Data and Task Parallelism

- Many definitions … parallelize the data or work?
- In a data parallel computation the parallelism is applied by performing the same (or similar) operations to different items of data at the same time; the parallelism grows with the size of the data
- In a task parallel computation the parallelism is applied by performing distinct computations -- or tasks -- at the same time; with the number of tasks fixed, the parallelism is not scalable

Contrast solutions to preparing a banquet

Peril-*L* …

- A pseudo-language to assist in discussing algorithms and languages
- Don't panic--the name is just a joke
- Goals:
  - Be a minimal notation to describe parallelism
  - Be universal, unbiased towards languages or machines
  - Allow reasoning about performance (using the CTA)

Base Language is C

- Peril-*L* uses C as its notation for scalar computation, but any scalar language is OK
- Advantages
  - Well known and familiar
  - Capable of standard operations & bit twiddling
- Disadvantages
  - Low level
  - No goodies like OO

This is not the way to design a || language

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
- The execution relationships among threads are 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 there; then they're all released

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
- Names (usually indexed) 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

Notice that the default vars are local vars

Memory Read Write Semantics

- Local Memory behaves like the RAM 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

In PRAM terminology, this is CREW, but it's not a PRAM

Example: Try 1

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

```
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)
      count++;
    }
}
```

Why Is This Not Shared Memory?

- Peril-*L* is not a shared memory model because:
  - It distinguishes between local and global memory costs … that's why it's called "global"
- Peril-*L* is not a PRAM because
  - It is founded on the CTA
  - By distinguishing between local and global memory, it distinguishes their costs
  - It is asynchronous

These may seem subtle but they matter

Getting Global Writes Serialized

- To insure the exclusive write Peril-*L* has

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

- The semantics are that 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

Exclusive gives behavior, not mechanism

Example: Try 4

- The final (correct) Count 3s program

```
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; }
}
```

Padding is irrelevant … it's implementation

Full/Empty Memory

- 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 w/value
Full	Empty of value	Stall

- Scheduling stalled threads may not be fair

We'll find uses for this next week

Reduce and Scan

- Aggregate operations use APL syntax
  - Reduce: <*op*>/<*operand*> for <*op*> in {+, *, &&, ||, max, min}; as in `+/priv_sum`
  - Scan: <*op*>\<*operand*> for <*op*> in {+, *, &&, ||, max, min}; as in `+\local_finds`
- To be portable, use reduce & scan rather than programming them

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

Reduce/Scan Imply Synchronization

Reduce/Scan and Memory

- When reduce/scan involve local memory

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

  - The local is assigned the global sum
  - This is an implied broadcast

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

  - The local is assigned the prefix sum to that pt
    - So order (of the forall) matters
  - No implied broadcast

