A Multi-Paradigm Approach to High-Performance Scientific Programming

Pritish Jetley
Parallel Programming Laboratory

What will the language of tomorrow look like?

- Language support for modularity
- Abstraction → Productivity

- Runtime assistance
- No sacrifice of performance

The future is now...

Charm++

PGAS (UPC, CAF, X10, Chapel, Fortress,...)

MPI/PGAS Hybrids

...or is it?

How abstract can languages be?

Can we reconcile program & language semantics?

Can we express algorithms naturally?

Our premise

- Productivity comes from abstractions
- Specialization of abstractions also yields better parallel performance
  - e.g. relaxed semantics in Global Arrays

Our approach

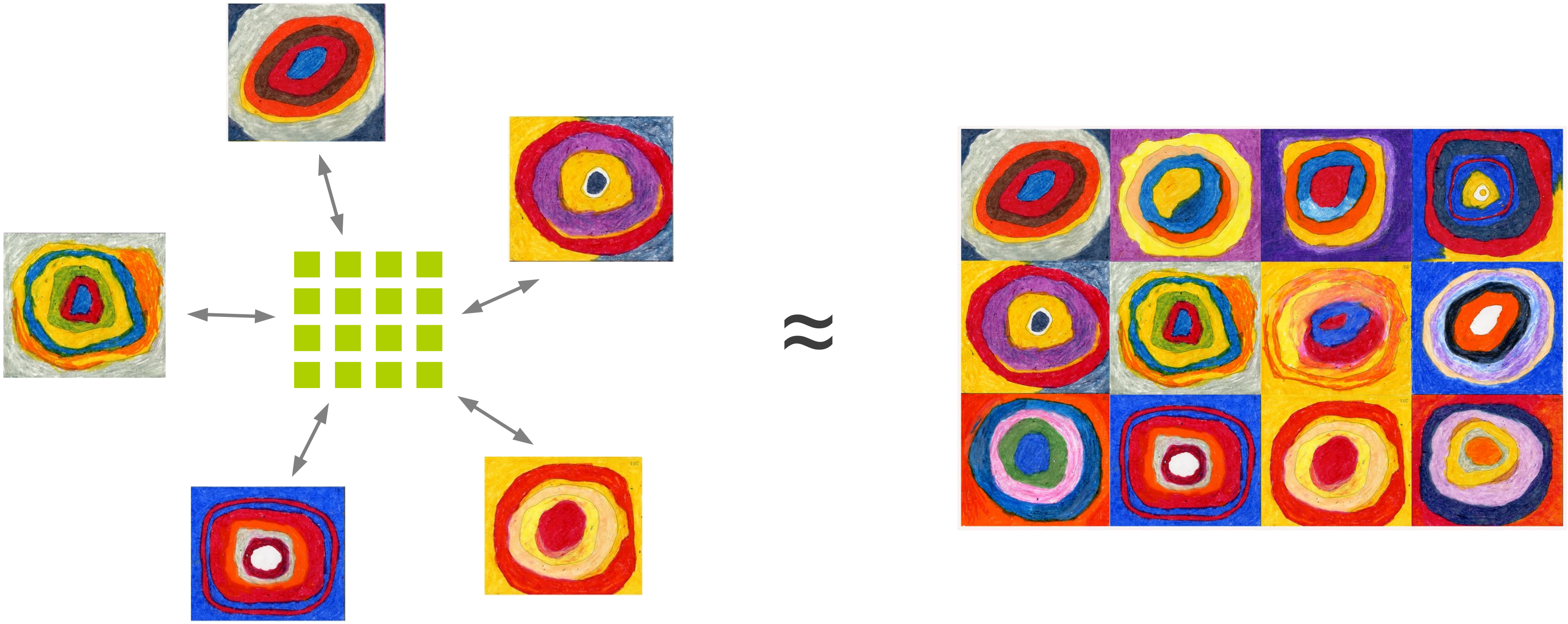
- Plurality
- Specialization
- Interoperability

Our agenda

Complete set of incomplete, interoperable languages

Abstract, specialized languages

Completeness through ***interoperation***



This talk

Productive message-driven programming (Charj)

Static data flow (Charisma)

Generative recursion (Divcon)

Tree-based algorithms (Distree)

Disciplined sharing of global data (MSA)

Productive message-driven programming

With ***Charj***

Charj

- **Char**m++/**J**ava = Charj
- Keep the good bits of Charm++:
  - Overdecomposition onto migratable objects
  - Message driven execution
  - Asynchrony
  - Intelligent runtime system (load balancing, message combination, etc.)
- But use a source-to-source compiler to address its drawbacks

Compiler intervention for productivity

- Automatically determines parallel interfaces

```
// foo.ci
entry void bar();

// foo.h
void bar();

// foo.cpp
void Foo::bar() {...}
```

```
// foo.cj
void bar();
```

Compiler intervention for productivity

- Automatically generate per-entry (de)serialization code

```
class Particle {
   Vec3 position, accel, vel;
   Real mass, charge;
}
```

```
class Compute {
   void pairwise(Array<Particle> first,
                 Array<Particle> second){
      // only uses Particle position, charge
   }
}
```

Compiler intervention for productivity

- Semantic checking and type safety

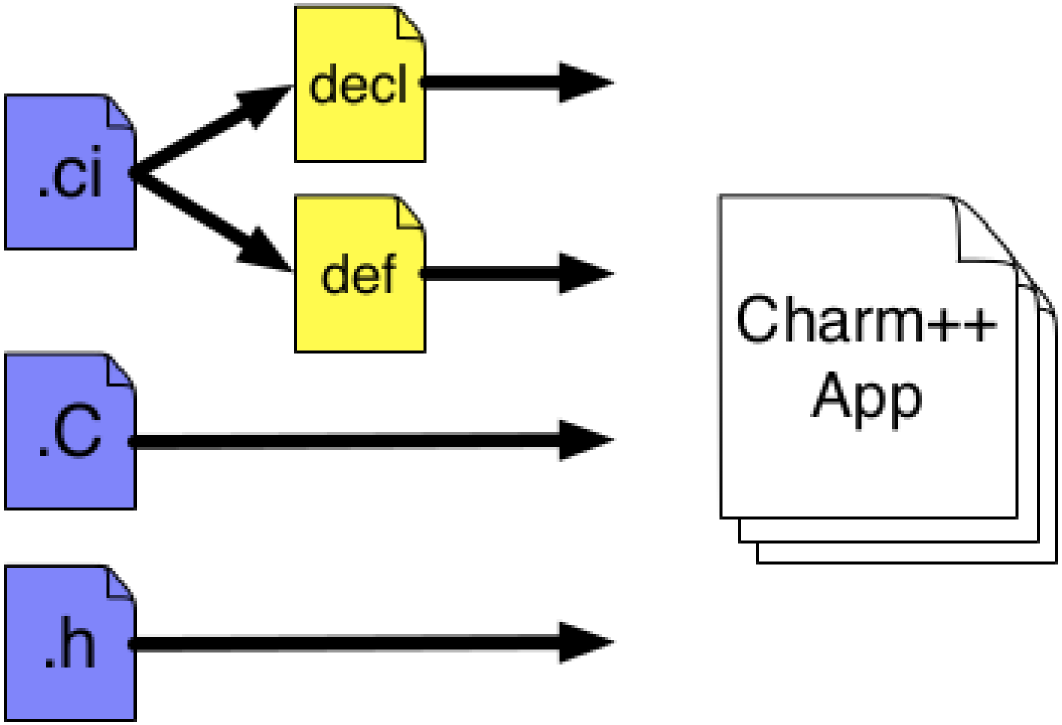
```
w.foo();  // “plain”: asynchronous
x.foo();  // local: preempts
y.foo();  // sync: blocks
z.foo();  // array: multiple invocations
```

```
// foo.ci
readonly int n;

// foo.cpp
int n;
…
n = 17; // bug (?)
```

