CpSc 448B

Homework 3

Due: Nov. 13, 2011

Please submit your solution using the handin program. See homework 1 for more instructions on handing in homework.

1. **LogP** (**20 points**)
   Consider the two approaches to broadcast described in the Oct. 18 lecture: a simple, balanced binary tree, and the optimal, unbalanced binary tree (see Figure 1).

   (a) (**2 points**) Which is easier to implement? Give a one or two sentence justification for your answer.

   Solution: The balanced tree. The logP approach requires computing a schedule which basically involves simulating the broadcast. The balanced tree simply generates two branches at every non-leaf node.

   (b) (**3 points**) Which version is easier to port between different computers? Give a one or two sentence justification for your answer.

   Solution: The balanced tree. The balanced tree method is indpendent of the machine parameters, whereas the logP broadcast changes according to the relations of the parameter values.

   (c) (**15 points**) By definition, the broadcast using the optimal tree must be at least as fast as broadcast using the balanced tree (I will assume that logP estimates execution time correctly). Use the parameter values for l, o, and g from Table 1 of the logP paper and the associated discussion, and considering P in the range $[2, \dots 1024]$. What is the biggest speed-up of the optimal tree compared to the balanced tree for each of the nCUBE/2, the TMC CM5, the Intel Paragon, Dash, and the J machine?

   You answers should be a table with three columns: machine, $P_{\max}$, and $S_{\max}$, where $P_{\max}$ is the number of processors for that machine such that the optimal tree offers the greatest speed-up relative to the balanced tree, and $S_{\max}$ is that speed-up. Show how you got your results including any code or derivations that you used along with a short, English summary of your approach.

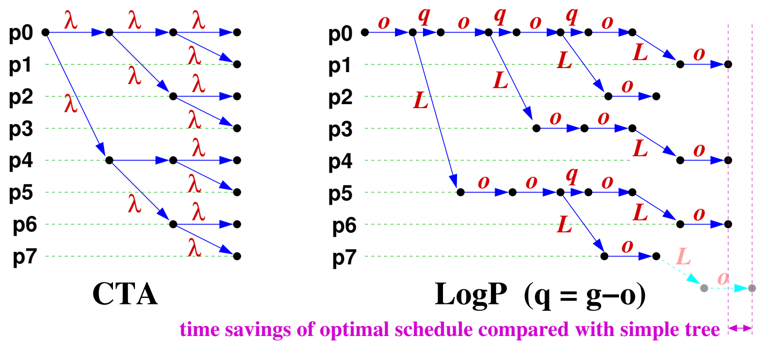

Figure 1: Two Broadcast Trees

Solution: First, we estimate parameter values for each of the five machines requested in the problem, and we immediately hit a snag. Figure 2 of the paper clearly labels L as the network latency, but the numbers in Table 5 have L as the total time (i.e. network latency plus $2o$). I wrote my code (and lecture slides) to be consistent with Figure 2 of the paper, but I'll have to compensate when using the numbers from the table. So, I'll use $L = \lceil \frac{M}{w} \rceil + Hr$. This number should be the same as the L from the table minus $T_{snd} + T_{rcv}$, but (of course) it's not. For example, for the CM-5, the first method gives $L = 112.4$ and the second gives a value of 114. Not a big difference, but this is just one more place that shows that no one bothered to proof read the paper.

I determined g using $g = P/(2B)$ where B is the cross-sectional bandwidth of the machine. The factor of 2 is because half of the processors are on each side of the cross-section.

The parameters values that I get are:

machine	L	o	g	P
nCUBE/2	360	3200	160	1024
CM-5	112	1800	???	1024
Paragon	150	2150	160	1024
Dash	23	15	80	1024
J	44	8	160	1024

Notes:

nCUBE/2: the cross-sectional bandwidth, B, of a hypercube with P processors is $P/2$ times the bandwidth of an individual link. I measure bandwidth in terms of the 160 byte messages postulated in the paper.

CM-5: Given that L in Table 1 differs from L in Figure 2 just by an offset of $2o = T_{snd} + T_{rcv}$, one should be able to get the L I want by subracting $T_{snd} + T_{rcv}$ from the value in the "L" column. For the CM-5, the L computed by taking the differnce of table entries is 114. If I use the formula of $L = \lceil \frac{M}{w} \rceil + Hr$, I get a value of 112.4. Not a big difference, but it would have been nice if someone had proofread the paper.

