CSEP 524: Parallel Computation
(week 5)

Brad Chamberlain

Tuesdays 6:30 – 9:20

MGH 231

Our goal for tonight

- Wrap up all major shared memory topics
- Transition to data parallelism
- Set up to switch to distributed memory next week
  - distributed memory architectures
  - SPMD programming/execution model
  - Message Passing / MPI
  - distributed memory algorithms
  - …

Search and Eureka

Parallel Algorithms: Search

Search: search some space for answer(s)

- Could be a data structure (graph, tree, database, …)
- Could be a conceptual space (molecules, passwords, …)
  - potentially infinite or at least combinatorially huge
- What are we looking for?
  - any valid answer?
  - all valid answers?
  - the "best" answer according to some metric?

- Ability to prune makes search interesting
  - Has the potential to scale superlinearly or not at all

Terminating Searches Early: Eureka

Eureka: "I found the answer, everybody else quit!"

- an intuitive, but advanced, form of synchronization

Two main varieties:

- passive/reactive
  - upon finding solution, task sets a shared flag ("I found it!")
  - other tasks periodically check flag to see if they should quit
  - (essentially what you were asked to do in HW#2)
- aggressive/proactive
  - upon finding solution, task terminates its siblings

  \+ less overhead for searching tasks to look over shoulder

  \+ less overhead for unwinding stacks of terminated tasks

  − challenges w.r.t. tracking active tasks and terminating them

Note on Passive Eureka, HW, and MCM

Task Parallelism / Tasks and Threads

Task Parallelism

Task Parallelism: What we've been doing so far

- expressed in terms of what each task will do

  e.g., cobegin { foo(); bar(); } *// one task does foo(), the other bar()*

  e.g., coforall tid in 0..#nTasks do foo(); *// nTasks tasks each do foo()*

- generally more explicit

  \+ provides more generality and control

  − more opportunities for problems (deadlock, livelock, …)

- two flavors of task parallelism:
  - *"may":* would work correctly even if multiple tasks were not used
    - e.g., tree search ("parallel")
  - *"must":* multiple tasks are required for correctness
    - e.g., producer/consumer ("concurrent")

Tasks/Threads in Pthreads

- As we've used Pthreads, task == thread
  - created thread

- Alternatively, could have each thread run some sort of "work manager" function rather than a "task"
  - e.g., "wait until a task becomes available… then run it"
    - could implement using bounded buffers of tasks
  - more complicated to code up
  + amortizes overhead of creating/destroying threads

Tasks/Threads in Chapel

- Chapel has multiple tasking layers
  - Each has its own implementation and policies
  - Default layer (CHPL_THREADS = "fifo"):
    - program with 1 thread running main()
    - new thread created for each new task…
      …unless a thread is sitting around bored in the pool… see below
      …or there aren't enough resources to create one
      …or we hit the user specified limit (numThreadsPerLocale)
      in which case, the task is put into a task pool for execution later
    - each thread runs its task to completion
      - task can also help with its cobegin/coforall tasks ("nothing else to do")
    - upon completion, runs an unclaimed task if one exists
    - otherwise, enters thread pool waiting for more tasks to show up

Tasks/Threads in Chapel

- Chapel has multiple tasking layers
  - each has its own implementation and policies
  - Most other layers (qthreads, massivethreads, nanox)
    - primarily utilize user-level lightweight threading
    - create # pthreads equal to # cores (or user-specified value)
    - each pthread multiplexes between multiple tasks
      - typically switches on blocking events like sync var reads/writes
      - sometimes switches on long-latency events like communication
  - Also a HW multithreading layer (mta)
    - map each task to its own HW thread context (~128 per node)
    - HW switches between tasks

For more info: doc/README.tasks

Tasks/Threads and Virtualization

- In *any* parallel programming environment, whenever # tasks > # cores, something must give
  - OS can multiplex between system-level threads
  - runtime can multiplex tasks/user-level threads over system threads
  - tasks can stall and wait for resources to become available

- Attention to these issues can be crucial to obtaining top performance

How Many Tasks Should I Use?

- It depends... (on your algorithm and architecture)
  - For many problems # tasks == # cores can be ideal
    - maximize use of HW without oversubscribing
    - a CPU-centric view of computation
  - # tasks > # cores can be useful…
    - if algorithm inherently wants to use many distinct tasks
    - as a task-driven way of doing dynamic load balancing
    - to hide memory latencies by switching between tasks (?)
  - If thrashing memory, maybe # tasks < # cores is better?

Data Parallelism

Task vs. Data Parallelism

Data Parallelism:

- expressed in terms of a data set that drives the parallelism
  - "data set" = typically an array, data structure, or set of indices

  e.g., forall i in 1..n do … *// for all integers/indices 1 thru n do…*

  e.g., forall a in A do … *// for all elements in array A do…*
- generally more implicit

  + a simpler concept, easier for programmers to grasp

  + abstracts details of implementation to some lower level SW/HW

  − not as general as task parallelism
    - but an important common case to support and optimize for
    - can typically be thought of as a special case of "may" parallelism

(of course, in practice, data parallelism is implemented using tasks; and in practice most task parallel programs operate on some sort of data, so the line between the two can be a little fuzzy)

Example of Task- vs. Data-Parallelism

- Reductions
  - collective *("members contribute")* == a task-parallel reduction

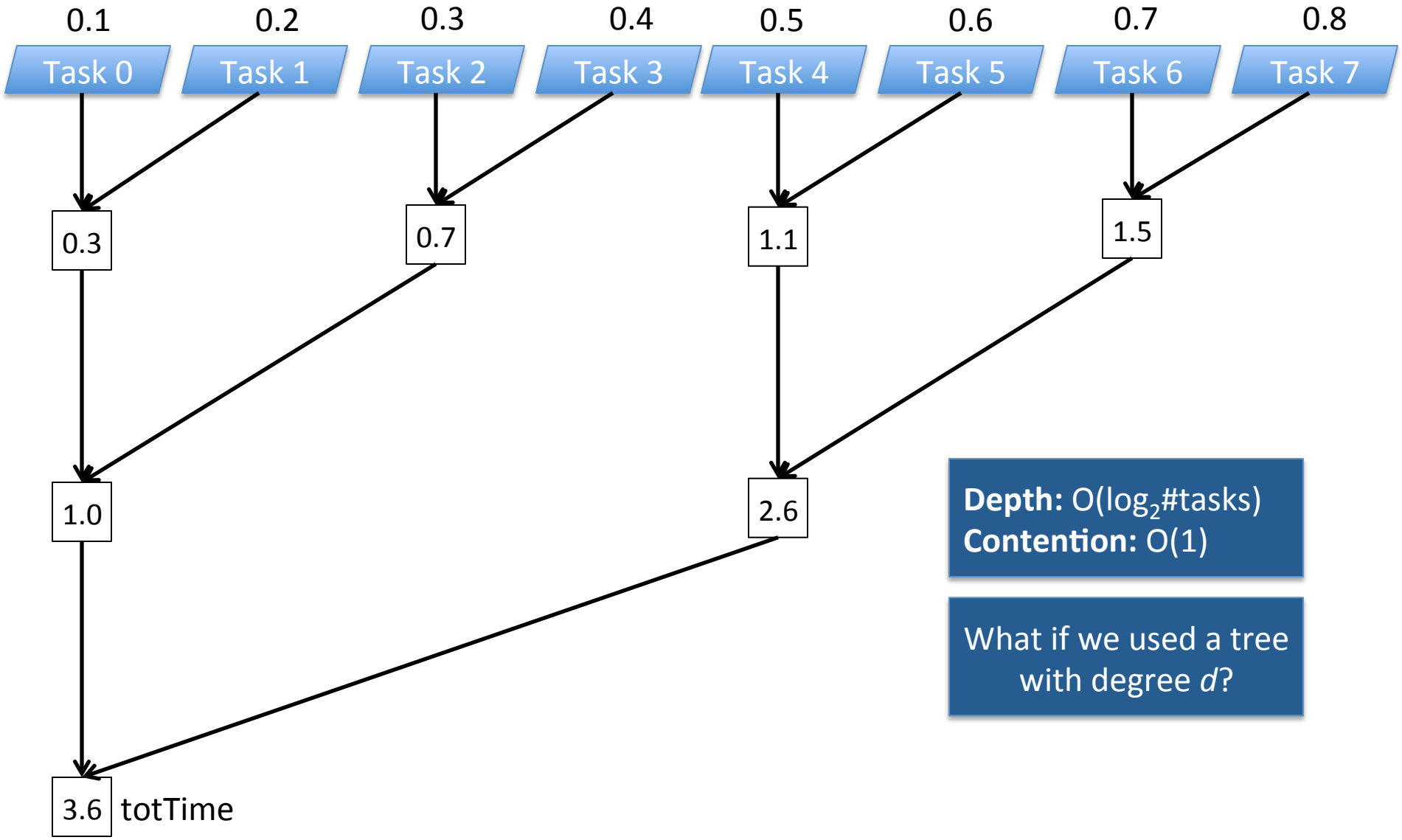
```
coforall tid in 0..#numTasks {
  const myContribution = doSomeWork(…);
  const total = sumReduceAll(myContribution);
}
```

  - global-view *("holistic")* == a data-parallel reduction

```
const total = + reduce A;  // sum A's elements
```

Speaking of reductions… where were we?

Use a *Reduction*



Depth: O($\log_2$#tasks)
Contention: O(1)

What if we used a tree with degree *d*?

