Programming Languages for Large Scale Parallel Computing

Marc Snir

Focus

- Very large scale computing (>> 1K nodes)
  - Performance is key issue
  - Parallelism, load balancing, locality and communication are algorithmic issues, handled (at some level) by user
- Scientific computing
  - Transformational, rather than reactive code
    - Memory races are bugs, not features!
    - Programmers expect reproducibility and determinism (for numerical analysis)
    - (partial exception) – associative/commutative operations (e.g., reductions)
- Large codes (>> 100 KLOC)
  - OO methodology

Current and Near Future High End Parallel Architectures

- Predominantly cluster architectures
  - each node is commodity CPU (Multi-Core Processor)
  - Nodes are connected via specialized interconnect
    - hardware/firmware support for rDMA (put, get)
    - no global cache coherence
- Assumptions:
  - Language handles only one level hierarchy (local/remote)
  - Language does not handle further ramifications of HPC architecture bestiary (vector, multithreading, heterogeneous architectures...)

Current Programming Environments

- C++ – provides OO support for large frameworks
- Fortran – provides performance for computational kernels
- MPI – provides interprocess communication
  - fixed number of processors, one MPI process per processor
  - single program
  - loosely synchronous
  - implicitly assume dedicated environment of processors running at same speed.

The Programming Language Domain

- Three dimensions:
  - Application type: Scientific computing, transaction server, client application, web services...
  - Software type: large, long-lived application, small prototype code...
  - Platform type: Uniprocessor, small MCP/SMP, large cluster...
- One size does not fit all! Different solutions may be needed in different clusters.
  - Polymorphic, interpretative language (e.g., MATLAB) for programming in the small
  - Transaction oriented languages for reactive code
- Q: How many different solutions do we need/can we afford? How do we share technology across different solutions?

Do we Really Need New Languages?

- New languages will make programmers more productive (HPCS premise)
  - MPI codes are larger
  - MPI is "low level"
- However:
  - MPI (communication) is small fraction of large frameworks and is hidden at bottom of hierarchy
  - Empirical studies show some problems are coded faster using MPI, other problems are coded faster using OpenMP (V. Basili)
  - Code size is bad predictor of coding time
  - Coding is small fraction of code development time, for large programs
  - Tuning is harder with higher-level languages
  - Other SE aspects of coding process and of ADE's may have more impact
  - Parallel compilers are rarely of high quality

What Features Do We Want in a New Language? (1)

1. Performance: can beat "normal" MPI codes
   - Fortran replace assembly when it proved to achieve better performance, in practice!
   - Opportunities:
     - faster, compiled communication that avoid software overhead of MPI
     - Compiler optimizations of communications
2. Semantic & Performance transparency
   - Can analyze & understand outcome and performance of parallel code by looking at source code: language has simple (approximate) performance semantics
     - **Time = Work/p + Depth**. Need approximate composition rules for Work and Depth. First usually holds true; second holds true only with simple synchronization models.
3. (Some) user control of parallelism (control partitioning), load balancing, locality and communication
   - Whatever is part of algorithm design should be expressed in PL

What Features Do We Want in a New Language? (2)

4. Nondeterminism only when (rarely) needed
5. Support for iterative refinement
   - Can write code without controlling locality, communication, etc. if these are not critical; can refine later by adding control
6. Modularity & composability
   - A sequential method can be replaced by a parallel method with no change in invoking code
     - Requires support to nested parallelism!
   - Different parallel programs can be easily composed
   - Semantics and performance characteristics of parallel code can easily be inferred from semantics and performance characteristics of modules
7. Object Orientation
8. Backward compatibility
   - Interoperability with MPI codes
   - Similar to existing languages

9. Virtualization of Physical Resources

- Processor virtualization
  - Applications are written for virtual processors (aka locales); mapping of locales to processors is
    - done by runtime
    - is not necessarily one-to-one
    - can change over time (load balancing)
- Why not user controlled load balancing?
  - Change in number of available resources can be external
    - failures (especially for large multicore processors that may mask core failures)
    - dynamic power management
    - composition of large, independently developed codes in multidisciplinary applications
      - Each code needs to progress at “same rate”; progress rate may change as simulation evolves and resources may have to be moved from one component to another
- Processor virtualization is cheap (Kale and co.)

