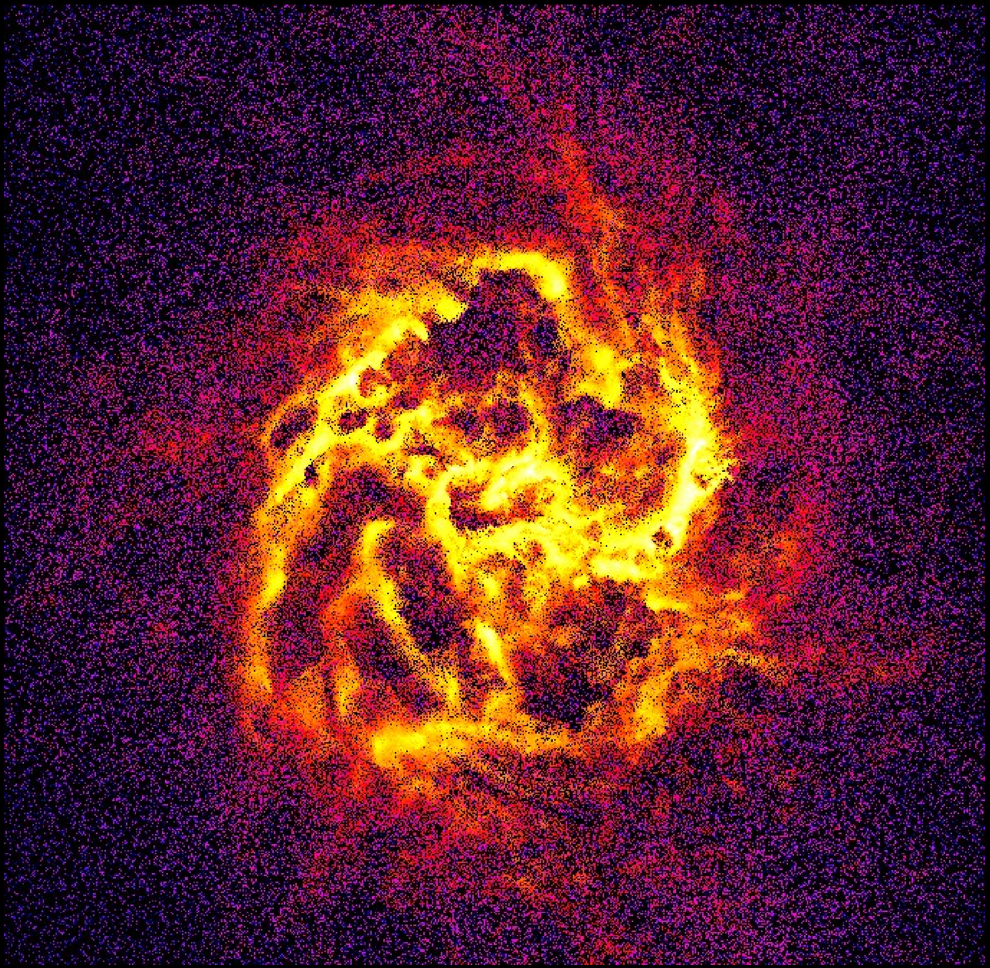
ChaNGa



CHArm N-body GrAvity

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

Laxmikant Kale
Lukasz Wesolowski
Harshitha Menon
Pritish Jetley
Gengbin Zheng
Celso Mendes
Filippo Gioachin
Amit Sharma
Edgar Solomonik
Orion Lawlor

Thomas Quinn
Graeme Lufkin
Joachim Stadel
James Wadsley
Greg Stinson

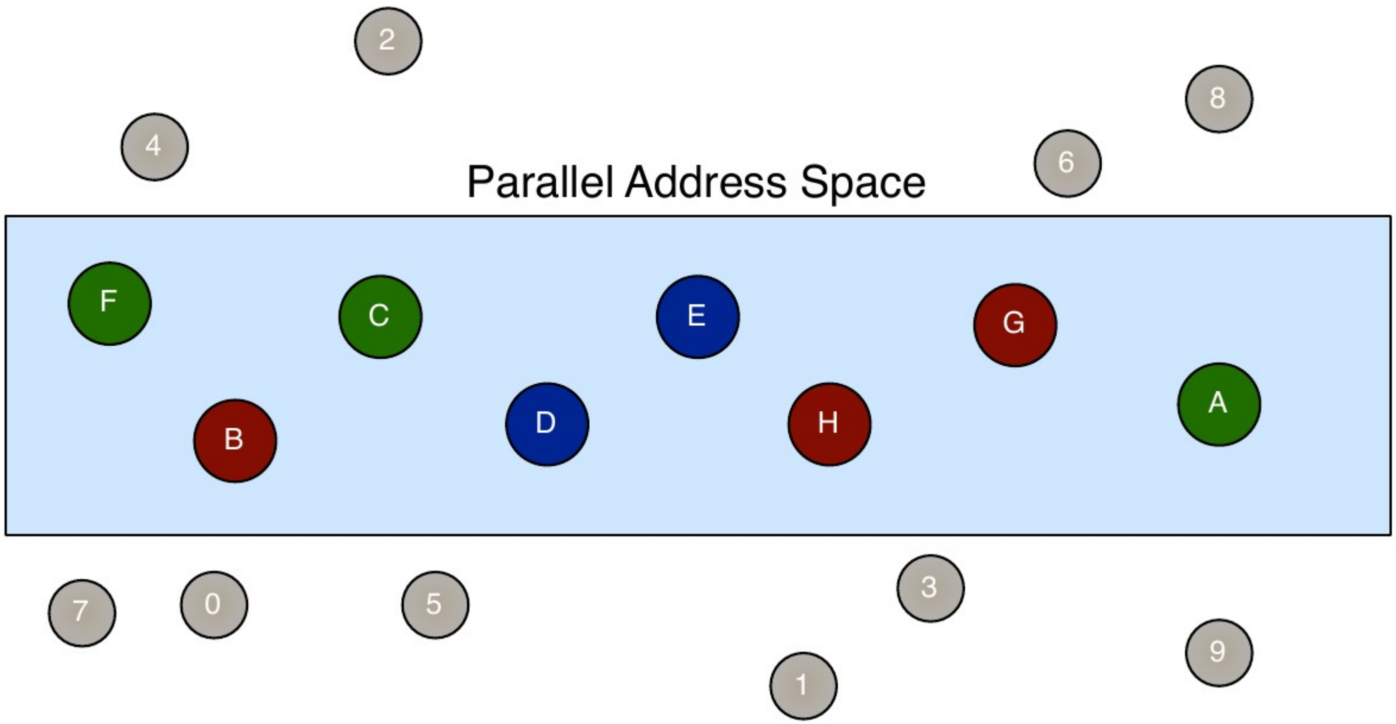
Outline

- Basic concepts of Charm++
  - Tutorial by S. Kale and the UIUC PPL group
- Charm++ paradigms: Chare arrays, method invocations, broadcasts, reductions
- Architecture of ChaNGa.
- Compiling and running Charm++ programs

Charm++

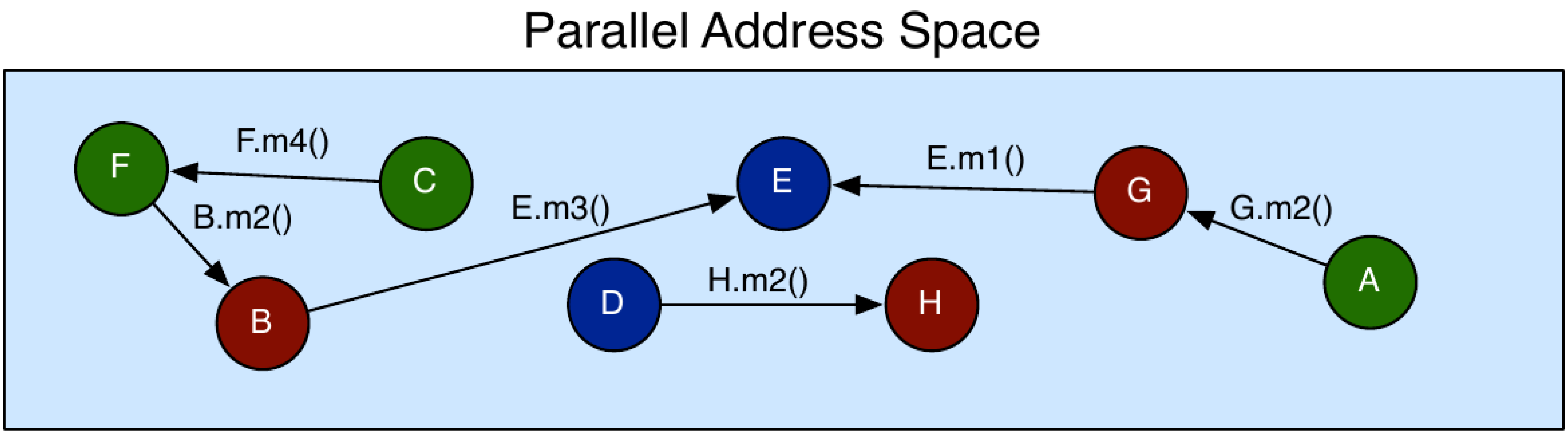
- C++-based parallel runtime system
  - Composed of a set of globally-visible parallel objects that interact
  - The objects interact by asynchronously invoking methods on each other
- Charm++ runtime
  - Manages the parallel objects and (re)maps them to processes
  - Provides scheduling, load balancing, and a host of other features, requiring little user intervention