Two Flavors of Reductions

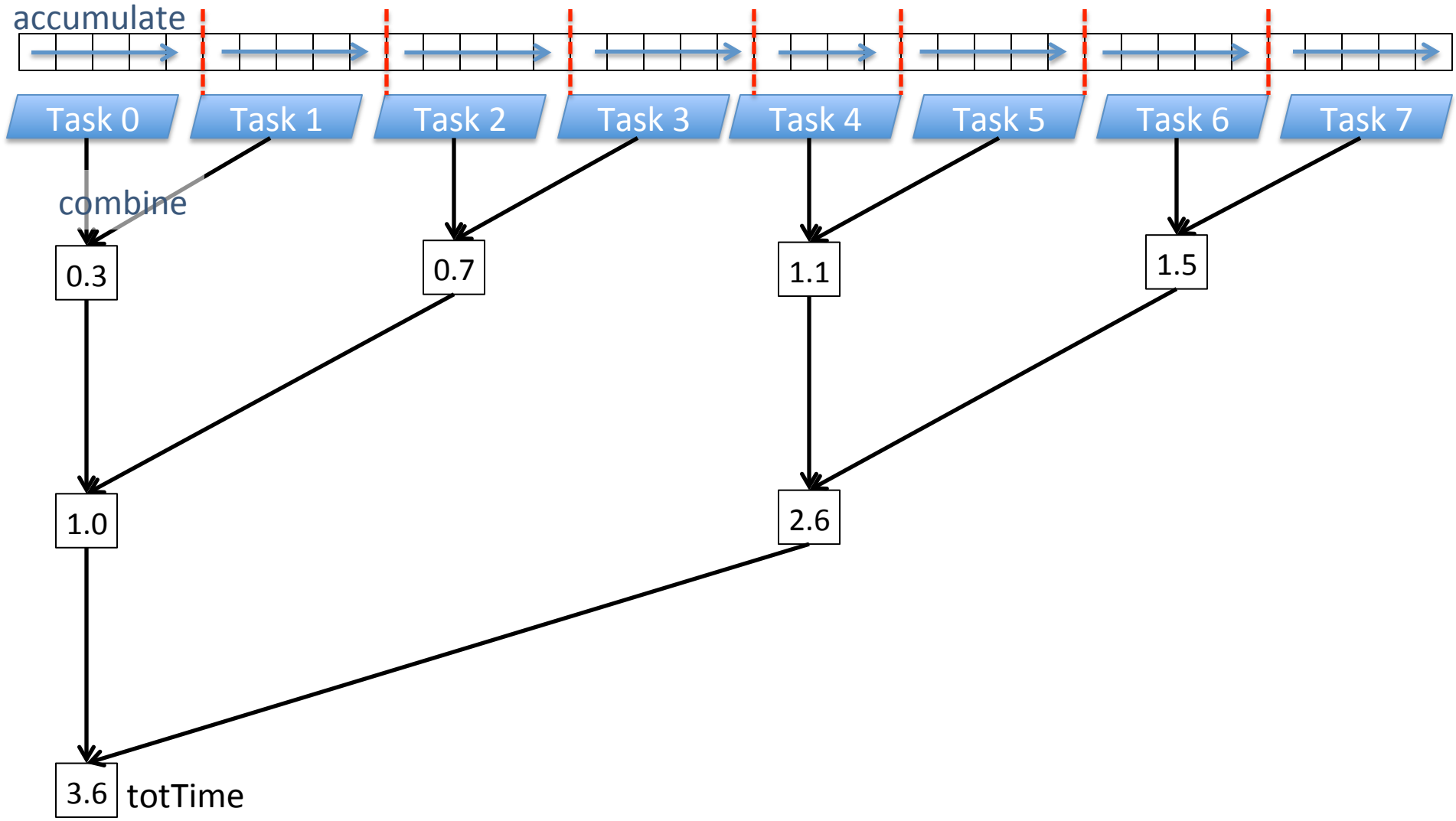
- collective *("members contribute")*

```
create tasks…
  const myContribution = doSomeWork(…);
  const total = sumReduceAll(myContribution);
join tasks…
```

- global-view *("holistic")*

```
const total = + reduce A;  // sum A's elements
```

Reductions on Arrays



Reductions on Multidimensional Arrays

1	2	3
4	5	6
7	8	9

- Full/Complete Reduction: collapse array to scalar

  + reduce = 45 min reduce = 1

- Partial Reduction: collapse a subset of array dims
  - reduce along rows:

    + → | 6 |
        | 15 |
        | 24 |

    min → | 1 |
          | 4 |
          | 7 |

  - reduce along cols: | 12 | 15 | 18 | and | 1 | 2 | 3 |

(higher-D arrays can be reduced to planes or vectors or …)

Reduction Operators in Chapel

Built-in

- +, *, &&, ||, &, |, ^, min, max
- minloc, maxloc
  - Takes a zipped pair of values and indices
  - Generates a tuple of the min/max value and its index

User-defined

- Defined via a class that implements a standard interface
- Compiler generates code that calls these methods

Defining Parallel Reductions

- What's required?
- More generally (result type != input type, or state is required)
  - An identity element
    - What should we initialize our state to?
  - An *accumulator* function
    - Combines an input value and a state value, creating a state value
  - A *combiner* function
    - Combines two state values, creates a state value
  - A *result* function
    - Transforms a state value into an answer

Discuss Map-Reduce Paper Here?

Scans: A Related Operation to Reductions

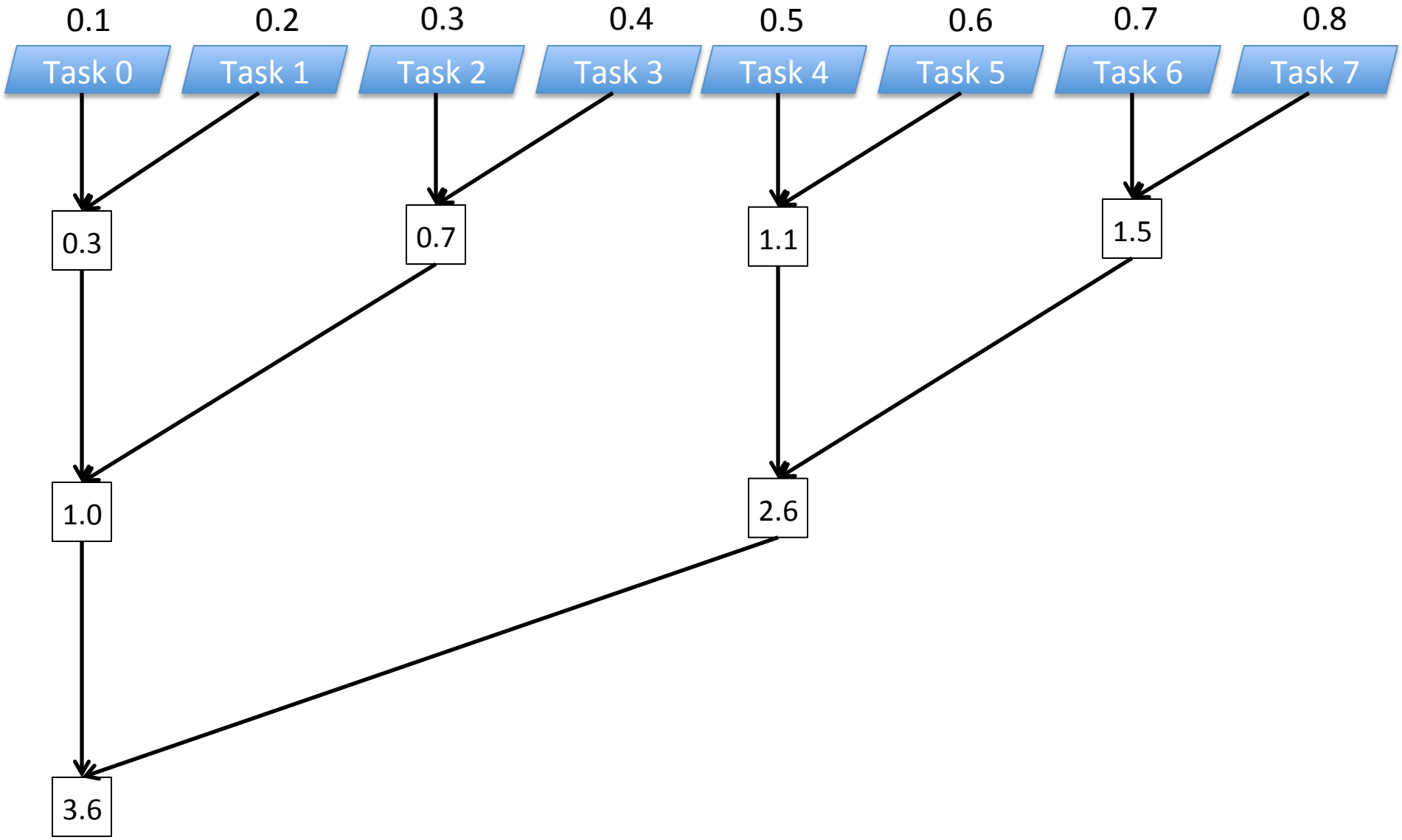
- Syntax

```
scan-expr:
  scan-op scan iterator-expr
```

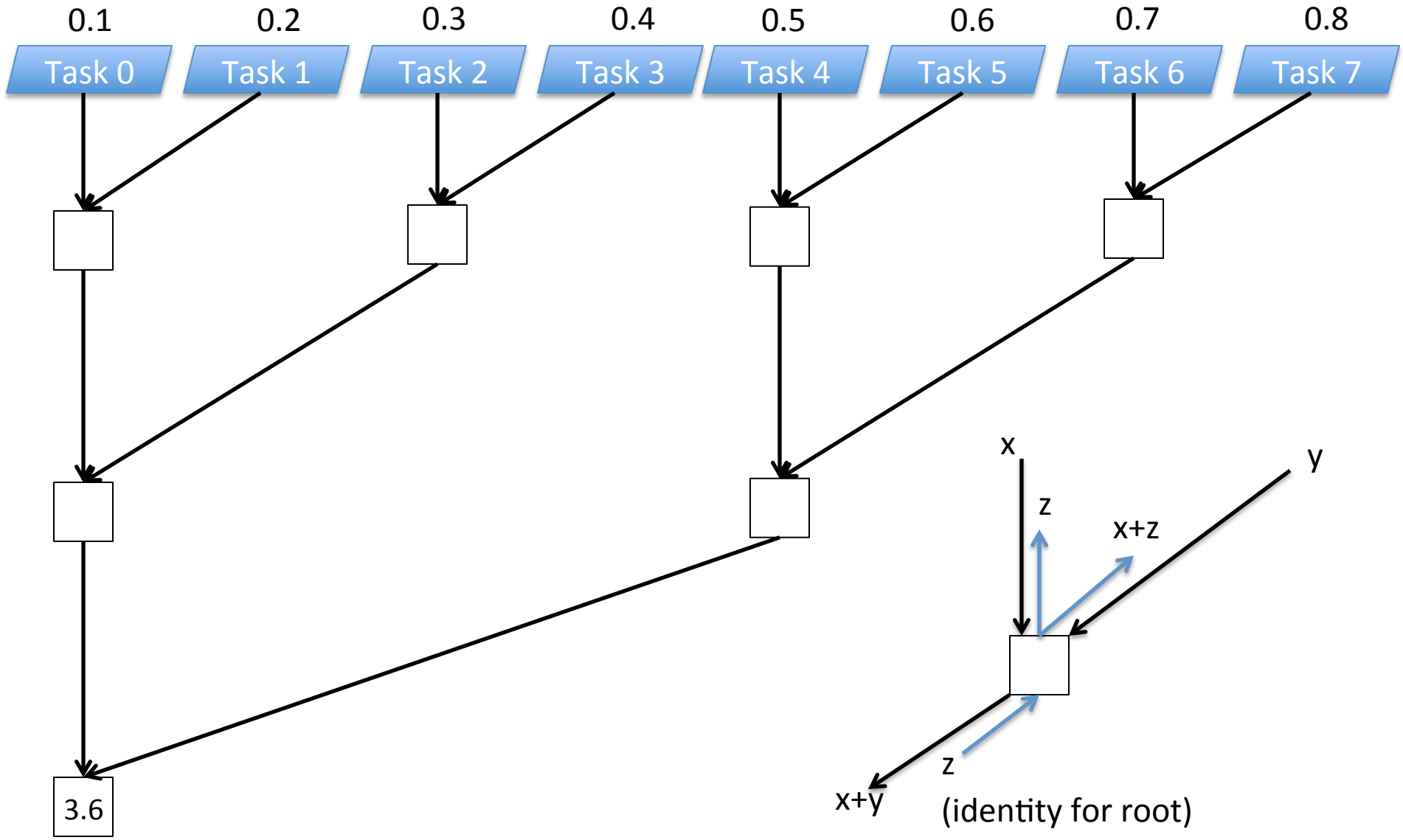
- Semantics
  - Computes parallel prefix over values using *scan-op*
    - Like a reduction, but leaves intermediate values behind
  - *Scan-op* may be any *reduce-op*
- Examples

```
var A, B, C: [1..5] int;
A = 1;                          // A:  1  1  1  1  1
B = + scan A;                   // B:  1  2  3  4  5
B[3] = -B[3];                   // B:  1  2 -3  4  5
C = min scan B;                 // C:  1  1 -3 -3 -3
```

