Peril-L Reduce and Scan: reduce (e.g., global sum) or scan (e.g., parallel prefix sum) allowed for any associate and commutative primitive operation: +, *, && (and), || (or), max, and min. A slash (/) for reduction and backslash (\) for scan.

Example	Description
`least = min/dataArray;`	Minimum across global `dataArray` stored in local `least` of each thread
`total = +/count;`	Sum of local `count` at each thread reduced and stored in each threads local variable `total`
`beforeMe = +\count;`	Parallel prefix sum based on threads' index values, i.e., i^{th} thread's `beforeMe` variable assigned the sum of the first i `count` values

NOTE:

- Implied reduce and scan barrier synchronization when they reference and assign to only local variables. For example in `total = +/count;` all threads must reach this statement and the assignment complete at all threads before any are allowed to continue. If the result is stored global (`globalTotal = +/count;`), a thread contributes its total and immediately continues executing without a barrier synchronization.
- You should performing global reductions by `globalTotal = +/count;`, instead of equivalent `exclusive { globalTotal = +/count;}` to aid compiler/human-coder recognize that +-reduction hardware or O(log P) tree algorithm can be used to avoid the O(P) serialization of mutual exclusion.

1. Count 3s Try 3 written in Peril-L notation (Figure 4.1)

```
 1  int array[length];                          The data is global
 2  int t;                                      Number of desired threads
 3  int total=0;                                Result of computation, grand total
 4  int lengthPer=ceil(length/t);
 5  forall(index in(0..t-1))
 6  {
 7    int priv_count=0;                         Local accumulation
 8    int i, myBase=index*lengthPer;
 9    for(i=myBase; i<min(myBase+lengthPer, length); i++)
10    {
11      if(array[i]==3)                         There's no concurrent read since
12      {                                       Array has been partitioned
13        priv_count++;
14      }
15    }
16    exclusive { total+=priv_count; }          Compute grand total
17  }
```

a) How can we improve on this code using `mySize` function, `localize` functions, and reduction (+/) ?

b) Why would we not want to write our parallel program to a fixed parallelism (e.g., hard code t = 8) to work on our current machine?

c) Why would we not want to write our parallel program assuming an infinite parallelism (e.g., P = n)?

d) Our goal is *scalable parallelism* when formulating our parallel algorithms. After determining how the components of the problem (data structures, work load, etc.) grow as the computation's size, n, grows, we formulate a set *S* of *substantial* subproblems in which *natural* units of the solution, of size *s*, are assigned to each subproblem and solved as *independently* as possible.

- Why would we want each thread to have a substantial amount of local?

- Why would we want each thread to has a natural unit (e.g., whole row of a matrix) of the subproblem?

- Why would we want each subproblem to be as independent as possible?

Alphabetizing Example: Want to alphabetize an array of records (L[n]) on some field x.

Helper Function	Description
`strcmp(str1, str2)`	Compares null-terminated strings **`str1`** and **`str2`** returning a value less than, equal to, or greater than zero; as found in **`string.h`**
`charAt(str, pos)`	Returns the character in the **`pos`** position (0-origin); as in JavaScript
`letRank(chr)`	Returns the rank (0-origin) of its argument letter
`alphabetizeInPlace()`	Reorders the argument array of records to be alphabetical

2. Unlimited Parallelism solution uses Odd/Even Exchange Sort

```
 1  bool continue=true;
 2  rec L[n];                                  The data is global
 3  while(continue) do
 4  {
 5    forall(i in(1:n-2:2))                    Stride by 2
 6    {
 7      rec temp;
 8      if(strcmp(L[i].x,L[i+1].x)>0           Is odd/even pair misordered?
 9      {
10        temp=L[i];                           Yes, fix
11        L[i]=L[i+1];
12        L[i+1]=temp;
13      }
14    }
15    forall(i in(0:n-2:2))                    Stride by 2
16    {
17      rec temp;
18      bool done = true;                      Set up for termination test
19      if(strcmp(L[i].x,L[i+1].x)>0)  Is even/odd pair misordered?
20      {
21        temp=L[i];                           Yes, interchange
22        L[i]=L[i+1];
23        L[i+1]=temp;
24        done=false;                          Not done yet
25      }
26      continue=!(&&/done);                   Were any changes made?
27    }
28  }
```

a) Trace on small array, L:

0	1	2	3	4	5	6	7
20	35	15	10	50	30	5	40

b) Which sequential sort is this most like?

c) How much parallelism exists?

d) How much global copying occurs?