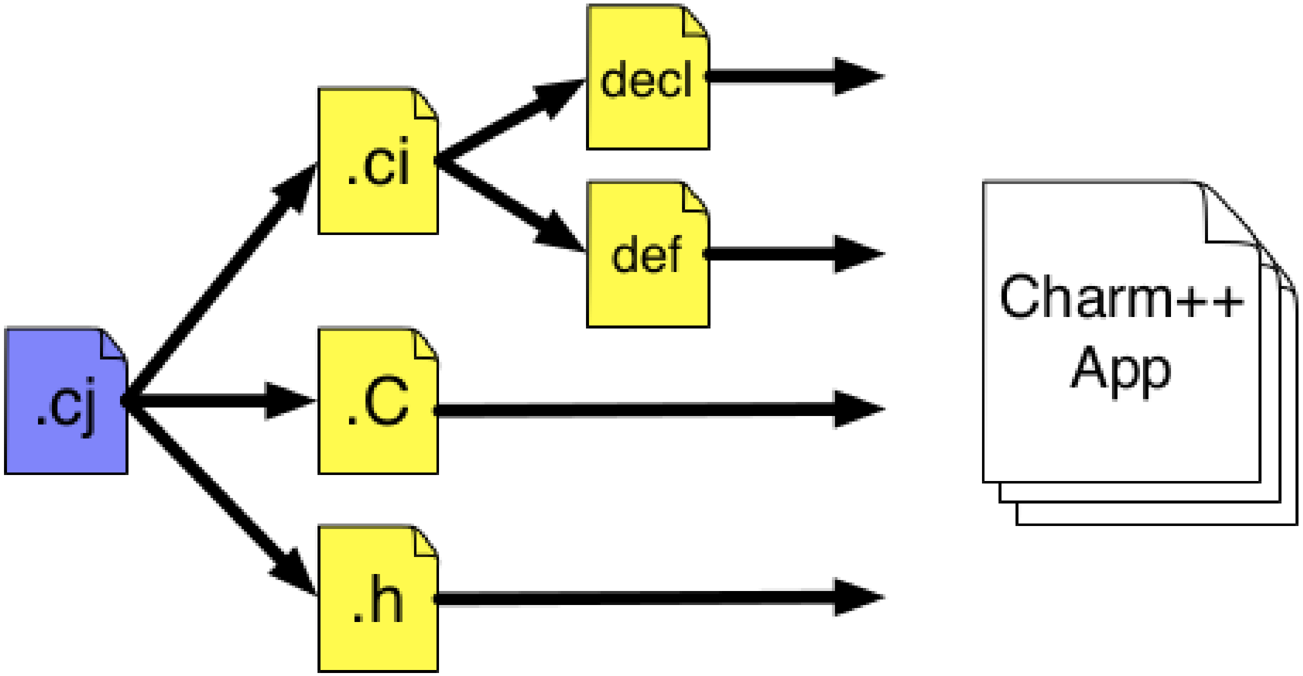
Compiler intervention for productivity

- Simple optimizations such as live variable analysis
  - Minimize checkpoint footprint
  - Find pertinent data to be offloaded to GPU

Charm++ workflow



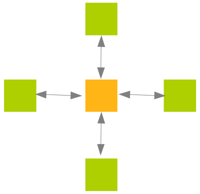
Charj workflow



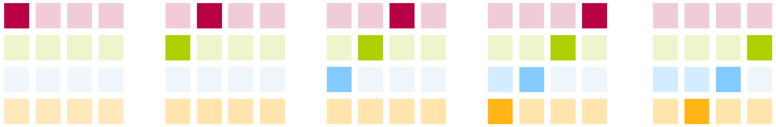
Static data flow programming with ***Charisma***

Expressive scope of Charisma

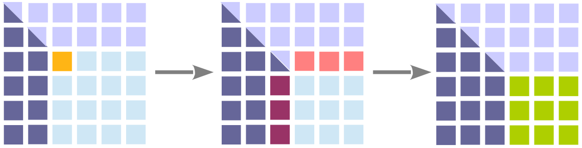
- Structured grid methods



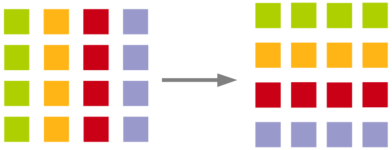
- Wavefront computations



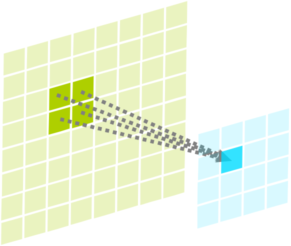
- Dense linear algebra



- Permutation



- MG



Charisma

- Salient features
  - Object-oriented
  - Programmer decomposes work
  - Global view of data and control
  - Publish-consume model for data dependencies
  - Separation of parallel structure & serial code
  - Compiled into message-driven Charm++ specification

A Charisma program *orchestrates* the interactions of ***collections of objects***

Indexed collections of objects

- Objects encapsulate *data* and *work*
  - Explicit specification of grain size and locality
  - Allows for adaptive overlap of comm./comp.
  - Load balancing, check pointing, etc.
- Unit of work is a method invocation

Objects communicate
by ***publishing*** and ***consuming***
values

Communication between objects

- Method invocations **publish, consume** values
- Publish-consume pattern → data dependencies
- Parsed by compiler to generate code

```
(p) ← obj1.foo();

obj2.bar(p);
```

Parallelism across objects
is specified via the
foreach construct

Object parallelism

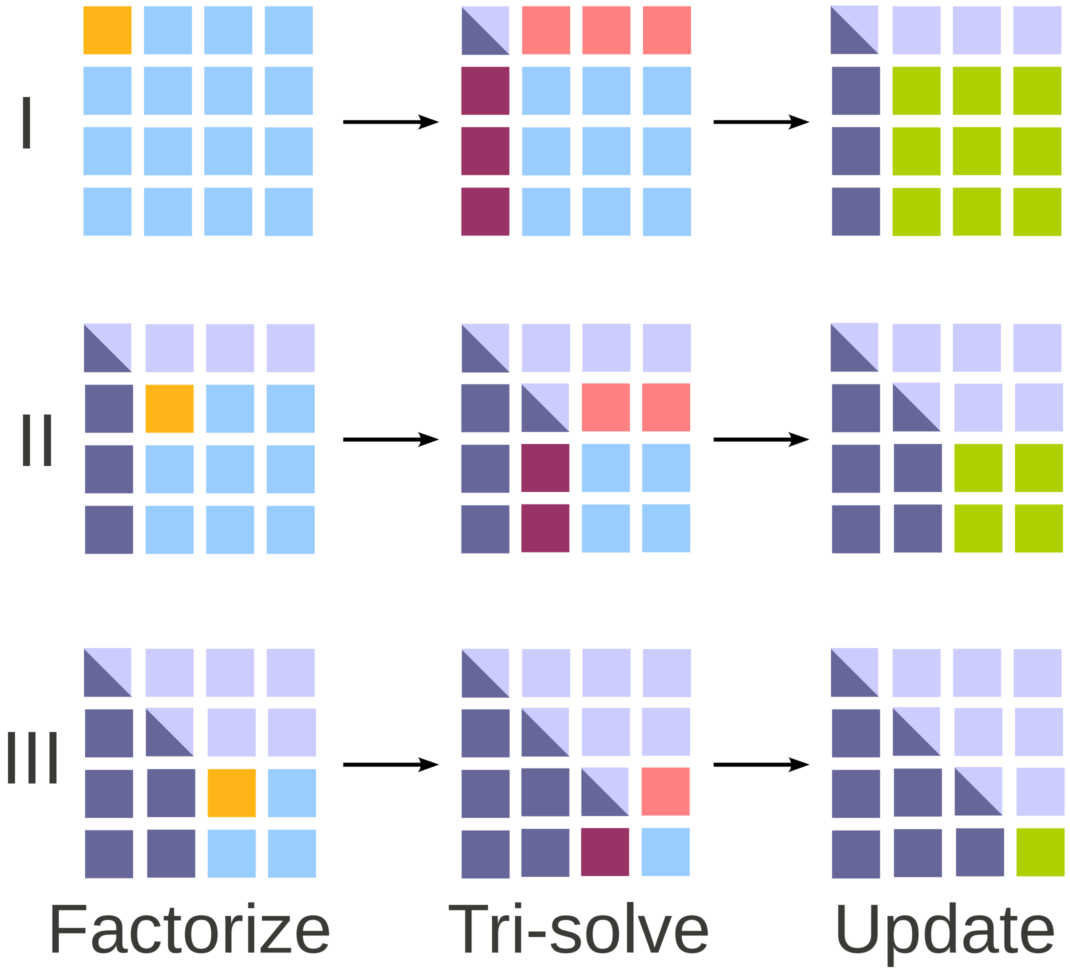
- Invoke `foo()` on all objects in collection A

```
ispace S = {0:N-1:1};

foreach (x,y in S * S){
 A[x,y].foo();
}
```