10. Global Name Space

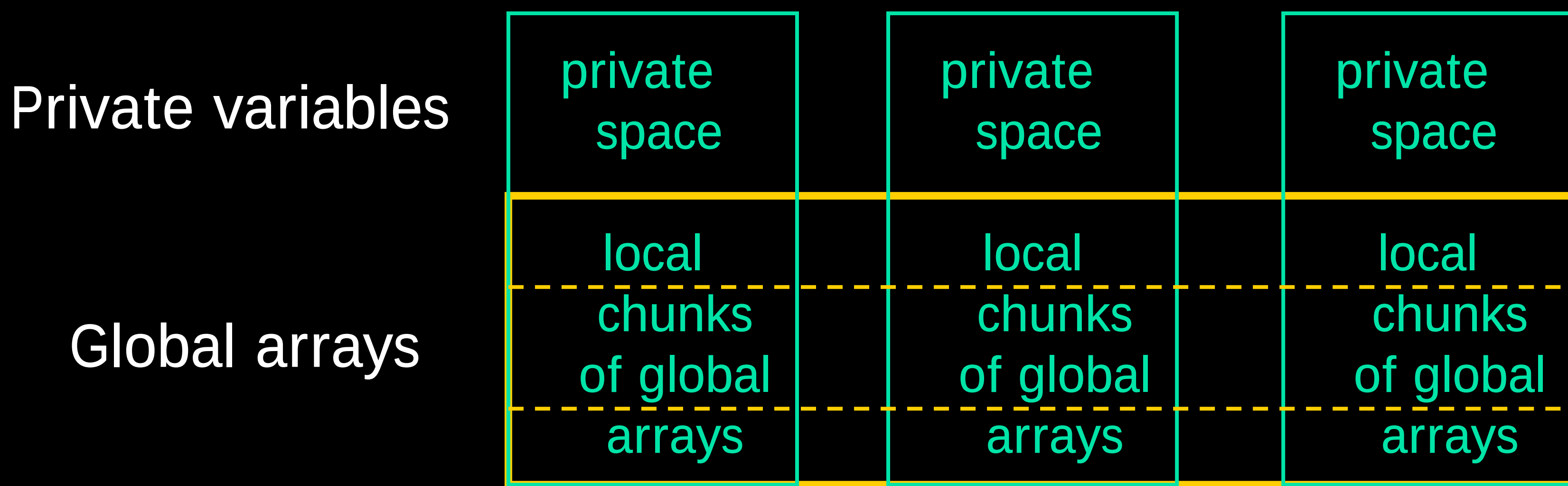
- Variable has same name, wherever it is accessed
  - Still need local copies, for performance
    - caching, rather than copying: location is not part of name!
- Software caching: software manages (changing) association of global name to local address
  - Correspondence between global name and local address can be
    - compiled, if association is persistent (e.g., HPF-like partitions)
    - managed by run-time, otherwise (hash table)
    - optimizations possible if association is slow changing (inspector-executor)
    - run-time compilation can be used here!
  - It is necessary to support dynamically changing association!

11. Global Control and Dynamic Parallelism

- MPI: partition of control is implicit (done at program start; actual code describes actions of individual processes; program presents local view of control and global computation is inferred by reasoning about the global effect of the individual executions
- OpenMP: partition of control is explicit (parallel block or loop); program presents global view of control
- Global view (+ virtualization) supports *dynamic parallelism* – number of concurrent actions can vary
  - Needed for composability
  - Needed for iterative refinement

Partitioned Global Array Languages (PGAS)

- Unified Parallel C (UPC) and Co-Array Fortran (CAF)
  - global references are syntactically distinct from local references
    - local references have no overheads
  - sequential code executed once on each processor (local view of control)
    - with the addition of global barriers and forall in UPC



A Critique of PGAS

1. ☺ Performance : CAF can beat MPI
   - Advantage of compiled communication
2. ☺ Semantic & performance transparency – simple model
3. 😐 User control of data and control partitioning – at level of MPI
4. ☹ Nondeteminism: can have conflicting, unsynchronized accesses to shared memory locations
5. ☹ Iterative refinement: like MPI (need to start with parallel control and distributed data)
6. ☹ Composability, modularity: cannot easily compose two CAF/UPC programs; have no nested parallelism
7. ☹ Object orientation: no UPC++ (dynamic type resolution screws up compiler)
8. ☺ Backward compatibility: easy
9. 😐 Virtualization: not done but doable
10. ☹ Global name space: no caching, only copying
11. ☹ Dynamic parallelism: none

Similar Critique Applies to HPCS Languages

- X10
  - No global name space with caching
  - No simple performance model (asynchronous RMI)
  - Focus on constructs needed for reactive codes (atomic sections, futures, async actions...)
  - No support for iterative refinement, modularity and composability
- Chapel
  - ...
- Fortress
  - ...