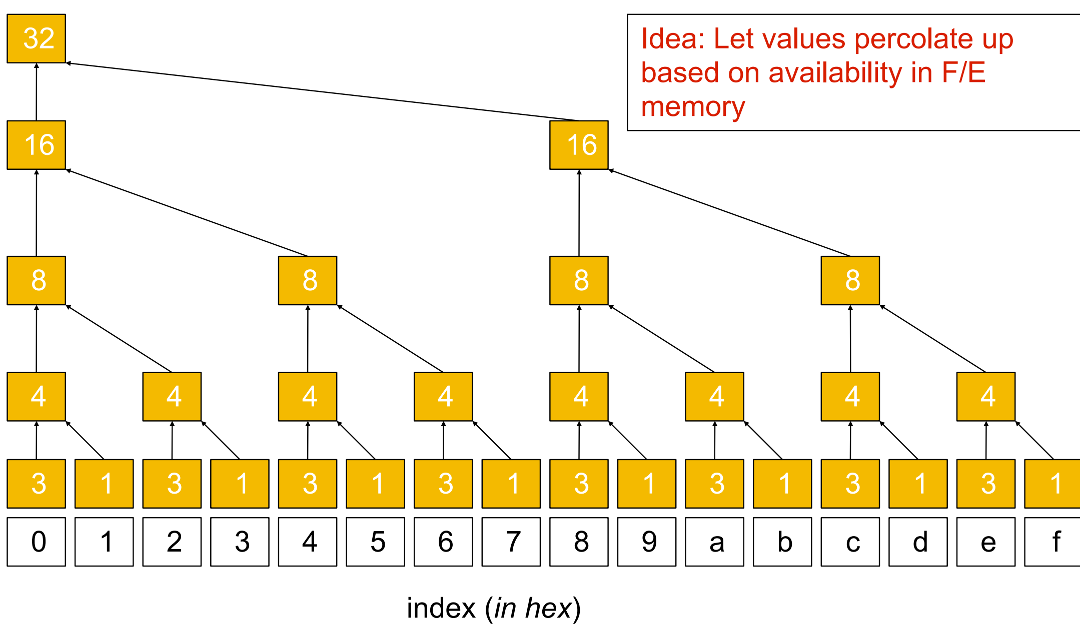
Peril-*L* Summary

- Peril-*L* is a pseudo-language
- No implementation is implied, though performance is
- Discuss: How efficiently could Peril-*L* run on previously discussed architectures?
  - CMP, SMPbus, SMPx-bar, Cluster, BlueGeneL
  - Features: C, Threads, Memory (G/L/f/e), /, \

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

Slick Tree Accumulate Using F/E



Idea: Let values percolate up based on availability in F/E memory

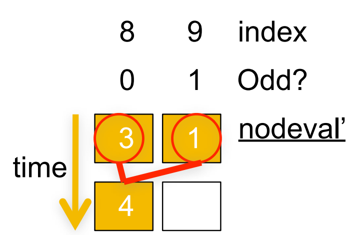
Naïve F/E Tree Accumulation

```
int nodeval'[P];                     Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
 int val2accum; int stride = 1;      val2accum: locally computed val
 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
  }
 }
}
```

Caution: This implementation is wrong ...

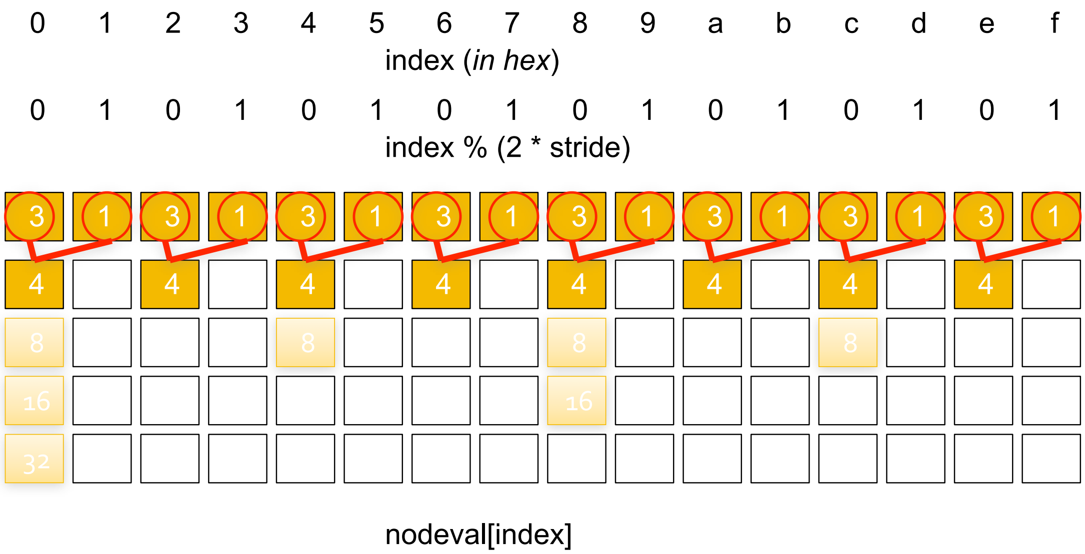
Naïve F/E Tree Accumulation

```
int nodeval'[P];                     Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
 int val2accum; int stride = 1;      val2accum: locally computed val
 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
  }
 }
}
```