Globally Visible Objects



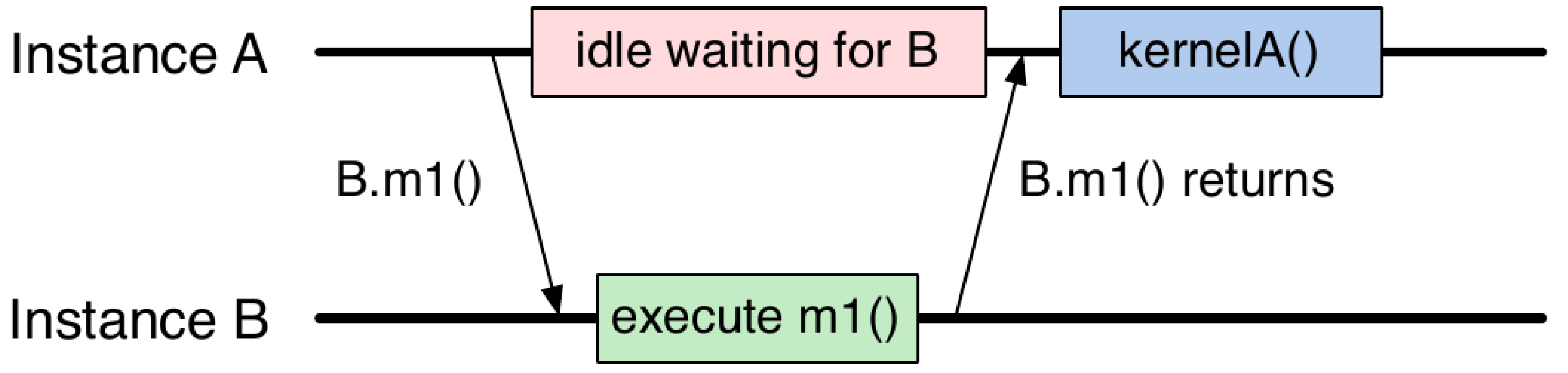
- Certain “special” object instances are
  - First-class citizens in the parallel address space
  - With unique location-independent names
- Under the hood, the runtime handles locality and provides mechanisms to promote objects to the parallel space

Globally-Visible Methods



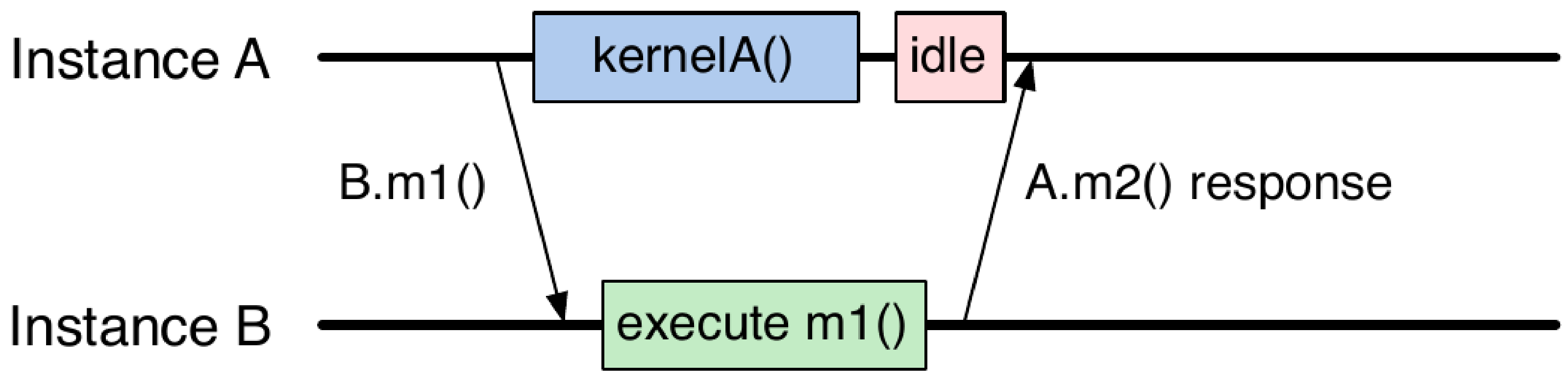
- How can objects communicate across address spaces?
  - Just like a sequential language: use object reference to invoke a method
  - Location independent handle
  - Method invocation becomes a communication

Method-Driven Asynchronous Communication



- What happens if an object waits for a return value from a method invocation?
  - Performance
  - Latency

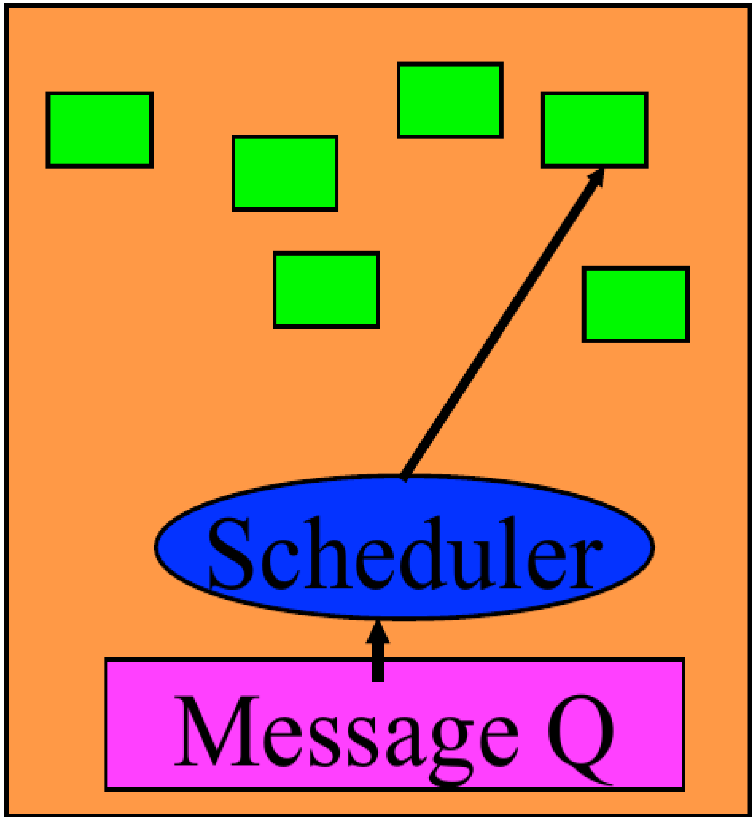
Design Principle: Do not wait for remote completion



- Hence, method invocations should be asynchronous
    - No return values
- Computations are driven by the incoming data
    - Initiated by the sender or method caller

The Execution Model

- Several objects live on a single PE
  - I.e. Core or processor
- As a result
  - Method invocations directed at objects on that processor will have to be stored in a pool,
  - And a user-level scheduler will select one invocation from the queue and runs it to completion
  - A PE is the entity that has one scheduler instance associated with it.