We Can, Perhaps, Do Better: PPL1

- Start with a good OO language (Java, C++, C#...): started with Java
  - simpler, better defined semantics
  - simpler type and inheritance models
- Remove Java restrictions to good performance
  - do not need exact reproducibility (floating point reproducibility, precise exceptions)
  - can live without dynamic loading (or with expensive dynamic loading)
  - can live without JVM
  - can live without reflection
  - ...

PPL1 (2)

- Add extensions needed for scientific computing convenience and performance
  - True multidimensional arrays for more efficient indexing
  - Immutable classes (Titanium): a class that "behaves like a a value"; e.g., for efficient support of complex numbers
  - Operator overloading (e.g., for convenient support of complex numbers).
  - Deep copying (at least for immutable classes)

```
Complex a,b,c;
...
a := b+5*c;
```

Shallow vs. Deep Copying

Matrix a = new Matrix(...)

Matrix b = new Matrix(...)

Matrix c = new Matrix(...)

a = b;

a := 1;

c := a;

c := c+2;

~~b = 3;~~

Compiler Support for General Data Structures

- Modern scientific applications increasingly use “irregular” data structures
  - sparse arrays
  - graphs (irregular meshes)
- The mapping of the data structure into a linear space is managed by user/library software, not compiler
  - one misses optimization opportunities
- Should use type and compiler analysis to capture as much information on data structure as possible and let compiler do the mapping

Example: Ordered Set

- How dynamic is set (are elements added deleted)?
- How much space is needed to represent set
- How easy it is to access an element?
- How efficient it is to iterate over the set? (or over "meaningful" subsets?)
- Assume fixed set of integer tuples (*points*)
  - set of indices of elements in a (sparse/dense) array
  - meaningful subsets: rows/columns (projections)

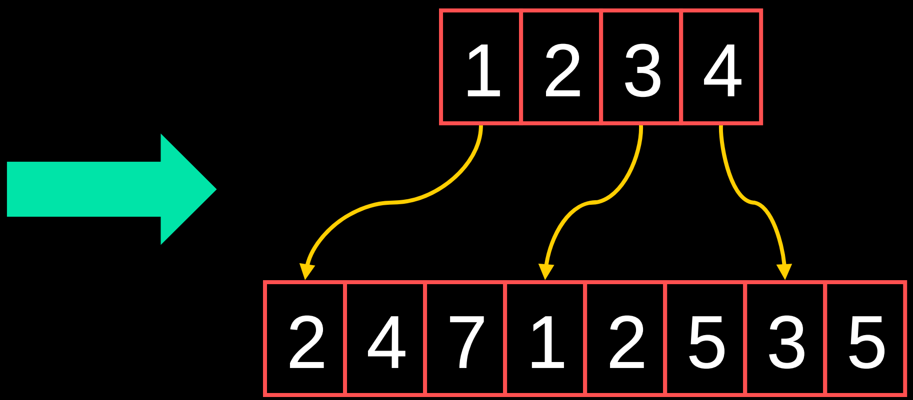
Set of Points (1)

- **General set**: use hash table
  - storage: $(1+\lambda)\times$#points $\times$ arity
  - can iterate efficiently over all set, not over "meaningful" subsets (would need additional linked lists)
    - Spatial locality is not perfect or hash is more complex
  - search for item requires constant number of memory accesses

Set of Points (2)

- **Semi-dense set**: use standard representation for sparse arrays

(1,2), (1,4), (1,7),
(3,1), (3,2),
(4,3), (4,5),



- Storage: $(1+\varepsilon)\times$#tuples provided rows are reasonably dense
- Element access: log(row_density) (unless have added hash tables)
- Iterator: very efficient (good locality) for global iterations and row iterations

