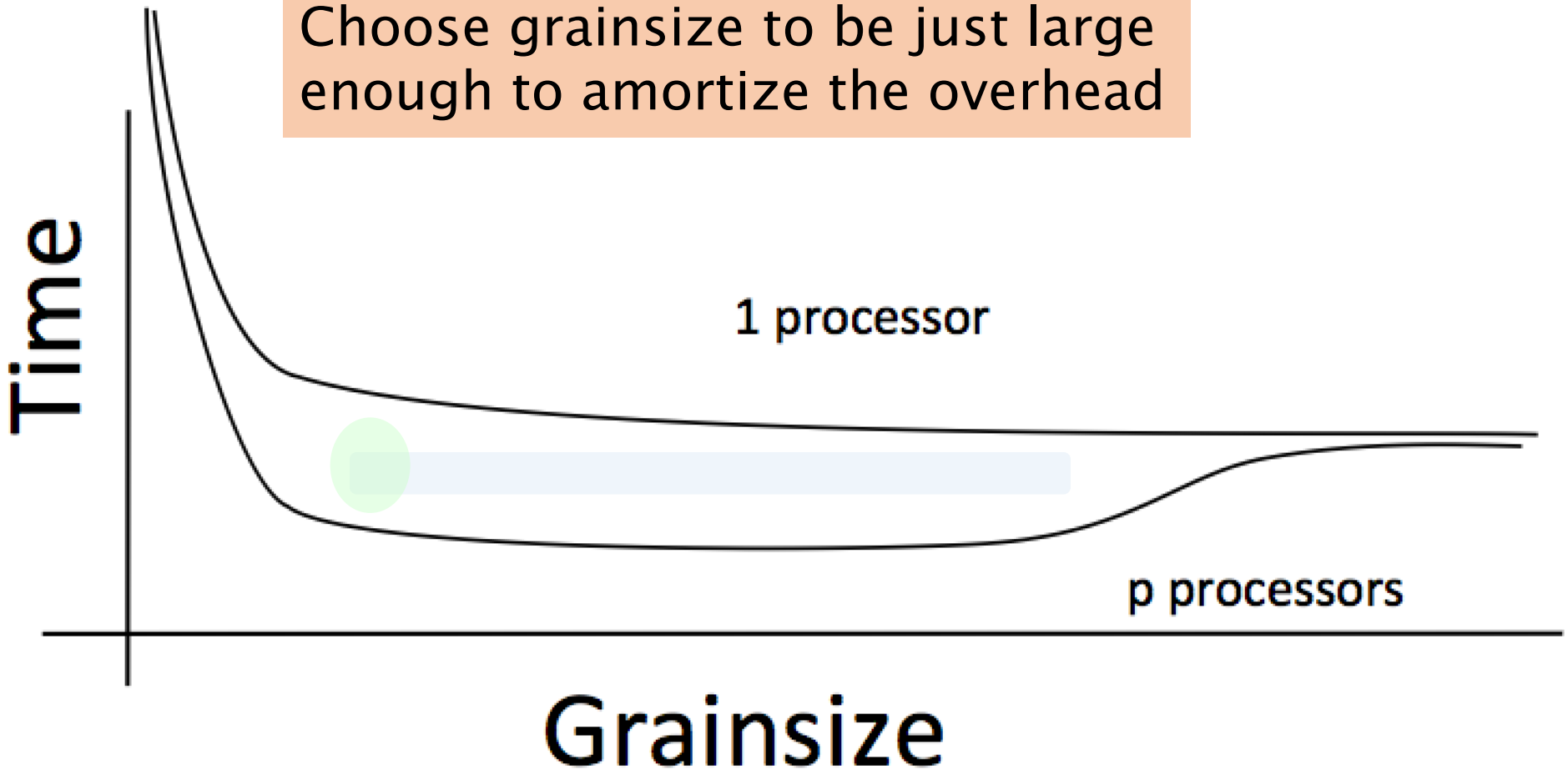




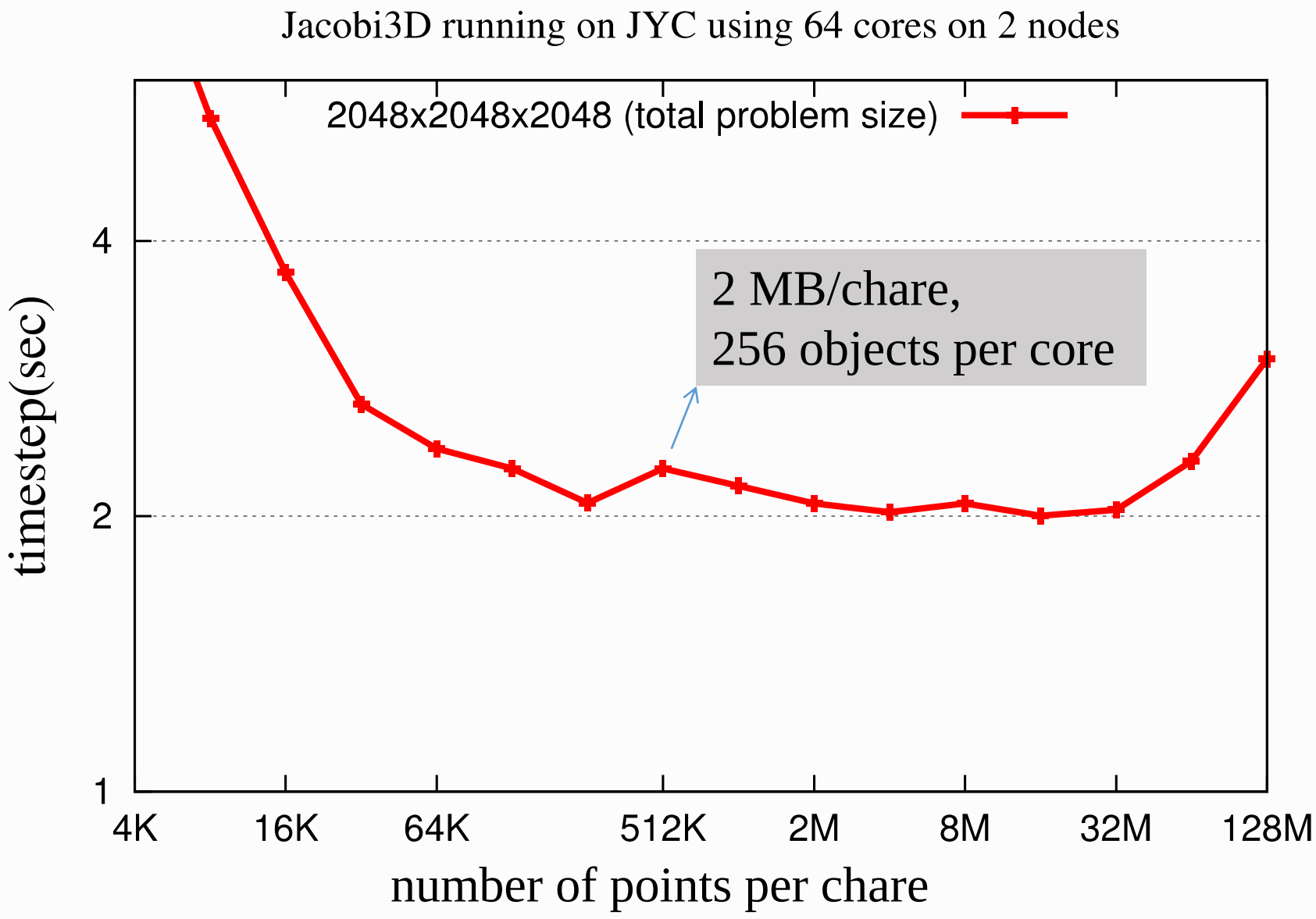
Decomposition into 16 chunks (left) and 128 chunks, 8 for each PE (right). The middle area contains cohesive elements. Both decompositions obtained using Metis. Pictures: S. Breitenfeld, and P. Geubelle