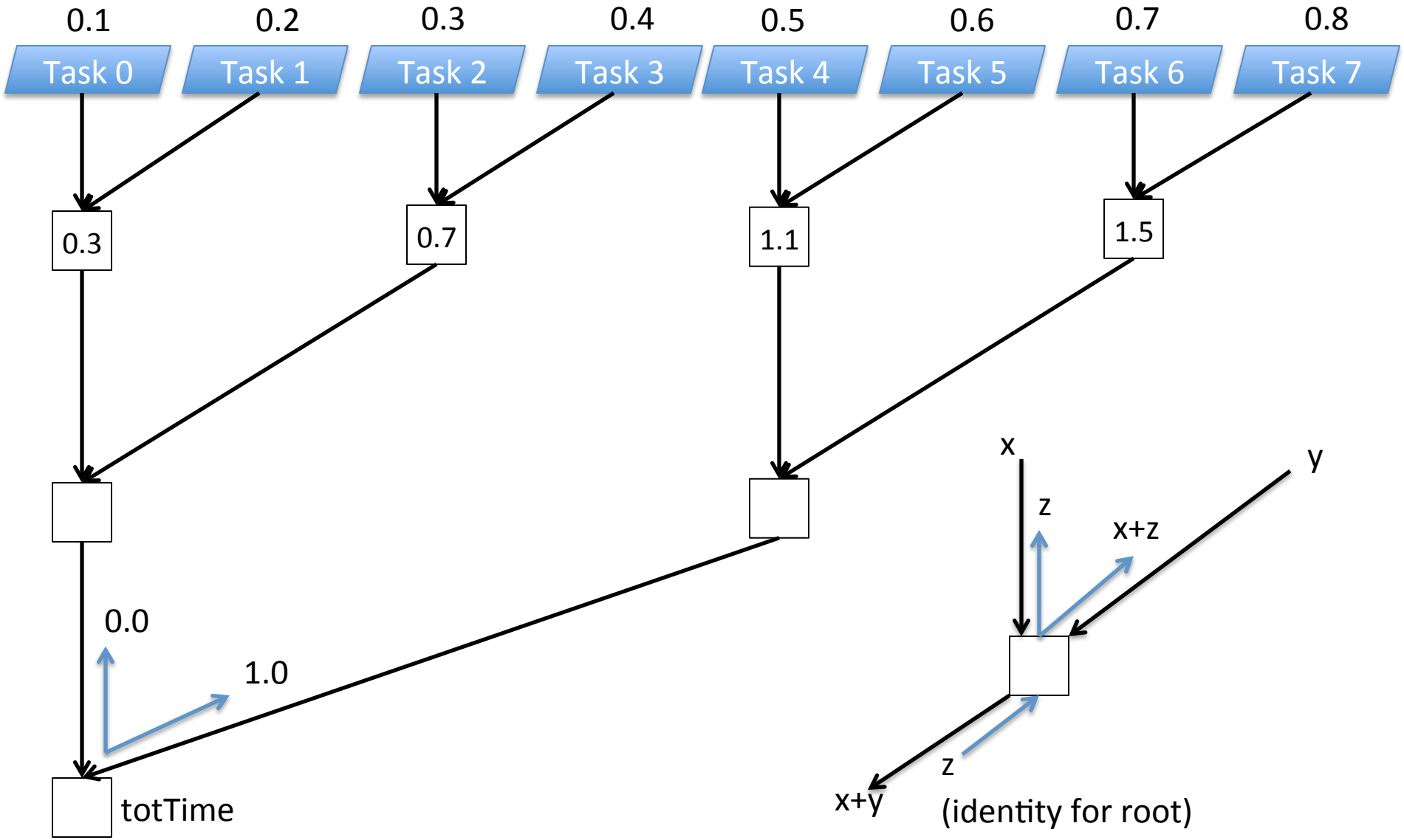
Scans, Step 1: Compute Reduction



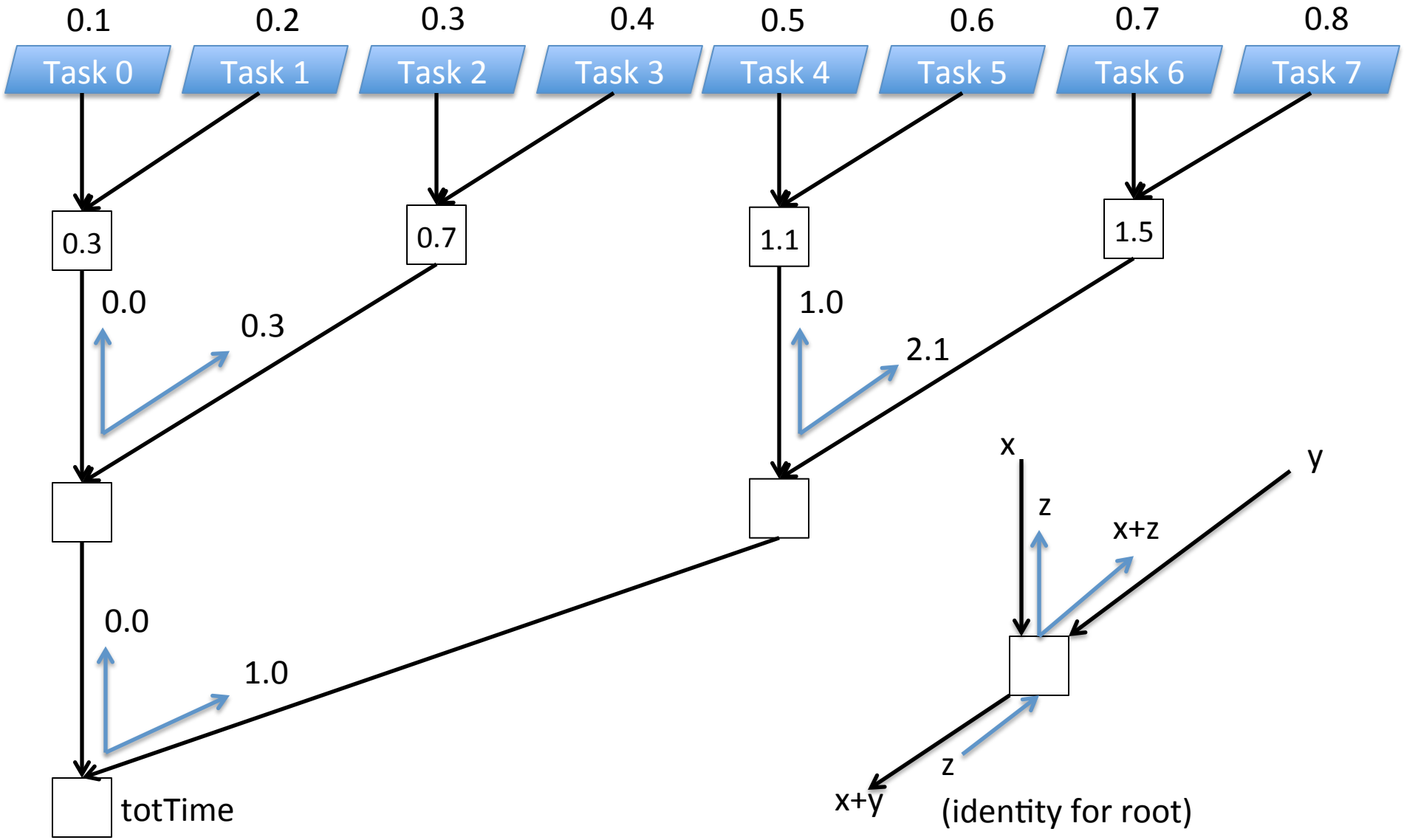
Scans, Step 2: Propagate Back



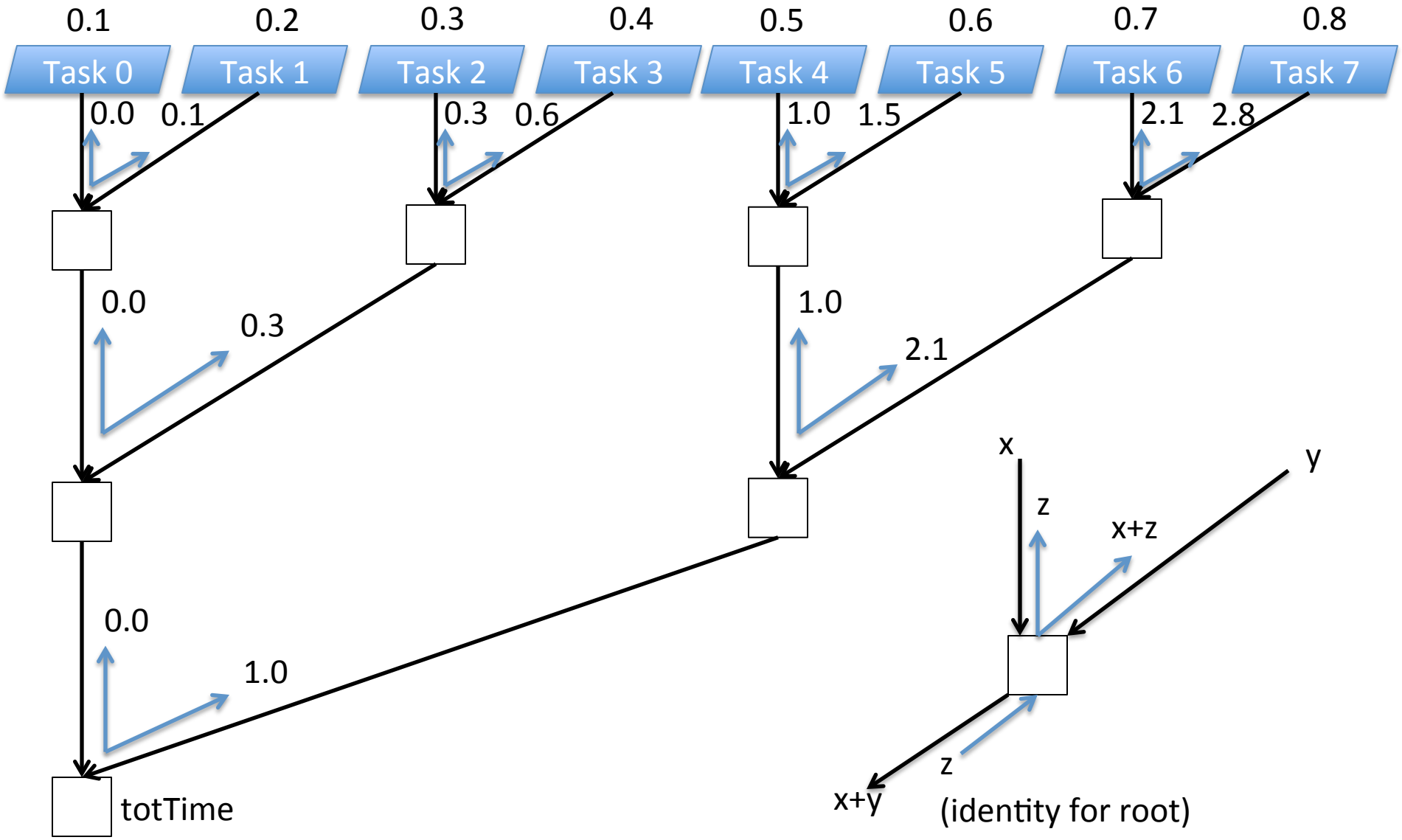
Scans, Step 2: Propagate Back



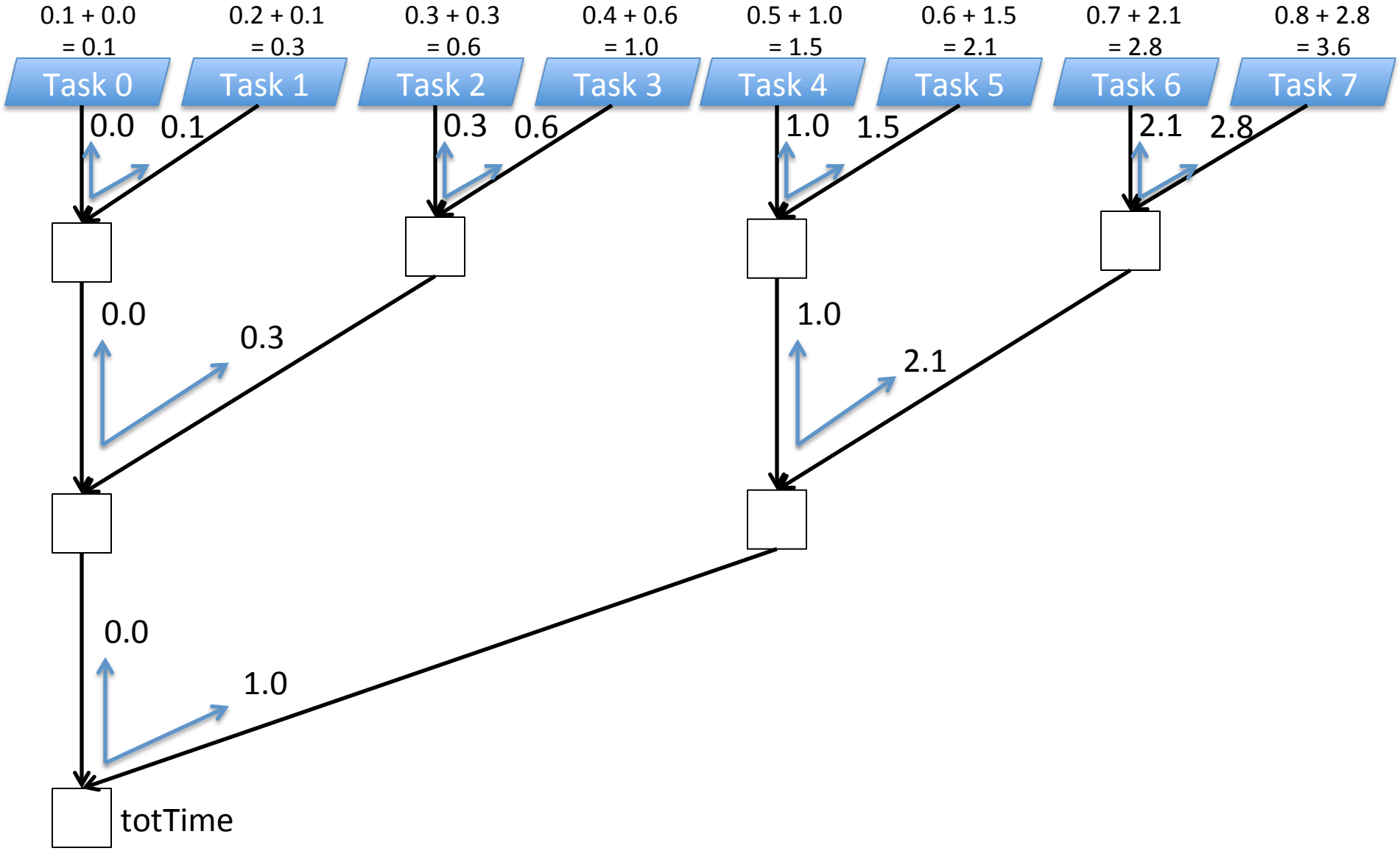