Set of Points (3)

- **Rectangular grid**: store two opposite corners of box.
  - storage: $2\times$arity
  - can iterate efficiently over all set, over rows, columns, etc.
  - search for item requires constant number of operations (often no memory accesses)

Set of Points (4,5…)

- Sparse array consisting of dense subarrays
  - ...
- Banded matrices
  - ...
- Current prototype implementation distinguishes general sets, sets of points and grids
  - could add more types (does not make language more difficult, with right class hierarchy)
  - could have compiler guess right implementation

Basic PPL1 Types (1)

- Java + (modified parts of) Java Collection Framework
- Ordered sets
  - cannot modify sets
  - set operations (S := S+T)
  - element-wise operations (not specified yet)
  - reduction operations (s := S.sum())

Basic PPL1 Types (2)

- Maps
  - cannot modify domain values; can update range values
  - map access and update
    - One element

      a = M[i];
      M[i] = 3;
    - Multiple elements

      M := M1[M2]; \\ composition: M[i] == M1[M2[i]], for all i
      M1[M2] := M3 \\ (M1[M2[i]] == M3[i], for all i in domain of M3;
      \\ other locations are unchanged
      - one element is particular case of multiple elements
  - element-wise operations (M1 := M1+M2)
  - reductions (s = M.sum())
- Array: map with grid domain (distinct type)

Parallelism

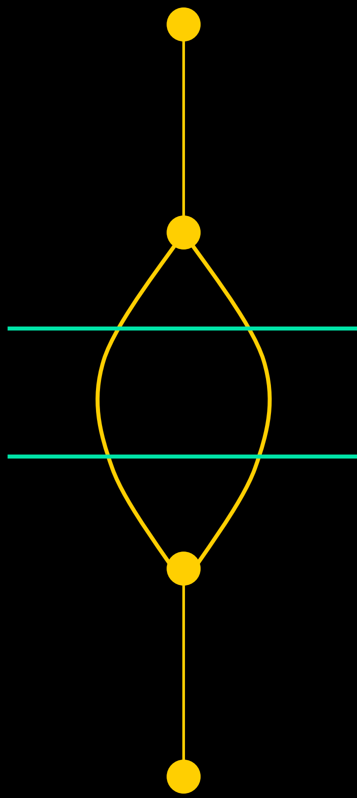
- Want virtual “processors” (resources executing threads)
- Want the ability to specify that a datum is located where a thread executes (associate variable location with thread location)
- Assume “locations” (or *sites*) that are virtual, but not remapped too frequently;
  - user can associate the execution of (at most) one thread with a site
  - user can (cache) data at a site.
- *Cohort*: set of sites
  - New cohorts can be created dynamically
- Sites are associated with properties that provide some control on the physical location
  - collocates sites
  - anti-located sites
  - “persistent storage” sites: I/O can be a form of data caching

A Short Trip into History

- Goto statement considered harmful (Dijkstra, 68)
  - Goto's are harmful because it is hard to specify the "coordinates" of a point in the program execution
    - In a structured program need to specify the, PC, the stack of calls and the index of each loop in the current loop nest
    - In an unstructured program need to specify the entire trace of basic blocks
  - Goto's are unnecessary because a goto program can be transformed into a gotoless program that has close to same running time
- Shared variables considered harmful
  - Unrestricted use of shared variables is harmful because it is hard to specify the "coordinates" of a point in the program execution
    - Need to specify the interleaving of shared variable accesses
  - Such use is unnecessary because a PRAM program can be transformed into a Bulk Synchronous Parallel program that does about the same amount of work (assuming logarithmic parallel slack)
  - BSP model: any two conflicting accesses to shared variables are ordered by a barrier