Working definition of grainsize:
amount of computation per remote interaction

Choose grainsize to be just large enough to amortize the overhead

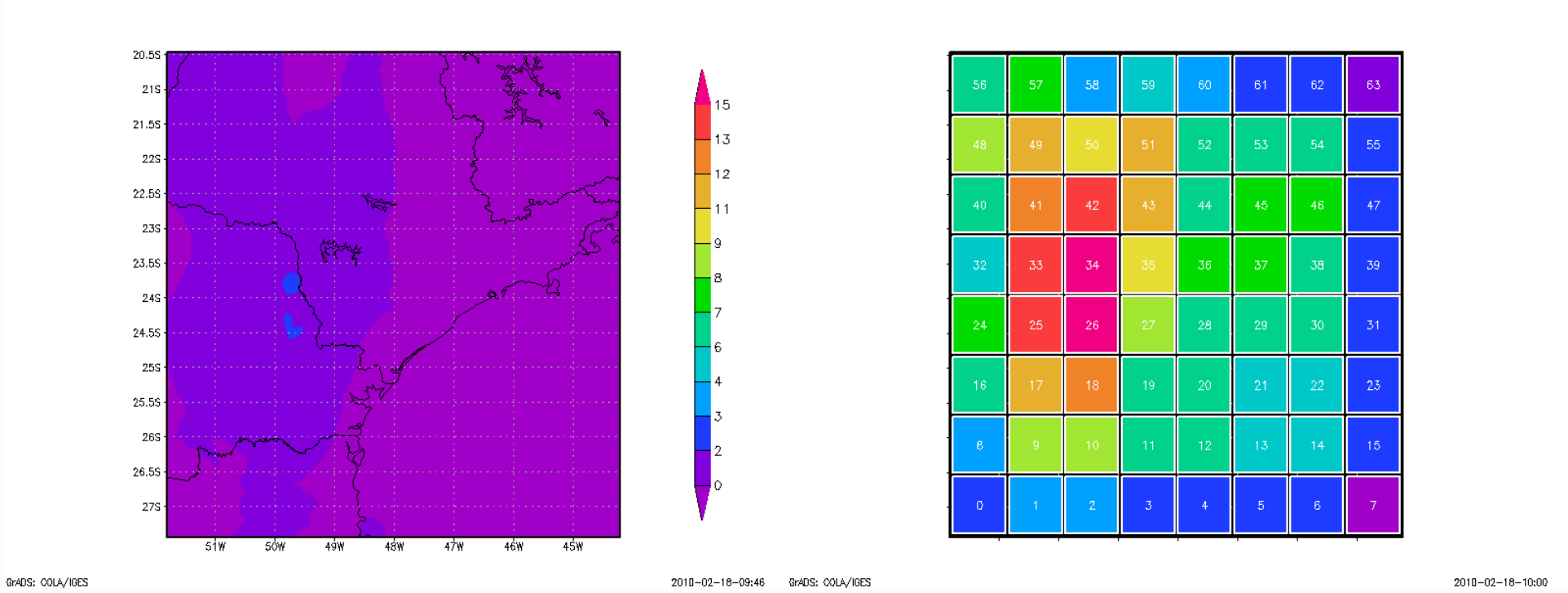


Grainsize in a common setting



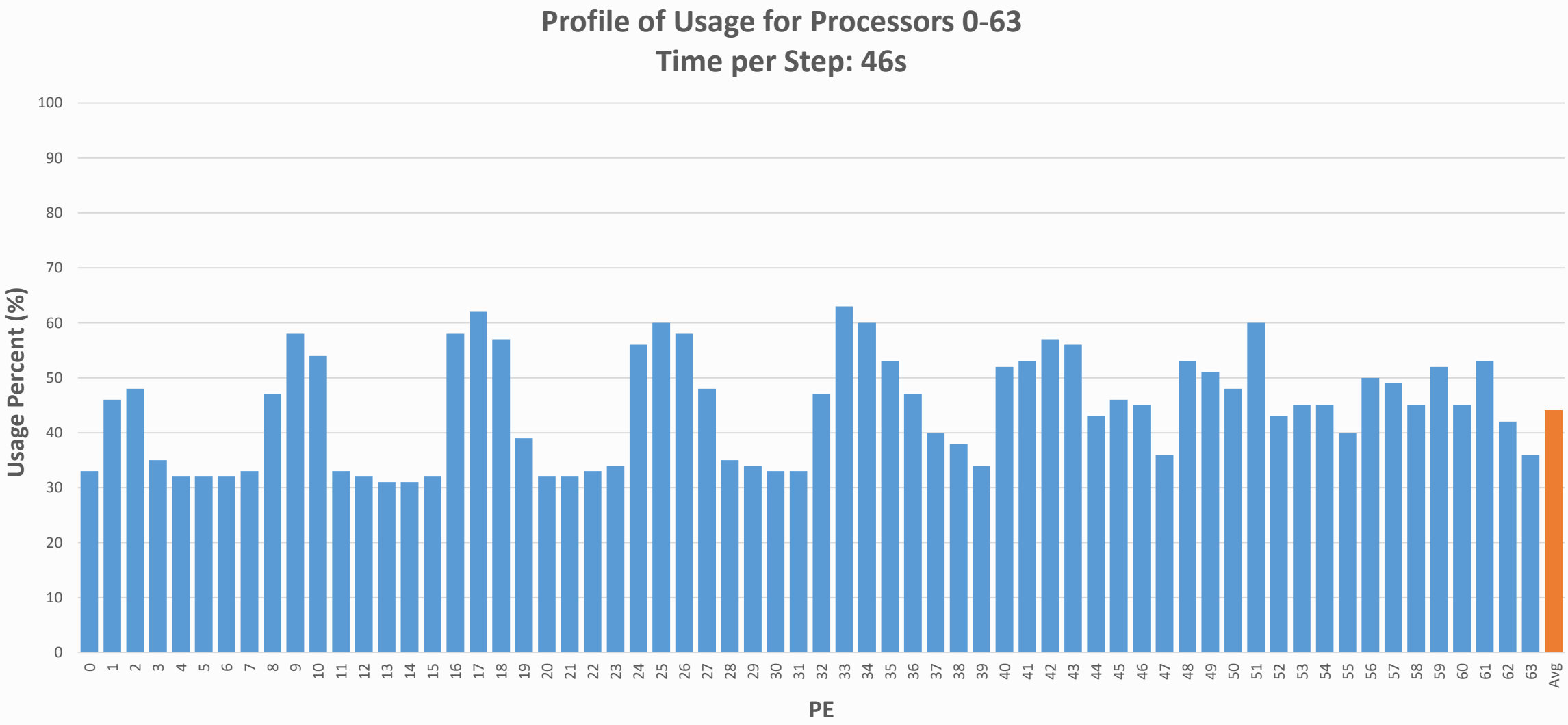
Grainsize: Weather Forecasting in BRAMS

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

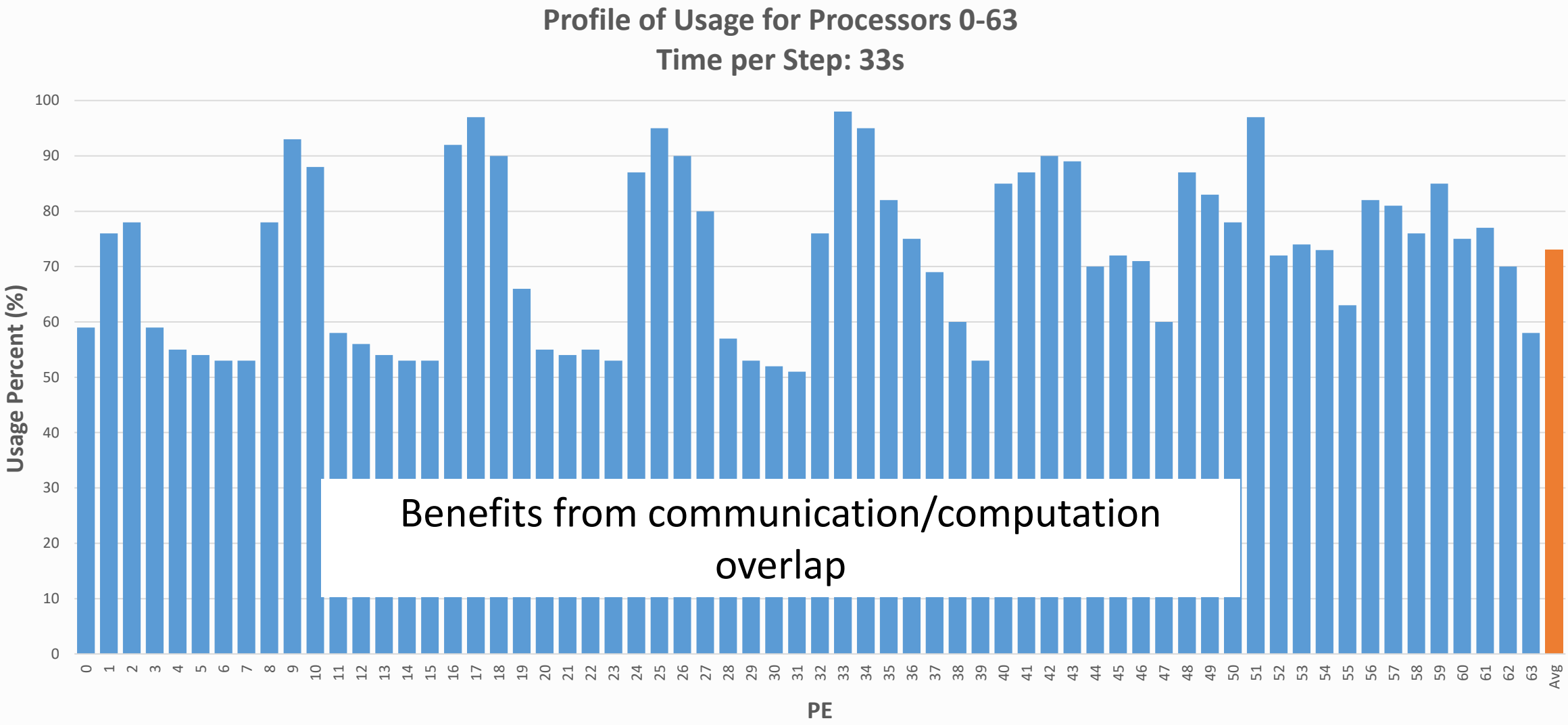


Instead of using 64 work units on 64 cores, used 1024 on 64

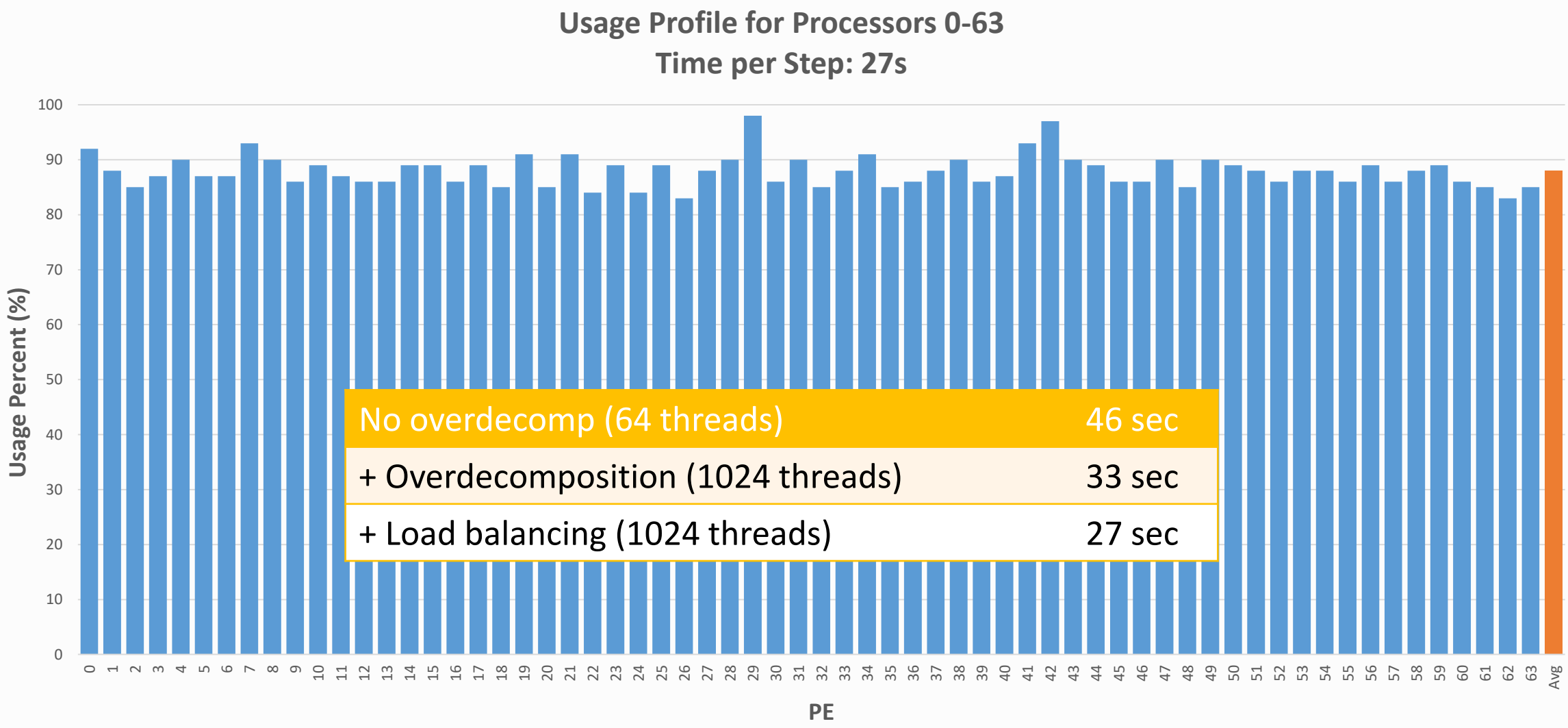
Baseline: 64 Objects



Overdecomposition: 1024 Objects

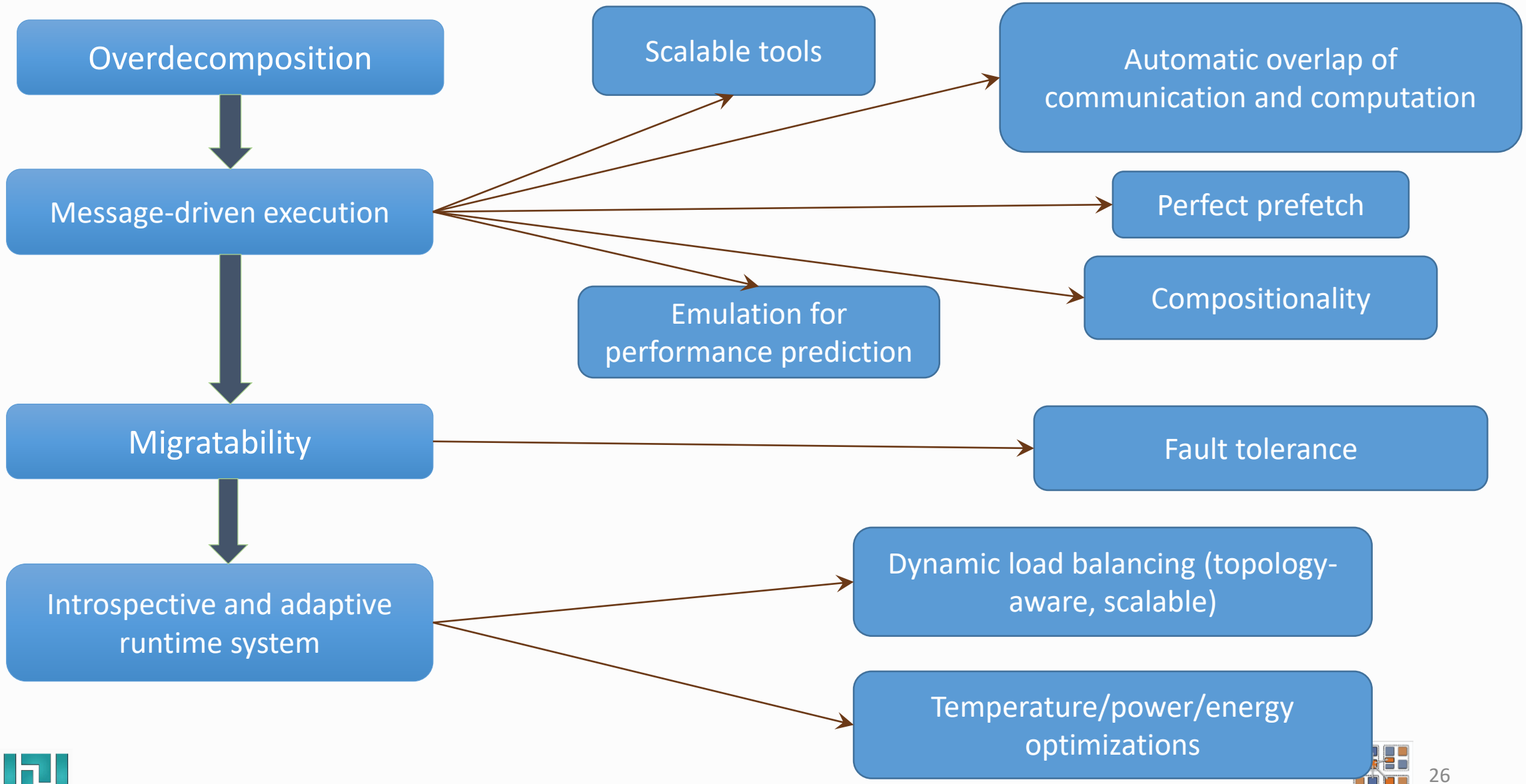


With Load Balancing: 1024 objects



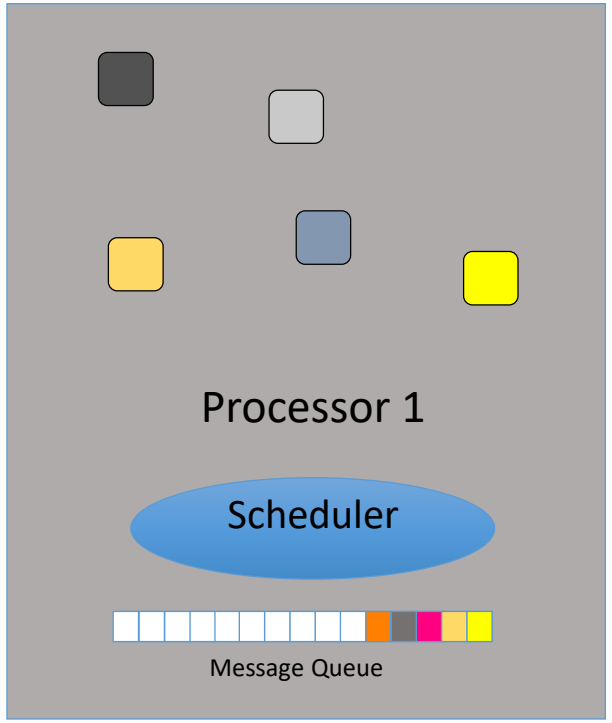
No overdecomp (64 threads)	46 sec
+ Overdecomposition (1024 threads)	33 sec
+ Load balancing (1024 threads)	27 sec

Charm++ Benefits



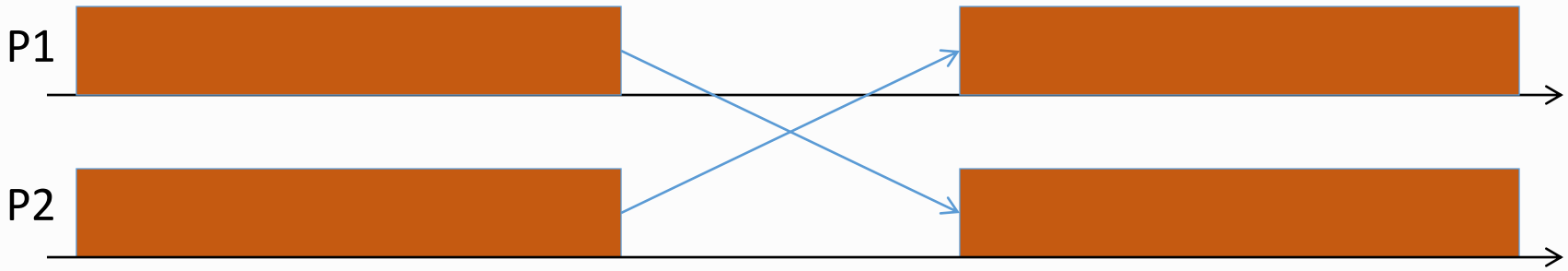
Locality and Prefetch

- 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 scratchpad memories, for example



Impact on Communication

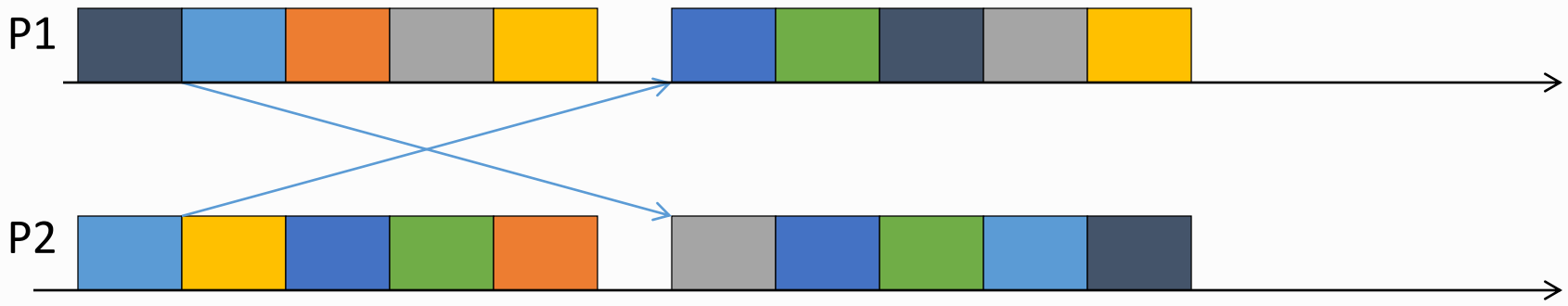
- Current use of communication network:
  - Compute-communicate cycles in typical MPI apps
  - The network is used for a fraction of time
    - And is on the critical path
- Current communication networks are over-engineered by necessity