I can't determine g for the CM-5 because I can't find anything that says what the value of α was for the CM-5's fat-tree. However, o is so big that I'm going to guess that g is dominated by o.

Paragon: The cross-sectional bandwith of a $\sqrt{P} \times \sqrt{P}$ mesh is $\sqrt{P}$ times the bandwith of an individual link.

As with the CM-5, the numbers for L don't work out. Taking the difference of table columns gives a value of 150. On the other hand, $L = \lceil \frac{M}{w} \rceil + Hr = [178, 220]$; it's an interval because $r = [7, 10]$. I used 150 because it's easier to work with than an interval.

Dash: The cross-sectional bandwith of a $\sqrt{P} \times \sqrt{P}$ mesh is $2\sqrt{P}$ times the bandwith of an individual link. This is a processor for which $g > o$. That's because cache coherence actions are done by the hardware and don't have the overhead of system calls.

Note that LogP will give a highly pessimistic estimate of the performance for broadcast for Dash (and the J machine). The issue is that broadcast does not need to stress the bandwidth of these networks. The messages near the top of the broadcast tree should be sent across the critical cross-sections because there are very few such messages. The messages near the bottom of the broadcast tree should be between processors that are close to each other in the network. LogP has no way to describe network topology; so, it can't model this in a realistic way.

In fact, the unbalanced algorithm from the LogP paper does something that is quite amusingly silly in this case. If processor p_i sends a message to processor p_j, processor p_j will have received the message and be eligible to send another message after time $L + 2o = 53$ clock cycles. While processor p_i will have completed its send overhead after 15 cycles, it isn't eligible to send again (according to LogP) for another 65 cycles. This extra 65 cycle wait is not imposed by the hardware, it's just an artifact of the LogP model.

J-Machine: I'll assume that a 3d-mesh with 1024 processors is $8 \times 8 \times 16$ processors. The cross-section is given by the product of the two smaller dimensions, i.e. 64 times the link bandwidth. This is another machine for which $g > o$. This means that the LogP prediction for the J-Machine is unrealistic just as it was for Dash.

Now that I have model parameters, I wrote two functions to compute the time to complete a broadcast with a balanced tree and with an optimal (by LogP), unbalanced tree. I used Matlab, but any programming language would be fine. I also wrote a function to evaluate both broadcast trees on a range of values of P to determine S_max and P_max for each of the five machines. The code for these thre functions is listed below.

```
% t = balanced_broadcast(l, o, g, p)
% Compute the time to broadcast a message using a balanced tree. We just compute the time
% to all leaves. We do this recursively. Let t_left be the time to do a broadcast
% in the left subtree. Then, the longest time from our root to a leaf in the left subtree is
% t_left + g because the same processor is the root of this subtree and the left subtree,
% but it must wait g time units between sending messages. Let t_right be the
% time to do a broadcast in the right subtree. Then, the longest time from our root to a leaf
% in the right subtree is t_right + L + 2*o because the the root of the right subtree is
% a different processor, and it takes L + 2*o time units to send a message between processors.
function t = balanced_broadcast(l, o, g, p)
   if(p == 1)
      t = 0;
   else
      % I'll just compute the times to all leaves and take the maximum
      p0 = ceil(p/2); % number of elements in right tree
      p1 = p - p0;    % number of elements in left tree

      t_left = balanced_broadcast(l, o, g, p0);
      t_right = balanced_broadcast(l, o, g, p1);
      t = max(t_left + (p0 > 1)*g, t_right + l+2*o);
   end
end % balanced_broadcast
```

```
% t = unbalanced_broadcast(l, o, g, p)
% Compute the time to broadcast a message using an optimal unbalanced tree. We compute a schedule greedily.
% At t = 0 processor 0 is ready to send the message. It sends a message to processor 1, and we record the
% times at which processors 0 and 1 are eligible to send a new message. Then, we repeatedly find the
% next processor that is ready to send a message, and have it send to the next processor that hasn't received
% the message yet. We return the time at which the final processor receives the message.   function t = unbalance
   if(g < o) g = o; end;
   next = zeros(p,1);
   t = 0;
   for i = 2:p
      [now, who] = min(next(1:(i-1)));
      t = now + l + 2*o;
      next(i) = t;
      next(who) = now + g;
   end
end % unbalanced_broadcast


% [s_max, p_max] = bspeed(l, o, g, p)
% Evaluate the time to performan a broadcast using a balanced, binary tree and using an optimal, unbalanced tree.
% Do this for each element of the vector p, and find the choice for which the ratio of the time for the balanced
% broadcast to the time for the unbalanced broadcast is maximized. Return this ratio and the value of p for which
% it was achieved.   function [s_max, p_max] = bspeed(l, o, g, p)
   s_max = [];
   p_max = [];
   for i = 1:length(p)
      t0 = balanced_broadcast(l, o, g, p(i));
      t1 = unbalanced_broadcast(l, o, g, p(i));
      s = t0/t1;
      if(isempty(s_max) || (s > s_max))
         s_max = s;
         p_max = p(i);
      end
   end
end % bspeed
```

