CSEP 524 – Parallel Computation University of Washington

Lecture 3: Understanding Performance; Intro to Algorithms

Michael Ringenburg

Spring 2015

Projects!

- The course project information has been posted on the web page
  - Option 1: Read, explore, try out a topic we did not cover
  - Option 2: Program an algorithm(s)/framework(s)/ language(s) we did not study.
- Scope: About 3 homeworks (or so).
  - ~2-4 papers for a reading project
- Deliverables:
  - Written report
  - Oral Presentation (last two class sessions, maybe a couple early?)

Project Timeline

- Next Tuesday (4/21): Select topic, locate resources (e.g., papers, tutorials, software), indicate presentation date preference
- May 1: Finalize topic
- May 26, June 2: Presentations in class
- June 5, midnight: Written reports due by midnight
  - Can turn in earlier, e.g., day of presentation

Recall last week:Coherence vs Consistency

- ***Cache coherence***: Ensuring all caches have an identical view of memory
- ***Sequential consistency***:
  - All memory ops within a thread complete in program order
  - Across tasks, memory ops are interleaved in a consistent total order (everyone sees same interleaving)
- **Question**: Does coherence guarantee sequential consistency? Why or why not?

Coherence vs Consistency

- **Question**: Does coherence guarantee sequential consistency? Why or why not?
- **Answer**: No…
  – Cache consistency removes *one* source of inconsistency (different views of memeory), but others remain, e.g.,
  – Compiler reordering
  – Processor reordering
  – Network reordering

Today's topics

- Part I: Performance
  - Quickly go through chapter 3 of your text
  - Encourage you to read it in more depth
- Part II: Start talking about parallel abstractions and algorithms (chapters 4-5 of you book)
  - Will spend more time on this next week
- Part III: Discussion!
  - Parallel Models, MCMs

Two types of performance

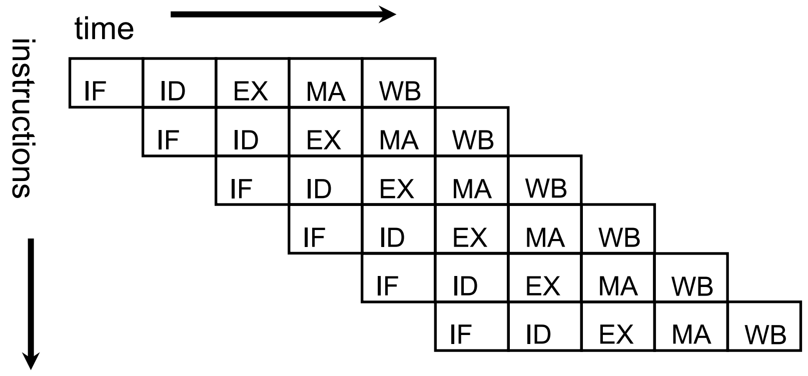
- Latency – time before a computation results is available
  - Also called *transmit time*, *execution time*, or just *time*
- Throughput -- amount of work completed in a given amount of time
  - Measured in "work"/sec, where "work" can be bytes, instructions, jobs, etc.; also called *bandwidth* in communication

Latency

- Often the goal of parallelism – get my job done faster!
- There is upper limit on reducing latency
  - Speed of light, esp. for bit transmissions
  - In networks, switching time (node latency)
  - (Clock rate) x (issue width), for instructions
  - Diminishing returns (overhead) for problem instances
  - Hitting these rare in practice …

Throughput

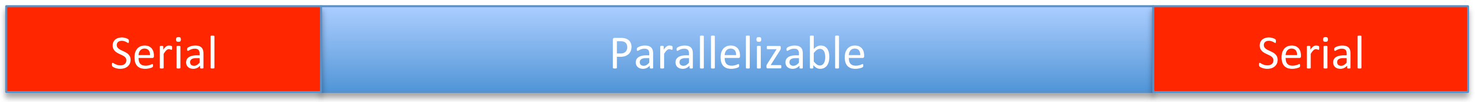
- Another common goal – get as many jobs done as possible (or process as much data…)
- Often easier to achieve than latency by adding HW
  - More wires = more bits/second
  - Separate processors run separate jobs
  - Pipelining is a powerful technique to increase serial operation throughput:



Does Parallelism = Performance?

- Many assume using *P* processors will give *P* speedup (e.g., 4x processors => ¼ the time)
  - This is called "perfect", "ideal", or "linear" speedup
  - Generally an upper bound/absolute best case
- Very rare in practice
  - Overheads
  - Necessity of changing algorithm
- With a fixed problem size, speedup often farther from linear the larger *P* gets
  - Keep adding overheads, but less gain from dividing work
  - E.g., 2->4 processors, vs 128->256 procs

Amdahl's Law



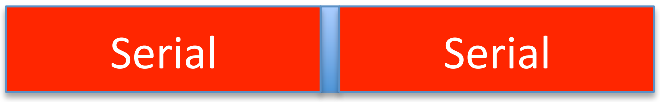
- If a fraction S of a computation is inherently sequential, then the maximum performance improvement is bounded by

$$T_P \geq S \times T_S + (1\text{-}S) \times T_S / P$$

T_S=sequential time
T_P=parallel time
P =no. processors

- In other words, you can never do better than ($S \times T_S$), no matter how large P is

Amdahl's Law



- If a fraction S of a computation is inherently sequential, then the maximum performance improvement is bounded by

$$T_P \geq S \times T_S + (1-S) \times T_S / P$$

T_S=sequential time
T_P=parallel time
P =no. processors

- In other words, you can never do better than ($S \times T_S$), no matter how large P is

However …

- Amdahl's Law assumes a fixed problem instance: Fixed *n*, fixed input, perfect speedup
  - The algorithm can change to become more ||
  - Problem instances grow implying proportion of work that is sequential may be smaller %
  - Can sometimes find parallelism in sequential portion
- *Amdahl is a fact; it's not a show-stopper*
- Next, let's consider what makes us not hit Amdahl's law limits … and what we can do …

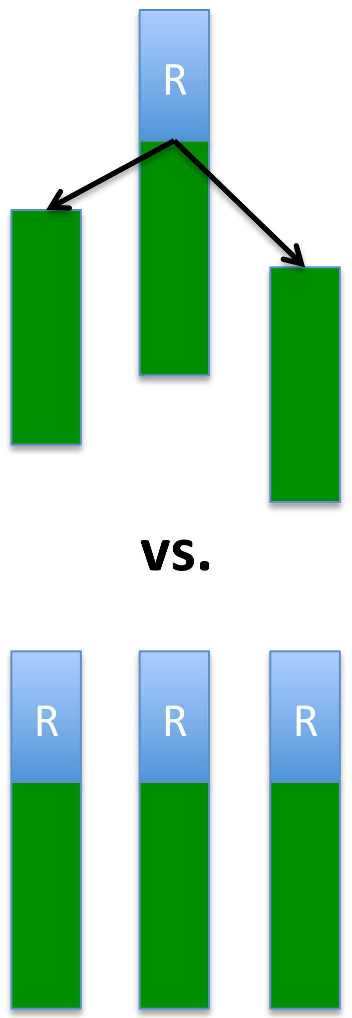
Performance Loss: Overhead

- Implementing parallelism has costs not present in serial codes
  - Communication costs: locks, cache flushes, coherency, message passing protocols, etc.
  - Thread/process startup and teardown
  - Lost optimizations – depending on consistency model, some compiler optimizations may be disabled/modified
  - Many of these costs increase as # processors increases

A "trick" to reduce communication overhead

- Can often trade off extra computation for reduced communication overhead.
- Works when recomputing is cheaper than communicating
- Can you think of a case where we still might want to avoid this?

Example: Need a random number on each thread:
(a) Generate one copy, have all threads reference it, or …
(b) Each thread generates its own random number from common seed. Removes communication and gets parallelism, but by increasing instruction load.

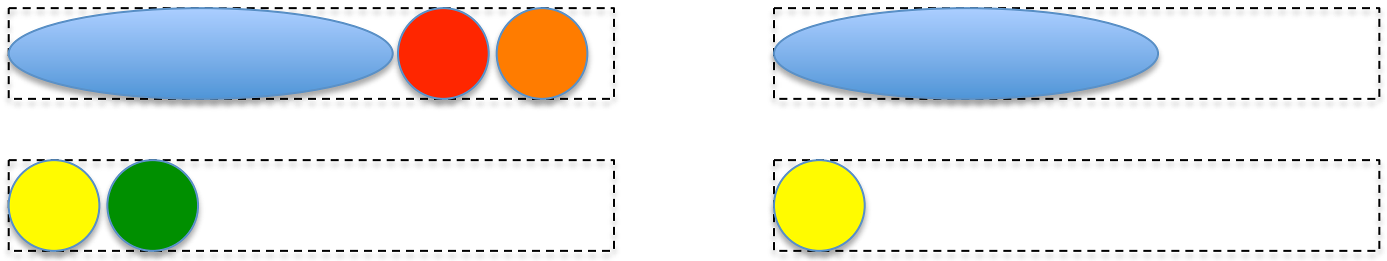


Performance Loss: Contention

- Contention – one processor's actions interfere with another processor
  - Lock contention: One processor's lock stops other processors from referencing; they must wait
  - Bus contention: Bus wires are in use by one processor's memory reference
  - Network contention: Wires are in use by one packet, blocking other packets ("traffic jams" in the network – very real issue)
  - Bank contention: Multiple processors try to access different locations on one memory chip simultaneously
- Very time-dependent - can vary greatly between runs.

