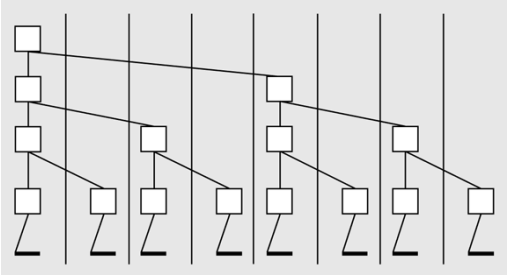
Chapter 5: Scalable Algorithmic Techniques - focuses on data parallelism instead of task parallelism since task parallelism does scale well with P (i.e., the number of tasks determines the parallelism -- however, each task can will often have data parallelism)

Guiding principle:
Parallel programs are more scalable when they emphasize blocks of computation -- typically the larger the block the better -- that minimize the inter-thread dependencies.

Schwartz' Algorithm - applies this guiding principle to tree operation (e.g., +-reduction) where P < n.
a) Which of the following approaches would be better?

Approach (1)	Approach (2)
• Create n/2 logically threads evenly across the P processors, so each processor had n/2P threads. • Have the logical threads perform the binary reduction	• Have each processor sum its n/P local values • Have the P processors perform the binary reduction on a P local sums

Figure 5.1 illustrates Schwartz' Algorithm for tree operation (e.g., +-reduction)



Notes:
- bold dark lines denote local computations

Figure 5.2 showing Peril-L algorithm for Schwartz' Algorithm

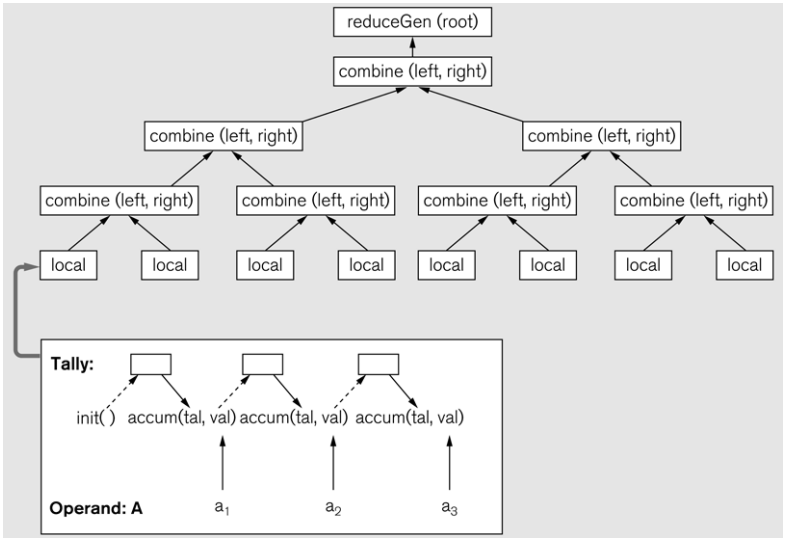
```
 1  int nodeval'[P];                          Global full/empty variables for
 2                                            saving the value from right child
 3  forall(index in(0..P-1))
 4  {
 5    int tally;
 6    stride=1;
 7    ...                                     COMPUTE tally HERE
 8    nodeval'[index]=tally;                  Send initially to tree node
 9
10    while(stride<P)                  Begin logic for tree
11    {
12      if(index%(2*stride)==0)
13      {
14        nodeval'[index]=nodeval'[index]+
15                        nodeval'[index+stride];
16        stride=2*stride;
17      }
18      else
19      {
20        break;                                Exit, if no longer a parent
21      }
22    }
23  }
```

b) Explain how nodeval'[P] is used to synchronize and communicate between threads.

2. Author's claim that reduction and scan abstractions can be generalized to perform more operations than the simple +, *, &&, ||, min, and max.

- **Reduce** - combines a set of values to produce a single value. Frequently needed in parallel computations to compare or combine results produced by different threads to either summarize the computation or to control its execution. Examples of problems solvable by the generalized reduce operation:
  - Find the second smallest array element.
  - Given an array of values, compute a histogram with k intervals.
  - Given a binary array, compute the longest run of consecutive 1s.
  - Given an array, find the index of the first occurrence of some target value.
- **Scan** (parallel prefix computation) - embodies the logic that performs a sequential operation in parts and carries along the intermediate results. Loop iterations often appear to be sequential because they accumulate information in order as they iterate, but a scan can often be used to enable more parallelism.