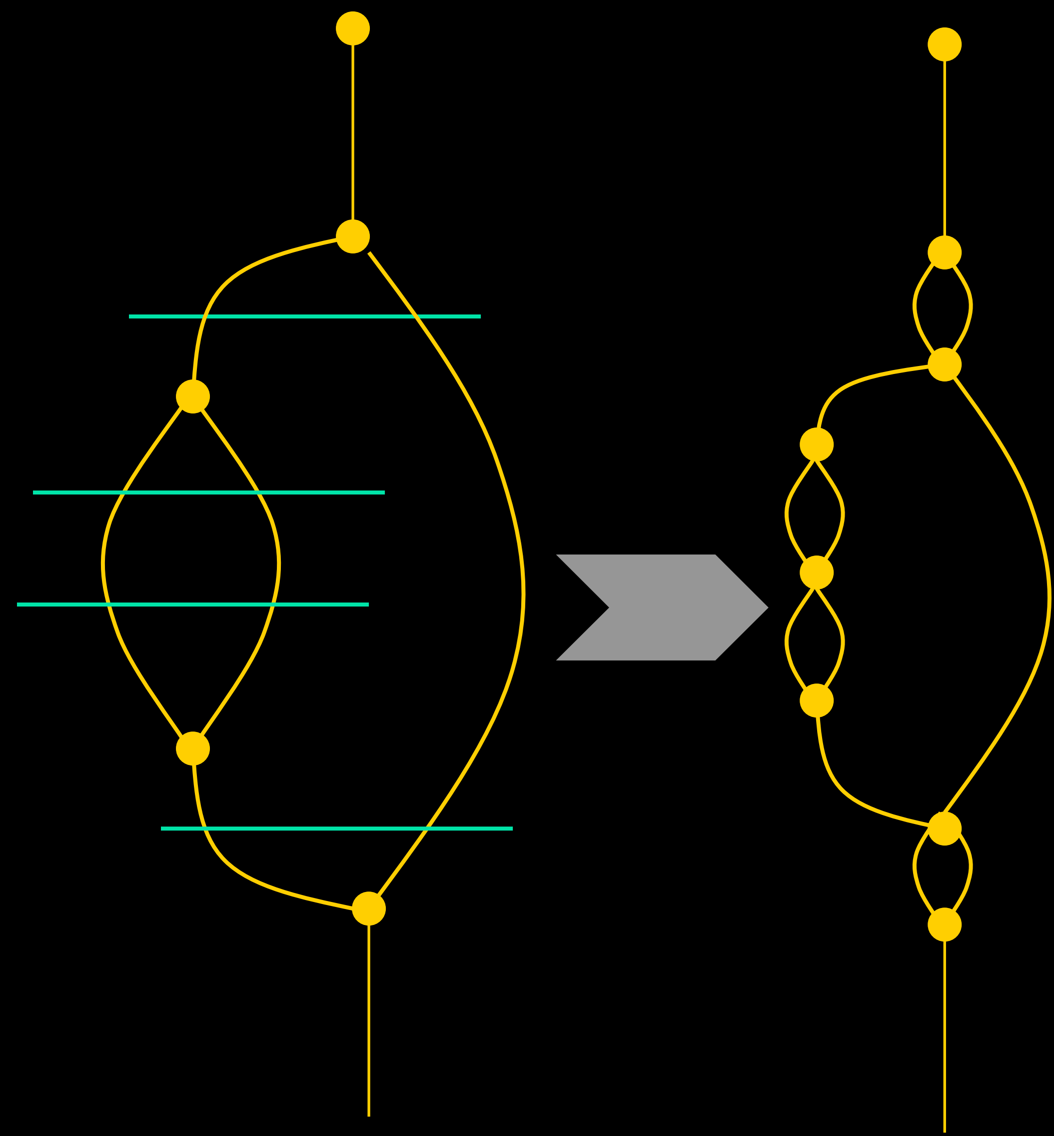
A Simple Incarnation of the BSP model

- Sequential code + (unnested) parallel (forall) loops; iterations within a parallel loop do not execute conflicting accesses.
  - History of program entirely determined by
    - global history
    - "local history" of each parallel iterate, if within parallel loop.
  - Still true if allow global barriers in parallel loops