BSP based application

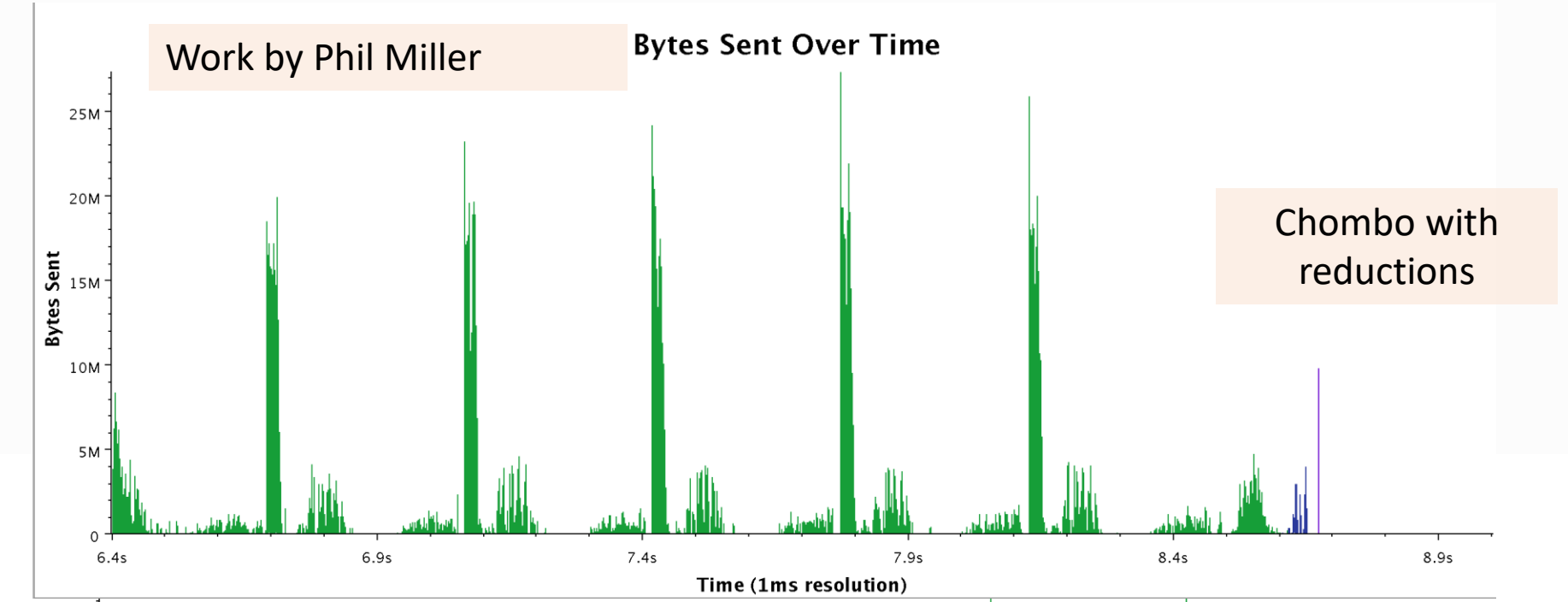
Impact on Communication

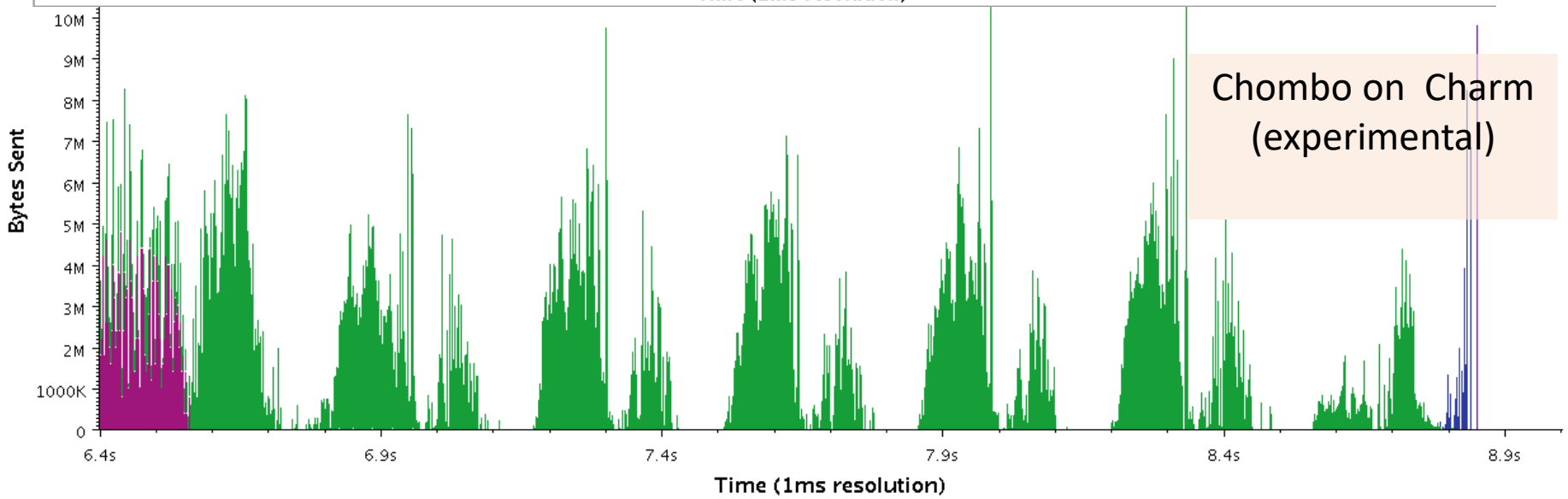
- With overdecomposition:
  - Communication is spread over an iteration
  - Adaptive overlap of communication and computation



Overdecomposition enables overlap

Communication Data from Chombo



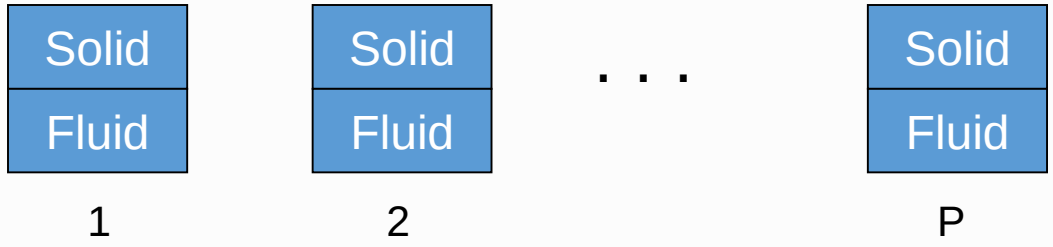


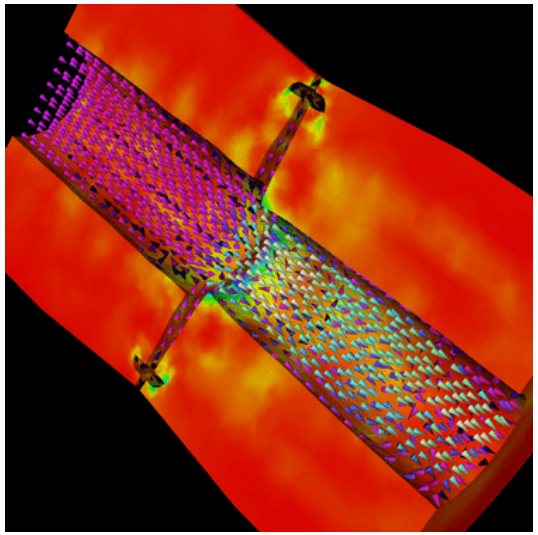
Decomposition Challenges

- Current method is to decompose to processors
  - 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 – enabled by object based decomposition
- Let runtime system (RTS) assign objects to available resources adaptively

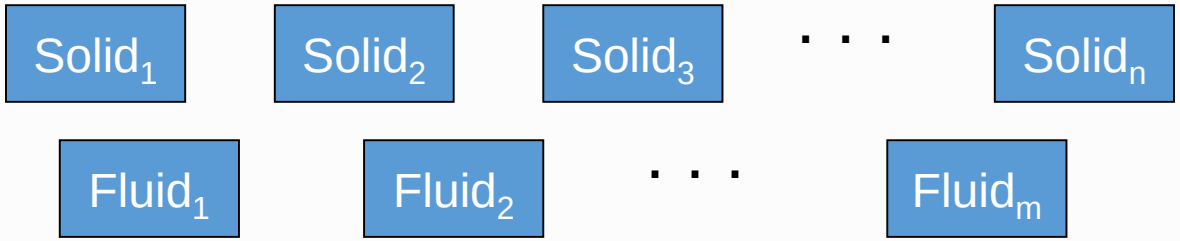
Decomposition Independent of numCores

- Rocket simulation example under traditional MPI





- With migratable-objects:



  - Benefit: load balance, communication optimizations, modularity

Simple Code Example

Hello World with Chares

hello.ci

```
mainmodule hello {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
  chare Singleton {
    entry Singleton();
  };
};
```

ci file is processed to generate code for classes such as Cbase_Main, Cbase_Singleton, Cproxy_Singleton

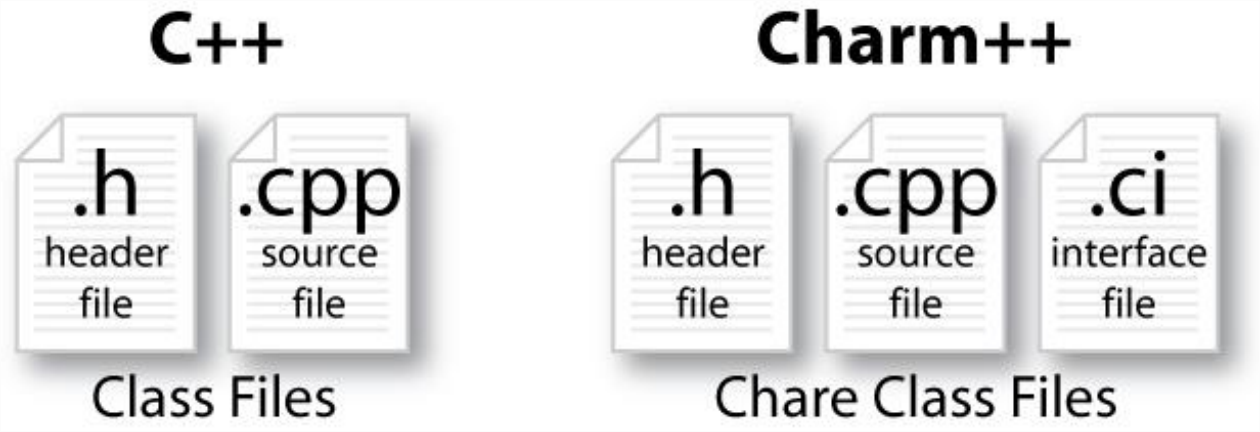
hello.cpp

```
#include "hello.decl.h"

class Main : public CBase_Main {
  public: Main(CkArgMsg* m) {
    CProxy_Singleton::ckNew();
  };
};
class Singleton : public CBase_Singleton
{
  public: Singleton() {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};
#include "hello.def.h"
```

Charm++ File Structure

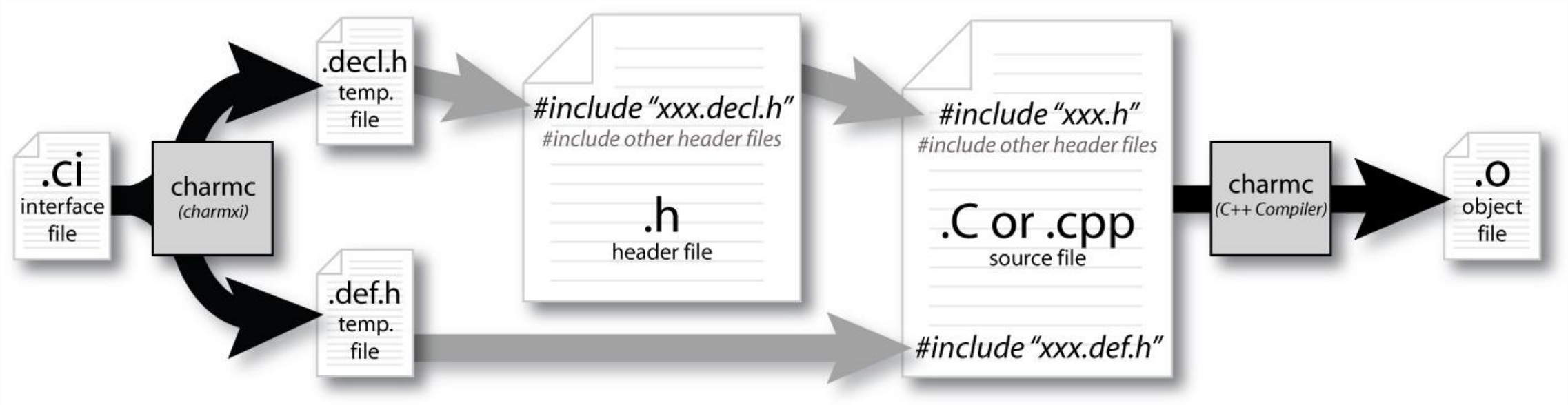
- C++ objects (including Charm++ objects)
  - Defined in regular .h and .cpp files
- Chare objects, entry methods (asynchronous methods)
  - Declared in .ci file
  - Implemented in the .cpp file



Hello World Example

- Compiling
  - `charmc hello.ci`
  - `charmc -c hello.cpp`
  - `charmc -o hello hello.o`
- Running
  - `./charmrun +p7 ./hello`
  - The +p7 tells the system to use seven cores

Compiling a Charm++ Program



Hello World with Chares

hello.ci

```
mainmodule hello {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
  chare Singleton {
    entry Singleton();
  };
};
```

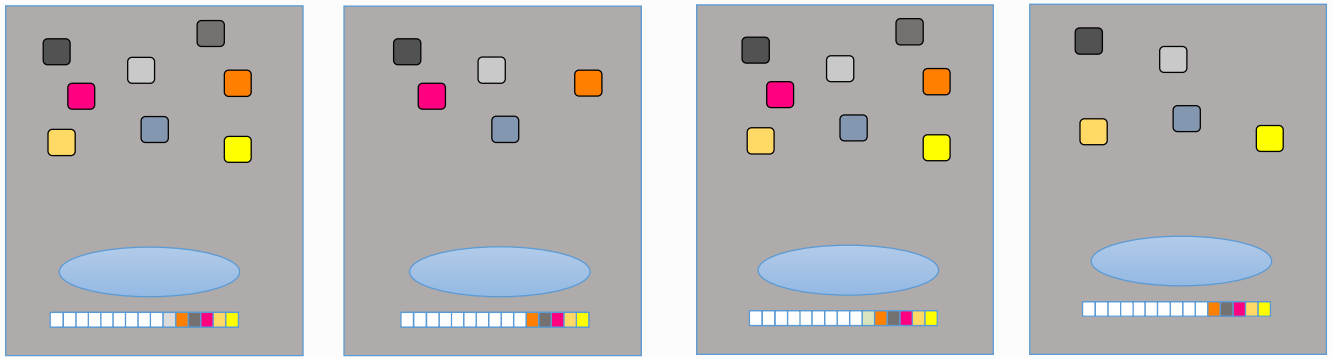
hello.cpp

```
#include "hello.decl.h"

class Main : public CBase_Main {
  public: Main(CkArgMsg* m) {
    CProxy_Singleton::ckNew();
  };
};
class Singleton : public CBase_Singleton
{
  public: Singleton() {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};
#include "hello.def.h"
```

Charm Termination

- There is a special system call `CkExit()` that terminates the parallel execution on all processors (but it is called on one processor) and performs the requisite cleanup
- The traditional `exit()` is insufficient because it only terminates one process, not the entire parallel job (and will cause a hang)
- `CkExit()` should be called when you can safely terminate the application (you may want to synchronize before calling this)



Entry Method Invocation Example: .ci file

```
mainmodule MyModule {
    mainchare Main {
        entry Main(CkArgMsg *m);
    };

    chare Simple {
        entry Simple(double y);
        entry void findArea(int radius, bool done);
    };
};
```

Does This Program Execute Correctly?

```
struct Main : public CBase_Main {
   Main(CkArgMsg* m) {
      CProxy_Simple sim = CProxy_Simple::ckNew(3.1415);
      for (int i = 1; i < 10; i++)
           sim.findArea(i, false);
      sim.findArea(10, true);
   }
};

struct Simple : public CBase_Simple {
   double y;
   Simple(double pi) { y = pi; }
   void findArea(int r, bool done) {
      ckout << "Area:" << y*r*r << endl;
      if (done)
           CkExit();
   }
};
```

No! Methods Are Asynchronous

- If a chare sends multiple entry method invocations

```
sim.findArea(1, false);
...
sim.findArea(10, true);
```

- These may be delivered in ***any*** order

```
Simple::findArea(int r, bool done){
    ckout << "Area:" << y*r*r << endl;
    if (++count == 10)
        CkExit();
}
```

- Output:

```
Area: 254.34
Area: 200.96
Area: 28.26
Area: 3.14
Area: 12.56
Area: 153.86
Area: 50.24
Area: 78.50
Area: 314.00
```

or

```
Area: 28.26
Area: 78.50
Area: 3.14
Area: 113.04
Area: 314.00
```

Charm++ Messaging Models and Implementation

Charm++ Messaging Models

- Regular “Copy-based” API

```
// .ci Declaration
entry void sendBuffer(int buffer[size], int size);

// .cpp Call
int *buffer = new int[4000];
sim.sendBuffer(buffer, 4000);
```

