Charm++ and Adaptive MPI

Challenges in Parallel Programming

Challenges in Parallel Programming

- Applications are getting more sophisticated
  - Adaptive refinement
  - Multi-scale, multi-module, multi-physics
  - E.g. load imbalance emerges as a huge problem for some apps

Challenges in Parallel Programming

- Applications are getting more sophisticated
  - Adaptive refinement
  - Multi-scale, multi-module, multi-physics
  - E.g. 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

Challenges in Parallel Programming

- Applications are getting more sophisticated
  - Adaptive refinement
  - Multi-scale, multi-module, multi-physics
  - E.g. 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++?

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

What is Charm++?

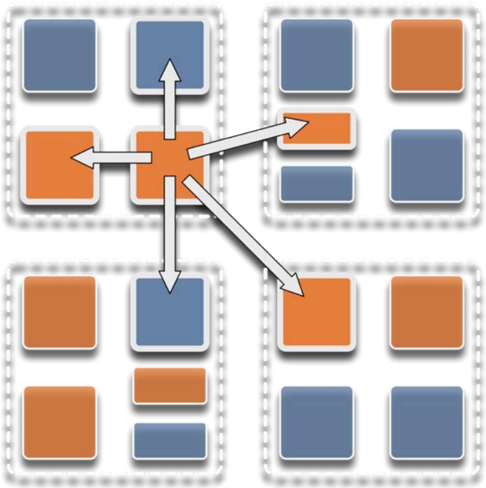
- Charm++ is a generalized approach to writing parallel programs
  - An alternative to the likes of MPI, UPC, GA etc.
  - But not to sequential languages such as C, C++, and Fortran
- Represents:
  - The style of writing parallel programs
  - The runtime system
  - And the entire ecosystem that surrounds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

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

Realization of This Model in Charm++

- Overdecomposed entities: chares
  - Chares are C++ objects

Realization of This Model in Charm++

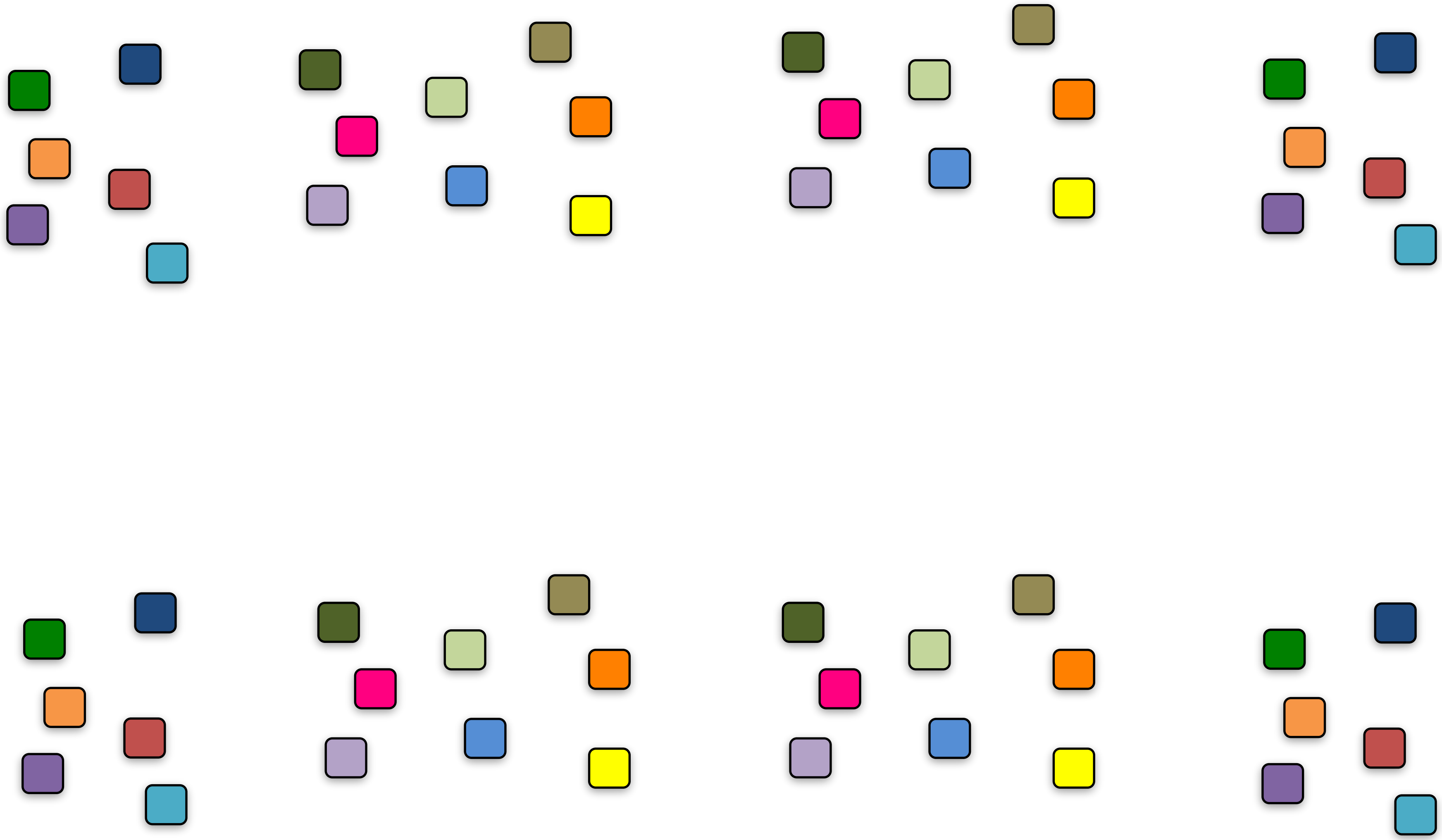
- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as “entry” methods
    - Which can be invoked asynchronously by remote chares

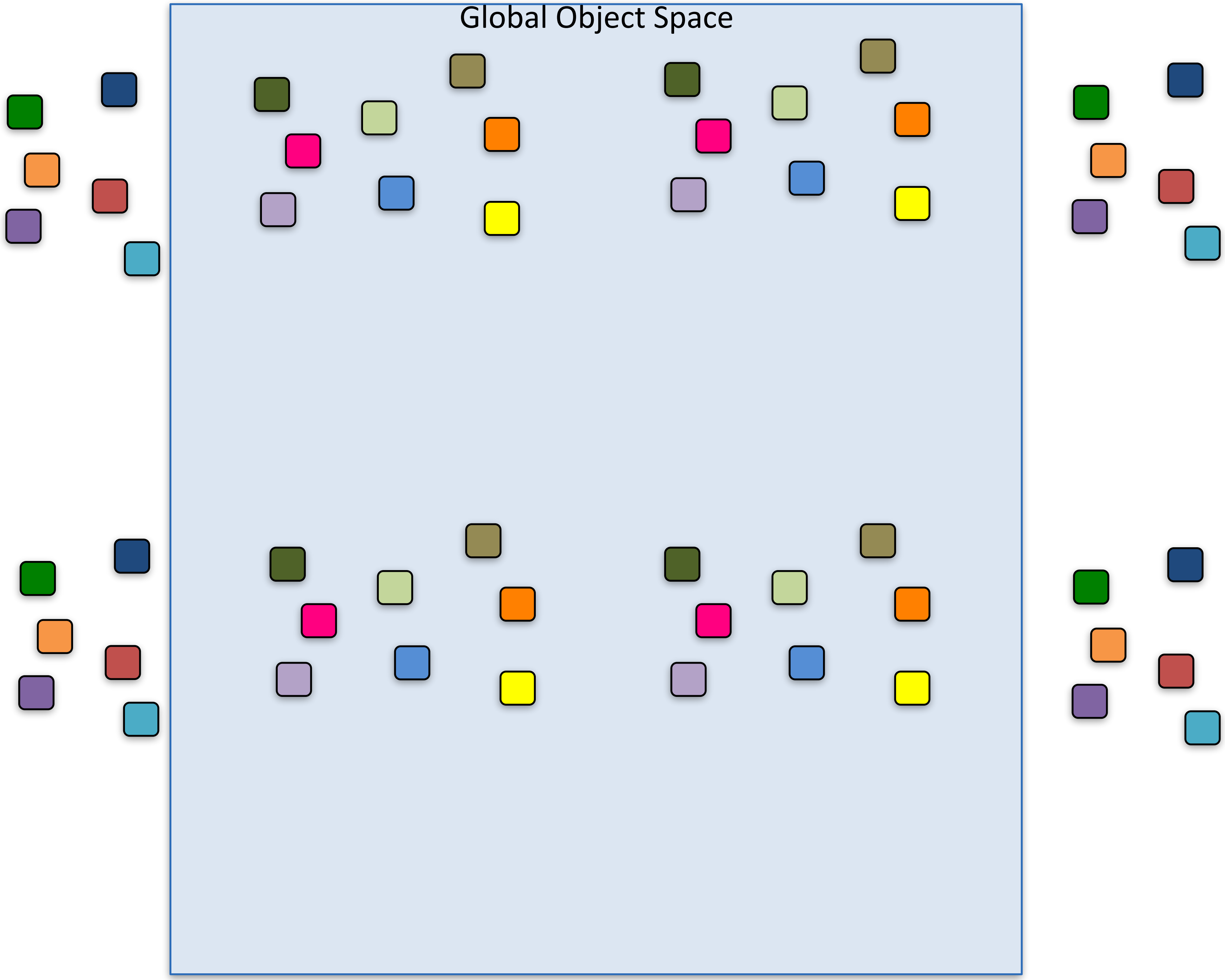
Realization of This Model in Charm++

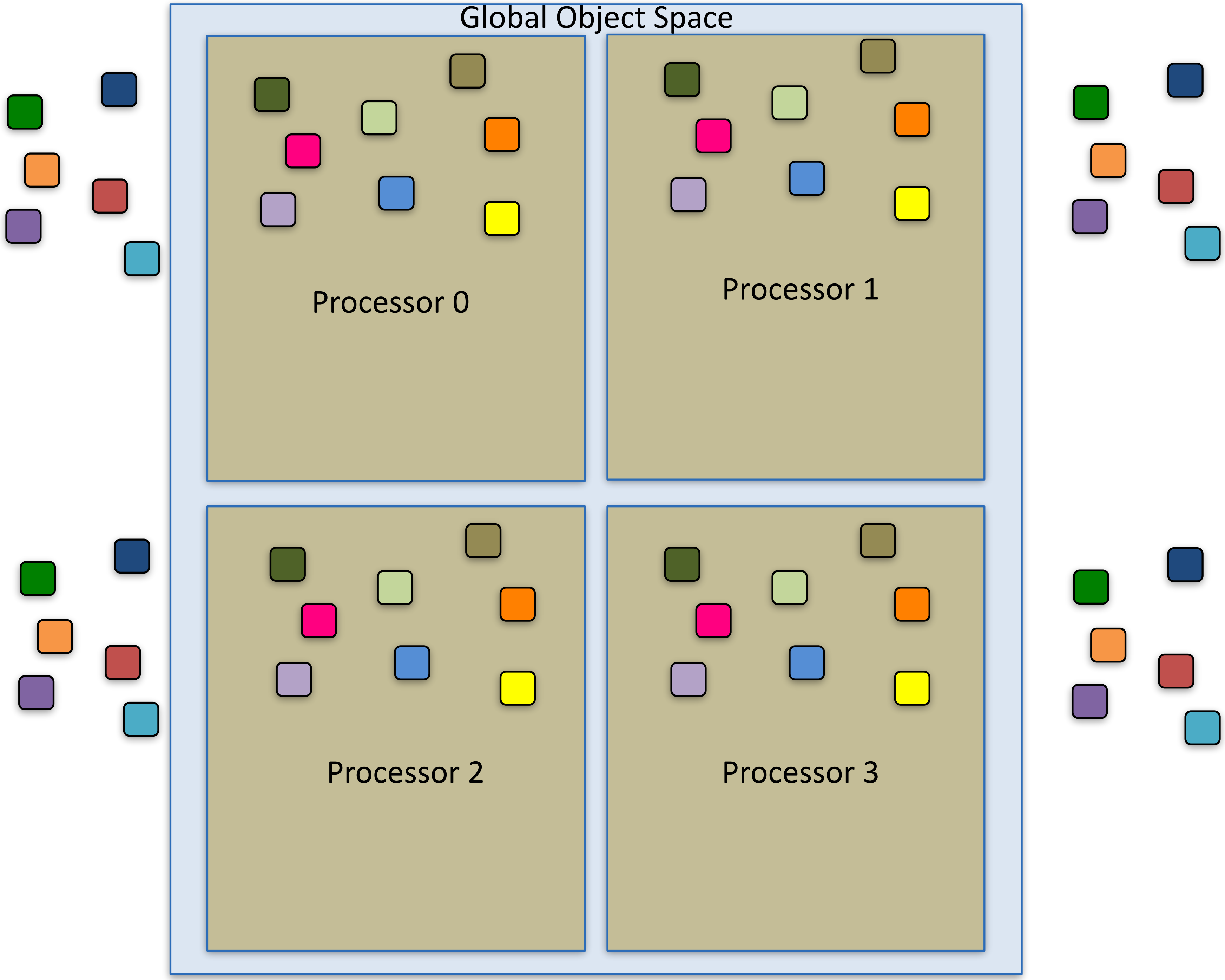
- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as “entry” methods
    - Which can be invoked asynchronously by remote chares
  - Chares are organized into indexed collections
    - Each collection may have its own indexing scheme
      - 1D, …, 6D
      - Sparse
      - Bitvector or string as an index

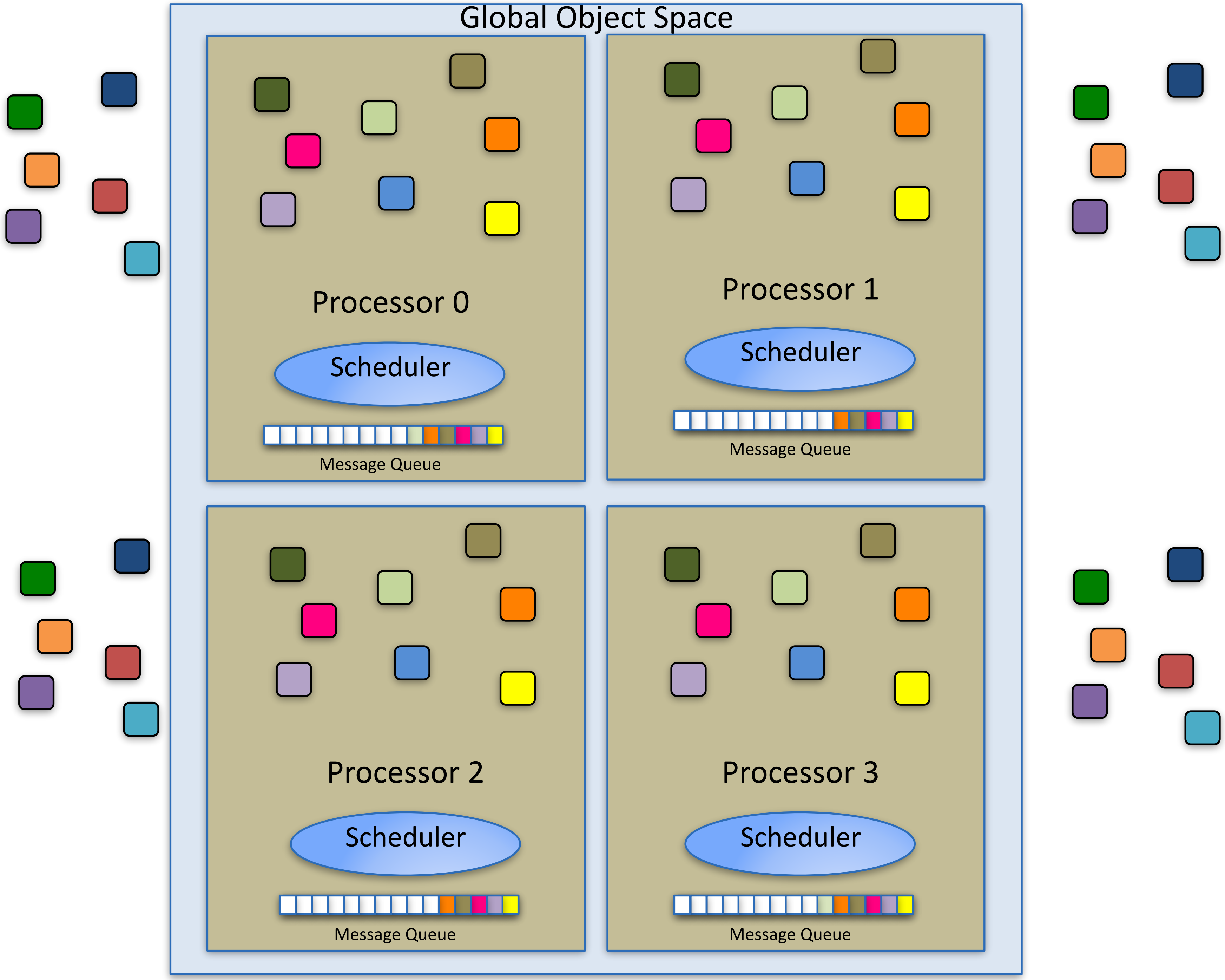
Realization of This Model in Charm++

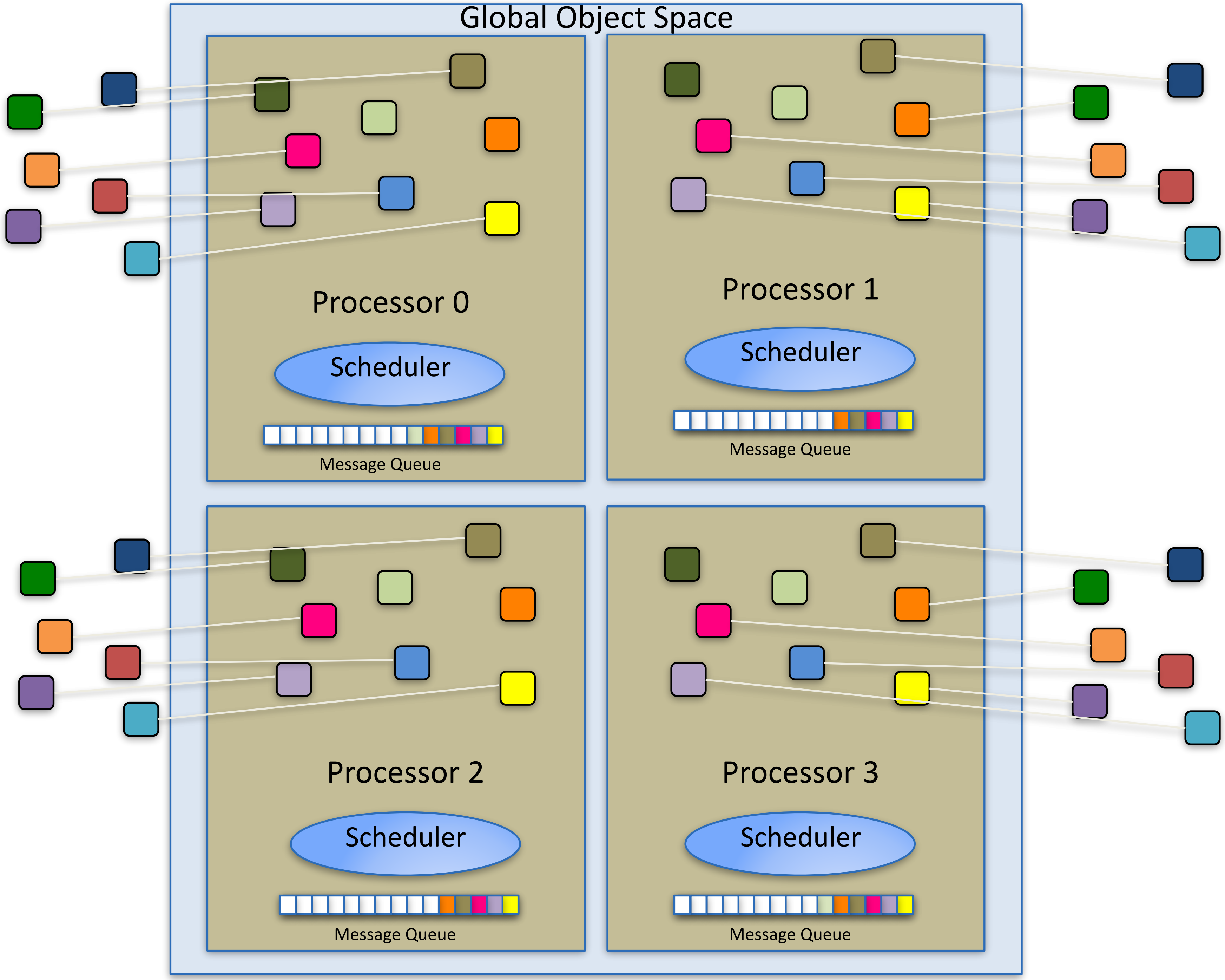
- Overdecomposed entities: chares
  - Chares are C++ objects
  - With methods designated as "entry" methods
    - Which can be invoked asynchronously by remote chares
  - Chares are organized into indexed collections
    - Each collection may have its own indexing scheme
      - 1D, ..., 6D
      - Sparse
      - 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.