Message-driven Execution

- Execution is triggered by availability of a “message” (method invocation)
- When an entry method executes
  - It may generate messages for other objects
  - The RTS deposits them in the message Q on the target processor

Migratability

- Once the programmer has written the code without reference to processors, all of the communication is expressed among objects
- The system is free to migrate the objects across processors as and when it pleases
    - It must ensure it can deliver method invocations to the objects, where ever they go
    - This migratability turns out to be a key attribute for empowering an adaptive runtime system

Load balancing

- Static
  - Requires accurate cost model
- Dynamic
  - Demanded by adaptive algorithms
  - Work needs to be migrated with, e.g., particles
- Migratable Objects a natural solution
  - Coupled with a load measurement infrastructure

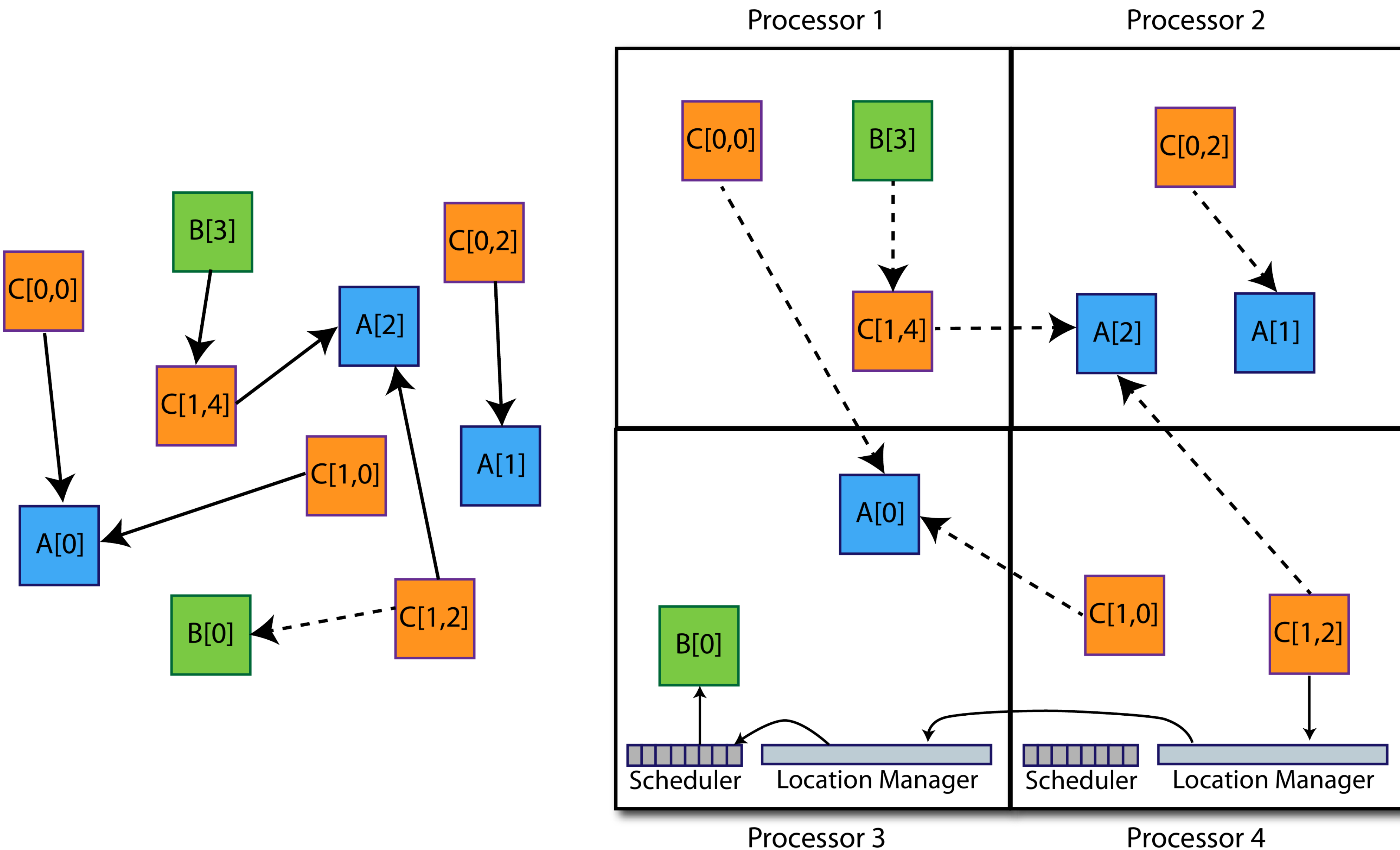
Collections of Objects

- "Chare arrays"
  - Structured: 1D, 2D, ..., 6D
  - Unstructured: anything hashable
  - Dense or Sparse
  - Static – all created at once
  - Dynamic – elements come and go
- Shadow arrays: share data but allow separate flow

Collections of Objects: Communication

- Point-to-point: to one element of a collection
- Broadcast: message to whole collection
- Multicast: message to subset of collection
- Reductions: message from (part of) collection
- Runtime system provides efficient delivery for all

Collections of Objects: user and machine view



Groups and NodeGroups

- Non-migratable chare array
- One element per PE (Group) or SMP node (NodeGroup)
- Share data among objects on a PE or node
  - E.g. Cooling table
- NodeGroups can have races.

Charm Array: Hello Example

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

Charm Array: Hello Example

```
#include ”arr.decl.h”
/*readonly*/ int arraySize;
struct Main : CBase Main {
   Main(CkArgMsg* msg) {
     arraySize = atoi(msg−>argv[1]);
     CProxy hello p = CProxy hello::ckNew(arraySize);
     p[0].printHello();
     }
};
struct hello : CBase hello {
   hello() { }
   hello(CkMigrateMessage*) { }
   void printHello() {
      CkPrintf(”%d: hello from %d\n”, CkMyPe(), thisIndex);
      if (thisIndex == arraySize − 1) CkExit();
      else thisProxy[thisIndex + 1].printHello();
      }
};
#include ”arr.def.h”
```

Migration: packing/unpacking data

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

```
void pup(PUP::er &p) {
  Cbase_MyChare::pup(p);
  p | a;
  p | b;
  p | c;
  p(localArray,
LOCAL_SIZE);
}
```

Measurement Based Load Balancing

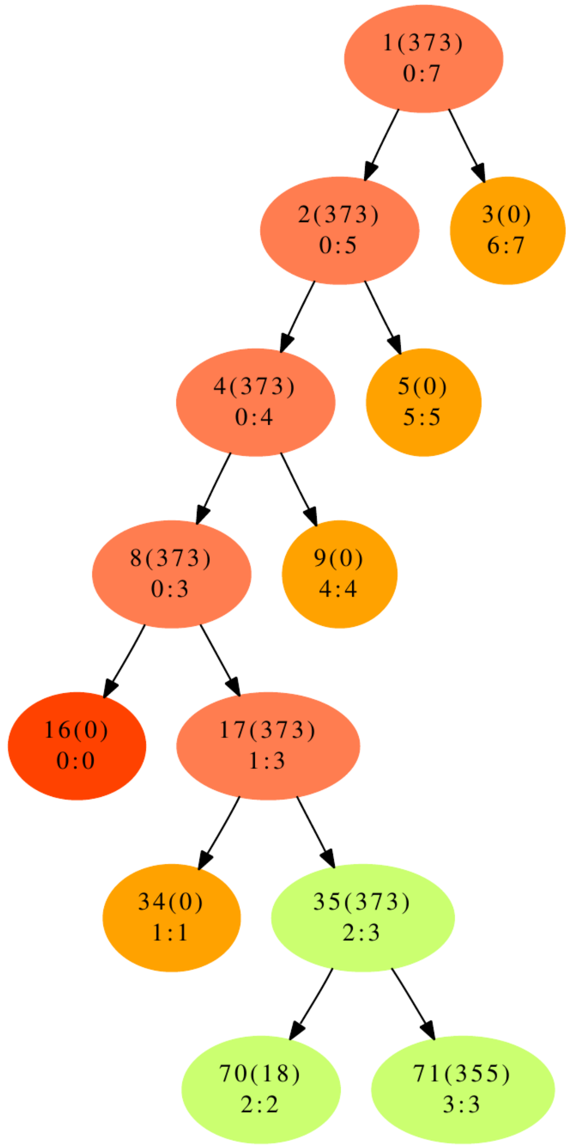
- Principle of Persistence
  - Object communication patterns and computational loads tend to persist over time
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
- Runtime instrumentation
  - Measures communication volume and computation time
- Measurement based load balancers
  - Use the instrumented data-base periodically to make new decisions
  - Many alternative strategies can use the database