Performance Loss: Load Imbalance

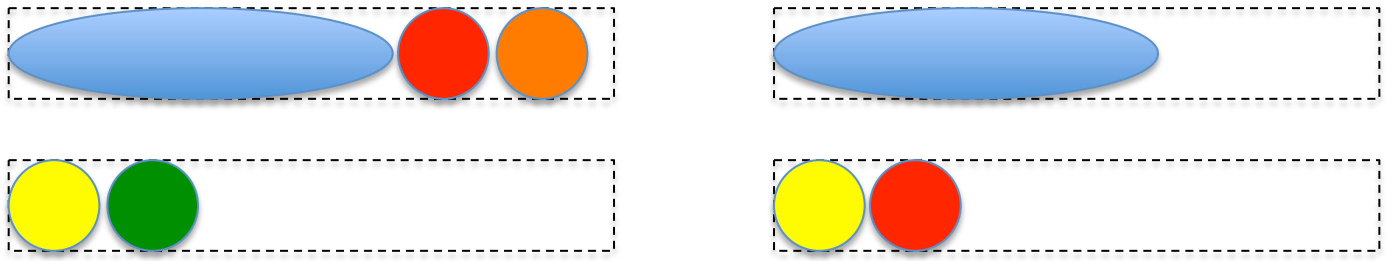
- Load imbalance: work not evenly assigned to the processors
  - Can cause processor underutilizations
  - Assignment of *work*, not data, is the key
  - Static assignments, being rigid, are more prone to imbalance



  - But dynamic assignment adds overhead – must be sure granularity of work large enough to amortize

Performance Loss: Load Imbalance

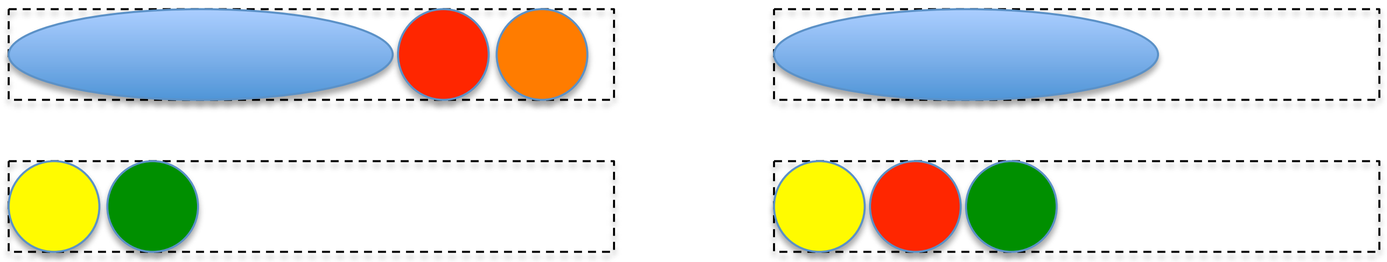
- Load imbalance: work not evenly assigned to the processors
  - Can cause processor underutilizations
  - Assignment of *work*, not data, is the key
  - Static assignments, being rigid, are more prone to imbalance



  - But dynamic assignment adds overhead – must be sure granularity of work large enough to amortize

Performance Loss: Load Imbalance

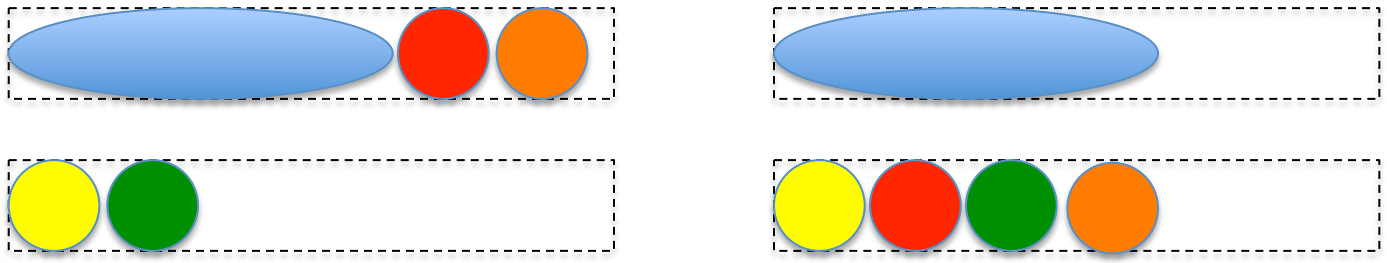
- Load imbalance: work not evenly assigned to the processors
  - Can cause processor underutilizations
  - Assignment of *work*, not data, is the key
  - Static assignments, being rigid, are more prone to imbalance



  - But dynamic assignment adds overhead – must be sure granularity of work large enough to amortize

Performance Loss: Load Imbalance

- Load imbalance: work not evenly assigned to the processors
  - Can cause processor underutilizations
  - Assignment of *work*, not data, is the key
  - Static assignments, being rigid, are more prone to imbalance



  - But dynamic assignment adds overhead – must be sure granularity of work large enough to amortize

Reducing performance loss

- How do we mitigate these factors?
- Best performance: processors executing continuously on local data without interacting with other processors
  - Less overhead & less contention
- What gets in the way of this? Dependencies…
  - A *dependence* is an ordering relationship between two computations
    - Dependences are usually induced by read/write
    - Dependences either prevent parallelization, or induce need for communication or synchronization between threads

Dependence types

- Dependences are orderings that must be maintained to guarantee correctness
  - Flow-dependence: read after write — True
  - Anti-dependence: write after read — False
  - Output-dependence: write after write — False
- True dependences arise from semantics of program (they are "real")
- False dependences arise from memory reuse

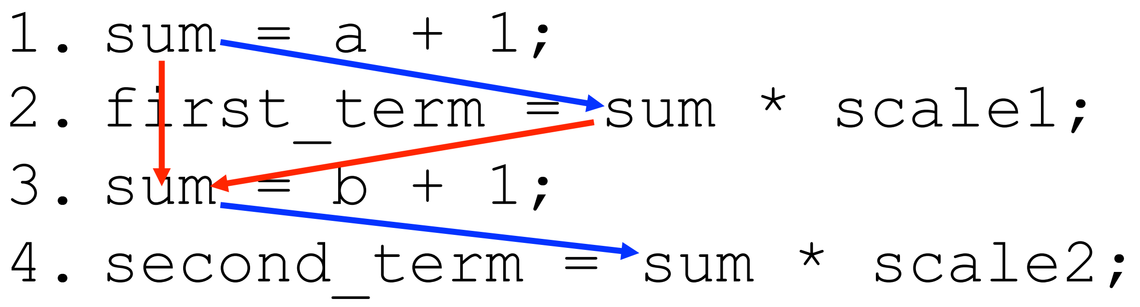
Dependence Example

- Can you find true and false dependencies in this example?

```
1. sum = a + 1;
2. first_term = sum * scale1;
3. sum = b + 1;
4. second_term = sum * scale2;
```

Dependence Example

- Can you find true and false dependencies in this example?



```
1. sum = a + 1;
2. first_term = sum * scale1;
3. sum = b + 1;
4. second_term = sum * scale2;
```

- Flow-dependence read after write; must be preserved for correctness
- Anti-dependence write after read, output dependence write after write; can be eliminated with additional memory …

Removing Anti-dependence

- Change variable names

```
1. sum = a + 1;
2. first_term = sum * scale1;
3. sum = b + 1;
4. second_term = sum * scale2;
```

$$\Downarrow$$

```
1. first_sum = a + 1;
2. first_term = first_sum * scale1;
3. second_sum = b + 1;
4. second_term = second_sum * scale2;
```

Measuring Performance

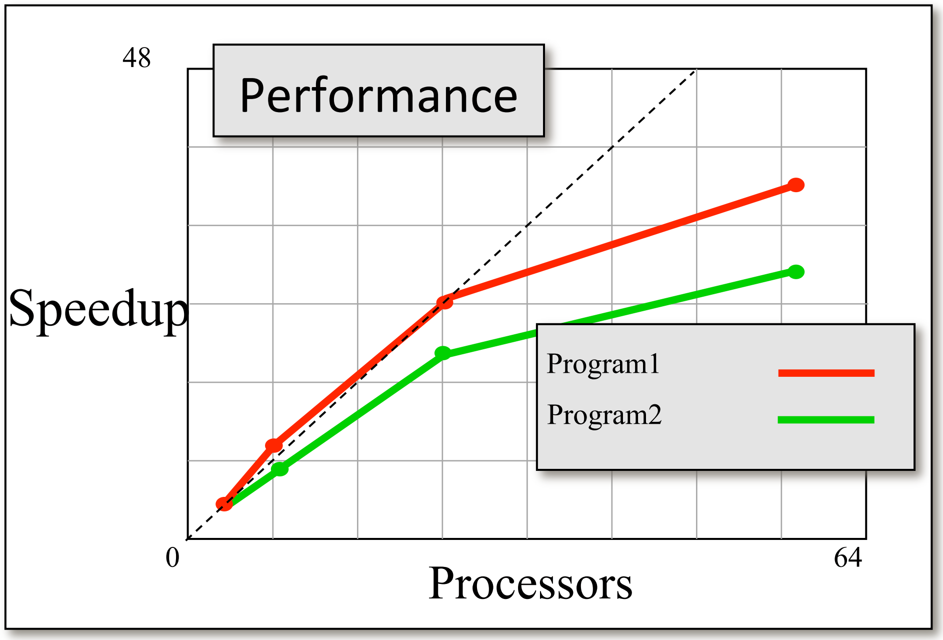
- How do we measure performance?
- Execution time … what's time?
  - 'Wall clock' time
  - Processor execution time
  - System time
- Paging and caching can affect time
  - Cold start vs warm start
- Conflicts with other users/system components
- Measure kernel or whole program

FLOPS

- Floating Point Operations Per Second is a common measurement for scientific programs. But not great …
  - Even scientific computations use many integers
  - Results can often be influenced by small, low-level tweaks having little generality: e.g., fused mult-add
  - Translates poorly across machines because it is hardware dependent
  - Limited application … but it won't go away!
    - (Top 500 list, e.g., …)

Speedup and Efficiency

- Another common measure: Speedup is the factor of improvement for P processors: T_S/T_P



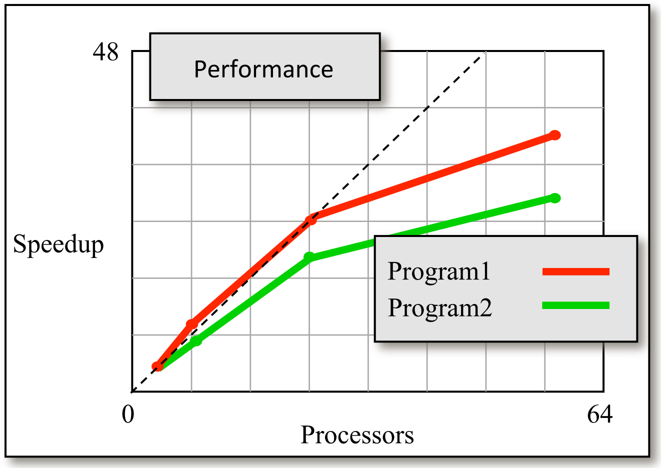
Efficiency = Speedup/P

Issues with Speedup, Efficiency

- Speedup is best applied when hardware is constant, or for family within a generation
  - Need to have computation, communication in same ratio
  - Issues: very sensitive to the T_S value
    - T_S should be time of best sequential program on one processor of the parallel machine
    - But sometimes studies cite *relative* speedup (one processor of *parallel program*)
    - *What is the importance of this distinction?*

Scaled v. Fixed Speedup

- As *P* increases, the amount of work per processor diminishes, often below the amount needed to amortize costs
- Speedup curves bend down
- Scaled speedup keeps the work per processor constant, allowing other effects to be seen
- Both are important



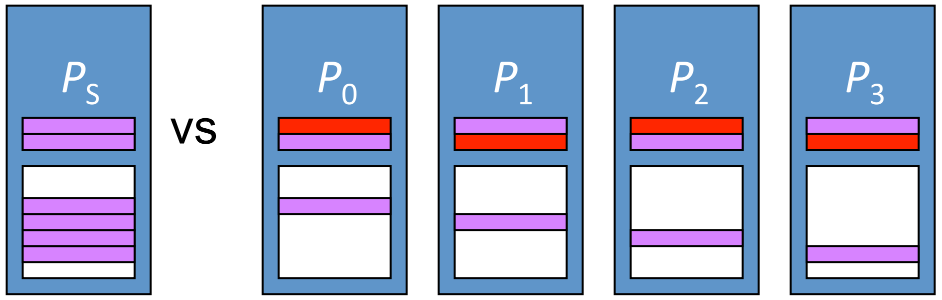
If not stated, speedup is fixed speedup

What If Problem Doesn't Fit?