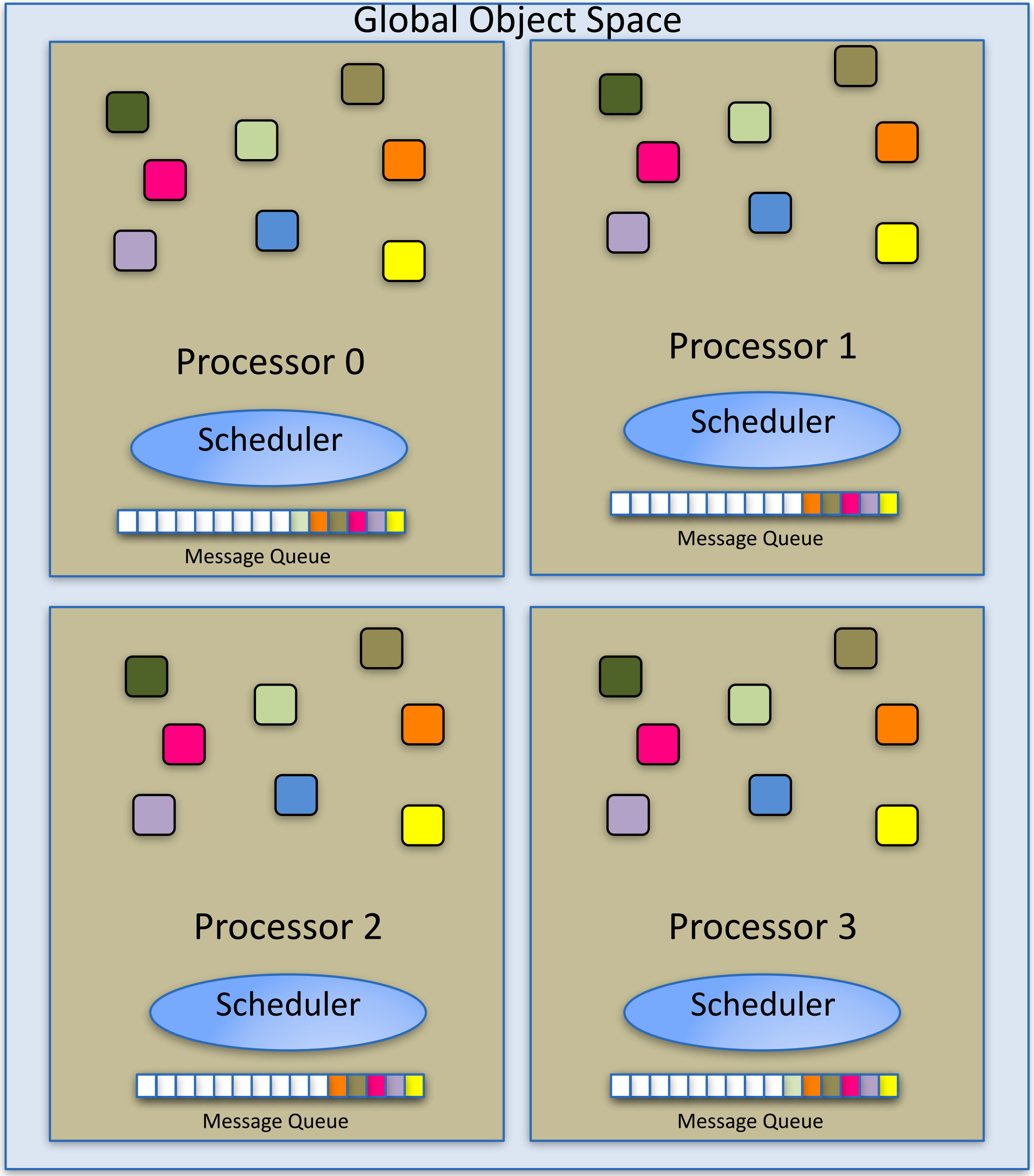



Global Object Space

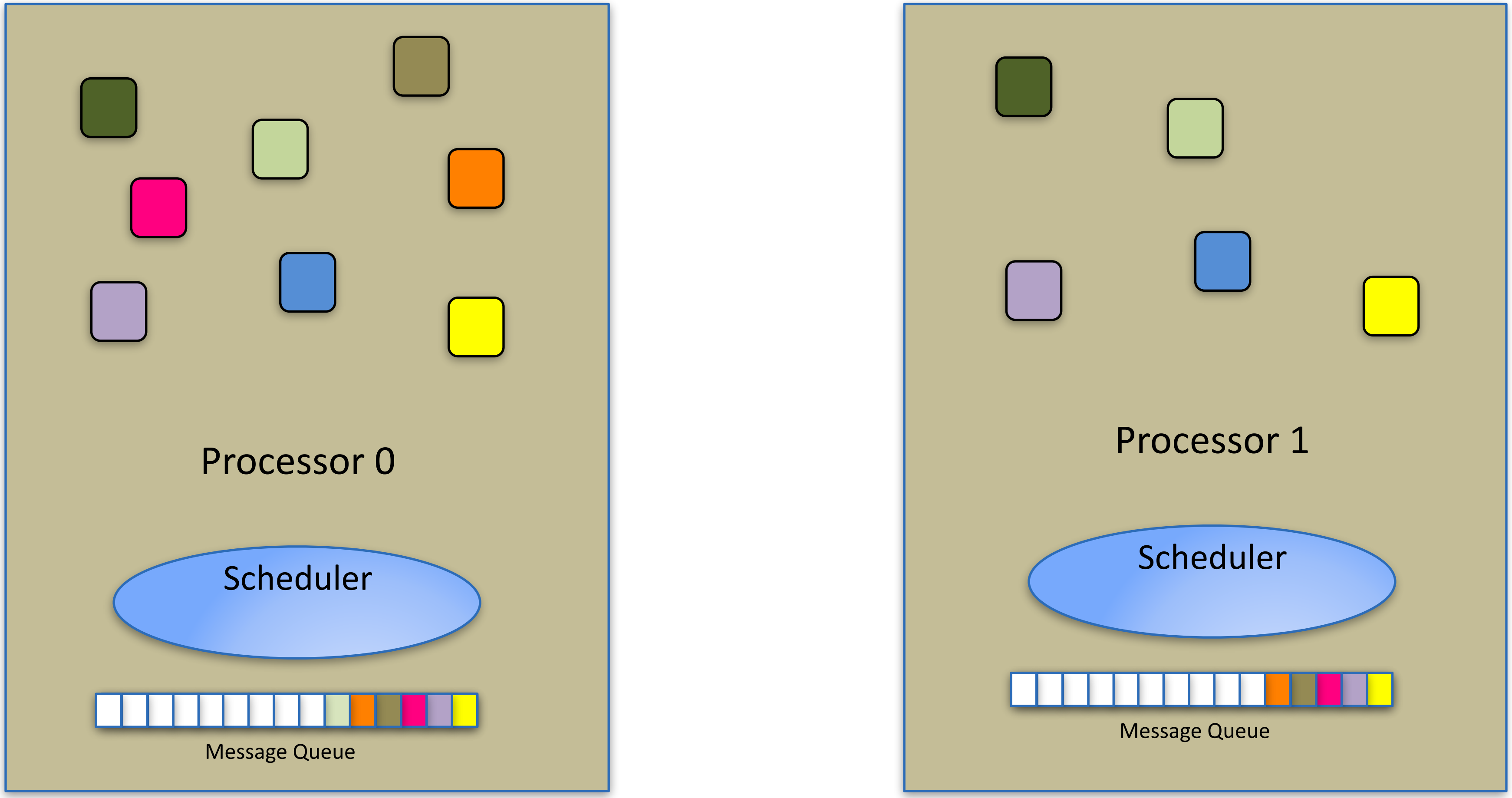

Global Object Space
Processor 0
Processor 1
Processor 2
Processor 3


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

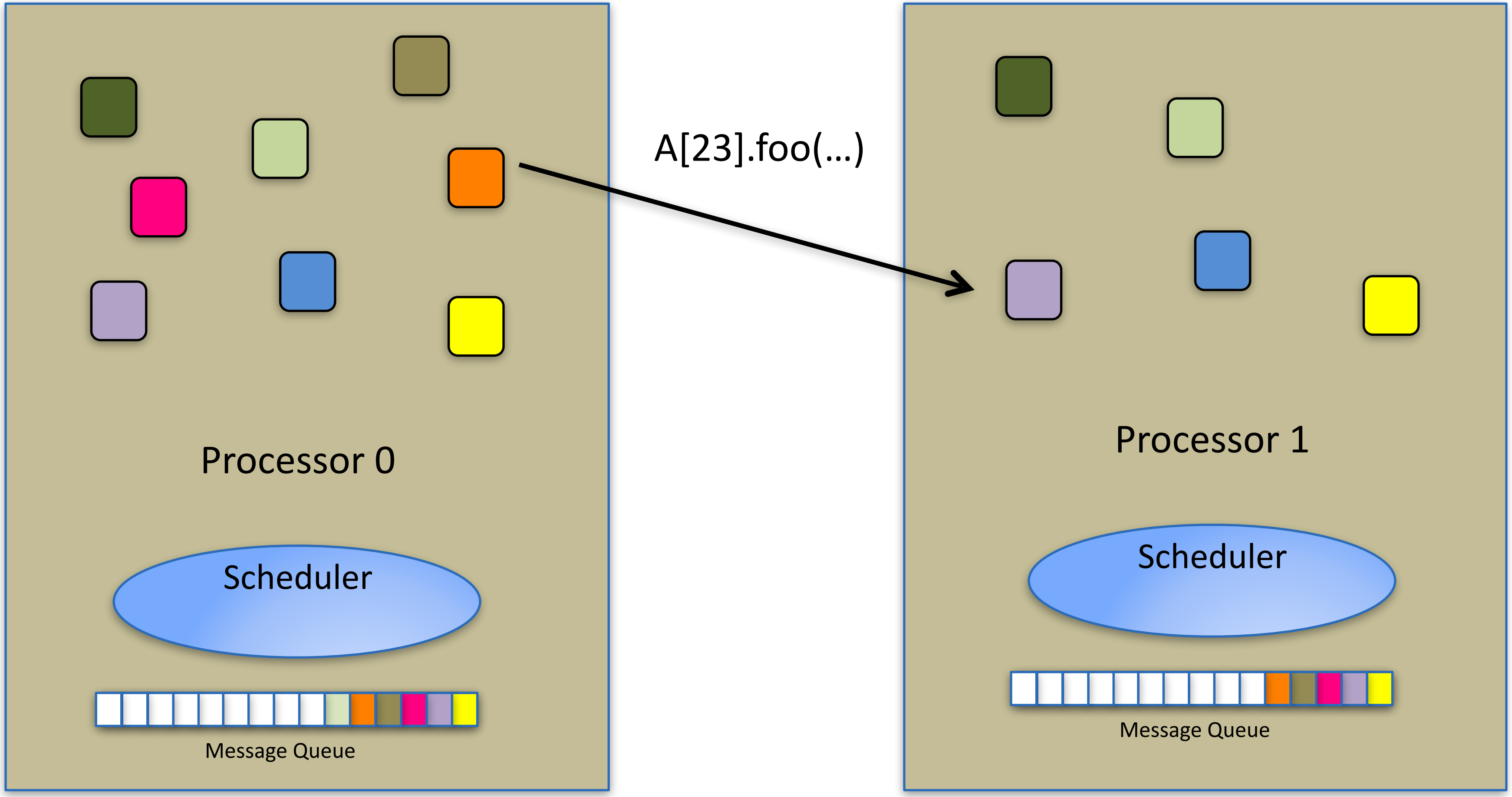

Global Object Space
Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue


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

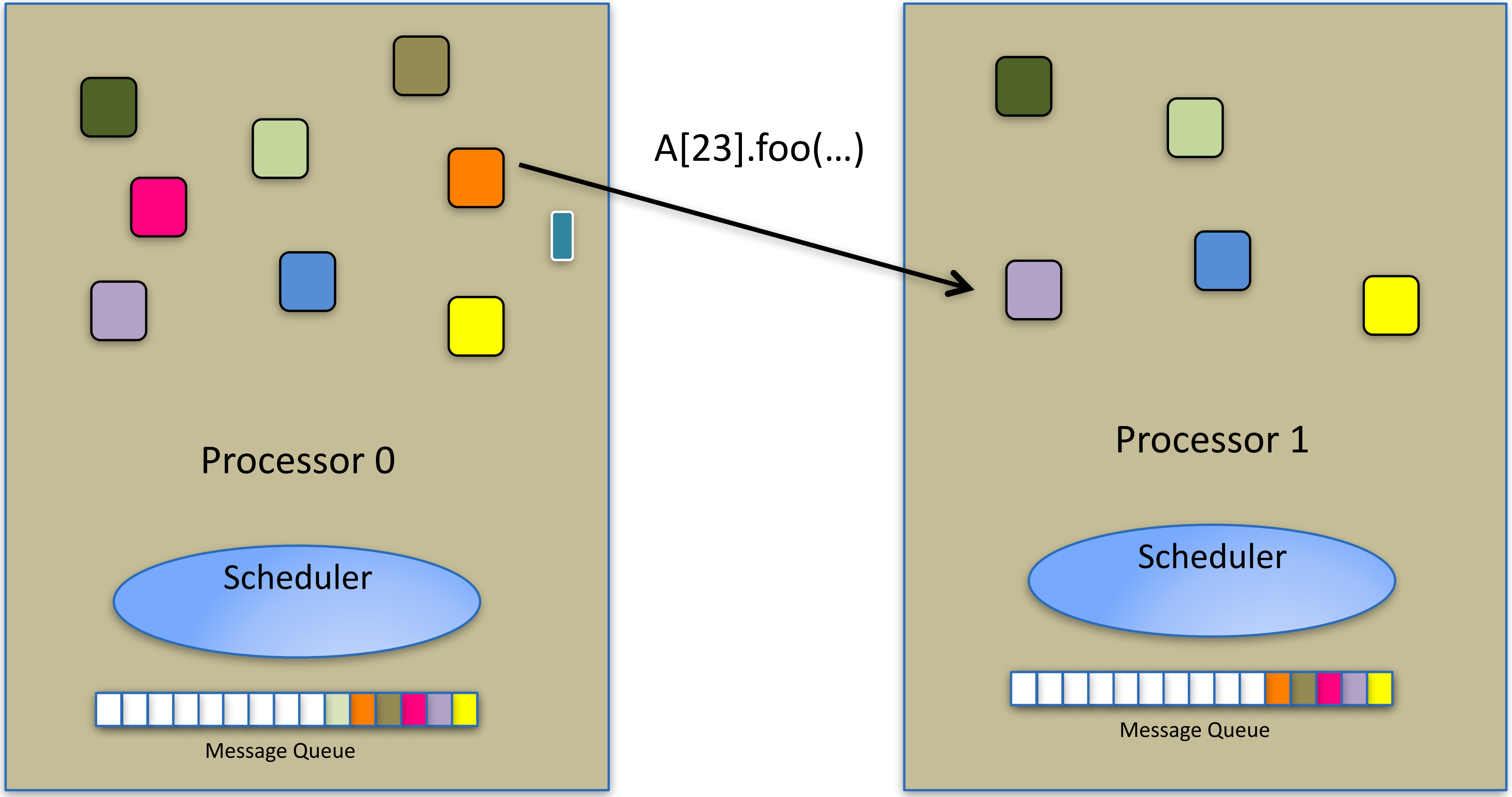
Message-driven Execution



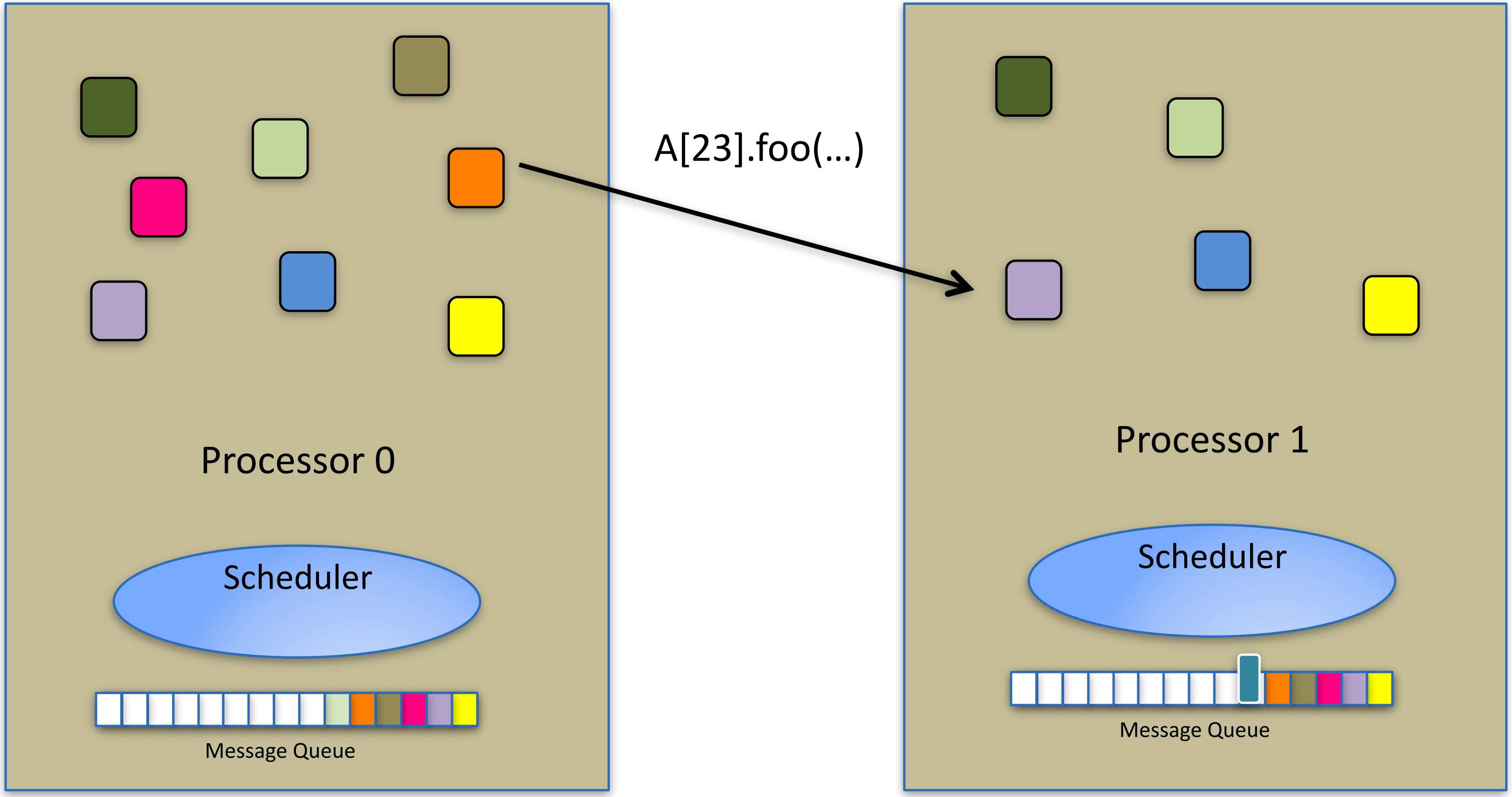
Message-driven Execution



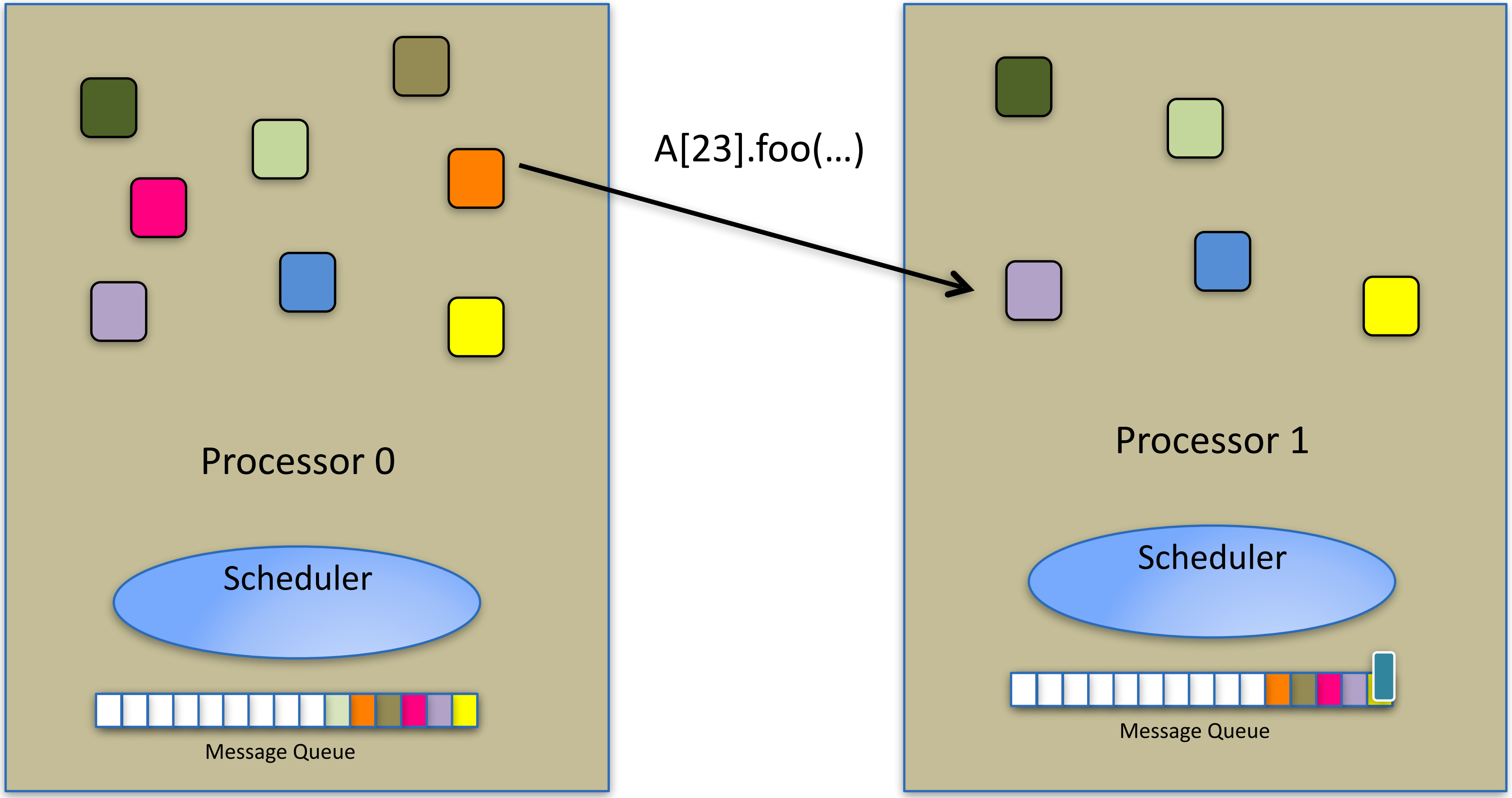
Message-driven Execution



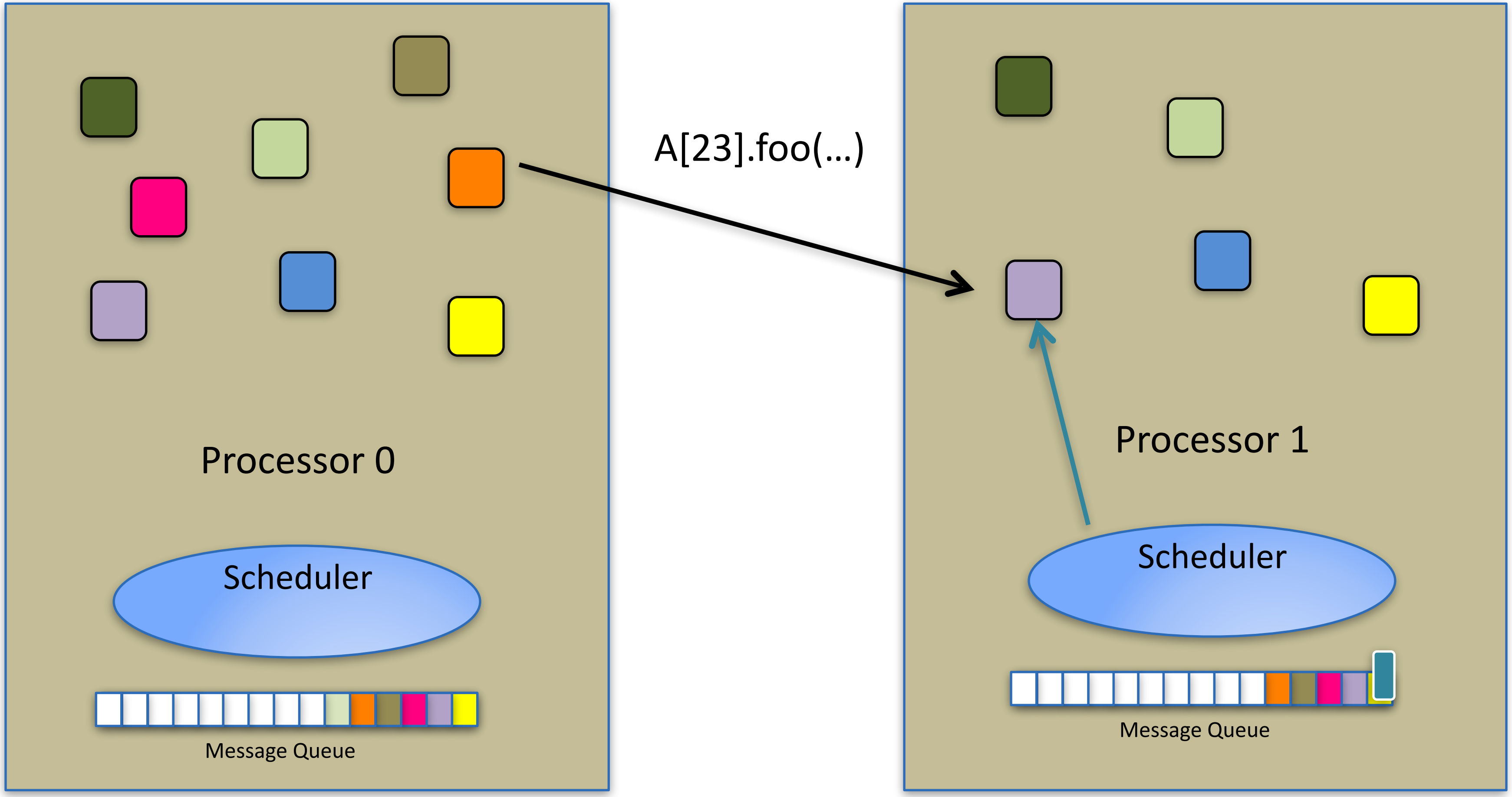
Message-driven Execution



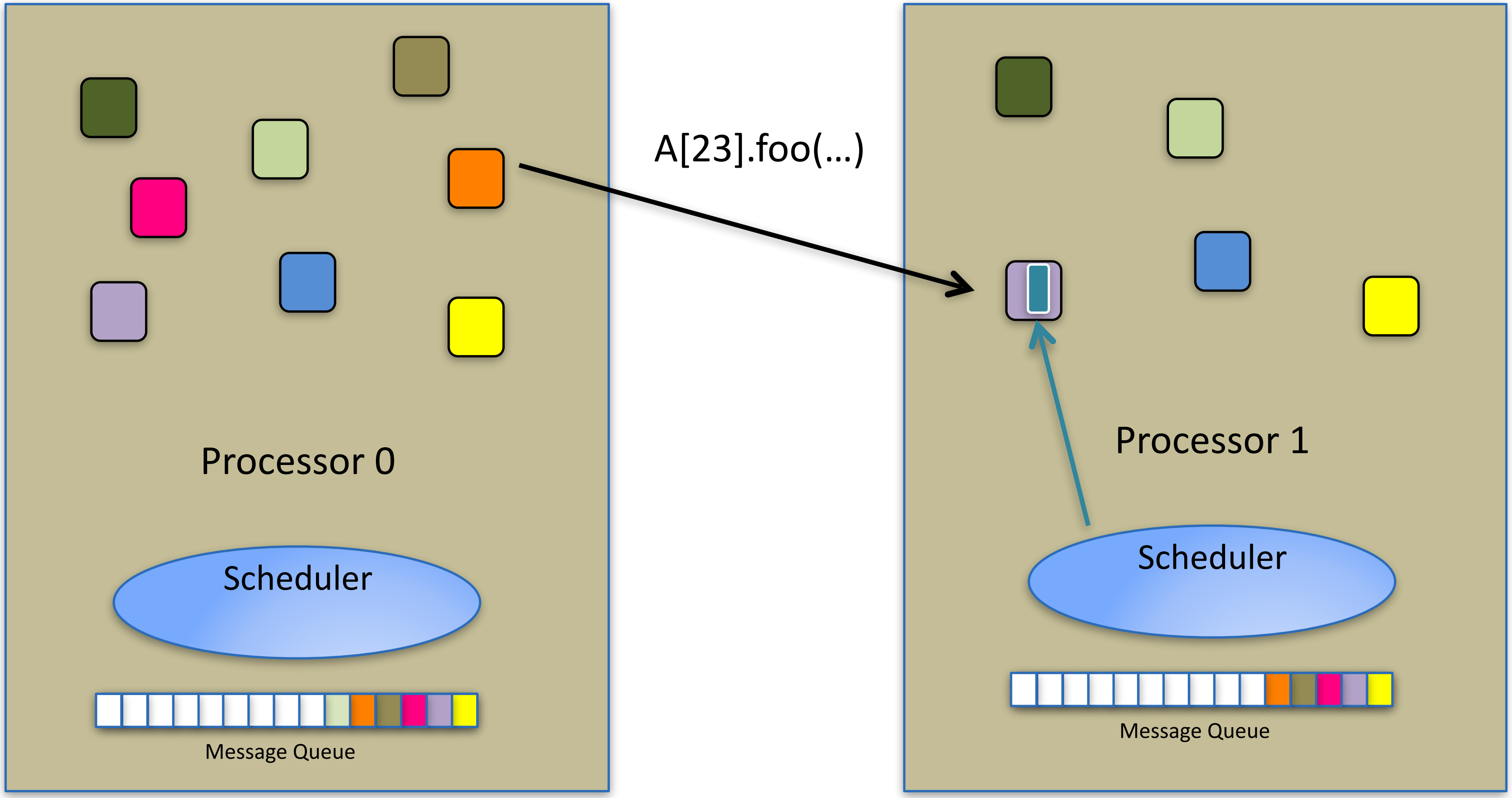
Message-driven Execution

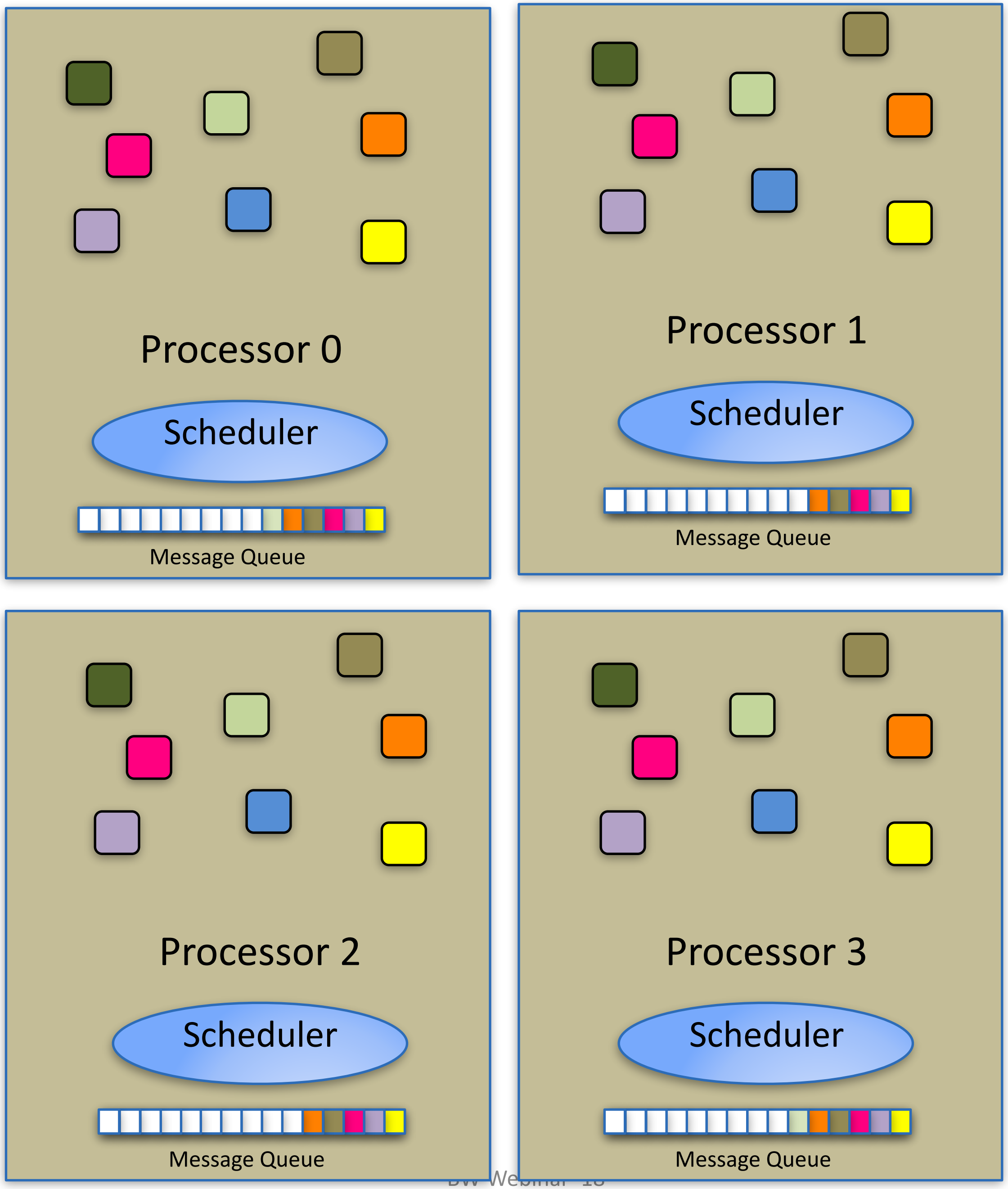


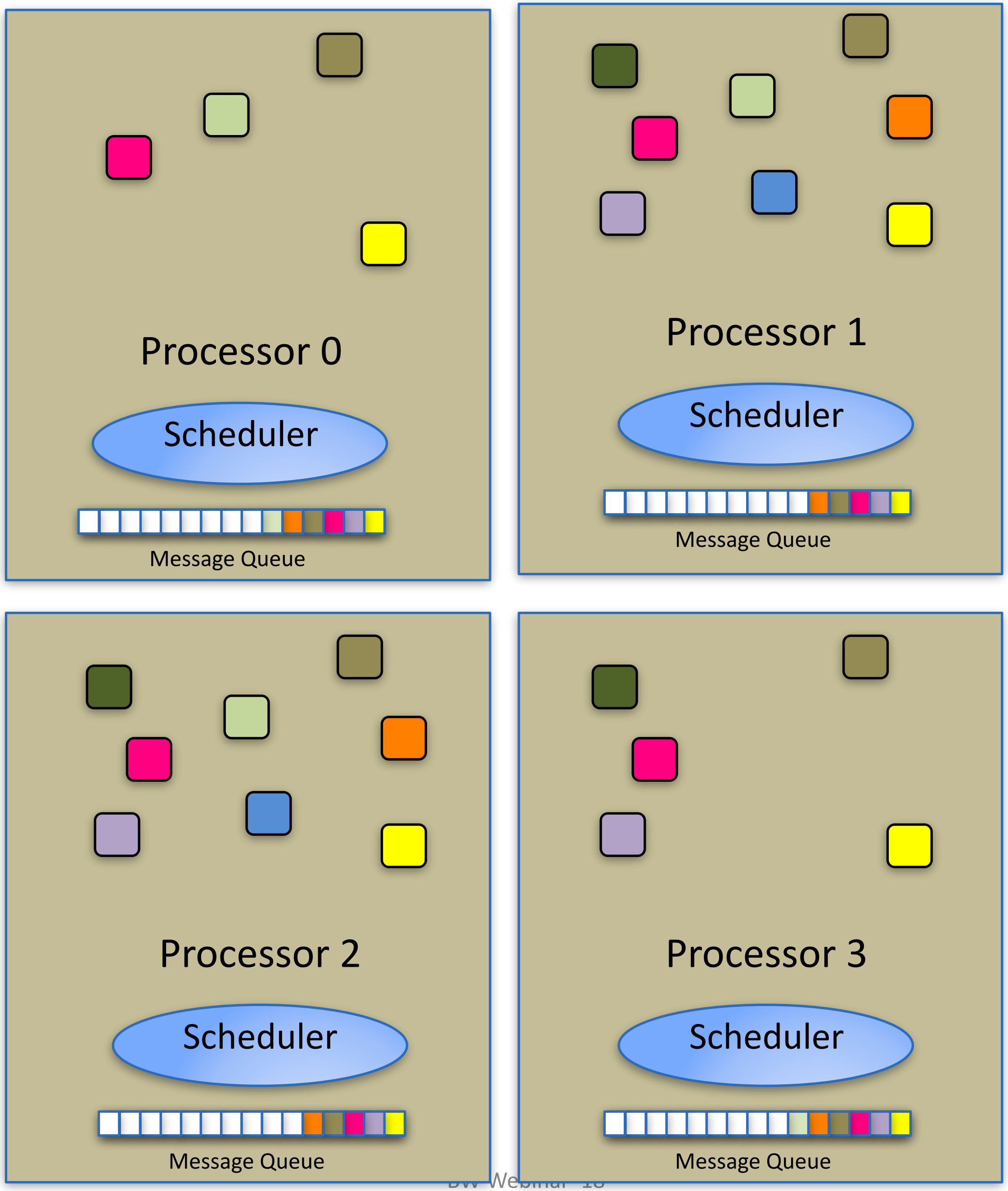
Message-driven Execution

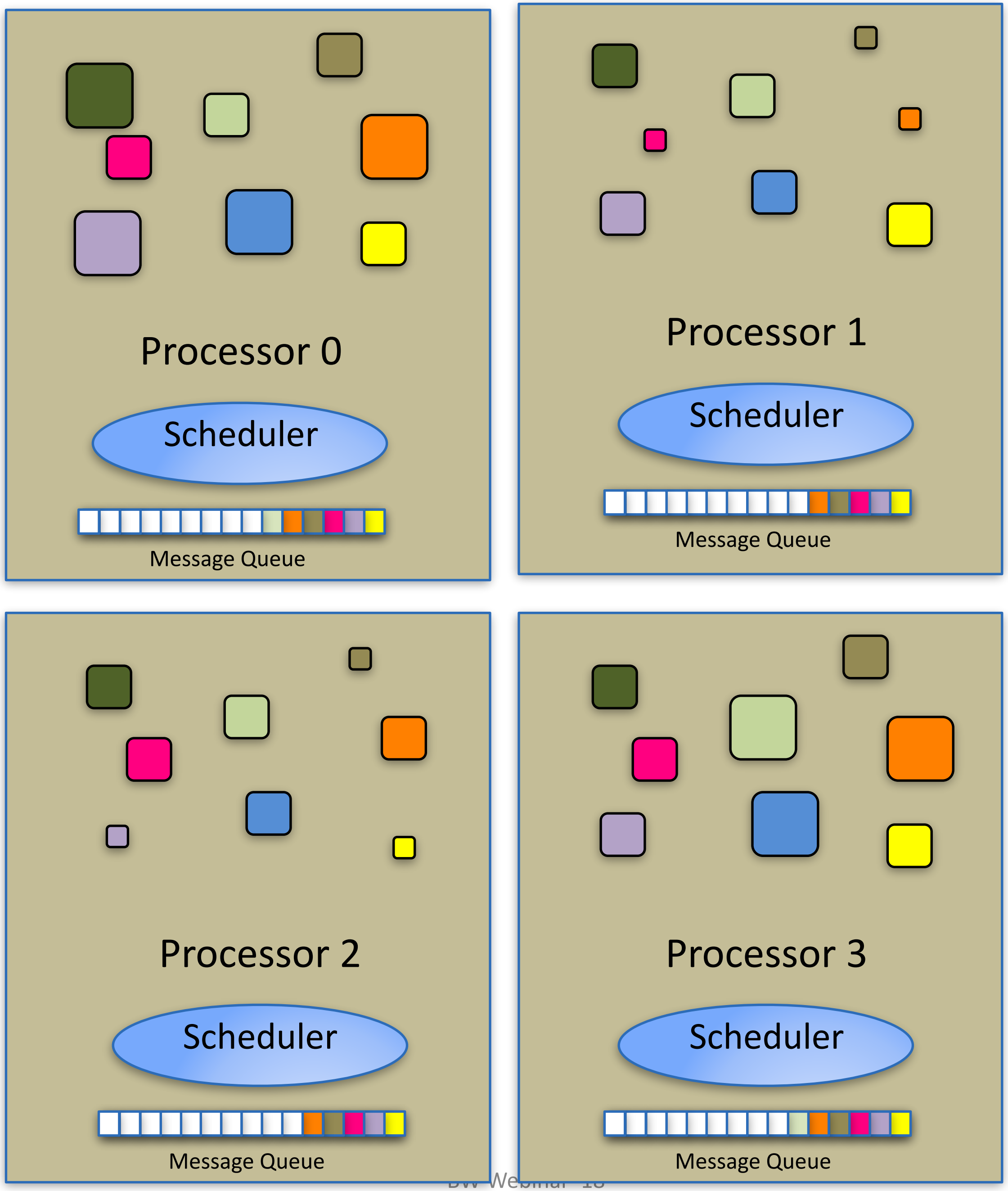


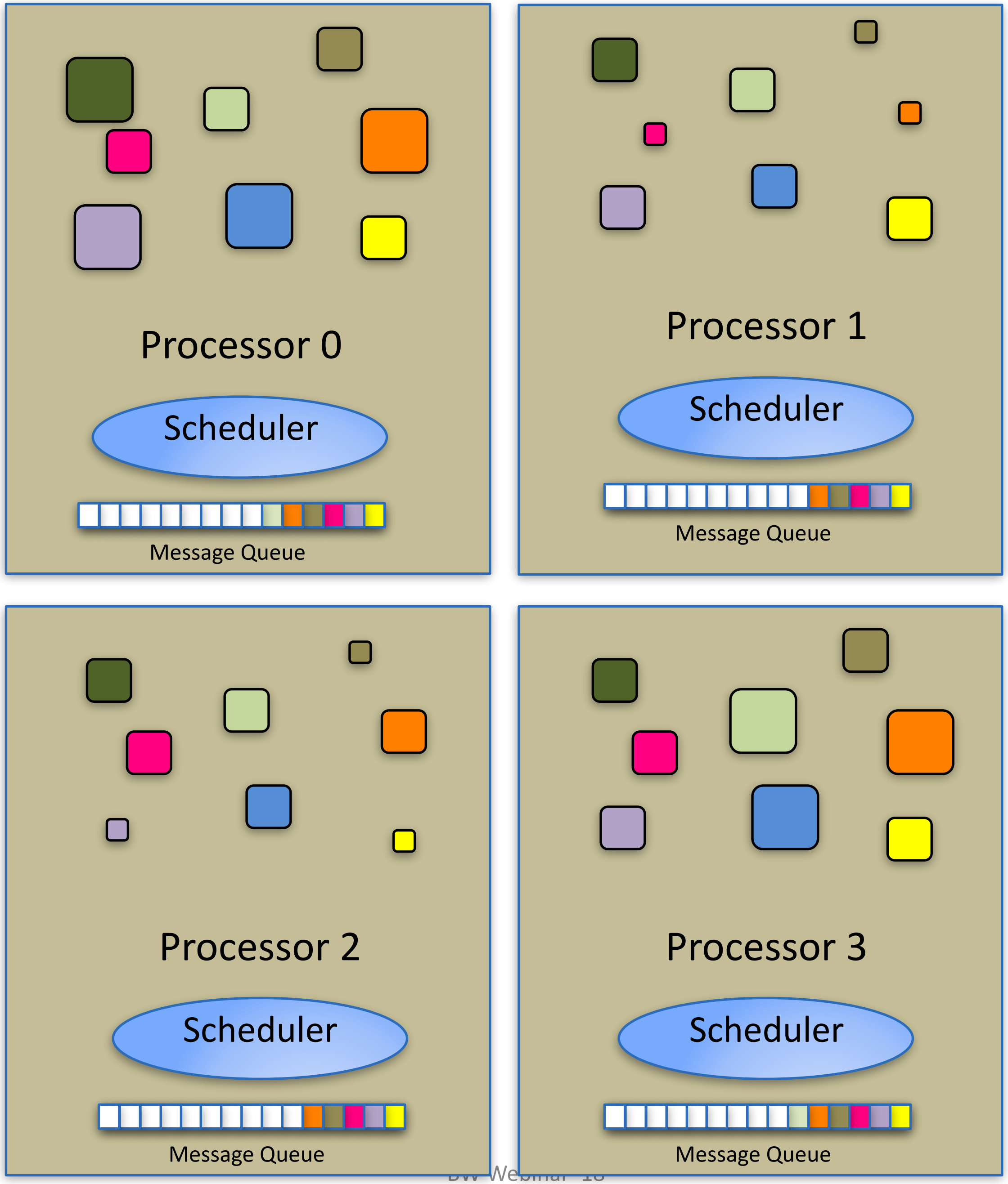
Message-driven Execution

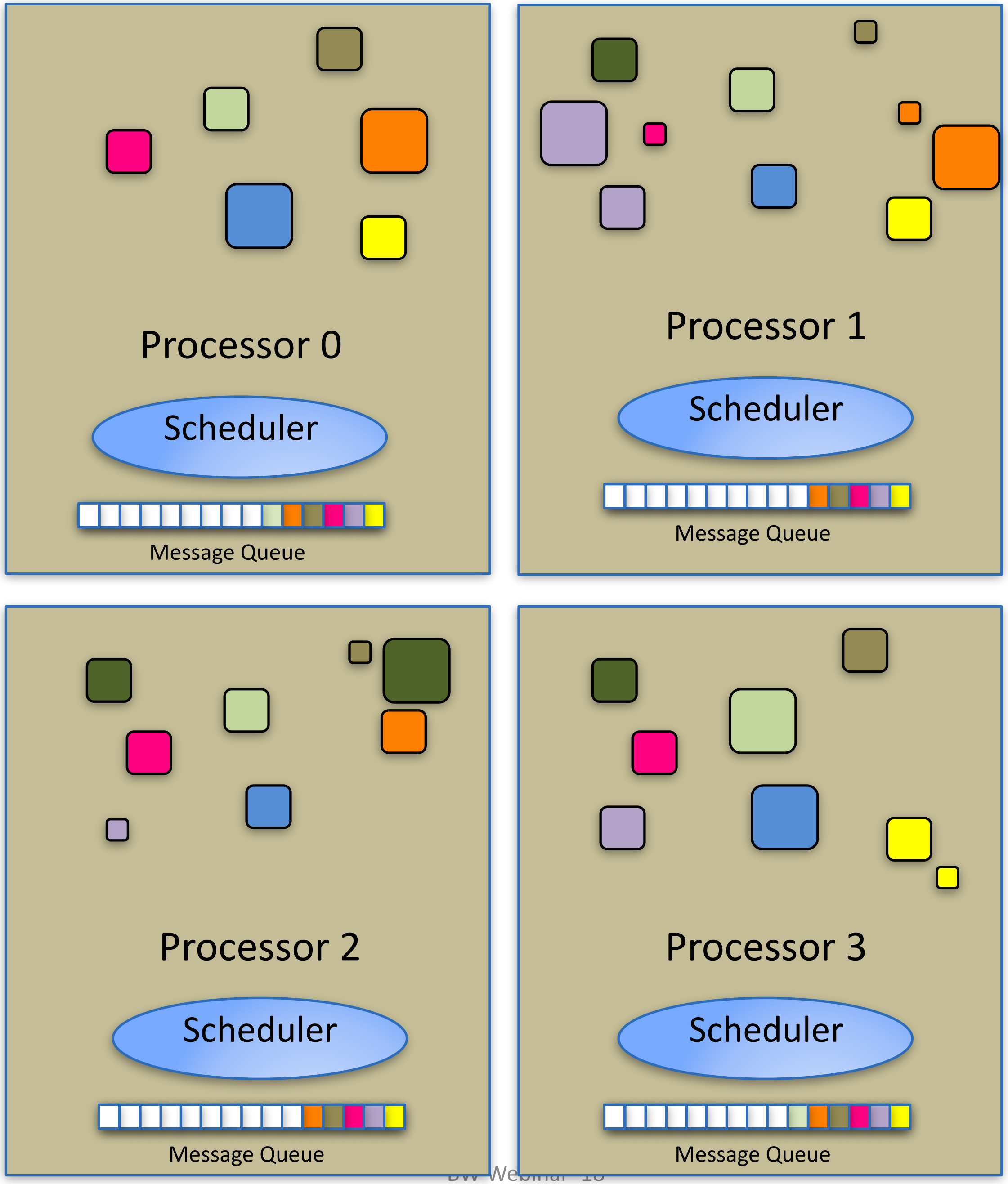



Processor 0
Scheduler
Message Queue
Processor 1
Scheduler
Message Queue
Processor 2
Scheduler
Message Queue
Processor 3
Scheduler
Message Queue


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


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

Asynchrony

Overdecomposition

Migratability