ChaNGa Features

- N-body/SPH solver
- Very latency tolerant
- SMP aware
- Dynamic load balancing with choice of strategies
- Checkpointing (via migration to disk)
- Visualization

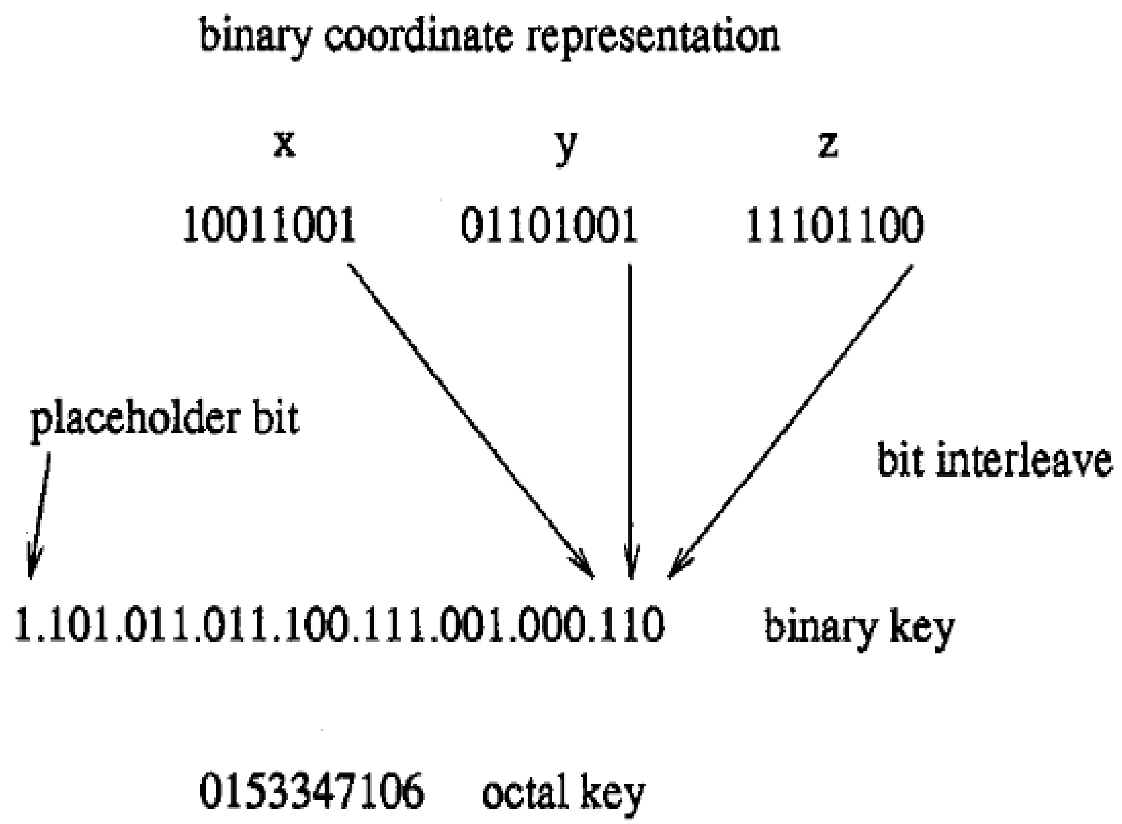
TreePiece: basic data structure

- A “vertical slice” of the tree, all the way to the root.
- Nodes are either:
  - Internal
  - External
  - Boundary (shared)

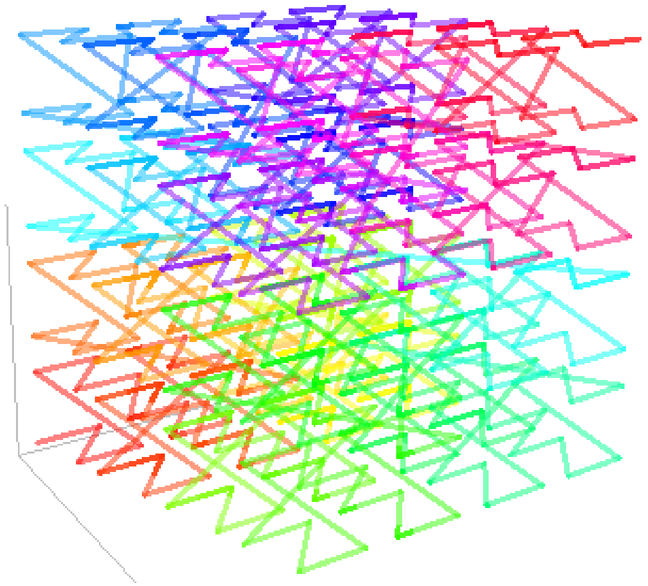


Domain Decomposition

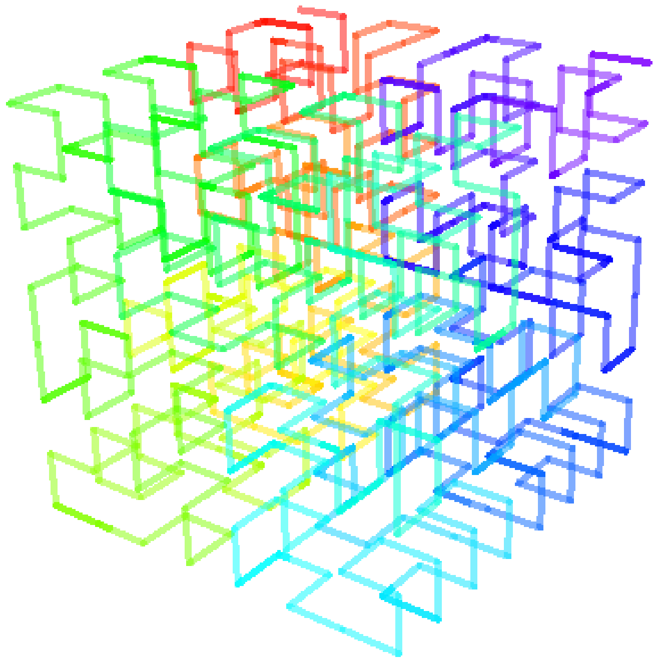
- Particles are identified by “Keys” (Warren & Salmon, 1993)
- Keys also define domains
- Decomposition is a “sort”.



Domain Decomposition Options



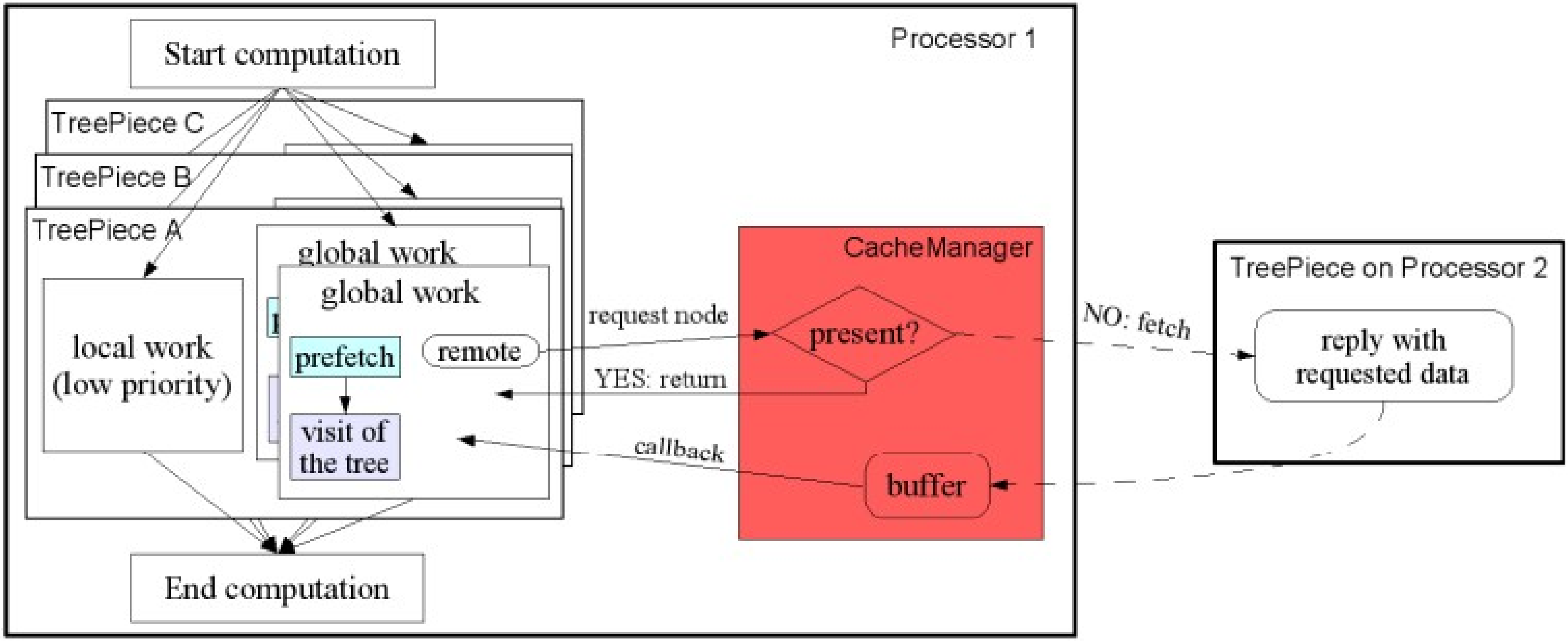
- Space-filling curves
  - Morton ordering
  - Peano-Hilbert
- “Oct”: fully contained nodes
  - Less communication
  - Harder load balancing
- ORB (orthogonal recursive bisection)
  - Poor gravity