Nested BSP Model

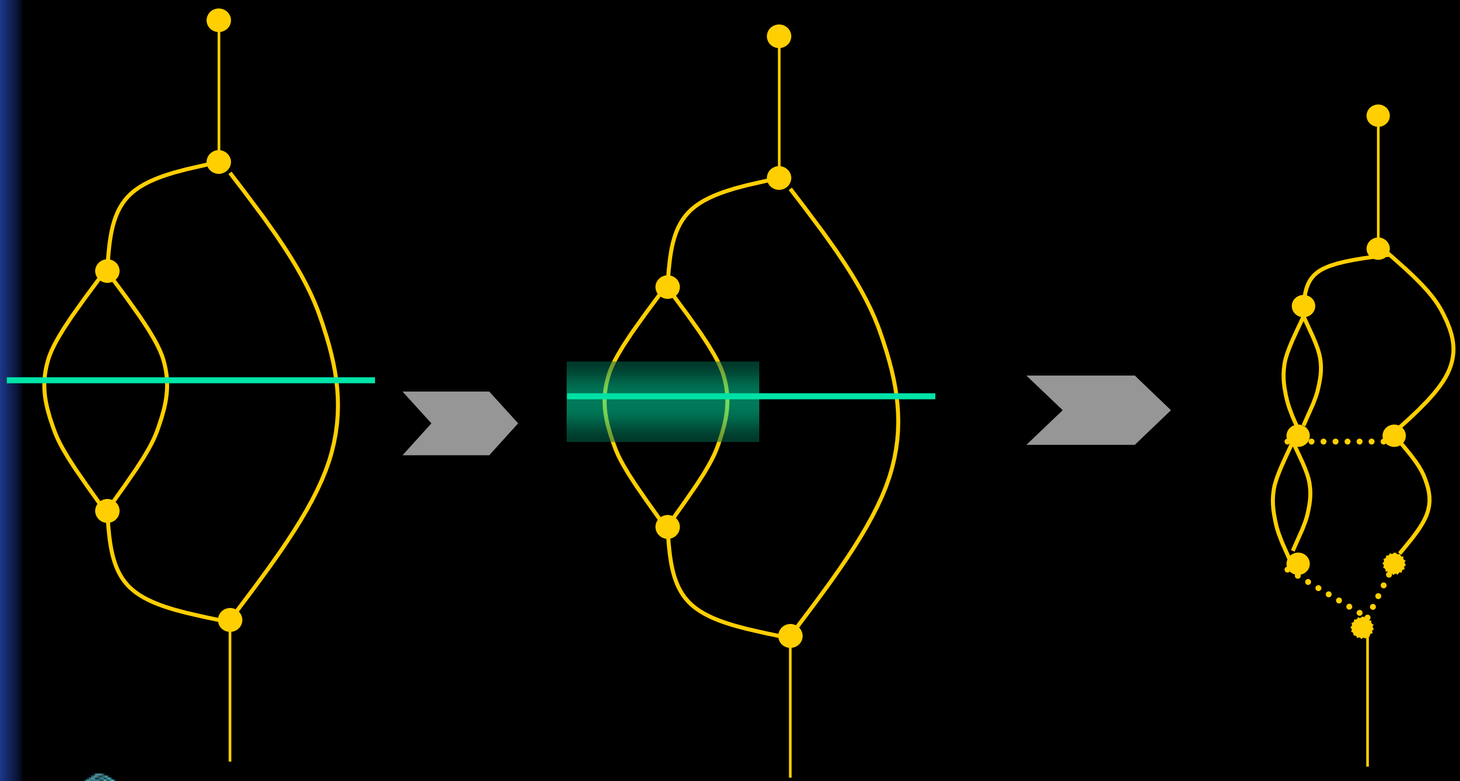
- Allow nested parallel statements
- Continue to disallow concurrent conflicting accesses to shared variables
- Execution state still has a simple description
- Compiler run-time optimization: synchronization weakening
- Well structured program: parallel control flow is represented by series-parallel graph



Marc Snir

Nested Barriers

- Useful for multiphysics codes
- Solution A: provide named barriers
  - creates opportunities for deadlock and for ill-structured program
- Solution B: have barrier set of sites synchronized by barrier determined by scope
- Solution C: allow code (including barriers) within barrier constructs; barrier code behaves as if executed in parent locale