- `ispace` construct gives *index space*

Section communication

- Dense linear algebra (e.g. LU)



LU in Charisma

```
for(K = 0; K < N/g; K++){
 ispace Trailing = {K+1 : N/g-1};
  // factorize diagonal block, and mcast
 (d) ← A[K,K].factorize();

 // update active panels, and mcast
 foreach(j in Trailing){
  (c[j]) ← A[K,j].utri(d);   // row
  (r[j]) ← A[j,K].ltri(d);   // column
 }

 // trailing matrix update
 foreach(i,j in Trailing * Trailing){
  A[i,j].update(r[i], c[j]);
 }
}
```

Others too...

- Blelloch (work-efficient) scan
- MG
- Pipelining (Gauss-Seidel)
- Scatter-gather, reduction, multicasts (OpenAtom)
- Other dense linear algebra (Gaussian elimination, forward/backward substitution, etc.)
- MD

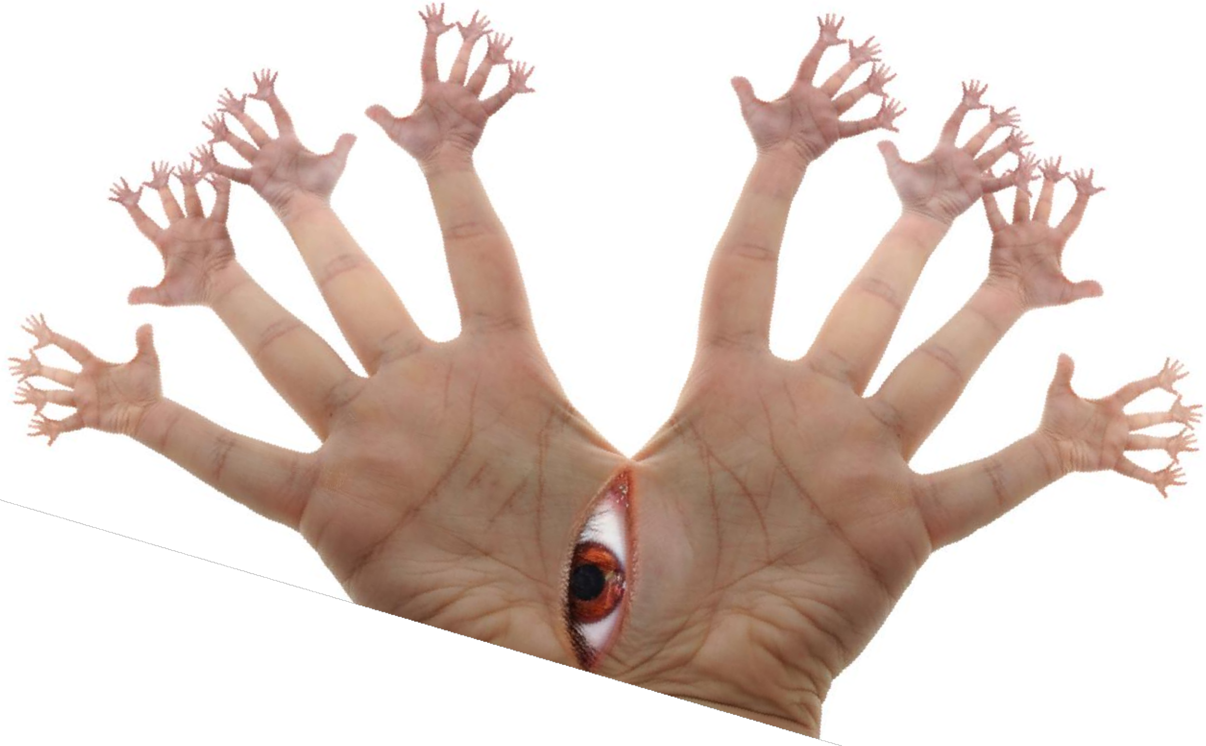
Expressing ***generative recursion*** with ***Divcon***

Generative Recursion

Elegant

Intuitive

Implicit, tree-structured parallelism



Examples

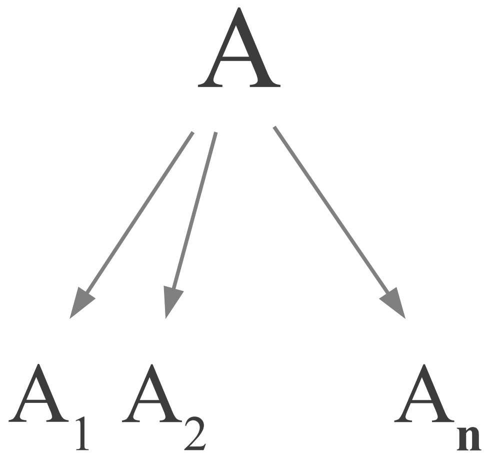
- Sorting, Closest pair
- Convex hull, Delaunay triangulation
- Adaptive quadrature, etc.