- Cases arise when data doesn't fit in one processor's memory
- Best solution is relative speed-up
  - Measure $T_{\pi=\text{smallest possible}}$
  - Measure T_P, compute T_π/T_P as having P/π potential improvement

Superlinear Speed up

- Interestingly, we occasionally see "better than perfect" speedup. Why?
  - One possibility: additional cache ...



- Can make execution time < *T*/*P* because data (& instruction) references are faster.
- Extra cache may mitigate parallelism costs
- Other ideas?

Break

Peril-*L* …

- A pseudo-language used by your text to assist in discussing algorithms and languages
- Play on words – doesn't really put us in peril …
- Goals:
  - Be a minimal notation to describe parallelism
  - Be universal, unbiased towards languages or machines
  - Allow reasoning about performance (using the CTA)
- We will quickly go through this, and try to use, to stay consistent with text

Base Language is C

- Peril-*L* uses C as its notation for scalar computation
- Advantages
  - Well known and familiar
  - Capable of standard operations & bit twiddling
- Disadvantages
  - Low level
  - No goodies like OO
  - Modern parallel languages generally are based on higher-level languages

Threads

- The basic form of parallelism is a thread
- Threads are specified by

```
forall
    <int var> in ( <index range spec> )  {<body> }
```

- Semantics: spawn *k* threads running *body*

```
forall thID in (1..12) {
  printf("Hello, World, from thread %i\n", thID);
}
```

<index range spec> is any reasonable (ordered) naming

Thread Model is Asynchronous

- Threads execute at their own rate – interleaving not known or predictable
- To cause threads to synchronize, we have

```
barrier;
```

- Threads arriving at barriers suspend execution until all threads in its `forall` arrive
- Reference to the `forall` index identifies the thread:

```
forall thID in (1..12) {
  printf("Hello, World, from thread %i\n", thID);
}
```

Memory Model

- Two kinds of memory: local and global
  - All variables declared in a thread are local
  - Any variable w/ <u>underlined_name</u> is global
- Arrays work as usual
  - Local variables use local indexing
  - Global variables use global indexing
- Memory is based on CTA, so performance:
  - Local memory references are unit time
  - Global memory references take λ time

Memory Read Write Semantics

- Local Memory behaves like the von Neummann model
- Global memory
  - Reads are concurrent, so multiple processors can read a memory location at the same time
  - Writes must be exclusive, so only one processor can write a location at a time; the possibility of multiple processors writing to a location is not checked and if it happens the result is unpredictable

Example: Count 3s

- Shared memory programs are *expressible*
- The first (erroneous) Count 3s program is

```
int *array, length, count, t;
  ... initialize globals here ...
forall thID in (0..t-1) {
   int i, length_per=length/t;
   int start=thID*length_per;
   for (i=start; i<start+length_per; i++) {
    if (array[i] == 3)
      count++;   // Concurrent writes - RACE
    }
}
```

Getting Global Writes Serialized

- To ensure exclusivity, Peril-*L* has

```
exclusive { <body> }
```

- A thread can execute <*body*> only if no other thread is doing so; if some thread is executing, then it must wait for access
  - Sequencing through `exclusive` may not be fair, e.g., no FIFO guarantee
  - Defining semantics, not implementation

Example: Fixed Count 3s

- Fix by adding exclusive – but slow

```
int *array, length, count, t;
  ... initalize globals here ...
forall thID in (0..t-1) {
   int i, length_per=length/t;
   int start=thID*length_per;
   for (i=start; i<start+length_per; i++) {
    if (array[i] == 3)
      exclusive { count++; }
    }
}
```

Example: Best Count 3s

- Speed up with private counters

```
int *array, length, count, t;
forall thID in (0..t-1) {
  int i, priv_count=0; len_per_th=length/t;
  int start=thID * len_per_th;
  for (i=start; i<start+len_per_th; i++) {
    if (array[i] == 3)
      priv_count++;
  }
  exclusive {count += priv_count; }
}
```

Full/Empty Memory

- Lightweight synchronization in Peril-L
- Memory usually works like information:
  - Reading is repeatable w/o "emptying" location
  - Writing is repeatable w/o "filling up" location
- Matter works differently
  - Taking something from location leaves vacuum
  - Placing something requires the location be empty
- Full/Empty: Applies matter idea to memory … F/E variables help serializing

Use the `apostrophe`' suffix to identify F/E

Treating memory as matter

- A location can be read only if it' s filled
- A location can be written only it' s empty

Location contents	Variable Read	Variable Write
Empty	Stall	Fill with value
Full	Empty of value	Stall

- Scheduling stalled threads may not be fair
- Side note: MTA/XMT has these available on every word (programming convention used $ not ')

Reduce and Scan

- Aggregate operations use APL syntax
  - Reduce: Combine elements using some associative operation:
    - <*op*>/<*operand*> for <*op*> in {+, *, &&, ||, max, min}; as in `+/priv_sum`
  - Scan: Compute prefixes as well as final result (prefix sum)
    - <*op*>\<*operand*> for <*op*> in {+, *, &&, ||, max, min}; as in `+\my_count`
- Portability: use reduce/scan rather than implementing

```
exclusive {count += priv_count; }  “WRONG”
count = +/priv_count;          “RIGHT”
```

- Synchronization implied

Reduce/Scan and Memory

- When reduce/scan target local memory

```
priv_count= +/priv_count;
```

  - The local is assigned the global sum
  - This is an implied *broadcast* (communicate common value to all threads)

```
priv_count= +\priv_count;
```

  - The local is assigned the prefix sum to that point
  - No implied broadcast
- Assigning R/S value to a local forces a barrier, but assigning R/S to a global does not (threads continue executing once they've contributed)

localize and mySize

- Recall this is the CTA model, so memory is globally addressable, but local to a specific process.
- Thus, can ask for the local section of a global array:

```
int localA[] = localize(globalA[]);
```

- Size of local portion of global:

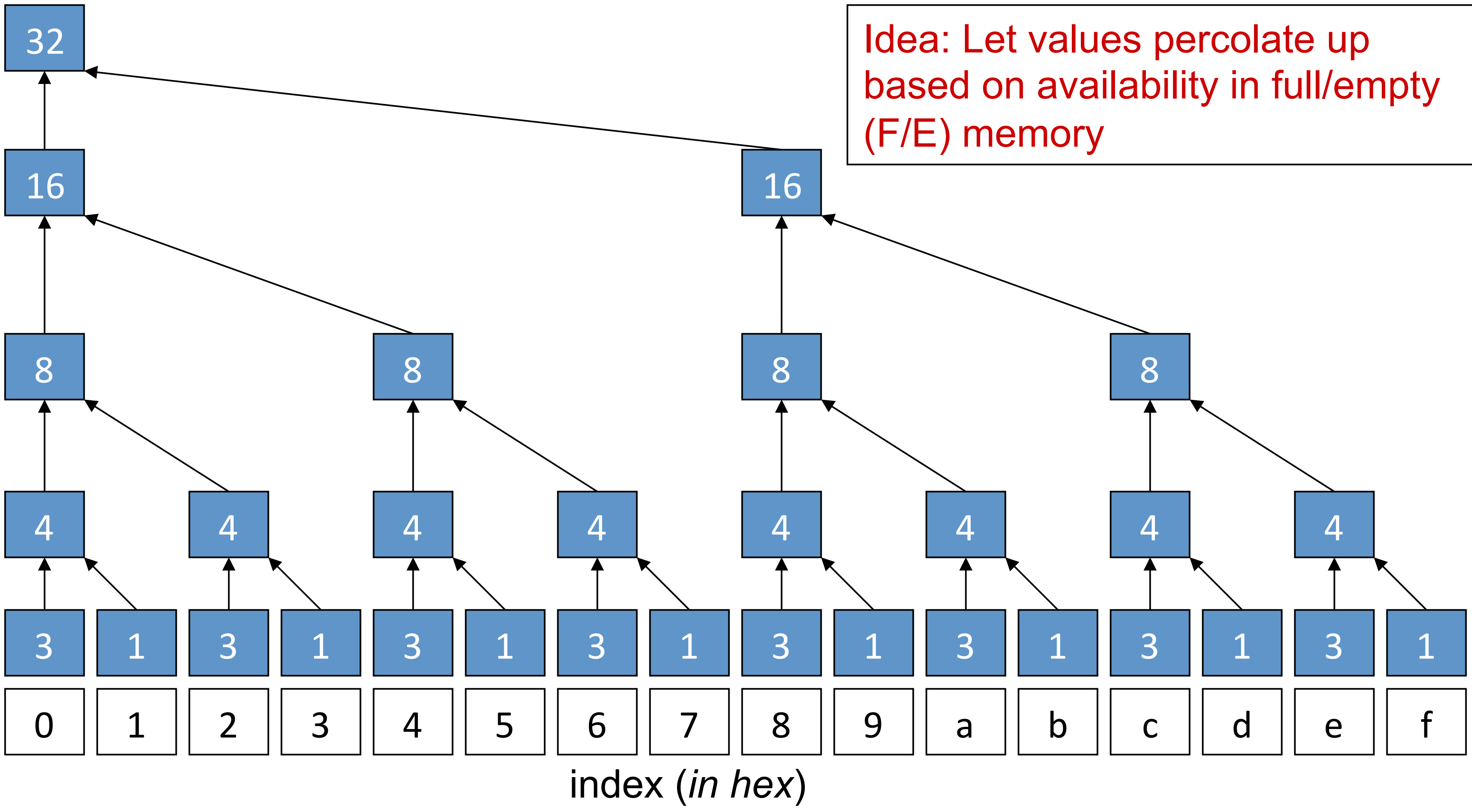
```
int size = mySize(globalA[], 0/*dimension*/);
```