Empowering the RTS

Adaptive
Runtime System

Introspection

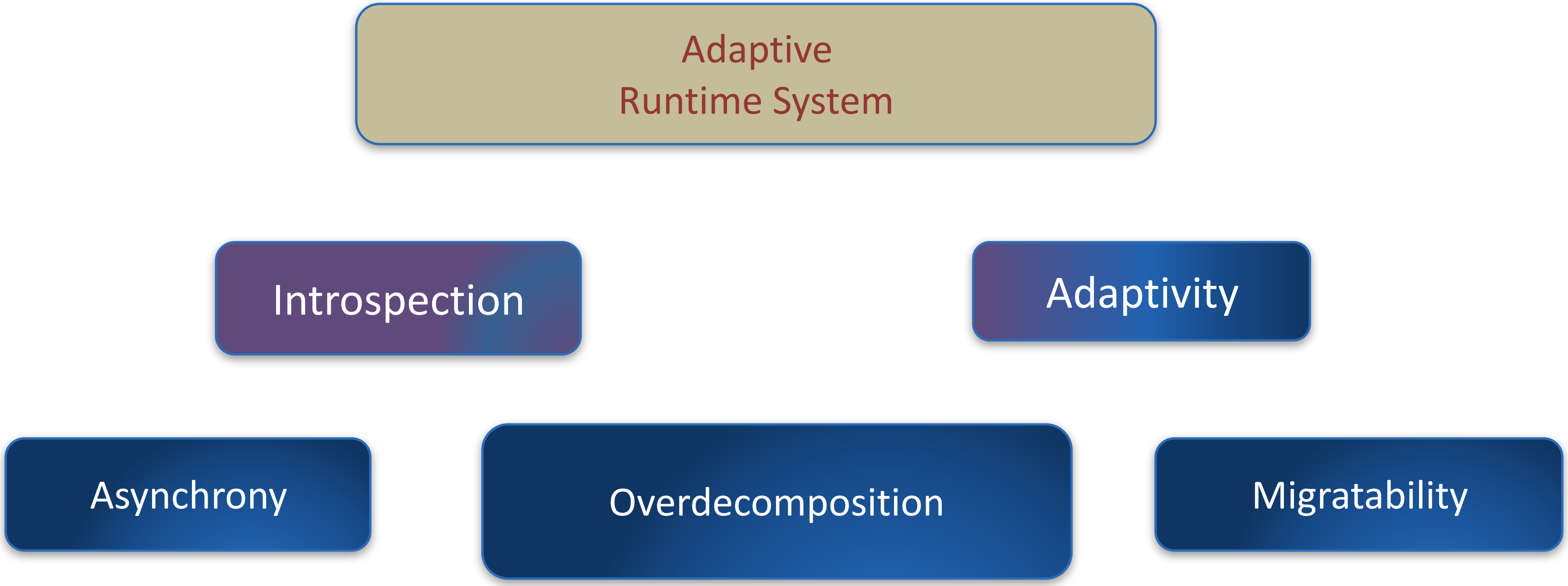
Adaptivity

Asynchrony

Overdecomposition

Migratability

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

Charm++ and CSE Applications

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

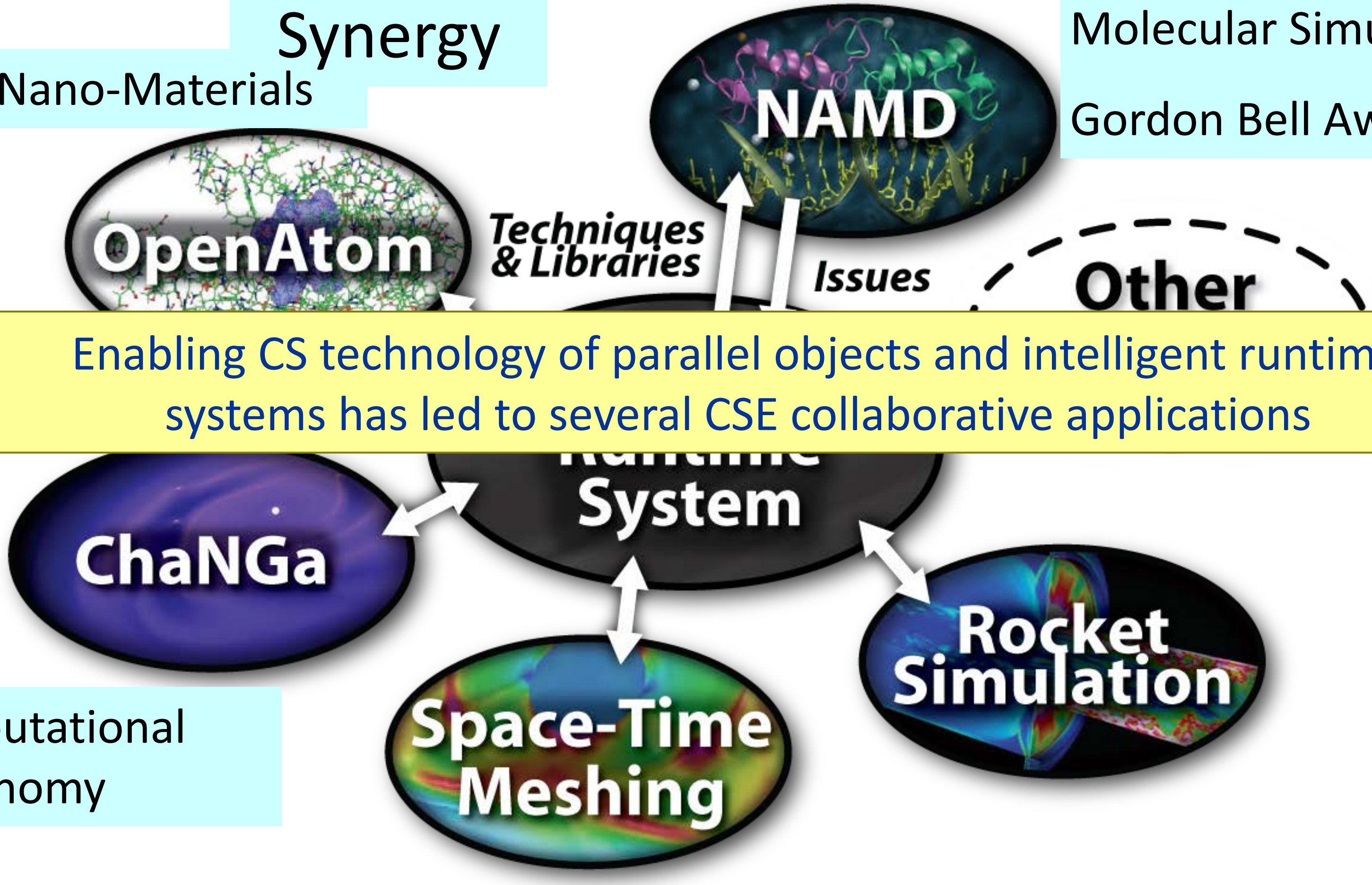
Charm++ and CSE Applications

Synergy

Nano-Materials

Well-known Biophysics Molecular Simulation App

Gordon Bell Award, 2002



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

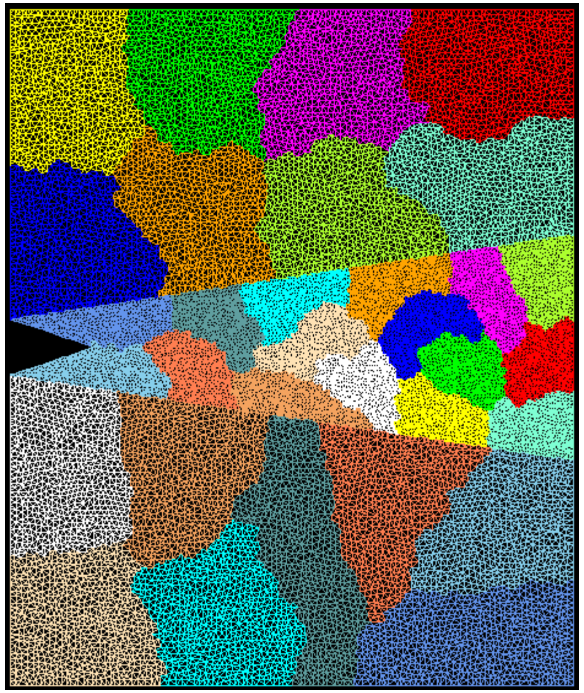
Grainsize

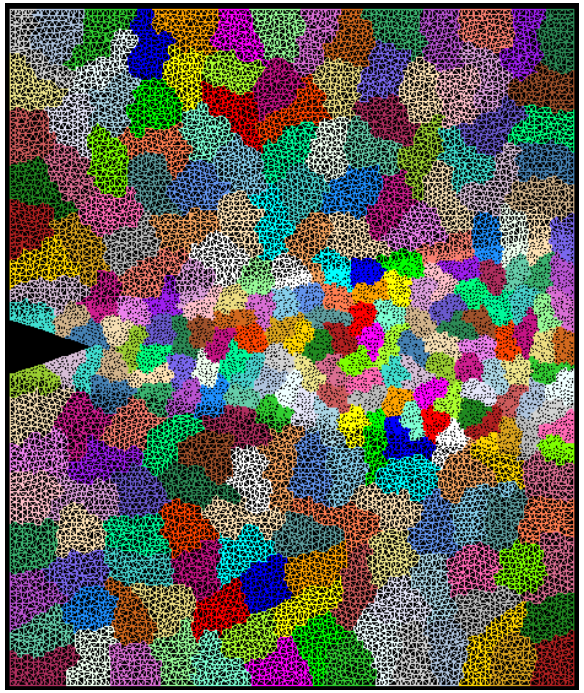
- Charm++ philosophy:
  - Let the programmer decompose their work and data into coarse-grained entities
- It is important to understand what I mean by coarse-grained entities
  - You don't write sequential programs that some system will auto-decompose
  - You don't write programs when there is one object 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

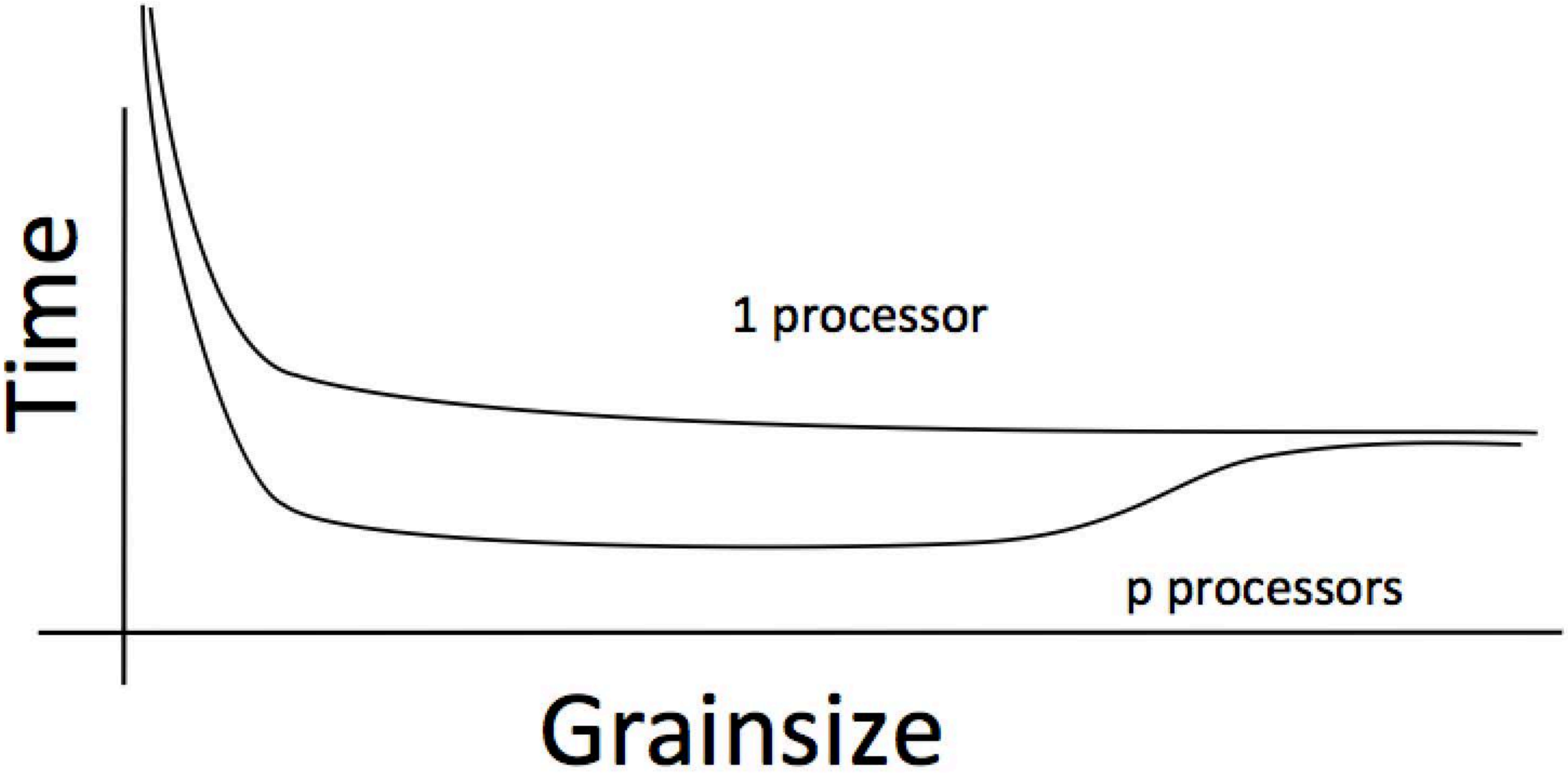




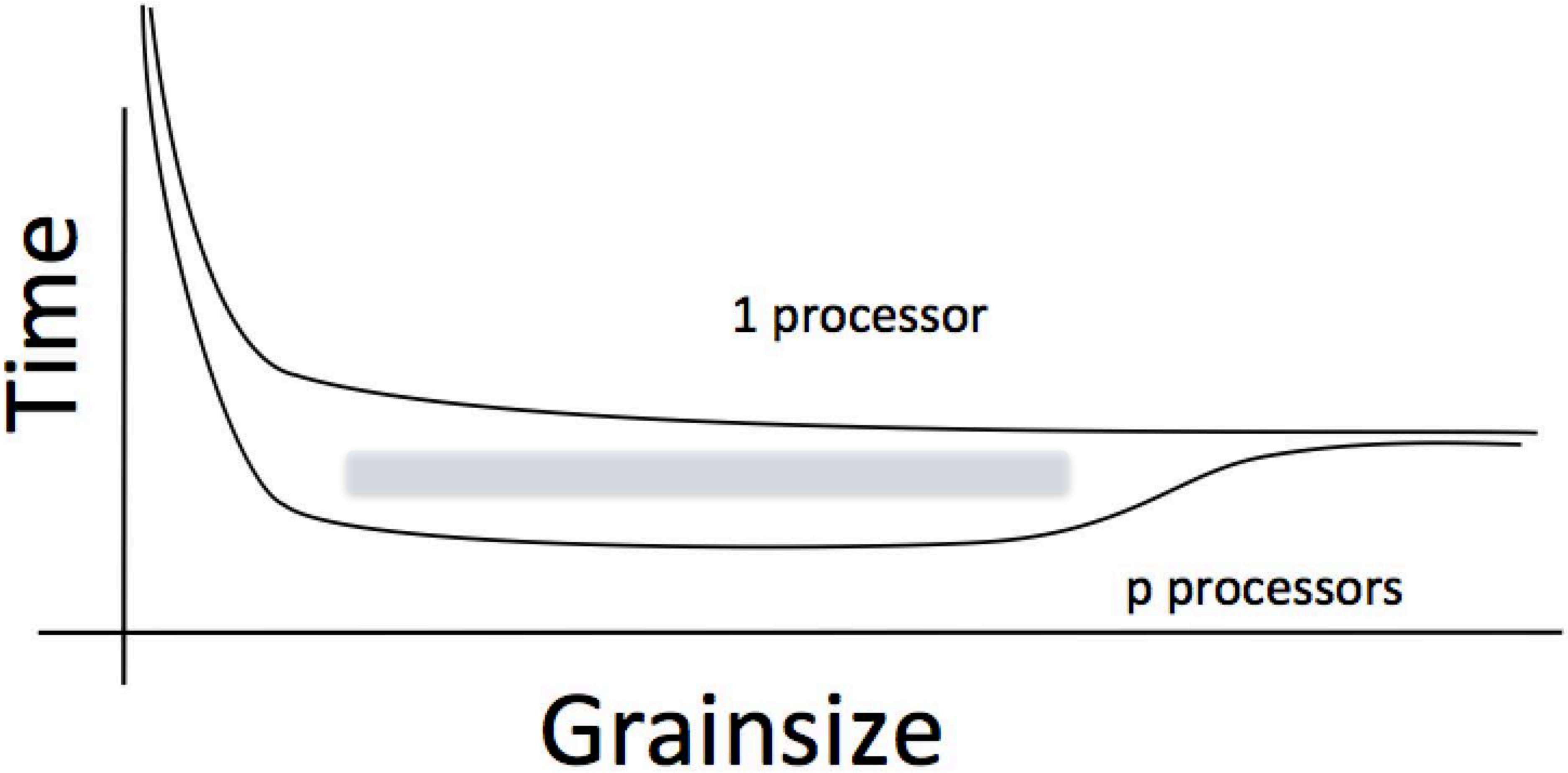
Decomposition into 16 chunks (left) and 128 chunks, 8 for each PE (right). The middle area contains cohesive elements. Both decompositions obtained using Metis. Pictures: S. Breitenfeld, and P. Geubelle

Working definition of grainsize:

amount of computation per remote interaction

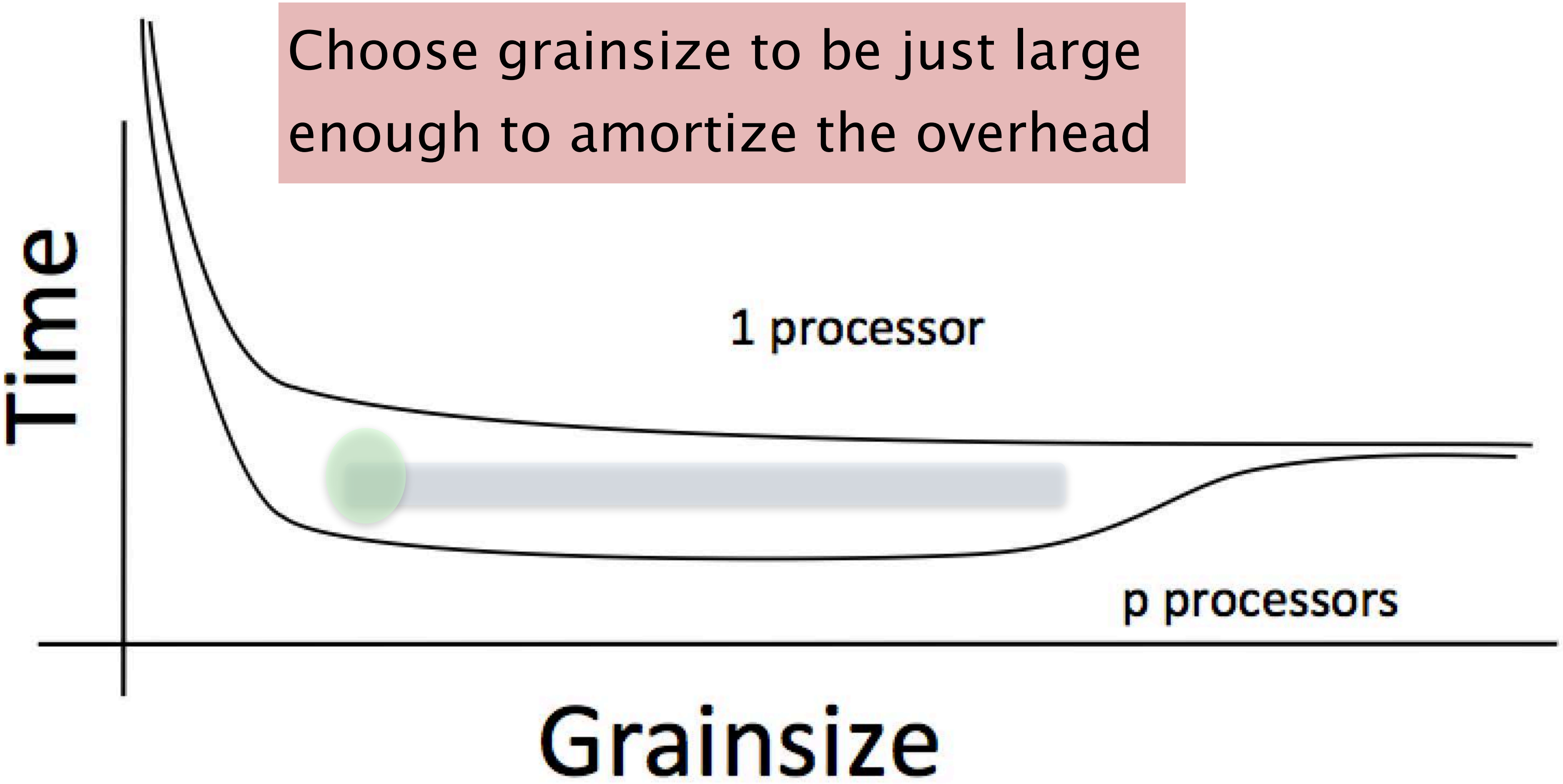


Working definition of grainsize:
amount of computation per remote interaction

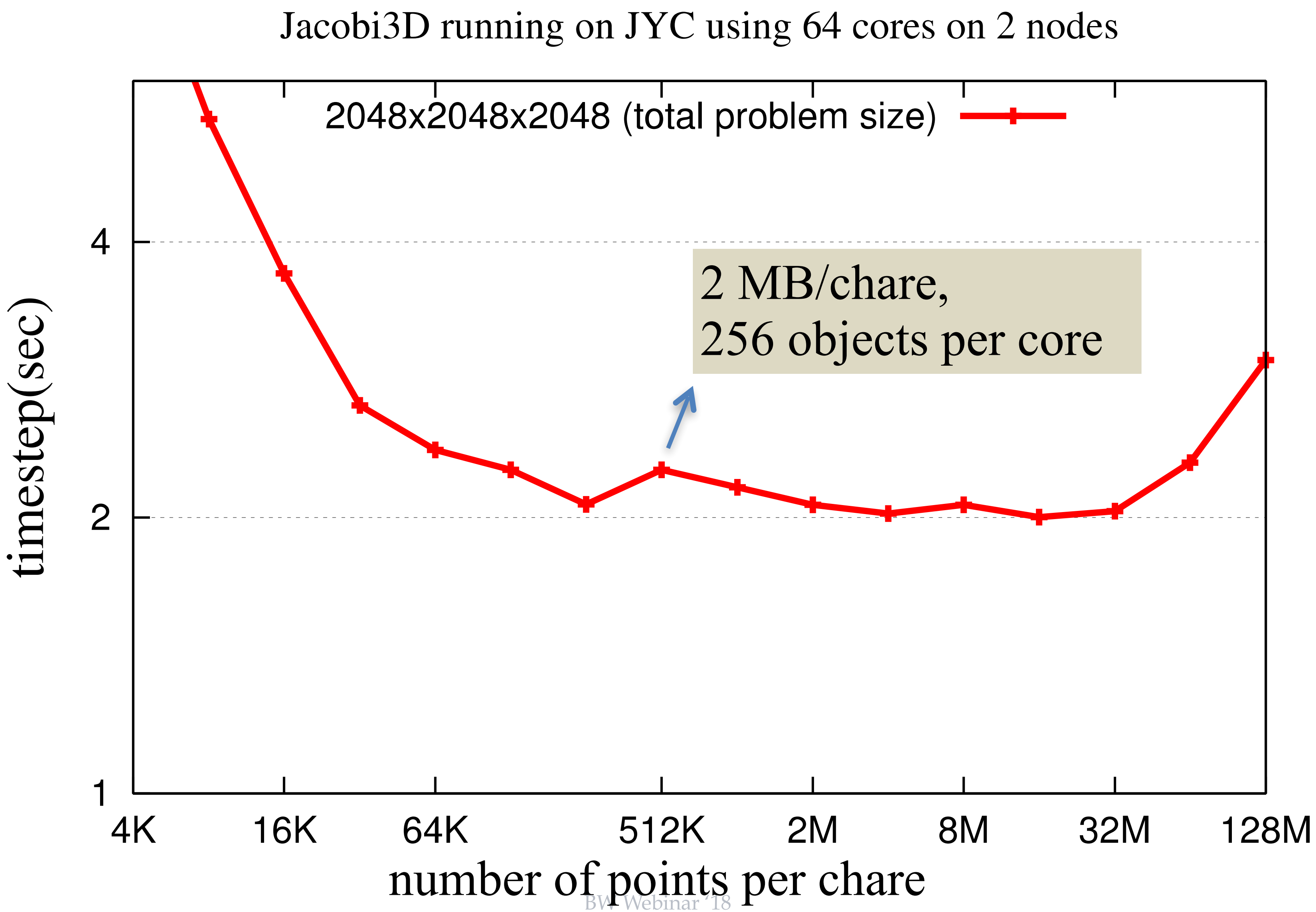


Working definition of grainsize:
amount of computation per remote interaction

Choose grainsize to be just large enough to amortize the overhead

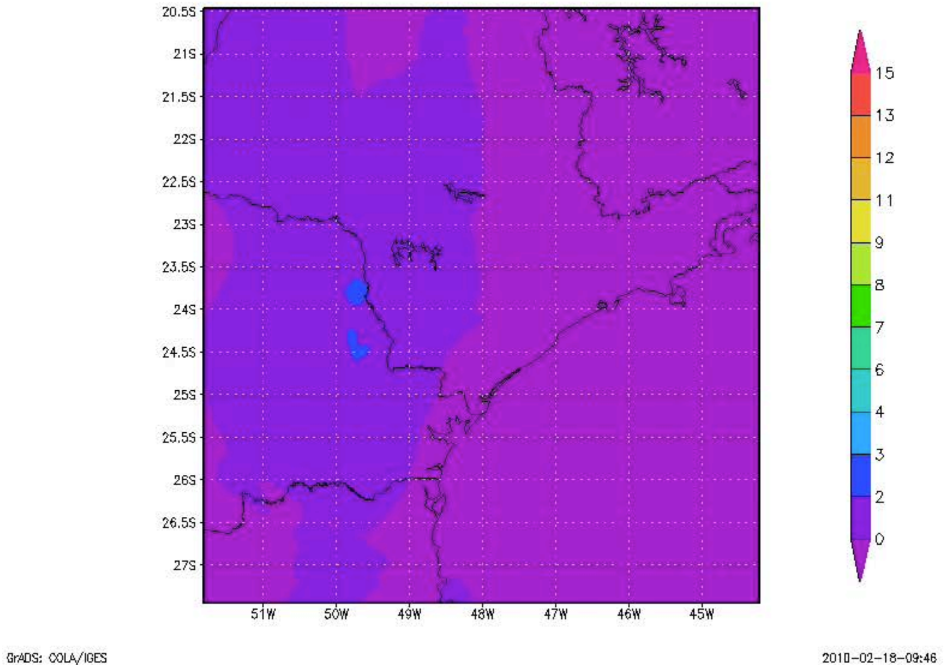


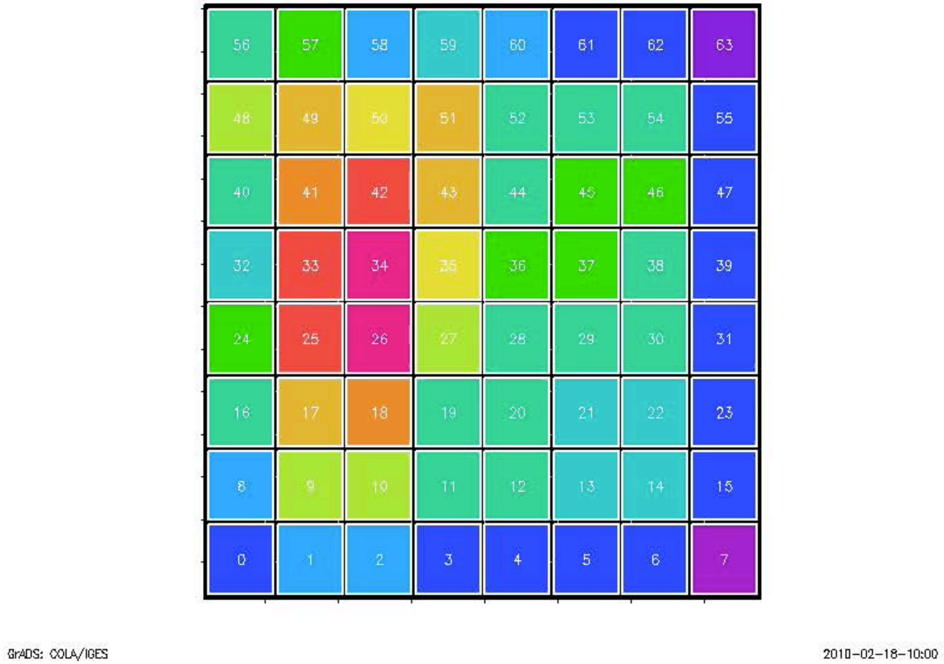
Grainsize in a common setting



Grainsize: Weather Forecasting in BRAMS

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

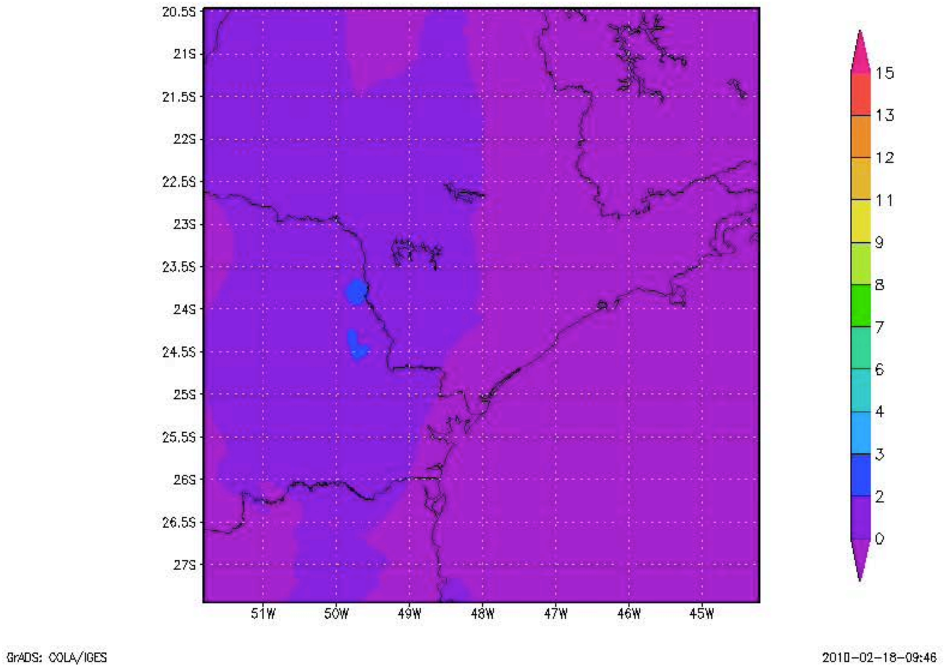


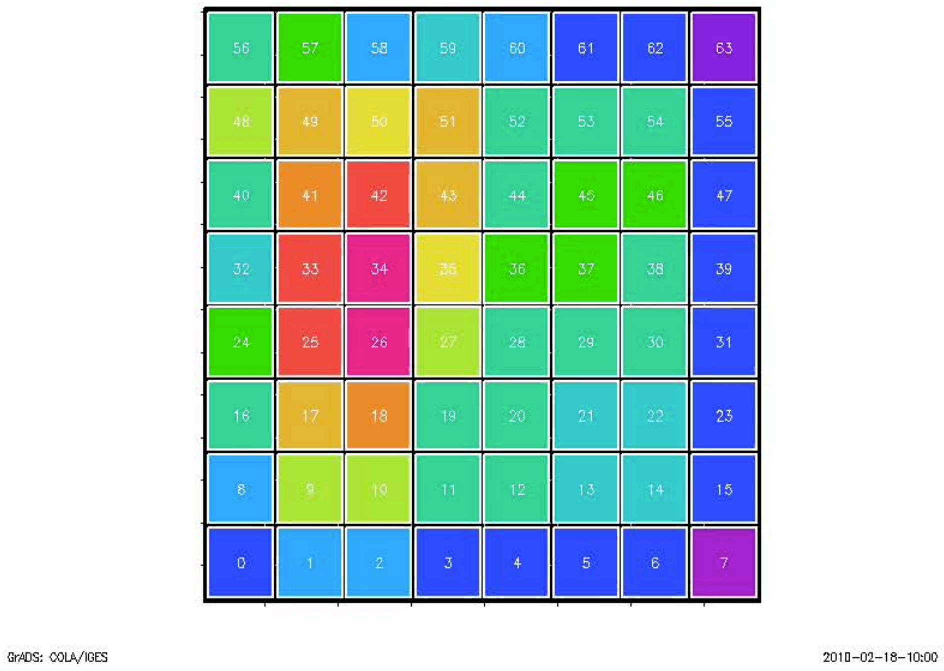


Instead of using 64 work units on 64 cores, used 1024 on 64

Grainsize: Weather Forecasting in BRAMS

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

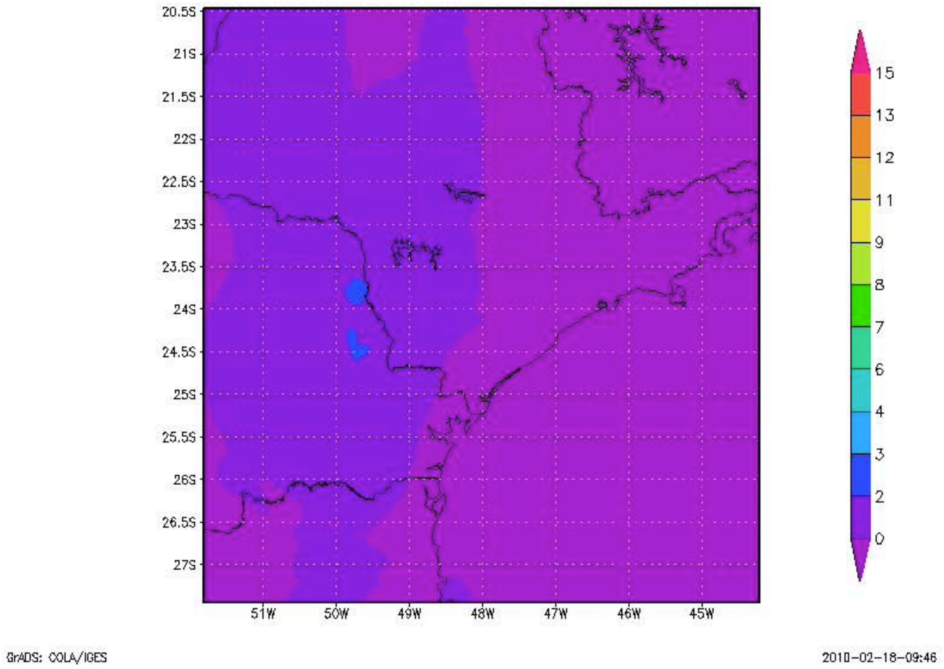


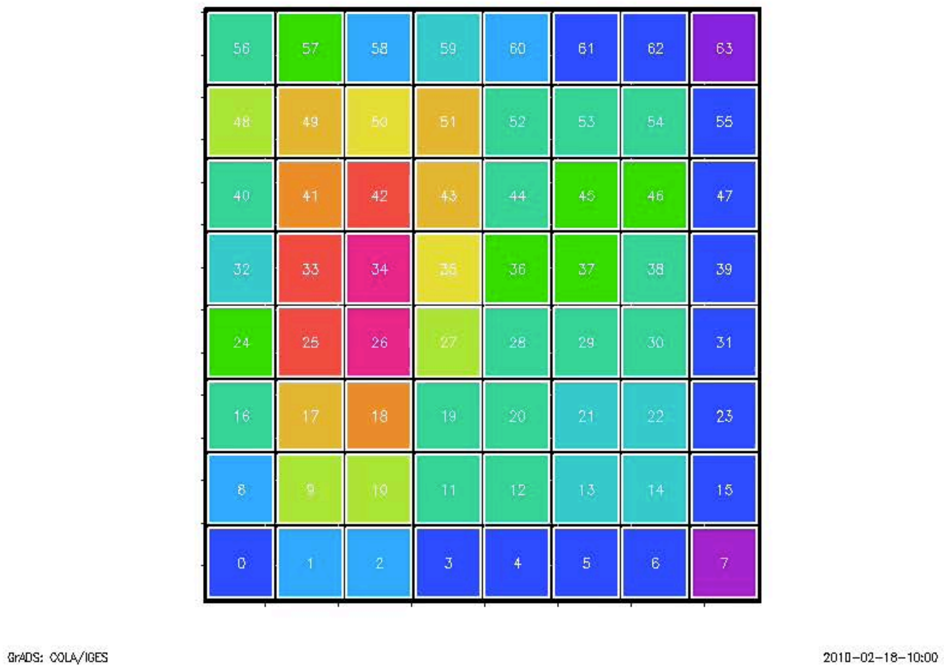


Instead of using 64 work units on 64 cores, used 1024 on 64

Grainsize: Weather Forecasting in BRAMS

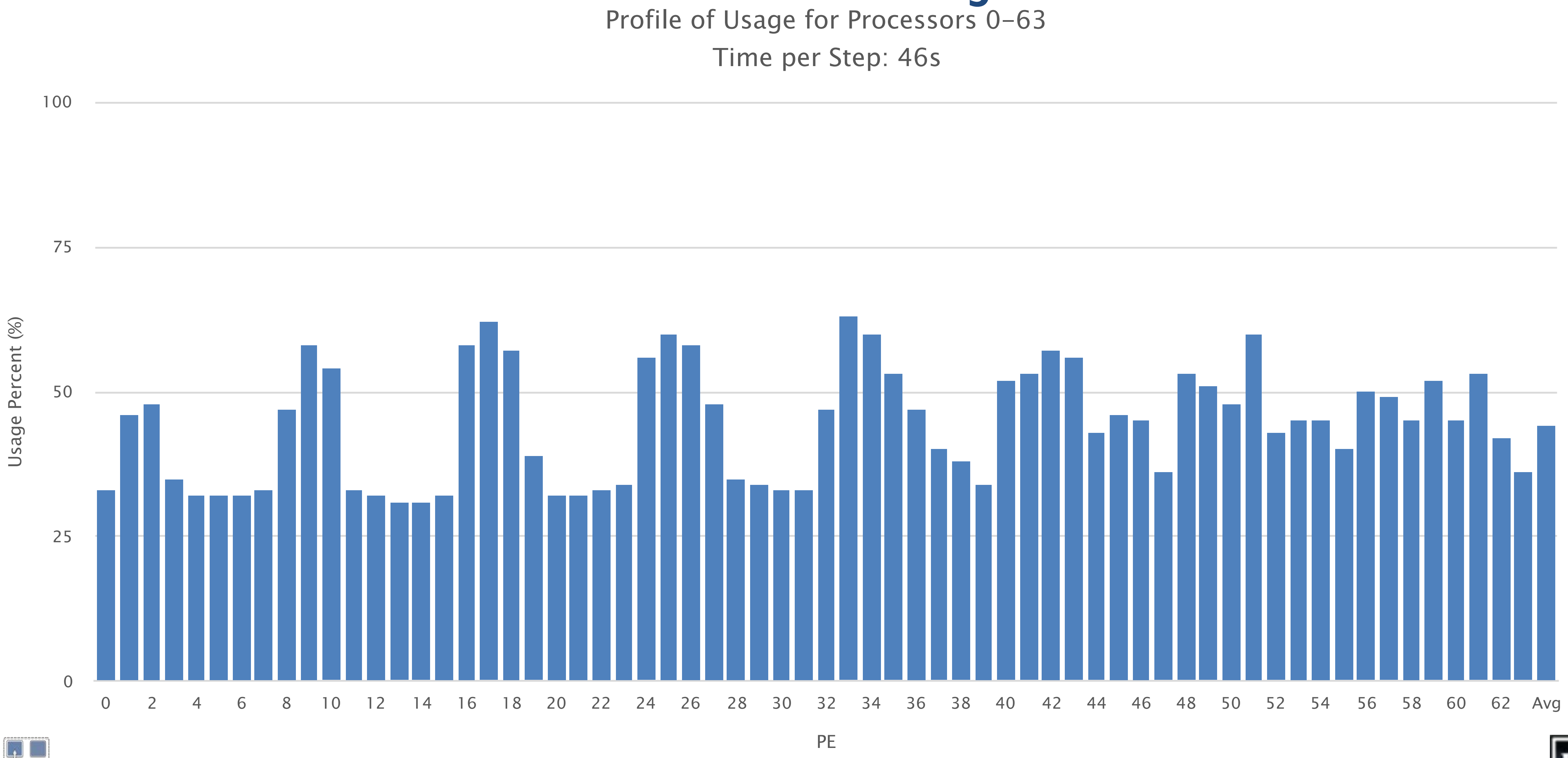
- BRAMS: Brazillian weather code (based on RAMS)
- AMPI version (Eduardo Rodrigues, with Mendes, J. Panetta, ..)



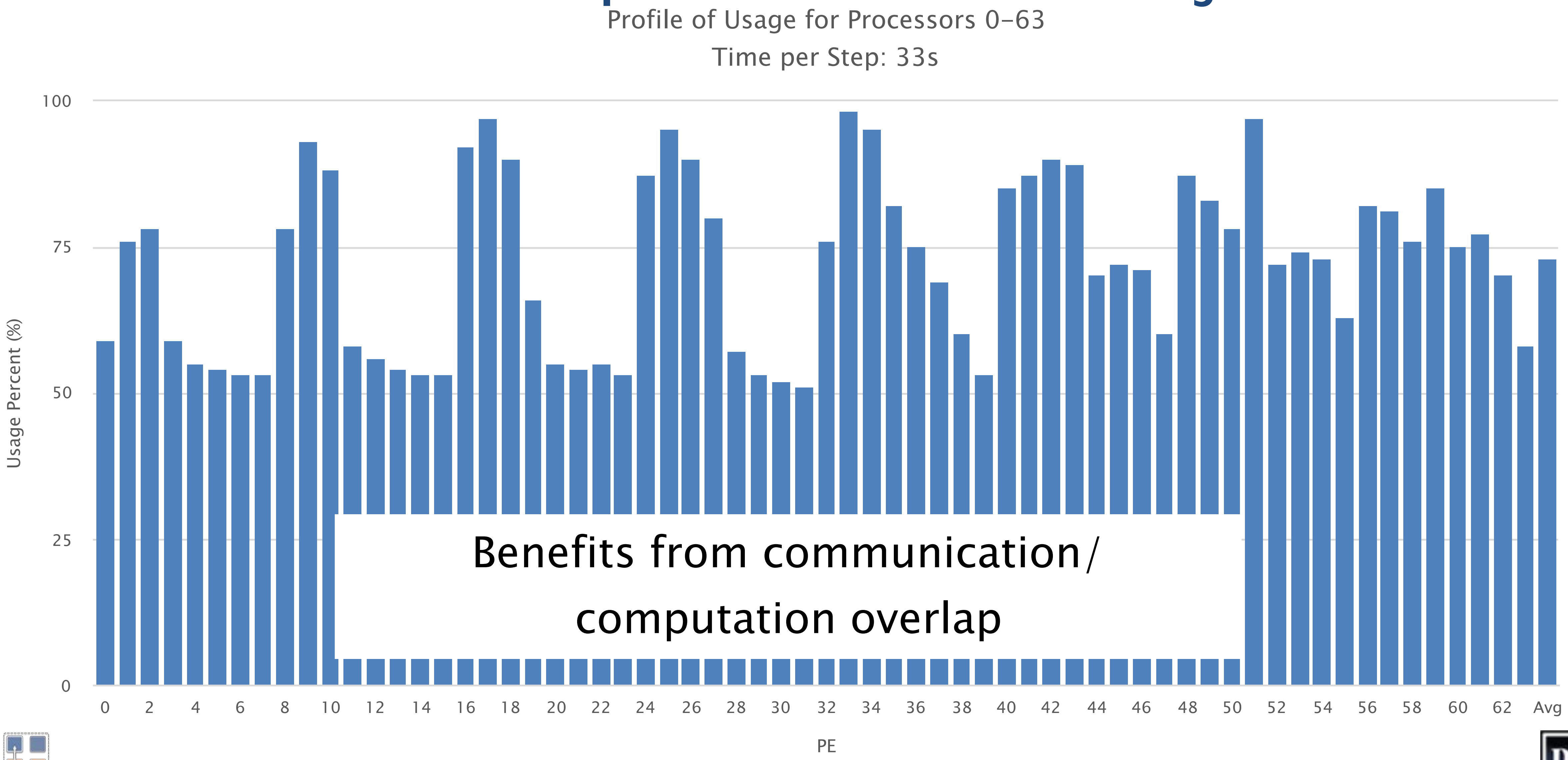


Instead of using 64 work units on 64 cores, used 1024 on 64

Baseline: 64 Objects

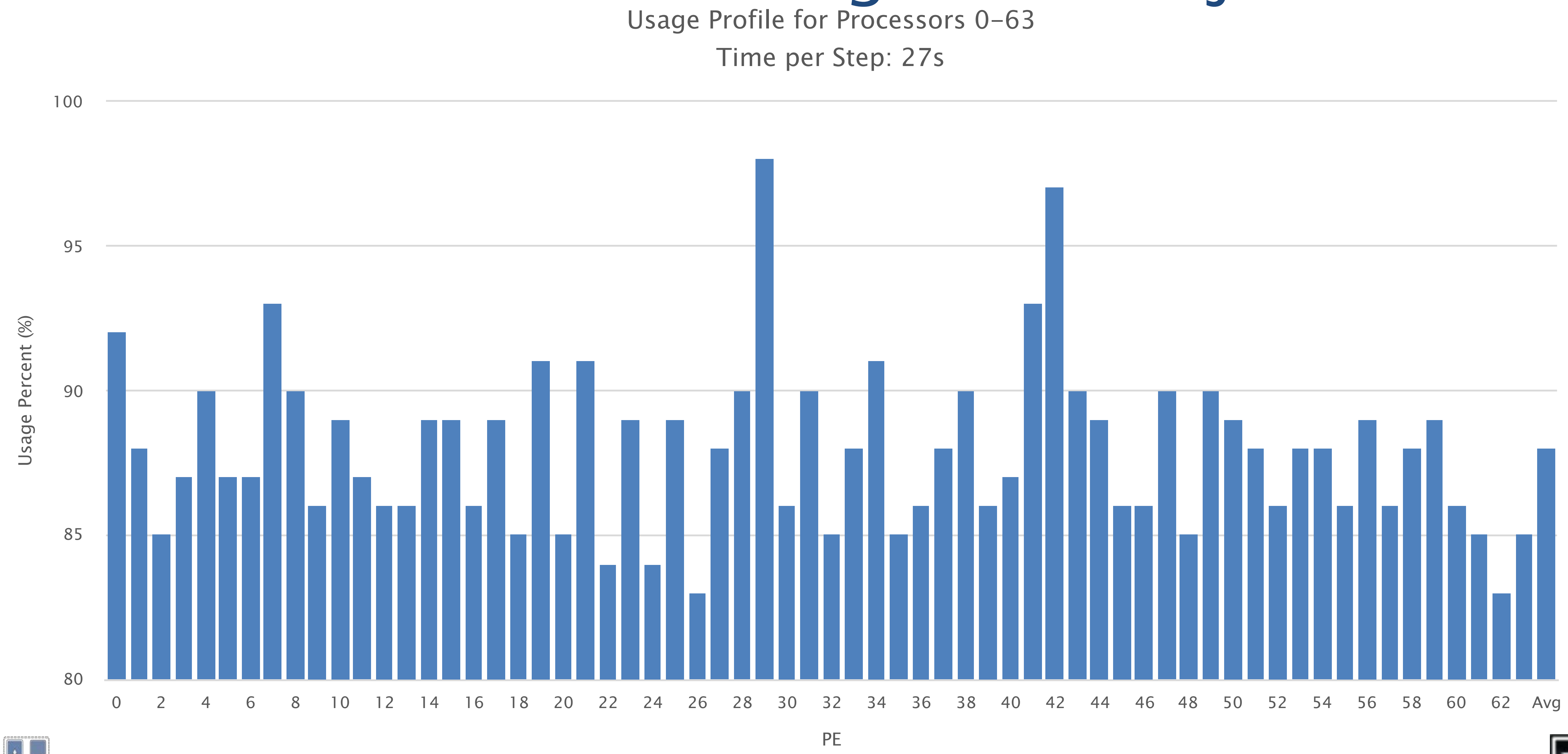


Overdecomposition: 1024 Objects

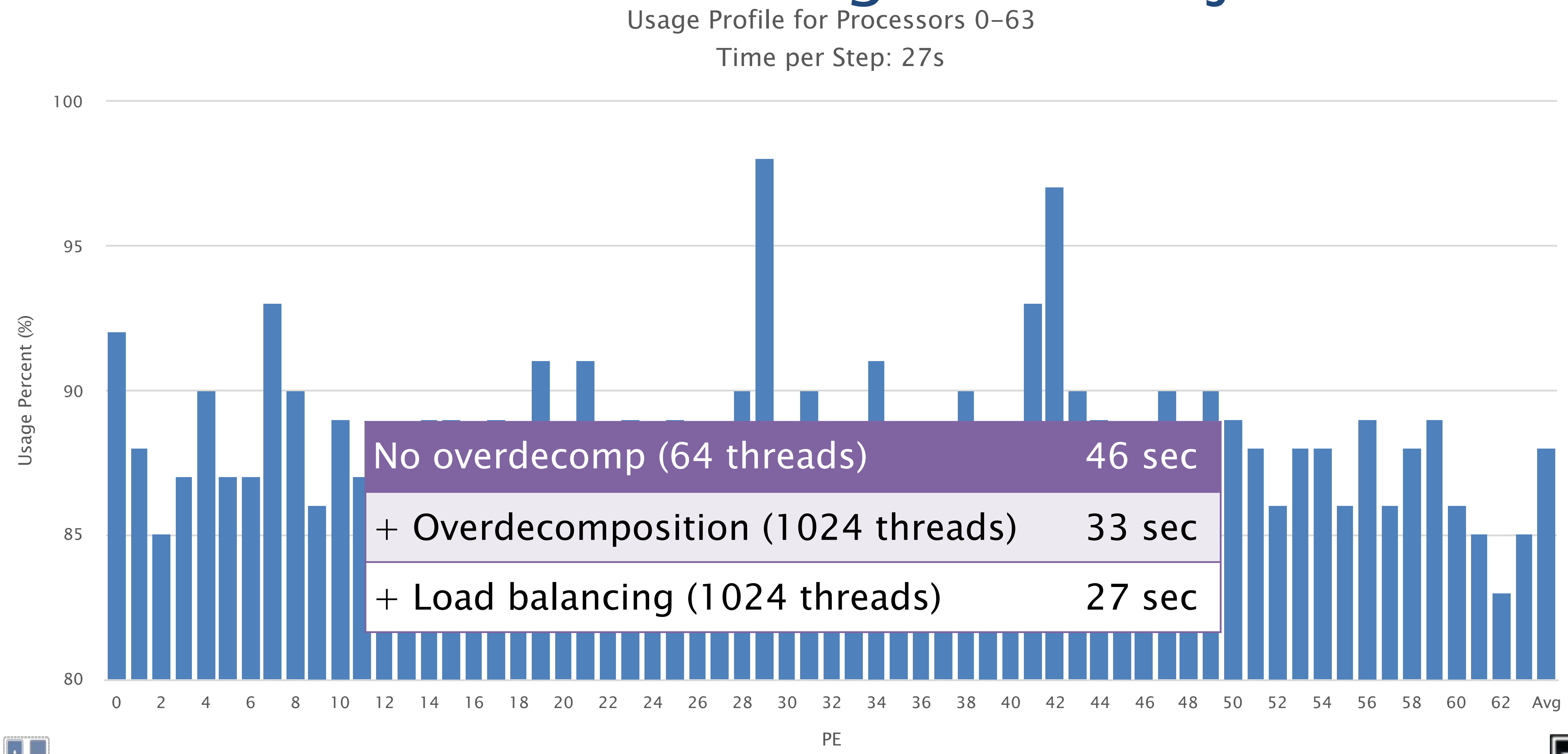


Benefits from communication/ computation overlap

With Load Balancing: 1024 objects

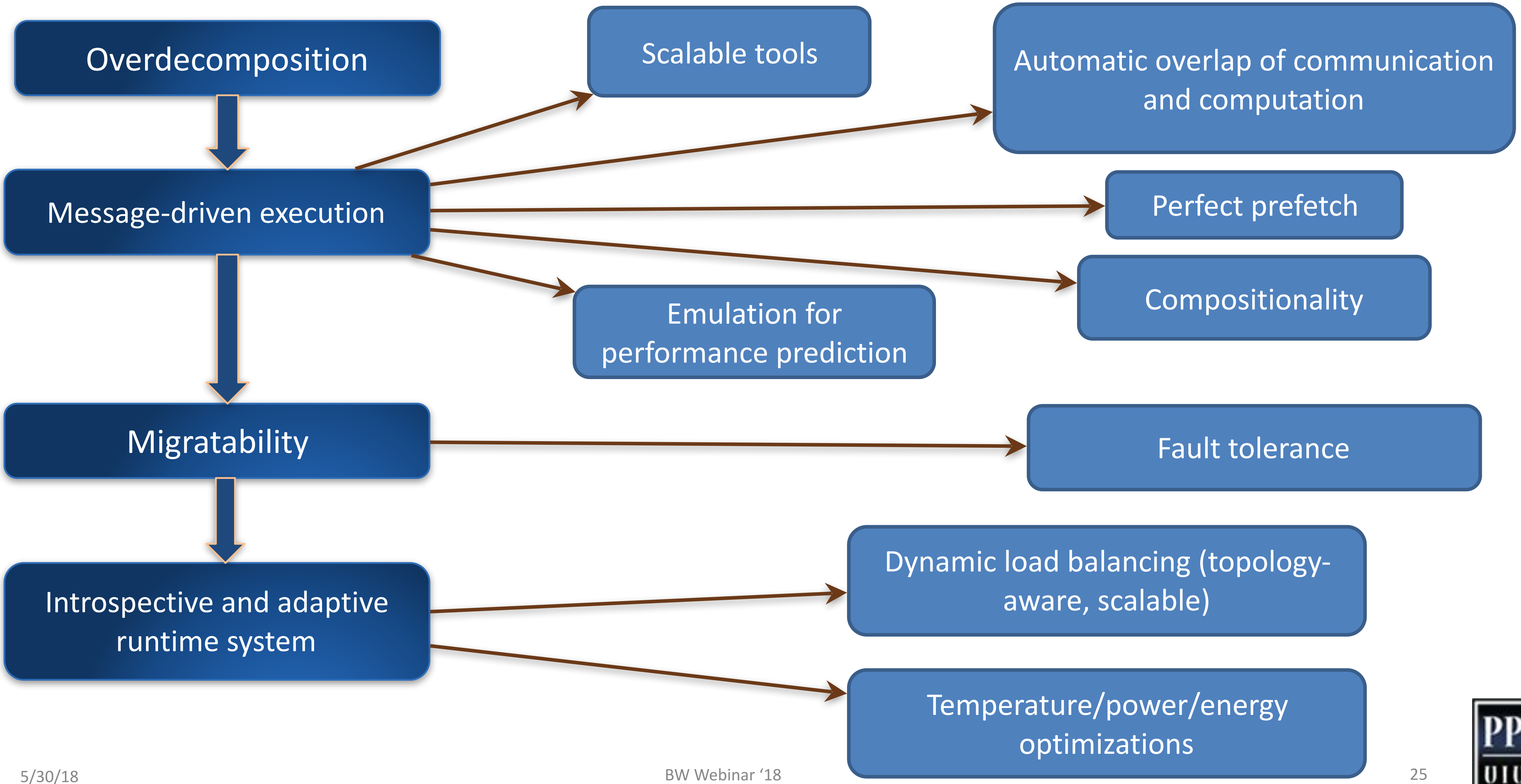


With Load Balancing: 1024 objects

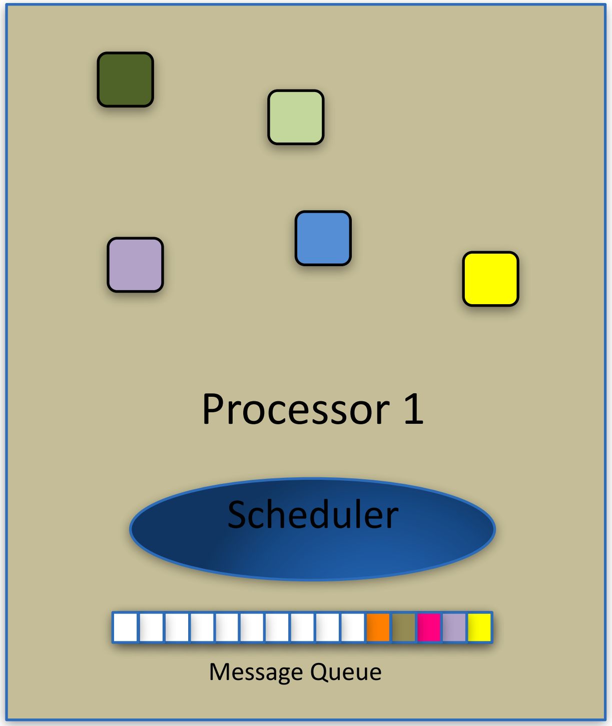


No overdecomp (64 threads)	46 sec
+ Overdecomposition (1024 threads)	33 sec
+ Load balancing (1024 threads)	27 sec