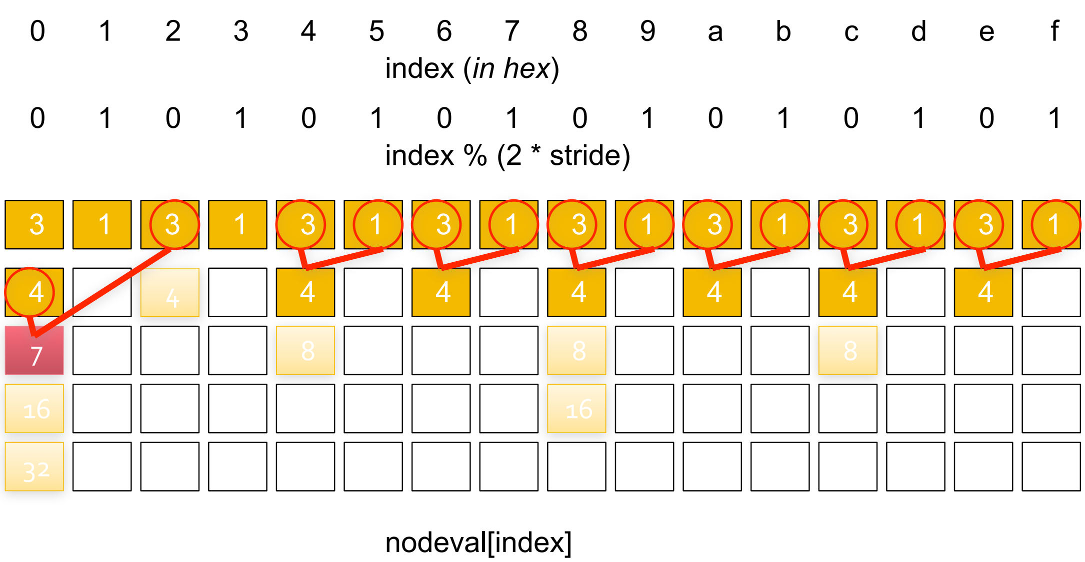
Caution: This implementation is wrong ...

Round 1 of Tree Accum …



Caution: This implementation is wrong …

But What If P_2 is Slow, P_0 Fast?



Caution: This implementation is wrong …

Introduce Barrier to Synch Levels

```
1 int nodeval'[P];                          Global full/empty vars to save right child val
2 forall ( index in (0..P-1) ) {
3  int val2accum; int stride = 1;           val2accum: locally computed val
4  nodeval'[index] = val2accum;             Assign initially to tree node
5  while (stride < P) {                     Begin logic for tree
6   if (index % (2*stride) == 0) {
7    nodeval'[index]=nodeval'[index]+nodeval'[index+stride];
8    stride = 2*stride;
9   }
10  else {
11   break;   Exit, if not now a parent
12  }
12.5 barrier;
13 }
14 }
```

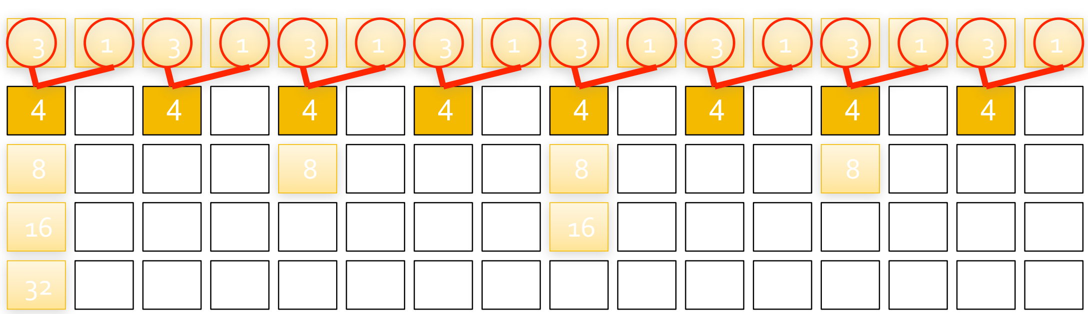
Barrier Stops Until Stable State

0 1 2 3 4 5 6 7 8 9 a b c d e f

index (*in hex*)

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

index % (2 * stride)



nodeval[index]

The Problem With Barriers

- In many places barriers are essential to the logic of a computation, but …
- In many cases they are just an implementational device to overcome (for example) false dependences
- Avoid them when possible
  - They force the ||-ism to drop to zero
  - Often costly even when all threads arrive at once