Recursive Structure

$$f(A) = g(f(A_1), f(A_2), \dots, f(A_n))$$

```
let
    A1 = f(p1(A)),
    A2 = f(p2(A)),
    ...
    An = f(pn(A))
in
    g(A1, A2, ..., An);
```

Data movement from $A \rightarrow A_i$



A

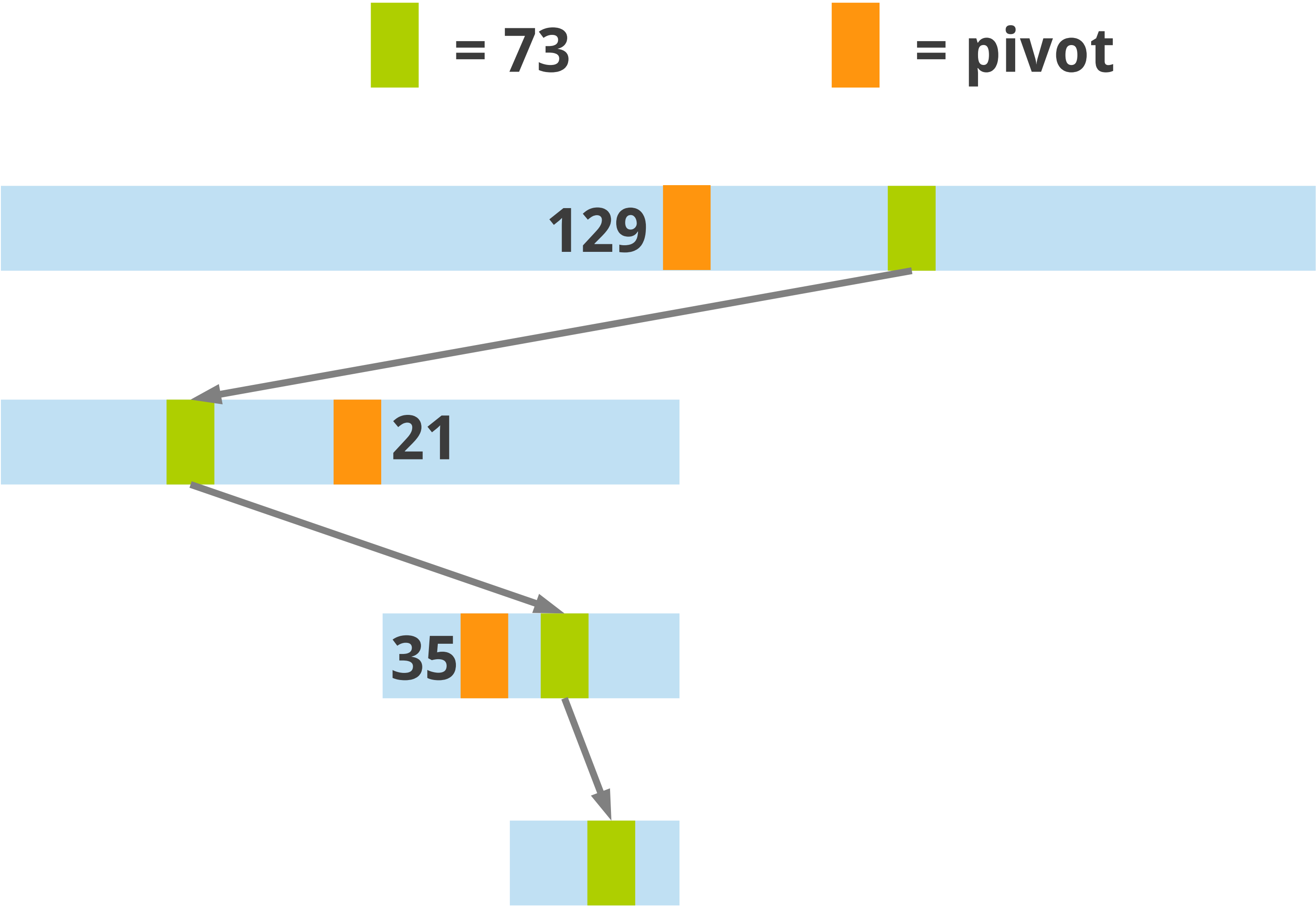
A_1 A_2 $A_{\mathbf{n}}$

- memcpy in shared memory systems
- Network communication in distributed memory!

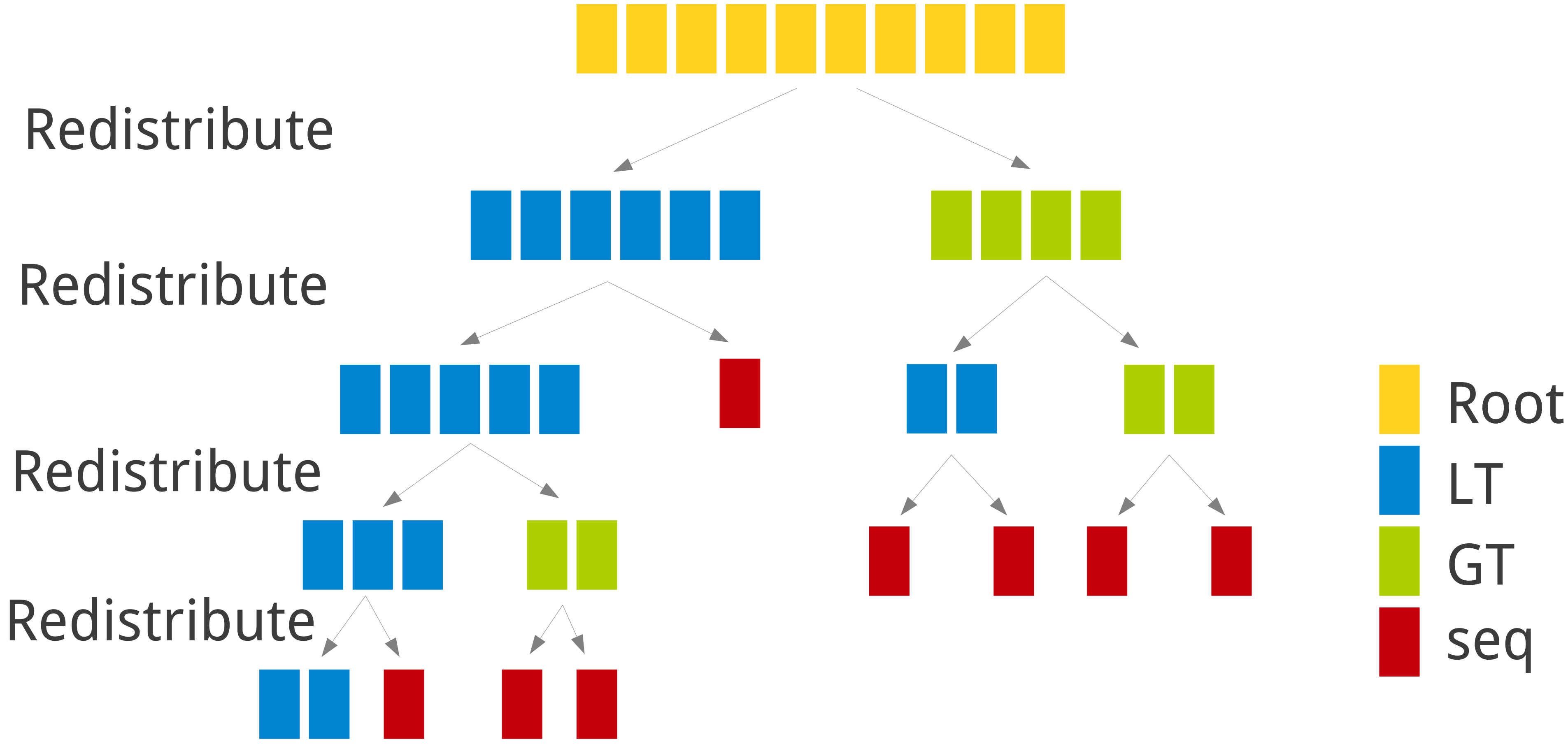
Quicksort

```
Array<int> qsort(Array<int> A){
    if(A.length() <= THRESH) return seq_sort(A);
    Array<int> LT,EQ,GT;

    int pivot = A[rand(0,A.length())];
    (LT,EQ,GT) = {partition(A,pivot)};
    return concat(qsort(LT),EQ,qsort(GT));
}
```