Using Peril-*L*

- The point of a pseudocode is to allow detailed discussion of subtle programming points without being buried by the extraneous detail
- To illustrate, consider some parallel computations …
  - Tree accumulate
  - Alphabetize (string sort)

Tree Accumulate Using Full/ Empty (F/E)

Idea: Let values percolate up based on availability in full/empty (F/E) memory



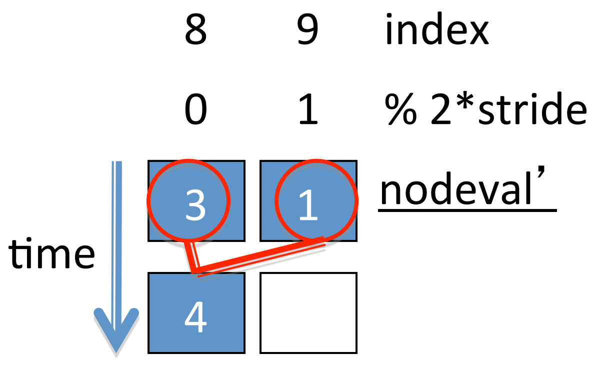
Naïve F/E Tree Accumulation

```
int nodeval'[P];                     Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
  int val2accum = …;                 locally computed val
  int stride = 1;
  nodeval'[index] = val2accum;             Assign initially to tree node
  while (stride < P) {                     Begin logic for tree
    if (index % (2*stride) == 0) {  Am I parent at next level?
      nodeval'[index]=nodeval'[index]+nodeval'[index+stride];
      stride = 2*stride;
    }
    else {
      break;   Exit, if not now a parent
    }
  }
}
```

Caution: This implementation is wrong …

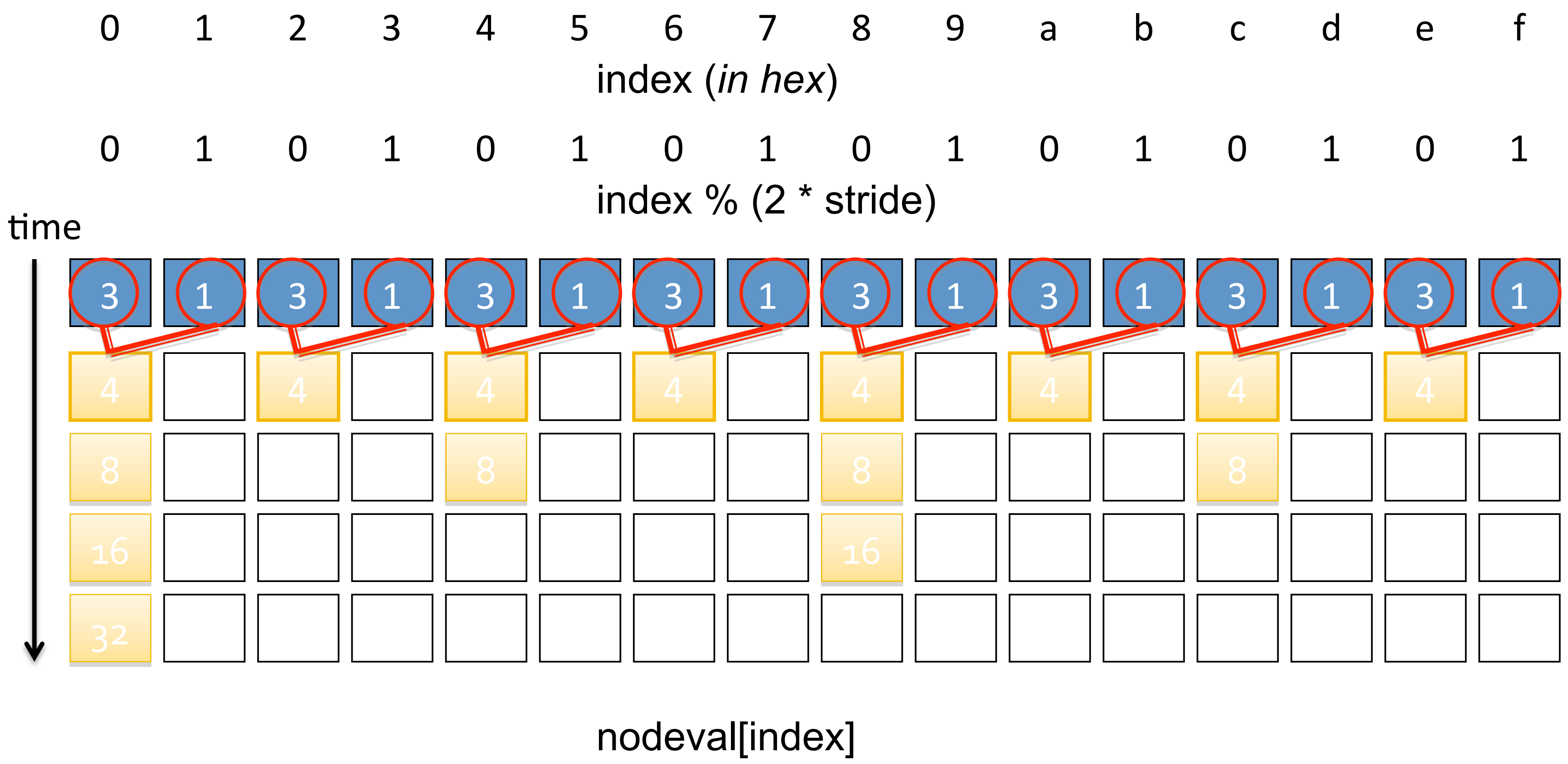
Naïve F/E Tree Accumulation

```
int nodeval'[P];                       Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
  int val2accum = …;                   locally computed val
  int stride = 1;
  nodeval'[index] = val2accum;         Assign initially to tree node
  while (stride < P) {                 Begin logic for tree
    if (index % (2*stride) == 0) {     Am I parent at next level?
      nodeval'[index]=nodeval'[index]+nodeval'[index+stride];
      stride = 2*stride;
    }
    else {
      break;   Exit, if not now a parent
    }
  }
}
```

