Reduce & Scan Solution Set

1. **Reduce vs Scan (5 points)**. Assume that we have some data (call it the *input*) distributed over a collection of worker processes (call them the *processes*) and we wish to compute some function of this data to produce some other data (call it the *output*). Our local parallel computing guru has assured us can be implemented with either a single reduce or a single scan operation. Briefly describe what properties of the *input*, *processes* and/or *output* would allow us to determine whether we should use a reduce or a scan.

   Solution: The distinguishing feature is that the *output* of a scan operation will be similar in size to the *input* (or perhaps even larger), while the *output* of a reduce operation is typically of constant size (often just a single value) no matter how large the *input*.

   It is true that a scan operation requires both an upward and a downward pass whereas a reduce can be completed in a single upward pass; however, the presence of upward and downward passes does not necessarily imply that a scan occurred; for example, in the `count3s()` reduce code presented in class the results of the reduction were broadcast back down the tree to all of the leaf nodes but that does not mean we completed a scan.

2. **Generalized reduce (10 points)**. As described in *Lin & Snyder* a generalized reduce is described by four functions: `init()`, `accum()`, `combine()` and `reduceGen()`. Assume that we have 1600 data elements to process and that we set up as balanced a tree as possible.

   (a) If we can give 100 data elements to each leaf process, what is the height of the tree (which is also the maximum number of messages that need to be passed to get between a leaf and the root)? What about if we can give only 60 data elements to each leaf process?

   Solution: Height for 100 / leaf: **4** 60 / leaf: **5**.

   - If we give 100 data elements to each leaf process, then the fully saturated binary tree of height 4 will be the most balanced tree we can set up. We start with a chunk of 1600 elements at the root, then divide into chunks of 800, then 400, 200 and finally 100. We will have 16 leaf processes each working on 100 elements. Here, the tree is perfectly balanced since for any node, the height of its left subtree and its right subtree are the same. Thus we cannot do better.
   - If the max capacity for each leaf process is 60 elements, then a tree of height 4 with 16 nodes can process at most $60 * 16 = 960$ elements and we must go to a tree of height 5. There are several ways to build such a tree. We could build a balanced tree of height 5 by dividing the 1600 by 2 each time: 800, 400, 200, 100 and then 50 to each of 32 leaf nodes. Or we could build an unbalanced tree with $\lceil 1600/60 \rceil = 27$ to 31 leaf nodes.

   The answer depends on whether you define the "height" of a tree as the maximum number of messages between a leaf and the root, or the number of layers of nodes. We use the former interpretation to choose the answer given above because the number of messages is the usual metric of interest in designing parallel algorithms; however, we accepted the latter (which is the answer given above plus one) as long as the answer was consistent.

   (b) In the 100 / leaf case, how many times are each of these functions called (over all processes)?

   Solution:

   `init()`: **16** `accum()`: **1600** `combine()`: **15** `reduceGen()`: **1**.

   - `init()` is called 16 times, since the result must be initialized once per leaf process.

- `accum()` is called 100 times per leaf process, or in other words, once per data element for each leaf process. This gives a total of 1600 times.
- `combine()` is called once per non-leaf node so 15 times: If the height of the tree is H, the total number of nodes in tree is $2^{H+1} - 1$ and the number of leaf nodes $= 2^H$. In this case $H = 4$, so the number of non-leaf nodes is $2^{(4+1)} - 1 - 2^4 = 15$.
- `reduceGen()` is called once at the root to process the final result.

(c) In the 60 / leaf case, how many times are each of these functions called (over all processes)?

Solution: The answers for `init()` and `combine()` depend on the tree, since they are the number of leaf and non-leaf nodes respectively. If we use the balanced tree with 32 leaf nodes each processing 50 elements, then:

`init()`: **32** `accum()`: **1600** `combine()`: **31** `reduceGen()`: **1**.

What about an unbalanced tree with 27 leaf nodes (the minimum)? The majority of this tree's leaf nodes require 5 messages to reach the root, but some have paths of length 3 or 4. There are 25 non-leaf nodes. Then

`init()`: **27** `accum()`: **1600** `combine()`: **25** `reduceGen()`: **1**.

Note that reducing the number of calls to `init()` and `combine()` may not get you the answer any more quickly; for example, if the time to send a message drastically dominates execution time of `init()` and `combine()`, then the time to complete the scan will depend primarily on the maximum height of the tree (which is 5 in either the balanced or unbalanced cases). Therefore, whether the additional conceptual complexity of an unbalanced tree results in reduced execution time will depend heavily on the choice of parallel hardware and software architecture.

3. **Generalized scan (10 points)**. As described in *Lin & Snyder* a generalized scan is described by four functions: `init()`, `accum()`, `combine()` and `scanGen()`. Assume that we have 1600 data elements to process and that we set up as balanced a tree as possible.

(a) How many times are each of these functions called (over all processes) if we can give 100 data elements to each leaf process?

Solution:

`init()`: **17** `accum()`: **1600** `combine()`: **30** `scanGen()`: **1600**.

- `init()` is called once per leaf node to start the upward pass (so 16 times), plus once at the root to start the downward pass, for a total of 17.
- `accum()` is called once per element so a total of 1600 times.
- `combine()` is called 30 times. We have 15 non-leaf nodes, and combine is run once at each of them for the upward pass and then once more at each of them for the downward pass.
- `scanGen()` is called once at each element so 1600 times.

(b) How many times are each of these functions called (over all processes) if we can give just 60 data elements to each leaf process?

Solution: Again, the answers for `init()` and `combine()` depend on the shape of the tree. For the balanced tree of height 5:

`init()`: **33** `accum()`: **1600** `combine()`: **62** `scanGen()`: **1600**.

If we use an unbalanced tree with 27 leaf nodes and 25 non-leaf nodes, then the number of calls to `init()` and `combine()` will decrease slightly.

`init()`: **28** `accum()`: **1600** `combine()`: **50** `scanGen()`: **1600**.