Asynchronous Tree Accumulate

```
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
   }
  }
}
```

The "full" Applies To Root Only

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

index (*in hex*)

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

index % (2 * stride)

3	1	2	2	2	2	3	1	2	2	3	1	3	1	2	2
4		4		4		4		4		4		4		4	
8				8				8				8			
16								16							
32															

nodeval[index]

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

Thinking About Parallel Algorithms

- Computations need to be reconceptualized to be effective parallel computations
- Three cases to consider
  - Unlimited parallelism -- issue is grain
  - Fixed ||ism -- issue is performance
  - Scalable parallelism -- get all performance that is *realistic* and *build in flexibility*
- Consider the three as an exercise in
  - Learning Peril-*L*

The Problem: Alphabetize

- Assume a linear sequence of records to be alphabetized
- Technically, this is parallel sorting, but the full discussion on sorting must wait
- Solutions
  - Unlimited: Odd/Even
  - Fixed: Local Alphabetize
  - Scalable: Batcher's Sort

Unlimited Parallelism (O/E Sort, 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 (O/E Sort, 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?
 }
}
```

Recall …

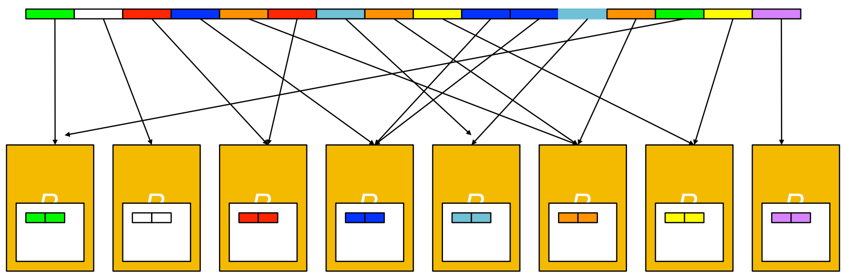
- We are illustrating the Peril-L notation for writing machine/language independent parallel programs
  - The "unlimited parallel solution" is O/E Sort
    - All data references were to global data
    - Threads spawned for each half step
    - Ineffective use of parallelism requiring threads to be created and implemented literally
  - Now consider a "fixed parallel solution"

Fixed Algorithm

- Postulate a process for handling each letter of the alphabet -- 26 Latin letters
- 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

Cartoon of Fixed Solution

- Move locally



- Sort
- Return

Fixed Part 1: Introduce 2 functions

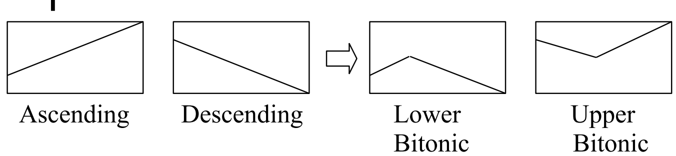
```
rec L[n];                                    The data is global
forall (index in (0..25)) {                  A thread for each letter
 int myAllo = mySize(L, 0);                  Number of local items
 rec LocL[] = localize(L[]);                 Make data locally ref-able