Code in Barrier

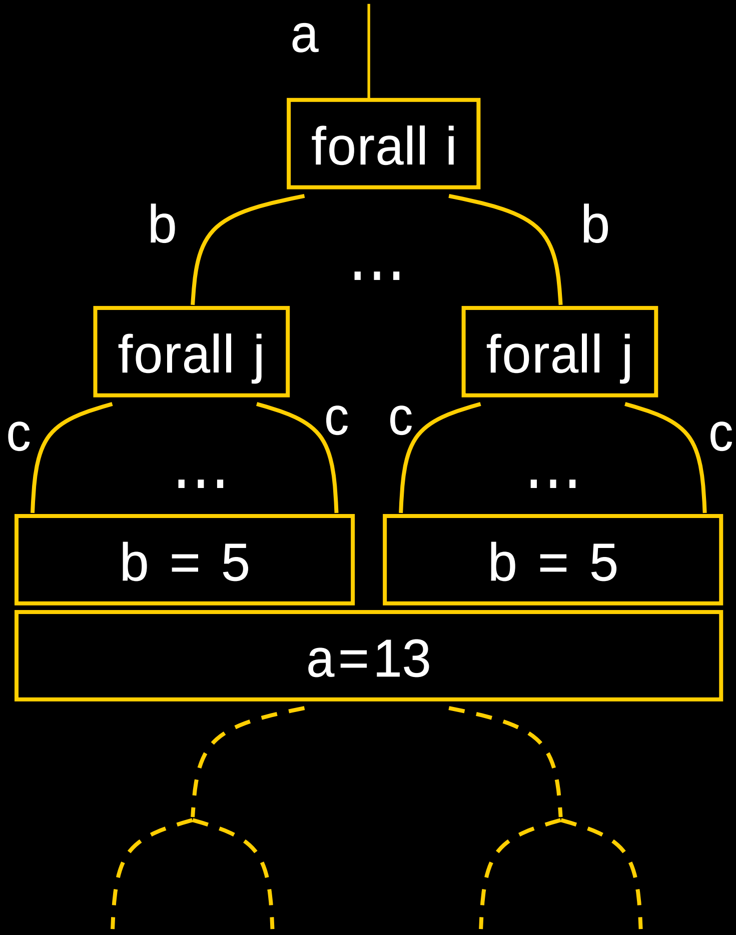
```
global int i, j, sum;
Site[] c = new Site[2];
sync {
   parallel {
      on c[0] : {
         i = 3;
         barrier();
         i = sum;
         }
      on c[1] : {
         j = 7;
         barrier();
         j = sum;
         }
   default: sum = i+j;
   }
```

i==j==sum==10

- parallel: syntactic sugar replacing forall when each site executes different code
- sync: barrier will be used in parallel construct
- default: default code executed in barrier
  - Could have multiple barrier labels and multiple actions

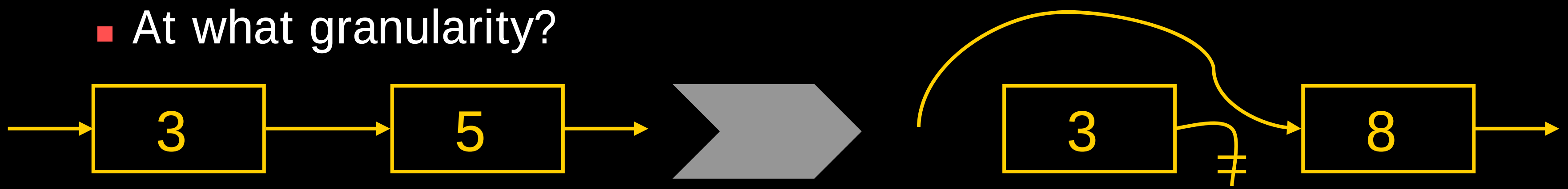
Nested Barrier Example

```
...
int a;
Site[] s = new Site[10];
sync {
  forall(int i : {0..9}; on s[i]) {
   int b = i;
   Site[] t = new site[5];
   sync {
      forall (int j : {0..4}; on t[j]) {
         int c = j;
         barrier()
         }
      default: {
         b = 5;
         barrier();
         }
      }
   }
   default: a=13;
}
```



Is Nested BSP Model Good Enough?

- Not for reactive codes – these need atomic transactions
- Need to allow reductions – concurrent commuting updates
  - Predefined / used defined
  - At what granularity?



  - Linked list reduction: need to modify atomically three records
- Q: assume transactional memory that supports transactions on few (3) locations. Does this cover all reductions of interest?
- Q: can we verify commutativity in all cases of interest?
- Q: can we have a practical race detection scheme, with a right mix of strong typing, compiler analysis, and run-time checks?