Significant redistribution costs



Parallel execution



Redistribute

Redistribute

Redistribute

Redistribute

🟨 Root
🟦 LT
🟩 GT
🟥 seq

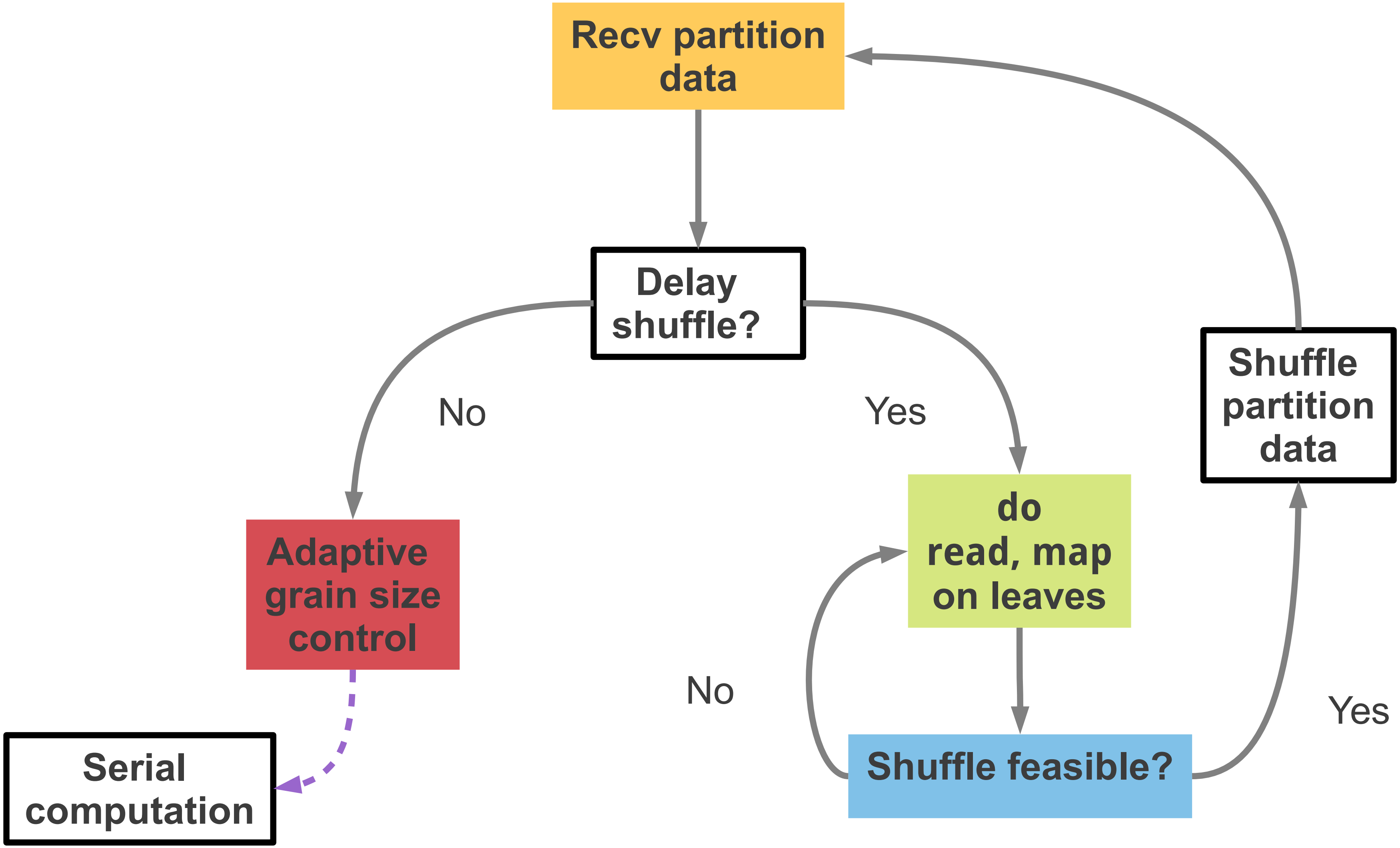
Delayed data redistribution

Amortize redistribution costs over several recursive invocations

Reduces communication

But lowers concurrency

Best of both worlds



Recv partition data

Delay shuffle?

No

Yes

Adaptive grain size control

Serial computation

do read, map on leaves

Shuffle feasible?

No

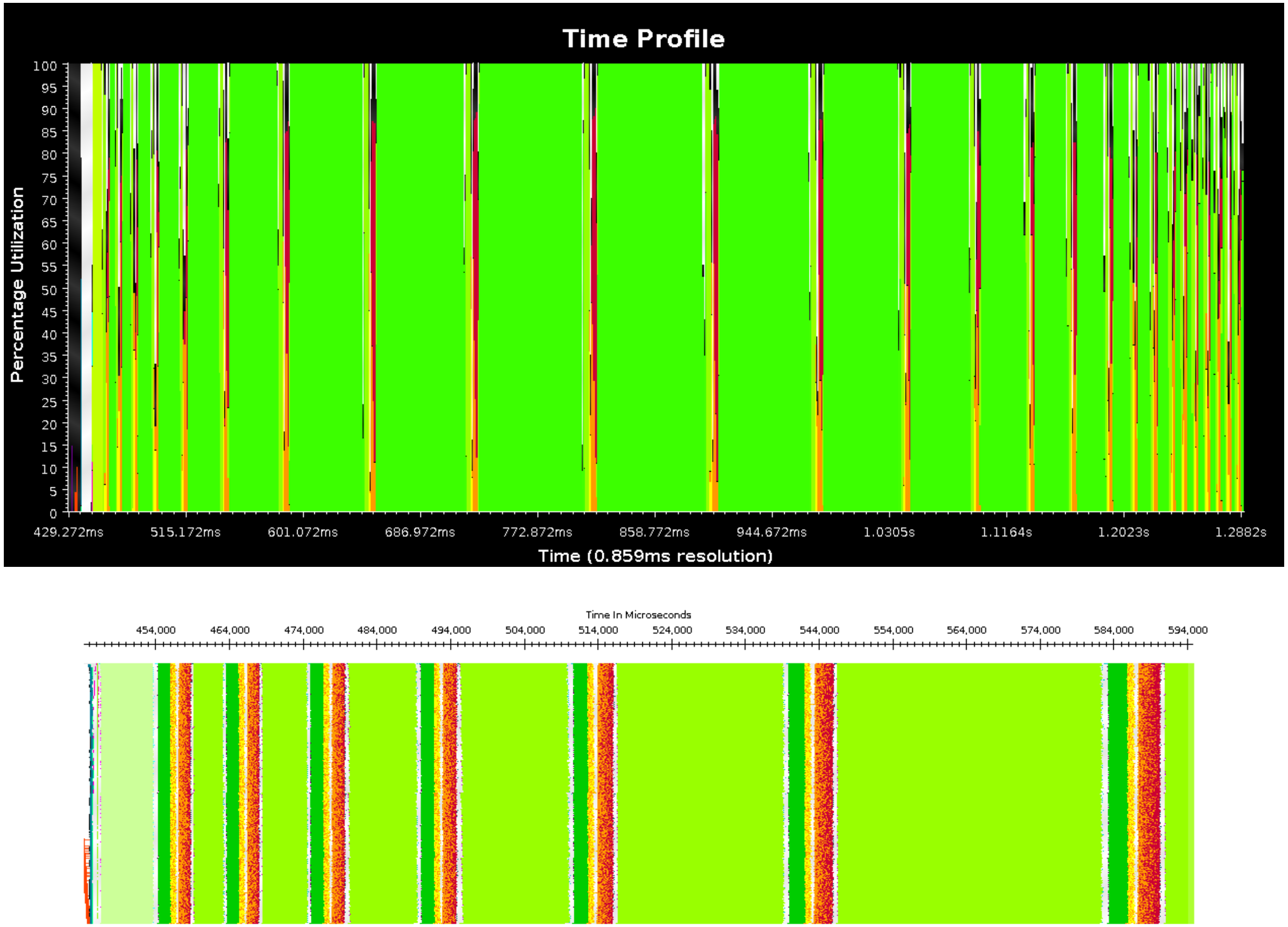
Yes

Shuffle partition data

Allows *consolidation*

- Redistribution delay → several (new) arrays distributed across same section of containers
- If operation-issuing tasks are kept on same PE, issued operations may be consolidated
- Consolidated operations applied together on target arrays