  - Allows immediate modification/reuse of the buffer
  - Eager for short messages, Rendezvous for large messages

Charm++ Messaging Models

- Zerocopy Send and Post API

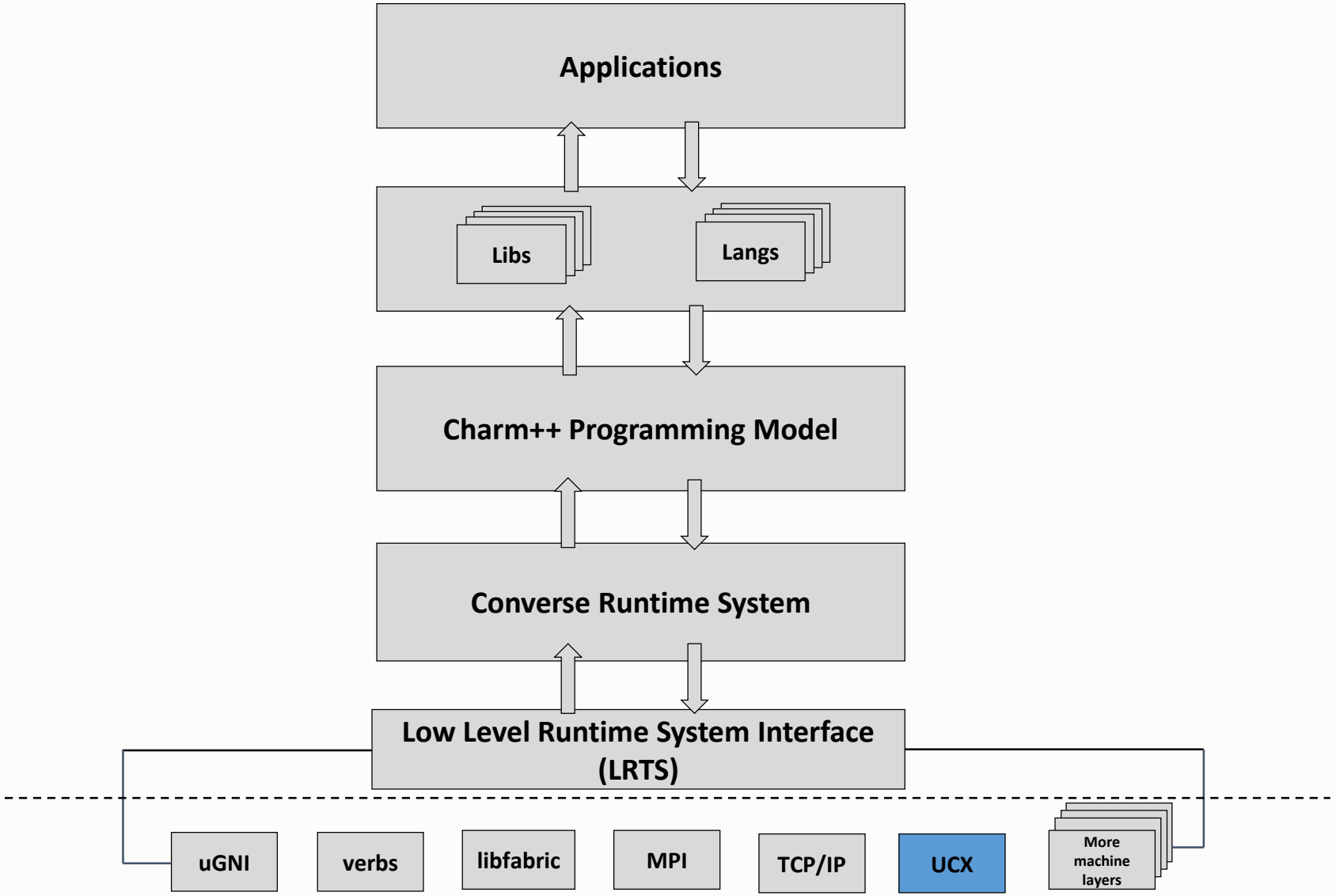
```
// .ci Declaration
entry void sendBuffer(nocopy int buffer[size], int size);
entry void sendBuffer(nocopypost int buffer[size], int size);



// .cpp Call
int *buffer = new int[4000];
sim.sendBuffer(CkSendBuffer(buffer, callback), 4000);
```

  - Buffer can be modified in the callback
  - Using rendezvous protocol

Charm++ Software Architecture



UCX Layer Implementation

- Init (in LrtsInit)
  - Process management: **simple pmi/slurm pmi/PMIx**
  - Each process :
    - **ucp_init**
    - **ucp_worker_create**
    - **ucp_ep_create**
    - Prepost recv buffers: **ucp_tag_recv_nb**
- Advance/Poll (in LrtsAdvanceCommunication)
  - **ucp_worker_progress**
- Regular API (in LrtsSendFunc)
  - Send: **ucp_tag_send_nb**
  - Recv: **ucp_tag_recv_nb/ucp_tag_msg_recv_nb**
- Zero copy API (in LrtsIssueRget/LrtsIssueRput)
  - Send metadata message using Regular API
  - RDMA operations using **ucp_get_nb/ucp_put_nb**

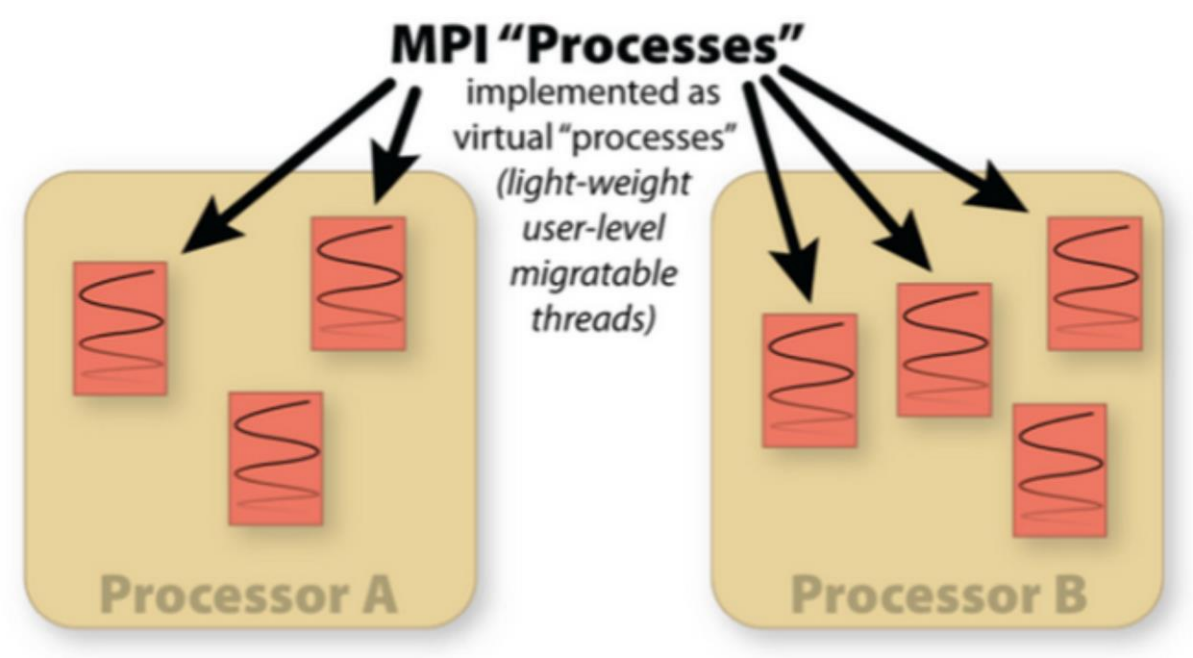
Other Important Features

Interoperability and Within Node Parallelism

- Charm++ interoperates with MPI
  - So, some modules can be written in Charm++, rest in MPI
- GPGPUs are supported
  - Via a “GPU Manager” module, with asynchronous callbacks into Charm++ code
- Multicore:
  - Charm++ has its own OpenMP runtime implementation (via LLVM)
    - Highly flexible nested parallelism
  - Charm++ can run in a mode with 1 PE on each process
    - Interoperates with regular OpenMP, OMPSS, other task models,

What is Adaptive MPI?

- AMPI is an MPI implementation on top of Charm++'s runtime system
  - Enables Charm++'s dynamic features for pre-existing MPI codes



Advanced Concepts

- Priorities
- Entry method tags
- Quiescence detection
- LiveViz: visualization from a parallel program
- CharmDebug: a debugging tool
- Projections: Performance Analysis and Visualization, really nice, and a workhorse tool for Charm++ developers
- Messages (instead of marshalled parameters)
- Processor-aware constructs:
  - Groups: like a non-migratable chare array with one element on each “core”
  - Nodegroups: one element on each process

Summary

- Charm++ embodies an adaptive, introspective runtime system

- Many applications have been developed in it
  - NAMD, ChaNGa, OpenAtom, Episimdemics, …
  - Many MiniApps and third party apps

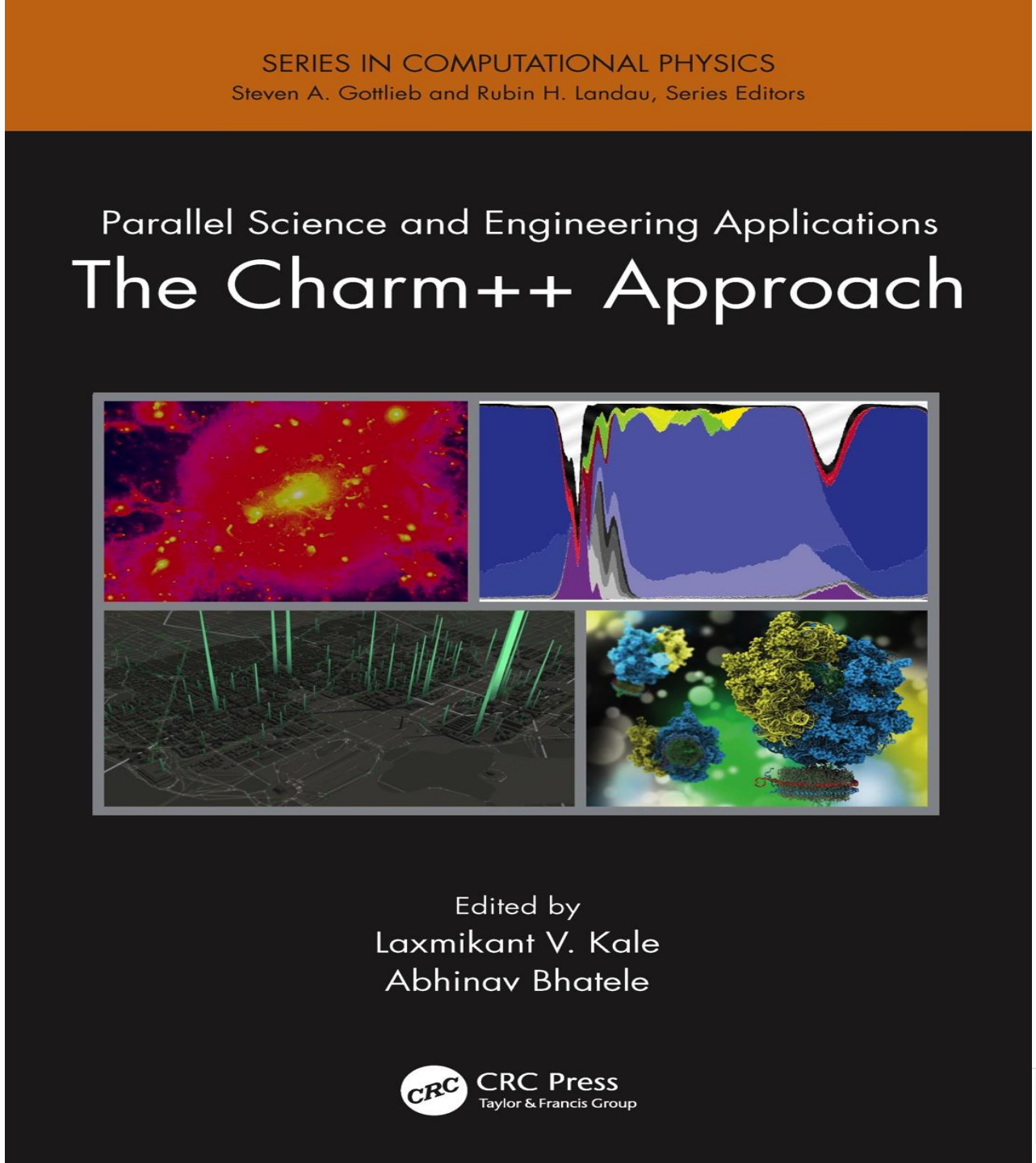
- Adaptivity developed for apps is useful for addressing exascale challenges
  - Resilience
  - Load Imbalance
  - Power/Temperature

More information

- A lecture series with instructional material coming soon
  - http://charmplusplus.org/ (under “Learn”)
- MiniApps (source code):
  - http://charmplusplus.org/miniApps/
- Research projects, papers, etc.
  - http://charm.cs.illinois.edu/
- Commercial support:
  - https://www.hpccharm.com/

Describes seven major applications developed using Charm++

More info on Charm++:
http://charm.cs.illinois.edu
Including the miniApps



Thank you!

Questions?