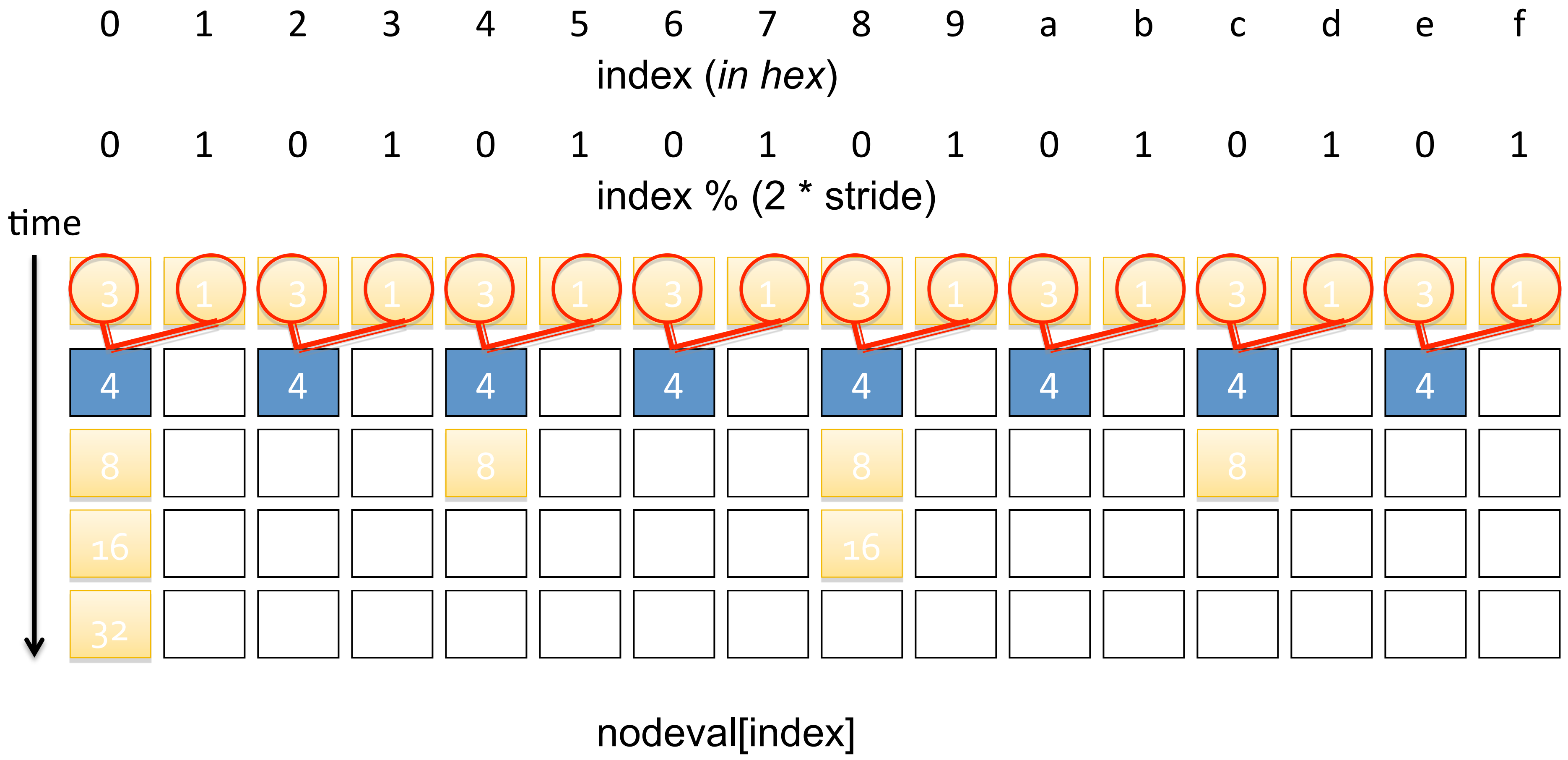


Caution: This implementation is wrong …

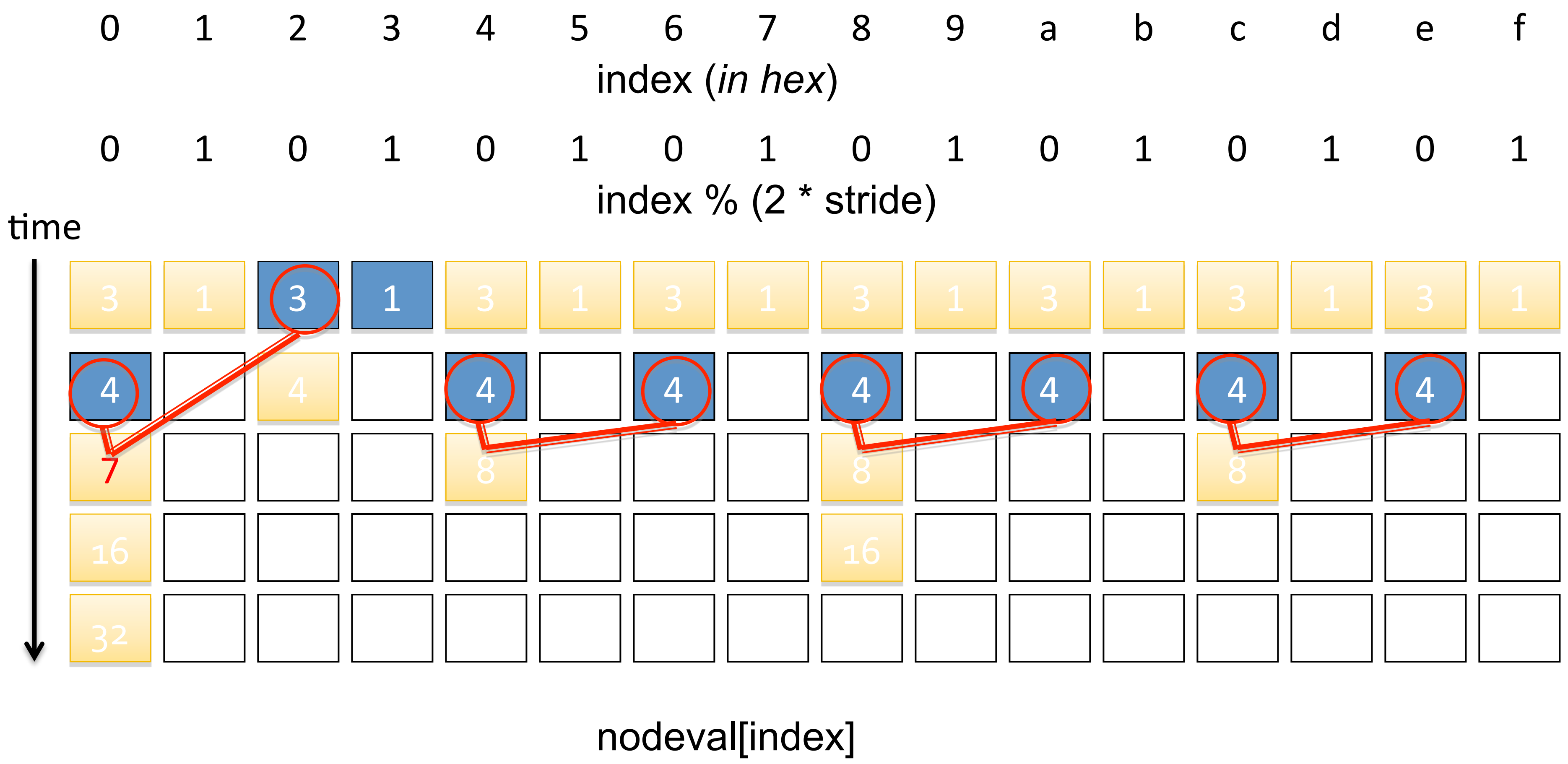
Round 1 of Tree Accum …



Round 1 of Tree Accum …



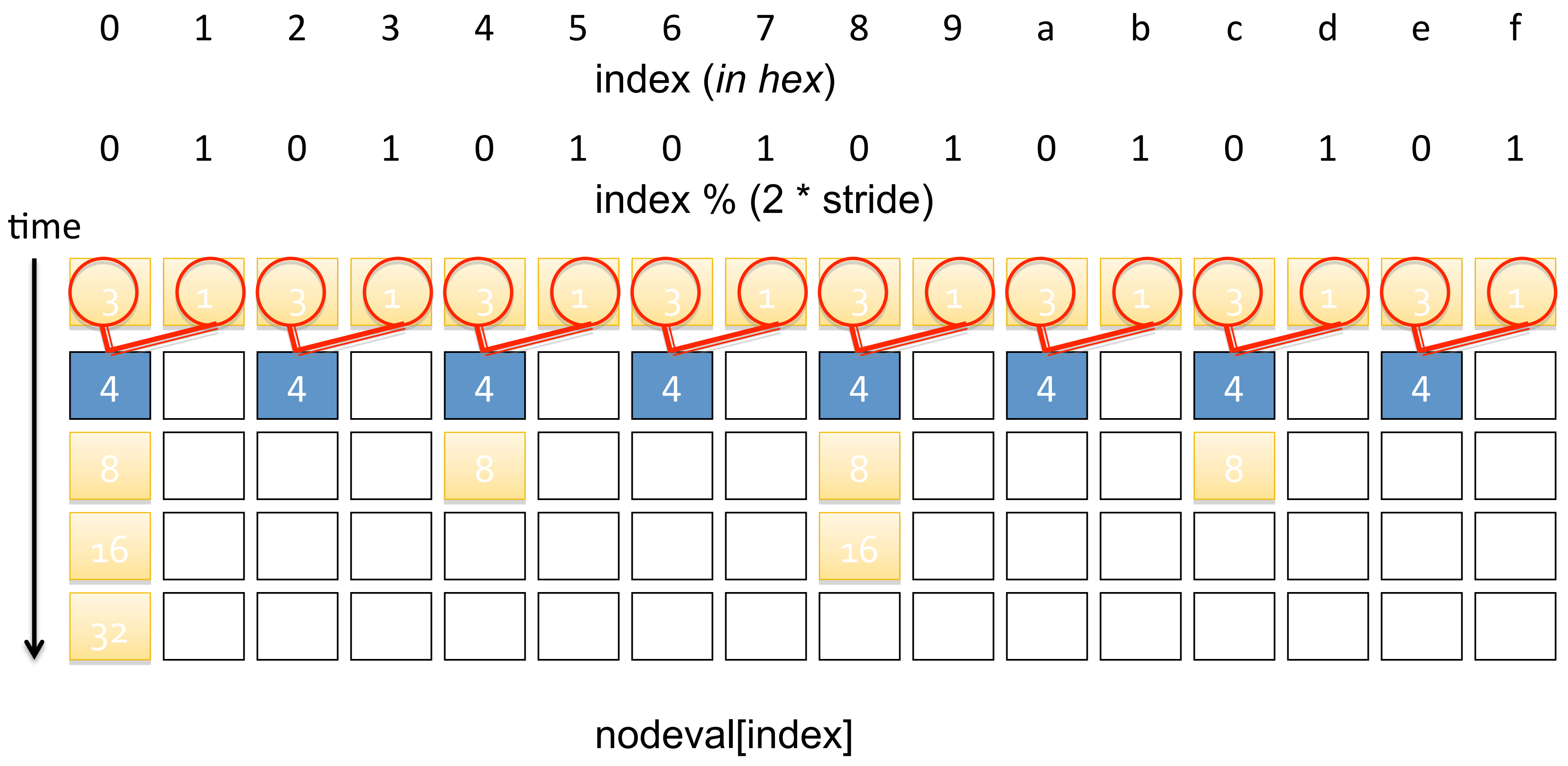
But What If Some Threads Slow?



Introduce Barrier to Synchronize Levels

```
int nodeval'[P];                    Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
  int val2accum = …;                locally computed val
  int stride = 1;
  nodeval'[index] = val2accum;        Assign initially to tree node
  while (stride < P) {                Begin logic for tree
   if (index % (2*stride) == 0) {
     nodeval'[index]=nodeval'[index]+nodeval'[index+stride];
     stride = 2*stride;
   }
   else {
     break;  Exit, if not now a parent
   }
  barrier;
  }
}
```