Scans, Step 2: Propagate Back



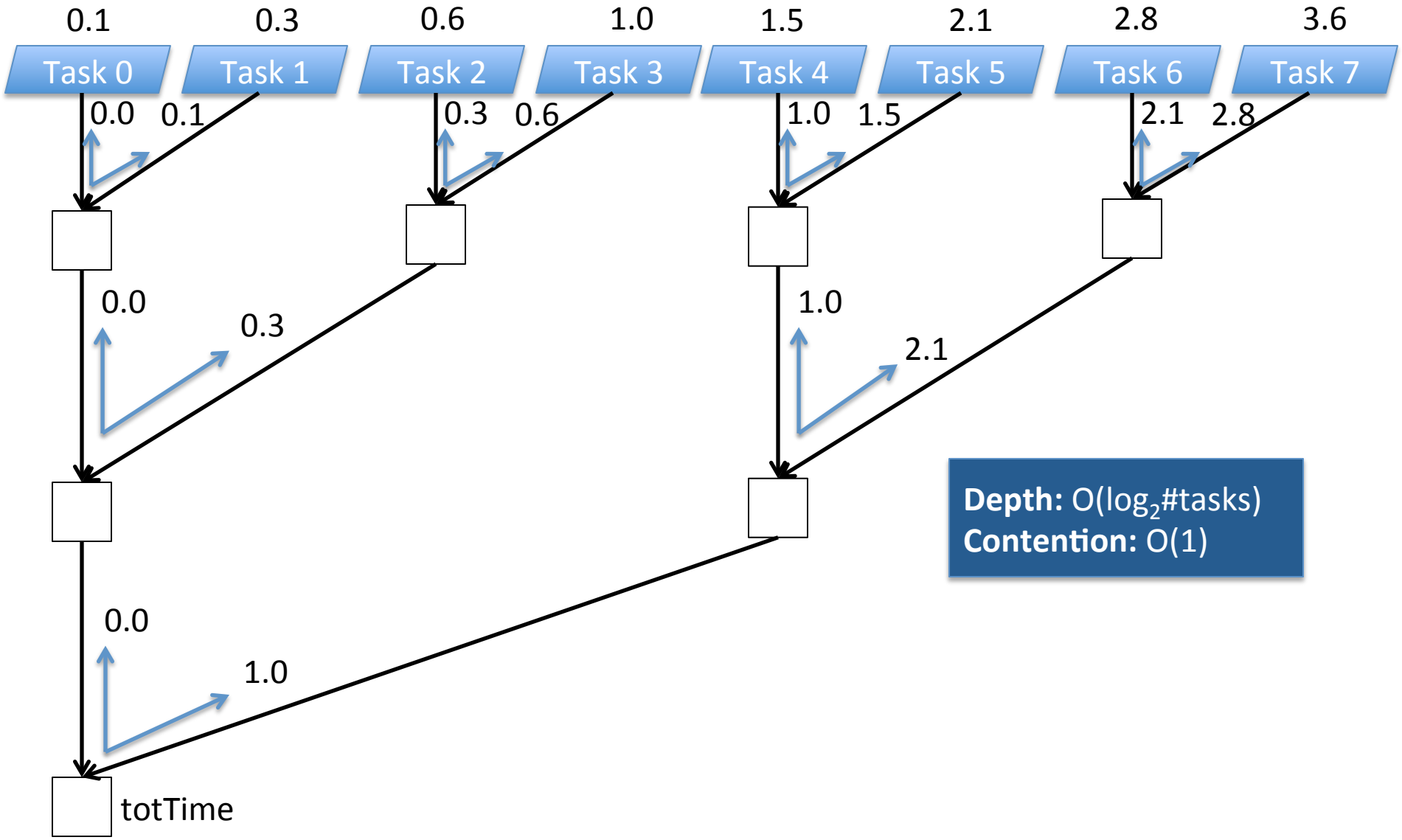
Scans, Step 2: Propagate Back



Scans, Step 3: Update Local Values



Scans, Step 4: Done



Scan: When would I ever use this?

Scan: When would I ever use this?