Charm++ Benefits

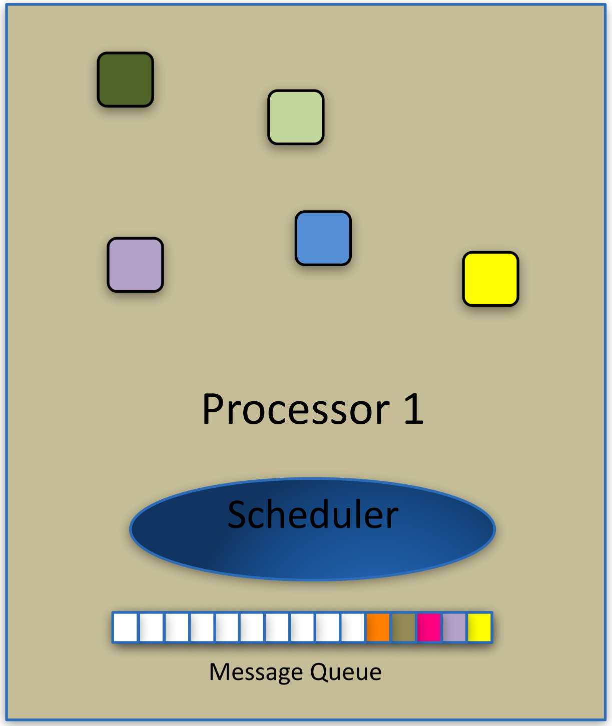


Locality and Prefetch



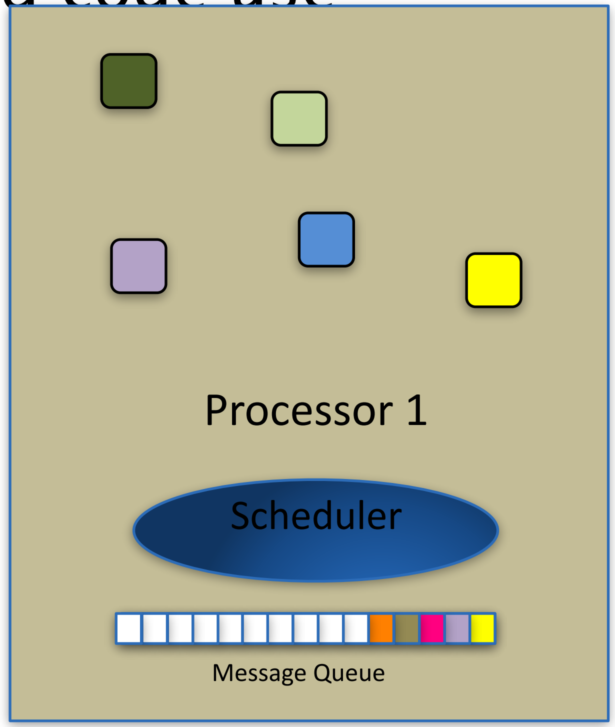
Locality and Prefetch

- Objects connote and promote locality



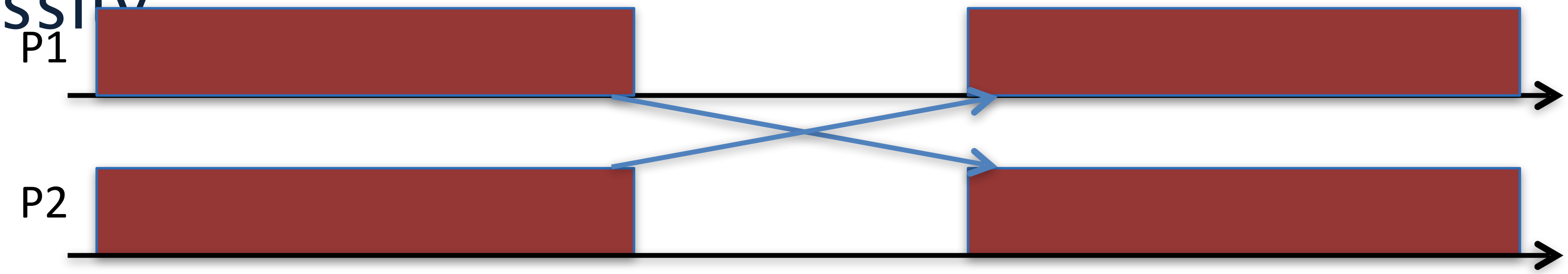
Locality and Prefetch

- 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

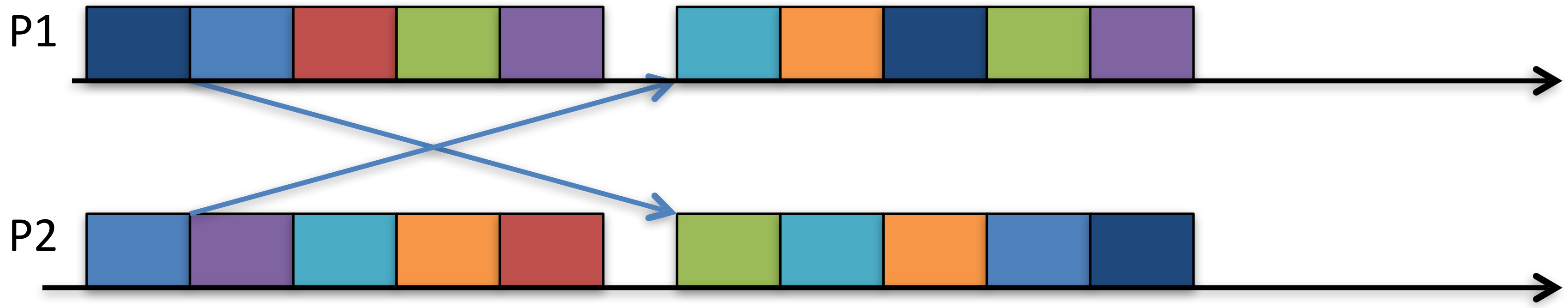
- Current use of communication network:
  - Compute-communicate cycles in typical MPI apps
  - The network is used for a fraction of time
    - And is on the critical path
- Current communication networks are over-engineered by necessity



BSP based application

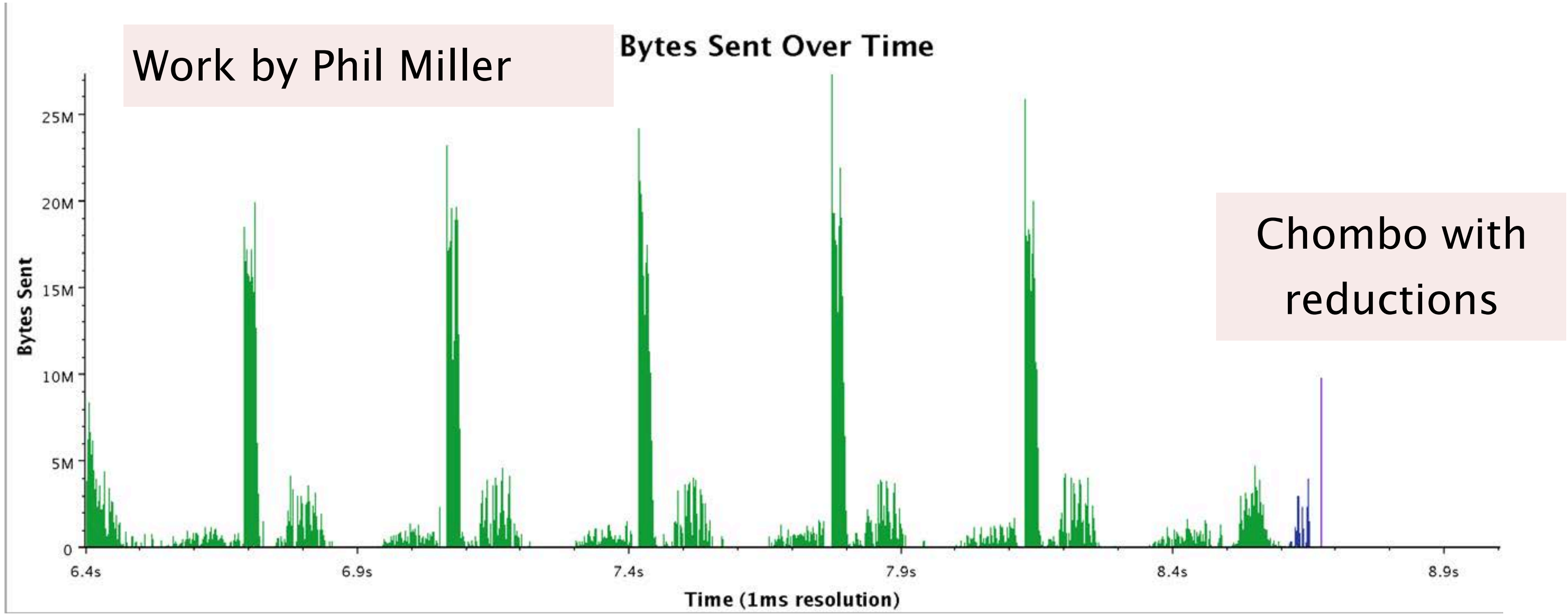
Impact on Communication

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

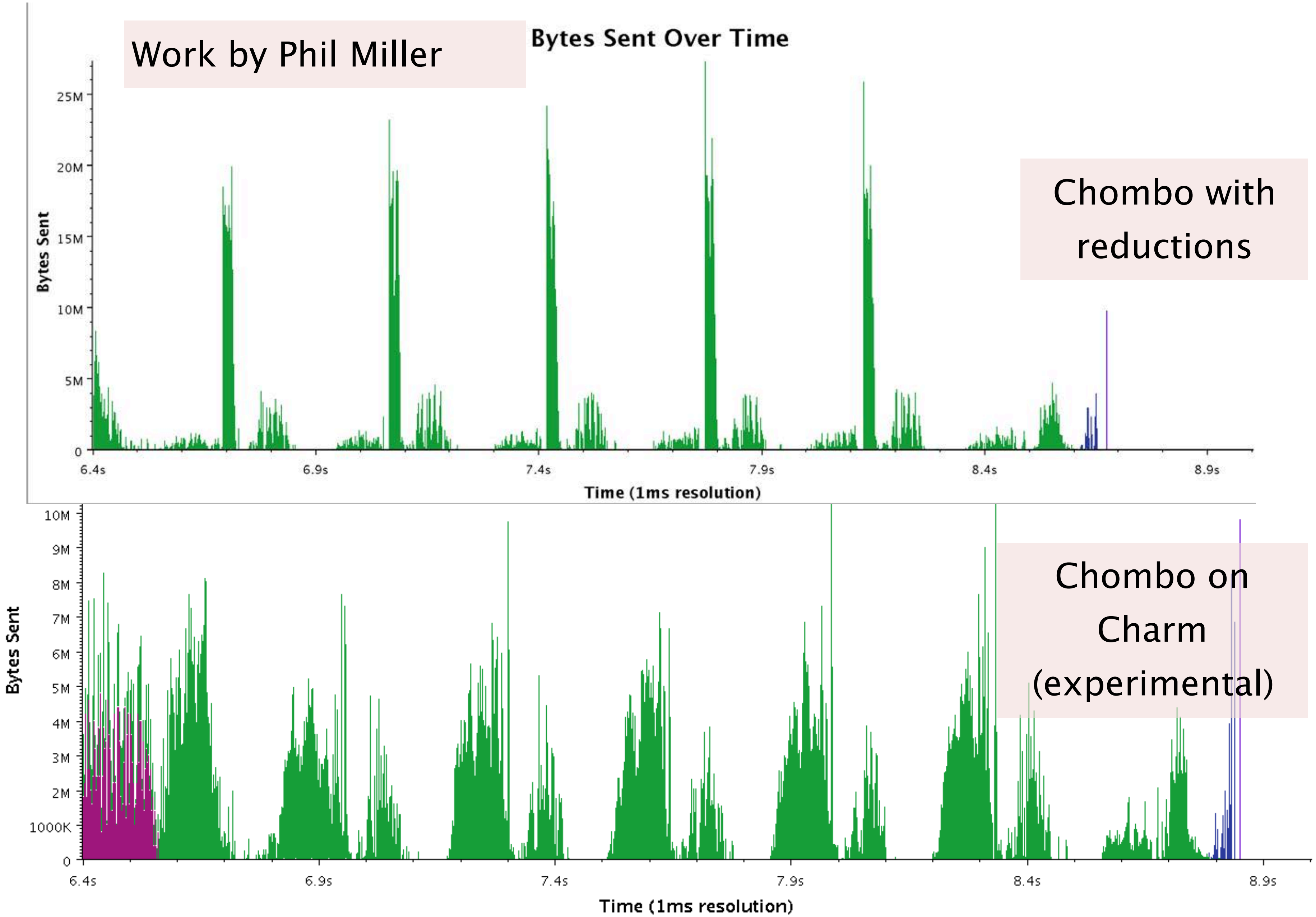


Overdecomposition enables overlap

Communication Data from Chombo



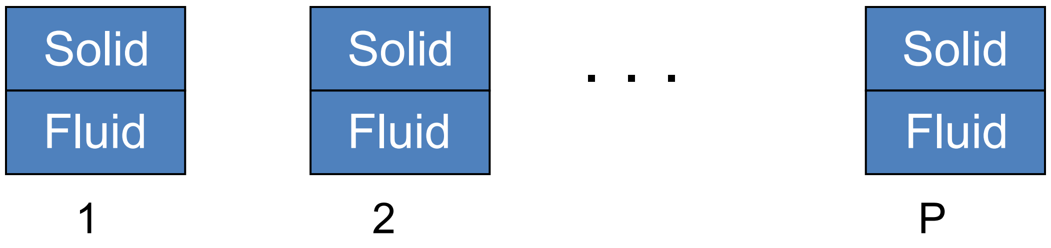
Communication Data from Chombo

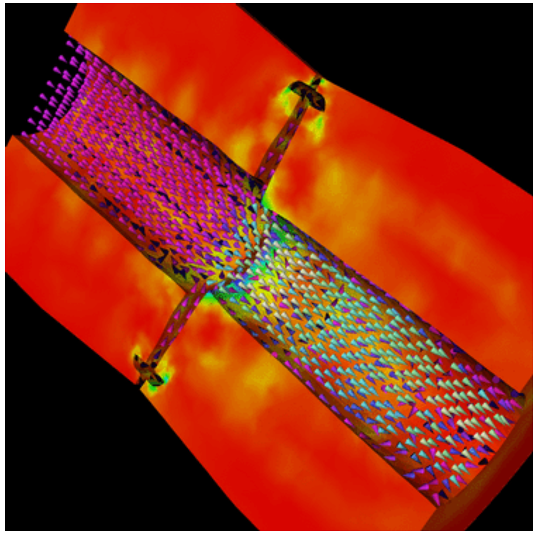


Decomposition Challenges

- 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

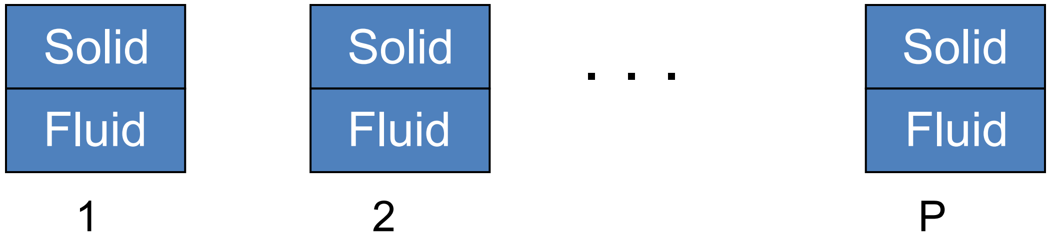
Decomposition Independent of numCores

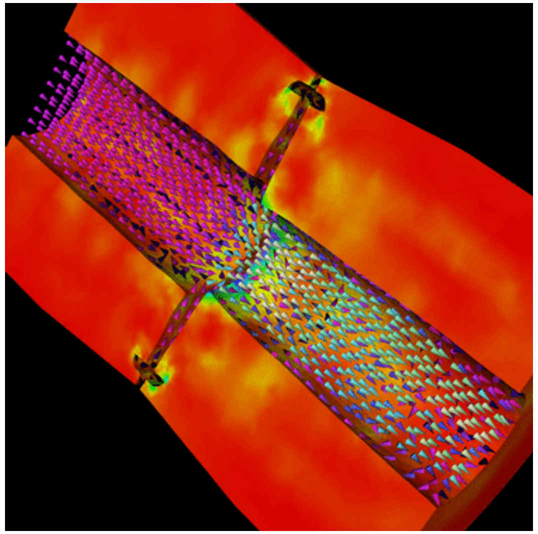




Decomposition Independent of numCores

- Rocket simulation example under traditional MPI

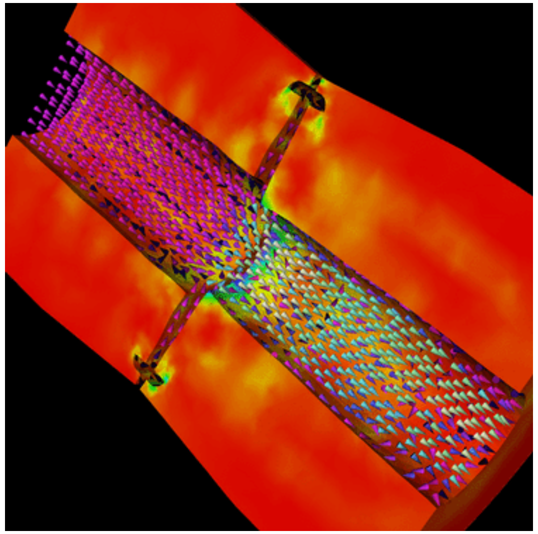




Decomposition Independent of numCores

- Rocket simulation example under traditional MPI

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



- With migratable-objects:

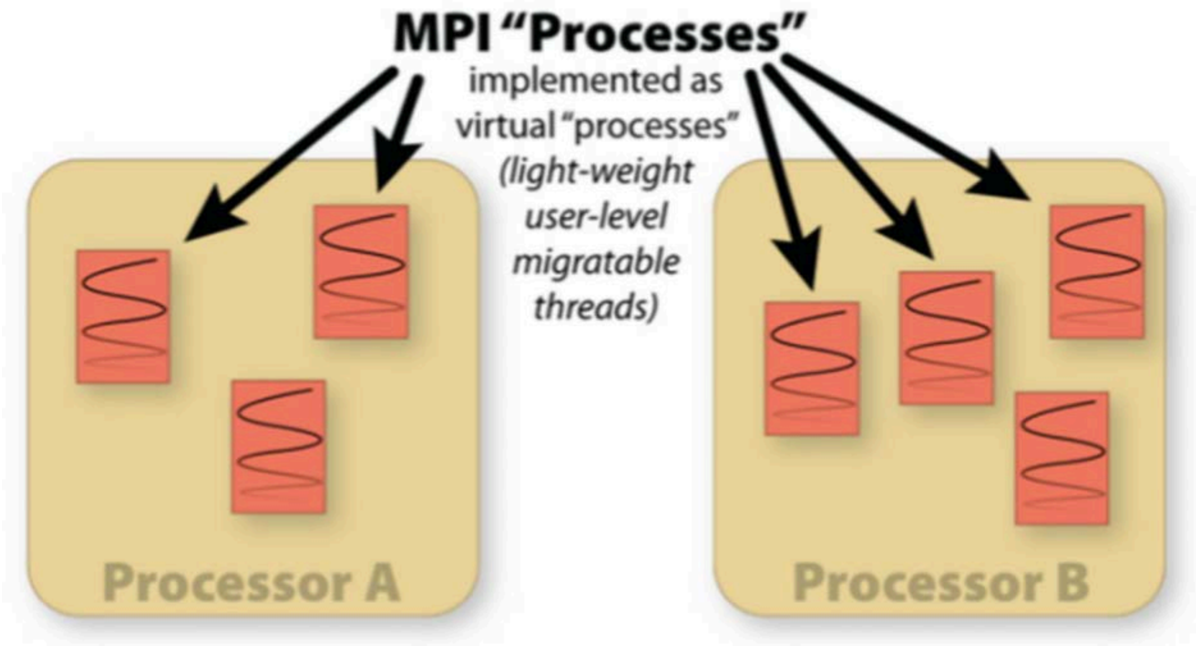
Solid_1 Solid_2 Solid_3 . . . Solid_n

Fluid_1 Fluid_2 . . . Fluid_m

  - Benefit: load balance, communication optimizations, modularity

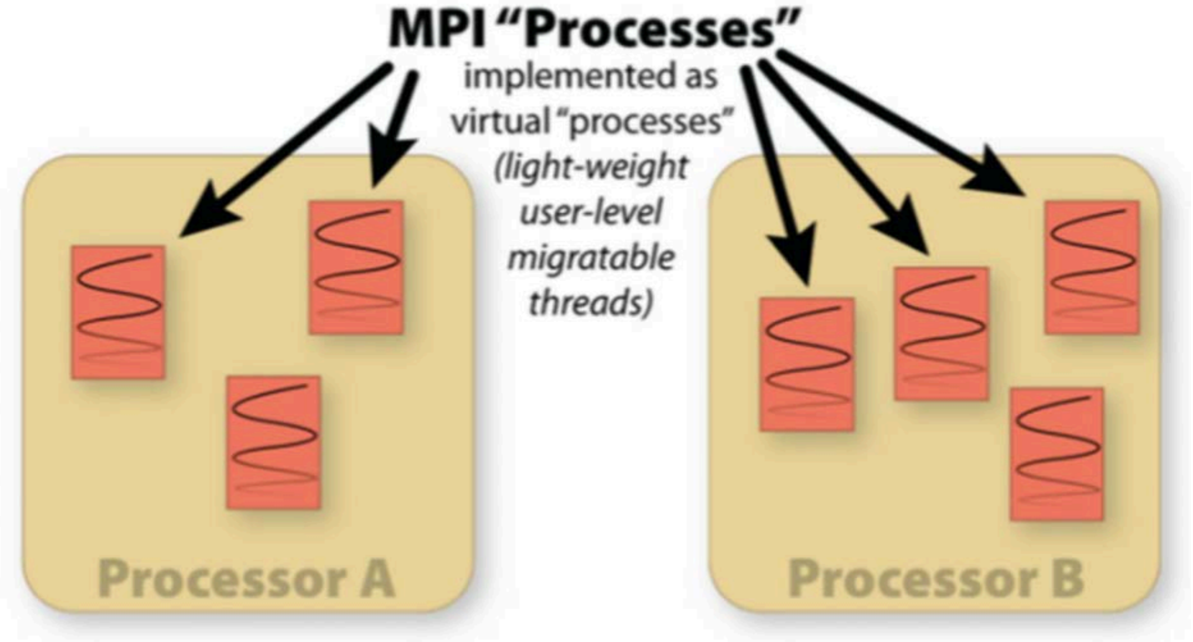
Adaptive MPI

What is Adaptive MPI?



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

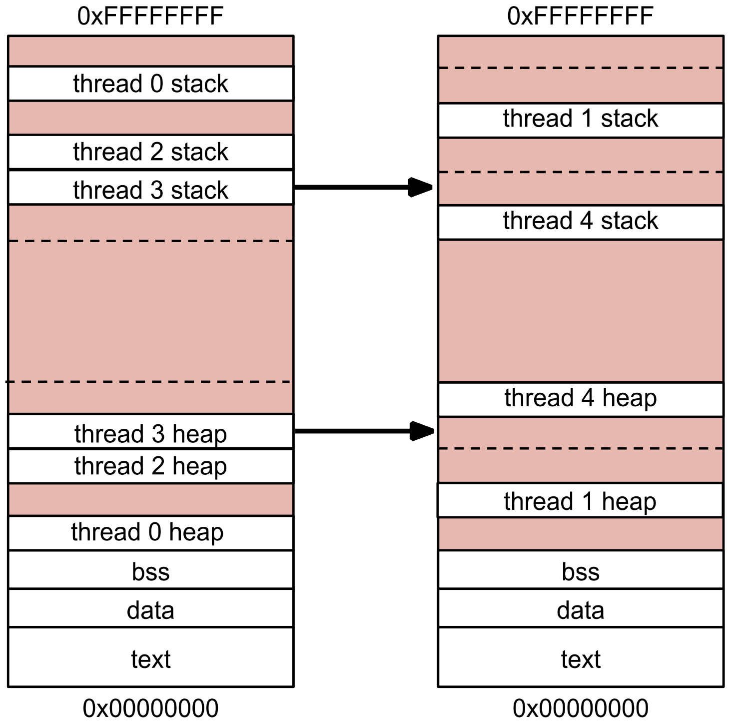


Process Virtualization

Process Virtualization

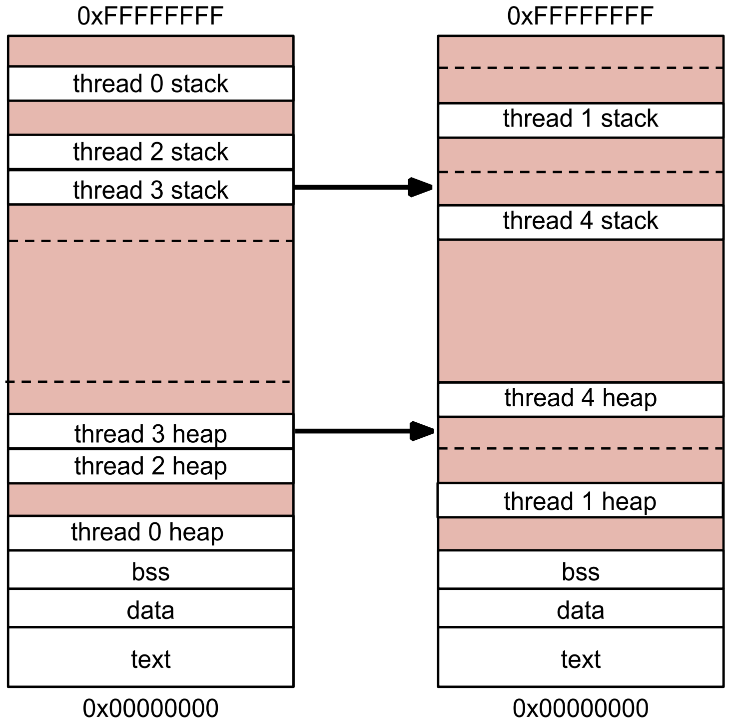
- AMPI virtualizes MPI "ranks", implementing them as migratable user-level threads rather than OS processes
  - Benefits:
    - Communication/computation overlap
    - Cache benefits to smaller working sets
    - Dynamic load balancing
    - Lower latency messaging within a process
  - Disadvantages:
    - Global/static variables are shared by all threads in an OS process scope
      - AMPI provides support for automating this at compile/run-time
      - Ongoing work to fully automate

Dynamic Load Balancing



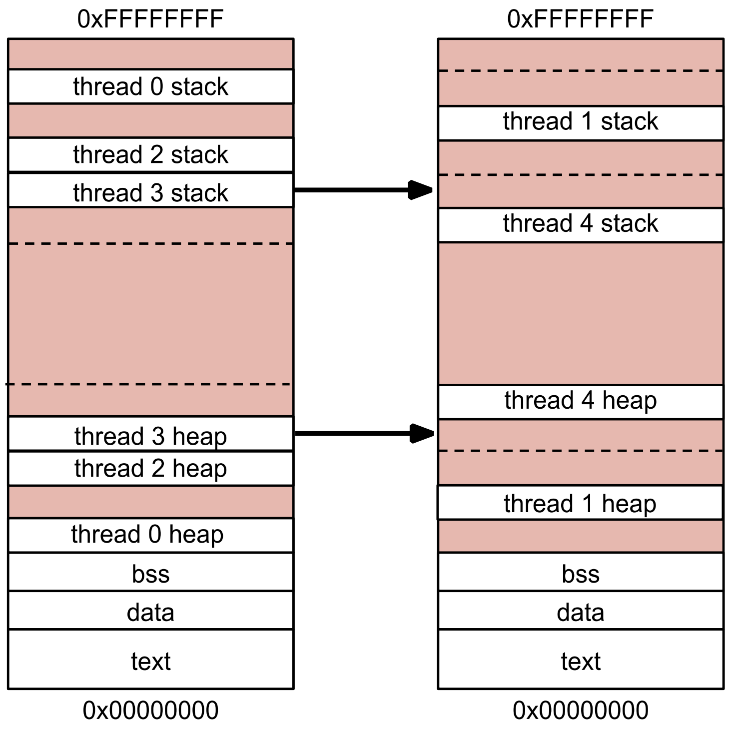
Dynamic Load Balancing

- 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



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



Fault Tolerance

Fault Tolerance

- AMPI ranks can be migrated to persistent storage or in remote memories for fault tolerance
  - Storage can be Disk, SSD, NVRAM, etc.

Fault Tolerance

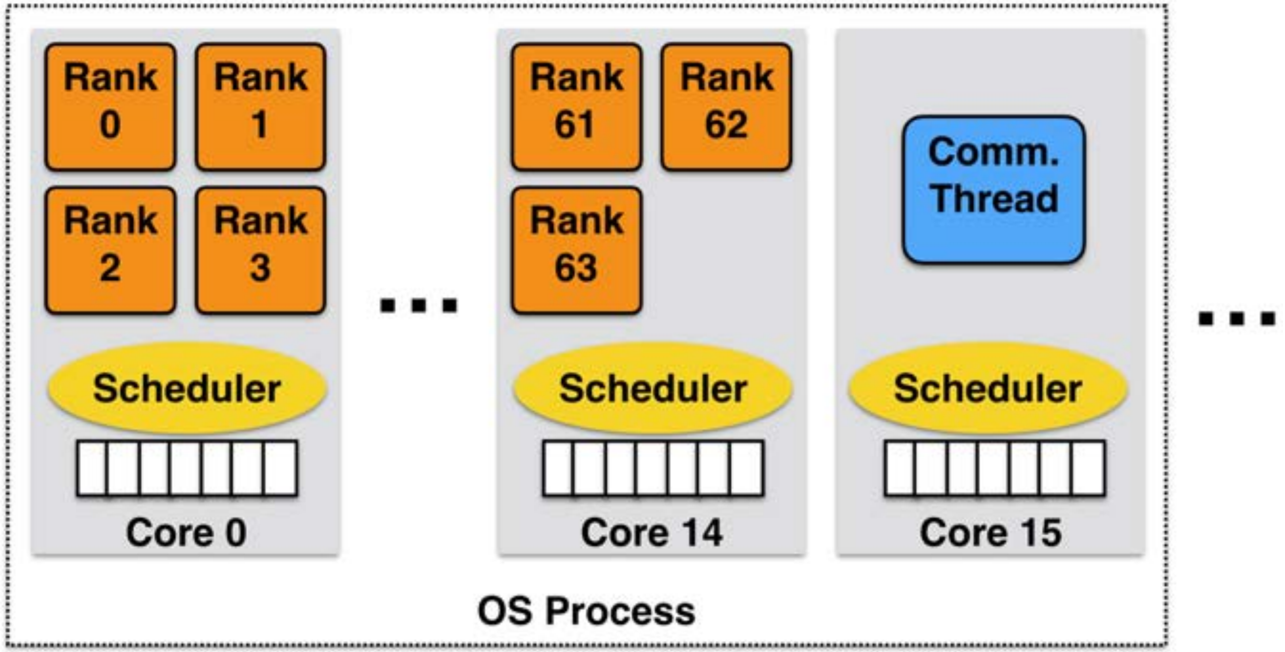
- AMPI ranks can be migrated to persistent storage or in remote memories for fault tolerance
  - Storage can be Disk, SSD, NVRAM, etc.