Barrier Stops Until Stable State



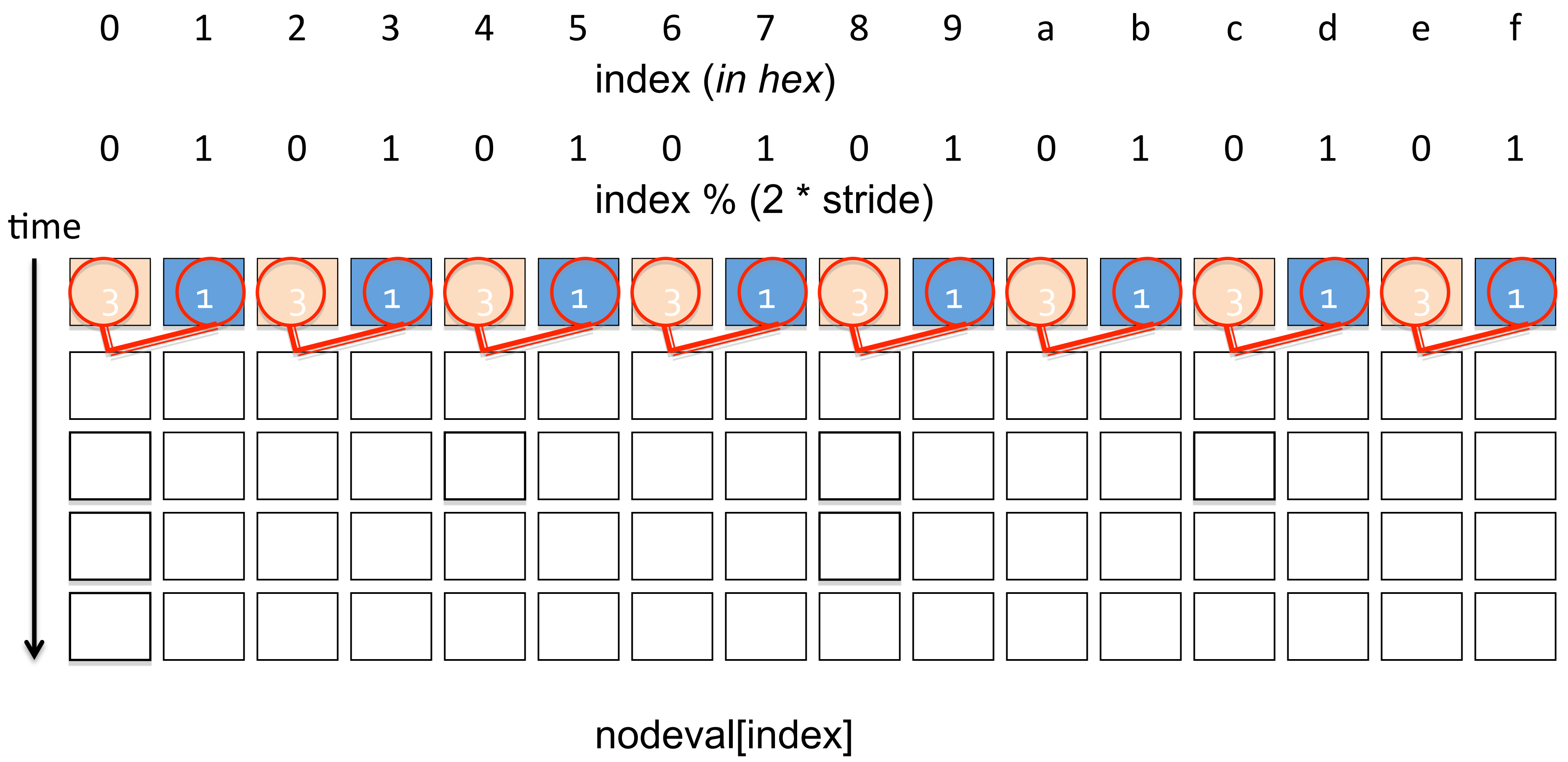
The Problem With Barriers

- In many places barriers are essential to the logic of a computation, but …
- They add overhead, and force processors to idle while slowpoke catches up …
- Avoid them when possible
  - Often not fundamental to computation, but rather to the way we've implemented it
  - For example, notice that in tree accumulate, we only need a value when from a processor when it is completely done executing …

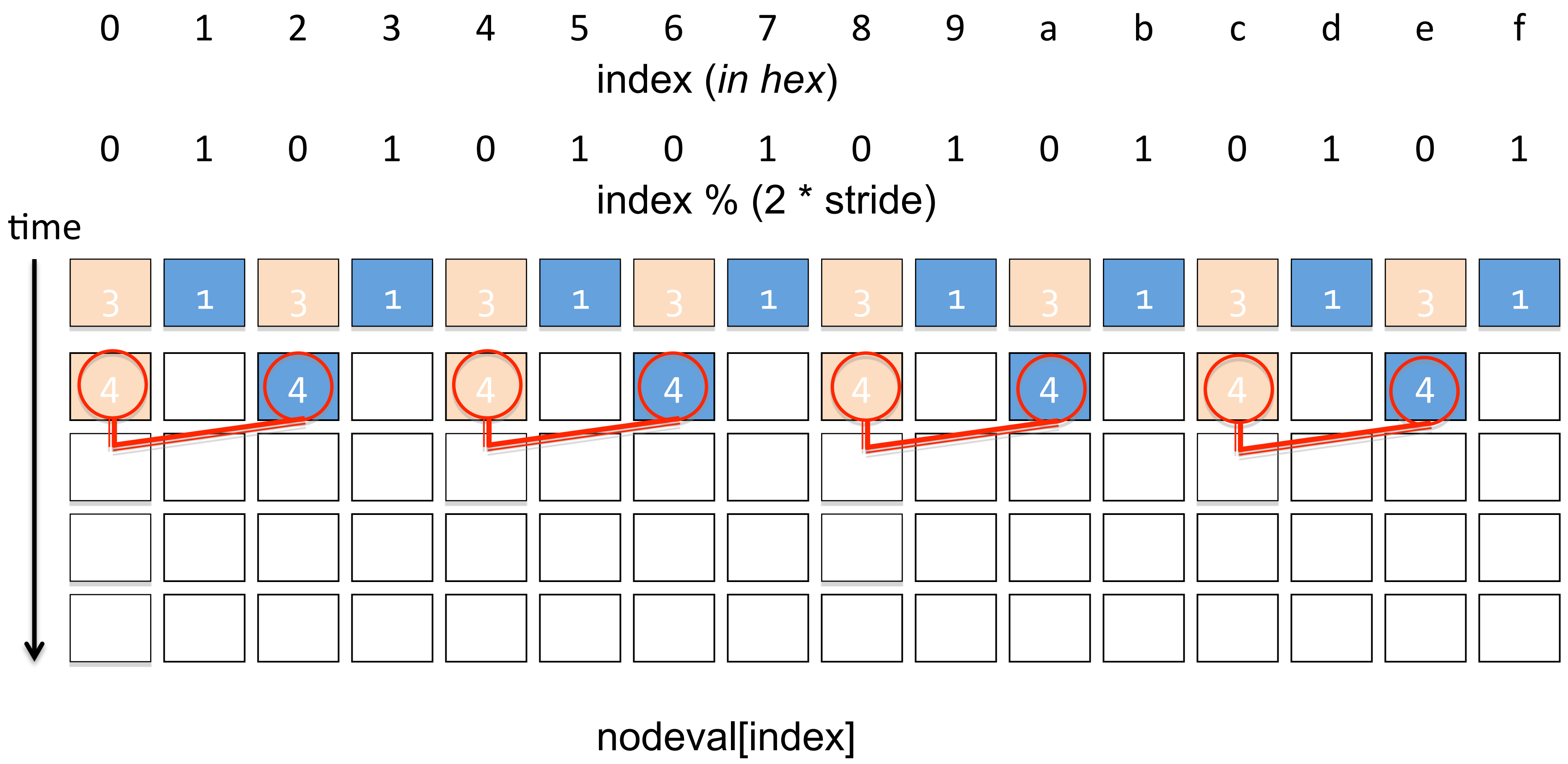
Better: accumulate locally, fill when done

```
int nodeval'[P];                       Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
 int val2accum=…;
 int stride = 1;
 while (stride < P) {                  Begin logic for tree
  if (index % (2*stride) == 0) {
    val2accum=val2accum+nodeval'[index+stride];
    stride = 2*stride;
  }
  else {
    nodeval'[index]=val2accum;         Assign val to F/E memory
    break;                             Exit, if not now a parent
  }
 }
}
```

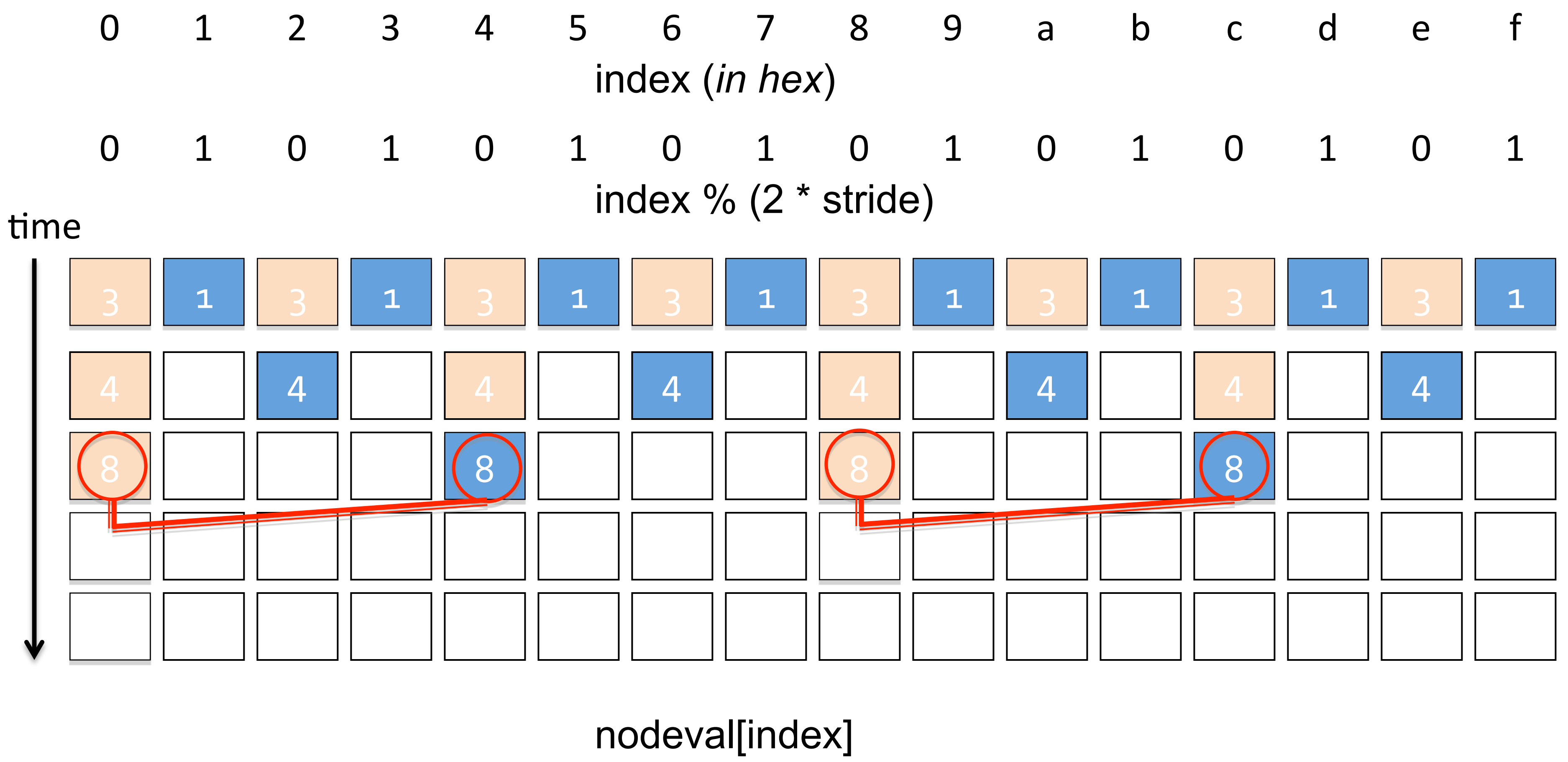
How does this work?