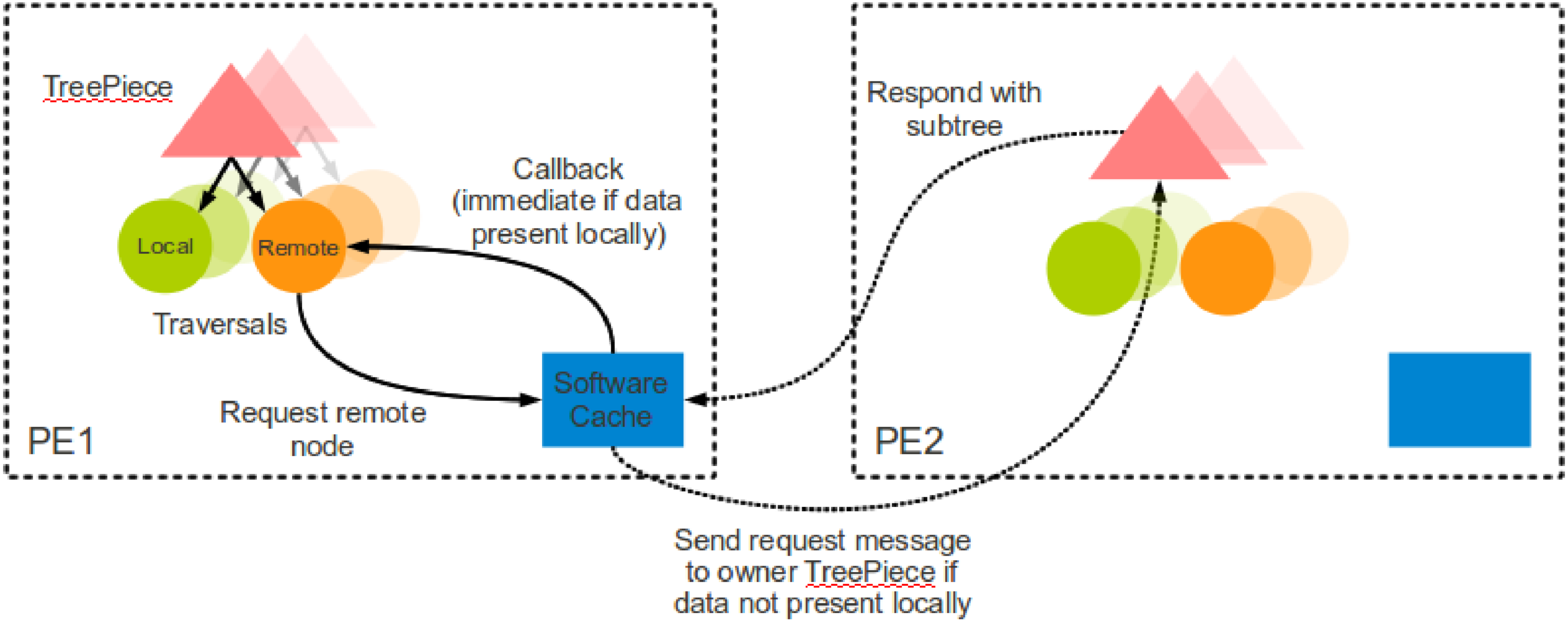
Tree Building

- Sort on Keys: particles are in tree order
- Determine count of particles in each Node
- Assign NodeKey: each bit a left-right branch
- Stop at “buckets”: each leaf contains a few particles.
- Construct multipole moments
  - Request moments of External Nodes
- Merge pieces on same address space.

Overall treewalk structure



Cache control flow



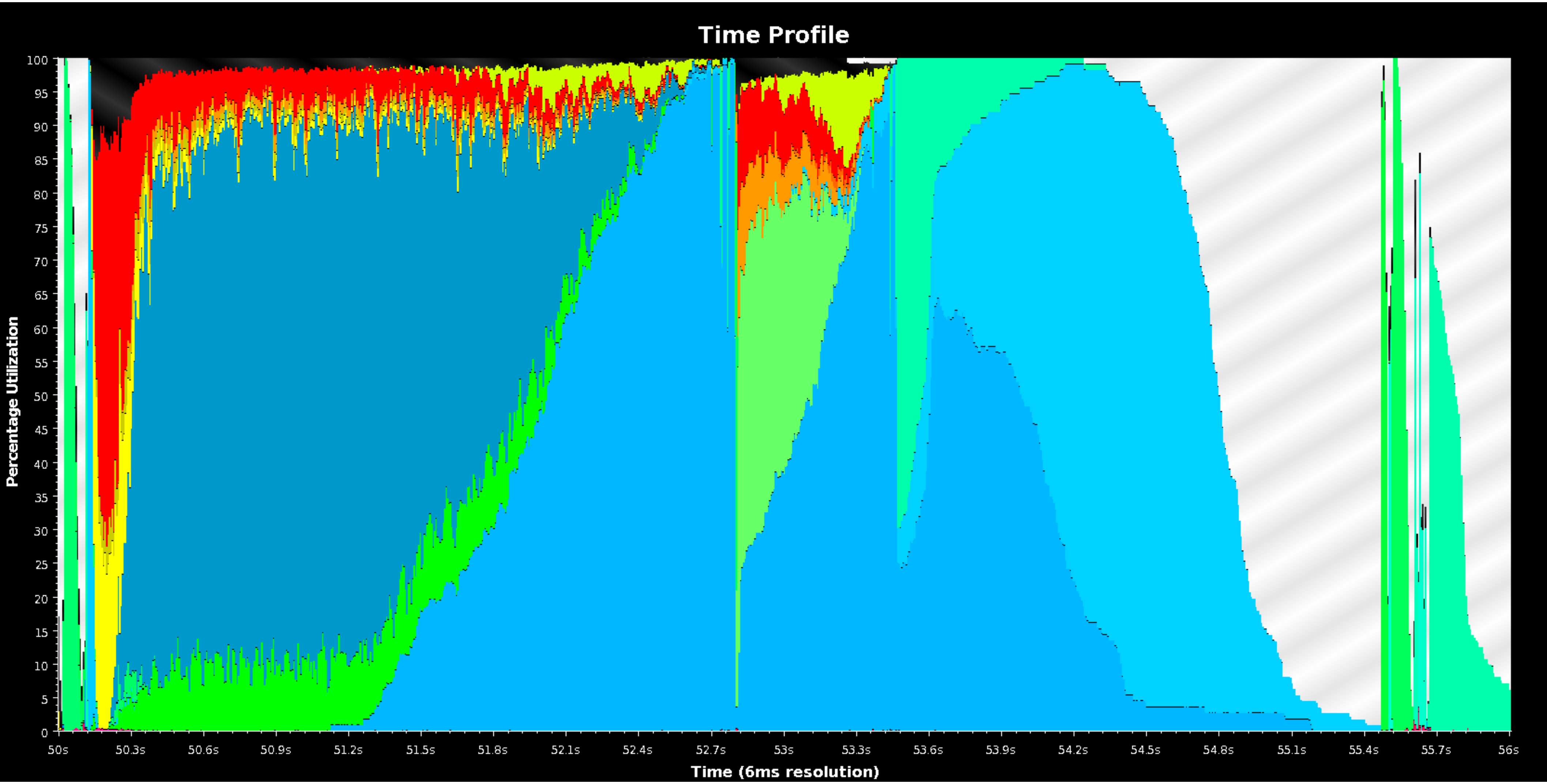
SPH Walks

- Two phases: density then pressure
- Symmetric forces => Cached data is written back to home piece.
- Multistepping: still need density of neighbors and particles for which I am a neighbor
  - Inverse neighbor search