Problem: Have p tasks write data to a file in parallel

Trivial Case: Binary file (~~embarrassingly~~ pleasingly! parallel)

- Each task can trivially compute where its data should go:
  1) seek to file offset: sizeof(type) * myTaskID
  2) write my data

More Interesting Case: Text file

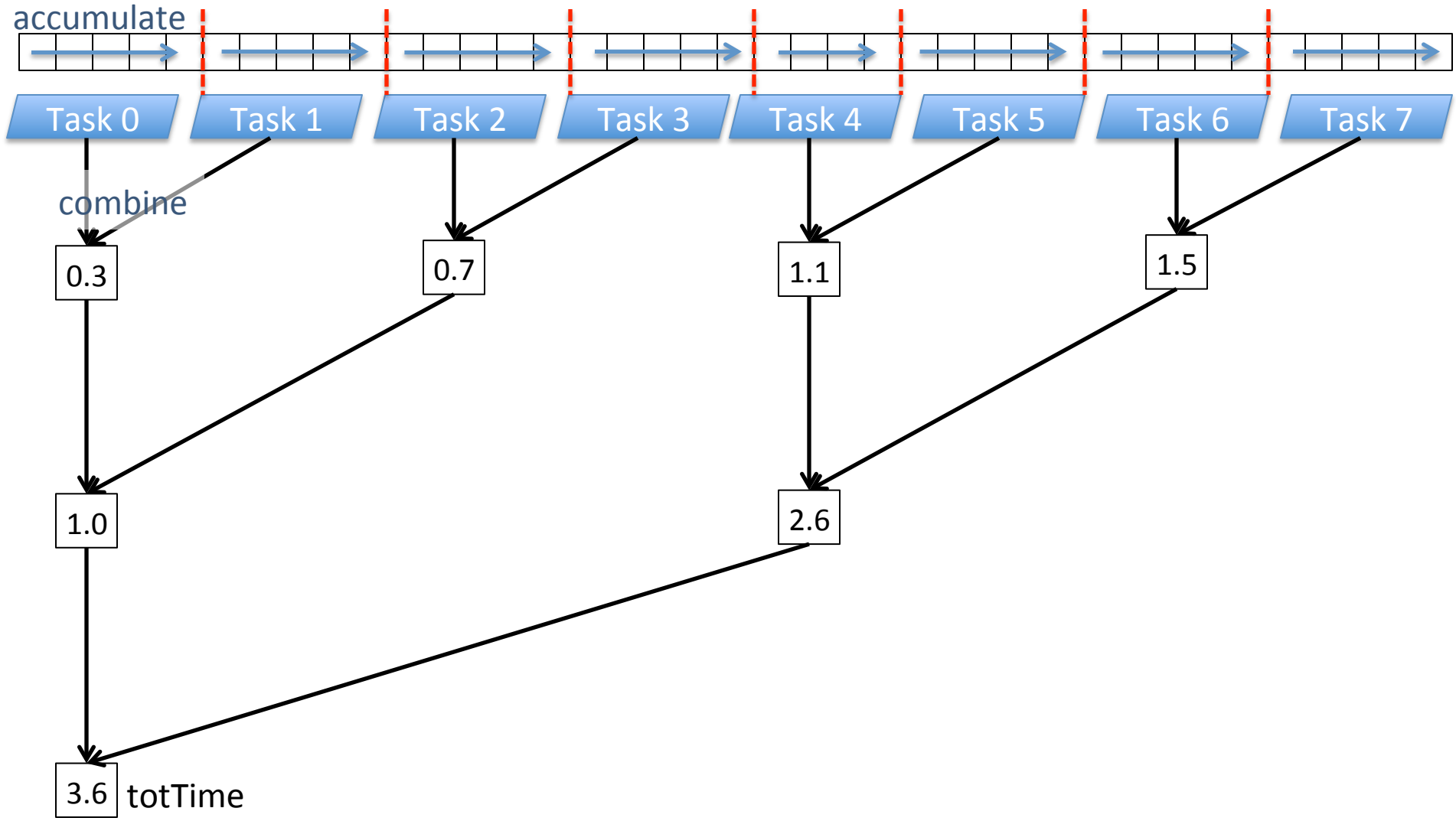
- Number of characters required per value may vary greatly
- So each task should:
  1. compute # of characters required to print my value + ' '
  2. compute a sum-scan of the offsets
  3. seek to file offset corresponding to my result value
  4. write my data

Inclusive vs. Exclusive Scans

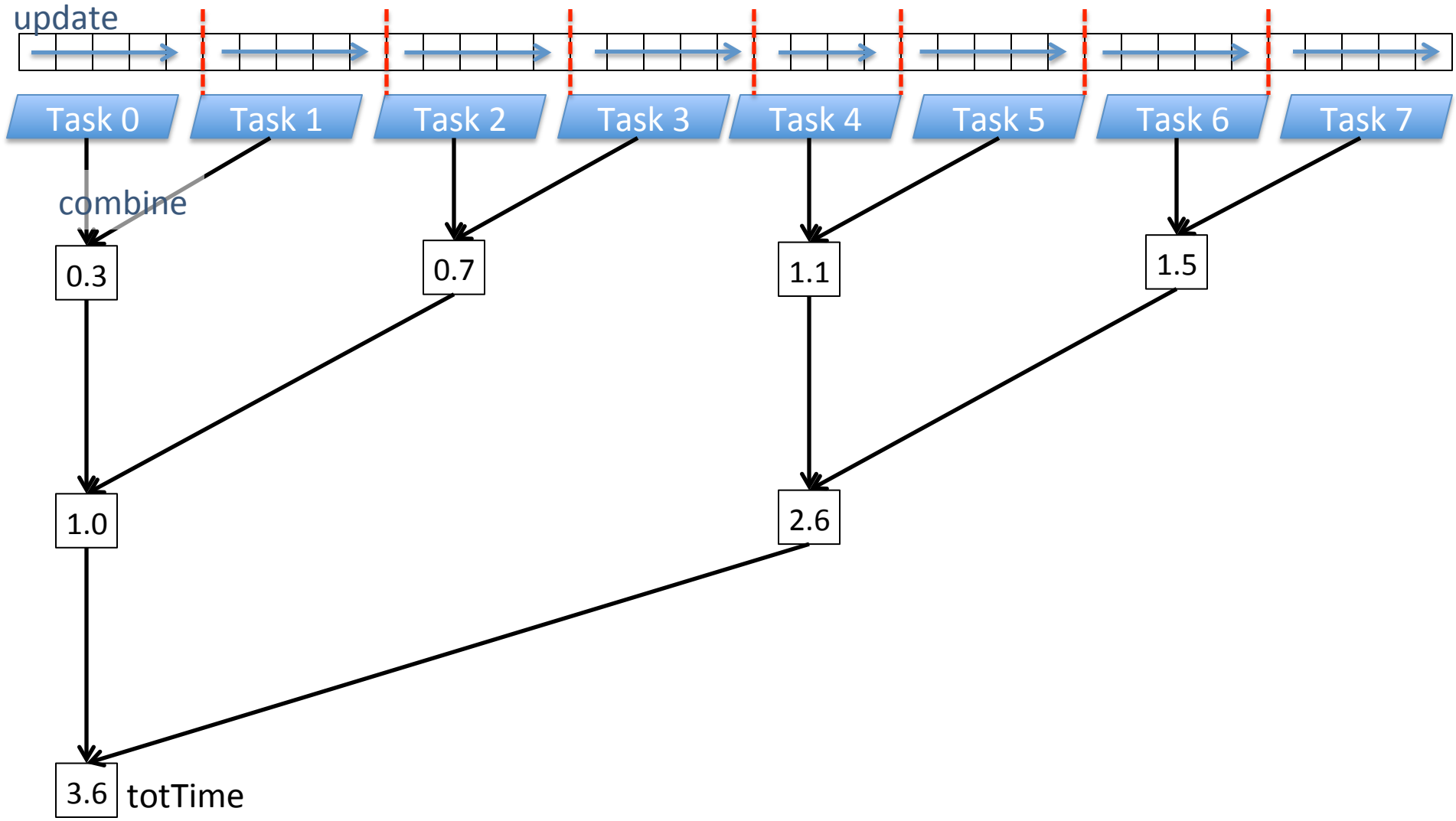
- Should the original item affect its result or not?
  - e.g., + **scan** [1, 1, 1, 1, 1, 1, 1, 1]
  - inclusive: [1, 2, 3, 4, 5, 6, 7, 8]
  - exclusive: [0, 1, 2, 3, 4, 5, 6, 7]

- Different scenarios may want different semantics
- Note: given exclusive and input, inclusive can be computed

Scans on Arrays: Step 0: Accumulate



Scans on Arrays: Step 3': Update *all* Elements



Scans on Multidimensional Arrays

1	1	1
1	1	1
1	1	1

- Partial Scan: scan a subset of dims in given direction
  - + scan along rows, L->R:

1	2	3
1	2	3
1	2	3

along cols, B -> T:

3	3	3
2	2	2
1	1	1

- Full/Complete Scan: thread through dimensions
  - + scan in Row-Major Order:

1	2	3
4	5	6
7	8	9

Barrier Synchronization ("Barrier")

Barrier: All participating tasks must reach barrier before any may pass

…create tasks… {

```
   foo();
   barrier();
   bar();
```

}

Rough analogy: Barrier:Task Control Flow :: Fence:Memory Ops

Data Parallelism in Chapel

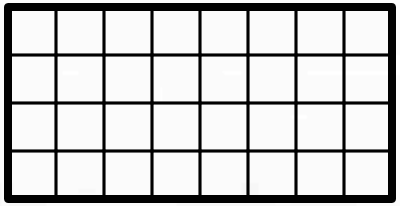
Domains

Domain: A first-class index set

- A fundamental Chapel concept for data parallelism
- Domains may optionally be distributed

Sample Domains

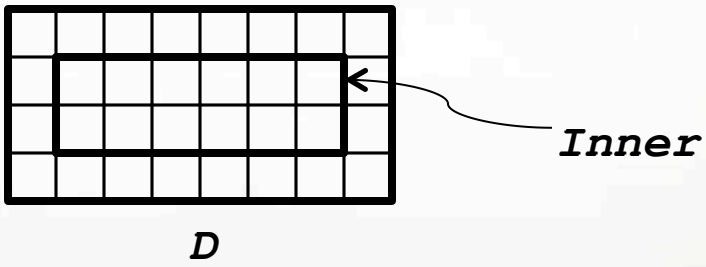
```
config const m = 4, n = 8;

var D: domain(2) = {1..m, 1..n};
```



D

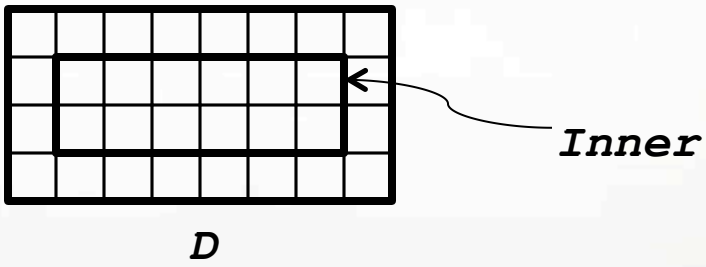
Sample Domains

```
config const m = 4, n = 8;

var D: domain(2) = {1..m, 1..n};

var Inner: subdomain(D) = {2..m-1, 2..n-1};
```



Sample Domains (Using Inferred Types)

```
config const m = 4, n = 8;

var D = {1..m, 1..n};

var Inner = D[2..m-1, 2..n-1];
```