- The runtime uses a scalable fault detection algorithm and restarts automatically on a failure
  - Restart is online, within the same job

Fault Tolerance

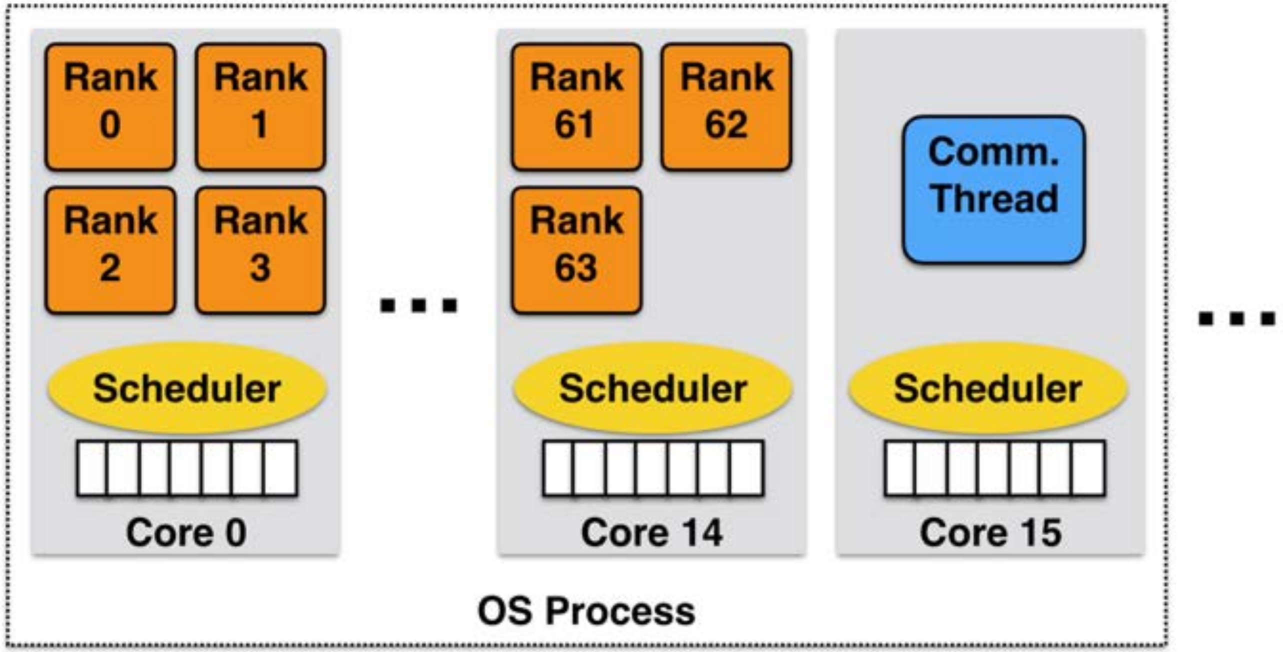
- AMPI ranks can be migrated to persistent storage or in remote memories for fault tolerance
  - Storage can be Disk, SSD, NVRAM, etc.
- The runtime uses a scalable fault detection algorithm and restarts automatically on a failure
  - Restart is online, within the same job
- Checkpointing strategy is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

Communication Optimizations

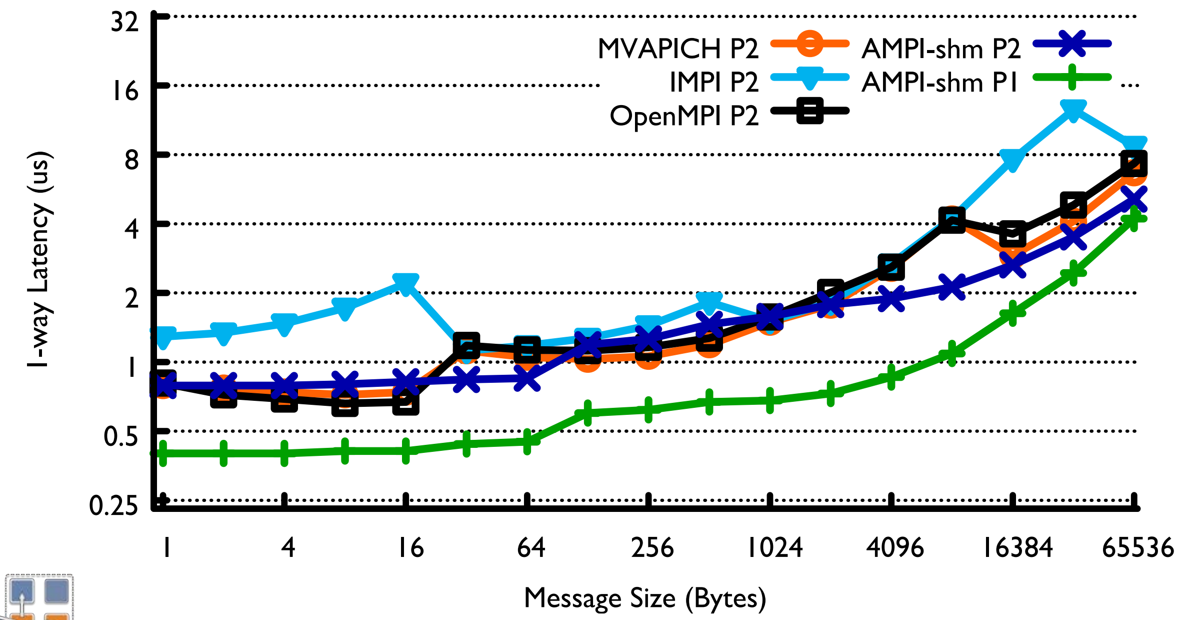


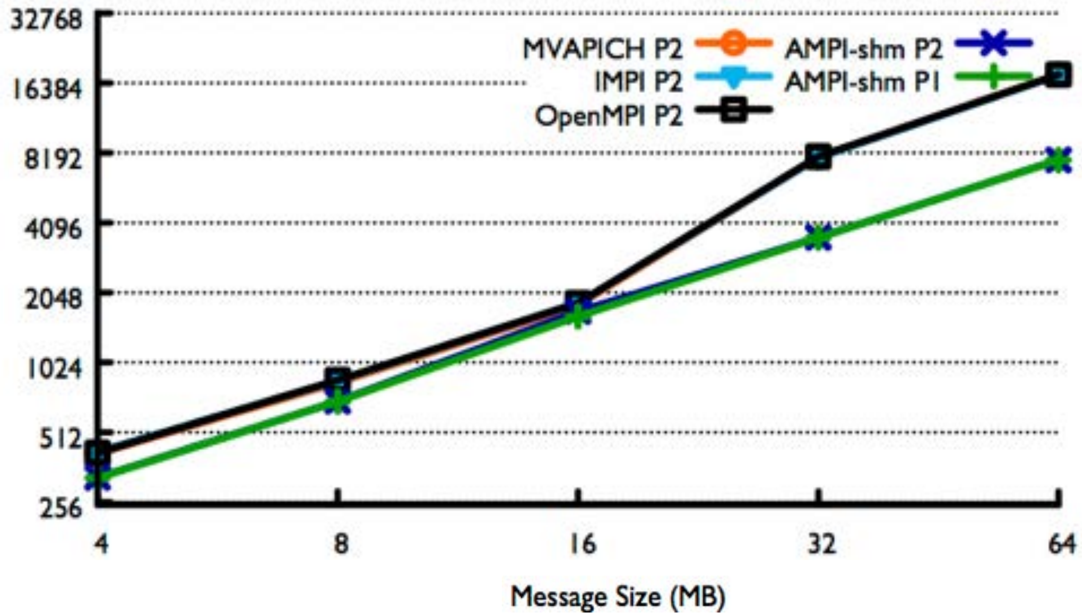
Communication Optimizations

- 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



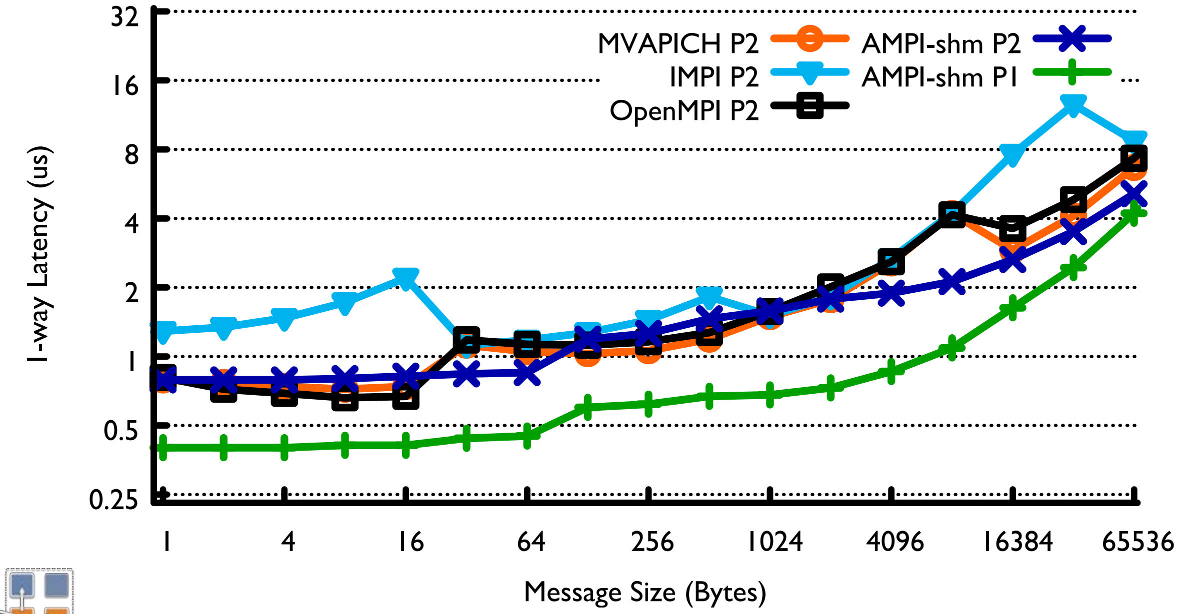
Communication Optimizations

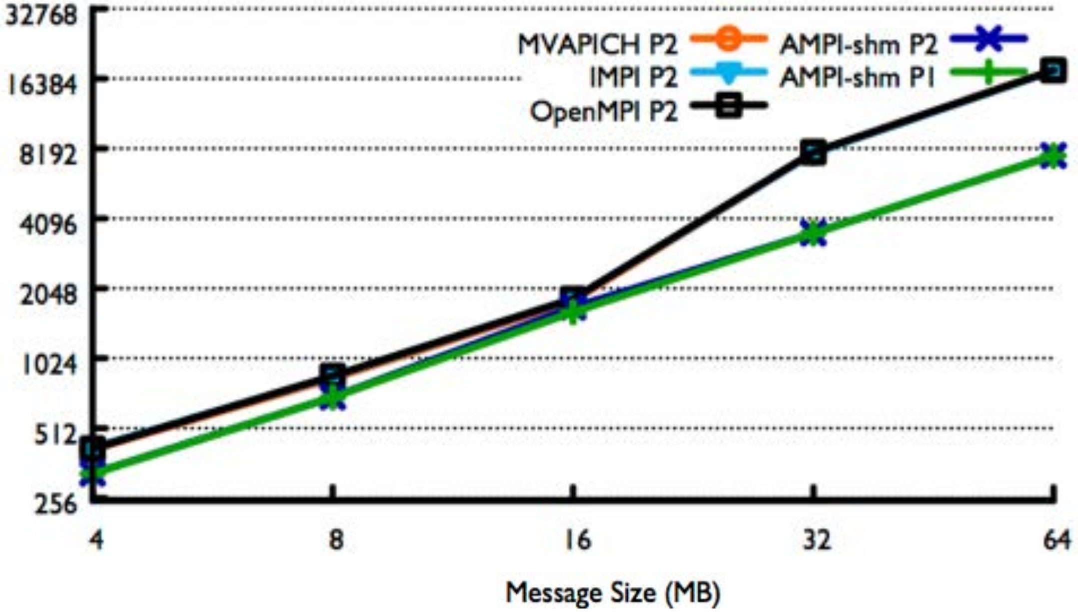




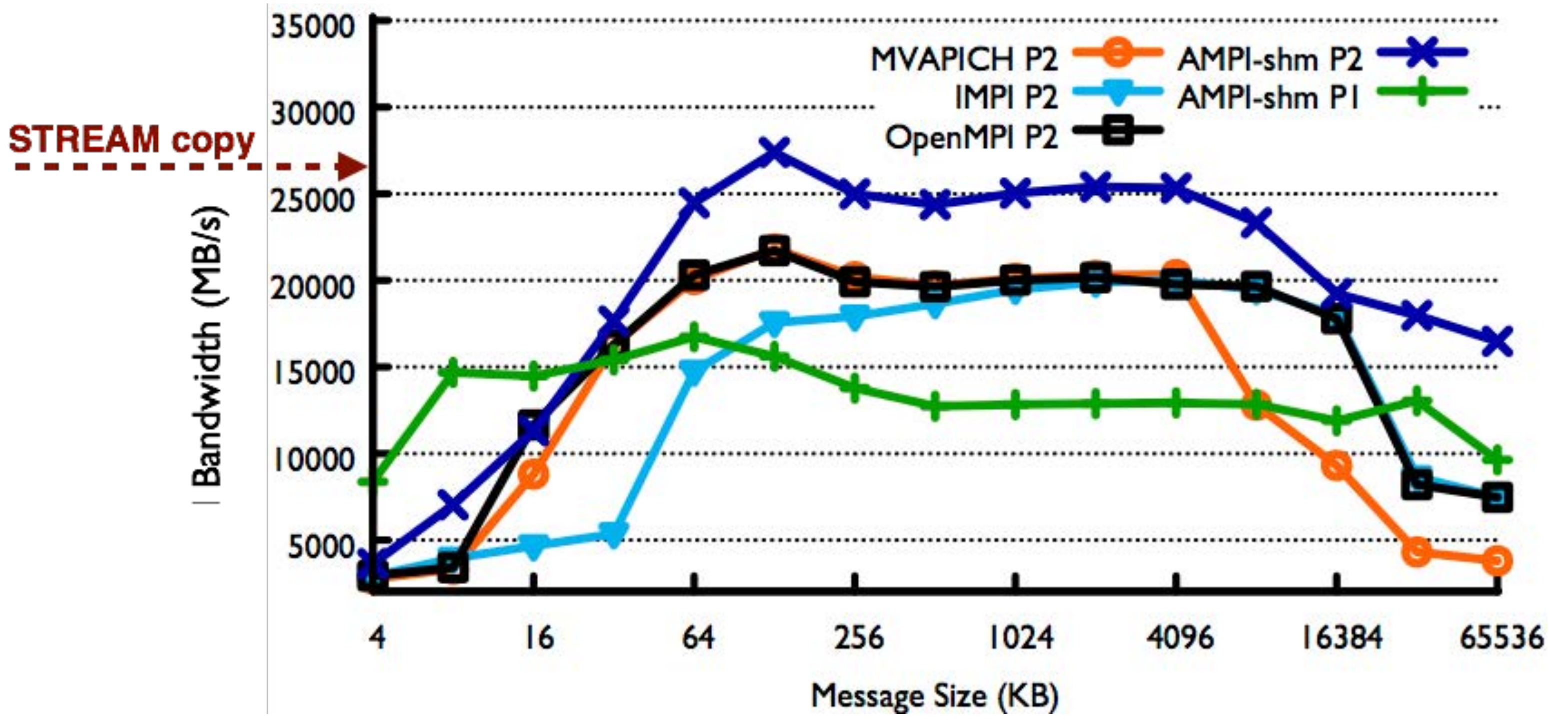
Communication Optimizations

- 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



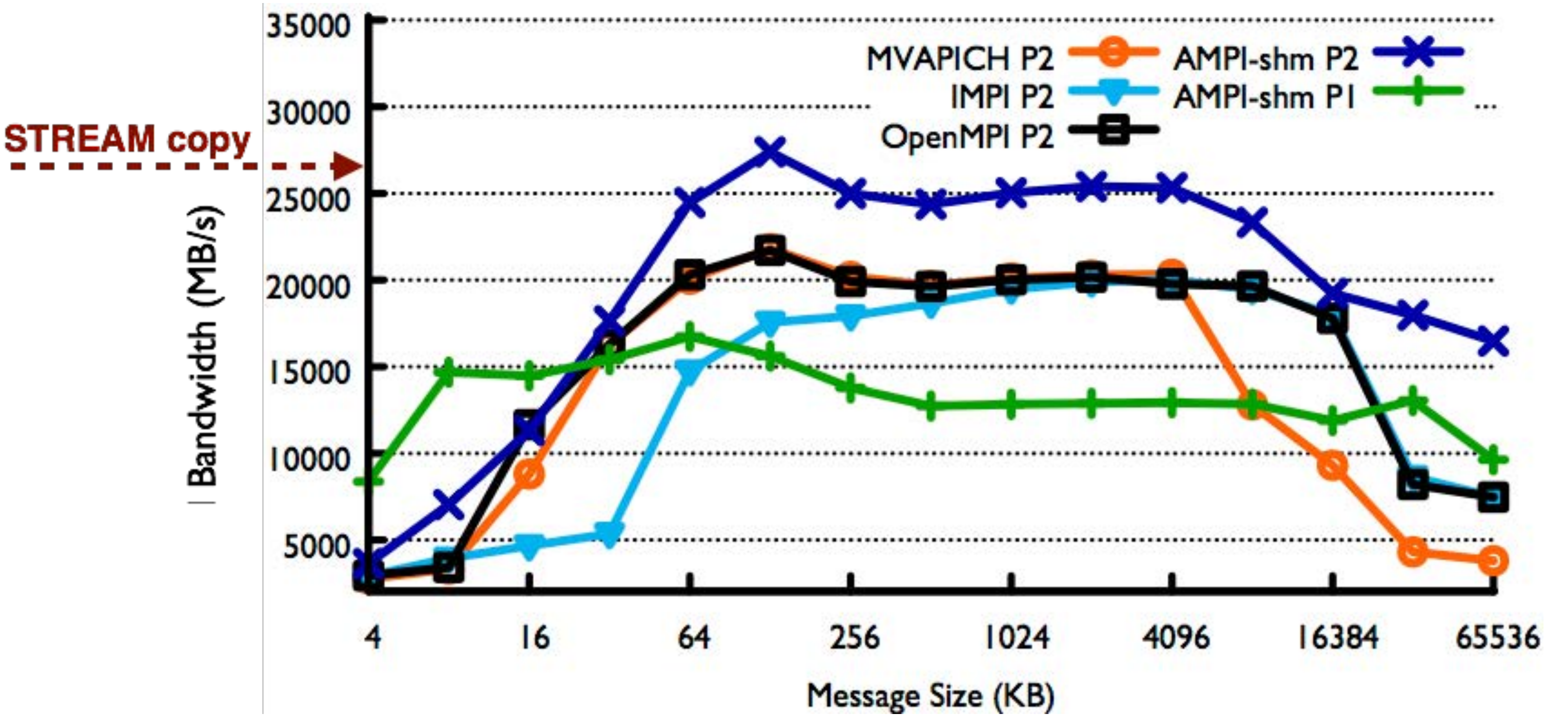


Communication Optimizations



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

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

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

Case Study

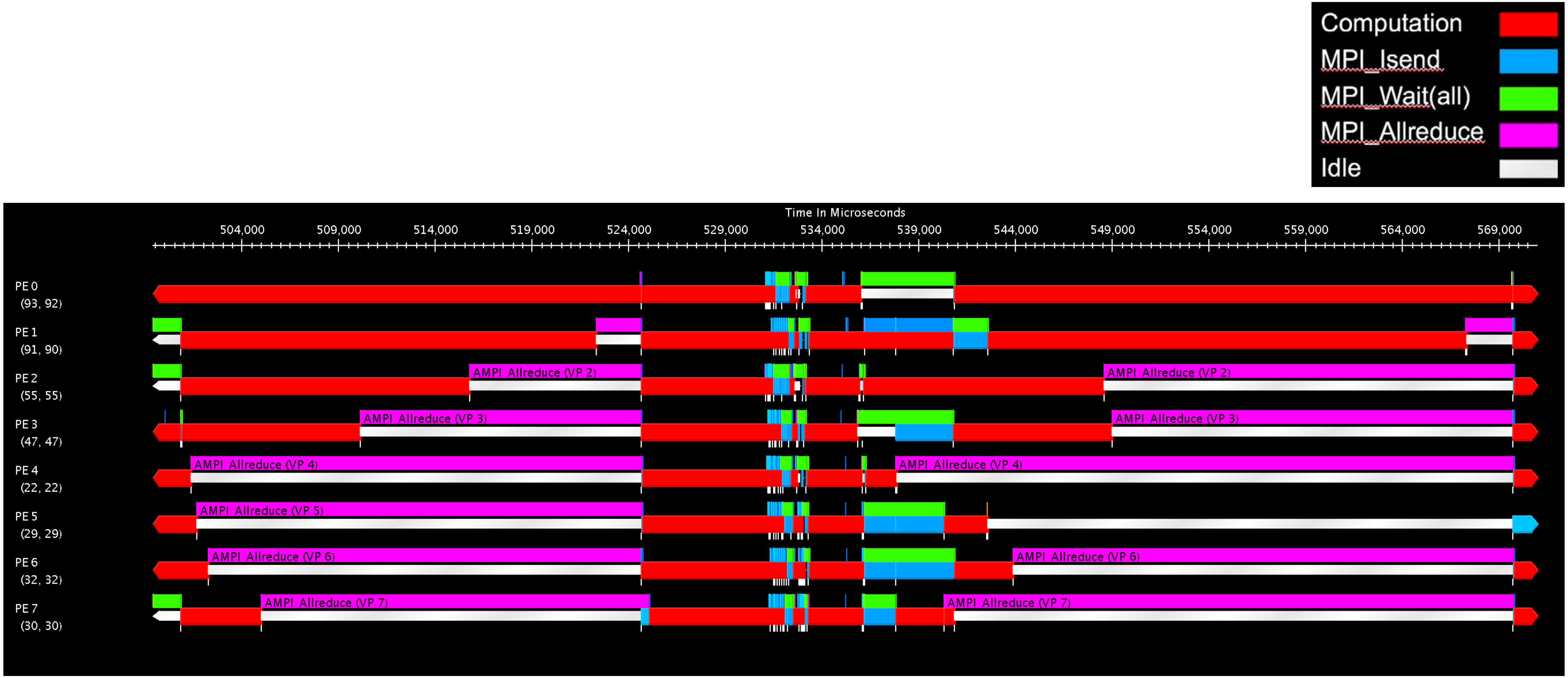
- LULESH proxy-application (LLNL)
  - Shock hydrodynamics on an unstructured mesh
  - With artificial load imbalance included to test runtimes

Case Study

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

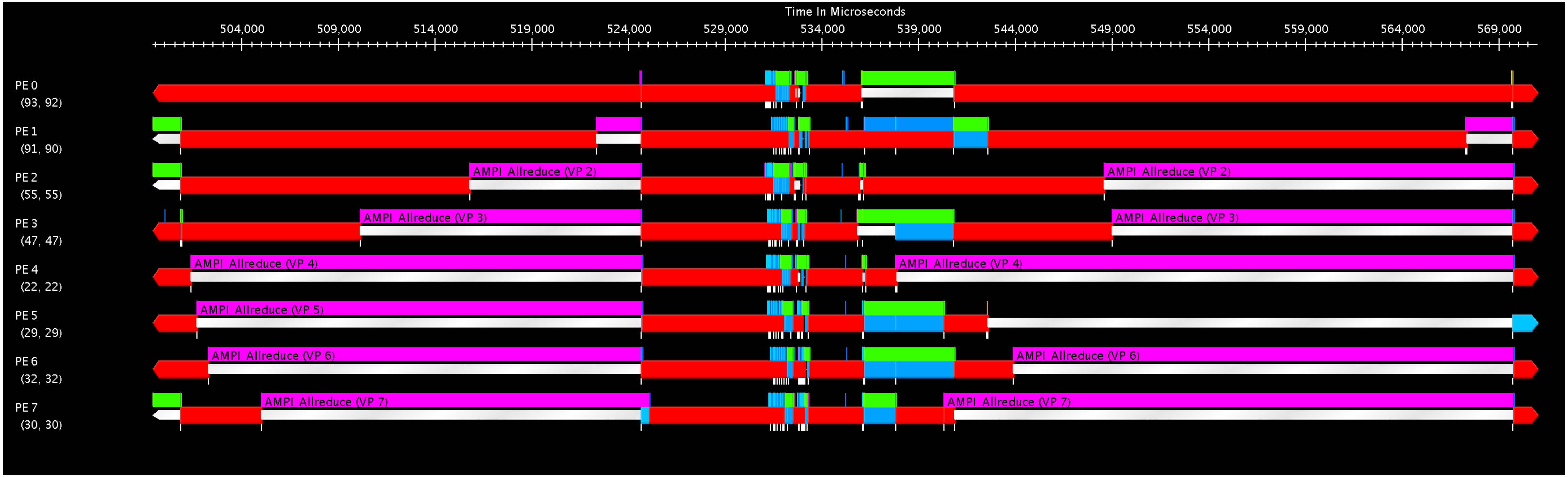
- No mutable global/static variables: can run on AMPI as is
  1. Replace mpicc with ampicc
  2. Link with "-module CommonLBs -memory isomalloc"
  3. Run with # of virtual processes and a load balancing strategy:
     - ./charmrun +p 2048 ./lulesh2.0 +vp 16384 +balancer GreedyLB