Extras

Adaptive MPI

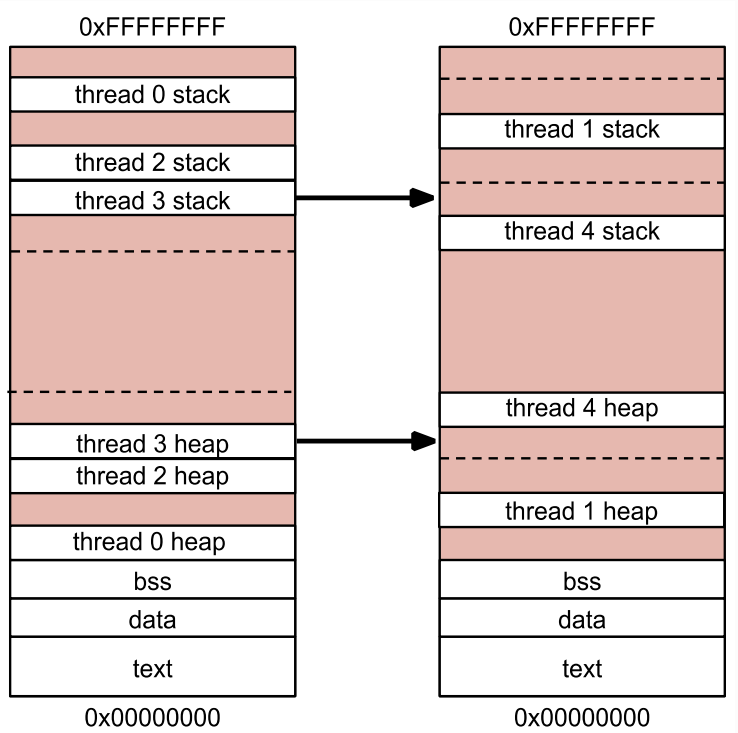
Process Virtualization

- AMPI virtualizes MPI “ranks”, implementing them as migratable user-level threads rather than OS processes
  - Benefits:
    - Communication/computation overlap
    - Cache benefits to smaller working sets
    - Dynamic load balancing
    - Lower latency messaging within a process
  - Disadvantages:
    - Global/static variables are shared by all threads in an OS process scope
      - Not an issue for new applications
      - AMPI provides support for automating this at compile/run-time
      - Ongoing work to fully automate

Dynamic Load Balancing

- AMPI ranks are migratable across address spaces at runtime
  - Add a call to *AMPI_Migrate*(*MPI_Info*) in the application's main iterative loop

- Isomalloc memory allocator
  - No need for the user to explicitly write de/serialization (PUP) routines
  - Memory allocator migrates all heap data and stack transparently
  - Works on all 64-bit platforms except BGQ & Windows



Saving Cooling Energy

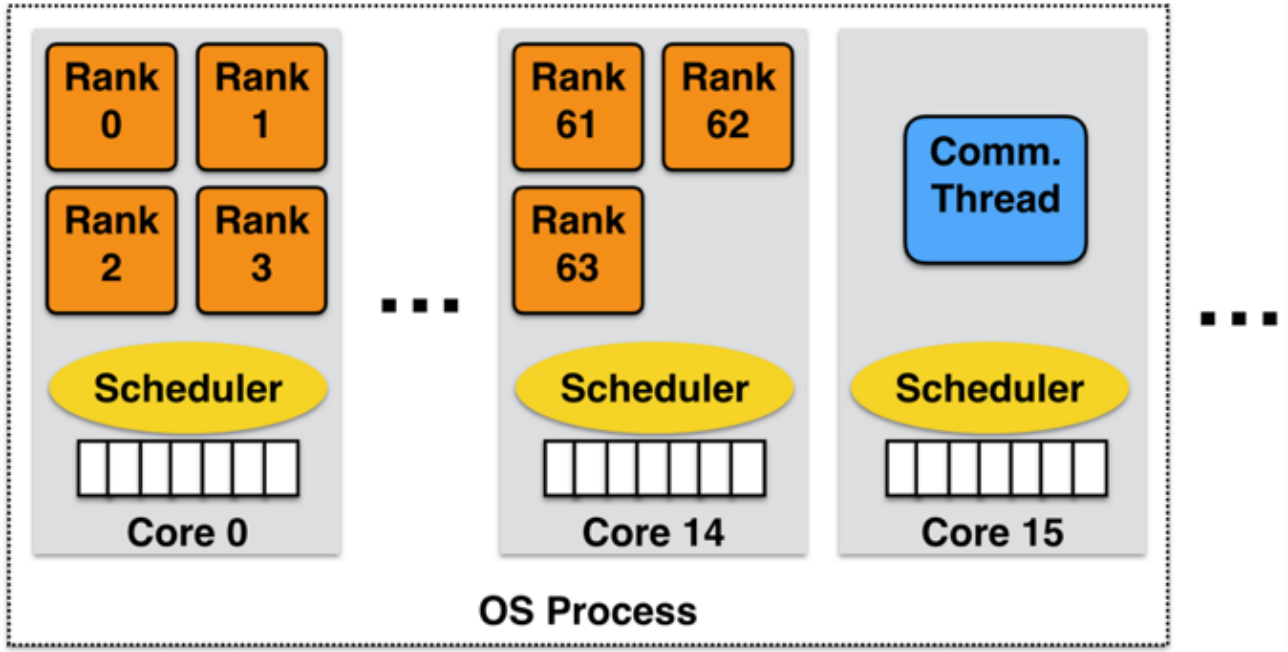
- Easy: increase A/C setting
  - But: some cores may get too hot
- So, reduce frequency if temperature is high (DVFS)
  - Independently for each chip
- *But,* this creates a load imbalance!
- No problem, we can handle that:
  - Migrate objects away from the slowed-down processors
  - Balance load using an existing strategy
  - Strategies take speed of processors into account
- Implemented in experimental version
  - SC 2011 paper, IEEE TC paper
- Several new power/energy-related strategies
  - PASA '12: Exploiting differential sensitivities of code segments to frequency change

Fault Tolerance

- AMPI ranks can be migrated to persistent storage or in remote memories for fault tolerance
  - Storage can be Disk, SSD, NVRAM, etc.
- The runtime uses a scalable fault detection algorithm and restarts automatically on a failure
  - Restart is online, within the same job
- Checkpointing strategy is specified by passing a different *MPI_Info* to *AMPI_Migrate*()

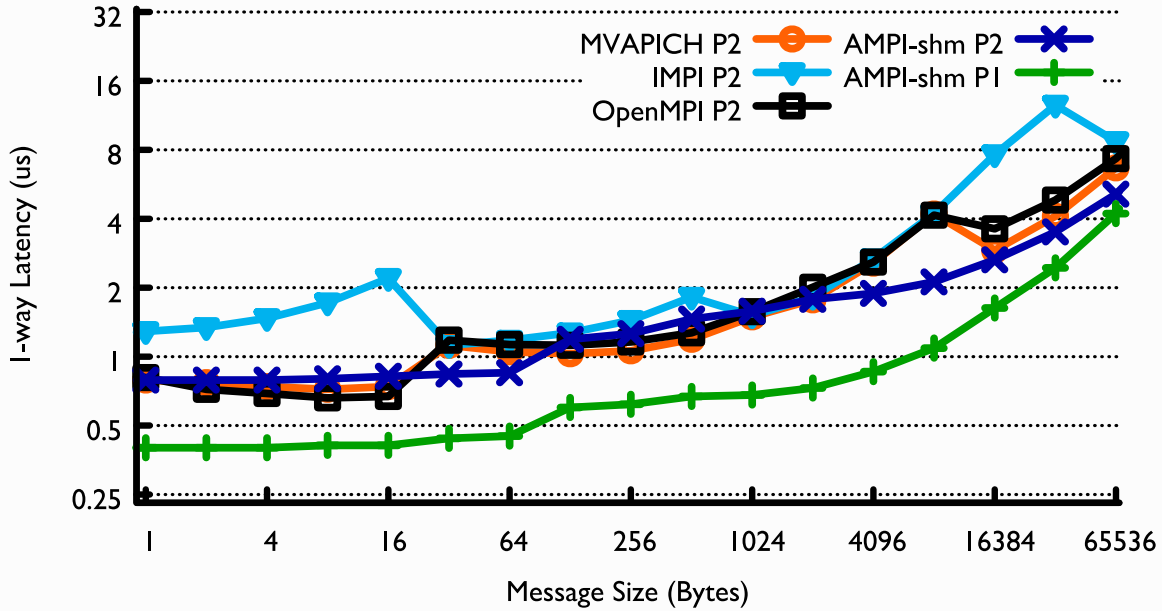
Communication Optimizations

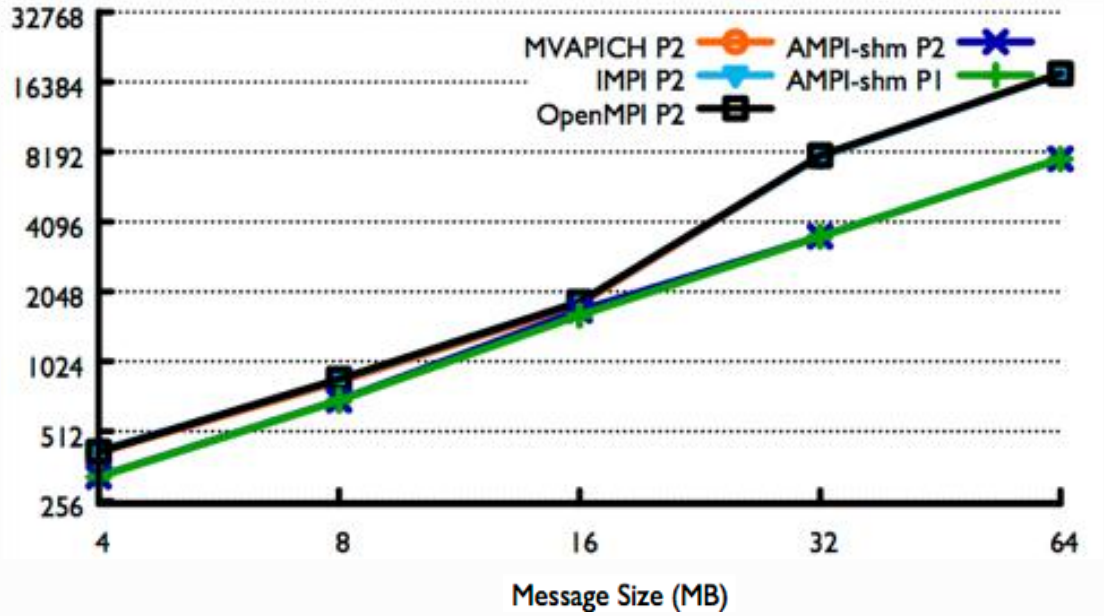
- Along with overlapping communication, AMPI optimizes for communication locality:
  - Within a core, within a process, within a host, etc.
  - Communication-aware load balancers can maximize locality



Communication Optimizations

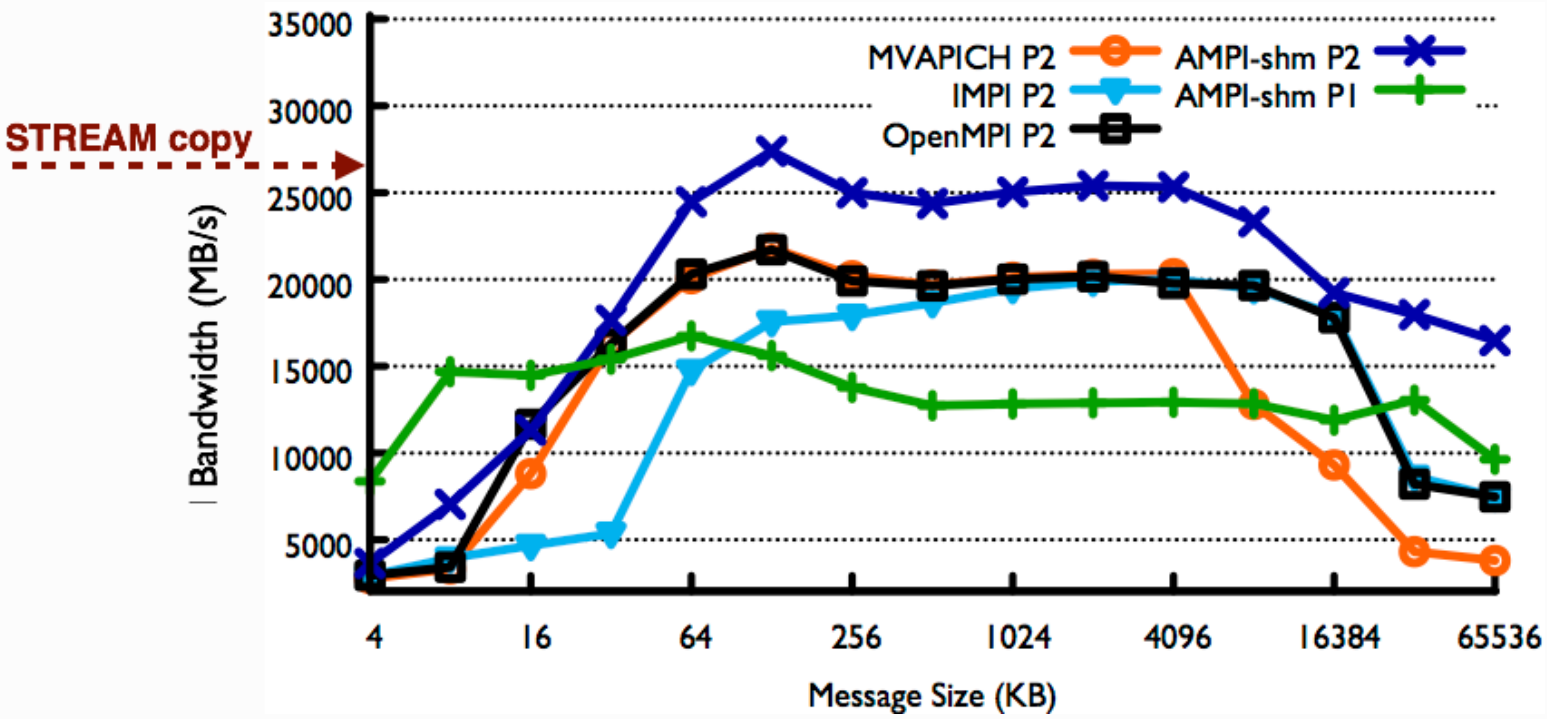
- AMPI outperforms process-based MPIs for messages within a process
  - All messaging is done in user-space: no kernel involvement
    - Below: OSU MPI Benchmarks on Quartz, an Intel Omni-Path cluster at LLNL





Communication Optimizations

- AMPI outperforms process-based MPIs for messages within a process
  - Utilize the full memory bandwidth on a node for messaging



Compiling & Running AMPI Programs

- To compile an AMPI program:
  - charm/bin/ampicc –o pgm pgm.o
  - For migratability, link with: -memory isomalloc
  - For LB strategies, link with: –module CommonLBs

- To run an AMPI job, specify the # of virtual processes (+vp)
  - ./charmrun +p 1024 ./pgm
  - ./charmrun +p 1024 ./pgm +vp 16384
  - ./charmrun +p 1024 ./pgm +vp 16384 +balancer RefineLB

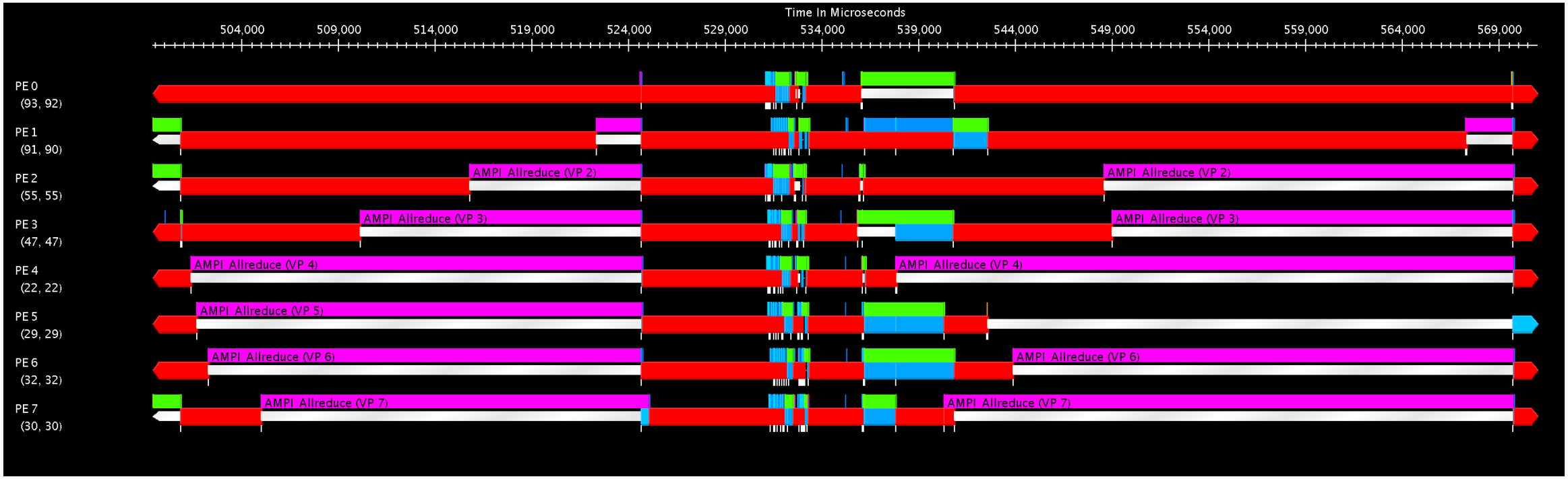
Case Study

- LULESH proxy-application (LLNL)
  - Shock hydrodynamics on an unstructured mesh
  - With artificial load imbalance included to test runtimes

- No mutable global/static variables: can run on AMPI as is
  1. Replace *mpicc* with *ampicc*
  2. Link with "-module CommonLBs –memory isomalloc"
  3. Run with # of virtual processes and a load balancing strategy:
     - ./charmrun +p 2048 ./lulesh2.0 +vp 16384 +balancer GreedyLB