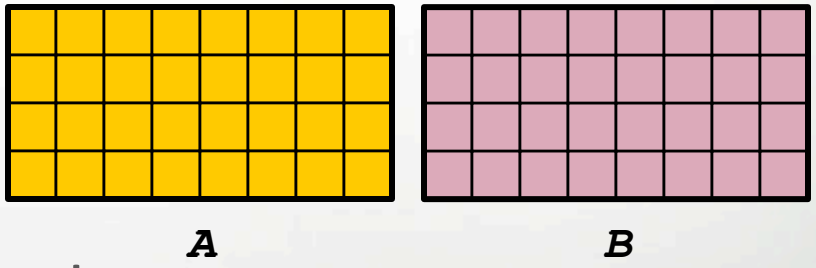
Domains Define Arrays

- Syntax

```
array-type:
  [ domain-expr ] elt-type
```

- Semantics
  - Stores an *elt-type* for each index in *domain-expr*
- Example

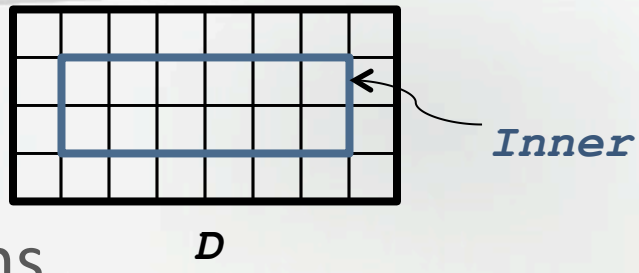
```
var A, B: [D] real;
```



- Earlier example, revisited

```
var A: [1..3, 1..5] real; // [1..3, 1..5] creates an
                          // anonymous domain
```

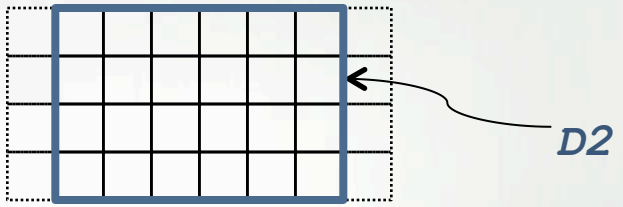
Domain Algebra



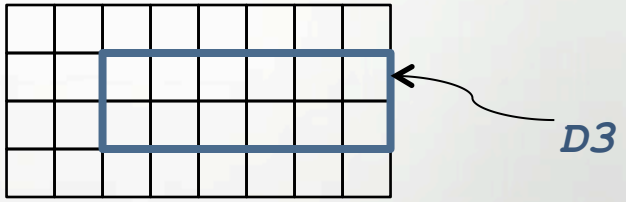
Domain values support...

- Methods for creating new domains

```
var D2 = Inner.expand(1,0);
```

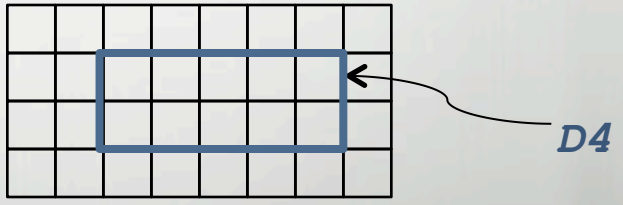



```
var D3 = Inner.translate(0,1);
```



- Intersection via Slicing

```
var D4 = D2[D3];
```

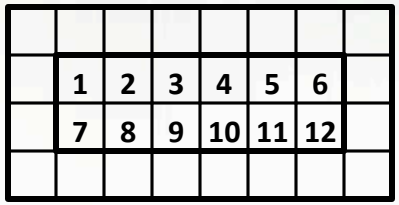


- Range operators (e.g., `#`, `by`, `align`)

Domain Iteration

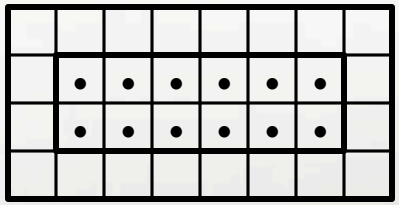
- For loops
  - Execute loop body once per domain index, serially

```
for i in Inner do ...
```



- Forall loops
  - Executes loop body once per domain index, in parallel
  - Loop must be *serializable* (executable by one task)

```
forall i in Inner do ...
```



- Loop variables take on `const` domain index values

Other Forall Loops

Forall loops also support...

- A shorthand notation:

```
[(i,j) in D] A[i,j] = i + j/10.0;
```

- Expression-based forms:

```
A = forall (i,j) in D do i + j/10.0;
```

```
A = [(i,j) in D] i + j/10.0;
```

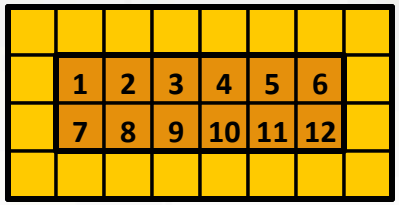
1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

A

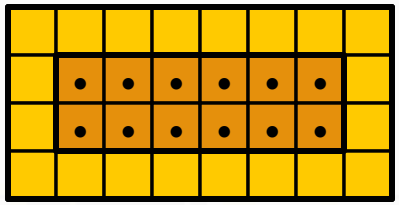
Array Iteration

- Array expressions also support for and forall loops

```
for a in A[Inner] do ...
```




```
forall a in A[Inner] do ...
```



- Array loop indices refer to array elements (can be modified)

```
forall (a, (i,j)) in zip(A, D) do a = i + j/10.0;
```

Note that forall loops support zippered iteration, like for-loops

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

Comparison of Loops: For, Forall, and Coforall

For loops: executed using one task

- use when a loop must be executed serially
- or when one task is sufficient for performance

Forall loops: typically executed using 1 < #tasks << #iters

- use when a loop *should* be executed in parallel…
- …but *can* legally be executed serially
- use when desired # tasks << # of iterations

Coforall loops: executed using a task per iteration

- use when the loop iterations *must* be executed in parallel
- use when you want # tasks == # of iterations
- use when each iteration has substantial work

How Much Parallelism?

By default*, controlled by three config variables:

--dataParTasksPerLocale=#

- Specify # of tasks to execute forall loops
- *Current Default:* number of processor cores

--dataParIgnoreRunningTasks=[true|false]

- If false, reduce # of forall tasks by # of running tasks
- *Current Default:* true

--dataParMinGranularity=#

- If > 0, reduce # of forall tasks if any task has fewer iterations
- *Current Default:* 1

*Default values can be overridden for specific domains/arrays

Promoting Functions and Operators

Functions/operators expecting scalars can also take...

…arrays, causing each element to be passed in

```
sin(A)
2*A
```

≈

```
forall a in A do sin(a)
forall a in A do 2*a
```

…domains, causing each index to be passed in

```
foo(Inner)
```

≈

```
forall i in Inner do foo(i)
```

Multiple arguments promote using zippered iteration

```
pow(A, B)
```

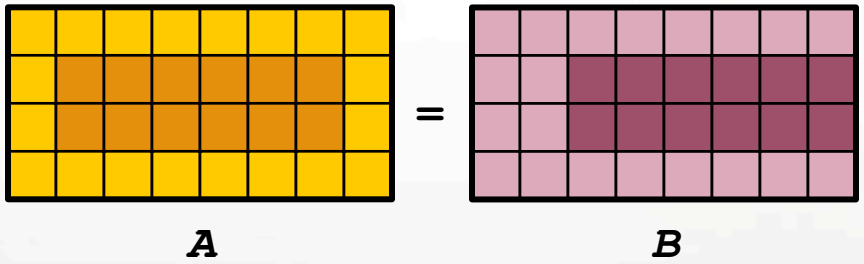
≈

```
forall (a,b) in zip(A,B) do pow(a,b)
```

Sub-Arrays/Array Slicing

Indexing into arrays with domain values results in a sub-array expression (an “array slice”)

```
A[Inner] = B[Inner.translate(0,1)];
```

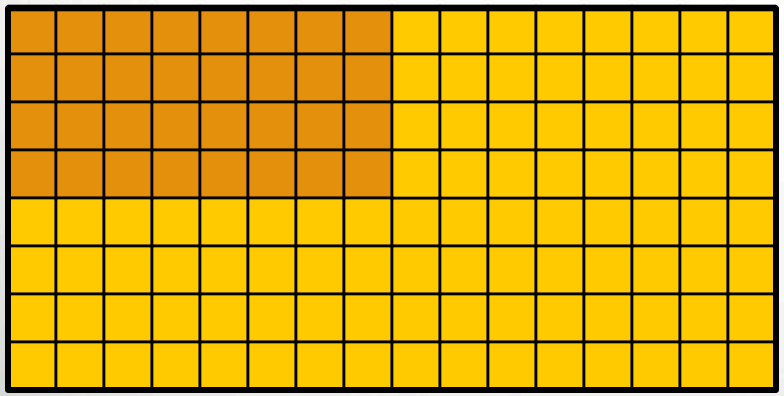


Array Reallocation

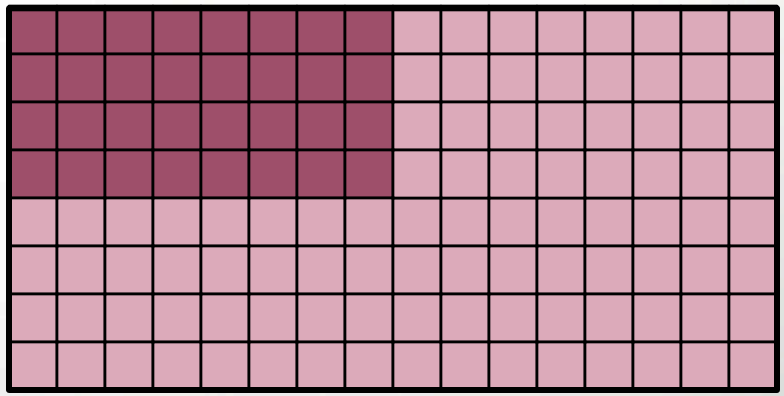
Reassigning a domain logically reallocates its arrays

- array values are preserved for common indices

```
D = {1..2*m, 1..2*n};
```



A

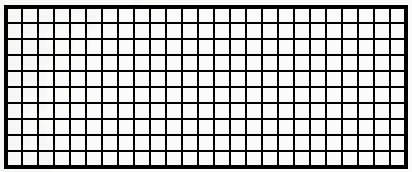


B

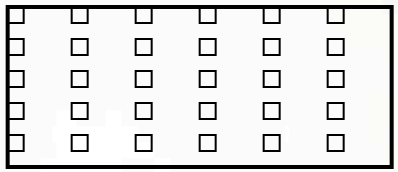
Chapel Domain Types

Chapel supports several domain types…

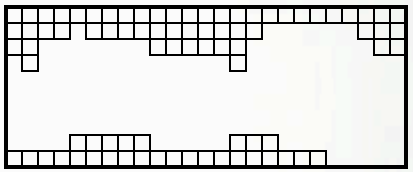
```
var OceanSpace = {0..#lat, 0..#long},
    AirSpace = OceanSpace by (2,4),
    IceSpace: sparse subdomain(OceanSpace) = genCaps();
```



dense

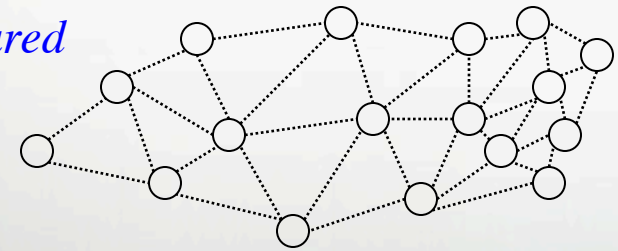


strided

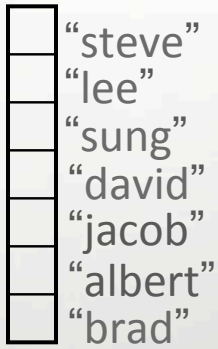


sparse

unstructured



associative

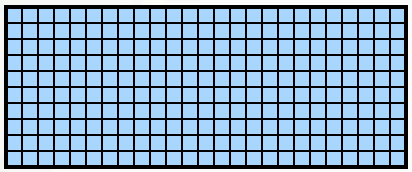
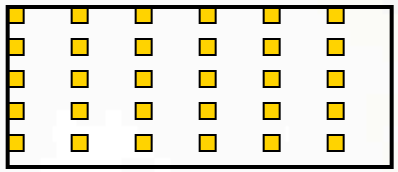
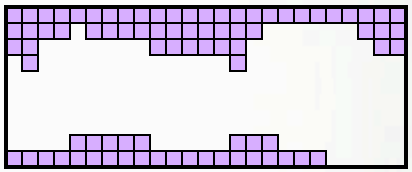