Structure of Generalized Reduce:



Local Computation functions:

init() - initializes the *tally* in preparation for local computation

accum() - performs a local accumulation of an operand element into the *tally*

Tree Computation functions:

combine() - composes the intermediate tally results from its two subtrees and passes the result to its parent

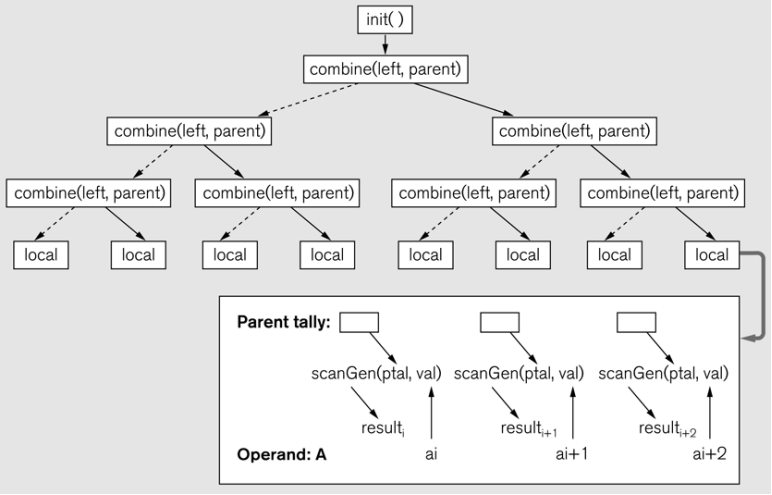
reduceGen() - takes the global result and generates the final answer of the reduce

```
 1  struct tally
 2  {
 3    float smallest1;                          Smallest element
 4    float smallest2;                          Second smallest
 5  };
 6
 7  tally init()                                Initialize tally
 8  {
 9    tally t;
10    t.smallest1=MAX_FLOAT;
11    t.smallest2=MAX_FLOAT;
12    return t;
13  }
14
15  tally accum(tally t, float elem)            Local accumulation
16  {
17    if(t.small1>elem)                         Is this a new smallest?
18    {
19      t.smallest2=t.smallest1;
20      t.smallest1=elem;
21    }
22    else
23    {
24      if(t.smallest2>elem)                    Is it a new second smallest?
25      {
26        t.smallest2=elem;
27      }
28      return t;
29    }
30  }
31
32  tally combine(tally left, tally right)      Combine into "left" by
33  {                                           accumulating right values
34    tally t;
35    t=accum(left, right.smallest1);
36    t=accum(t, right.smallest2);
37    return t;
38  }
39
40  float reduceGen(tally t)
41  {
42    return t.smallest2;
43  }
```

Example for "Find the second smallest array element."