 int counts[26] = 0;                         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 no. 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 αbetize
 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];
}
}
```

Scalable Sort

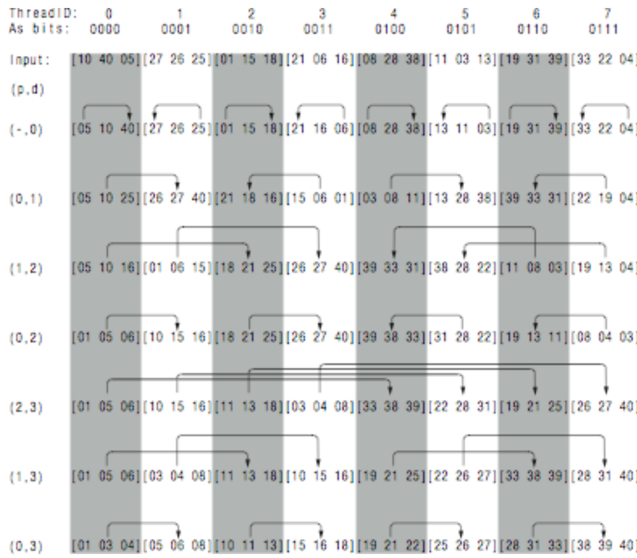
- Batcher's algorithm -- not absolute best, but illustrates a dramatic paradigm shift
- Bitonic Sort is based on a bitonic sequence:
- a sequence with increasing and decreasing subsequences



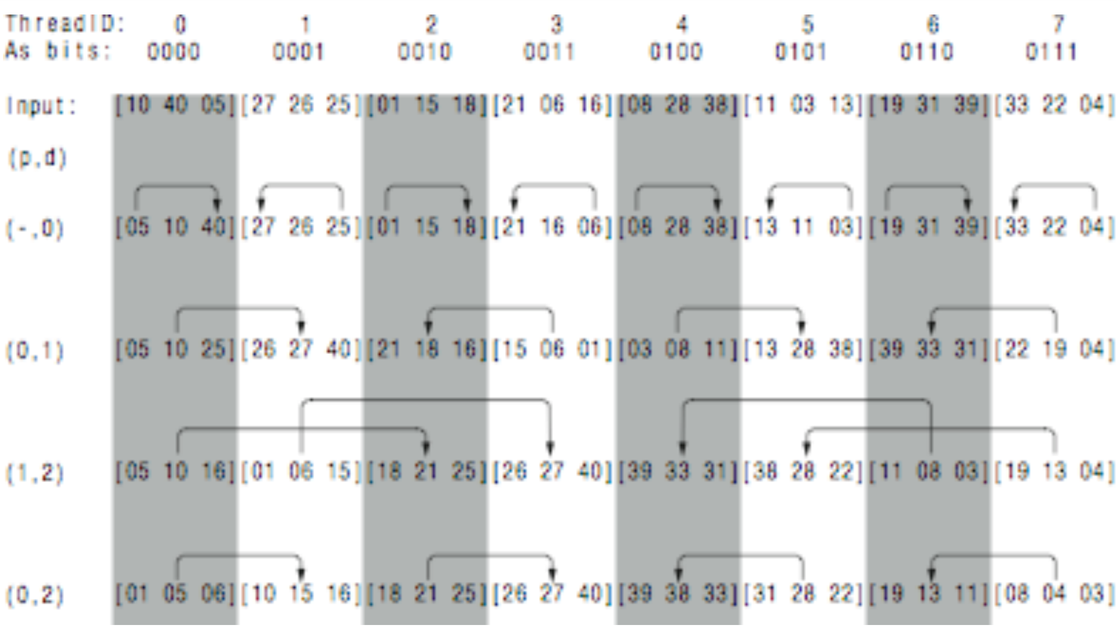
- Merging 2 sorted sequences makes bitonic

Batcher's Sort

- Skip recursive start; start w/ local sort
- Control by thread ID of paired processes
- (*p*,*d*) controls it: start at (-,0), *d* counts up, *p* down from *d*-1
- *p* = process pairs
- *d* = direction is d^{th} bit



Bitonic Sort, Closer Look

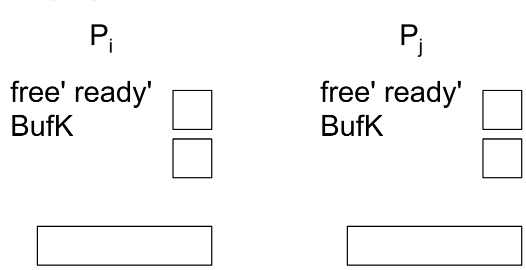


Logic of Batcher's Sort

- Assumption: 2^x processes, ascending result
- Leave data in place globally, find position
  - Reference data locally, say *k* items
  - Create (key, input position) pairs & sort these
  - Processes are asynch, though alg is synchronous
  - Each process has a buffer of size *k* to exchange data -- write to neighbor's buffer
  - Use F/E var to know when to write (other buffer empty) and when to read (my buffer full)
  - Merge to keep (lo or hi) half data, and insure sorted
  - Go till control values end; use index to grab original rec

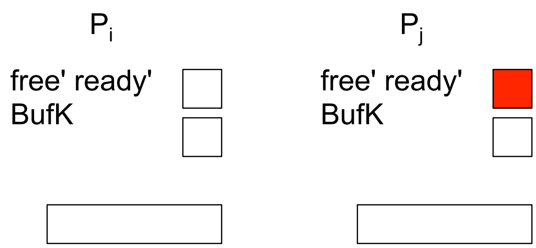
Data Transfer

- Use one buffer per processor plus two F/E variables: free' and ready'
  - free' is full when neighbor's buffer can be filled
  - ready' is empty until local buffer is filled



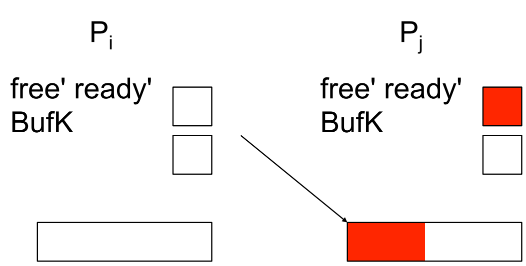
Data Transfer

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor’s buffer can be filled
  - ready' is empty until local buffer is filled



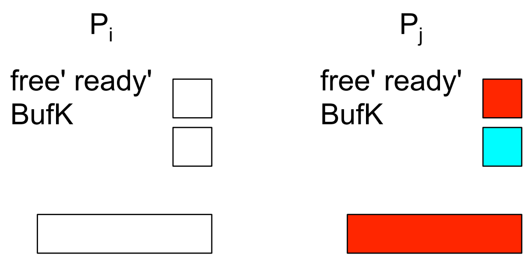
Data Transfer

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor’s buffer can be filled
  - ready' is empty until local buffer is filled



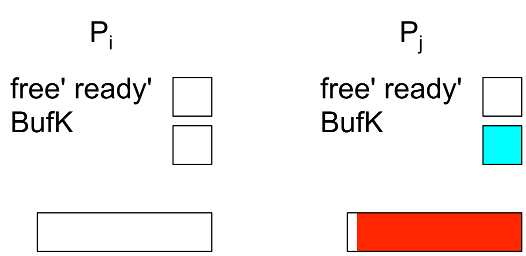
Data Transfer

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor’s buffer can be filled
  - ready' is empty until local buffer is filled



Data Transfer

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor’s buffer can be filled
  - ready' is empty until local buffer is filled



Details on Data Transfer