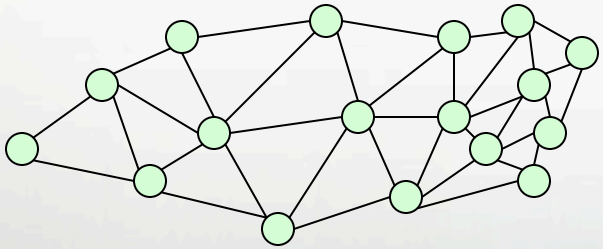

```
var Vertices: domain(opaque) = …,  People: domain(string) = …;
```

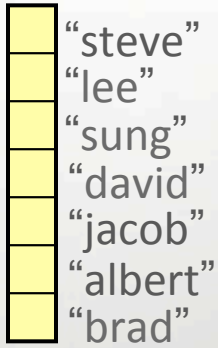
Chapel Array Types

All domain types can be used to declare arrays…

```
var Ocean: [OceanSpace] real,
    Air: [AirSpace] real,
    IceCaps[IceSpace] real;
```



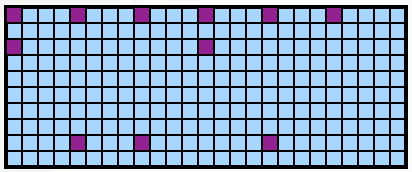


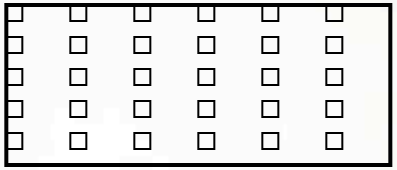

```
var Weight: [Vertices] real,        Age: [People] int;
```

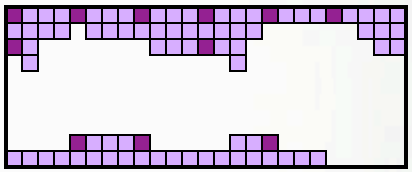
Iteration

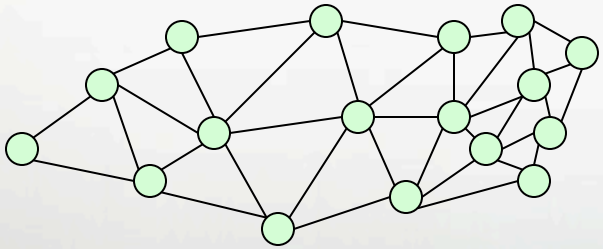
…to iterate over index sets…

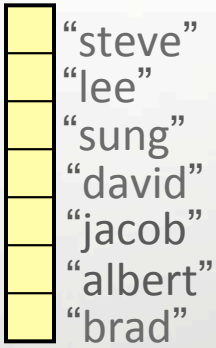
```
forall ij in AirSpace do
  Ocean[ij] += IceCaps[ij];
```











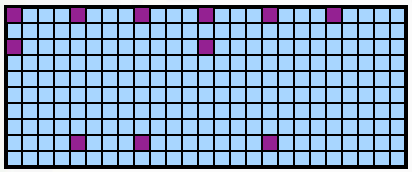

```
forall v in Vertices do
  Weight[v] = numEdges[v];
```

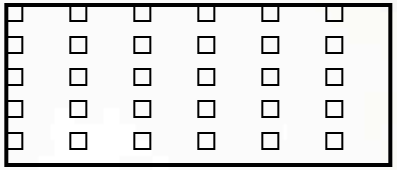
```
forall p in People do
  Age[p] += 1;
```

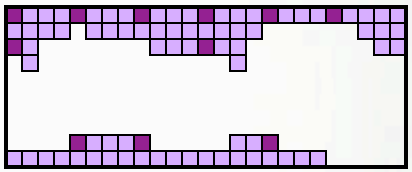
Slicing

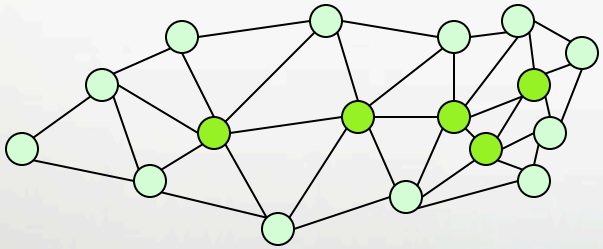
…to slice arrays…

```
Ocean[AirSpace] += IceCaps[AirSpace];
```









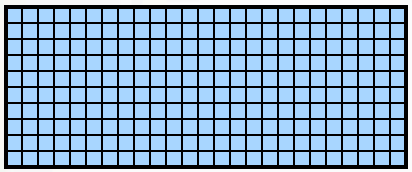
…Vertices[Interior]…

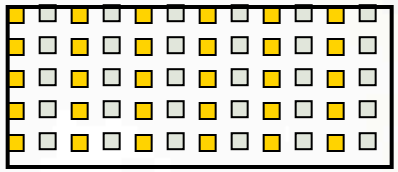
…People[Interns]…

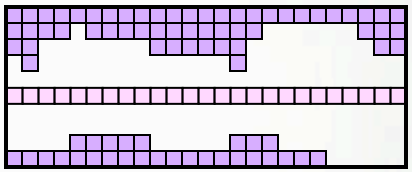
Reallocation

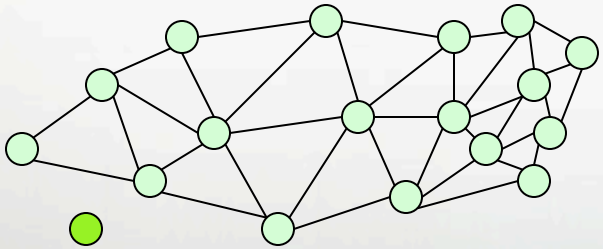
…and to reallocate arrays

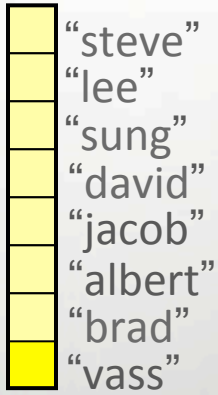
```
AirSpace = OceanSpace by (2,2);
IceSpace += genEquator();
```









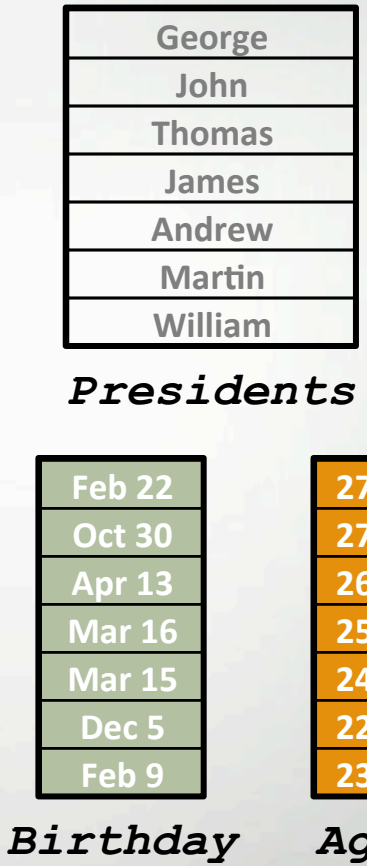



```
newnode = Vertices.create();
```

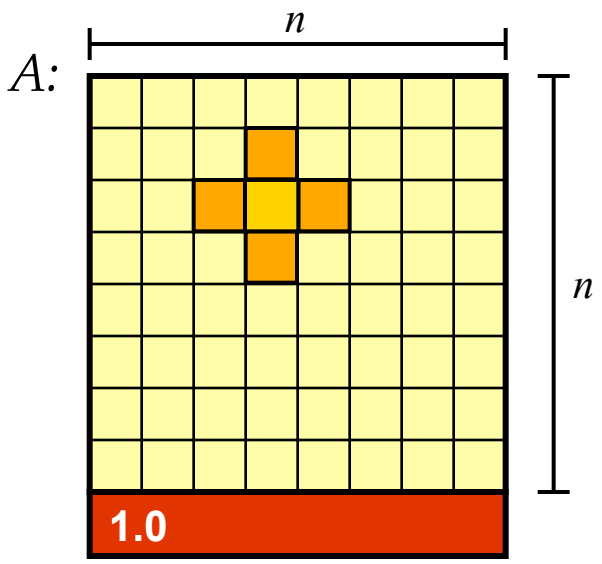
```
People += "vass";
```

Associative Domains and Arrays by Example

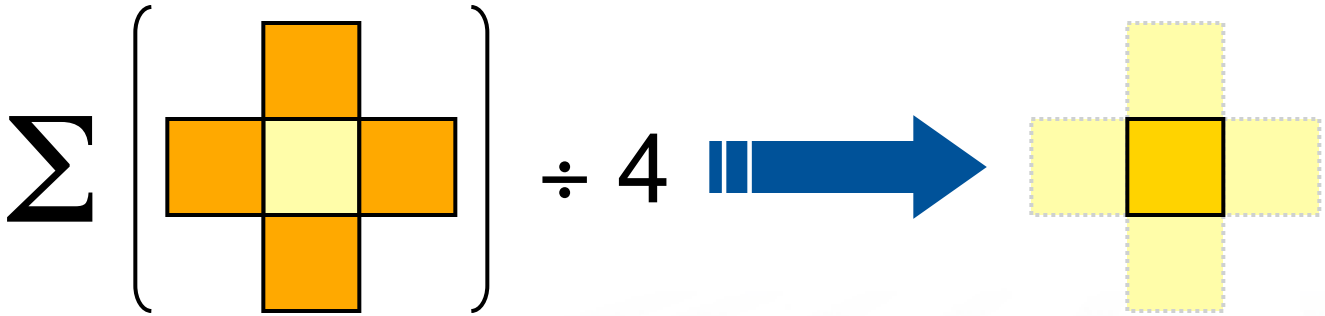
```
var Presidents: domain(string) =
      {"George", "John", "Thomas",
       "James", "Andrew", "Martin"};

Presidents += "William";


var Age: [Presidents] int,
    Birthday: [Presidents] string;

Birthday["George"] = "Feb 22";


forall president in President do
  if Birthday[president] == today then
    Age[president] += 1;
```