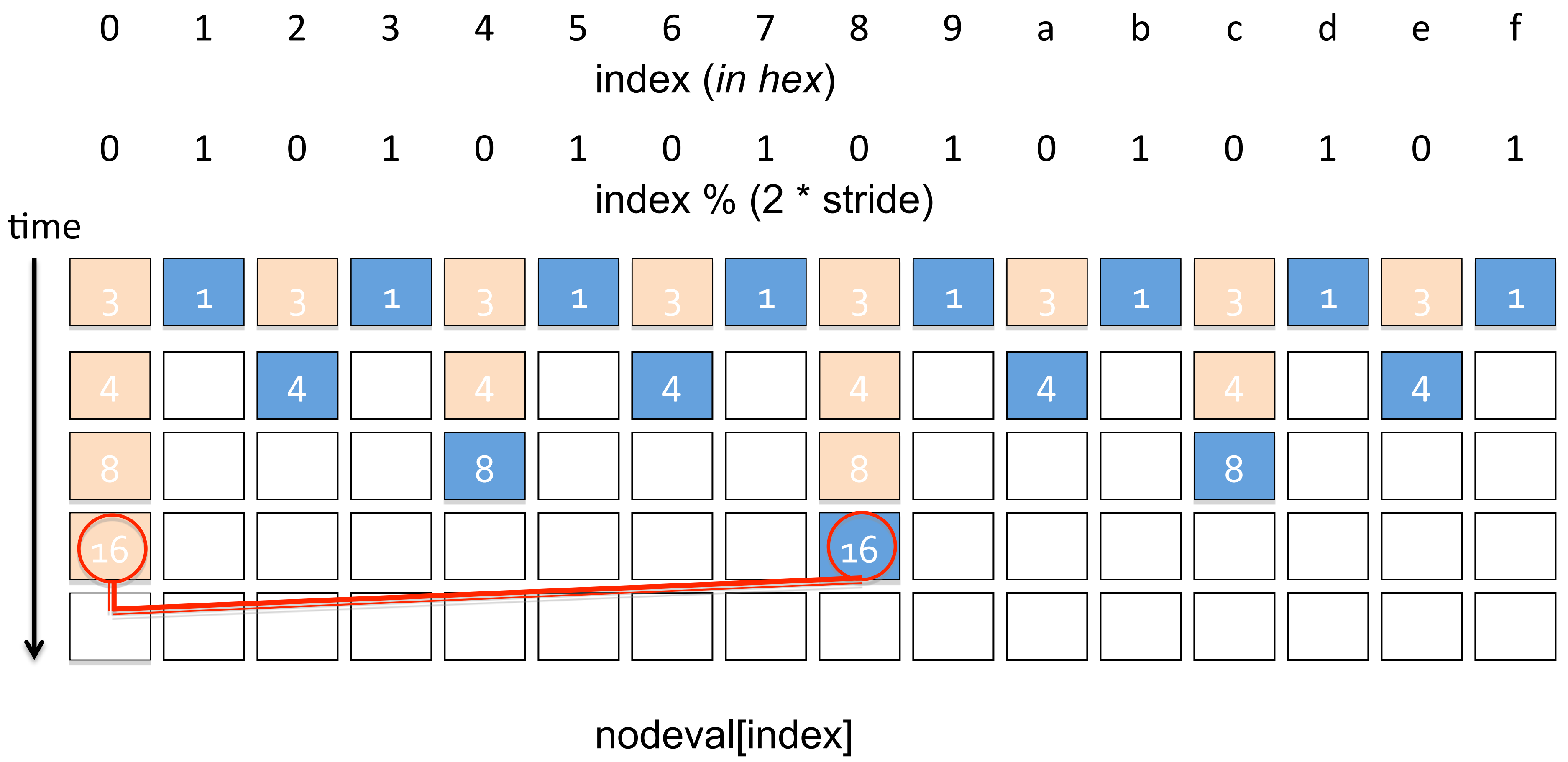
How does this work?



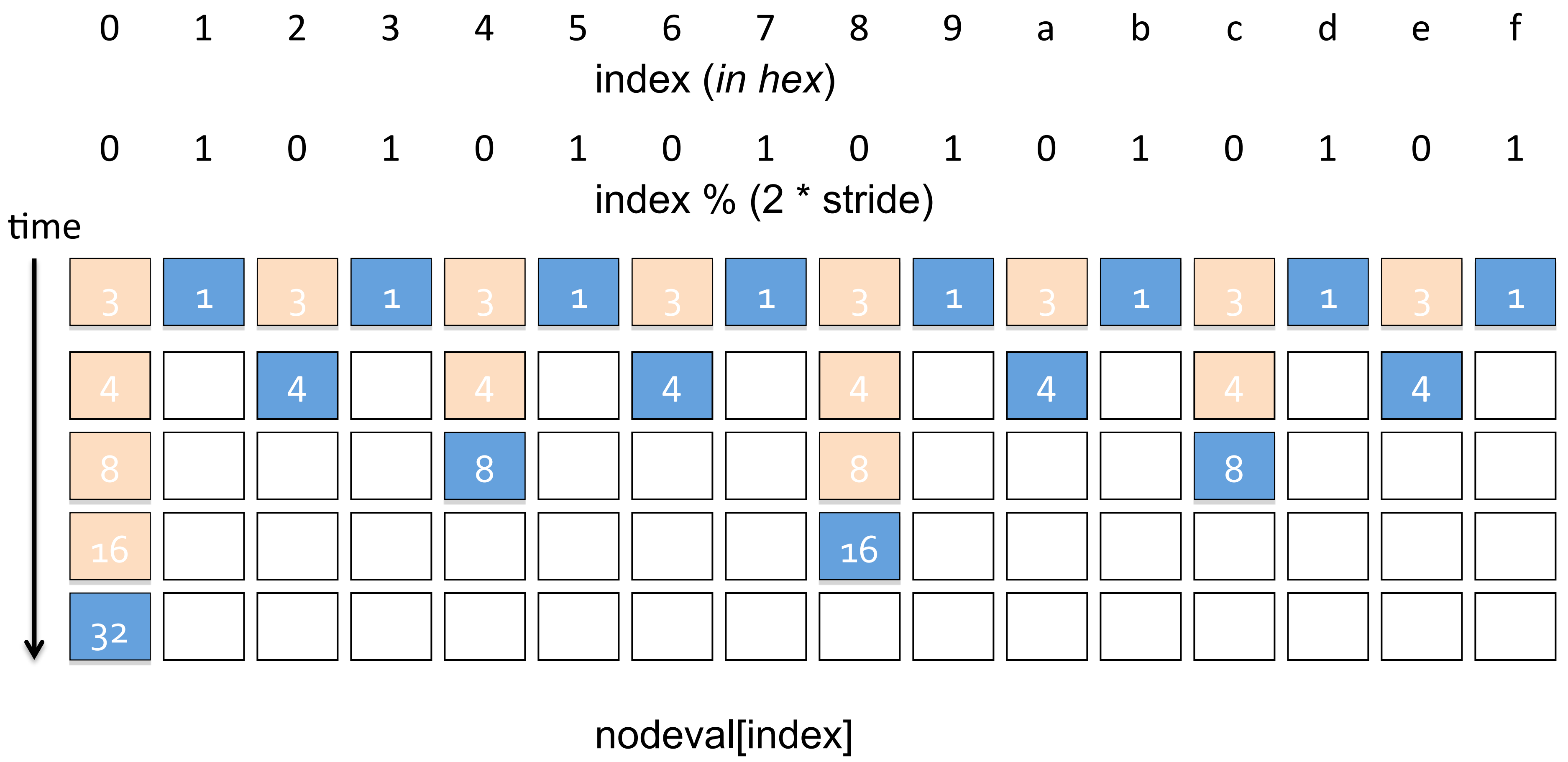
How does this work?



How does this work?



How does this work?



Critique of Tree Accumulate

- Both the synchronous and asynchronous accumulates are available to us, but we usually prefer the asynch solution
- Notice that the asynch solution uses data availability as its form of synchronization
  - This is the cool thing about full/empty bits ... synchronization is inherently tied to data readiness
  - Most effective uses of f/e take advantage of this

Thinking About Parallel Algorithms

- Computations need to be reconceptualized to be effective parallel computations
- Three possible ways to formulate parallelism
  - Fixed ||ism – assume constant # C cores, get best performance
  - Unlimited parallelism – assume unlimited cores, maximize amount of parallelism
  - Scalable parallelism – increase parallelism as problem size, core count increases
- Consider the three as an exercise in
  - Learning Peril-*L*
  - Thinking in parallel and discussing choices

The Problem: Alphabetize

- Assume a linear sequence of records to be alphabetized
- A simple form of sorting
- Solutions
  - Unlimited: Odd/Even
  - Fixed: Local Alphabetize
  - Scalable: Batcher's Sort

Unlimited Parallelism (Odd/Even Sort, part I)

```
bool continue = true;
rec L[n];                                  The data is global
while (continue) do {
 forall (i in (1:n-2:2)){                  Stride by 2
 rec temp;
 if (strcmp(L[i].x,L[i+1].x)>0){   Is o/even pair misordered?
    temp   = L[i];                         Yes,fix
    L[i]   = L[i+1];
    L[i+1] = temp;
  }
 }
```

Data is referenced globally

Unlimited Parallelism (Odd/Even Sort, part II)

```
forall (i in (0:n-2:2))    {  Stride by 2
  rec temp;
  bool done = true;                Set up for termination test
  if (strcmp(L[i].x,L[i+1].x)>0){  Is e/odd pair misordered?
    temp   = L[i];                   Yes, interchange
    L[i]   = L[i+1];
    L[i+1] = temp;
    done   = false;                Not done yet
  }
  continue= !(&&/ done);          Were any changes made?
 }
}
```

Reflection on Unlimited Parallelism

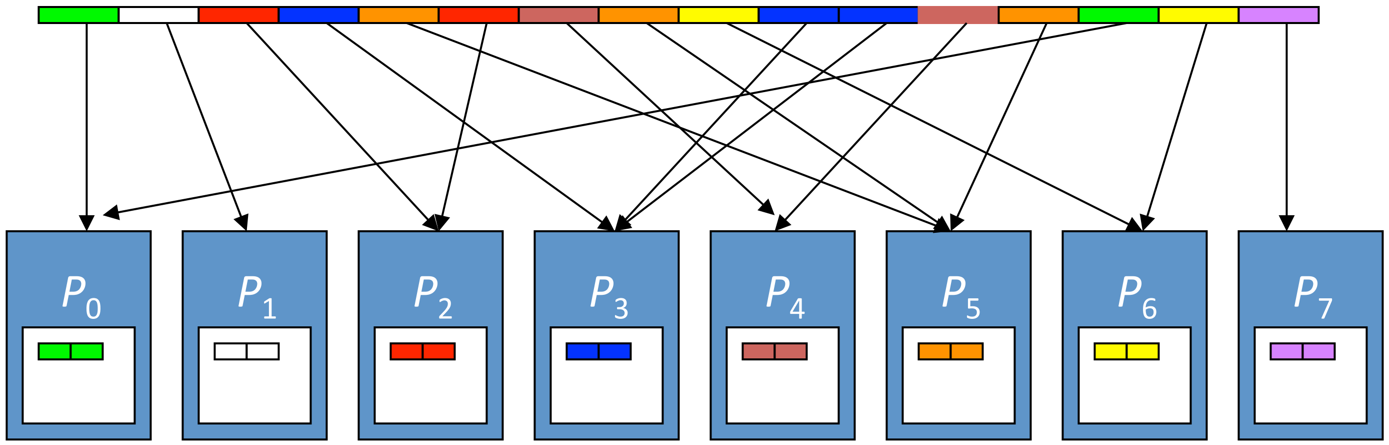
- Is solution correct?
  - Are writes exclusive?
- Are we maximizing parallelism?
- What's the effect of process spawning overhead?
- What is the effect of communication overhead?