Running the *bspeed* function with the parameters derived above, I get:

machine	L	o	g	$P_{\max}$	$S_{\max}$
nCUBE/2	360	3200	160	512	1.32
CM-5	112	1800	???	512	1.30
Paragon	150	2150	160	512	1.30
Dash	23	15	80	9	1.38
J	44	8	160	3	1.83

For most of the machines, the unbalanced broadcast offers a speed-up of 30-40%.
The J-machine numbers are a bit strange. Looking at the $P = 3$ case, both schedules (balanced and unbalanced) have processor 0 send a message to processor 1 at time $t = 0$. Processor 1 receives this message at time $t = 60$. Here is where the schedules differ. The balanced tree waits until $t = 160$ and then has processor 0 send a message to processor 2. The unbalanced tree has processor 1 send a message to procesosr 2 at $t = 60$. As there is only one message in the network at a time for either of these scenarios, cross-sectional bandwidth is not an issue. This is an example of where the g parameter in LogP is leads to absurd answers.
Noting that a broadcast tree won't saturate the network (as long as the processors are placed in a sensible way), I tried again with $g = 0$. In this case, the J-machine has a $S_{\max} = 2.08$ with $P_{\max} = 1024$. In this case, the large ratio is because the network latency, L, is much greater than the overhead. Thus, in the unbalanced network, processor 0 can send 6 more message before processor 1 has received the first message. More refinements could be made by designing a broadcast tree that considers the mesh topology of the J-machine, and does better than the "average" message latency.
The advantages of 30-40% are enough to be worth considering, especially if a computation is limited by overheads (e.g. idle processors) during broadcasts. The factor of 2 with the J-machine is even more significant (but may be just an artifact of the model ignoring the network topology). It would be an interesting exercise to compare a simple tree of degree $\frac{L}{o} + 1$ with the optimal schedule, but that wasn't this problem.

2. **Computation vs. Communication (25 points)**
Consider implementing the FFT algorithm for N points using P processors. Assume that the P processors are arranged as a $M \times M$ mesh, where $M = \sqrt{P}$. To keep this simple, assume that N and P are both powers of two; P is a perfect square, and $P \leq \sqrt{N}$. With these assumptions, the computation proceeds as follows:

Step 1: Each processor computes $\frac{\sqrt{N}}{P}$ FFTs of length $\sqrt{N}$ operating on data stored in local memory (no communication).

Step 2: Each processor sends $P - 1$ messages, one message to each processors, each message consists of $N/(P^2)$ data values. Basically, each processor is "dealing" (like a deck of cards) its N/P words evenly amongst the P processors (including itself).

Step 3: Each processor computes $\frac{\sqrt{N}}{P}$ FFTs of length $\sqrt{N}$ operating on data stored in local memory (no communication).

Computing a FFT on N data values on a single processor takes $O(N \log N)$ time. For simplicity, I'll assume that it takes exactly $N \log_2 N$ processor clock cycles. Assume that each link of the 2D mesh has sufficient bandwidth to transfer one data value in each direction for every processor clock cycle.