Jacobi Iteration in Pictures



repeat until max change < ε



Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[Las

do {
  [(i

  con
  A[D
} whi

writeln(A);
```

Declare program parameters

const ⇒ can't change values after initialization

config ⇒ can be set on executable command-line

```
prompt> jacobi --n=10000 --epsilon=0.0001
```

note that no types are given; inferred from initializer

n ⇒ **default integer** (32 bits)

epsilon ⇒ **default real floating-point** (64 bits)

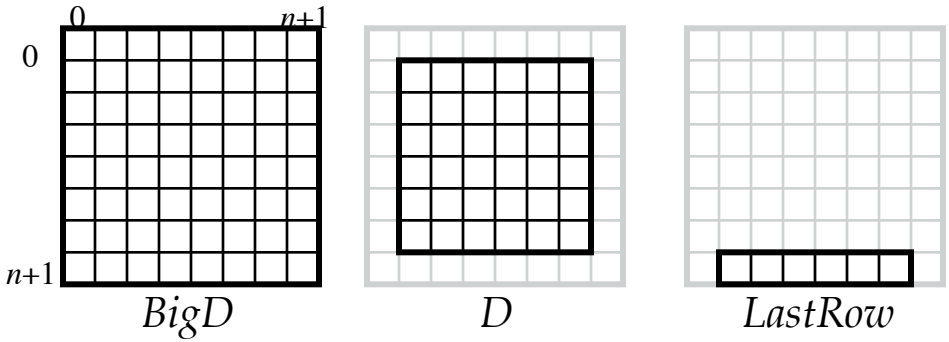
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
   LastRow: subdomain(BigD) = D.exterior(1,0);
```

Declare domains (first class index sets)

domain(2) ⇒ 2D arithmetic domain, indices are integer 2-tuples

subdomain(*P*) ⇒ a domain of the same type as *P* whose indices are guaranteed to be a subset of *P*'s



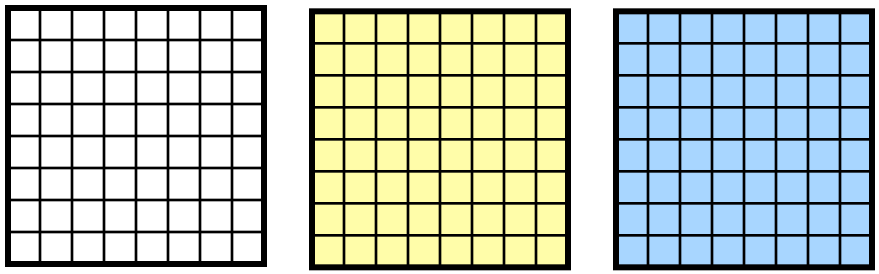
exterior ⇒ one of several built-in domain generators

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
```

Declare arrays

var ⇒ can be modified throughout its lifetime
: ***[BigD] T*** ⇒ array of size *BigD* with elements of type *T*
(no initializer) ⇒ values initialized to default value (0.0 for reals)

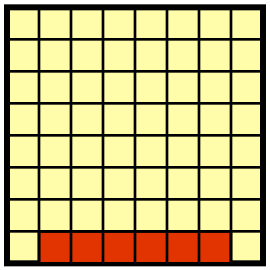


Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
```

Set Explicit Boundary Condition

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation



A

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;
```

Compute 5-point stencil

[(*i,j*) in *D*] ⇒ parallel forall expression over *D*'s indices, binding them to new variables *i* and *j*

```
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
```

Compute maximum change

***op* reduce** ⇒ collapse aggregate expression to scalar using *op*

Promotion: *abs()* and – are scalar operators, automatically promoted to work with array operands

```
do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
   LastRow: subdomain(BigD) = D.exterior(1,0);

var

A[La

do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard *do…while* loop construct

DARPA HPCS

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = {0..n+1, 0..n+1},
          D: subdomain(BigD) = {1..n, 1..n},
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp[i,j] = (A[i-1,j] + A[i+1,j]
                          + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Forall Loops: Lingering Questions

```
forall a in A do
  writeln("Here is an element of A: ", a);
```

- How many tasks will be used?
- How are iterations mapped to the tasks?

```
forall (a, i) in zip(A, 1..n) do
  a = i / 10.0;
```

Forall-loops may be zippered, like for-loops

- Corresponding iterations must match up
- But how does this work?

Array Indexing

- Arrays can be indexed using variables of their domain's index type (tuples) or lists of integers

```
var i = 1, j = 2;
var ij = (i,j);

A[ij] = 1.0;
A[i, j] = 2.0;
```

- Array indexing can use either parentheses or brackets

```
A(ij) = 3.0;
A(i, j) = 4.0;
```

Array Arguments and Aliases

- Arrays are passed by reference by default

```
proc zero(X: []) { X = 0; }

zero(A[Inner]);  // zeroes the inner values of A
```

- Formal array arguments can reindex actuals

```
proc f(X: [1..b,1..b]) { … } // X uses 1-based indices

f(A[lo..#b, lo..#b]);
```

- Array alias declarations provide similar functionality

```
var InnerA => A[Inner];
var InnerA1: [1..n-2,1..m-2] => A[2..n-1,2..m-1];
```

OpenMP

(switch to Alex Duran's slide deck here)

Using OpenMP

- Supported by gcc
  - but must use –fopenmp flag
  - OpenMP 3.1 supported in gcc 4.7 onwards
    - (the version that's available on our Fedora VM)
    - HW makes use of min/max reductions which are new as of v3.1

OpenMP Summary

- Lots of support for things we've done manually
  - parallel loops via block, cyclic, block-cyclic, dynamic schedules
  - collective reductions
  - critical sections (lock-protected code segments)
- Support for concepts that we've been using
  - creation of threads/tasks
  - locks
- Support for things we've talked about tonight
  - atomic operations
  - barriers

OpenMP Characterizations

- Relaxed memory consistency model
- *May*-style task parallelism

Lock-Free Programming (Atomic Computations)

Writing Deadlock-Free Lock Code

3) Use atomic operations

("atomic" in the sense of "indivisible", not "boom!")

Concept:

- never block
  - gets rid of deadlock issues
  - livelock can still be a potential issue in some cases
- instead, ensure no other task can see intermediate state
  - analogy to databases…

Two Forms of Atomic/Lock-Free Mechanisms

- General Atomic Statements (STM/HTM)
- Atomic Variables/Operations

Software Transactional Memory (STM)

Atomic

An *easier-to-use* and *harder-to-implement* primitive

```
void deposit(int x){
synchronized(this){
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

lock acquire/release

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

(behave as if)
no interleaved computation

Source: Dan Grossman, Software Transactions

So… Where are my atomics?

- Has not yet made it from research to production
- Challenges to adoption:
  - semantic questions/challenges
  - performance relative to locks
  - complete, production-grade implementation
- Two prevailing views:
  - STM is like GC in the 80's… en route
  - STM is unlikely to ever be adoptable

In the meantime…
Atomic Variables and Operations

Atomic Variables/Operators

Concept:

- supply special variable types
- with fixed, built-in set of atomic operators
- results in a code style called *lock-free programming*

Atomic Variables in Chapel

- Syntax

```
sync-type:
  atomic type
```

- Semantics:
  - Supports operations on variable atomically w.r.t. other tasks
  - Based on C/C++ atomic operations
  - Currently supported atomic types: ints, uints, reals

- Status note:
  - Passing by blank/default intent doesn't use 'ref' by default
    - makes local copy of procedure instead
    - workaround: use explicit `ref` intent

Atomic Methods: Reading and Writing

- `read():t` return current value
- `write(v:t)` store *v* as current value
- `exchange(v:t):t` store *v*, returning previous value
  - like read and write bundled together

- `waitFor(v:t)` wait until the stored value is *v*

- `testAndSet()` like *exchange(true)* for atomic bool
- `clear()` like *write(false)* for atomic bool

Atomic Methods: Simple Operations

- **`add(v:t)`** add *v* to the value atomically
- **`fetchAdd(v:t)`** same, and return sum

(also support for *sub*, *or*, *and*, *xor* operations)

- Example: Trivial barrier (supports one use only)

```
var count: atomic int,
    done: atomic bool;

proc barrier(numTasks) {
  const myCount = count.fetchAdd(1);
  if (myCount < numTasks) then
    done.waitFor(true);
  else
    done.testAndSet();
}
```

Fixing RRWW bugs with atomics

Atomic Statement

```
var totTime: real;

coforall tid in 0..#numTasks {
  …
  atomic {
    totTime += myTime;
  }
  …
}
```

Note: Not supported much of anywhere (yet)…

Atomic Variables

```
var totTime: atomic real;

coforall tid in 0..#numTasks {
  …
  totTime.add(myTime);
  …
}
```

Atomic Methods: Compare-and-Swap (CAS)

- **`compareExchange(old:t,new:t):bool`**
  store *new* iff previous value was *old*; returns true on success

Classic example: lock-free enqueue in Chapel*:

```
class Node { var data: int;
             var next: Node; }
var head: atomic Node = nil;
coforall tid in 0..#numTasks {
  var newNode = new Node(data = tid);
  do {
    const oldHead = head.read();
    newNode->next = oldHead;
  } while (!head.compareExchange(oldHead, newNode));
}
```

* = except that Chapel doesn't yet support atomic class refs ☹

Comparison of Synchronization Types in Chapel

sync/single:

- Best for producer/consumer style synchronization
- Imply a memory fence w.r.t. other loads/stores
- Use single to express write-once values

atomic:

- Best for uncoordinated accesses to shared state

Atomic Operations in Adopted Languages

C/C++: C11/C++11 has just added atomic ops

- Chapel's design was based on this

Java: see Java.util.concurrent.atomic

C#: not sure…

Fixing RRWW bugs with atomics

Atomic Statement

```
var totTime: real;

coforall tid in 0..#numTasks {
  …
  atomic {
    totTime += myTime;
  }
  …
}
```

Note: Not yet supported much of anywhere (yet)…

Atomic Variables

```
var totTime: atomic real;

coforall tid in 0..#numTasks {
  …
  totTime.add(myTime);
  …
}
```

This Week's Homework

- Reading:
  - LogP (1990's paper on abstract dist. mem. machine models)
  - Chapter 2, Lin & Snyder
  - data parallelism Chapel section
- Written Questions
  - figure out how to do full scans
  - create a new lock-free operation
- Coding: (Data Parallelism, should be easy)
  - OpenMP: 9-point stencil
  - OpenMP or Chapel: Mandelbrot