Latency hiding strategies

- Multiple “treepieces” per core
- Division into multiple work units (*all concurrently*)
  - Off processor gravity treewalk
  - SPH treewalk
  - Local gravity treewalk
  - Ewald summation
- Method prioritization
  - Data requests get high priority

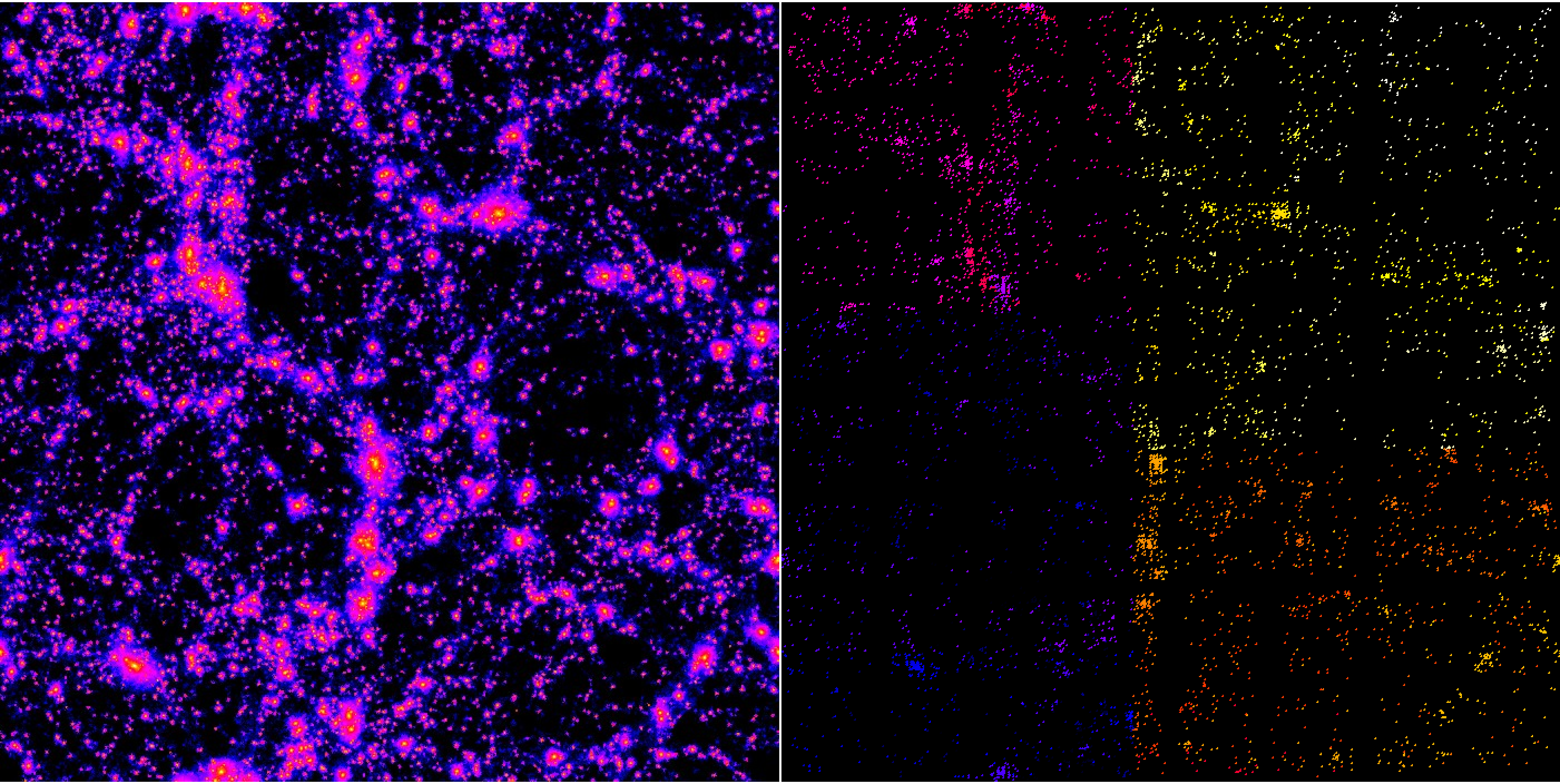
Overlap of Phases



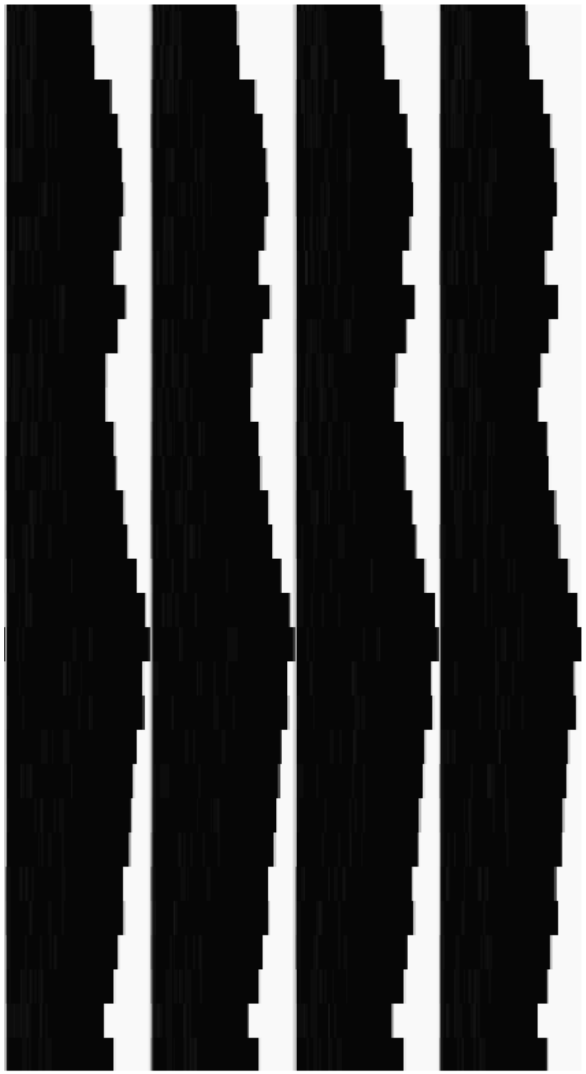
Load Balancing

- ORB load balancing:
  - Treepiece centroids sent to load balancer
  - Load balancer evenly divides work across x, y or z split.
  - Minimizes communication
- MultistepLB
  - Use load information from last timestep at current “rung”.