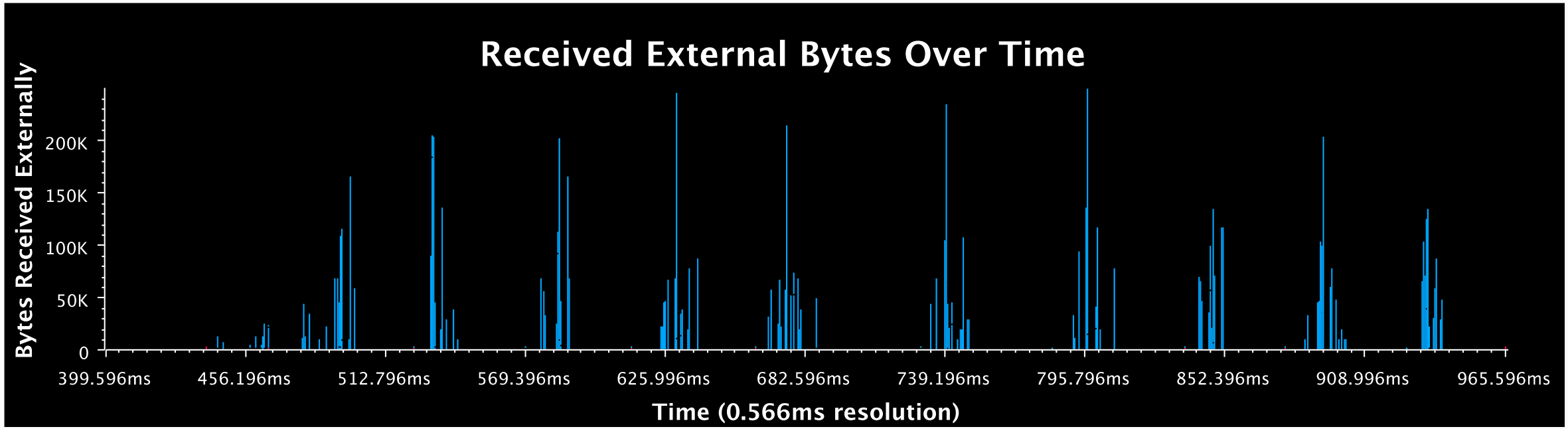
LULESH: Without Virtualization & LB

- Load imbalance appears during pt2pt messaging and in MPI_Allreduce each timestep



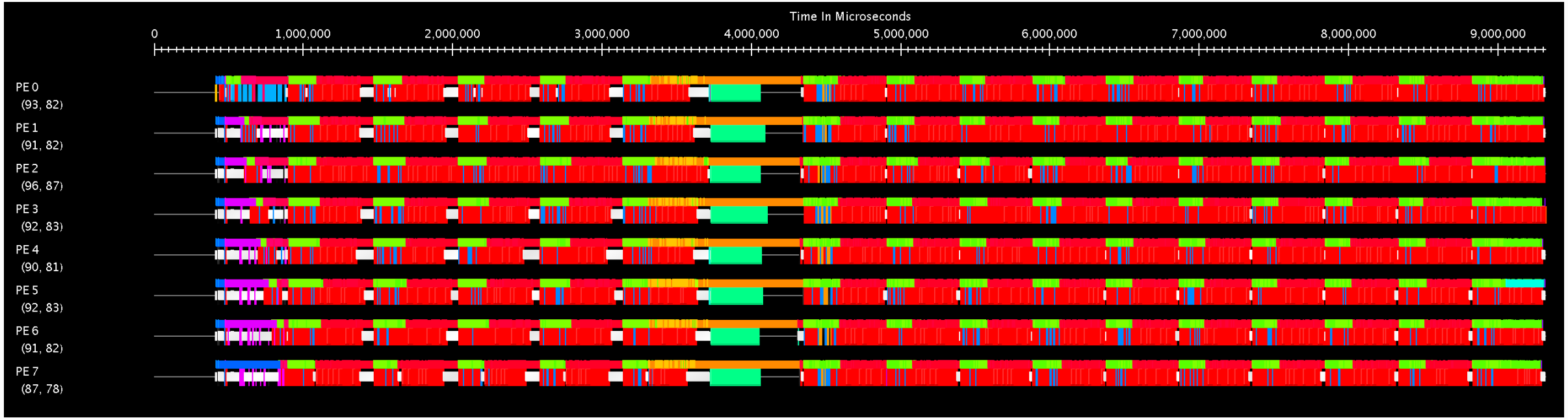
LULESH: Without Virtualization & LB

- Communication/computation cycles mean the network is underutilized most of the time



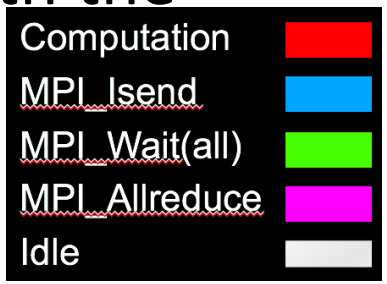
LULESH: With 8x Virtualization & LB

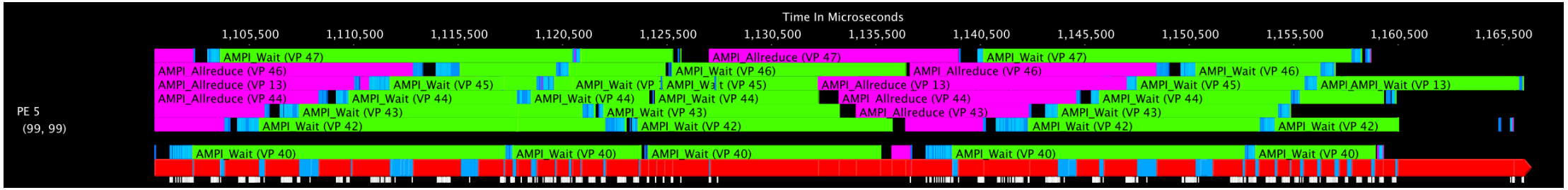
- Most of the communication time is overlapped by computation after load balancing



LULESH: With 8x Virtualization & LB

- The communication of each virtual rank is overlapped with the computation of others scheduled on the same core

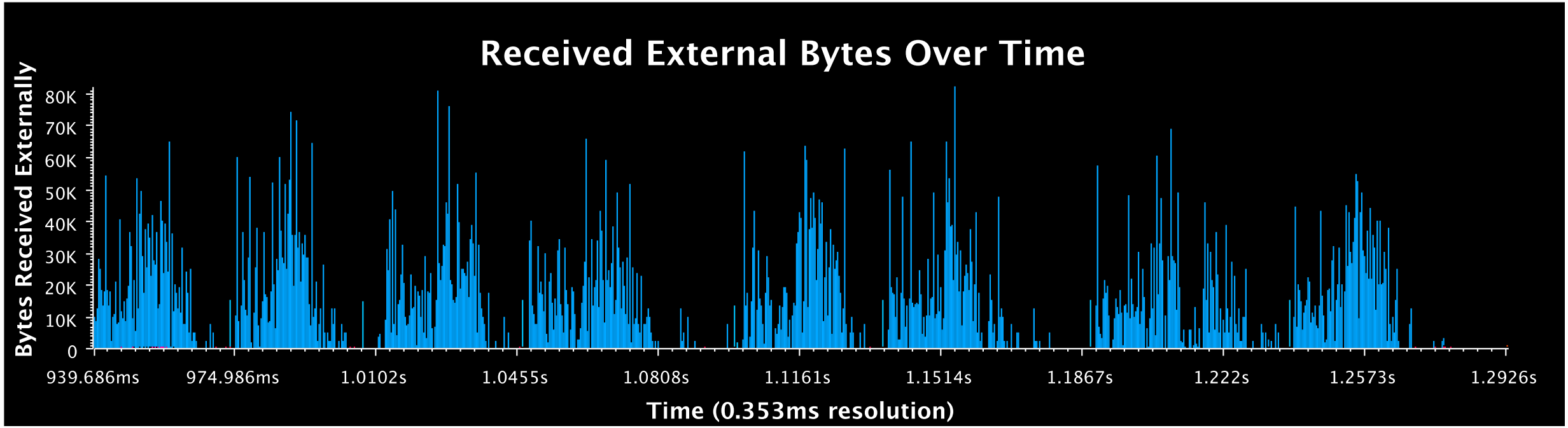




  - Projections allows viewing all virtual ranks on a PE, not only what is currently scheduled on one
    - In Projections Timeline, select: *View -> Show Nested Bracketed User Events*

LULESH: With 8x Virtualization & LB

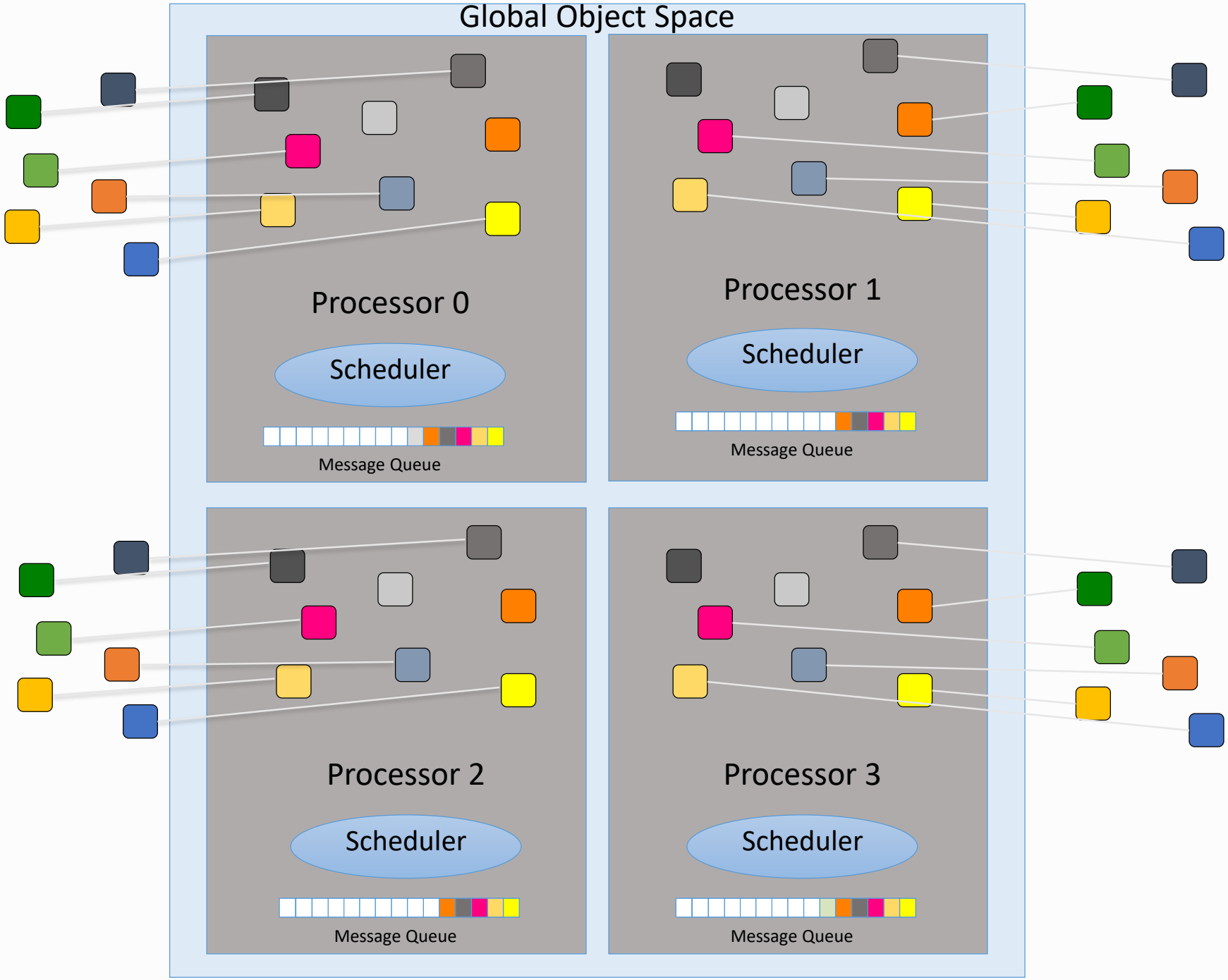
- Communication is spread over the whole timestep
  - Peak network bandwidth used is reduced by 3x



AMPI Summary

- AMPI provides the dynamic RTS support of Charm++ with the familiar API of MPI
  - Communication optimizations
  - Dynamic load balancing
  - Automatic fault tolerance
  - Checkpoint/restart
  - OpenMP runtime integration

- See the AMPI Manual for more info.

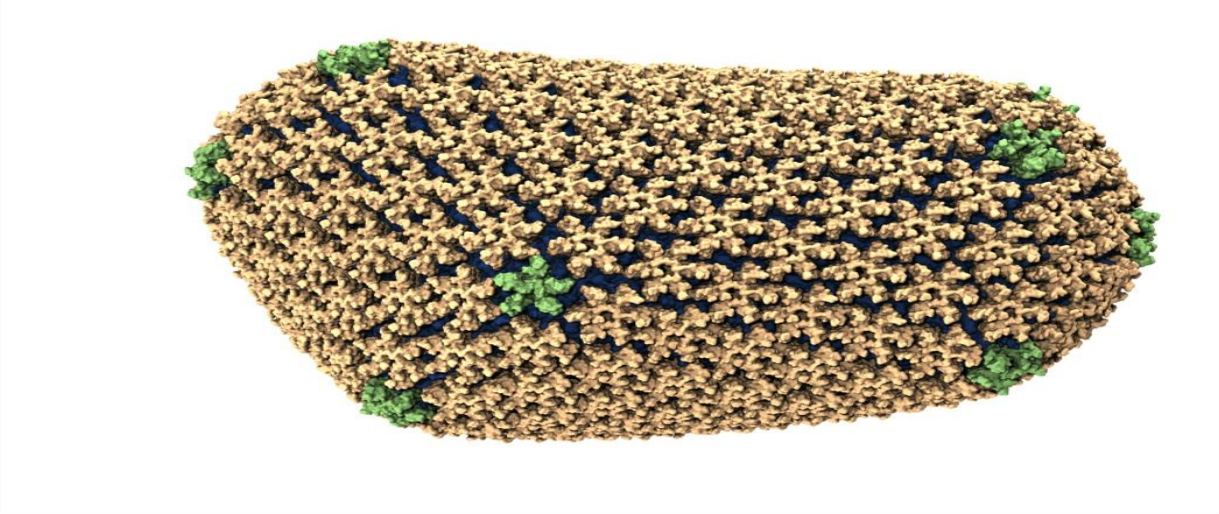

Global Object Space
Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue

Control flow within chare

- Structured dagger notation
  - Provides a script-like language for expressing dag of dependencies between method invocations and computations
- Threaded Entry methods
  - Allows entry methods to block without blocking the PE
  - Supports futures, and
  - ability to suspend/resume threads