Fixed Algorithm

- Let one thread/process handle each letter of the 26 letter latin alphabet
- Logic
  - Processes scan records counting how many records start w/their letter handle
  - Allocate storage for those records, grab & sort
  - Scan to find how many records ahead precede
- Essentially parallel bucket sort

Cartoon of Fixed Solution

- Move locally



- Sort
- Return

Fixed Part 1

```
rec L[n];                                      The data is global
forall (index in (0..25)) {                    A thread for each letter
 int myAllo = mySize(L, 0);                    Number of local items
 rec LocL[] = localize(L[]);                   Make data locally ref-able
 int counts[26] = 0;                           Count # of each letter
 int i, j, startPt, myLet;
 for (i=0; i<myAllo; i++)   {                  Count number w/each letter
   counts[letRank(charAt(LocL[i].x,0))]++;
 }
 counts[index] = +/ counts[index]; Figure # of each letter
myLet = counts[index];                         Number of records of my letter
 rec Temp[myLet];                              Alloc local mem for records
```

Fixed Part 2

```
j = 0;                                  Index for local array
for(i=0; i<n; i++) {                    Grab records for local alphabetize
 if(index==letRank(charAt(L[i].x,0)))
   Temp[j++]= L[i];                     Save record locally
}
alphabetizeInPlace(Temp[]);             Alphabetize within this letter
startPt=+\myLet;                        Scan counts # records ahead
                                          of these; scan synchs, so
                                          OK to overwrite L, post-sort
j=startPt-myLet;                        Find my start index in global
for(i=0; i<count; i++){                 Return records to global mem
 L[j++]=Temp[i];
}
}
```

Reflection on Fixed ||ism

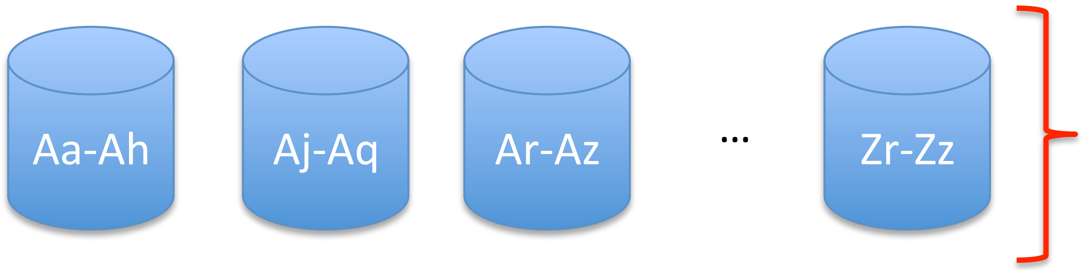
- Is solution correct … are writes exclusive?
- Is "moving the data twice" efficient?
  - Compare to odd/even …
  - (Note that same applications may not require the second data movement – e.g., each node can just write directly to distributed filesystem.)
- What happens if P > 26? Or P >>> 26?
  - Is it a good idea to assume this won't happen?

Scalable Parallelism, cont

- How would we do a scalable alphabetization?

Scalable Parallelism, cont

- How would we do a scalable alphabetization?
  - Option 1: Finer bucket granularity – match P



Can also use knowledge of data distribution to size buckets. E.g., fewer words that start with Z.

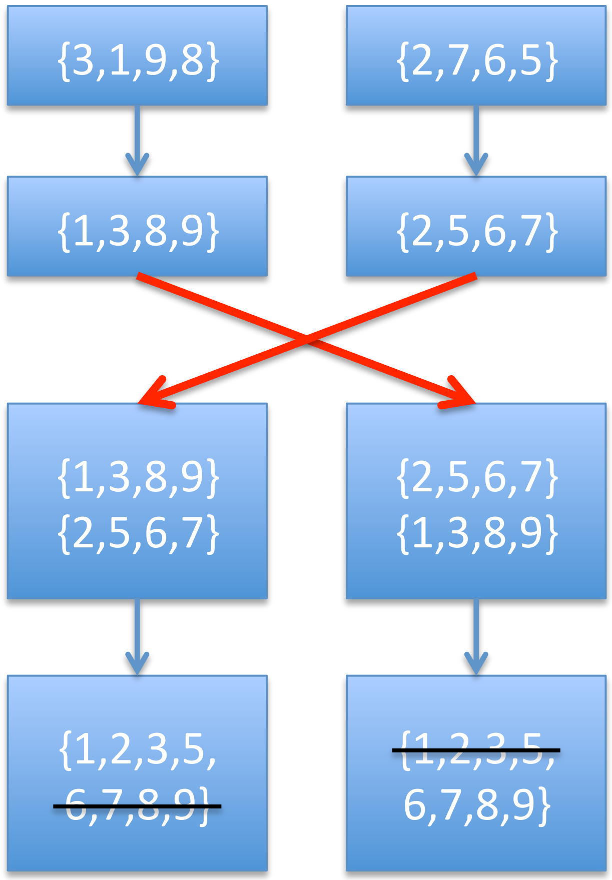
  - Option 2: Local sort, merge with other nodes
- Both are implemented in practice, both have advantages and disadvantages…

Bucket Sort

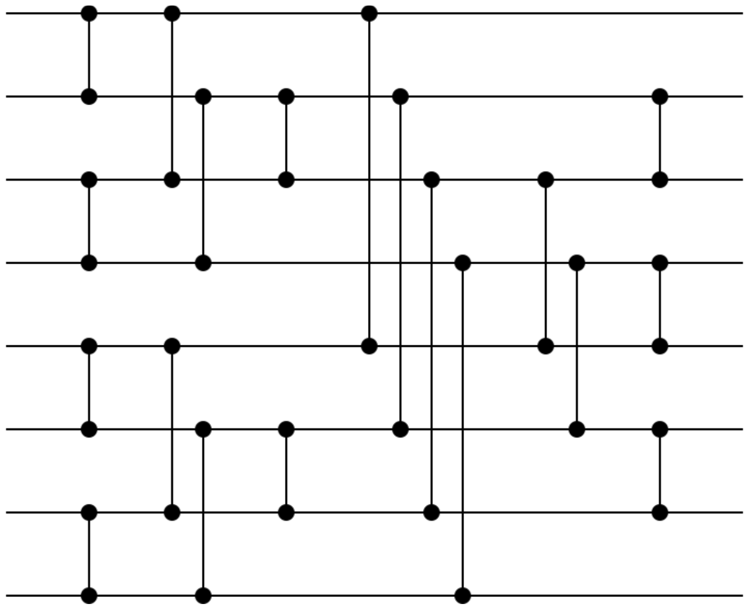
- Simple generalization of fixed approach
  - Replace letRank routine with generalized calcBucket, and pass it first C characters (however many needed to compute bucket)
- Works well with known minimum, maximum, and data distribution
  - Especially easy to get good bucketing with uniform random distribution
- Some of the fastest very large parallel sorts ever recorded use this method (e.g., world record setting Spark sort)
- Disadvantage – if you don't know the data distribution/guess wrong, you can get *very* bad load imbalance (one or a few cores doing most of the work).
  - Sometimes solve this with "sampling" to estimate distribution
- Requires knowing the absolute max and min (can just use INT64_MAX and INT64_MIN, but likely to lead to poorly distributed buckets).

Parallel Merge (sorting network based)

- Global array distributed among processors (CTA)
- Each processor locally sorts its piece of array (*must be equal-sized – see later*)
- Perform a series of merges according to some *sorting network*
  - Network specifies a partner for each processor at each step
  - Each proc sends its local array to its partner
  - Each proc does O(n/p) sorted merge of its array with partners array. One proc keeps low half, other keeps high half.
  - Can reduce communication by sending data as needed, *but in most real networks, one big message is better than many small messages*.



Networks



- Best real networks guarantee global sort after O($\log^2$ P) steps.
  - Batcher's odd-even mergesort →
  - Bitonic sort (textbook)
- AKS sorting network
  - Best asymptotic/theoretical time: O(log P) steps
  - But awful in practice – HUGE constants hidden by big-O notation
- Some instead optimize for "local communication", i.e., most merges are with "nearby" processors.
  - Benefit depends on interconnect characteristics, but it is easy to swap out sorting network

Important Point

- Sorting networks originally designed for single element nodes and compare-exchanges, *not* merging lists
- It can be proven that sorting networks will also work with a list at each node, and merging and splitting as we described, but the proof ***requires*** that all lists have the same length.
  - (See Knuth, vol. III, section 5.3, exercise 38)
- Therefore , we **must ensure that each partition has an equal number of elements**
  - (And that this invariant is maintained at each stage)
- Easy way to ensure: pad nodes with "dummy" values at start
  - Can also rebalence if necessary to reduce number of dummies
  - Need some way to mark dummies so that they can be removed later

Peril-L pseudocode…

- **Homework:** Write Peril-L for Batcher Odd-Even Mergesort
  - Can assume you start with even distribution (all nodes have same # of elements)

Which is better?

- Depends on circumstances
- Bucket-based sorting minimizes communication (CTA likes, real world too)
  - But bad if can't determine data distribution – can end up effectively serializing code.
- Parallel merge algorithms more generally applicable (don't depend on data distribution)
  - Higher communication costs
  - But, log(P) large messages per processor is *not bad*, especially on a good interconnect (highest bandwidth for large messages).
- Bottom line: I'd try hard to do bucketing – 1 or 2 rounds of communication is *nice*

Discussion

- Compare and constract: CTA vs LogP
  - The LogP paper claims they cover more of the important parameters (e.g., latency, bandwidth)
  - These are real aspects of networks
  - Do you agree that modeling them is necessary for coming up with efficient algorithms? Or is the simplicity of CTA (local vs. global) better? Does CTA capture enough of what's important?
  - Note, there is no right answer (people disagree)
- Memory Consistency
  - What surprised you, if anything?