```
alphabetizeInPlace(K[],bit(index,0));  Local sort, up or
                                        down  based on bit 0
for(d=1; d<=m; d++)  {                  Main loop, m phases
 for(p=d-1; p<0; p--)  {                Define p for each sub-phase
   stall=free'[neigh(index,p)];         Stall till I can give data
   for(i=0; i<size; i++)  {             Send my data to neighbor
     BufK[neigh(index,p)][i]=K[i];
   }
   ready'[neigh(index,p)]=true;         Release neighbor to go
   stall=ready'[index];                 Stall till my data is ready
   … Merge two buffers, keeping half
  }
}
```

Bitonic Sort In Text

- Details are in the book …
- Discussion Question: What, if any, is the relationship between Bitonic Sort and Quick Sort?

- http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm

Sample Sort

- The idea of sending data to where it belongs is a good one … the Fixed Solution works out where that is, and Batcher's Sort uses a general scheme
- Can we figure this out with less work?
  - Estimate where the data goes by sampling
  - Send a random sampling of a small number (log *n*?) of values from each process to p_0
  - p_0 sorts the values and picks the *P*-1 "cut points",

Sample size depends on the values of *n* and *P*

Sample Sort (Continued)

- After receiving the "cut points" each process…
  - Sends its values to the process responsible for each range
  - Each process sorts
  - A scan of the actual counts can place the "cut points" into the right processes
  - An adjustment phase "scooches" the values into final position

Summary

- Peril-L is a useful notation for sketching a solution – you will probably implement it w/o much language support
  - Ideally, we should have language support
  - Hopefully, it helps working out subtle points, like synchronization behavior
- In algorithm design, maximizing parallelism is much less important that minimizing process-interactions