Allows *consolidation*



Quicksort on 256 BG/P cores

A framework for expressing ***tree-based algorithms***

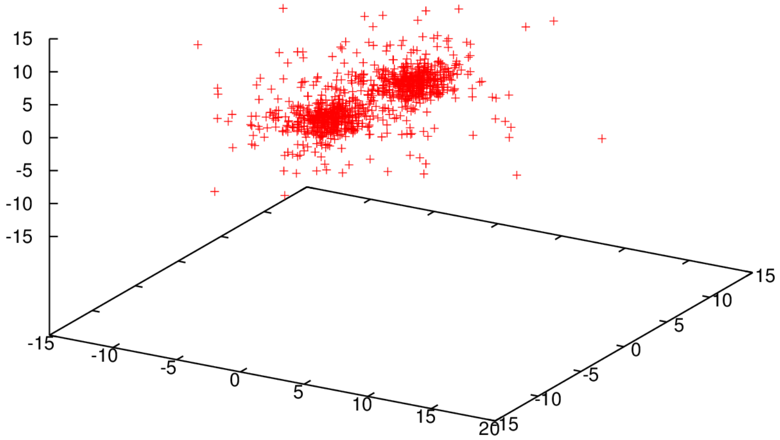
Tree-based algorithms

Structural (as opposed to *generative*) recursion

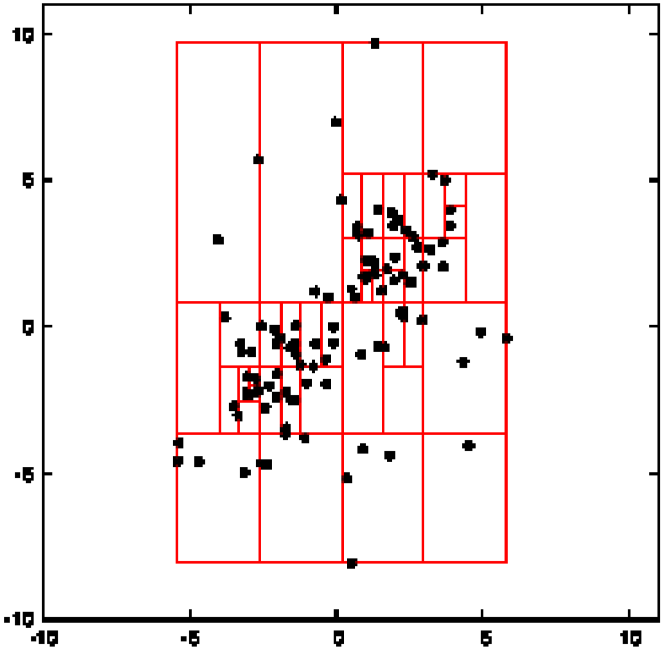
N-body codes, granular dynamics, SPH,...

Distributed tree + recursive traversal procedure

Data decomposition

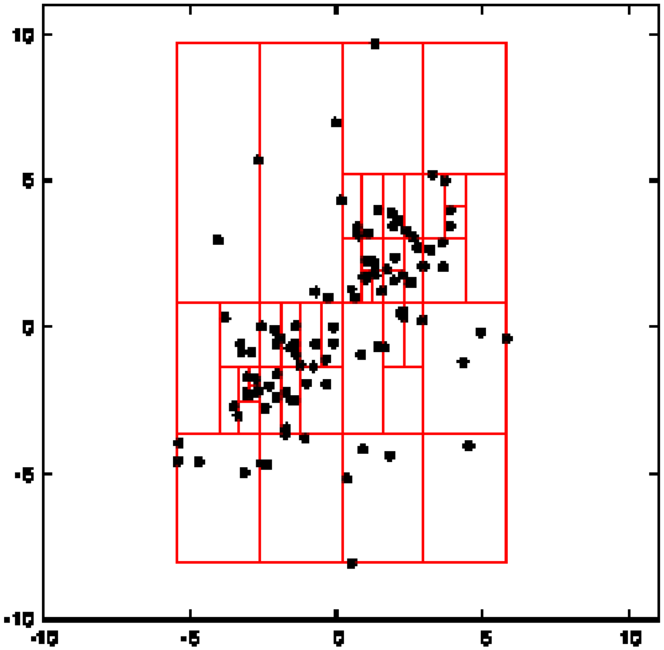


Spatial entities



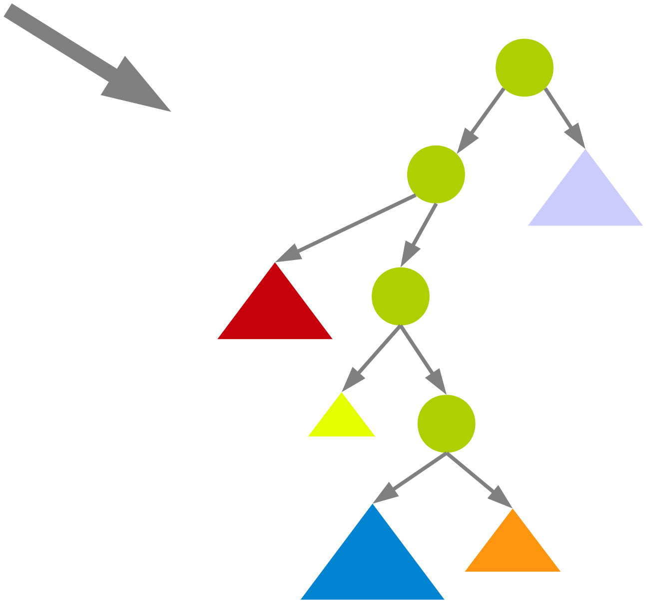
Compact spatial partitioning of data over chares