Variable Types and Modes

- Types: Local (accessible only at site where instantiated) vs. global (can be shared)
- Modes of a variable at a site:
  - Private (read/write)
    - variable is invalid at all other sites
  - Shared (read-only)
    - variable is shared or invalid at all other sites
  - Protected (accessible within atomic section)
    - variable is transactional or invalid at all other sites
  - Invalid (not accessible)

Example of Modes

```
global class Point {
   int x, y;
   Point(int x, int y) {this.x = x; this.y = y; }
   global static Point origin = new Point(0,0);
   }
class Test {
   public static void main(String[] args) {
      global Point p = new Point(3,5);
      global Point q = p;
      Site s[] = new Site[3];
      shared s : origin;       // origin can be concurrently accessed
                               // on all sites of s
      protected s: p.x, p.y ;  // the coordinates of p can be accessed and
                               // within an atomic section at all sites of s
      private s[0]: q;         // variable q can be accessed and updated
                               // only on site s[0]
      forall( int i: {..2}; on s[i]) {
         atomic{ p.x = i};     // the final value of p.x is either 0, 1 or 2
         atomic{ p.y += i};    // the final value of p.y is 8
         if (i==0) q = origin;
      }
   }
```

Dynamic Mode Change

```
global int a;
...
private s[0] a;
   forall(int i:{..9}; on s[i]) {
      if (i==0) {
          private t[0] a;
               sync{
            forall(int j:{..4}; on t[j]) {
               ...
               barrier();
               ...
               }
            default: private t[1] a;
            }
         }
     else {...}
    }
...
```

Mode Change

- User code can change variable mode in forall preamble or forall barrier
  - Change is done “globally”, for all threads of forall
  - The user code can weaken, but not strengthen, variable mode
    - Mode change cannot violate caching protocol wrt to threads spawned before the mode change
- Need to check when parallel loop is instantiated that only one thread is executed per site
  - compile time for simple on expressions, runtime, otherwise
- Need to check that mode changes are consistent with current mode
  - compile time if only stronger modes can reach mode change expression; run-time otherwise
- Need to check, when access occurs, that access is consistent with variable mode
  - compile time if access can be reached only with right mode, run-time, otherwise
- Q: will run-time checks be sufficient most of the time?
  - probably need interprocedural analysis

Sharing of Arrays

- Arrays can be partitioned
  - regular partitions (block-cyclic)
  - semi-regular partitions (block-cycle, with variable size blocks) – HPF2
  - arbitrary partitions (defined by maps)
- Each partition can be handled as a “variable” wrt to caching protocols
  - user-defined cache lines!
- Mapping from global to local addresses will be cheaper or more expensive according to regularity and binding time of partitions
  - opportunities for run-time compilation?
- Conflict between desire to have similar syntax for sematically similar constructs (partitions) and desire to provide clear feedback to to user on performance issue
  - Thesis: conflict should be solved by ADE.

I/O

- File is array; parallel I/O operations are parallel array accesses and updates
- File is persistent if it is located on persistent site when computation ends
  - site attributes

Design principles Applied to PPL1

1. Performance: **TBD**
2. Semantic & Performance transparency: better than current
3. (Some) user control of parallelism (control partitioning), load balancing, locality and communication: control parallelism and communication; load balancing is done by run-time (could provide hints)
4. Support for iterative refinement : good; can start with unrestricted sharing + atomicity and refine
5. Modularity & composability: good
6. Object Orientation: good
7. Backward compatibility: can be easily achieved
8. Virtualization: yes
9. Global name space: yes
10. Global control: yes
11. Dynamic parallelism: yes

Summary

- It is not clear that a new PL is the solution to HPC productivity.
  - If it is, its design has to be driven by a good understanding of parallel programming patterns
- Research hypotheses:
  - Java's approach of static and dynamic checks resulted in (type, memory) safe codes, with acceptable overheads; a similar approach can be used to have concurrency safe codes, with acceptable overheads.
  - Scientific codes can be expressed efficiently using a nested BSP model, augmented with atomic section for commuting updates
  - One can provide similar syntax/semantics for regular/irregular static/dynamic sharing while leveraging compiler optimizations for the easy cases
  - One can develop and ADE that provides a useful feedback on performance aspects of the language without burdening the language design itself

Questions?