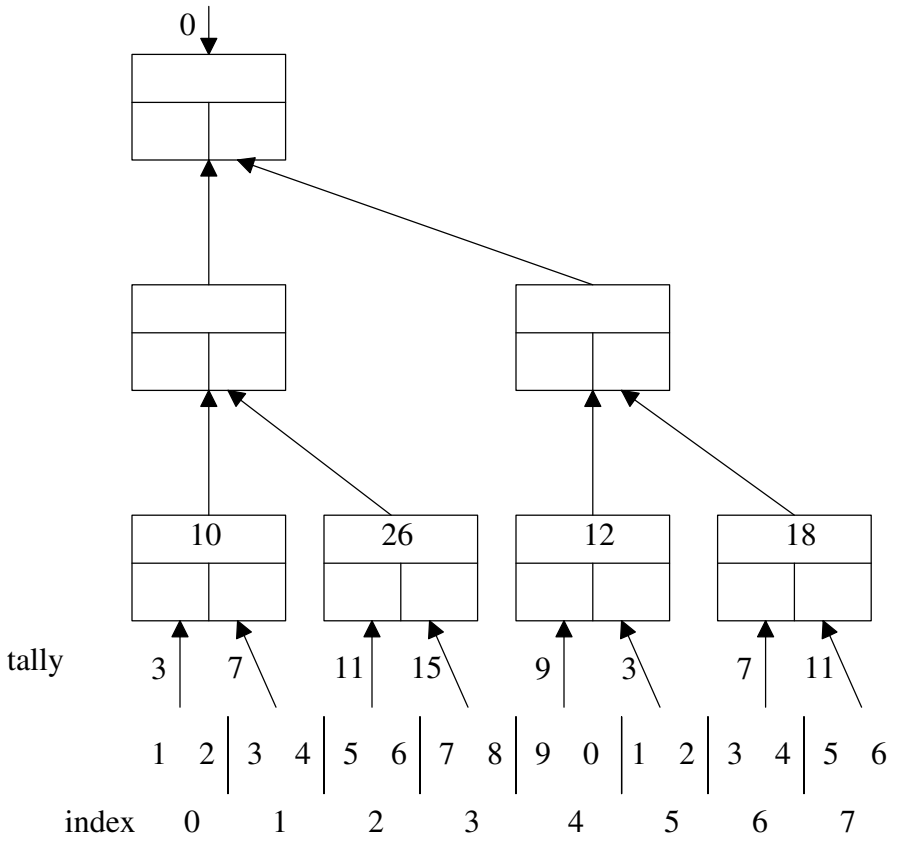
a) How would you formula these functions for the "Given a binary array, compute the longest run of consecutive 1s." problem?

Generalized Scan: Similiar to the reduce, but after the combining is complete the intermediate results must be passed back down the tree to complete the prefix computation on local values.



Trace the scan code using the picture on next page using below. Assume +\ - scan

```
 1  int nodeval'[P];                                  Global full/empty memory
 2  int ltally[P];                                    Store left operand of combine
 3  forall(index in(0..P-1))
 4  {
 5    int myData[size]=localize(operandArray[]);      Local data values
 6    int tally;                                      Tally
 7    int ptally;                                     Tally from parent
 8    int stride=1;
 9    tally=init ();                                  Initialize
10    for(i=0; i<size; i++)
11    {
12      tally=accum (tally, myData[i]);               Accumulate
13    }
14    nodeval'[index]=tally;                          Send initially to parent
15    while(stride<P)                                 Begin logic for tree
16    {
17      if(index%(2*stride)==0)
18      {                                             Combine
19        ltally[index+stride]=nodeval'[index];
20        nodeval'[index]=combine (ltally[index+stride],
21                                 nodeval'[index+stride]);
22        stride=2*stride;
23      }
24      else
25      {
26        break;
27      }
28    }
29    stride=P/2;
30    if(index==0)
31    {
32      ptally=nodeval'[0];                           Clear existing up sweep value
33      nodeval'[0]=init ();                          Set init() as parent input
34    }
35    while(stride>1)                                 Begin logic for tree descent
36    {
37      ptally=nodeval'[index];                       Grab parent value
38      nodeval'[index]=ptally;                       Send it down to left
39      nodeval'[index+stride]=                       Send parent + left child right
          combine (ptally, ltally[index+stride]);
40      stride=stride/2;                              Go down to next level
41    }
42    for(i=0; i<size; i++)
43    {
44      myResult[i]=scanGen (ptally, myData[i]);      Generate Scan
45    }
46  }
```



(after line 14) nodeval'

stride 1 ltally

nodeval'

stride 2 ltally

nodeval'

stride 4 ltally

nodeval'

ptally

stride 4 ltally

nodeval'