Distributed tree

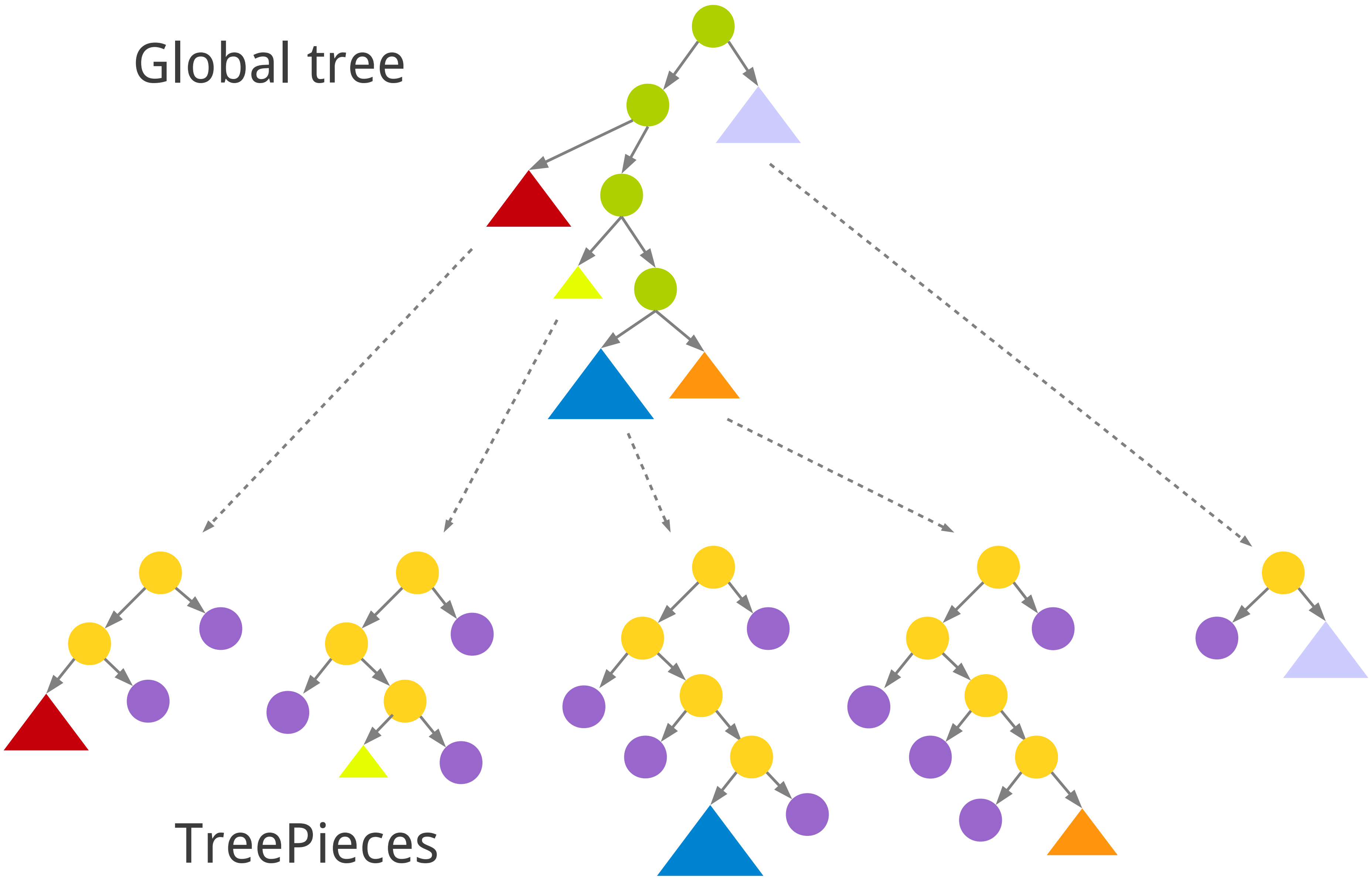


Compact spatial partitions

Global, distributed tree



“Chunked” distribution of data



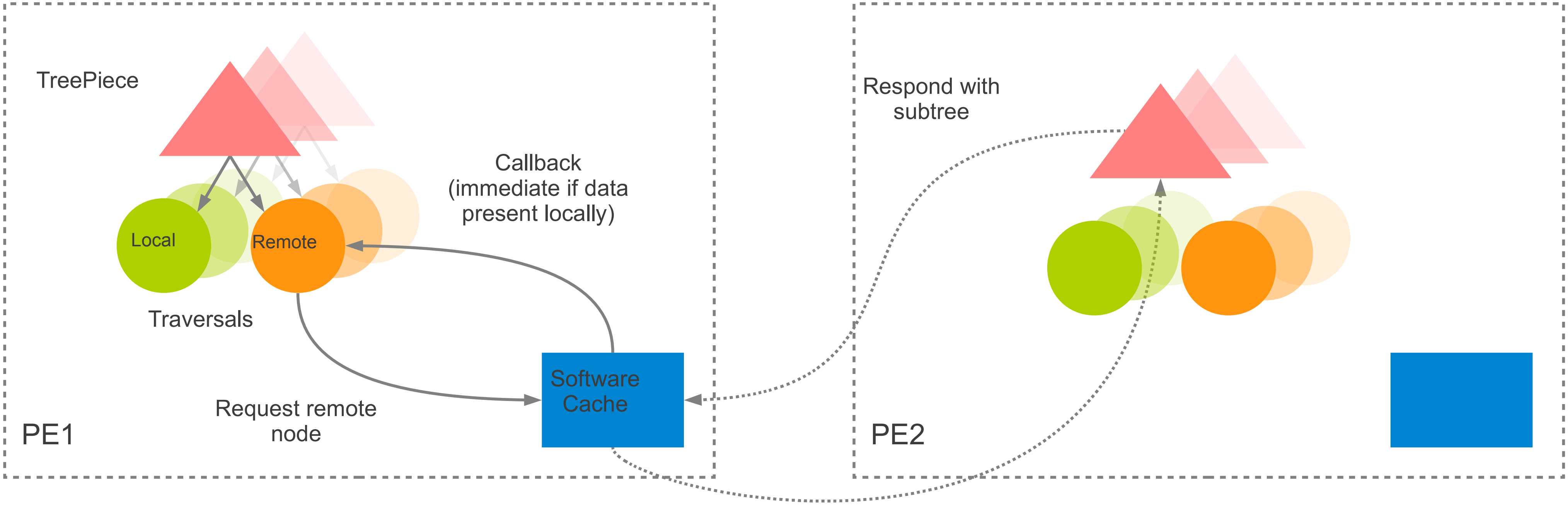
Global tree

TreePieces

Algorithm comprises concurrent traversals on pieces

- Visitor + Iterator pattern
- Visitor defines
  - node()
  - localLeaf()
  - remoteLeaf()
- Iterate over nodes using *traversal*
  - Order decided by traversal

Traversal with reuse



TreePiece

Local

Remote

Traversals

Callback (immediate if data present locally)

Software Cache

Request remote node

PE1

Respond with subtree

PE2

Send request message to owner TreePiece if data not present locally

Barnes-Hut control flow

```
for(int iteration = 0; iteration < parameters.nIterations; iteration++){
  // decompose particles onto tree pieces
  decomposerProxy.decompose(universe, CkCallbackResumeThread());

  // build local trees & submit to framework
  treePieceProxy.build(CkCallbackResumeThread());

  // merge trees
  mdtHandle.syncToMerge(CkCallbackResumeThread());


  …
}
```

Barnes-Hut control flow

```
for(int iteration = 0; iteration < parameters.nIterations; iteration++){
  …
  // initialize traversals
  topdown.synch(mdtHandle, CkCallbackResumeThread());
  bottomup.synch(mdtHandle, CkCallbackResumeThread());

  // start gravity and SPH computations
  treePieceProxy.gravity(CkCallback(CkReductionTarget(gravityDone), thisProxy));
  treePieceProxy.sph(CkCallback(CkReductionTarget(sphDone), thisProxy));

  // done with traversal
  topdown.done(CkCallbackResumeThread());
  bottomup.done(CkCallbackResumeThread());
  …
}
```

Barnes-Hut control flow

```
for(int iteration = 0; iteration < parameters.nIterations; iteration++){
  …
   // integrate particle trajectories
  treePieceProxy.integrate(CkCallbackResumeThread((void *&)result));

  // delete distributed tree
  mdtHandle.syncToDelete(CkCallbackResumeThread());
}
```

Visitor code

```
Class BarnesHutVisitor {

   bool node(const Node *n){
      bool doOpen = open(leaf_, n);
      if(!doOpen){
         gravity(n);
         return false;
      }
      return true;
   }

   ...

}
```