LULESH: Without Virtualization & LB

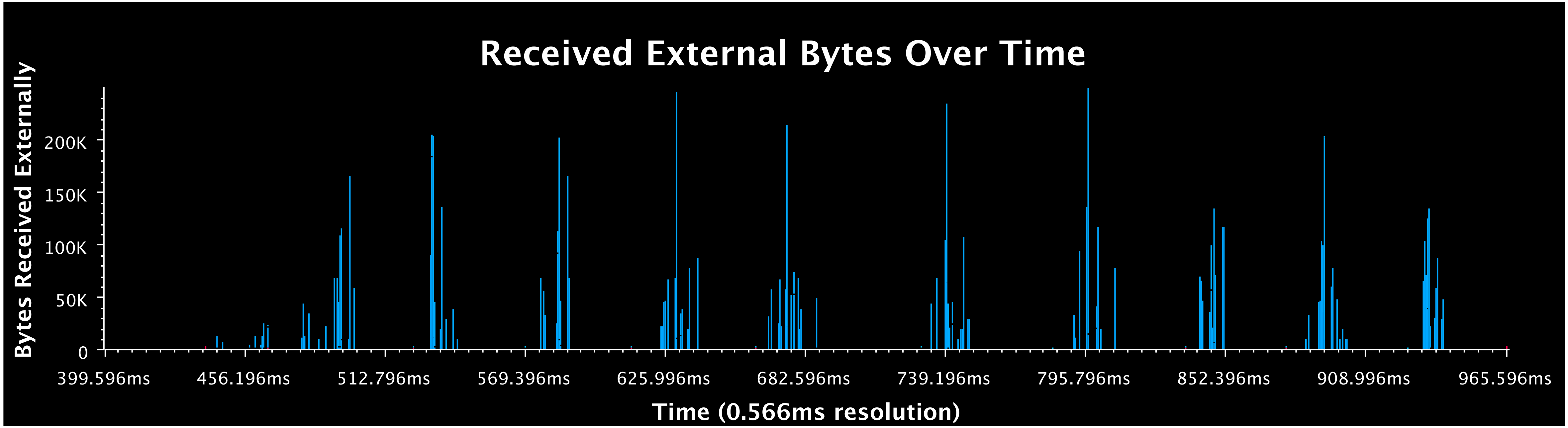


LULESH: Without Virtualization & LB

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

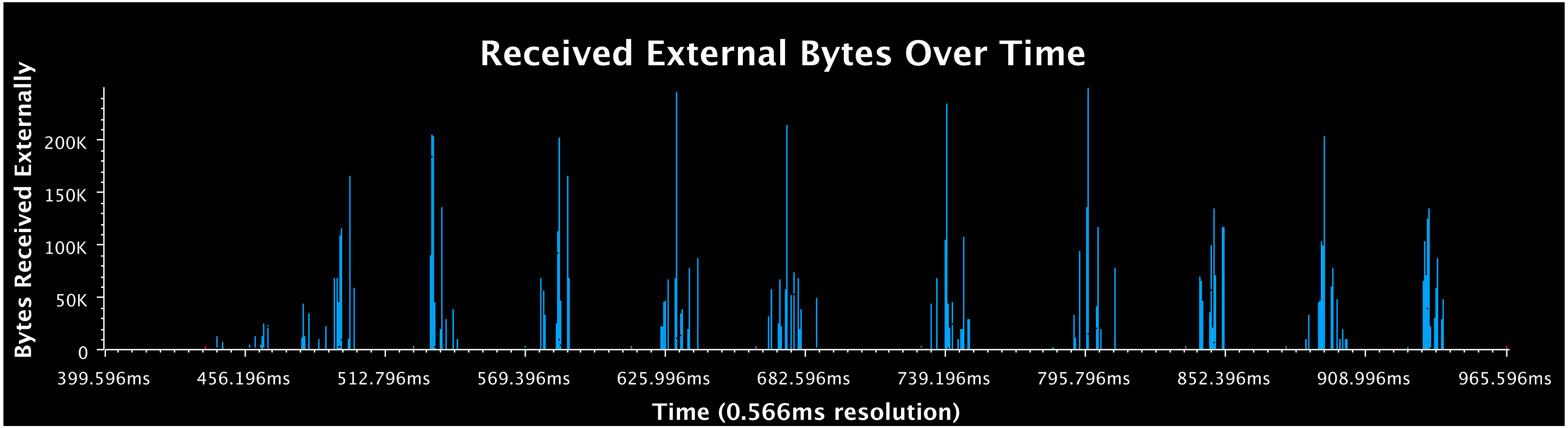


LULESH: Without Virtualization & LB

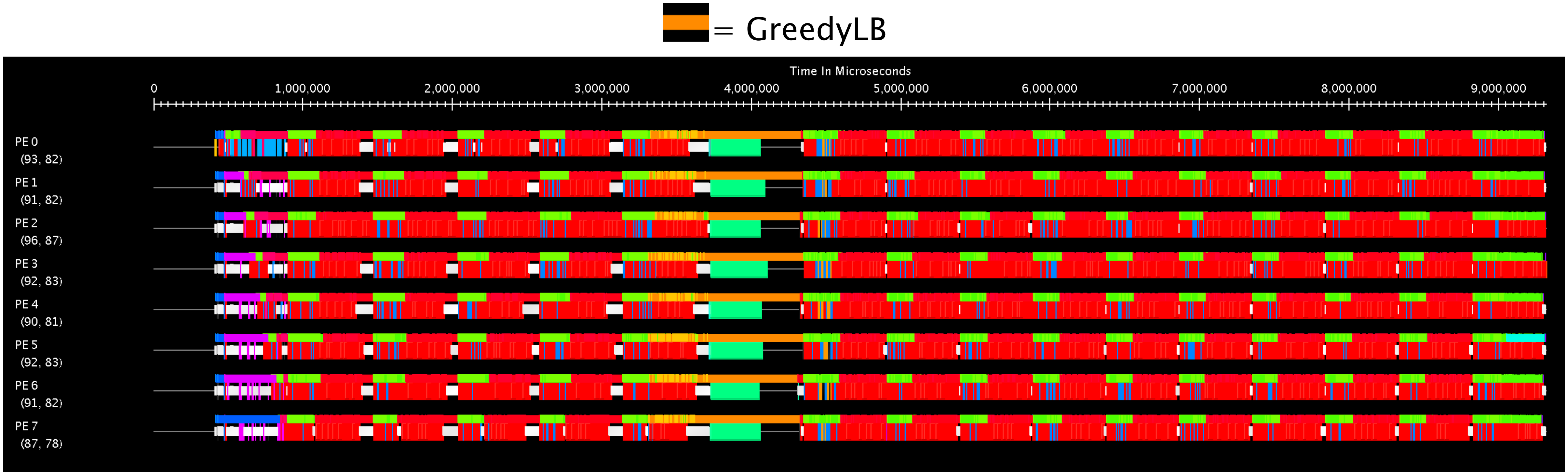


LULESH: Without Virtualization & LB

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

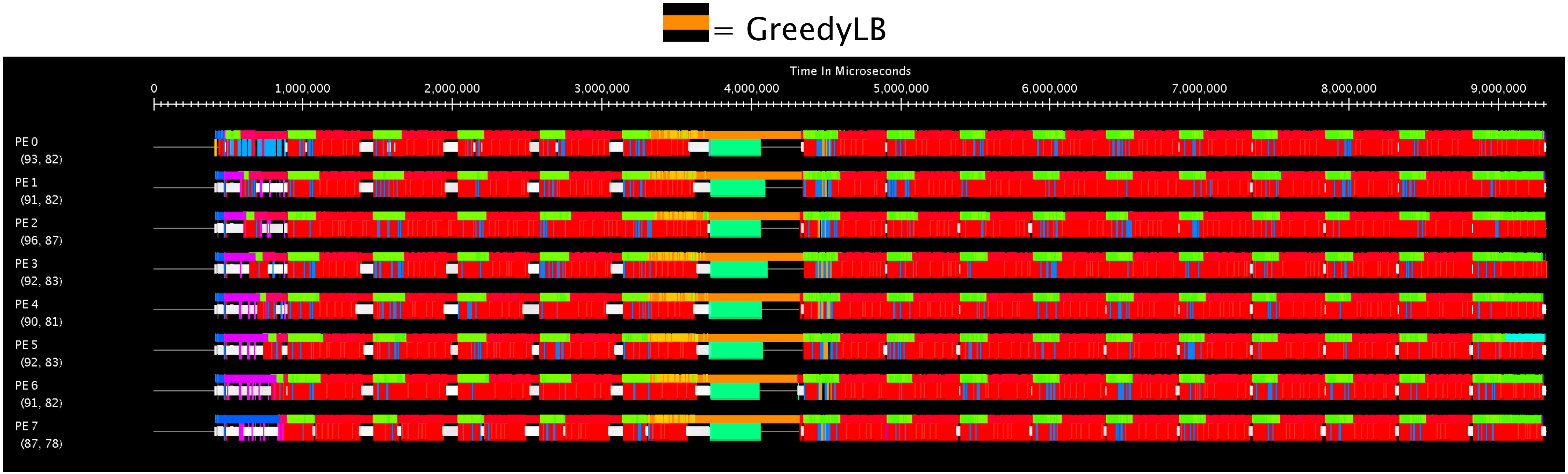


LULESH: With 8x Virtualization & LB

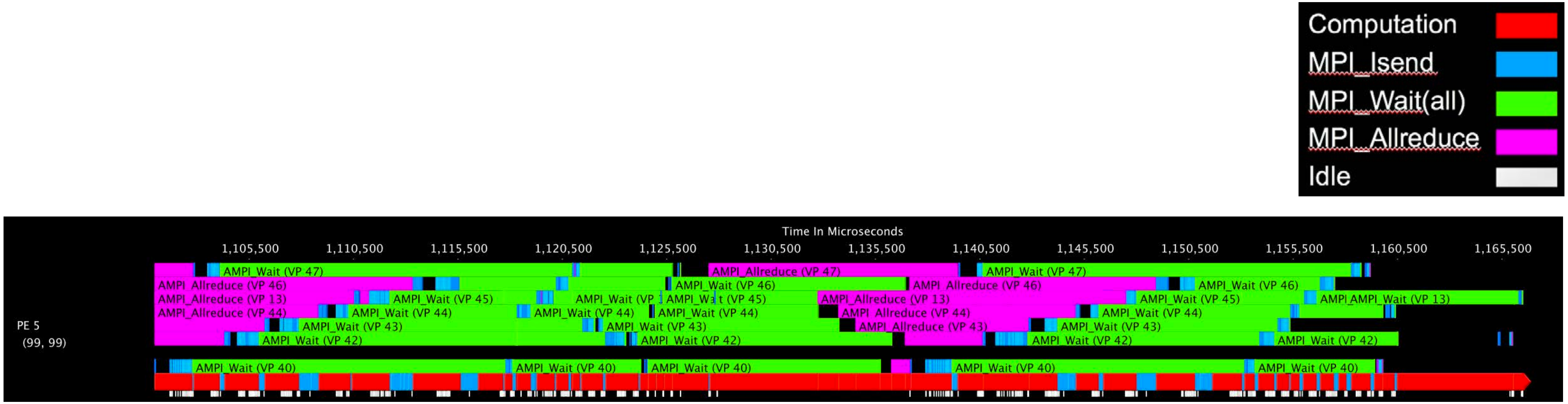


LULESH: With 8x Virtualization & LB

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

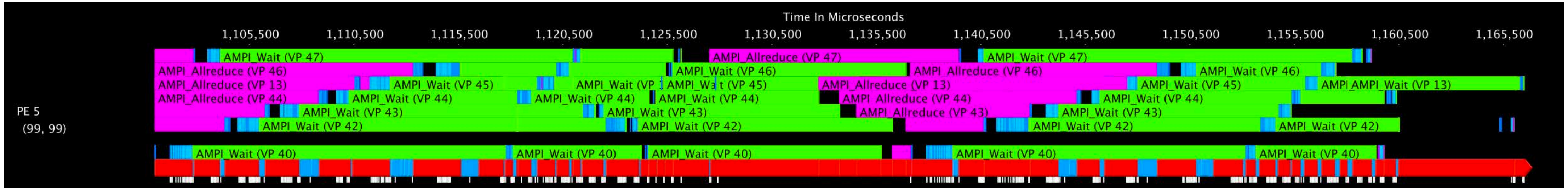


LULESH: With 8x Virtualization & LB



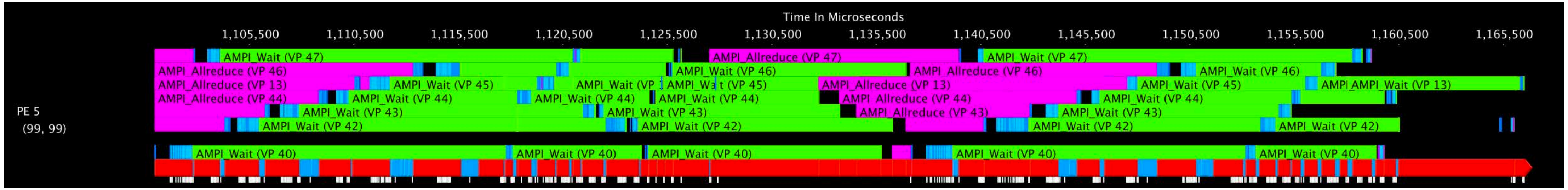
LULESH: With 8x Virtualization & LB

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



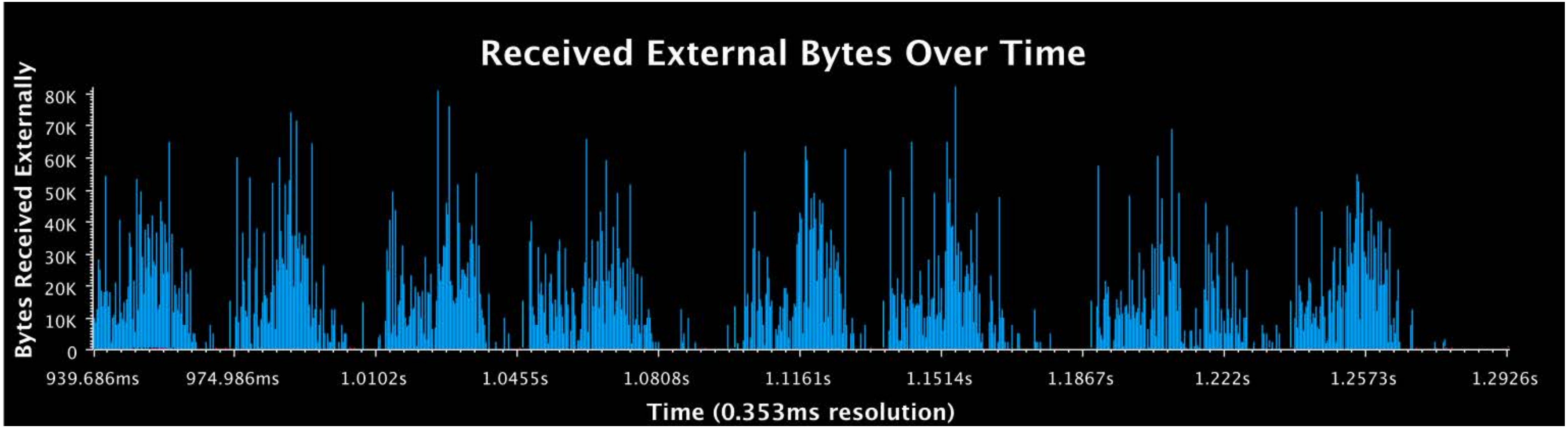
LULESH: With 8x Virtualization & LB

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



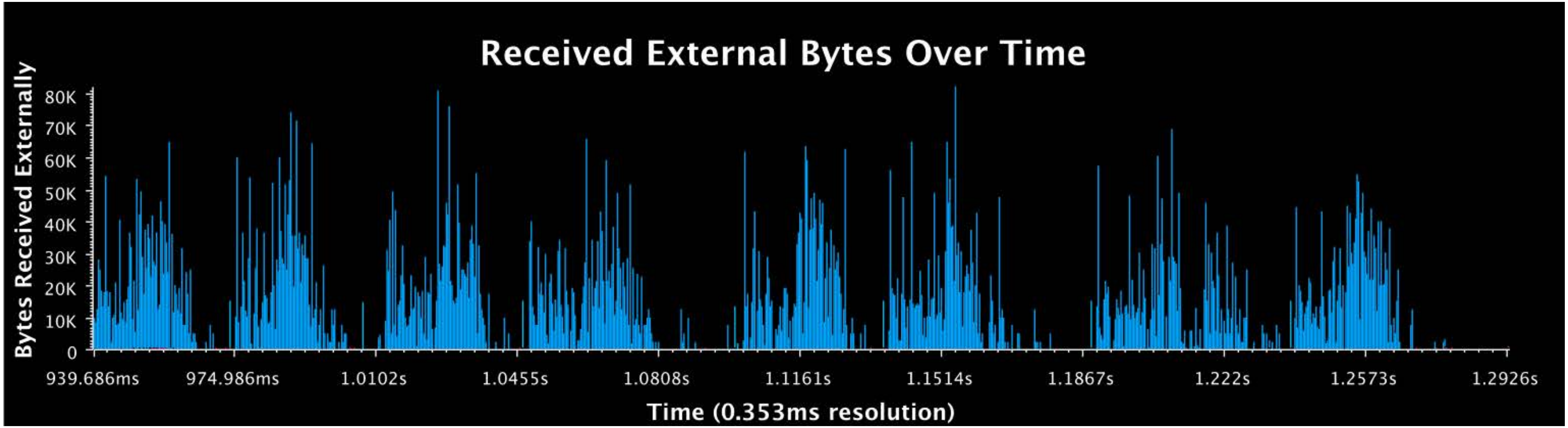
  - Projections allows viewing all virtual ranks on a PE, not only what is currently scheduled on one
    - In Projections Timeline, select: View -> Show Nested Bracketed User Events

LULESH: With 8x Virtualization & LB



LULESH: With 8x Virtualization & LB

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



AMPI Summary

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

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

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.

Hello World with Chares

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”
```

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”
```

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”
```

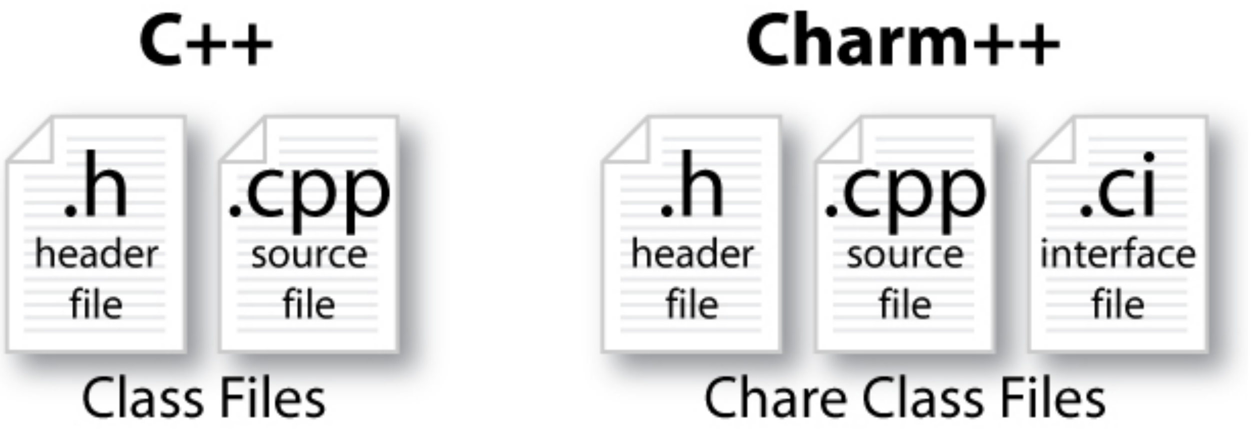
Charm++ File Structure

C++ **Charm++**

Class Files Chare Class Files

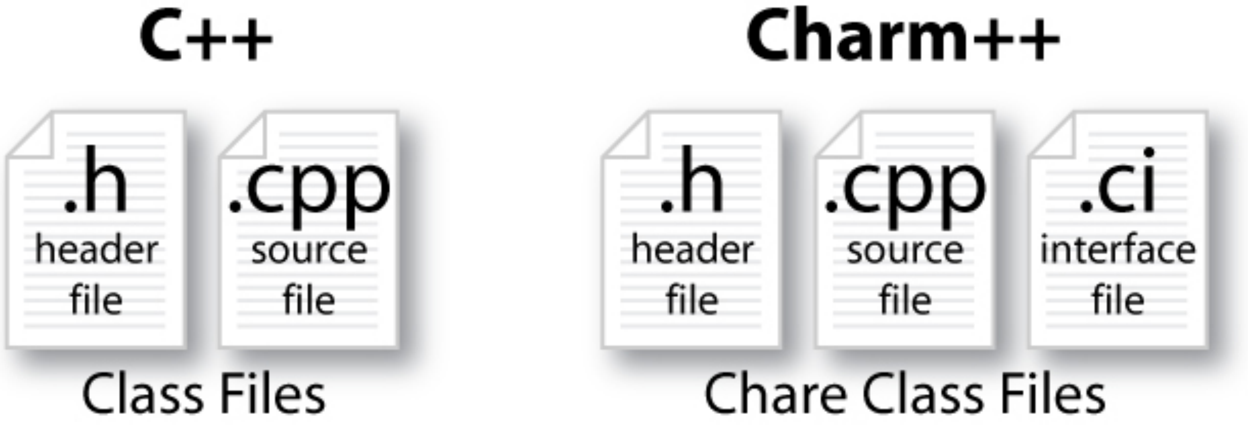
Charm++ File Structure

- C++ objects (including Charm++ objects)
  - Defined in regular .h and .cpp files



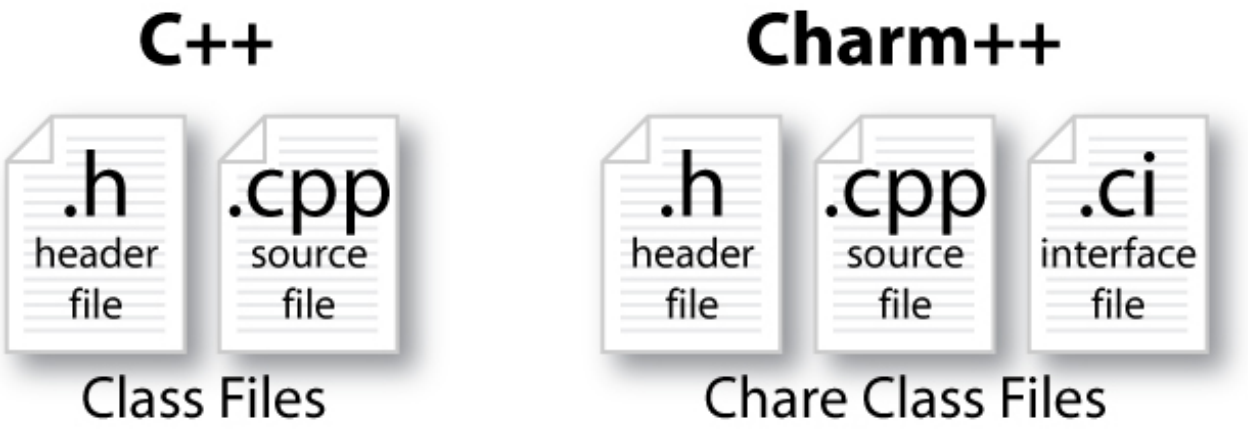
Charm++ File Structure

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



Charm++ File Structure

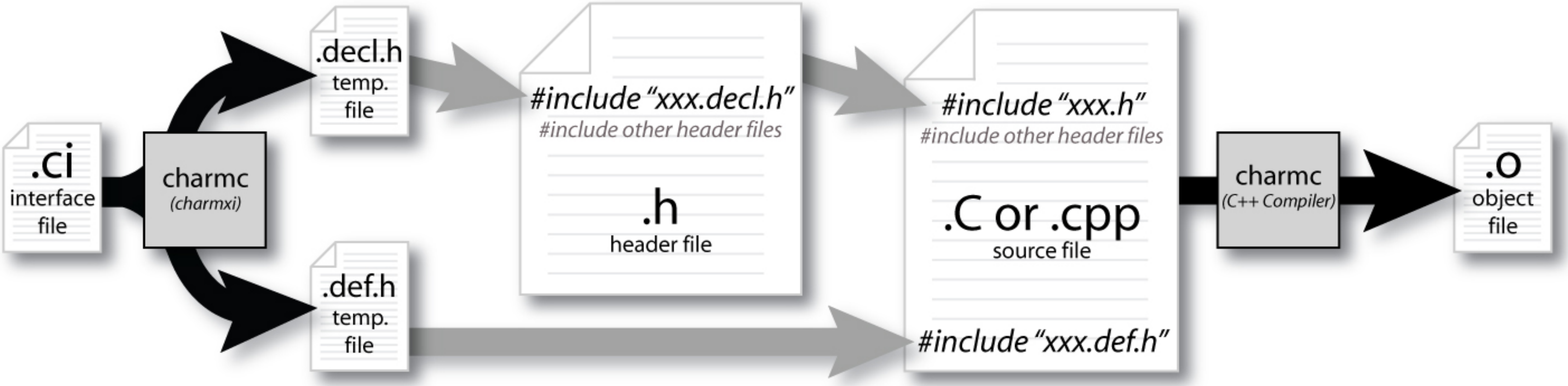
- C++ objects (including Charm++ objects)
  - Defined in regular .h and .cpp files
- Chare objects, entry methods (asynchronous methods)
  - Defined in .ci file
  - Implemented in the .cpp file



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.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”
```

Hello World with Chares

hello.ci

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

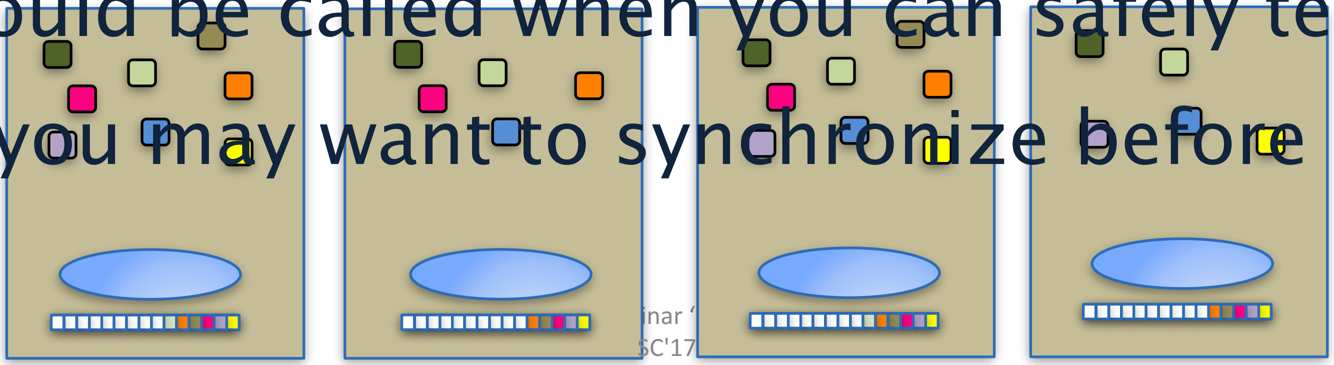
hello.cpp

```
#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?

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 (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 (done) 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
```

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 (done) 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
```

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
```

Chare Arrays

- Indexed collections of chares
  - Every item in the collection has a unique index and proxy
  - Can be indexed like an array or by an arbitrary object
  - Can be sparse or dense
  - Elements may be dynamically inserted and deleted
  - Elements are distributed across the available processors,
    - May be migrated to other nodes by the user or the runtime

- For many scientific applications, collections of chares are a convenient abstraction

Declaring a Chare Array

.ci file:

```
char        foo {
   entry foo(); // constructor
   // … entry methods …
}
char        bar {
   entry bar(); // constructor
   // … entry methods …
}
```

Declaring a Chare Array

.ci file:

```
array [1d] foo {
   entry foo(); // constructor
   // … entry methods …
}
array [2d] bar {
   entry bar(); // constructor
   // … entry methods …
}
```

Constructing a Chare Array

- Constructed much like a regular chare, using ckNew
- The size of each dimension is passed to the constructor at the end

```
void  someMethod() {
   CProxy_foo myFoo = CProxy_foo::ckNew(<params>, 10);   // 1d, size 10
   CProxy_bar myBar = CProxy_bar::ckNew(<params>, 5, 5); // 2d, size 5x5
}
```

- The proxy represents the entire array, and may be indexed to obtain a proxy to an individual element in the array

```
myFoo[4].invokeEntry(…);
myBar(2,4).method3(…);
```

thisIndex

- 1d: `thisIndex` returns the index of the current chare array element
- 2d: `thisIndex.x` and `thisIndex.y` return the indices of the current chare array element

.ci file:

```
array [1d] foo {
   entry foo();
}
```

.cpp file:

```
struct foo : public CBase_foo {
   foo() {
      ckout << “array index: ” << thisIndex;
   }
};
```

Chare Array: Hello Example

```
mainmodule arr {
   mainchare Main {
      entry Main(CkArgMsg*);
   }
   array [1D] hello {
      entry hello(int);
      entry void printHello();
   }
}
```

Chare Array: Hello Example

```
#include "arr.decl.h"
struct Main : CBase_Main {
   Main(CkArgMsg* msg) {
      int arraySize = atoi(msg->argv[1]);
      CProxy_hello p = CProxy_hello::ckNew(arraySize, arraySize);
      p[0].printHello();
   }
};
```

Chare Array: Hello Example

```
#include “arr.decl.h”
struct Main : CBase_Main {
   Main(CkArgMsg* msg) {
      int arraySize = atoi(msg->argv[1]);
      CProxy_hello p = CProxy_hello::ckNew(arraySize, arraySize);
      p[0].printHello();
   }
};
struct hello : CBase_hello {
   int arraySize;
   hello(int n) : arraySize(n) { }
   void printHello() {
      CkPrintf(“PE[%d]: hello from p[%d]\n”, CkMyPe(), thisIndex);
      if (thisIndex == arraySize – 1) CkExit();
      else thisProxy[thisIndex + 1].printHello();
   }
};
#include “arr.def.h”
```

Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object

Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object
- From a chare array element that is a member of the same array:

Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object
- From a chare array element that is a member of the same array:

```
thisProxy.foo();
```

Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object
- From a chare array element that is a member of the same array:

```
thisProxy.foo();
```

- From any chare that has a proxy p to the chare array

Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object
- From a chare array element that is a member of the same array:

```
thisProxy.foo();
```

- From any chare that has a proxy p to the chare array

```
p.foo();
```

Reduction

Reduction

- Combines a set of values:

Reduction

- Combines a set of values:
- The operator must be commutative and associative
  - `sum`, `max`, …

- Each object calls `contribute` in a reduction

Reduction: Example

```
#include “reduction.decl.h”
const int numElements = 49;
class Main : public CBase_Main {
public:
  Main(CkArgMsg* msg) { CProxy_Elem::ckNew(thisProxy, numElements); }
  void done(int value) { CkPrintf(“value: %d\n”, value); CkExit(); }
};

class Elem : public CBase_Elem {
public:
  Elem(CProxy_Main mProxy) {
    int val = thisIndex;
    CkCallback cb(CkReductionTarget(Main, done), mProxy);
    contribute(sizeof(int), &val, CkReduction::sum_int, cb);
  }
};
#include “reduction.def.h”
```

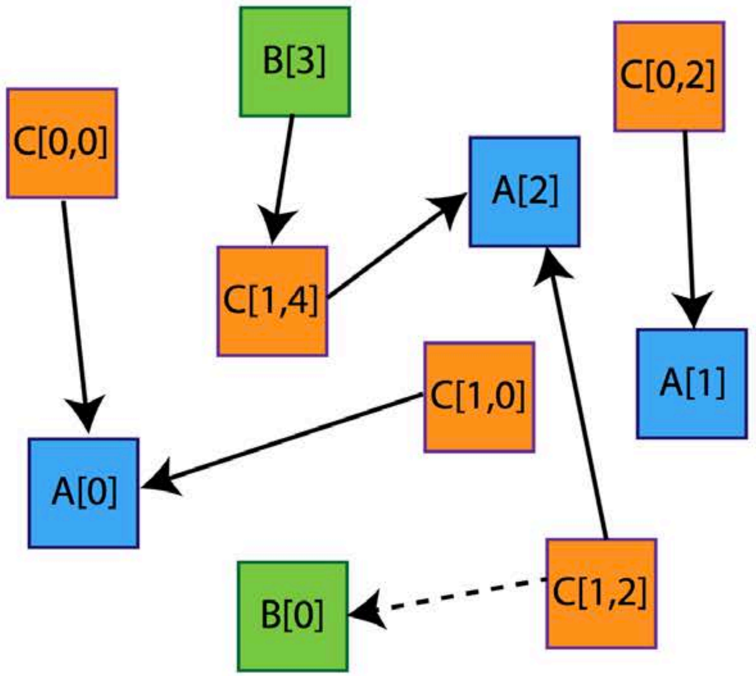
Reduction: Example

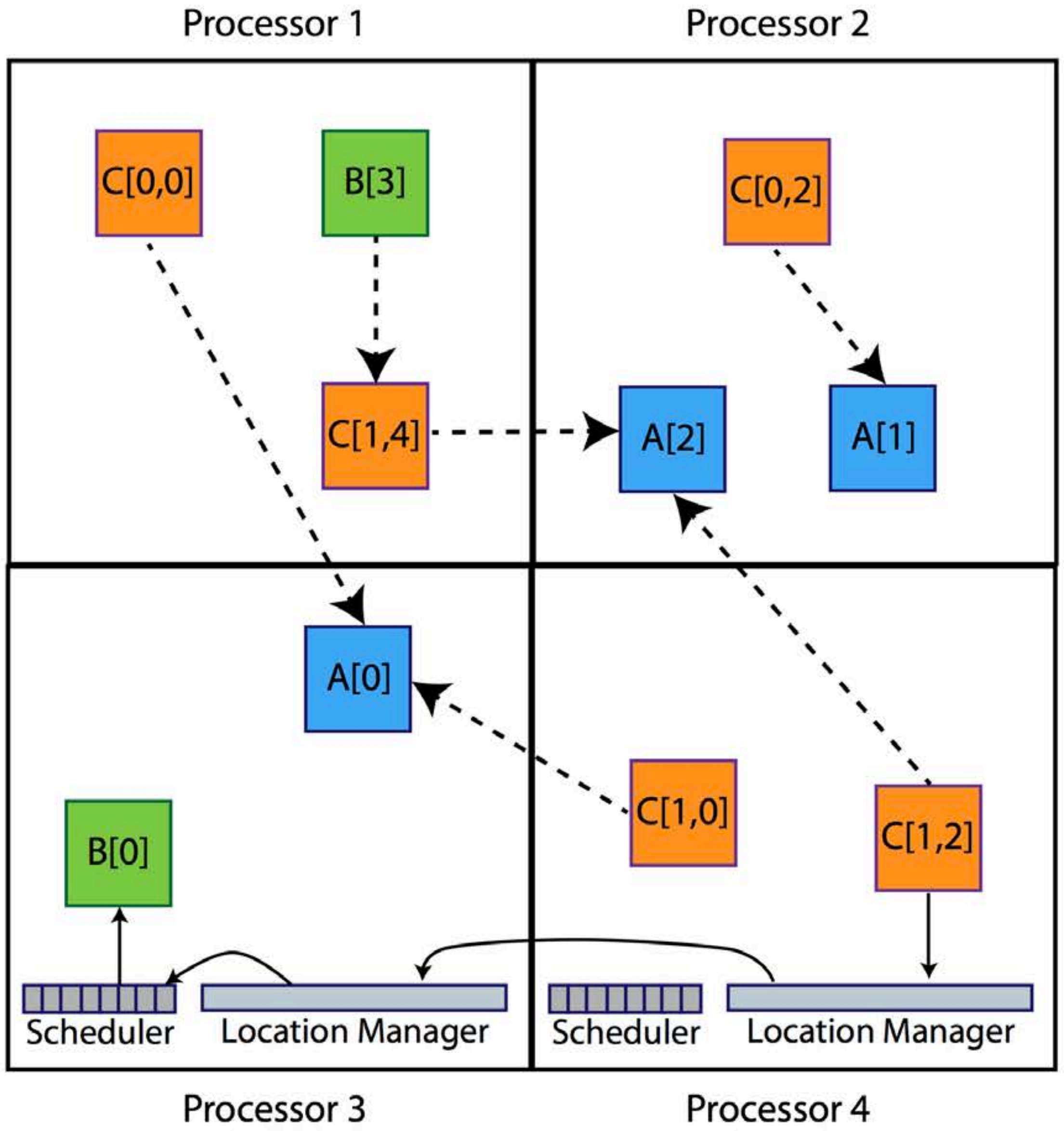
```
#include “reduction.decl.h”
const int numElements = 49;
class Main : public CBase_Main {
public:
  Main(CkArgMsg* msg) { CProxy_Elem::ckNew(thisProxy, numElements); }
  void done(int value) { CkPrintf(“value: %d\n”, value); CkExit(); }
};

class Elem : public CBase_Elem {
public:
  Elem(CProxy_Main mProxy) {
    int val = thisIndex;
    CkCallback cb(CkReductionTarget(Main, done), mProxy);
    contribute(sizeof(int), &val, CkReduction::sum_int, cb);
  }
};
#include “reduction.def.h”
```

Output
value: 1176
Program finished.

Chare Arrays view





Dynamic Load Balancing

- Object-based decomposition (i.e. virtualized decomposition) helps
  - Charm++ RTS reassigns objects to Pes to balance load
  - But how does the RTS decide?
    - Multiple strategy options
    - E.g. Just move objects away from overloaded processors to underloaded processors
  - How is load determined?

Measurement Based Load Balancing

Measurement Based Load Balancing

- Principle of Persistence
  - Object communication patterns and computational loads tend to persist over time
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
  - Recent past is a good predictor of near future

Measurement Based Load Balancing

- Principle of Persistence
  - Object communication patterns and computational loads tend to persist over time
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
  - Recent past is a good predictor of near future
- Runtime instrumentation
  - Measures communication volume and computation time

Measurement Based Load Balancing

- Principle of Persistence
  - Object communication patterns and computational loads tend to persist over time
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
  - Recent past is a good predictor of near future
- Runtime instrumentation
  - Measures communication volume and computation time
- Measurement-based load balancers
  - Measure load information for chares
  - Periodically use the instrumented database to make new decisions and migrate objects
  - Many alternative strategies can use the database

Using the Load Balancer

Using the Load Balancer

- Link a LB module
  - `-module <strategy>`
  - RefineLB, NeighborLB, GreedyCommLB, others
  - EveryLB will include all load balancing strategies

Using the Load Balancer

- Link a LB module
  - `-module <strategy>`
  - RefineLB, NeighborLB, GreedyCommLB, others
  - EveryLB will include all load balancing strategies
- Compile time option (specify default balancer)
  - `-balancer RefineLB`

Using the Load Balancer

- Link a LB module
  - -module <strategy>
  - RefineLB, NeighborLB, GreedyCommLB, others
  - EveryLB will include all load balancing strategies
- Compile time option (specify default balancer)
  - -balancer RefineLB
- Runtime option (override default)
  - +balancer RefineLB

Instrumentation

Instrumentation

- By default, instrumentation is enabled
  - Automatically collects load information

Instrumentation

- By default, instrumentation is enabled
  - Automatically collects load information
- Sometimes, you want LB decisions to be based only on a portion of your program
  - To disable by default, provide runtime argument `+LBOff`
  - To toggle instrumentation in code, use `LBTurnInstrumentOn()` and `LBTurnInstrumentOff()`

Code to Use Load Balancing

Code to Use Load Balancing

- Set `usesAtSync = true;` in chare constructor

Code to Use Load Balancing

- Set `usesAtSync = true;` in chare constructor
- Insert `AtSync()` call at a natural barrier
  - Call from every chare in all collections
  - Does not block

Code to Use Load Balancing

- Set `usesAtSync = true;` in chare constructor
- Insert `AtSync()` call at a natural barrier
  - Call from every chare in all collections
  - Does not block
- Implement `ResumeFromSync()` to resume execution
  - A typical `ResumeFromSync()` contributes to a reduction

Example: Stencil

```
// Synchronize at every iteration: Main starts next iteration
void Main::endIter(err) { if (err < T) CkExit();
                                     else stencilProxy.sendBoundaries(); }

// Assume a 1D Stencil chare array with near neighbor communication
void Stencil::sendBoundaries() {
  thisProxy(wrap(x-1)).updateGhost(RIGHT, left_ghost);
  thisProxy(wrap(x+1)).updateGhost(LEFT, right_ghost);
}

void Stencil::updateGhost(int dir, double ghost) {
  updateBoundary(dir, ghost);
  if (++remoteCount == 2) {
      remoteCount = 0;
      doWork(); } }
```

Example: Stencil cont.

```
void Stencil::doWork() {
  e = (computeKernel() < DELTA);

         contribute(8, e, CkCallback(CkReductionTarget(Main, endIter), mainProxy));
}
```

Example: Stencil cont.

```
void Stencil::doWork() {
  e = (computeKernel() < DELTA);
  if (++i % 10 == 0) { AtSync(); } // Allow load balancing every 10 iterations
  else { contribute(8, e, CkCallback(CkReductionTarget(Main, endIter), mainProxy)); }
}
```

Example: Stencil cont.

```
void Stencil::doWork() {
  e = (computeKernel() < DELTA);
  if (++i % 10 == 0) { AtSync(); } // Allow load balancing every 10 iterations
  else { contribute(8, e, CkCallback(CkReductionTarget(Main, endIter), mainProxy)); }
}

void Stencil::ResumeFromSync() {
  contribute(CkCallback(CkReductionTarget(Main, endIter), mainProxy));
}
```

Serialization and PUP

Serialization and PUP

- How can the RTS move arbitrary objects across nodes?