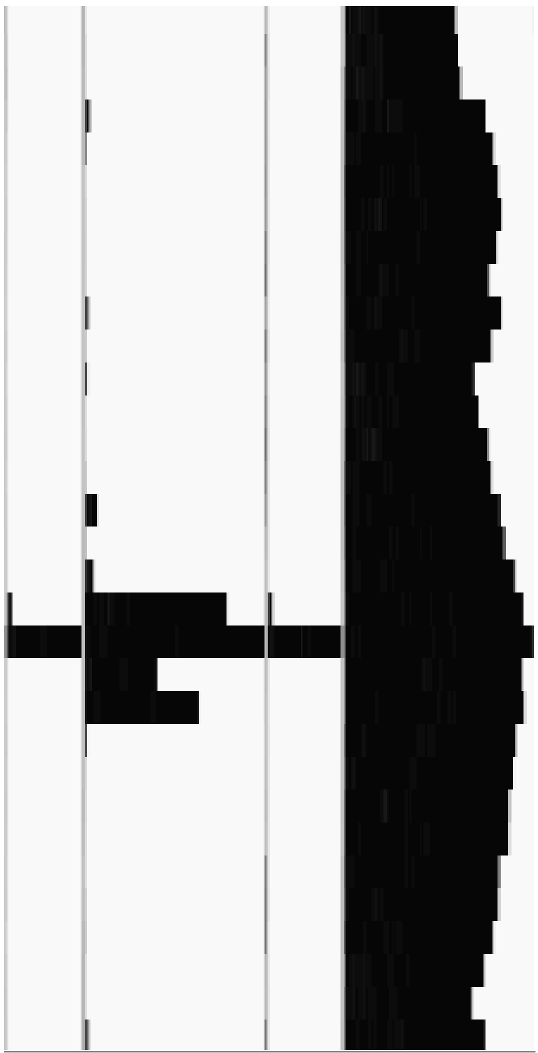
ORB3D Load Balancing



Multistepping: 3 rung example



613s



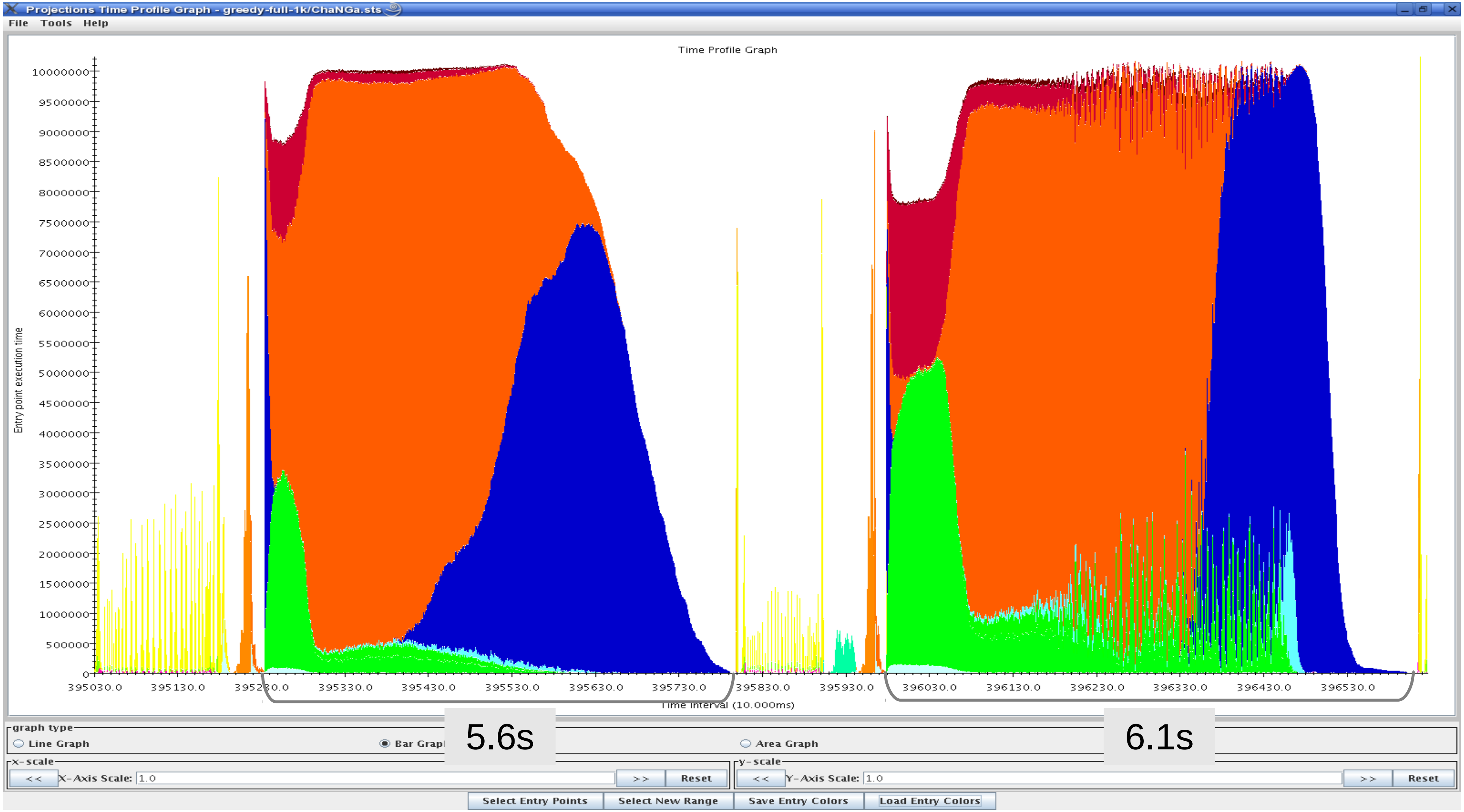
429s

228s

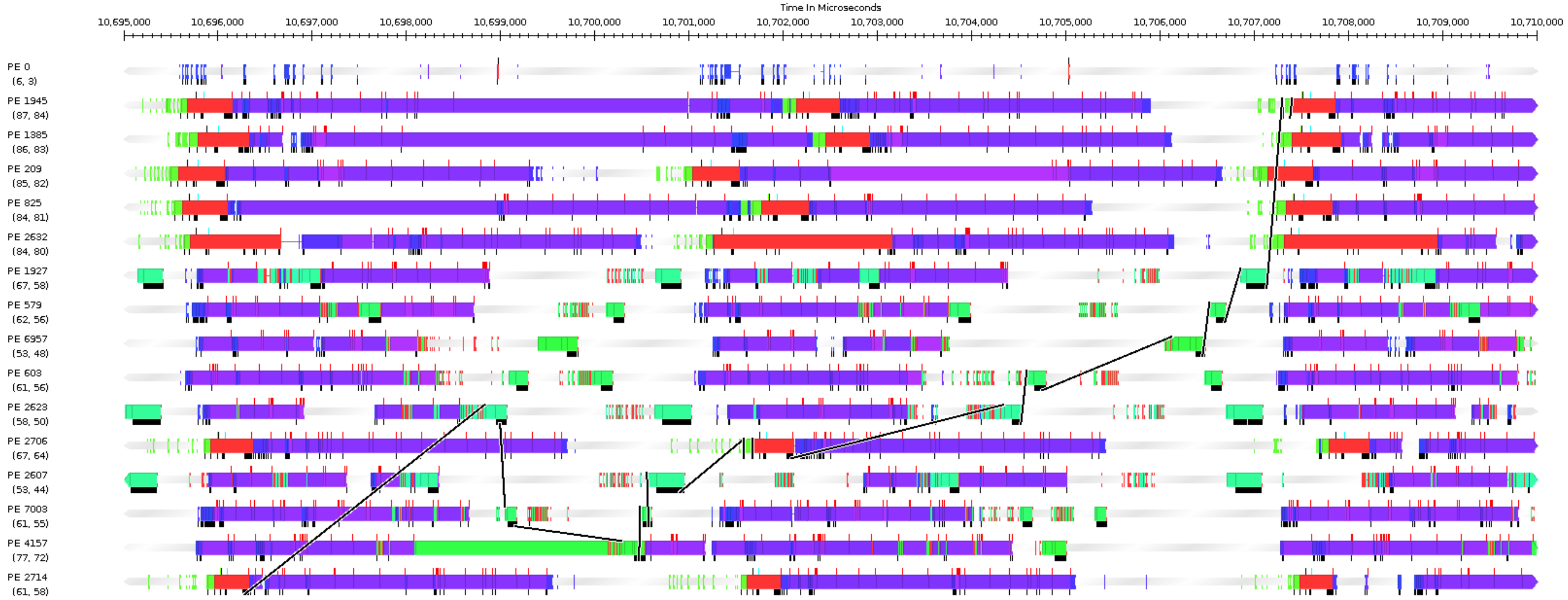
Performance Analysis Using Projections

- Instrumentation and measurement
  - Link program with -tracemode projections or summary
  - Trace data is generated automatically during run
  - User events can be easily inserted as needed
- Projections: visualization and analysis
  - Scalable tool to analyze up to 300,000 log files
  - A rich set of tool features : time profile, time lines, usage profile, histogram, extrema tool
  - Detect performance problems: load imbalance, grain size, communication bottleneck, etc

Projections example:
Testing load balancing on 1024 processors



Time Lines with Message Back Tracing



Charm++ features in ChaNGa

- Computation/communication overlap
- Entry method prioritization
- Flexible, customizable load balancing framework
- Composability
- Object Oriented: reuse of existing code.
- Porting to new architectures
  - Including GPGPUs