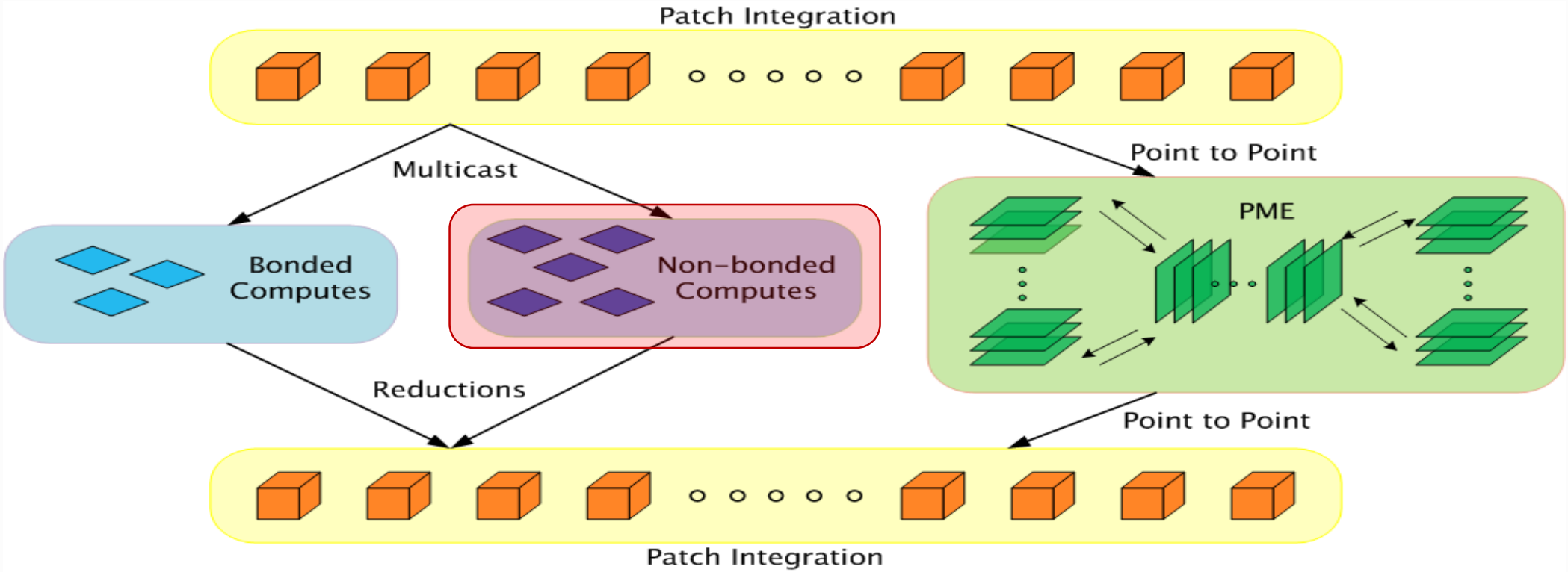
NAMD: Biomolecular Simulations

- Collaboration with K. Schulten
- With over 70,000 registered users
- Scaled to most top US supercomputers
- In production use on supercomputers and clusters and desktops
- Gordon Bell award in 2002



Determination of the structure of HIV capsid by researchers including Prof Schulten

Parallelization using Charm++



ChaNGa: Parallel Gravity

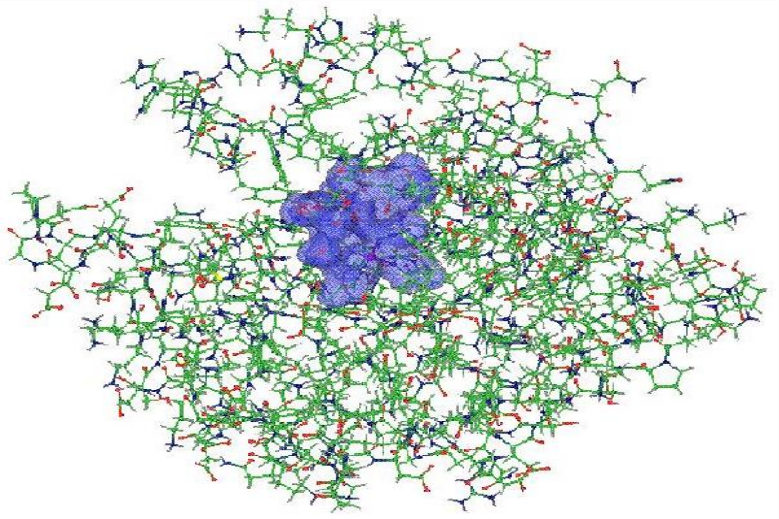
- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomp
  - Geometry has better aspect ratios, so you “open” up fewer nodes
  - But is not used because it leads to bad load balance
  - Assumption: one-to-one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

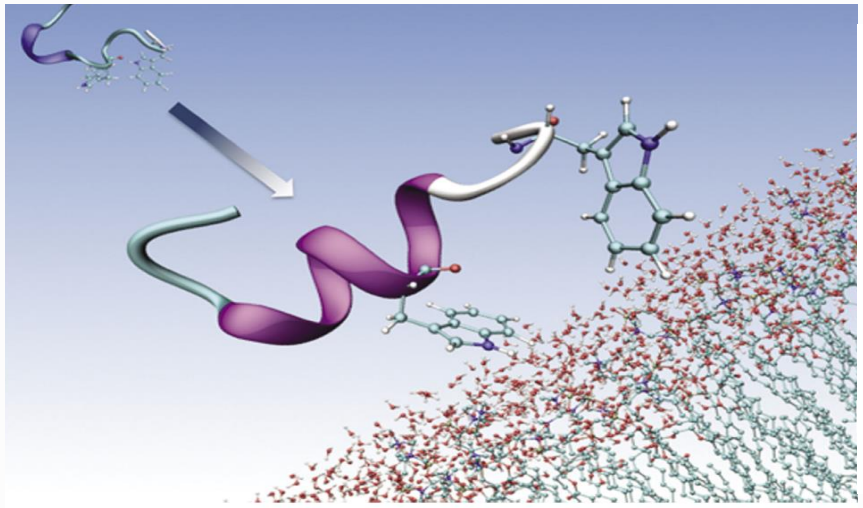
Evolution of Universe and Galaxy Formation

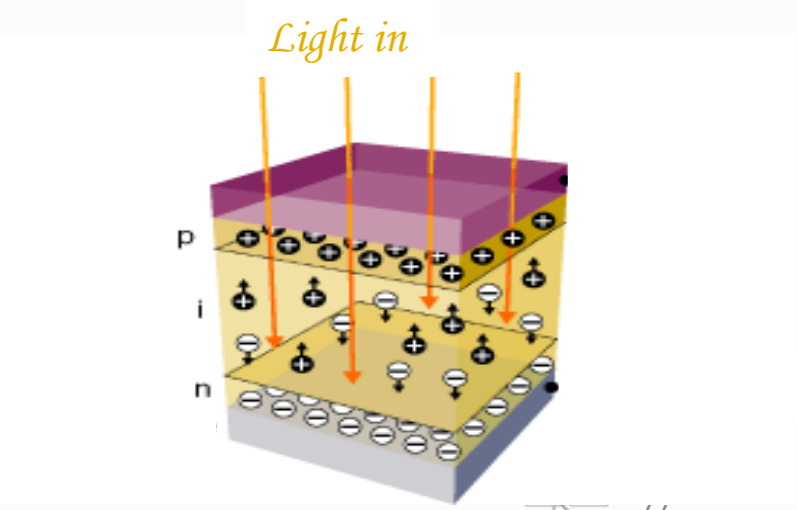
With Charm++: Use Oct-Tree, and let Charm++ map subtrees to processors

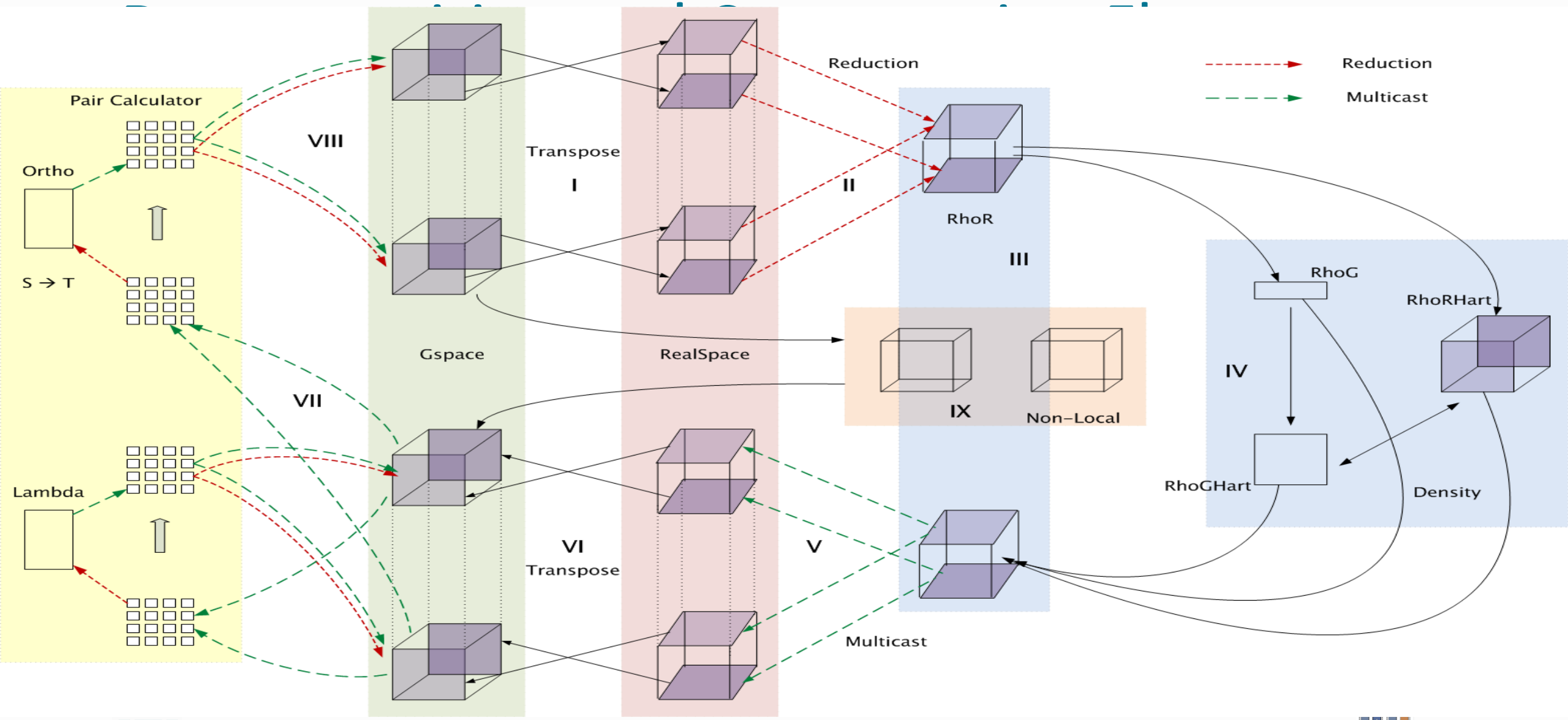
OpenAtom: On the fly ab initio molecular dynamics on the ground state surface with instantaneous GW-BSE level spectra

PIs: G.J. Martyna, IBM; S. Ismail-Beigi, Yale; L. Kale, UIUC;
Team: Q. Li, IBM, M. Kim, Yale; S. Mandal, Yale;
E. Bohm, UIUC; N. Jain, UIUC; M. Robson, UIUC;
E. Mikida, UIUC; P. Jindal, UIUC; T. Wicky, UIUC.








Reduction
Reduction
Multicast
Pair Calculator
Ortho
S → T
Lambda
VIII
VII
Transpose
I
II
RhoR
III
Gspace
RealSpace
IX
Non-Local
RhoG
RhoRHart
IV
RhoGHart
Density
VI
Transpose
V
Multicast

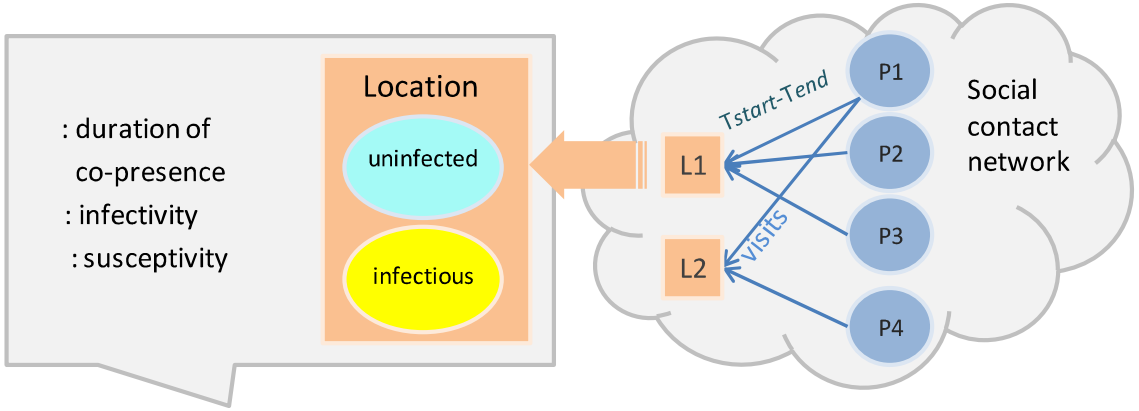
Episimdemics

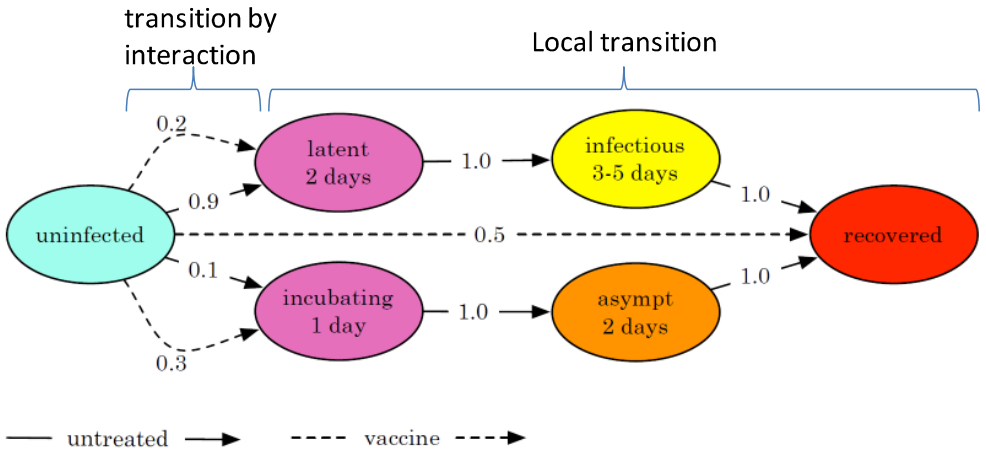
- Simulation of spread of contagion
  - Code by Madhav Marathe, Keith Bisset, .. Vtech
  - Original was in MPI
- Converted to Charm++
  - Benefits: asynchronous reductions improved performance considerably

Simulat

EpiS

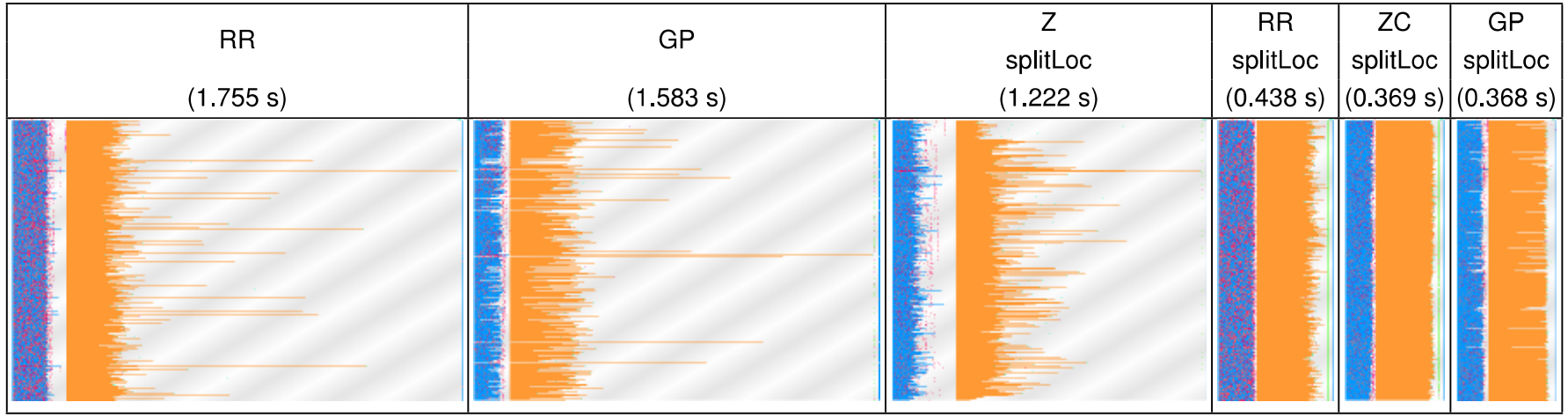
- Agent-
- Realis
- Interve
- Co-ev behav





[1] C. Barrett et al.,"EpiSimdemics: An Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks," SC08
[2] K. Bisset et al., "Modeling Interaction Between Individuals, Social Networks and Public Policy to Support Public Health Epidemiology," WSC09.