stride 2 ltally

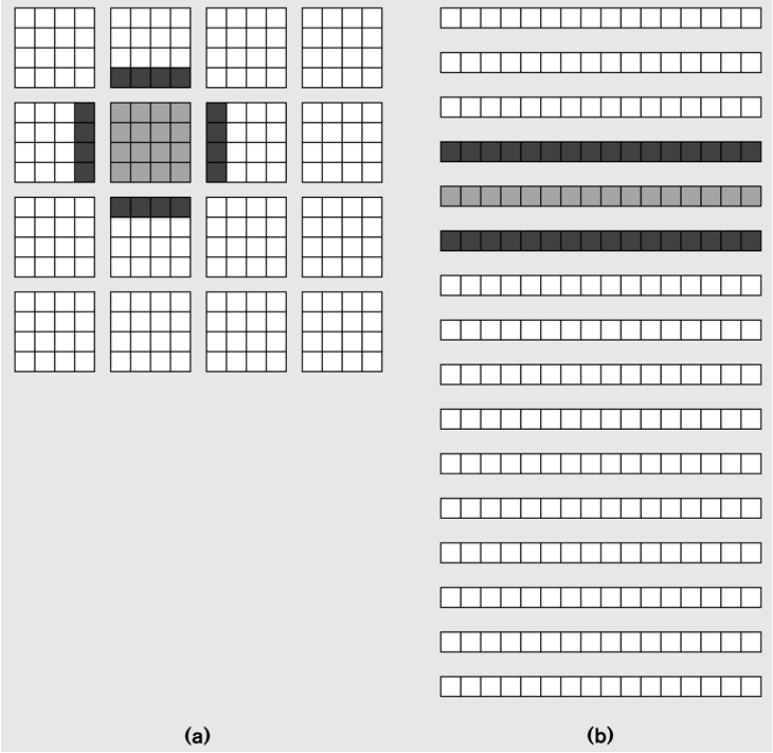
nodeval'

stride 1 ltally

nodeval'

Static Allocation of Data and Work: - allocate each thread its own block of data and the responsibility for computations on its data block.

Block Allocations:



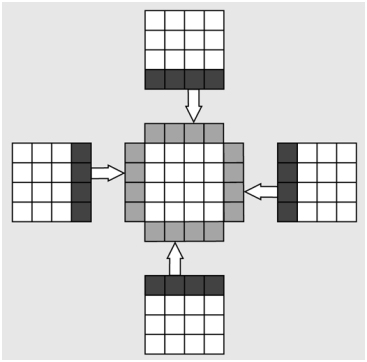
(a) 16 x 16 array to 16 processes by using two-dimensional blocks

(b) 16 x 16 array to 16 processes by rows

1. If the computation at each element is a *stencil computation* involving its four neighbors, then the dark elements show the nonlocal values needed.

a) Which allocations would require fewer elements being communicated (i.e., "surface area to volume advantage")?

To handle *overlapping regions* of a stencil computation, the following three steps are recommended:

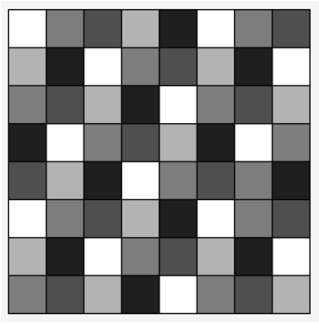


- When allocating space locally for the data block, allocate space for the overlapping regions.
- Fill the overlapping region by obtaining values from them from other processes
- Perform the computations on entirely local data

b) For distributed memory computers, the data transmission takes $t_o + d\ t_b$, where t_o is the setup overhead, d is the number of bytes, and t_b is the time per byte. Why is the above procedure for handling overlapping regions better than requesting each nonlocal element separately?

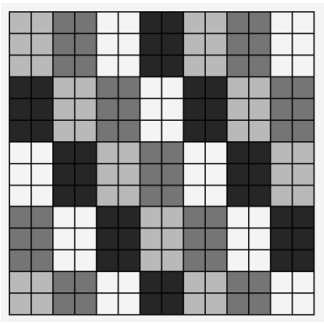
c) For shared memory computers, why does this procedure reduce communication costs by moving entire cache lines?

2. Cyclic Allocation: If the amount of work per block varies, then block allocation can cause poor load balancing. Cyclic allocation - allocates elements to a process in a round-robin fashion. For example 8 x 8 array to 5 processes:



a) What might be a disadvantage of cyclic allocation?

b) Block-cyclic allocation is a combination of the two. It allocates blocks in a round-robin fashion to the processes. For example, 14 x 14 array allocated in 3x2 size blocks to 4 processes. What advantages does a block-cyclic allocate have?



Irregular Allocations - not everything is best modeled by an array. For example, unstructured grids of irregularly shaped triangles in finite element computations. (e.g., fluid dynamics of an airplane wing)