Availability

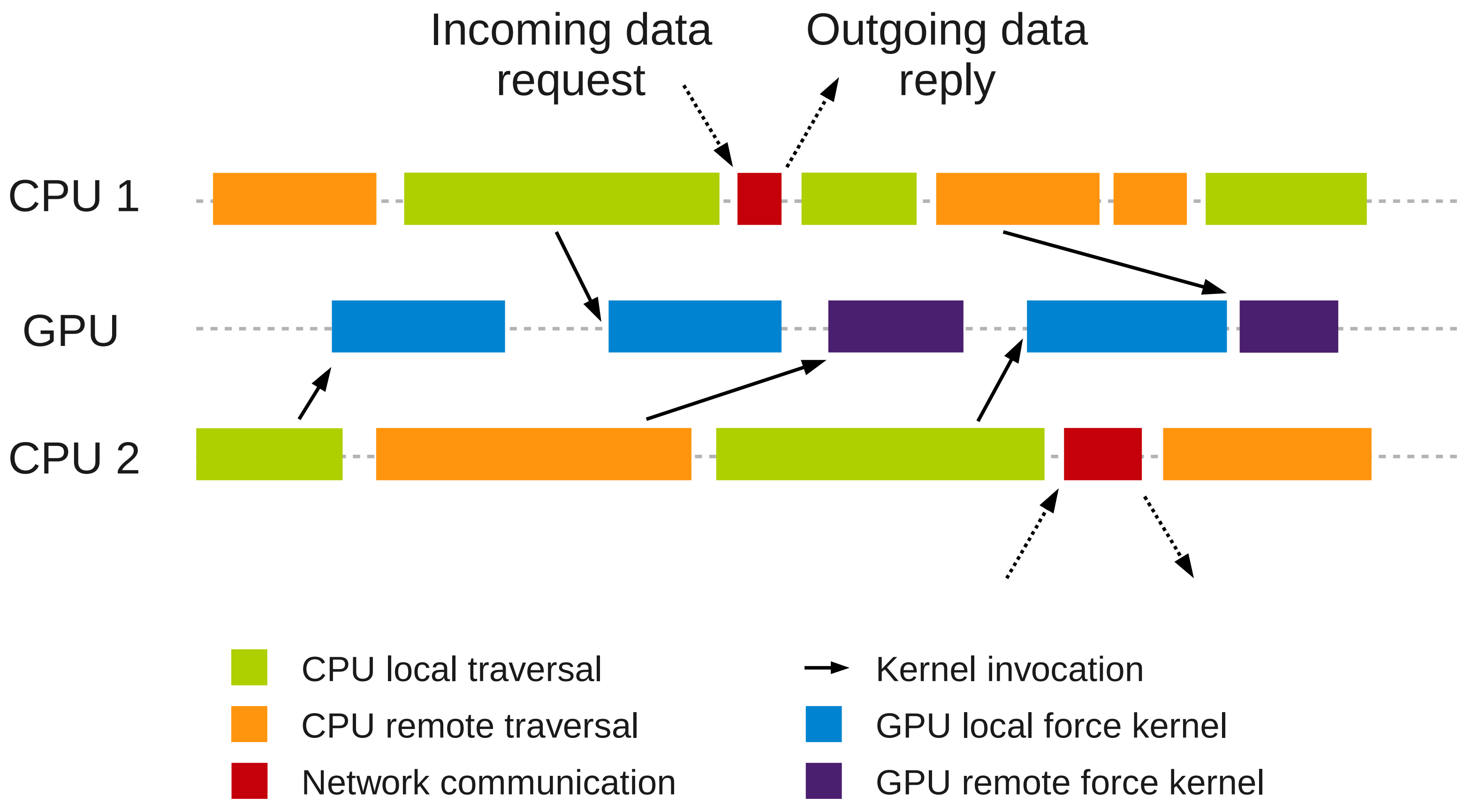
- Charm++: http://charm.cs.uiuc.edu
- ChaNGa download: http://software.astro.washington.edu/nchilada/
- Release information: http://hpcc.astro.washington.edu/tools/changa.html
- Mailing list: changa-users@u.washington.edu

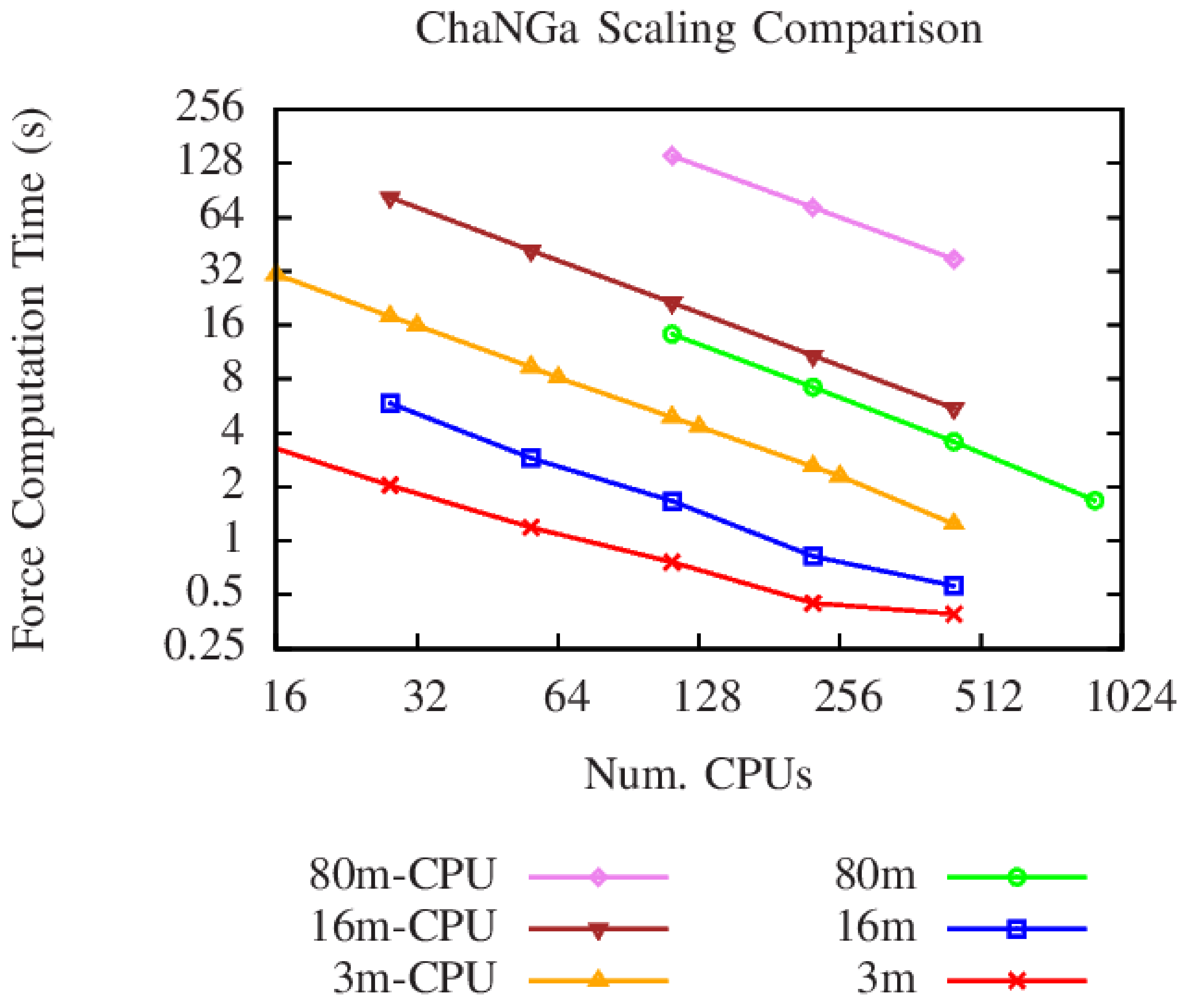
Acknowledgment: NSF-ITR, NSF-PRAC, NASA-AISR

GPU Manager

- User submits “work requests” with GPU kernel, associated buffers and callback
- System transfers memory between CPU and GPU, executes kernel, and returns via a callback
- GPU operations performed asynchronously
- Pipelined execution
- Consistent with Charm++ model
- Charm++ tools (profiler) available

GPU/CPU Timeline





ChaNGa Scaling Comparison
Force Computation Time (s)
256
128
64
32
16
8
4
2
1
0.5
0.25
16
32
64
128
256
512
1024
Num. CPUs
80m-CPU
16m-CPU
3m-CPU
80m
16m
3m