Serialization and PUP

- How can the RTS move arbitrary objects across nodes?
- Charm++ has a framework for serializing data called PUP

Serialization and PUP

- How can the RTS move arbitrary objects across nodes?
- Charm++ has a framework for serializing data called PUP
- PUP: Pack and Unpack

Serialization and PUP

- How can the RTS move arbitrary objects across nodes?
- Charm++ has a framework for serializing data called PUP
- PUP: Pack and Unpack
- With PUP, chares become serializable and can be transported to memory, disk, or another processor

Simple PUP for a Simple Chare

```
class MyChare :
public Cbase_MyChare {
  int a;
  float b;
  char c;
  double localArray[LOCAL_SIZE];
};
```

```
void pup(PUP::er &p) {

  p | a;
  p | b;
  p | c;
  p(localArray, LOCAL_SIZE);
}
```

Writing an Advanced PUP Routine

```
class MyChare : public Cbase_MyChare {
  int heapArraySize;
  float* heapArray;
  MyClass* pointer;
};
```

Writing an Advanced PUP Routine

```
class MyChare : public Cbase_MyChare {
  int heapArraySize;
  float* heapArray;
  MyClass* pointer;
};
```

```
void pup(PUP::er &p) {
  p | headArraySize;
  if (p.isUnpacking()) {
    heapArray = new float[heapArraySize]; }
  p(heapArray, heapArraySize);
  bool isNull = !pointer;
  p | isNull;
  if (!isNull) {
    if (p.isUnpacking()) {
      pointer = new MyClass(); }
      p | *pointer; }}
```

PUP Uses

PUP Uses

- Moving objects for load balancing

PUP Uses

- Moving objects for load balancing
- Marshalling user defined data types
  - When using a type you define as a parameter for an entry method
  - Type has to be serialized to go over network, uses PUP for this
  - Can add PUP to any class, doesn’t have to be a chare

PUP Uses

- Moving objects for load balancing
- Marshalling user defined data types
  - When using a type you define as a parameter for an entry method
  - Type has to be serialized to go over network, uses PUP for this
  - Can add PUP to any class, doesn’t have to be a chare
- Serializing for storage

Split Execution: Checkpoint Restart

Split Execution: Checkpoint Restart

- Can use to stop execution and resume later
  - The job runs for 5 hours, then will continue in new allocation another day!

Split Execution: Checkpoint Restart

- Can use to stop execution and resume later
  - The job runs for 5 hours, then will continue in new allocation another day!
- We can use PUP for this!

Split Execution: Checkpoint Restart

- Can use to stop execution and resume later
  - The job runs for 5 hours, then will continue in new allocation another day!
- We can use PUP for this!
- Instead of migrating to another PE, just “migrate” to disk

How to Enable Split Execution

How to Enable Split Execution

- Call to checkpoint the application is made in the main chare at a synchronization point

How to Enable Split Execution

- Call to checkpoint the application is made in the main chare at a synchronization point
- `log_path` is file system path for checkpoint

How to Enable Split Execution

- Call to checkpoint the application is made in the main chare at a synchronization point
- `log_path` is file system path for checkpoint
- `Callback cb` called when checkpoint (or restart) is done
  - For restart, user needs to provide argument `+restart` and path of checkpoint file at runtime

```
CkCallback cb (CkIndex_Hello:SayHi(), helloProxy);
CkStartCheckpoint("log_path", cb);

shell> ./charmrun hello +p4 +restart log_path
```

Chares Are Reactive

Chares Are Reactive

- The way we described Charm++ so far, a chare is a reactive entity:
  - If it gets this method invocation, it does this action,
  - If it gets that method invocation then it does that action
  - But what does it do?
  - In typical programs, chares have a life-cycle

Chares Are Reactive

- The way we described Charm++ so far, a chare is a reactive entity:
  - If it gets this method invocation, it does this action,
  - If it gets that method invocation then it does that action
  - But what does it do?
  - In typical programs, chares have a life-cycle
- How to express the life-cycle of a chare in code?
  - Only when it exists
    - i.e. some chares may be truly reactive, and the programmer does not know the life cycle
  - But when it exists, its form is:
    - Computations depend on remote method invocations, and completion of other local computations
    - A DAG (Directed Acyclic Graph)!

Structured Dagger

The serial construct

- The serial construct
  - A sequential block of C++ code in the .ci file
  - The keyword `serial` means that the code block will be executed without interruption/preemption, like an entry method
  - Syntax: `serial <optionalString> { /* C++ code */ }`
  - The `<optionalString>` is used for identifying the serial for performance analysis
  - Serial blocks can access all members of the class they belong to
- Examples (.ci file):

```
entry void method1(parameters) {
    serial {
        thisProxy.invokeMethod(10);
        callSomeFunction();
    }
};
```

```
entry void method2(parameters) {
    serial “setValue” {
        value = 10;
    }
};
```

Structured Dagger

The `when` construct

- The `when` construct
  - Declare the actions to perform when a message is received
  - In sequence, it acts like a blocking receive

```
entry void someMethod() {
  when entryMethod1(parameters) { /* block2 */ }
  when entryMethod2(parameters) { /* block3 */ }
};
```

Structured Dagger

The when construct: waiting for multiple invocations

- Execute `SDAG_CODE` when `method1` and `method2` arrive

```
    when method1(int param1, int param2),
         method2(bool param3)
       SDAG_CODE
```

- Which is semantically the same as this:

```
    when myMethod1(int param1, int param2) {
        when myMethod2(bool param3) { }
          SDAG_CODE
    }
```

Structured Dagger

The when construct : reference number matching

- The when clause can wait on a certain reference number
- If a reference number is specified for a when , the first parameter for the when must be the reference number
- Semantics: the when will “block” until a message arrives with that reference number

```
when method1[100](int ref, bool param1)
     /* sdag block */

serial {
    proxy.method1(200, false); /* will not be delivered to the when */
    proxy.method1(100, true); /* will be delivered to the when */
}
```

Structured Dagger

Other constructs

- if-then-else
  - Same as the typical C if-then-else semantics and syntax
- for
  - Defines a sequenced *`for`* loop (like a sequential C for loop)
- while
  - Defines a sequenced *`while`* loop (like a sequential C while loop)
- forall
  - Has “do-all” semantics: iterations may execute in any order
- overlap
  - Allows multiple independent constructs to execute in any order

http://charm.cs.illinois.edu/manuals/html/charm++/5.html

Interoperability and Within Node Parallelism

- 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,
- Charm++ interoperates with MPI
  - So, some modules can be written in Charm++, rest in MPI

Control flow within chare

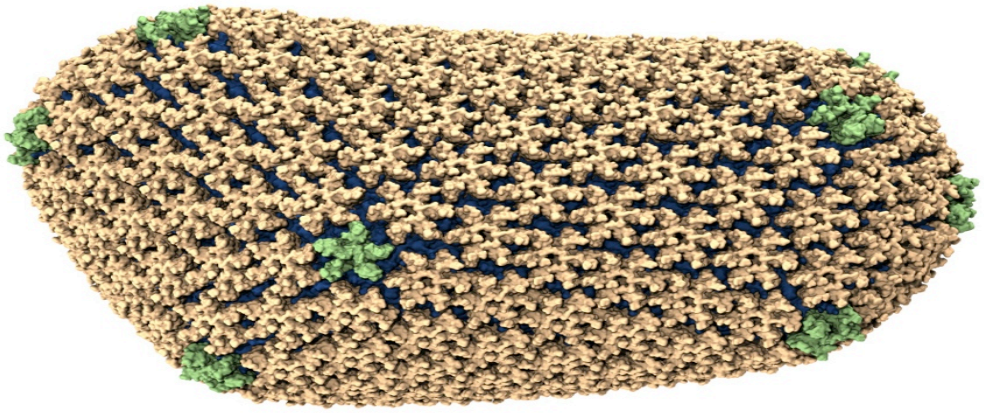
- Structured dagger notation
  - Provides a script-like language for expressing dag of dependencies between method invocations and computations
- Threaded Entry methods
  - Allows entry methods to block without blocking the PE
  - Supports futures, and
  - ability to suspend/resume threads

Advanced Concepts

- Priorities
- Entry method tags
- Quiescence detection
- LiveViz: visualization from a parallel program
- CharmDebug: a powerful 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

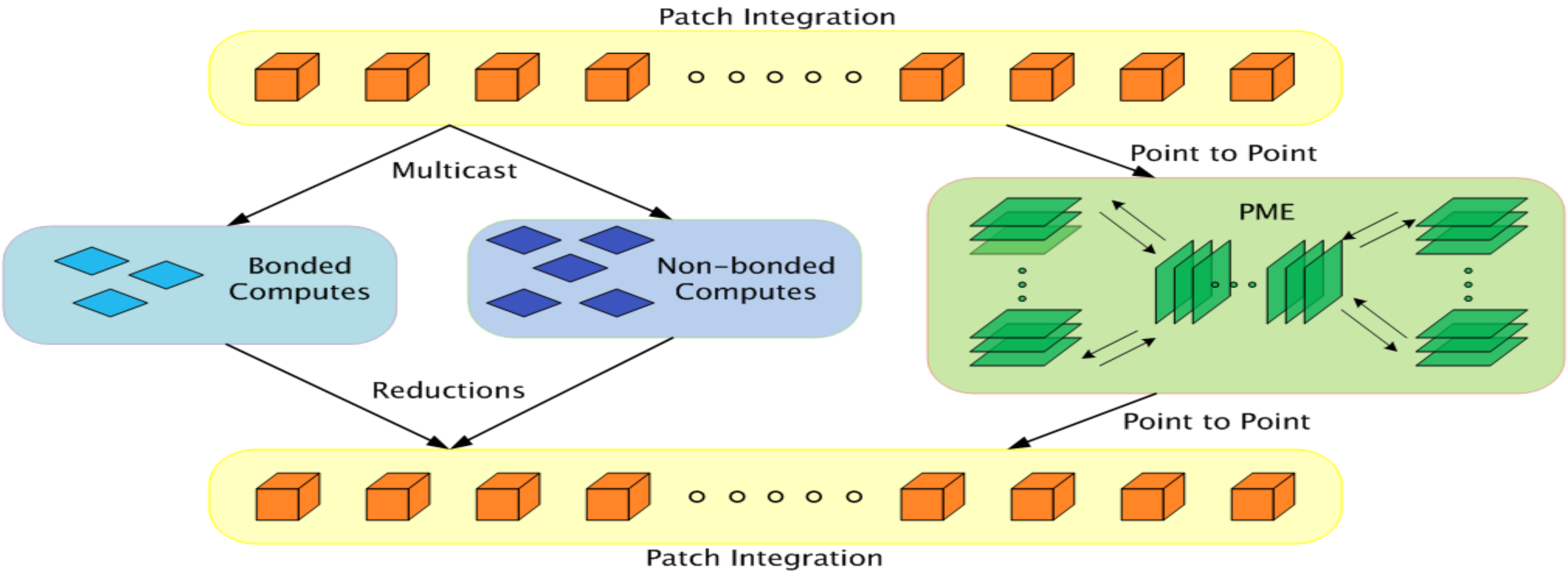
NAMD: Biomolecular Simulations

- 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

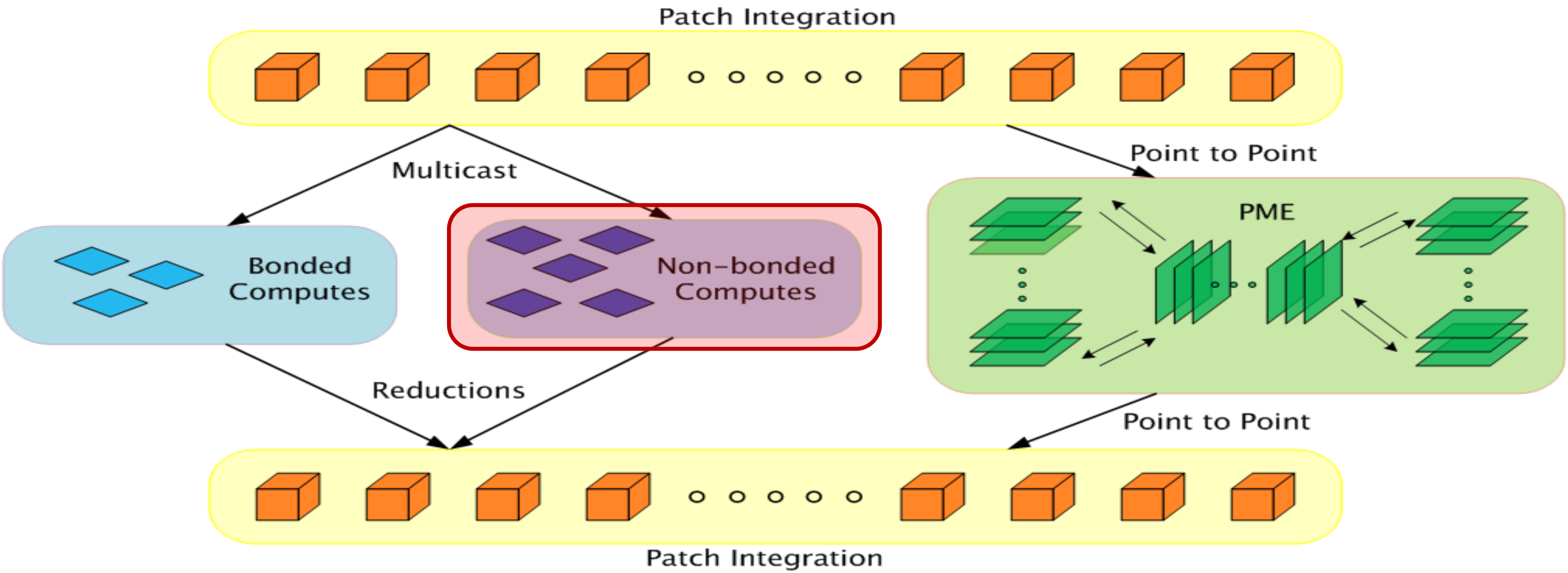


Determination of the structure of HIV capsid by researchers including Prof Schulten

Parallelization using Charm++



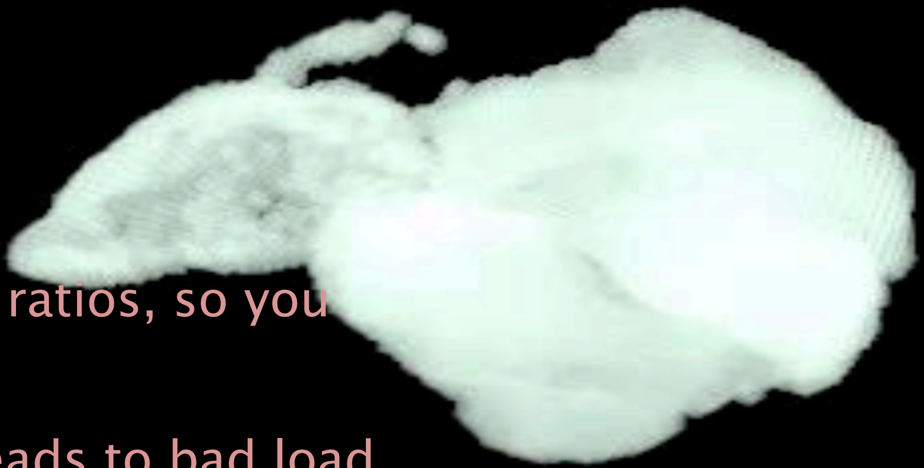
Parallelization using Charm++



ChaNGa: Parallel Gravity

Evolution of Universe and Galaxy Formation

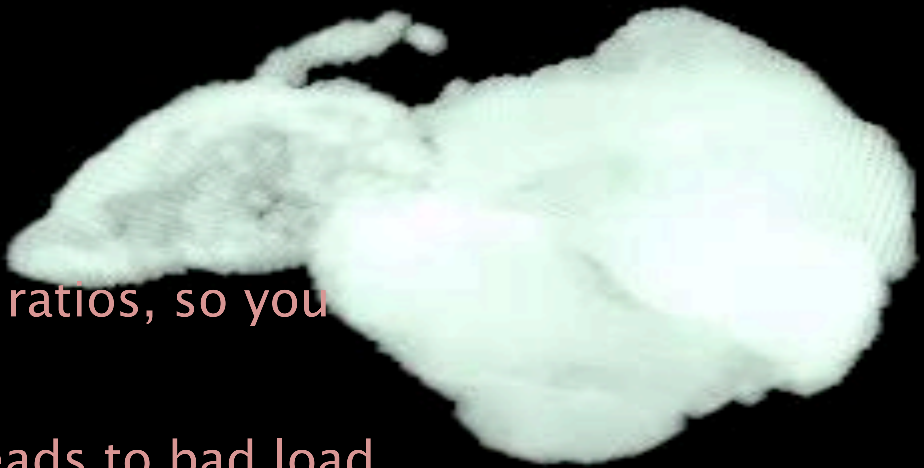
- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomp
  - Geometry has better aspect ratios, so you "open" up fewer nodes
  - But is not used because it leads to bad load balance
  - Assumption: one-to-one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



ChaNGa: Parallel Gravity

Evolution of Universe and Galaxy Formation

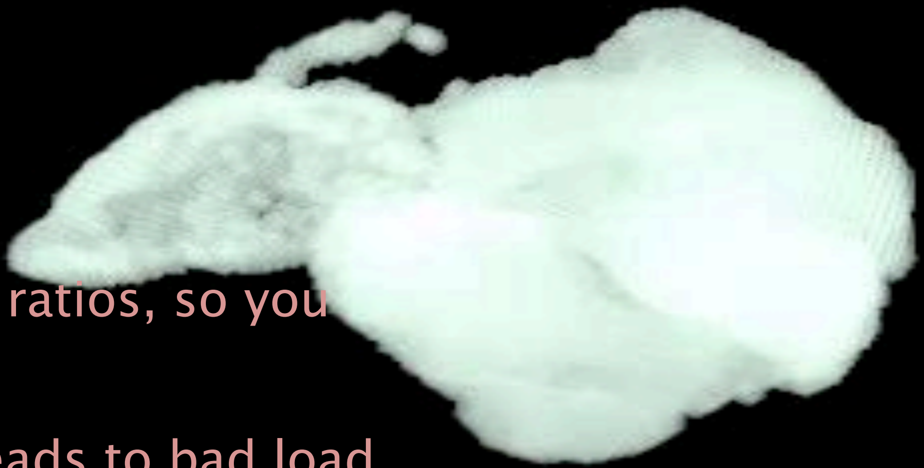
- Collaborative project (NSF)
  - with Tom Quinn, Univ. of Washington
- Gravity, gas dynamics
- Barnes-Hut tree codes
  - Oct tree is natural decomp
  - Geometry has better aspect ratios, so you "open" up fewer nodes
  - But is not used because it leads to bad load balance
  - Assumption: one-to-one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



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

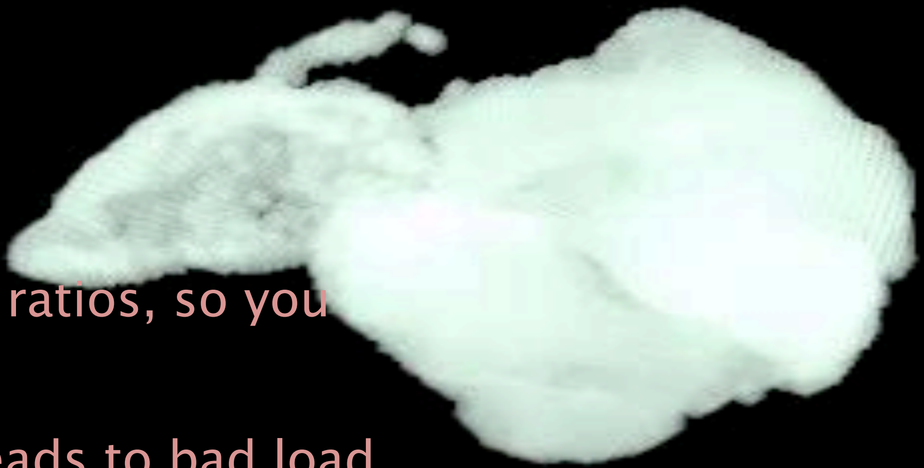


With Charm++: Use Oct-Tree, and let Charm++ map subtrees to processors

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

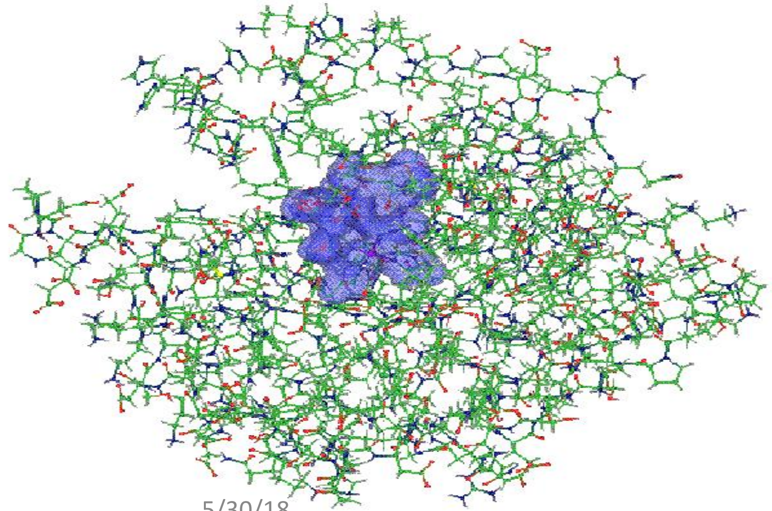
OpenAtom: On the fly ab initio molecular dynamics on the ground state surface with instantaneous GW-BSE level spectra

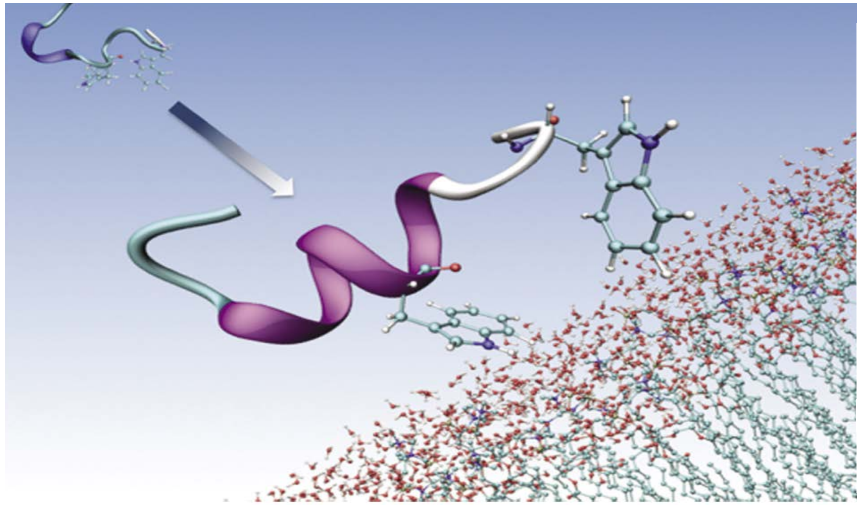
PIs: G.J. Martyna, IBM; S. Ismail-Beigi, Yale; L. Kale, UIUC;

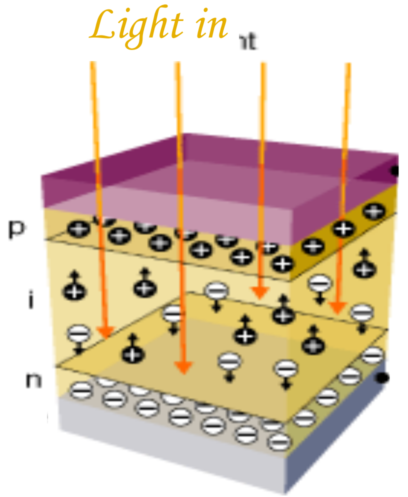
Team: Q. Li, IBM, M. Kim, Yale; S. Mandal, Yale;

E. Bohm, UIUC; N. Jain, UIUC; M. Robson, UIUC;

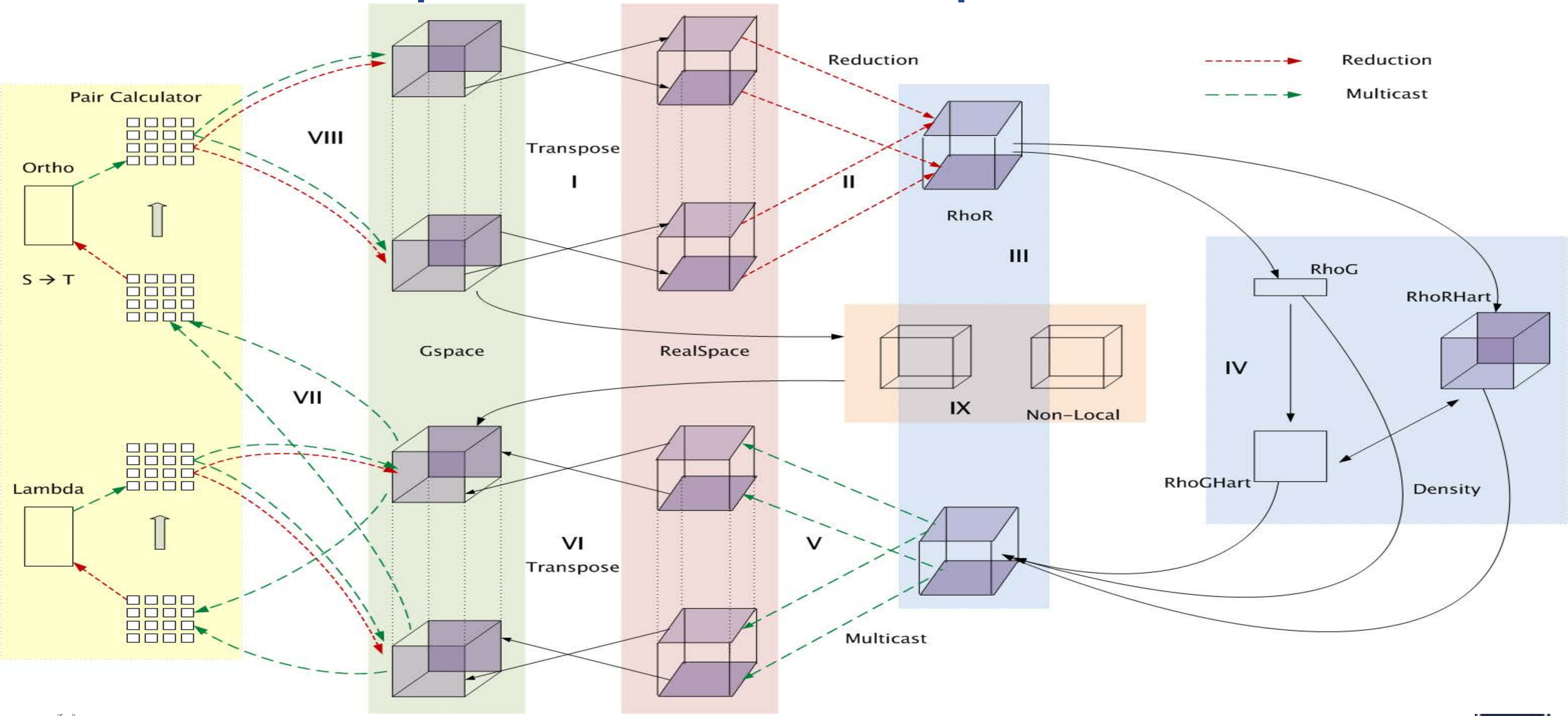
E. Mikida, UIUC; P. Jindal, UIUC; T. Wicky, UIUC.







Decomposition and Computation Flow



MiniApps

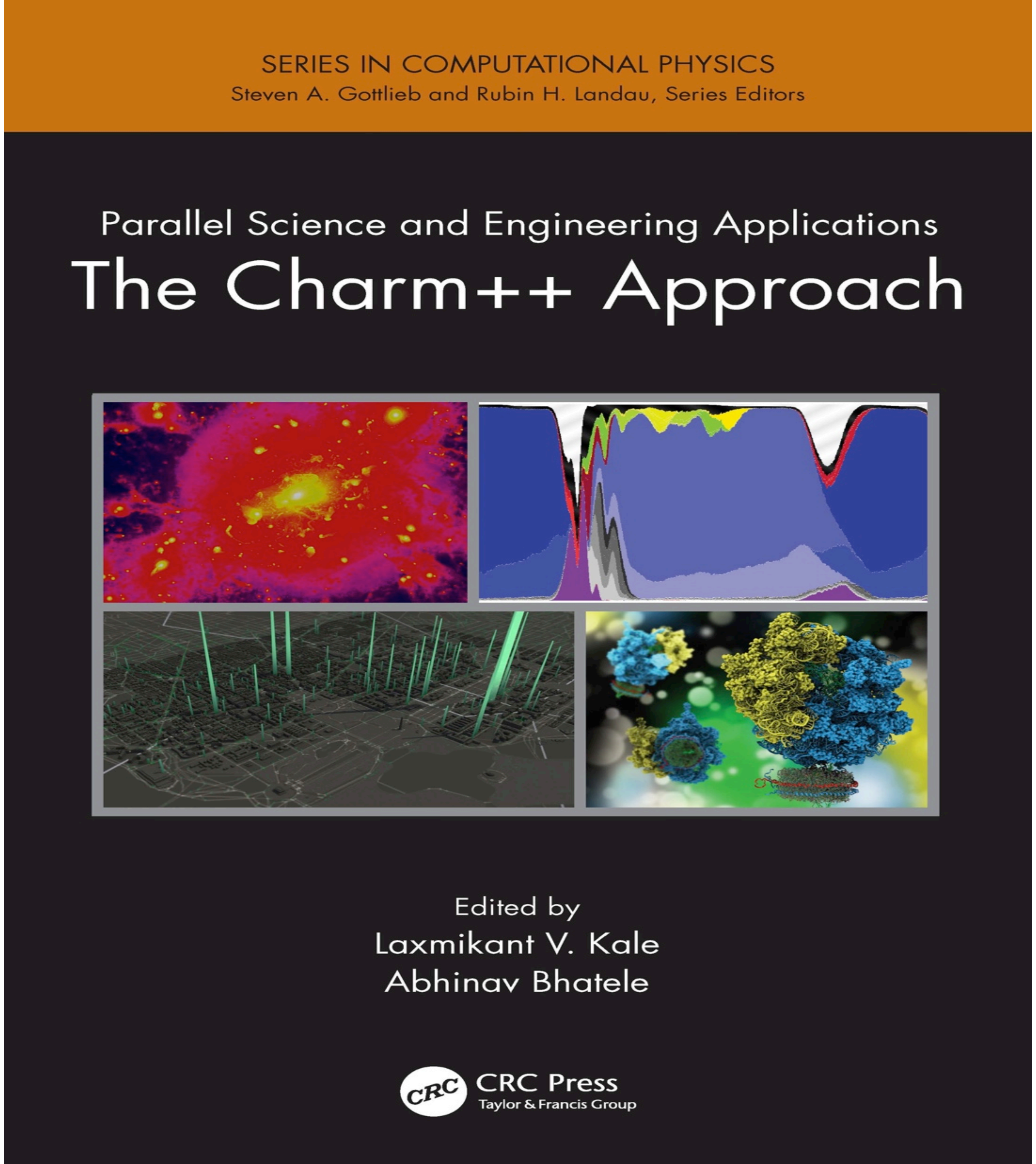
Available at: http://charmplusplus.org/miniApps/

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

Saving Cooling Energy

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

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!

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

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

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