Load distribution (Vulcan)



splitLoc: **no peak in location computation**
Z-splitLoc: **no load balance**

GP: **shorter person phase**
ZC-splitLoc: similar perf. w/ GP-splitLoc

Blue: person computation
Red: receiver's msg handling
Orange: location computation

X-axis: **Time** Y-axis: **Processor**

Timeline of an iteration from sampled subset of 332 cores of total 4K using Michigan data on Vulcan

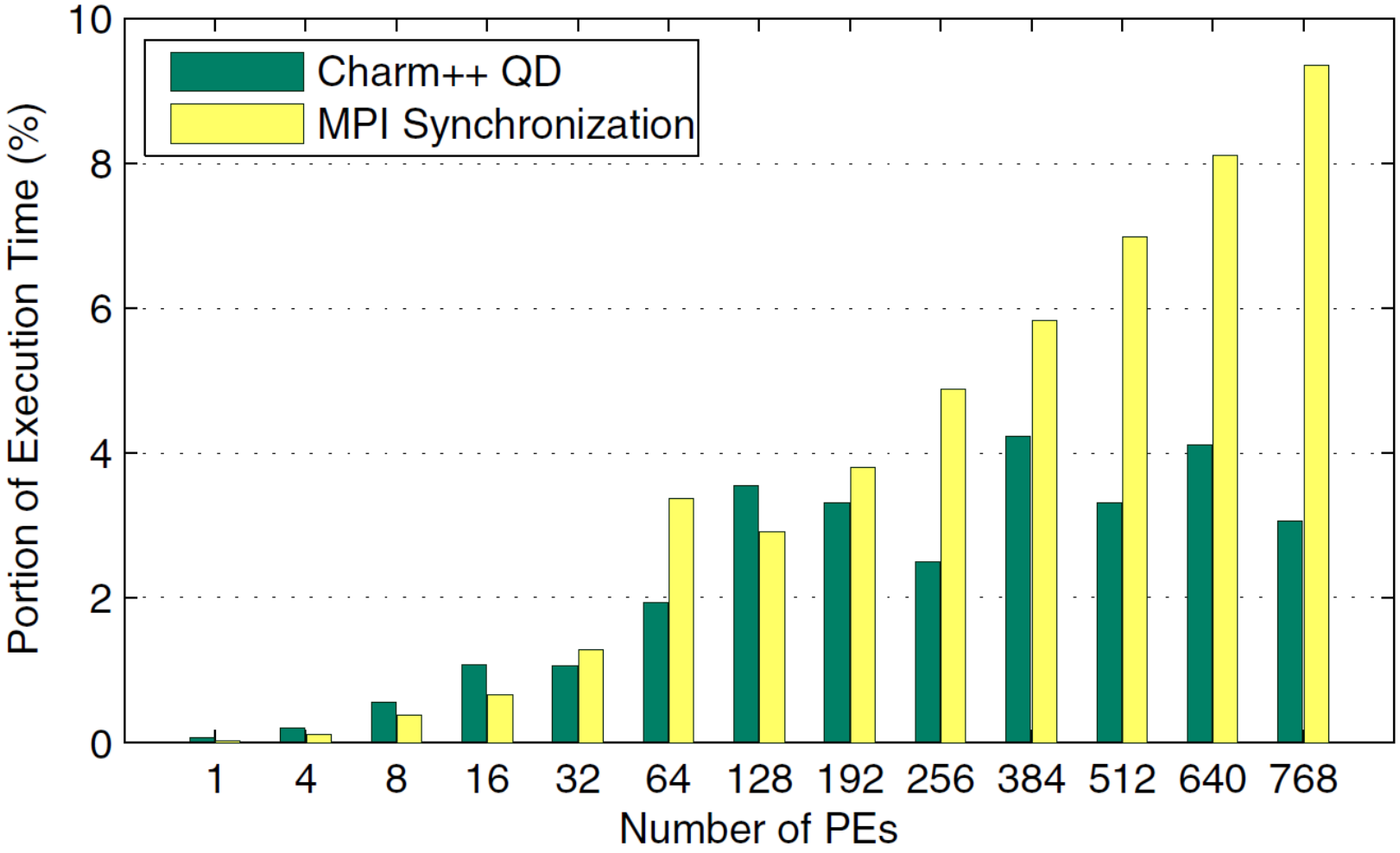
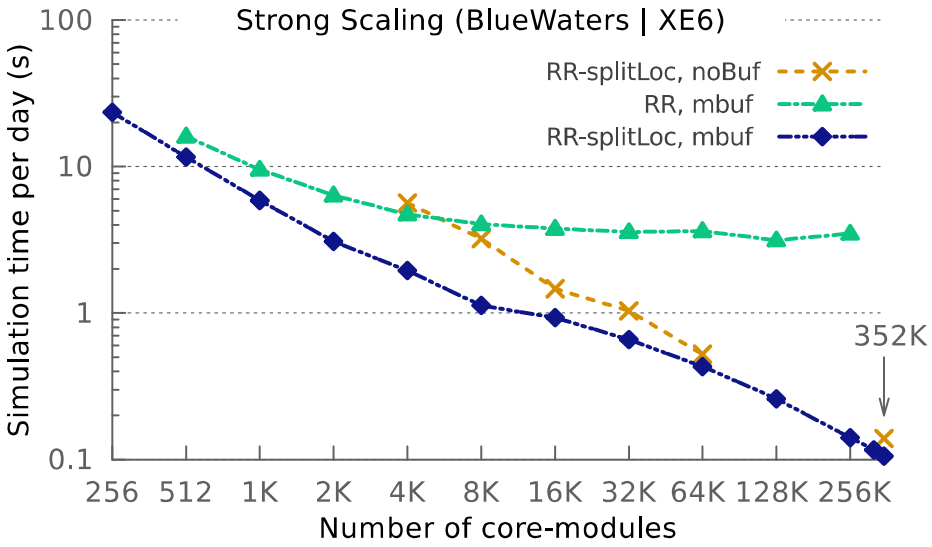
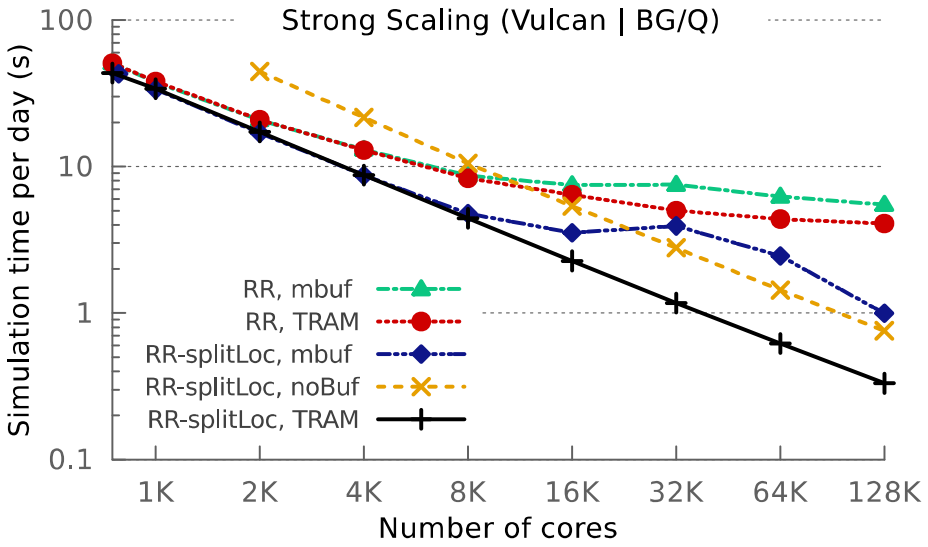
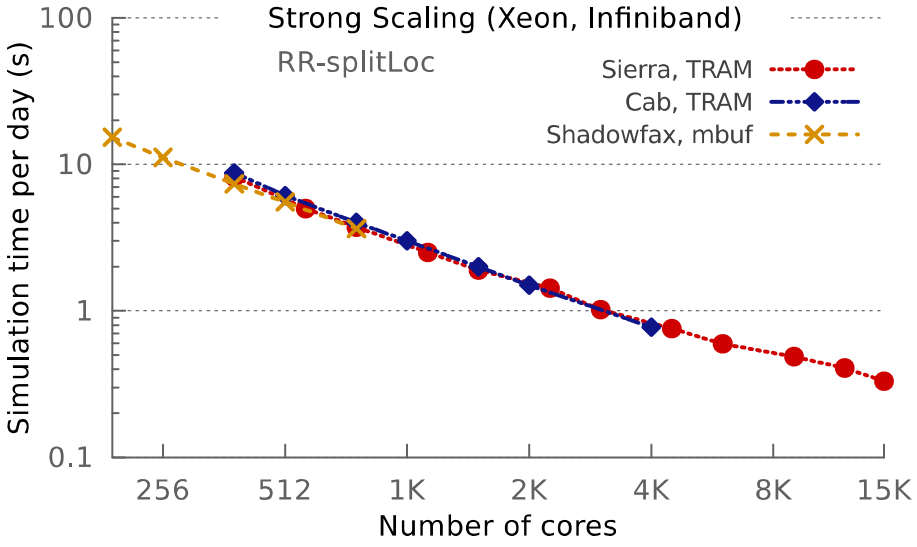


Figure 10. The synchronization cost using contribute() and QD method takes at most 4.23% of the total execution time while the MPI synchronization cost linearly increases up to 14.5% as the number of PEs used increases for simulating Arkansas population.

Strong scaling performance with the largest data set







- Contiguous US population data
- **XE6**: **the largest scale** (**352K cores**)
- **BG/Q**: good scaling up to 128K cores
- Strong scaling helps timely reaction to pandemic

Possible Future Collaboration of Interest

- **A Dynamic Unified Framework for Hurricane Storm Surge Analysis and Prediction Spanning across the Coastal Floodplain and Ocean**
  - With Joannes Westerink (Notre Dame), Ethan Kubatko, Clint Dawson (Utexas)
  - To :meld hydrology, hydraulics, and waves to a unified adaptable computational framework that uses unstructured grids spanning from the deep ocean to upland areas and across the coastal floodplain”
  - Idea will be apply Charm++ load balancing capabilities to this problem

Available at: http://charmplusplus.org/miniApps/

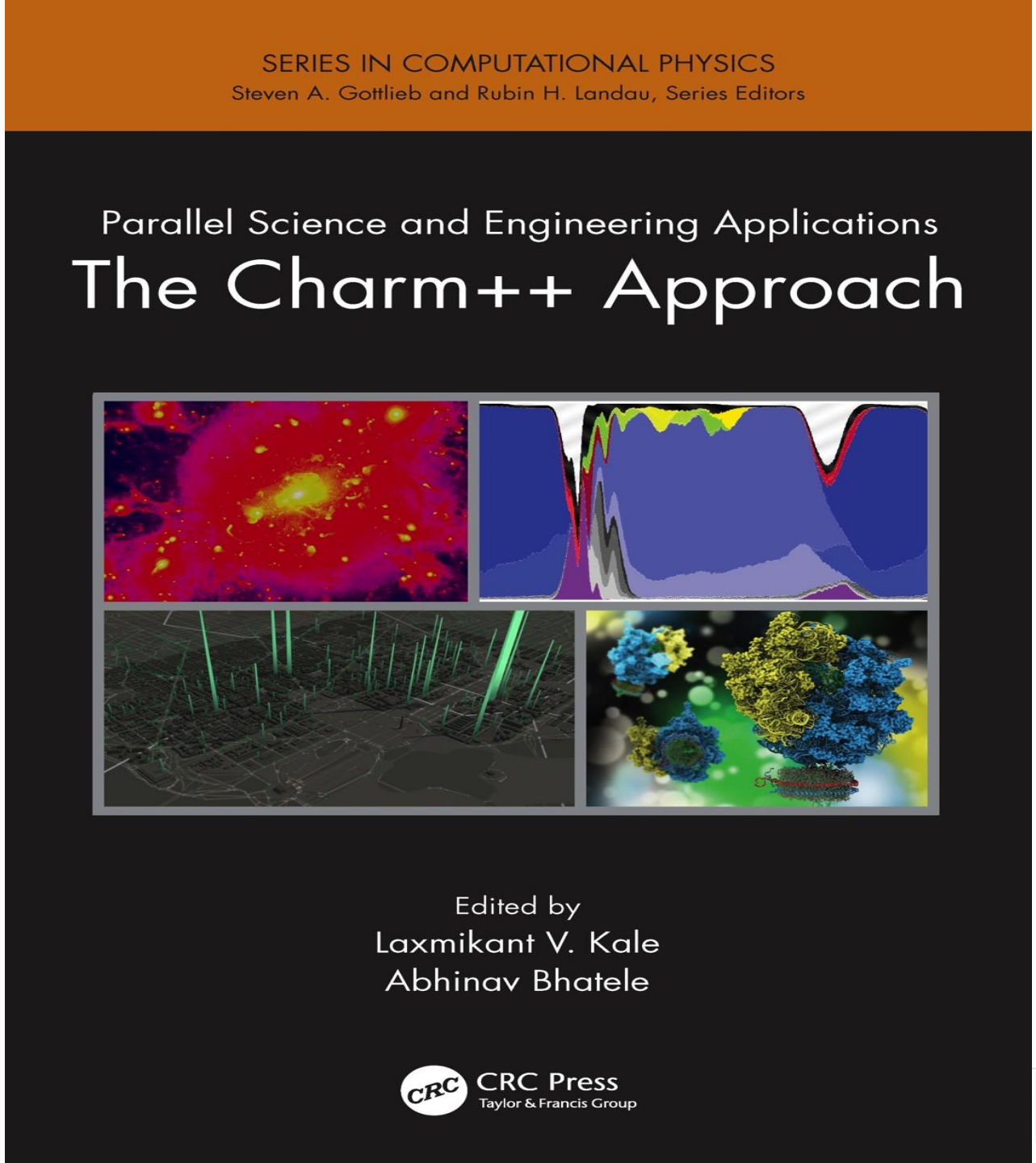
Mini-App	Features	Machine	Max cores
AMR	Overdecomposition, Custom array index, Message priorities, Load Balancing, Checkpoint restart	BG/Q	131,072
LeanMD	Overdecomposition, Load Balancing, Checkpoint restart, Power awareness	BG/P BG/Q	131,072 32,768
Barnes-Hut (n-body)	Overdecomposition, Message priorities, Load Balancing	Blue Waters	16,384
LULESH 2.02	AMPI, Over-decomposition, Load Balancing	Hopper	8,000
PDES	Overdecomposition, Message priorities, TRAM	Stampede	4,096

More MiniApps

Mini-App	Features	Machine	Max cores
1D FFT	Interoperable with MPI	BG/P BG/Q	65,536 16,384
Random Access	TRAM	BG/P BG/Q	131,072 16,384
Dense LU	SDAG	XT5	8,192
Sparse Triangular Solver	SDAG	BG/P	512
GTC	SDAG	BG/Q	1,024
SPH		Blue Waters	-

Describes seven major applications developed using Charm++

More info on Charm++:
http://charm.cs.illinois.edu
Including the miniApps



PARM:Power Aware Resource Manager

- Charm++ RTS facilitates malleable jobs
- PARM can improve throughput under a fixed power budget using:
  - overprovisioning (adding more nodes than conventional data center)
  - RAPL (capping power consumption of nodes)
  - Job malleability and moldability