3. Fixed Parallelism solution uses 26 threads -- one for strings starting with each letter of alphabet:

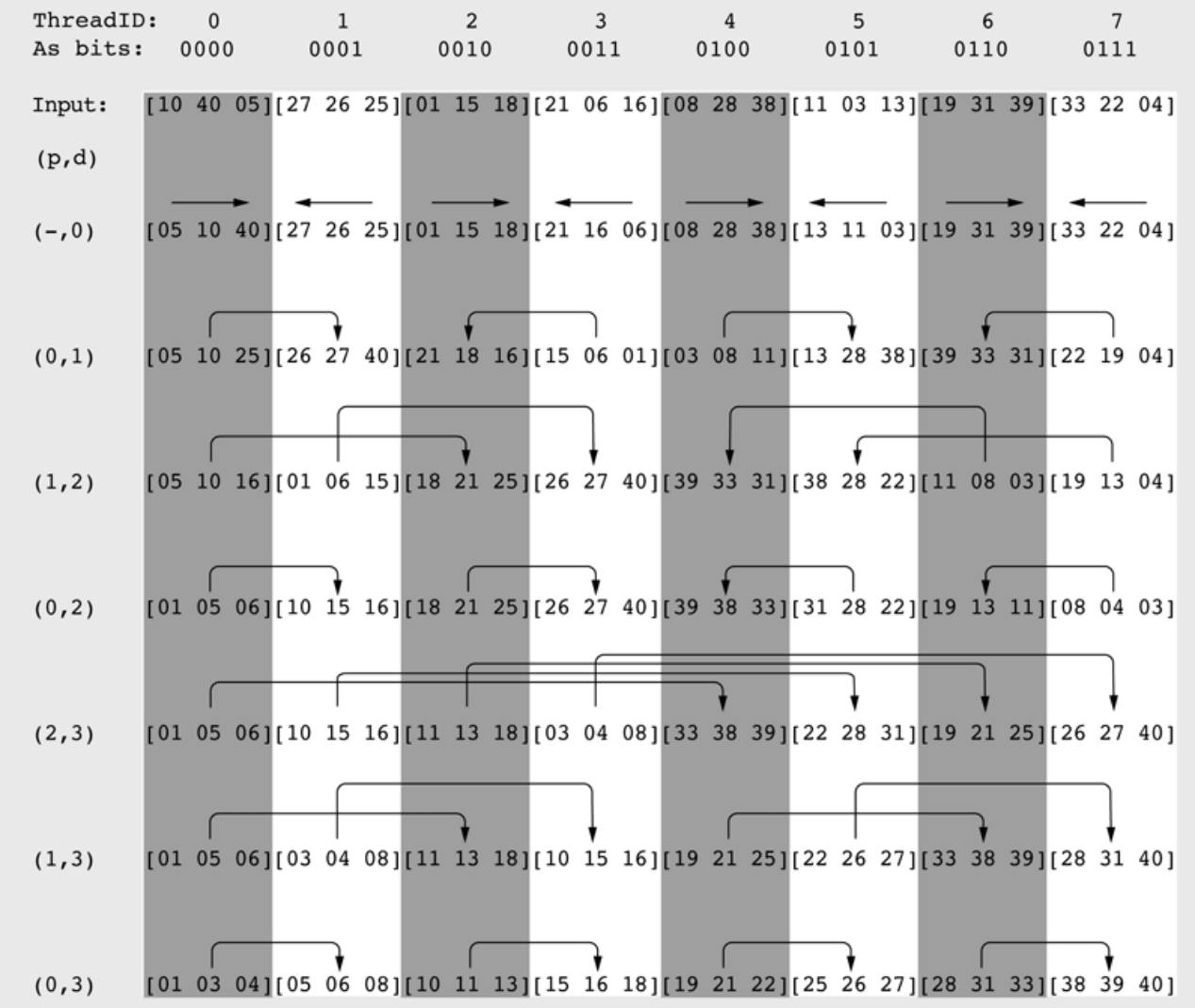
```
 1  rec L[n];                                  The data is global
 2  forall(j in(0..25))                        A thread for each letter
 3  {
 4    int myAllo=mySize(L, 0);                 Number of local items
 5    rec LocL[]=localize(L[]);                Make data locally referenceable
 6    int counts[26]=0;                        Count number of each letter
 7    int i, j, startPt, myLet;
 8    for(i=0; i<myAllo; i++)                  First, count number w/each letter; need this
 9    {
10      counts[letRank(charAt(LocL[i].x,0))]++;
11    }
12    counts[index]=+/counts[index];           Figure how many of each letter
13    myLet=counts[index];                     Number of records of my letter
14    rec Temp[myLet];                         Allocate local storage for records
16    j=0;                                     Index for local array
17    for(i=0; i<n; i++)                       Move records locally for local alphabetize
18    {
19      if(index==letRank(charAt(L[i].x,0)))
20      {
21        Temp[j++]= L[i];                     Save record locally
22      }
23    }
24    alphabetizeInPlace(Temp[]);              Alphabetize within this letter locally
25    startPt=+\myLet;                         Scan counts # records ahead of these; scan
                                               synchs, so okay to overwrite L, once sorted
26    j=startPt-myLet;                         Find my starting index in global array
27    for(i=0; i<count; i++)                   Return records to original global memory
28    {
29      L[j++]=Temp[i];
30    }
31  }
```

a) What's the general idea of this solution?

b) How much parallelism exists?

c) How much global copying occurs?