Visitor code

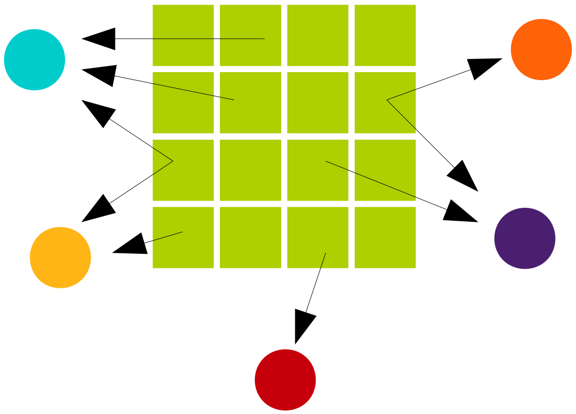
```
Class BarnesHutVisitor {
   void localLeaf(Key sourceKey,
                  const Particle *sources,
                  int nSources){
      gravity(sources, nSources);
   }

   void remoteLeaf(Key sourceKey,
                   const RemoteParticle *sources,
                   int nSources){
      gravity(sources, nSources);
   }
};
```

Distributed Shared Array programming with ***MSA***

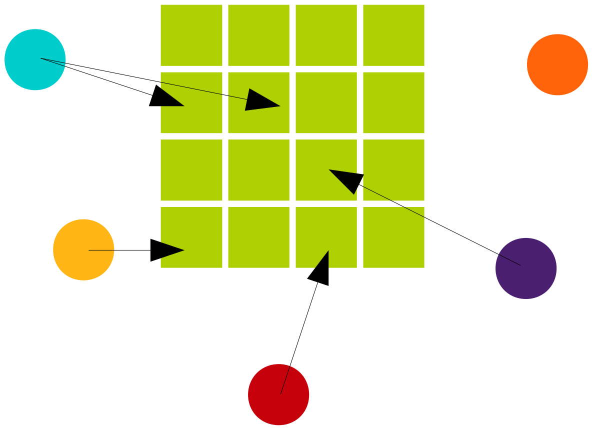
Multiphase Shared Arrays (MSA)

Disciplined shared address space abstraction

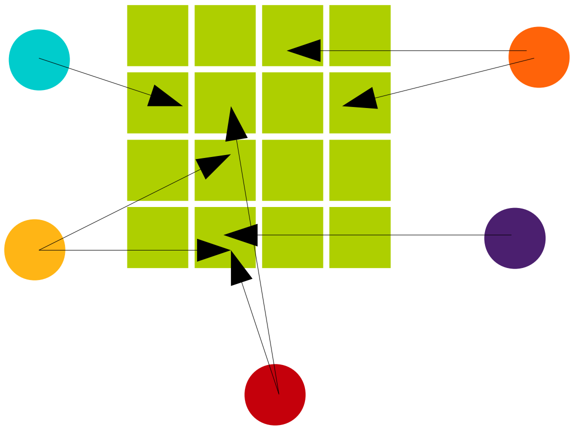
Dynamic *modes* of operation



Read-only

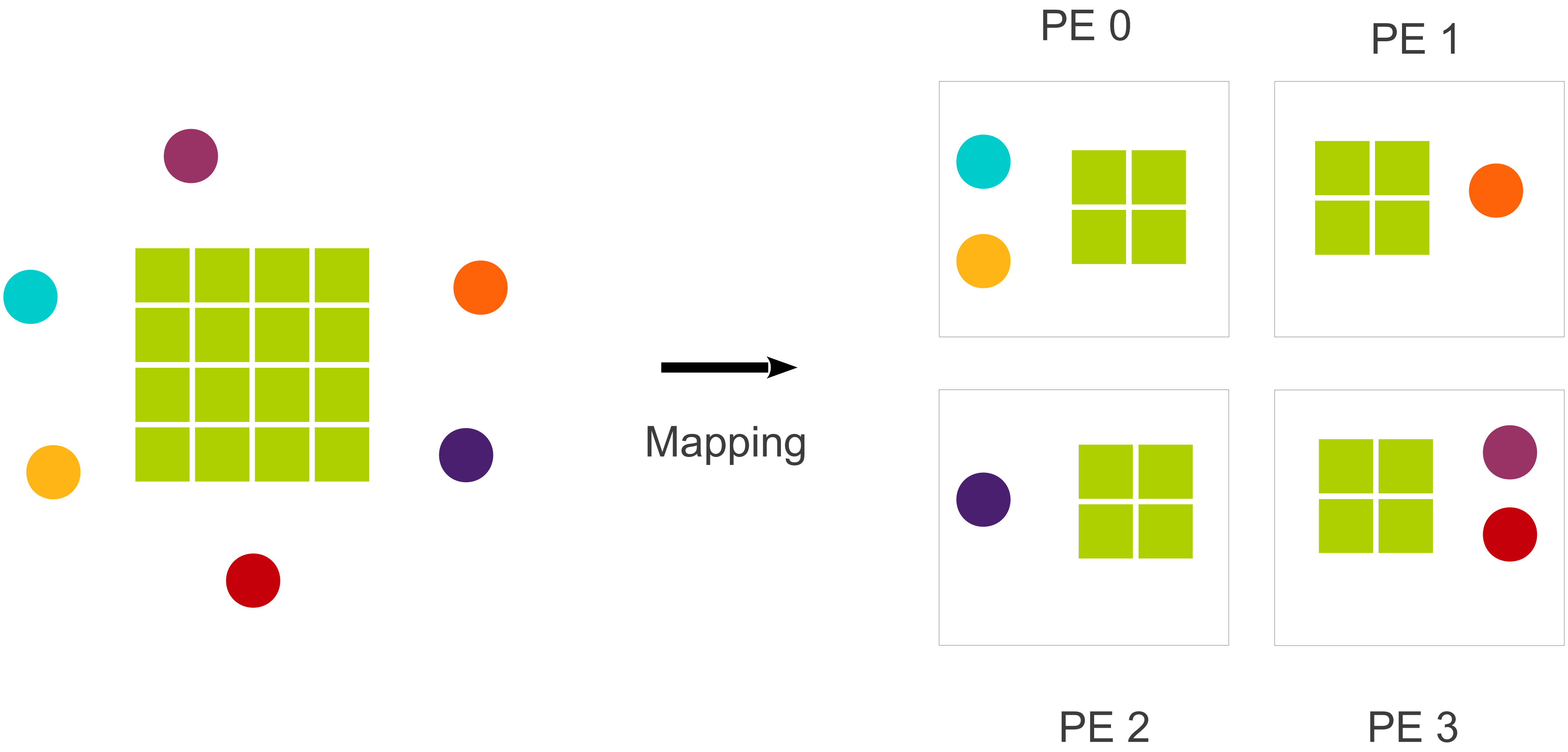


Write-exclusive



Accumulate

MSA Model



PE 0

PE 1

Mapping

PE 2

PE 3

Parallel histogramming in MSA

- Two MSAs, A() and Bins()
- A() in read mode, Bins() in accum mode

```
MSA1D<double> A;
MSA1D<int> Bins;

MSA1D::Read rd = A.getInitialWrite();
MSA1D::Accum acc = Bins.getInitialAccum();

For (int x = myStart; x < myStart + myNumElts; x++){
    acc(getBin(d.get(x)) += 1;
}
```

Compiler and Runtime optimizations

- Strip mining (Charj)
- Bipartite graph-based optimal placement
- Message combining

Conclusion

- Ecosystem of specialized languages
  - Productivity and performance
  - Higher-level constructs
- Common runtime substrate for interoperability
  - Completeness of expression