4. Scalable Parallelism solution uses Batcher's Bitonic Sort -- a parallel version of merge sort. Figure 4.6 example with n = 24 and t = 8 (with a thread index expressed in binary $b_3b_2b_1b_0$). Initially, each thread locally sorts their records (-,0) step. Then, threads merge their sequences according to a protocol specified by two parameters (p,d) where p indicates the pairs to merge, and d indicates the direction of the order. Threads whose ID # differing only in binary bit position p are paired to merge. If the d^{th} bit of the thread index $b_d = 0$, then the sort is ascending; otherwise (i.e., $b_d = 1$) the sort is descending.



a) How much parallelism exists?

b) How much global copying occurs?

Operation	How Index Is Related to Other of Pair	What to Keep from Merge
Merge Up	Smaller Index, $b_r \ldots b_{p+1} 0 b_{p-1} \ldots b_0$	Keep Lower Half
	Larger Index, $b_r \ldots b_{p+1} 1 b_{p-1} \ldots b_0$	Keep Upper Half
Merge Down	Smaller Index, $b_r \ldots b_{p+1} 0 b_{p-1} \ldots b_0$	Keep Upper Half
	Larger Index, $b_r \ldots b_{p+1} 1 b_{p-1} \ldots b_0$	Keep Lower Half

```
 1  int t;                                          Thread count, must be 2^m
 2  int m=log2(t);                                  Exponent of thread count
 3  rec L[n];                                       Records to be alphabetized
 4  int size=n/t;                                   Local portion; assume divisibility
 5  key BufK[m][size];                              Buffer to pass keys through
 6  bool free'[m] = false; ready'[m];               Full/empty variable to manage buffers
 7  forall(index in(0..t-1))                        Start threaded section
 8  {
 9    int i, d, p; bool stall;
10    rec LocL[size]=localize(L[]);                 Map global to local
11    rec inputCopy[size];                          Local copy of values simplifying synch
12    key Kn[size]=localize(BufK[]);                Map global buffer to local for fast access
13    key K[size];                                  Working sequence array
14    for(i=0; i<size; i++)                         Compress just to keys
15    {
16      K[i].x=LocL[i].x;                           Save letter string
17      K[i].home=localToGlobal(LocL,i,0);          Remember global index
18    }
19
20    alphabetizeInPlace(K[],bit(index,0));         Locally sort, up or down based on bit 0
21    for(d=1; d<=m; d++)                           Main loop, m phases
22    {
23      for(p=d-1; p<0; p--)                        Define p for each sub-phase
24      {
25        stall=free'[neigh(index,p)];              Stall till I can give data
26        for(i=0; i<size; i++)                     Send my data to my neighbor for this step
27        {
28          BufK[neigh(index,p)][i]=K[i];           Send mine to neighbor
29        }
30        ready'[neigh(index,p)]=true;              Release neighbor to compute
31        stall=ready'[index];                      Stall till my data is available
32        if(bit(index,d)==0)                       What direction to sort?
33        {
34          for(i=0; i<size; i++)                   Merge Up: move earlier data to the lower
35          {                                       index thread
36            if(bit(index,p)==0)                   Lower thread of pair
37            {
38              if(strcmp(Kn[index][i].x, K[i].x)>0)
39              {
40                K[i]=Kn[i];
41              }
42            }
43            else                                  Upper thread of pair
44            {
45              if(strcmp(Kn[index][i].x, K[i].x)<0)
```

```
46              {
47                   K[i]=Kn[i];
48              }
49            }
50          }
51        }
52        else
53        {
54          for(i=0; i<size; i++)              Merge Down: move earlier
55          {                                  data to higher index thread
56            if(bit(index,p)==1)              Lower thread of pair
57            {
58              if(strcmp(Kn[index][i].x, K[i].x)>0)
59              {
60                K[i] = Kn[i];
61              }
62            }
63            else                             Upper thread of pair
64            {
65              if(strcmp(Kn[index][i].x, K[i].x)<0)
66              {
67                K[i]=Kn[i];
68              }
69            }
70          }
71        }
72        alphabetizeInPlace(K[],bit(index,p));    Locally sort, up/dn based on bit p
73        free'[index]=true;                 Finished w/ buffer, enable
74      }                                    End of sub-phase loop
75    }                                      End of phase loop
76    for(i=0; i<size; i++)                  Grab records belonging with this thread
77    {
78      inputCopy[i]=L[K[i].home];           Get record and save locally
79    }
80    barrier;                               Wait until everyone is done
81    for(i=0; i<size; i++)1                 Make output available
82    {
83      LocL[i]=inputCopy[i];                Make records globally available
84    }
85  }
```