(a) (**5 points**) What is the total time required for steps 1 and 3 as a function of N and P?

Solution: For each step, each processor computes $\frac{\sqrt{N}}{P}$ FFTs of length $\sqrt{N}$. Computing one such FFT takes $\sqrt{N} \log_2 \sqrt{N}$ clock cycles (from the problem statement). Thus, computing $\frac{\sqrt{N}}{P}$ such transforms takes time $\frac{N}{2P} \log_2 N$ time. This time is incurred both in step 1 and in step 3. Thus, the total time for these two steps is:

$$\frac{N}{P} \log_2 N \textit{ clock cycles.}$$

Intuitively, this makes sense. The sequential time is $N \log_2 N$ clock cycles, and the work is evenly divide over the P processors.

(b) (**10 points**) What is the total time required for step 2 as a function of N and P?

Solution: Step 2 is limited by the bandwidth of the mesh. Consider a vertical line dividing the mesh into two, equal-sized sub-meshes. Each of the $P/2$ processors to the left of this line must send $P/2$ messages of N/P^2 words to the processors to the right of this line. Thus a total of $N/4$ words must cross this line from left to right. There are $M = \sqrt{P}$ links crossing this line from left to right, and each link can convey one word per clock cycle. Thus, step 2 takes at least time $N/(4M)$ clock cycles.

For moderate values of P or larger, we can arrange the messages so that those that have to go the furthest after crossing the vertical line cross it first. Thus, when the last messages cross the cross-section, they reach their destination processors immediately, and step 2 is completed. This means that the time for step 2 is $\sqrt{N}/(4M)$ clock cycles.

(c) (**10 points**) For what values of N and P is the time for computing the FFT dominated by the time for computation? For what values is communication time the most critical?

Solution: We want to compare the compute time, $\frac{N}{P}\log_2 N$ with the communcation time, $N/(4\sqrt{P})$. These two are equal when $P = 16(\log_2 N)^2$. If P is smaller than this value, then computation time (steps 1 and 3) dominates. If P is larger than this value, then communication time dominates.

By the problem statement, $P < \sqrt{N}$. So, the smallest "interesting" value of N is when $16(\log_2 N)^2 = \sqrt{N}$. For example if $N = 2^{20} = 1,048,576$, then $16(\log_2 N)^2 = 6400 > \sqrt{N}$. However, if $N = 2^{30} = 1,073,741,824$, then then $16(\log_2 N)^2 = 14400 < 32768 = \sqrt{N}$. Thus, the communication time becomes dominant for N in the range of billions and P in the range of tens of thousands. Note that a network that can convey one, double precison floating point value per link per processor clock cycle is definitely in the super-computer range. So, a N in the billions and P in the tens of thousands is realistic for such problems, and we can conclude that supercomputers are engineered close to the balance point of communication and computation bottlenecks.

The answers change dramatically with changes in the computing assumptions. For example, a quad-core processor could perform a floating point add and multiply on every clock cycle on every core. This could reduce the computation time by about a factor of four compared with the estimates from the problem. Using 1Gbit ethernet as the communcation network requires $> 64ns$ to send a word. That's 160 clock cycles for a 2.5GHz processor. We would then end up with a computation time of $\frac{N}{4P}\log_2 N$ and a communcation time of $40N/\sqrt{P}$. The crossover would then be when $P = (\log_2 N)^2/(25600)$. Thus, desktop computers connected with commodity ethernet become communcation bound for any practical value of N.

Observations:

(a) To bound the communication time, consider a geometric line that divides the mesh into two halves with $P/2$ processors on each half. Determine the total number of data values that must cross this line in step 2. Determine the total number of network links that cross this line (in particular, think of a line that minimizes this value). From the number of links, you can determine the maximum rate at which data can cross the line. Having determined how much data must cross the line, you can now figure out how much time it must take.

(b) I used FFT because it's an example from the logP paper. Similar considerations apply to many other algorithms. In particular, sorting can be implemented with a communication pattern that is very similar to that of FFT.

(c) My timing assumptions of one FFT "step" per processor clock cycle and one data transfer per link per processor clock cycle are both quite aggressive. They might apply to a high-end supercomputer. For more realistic computers, both would be slower. The number of cycles to transfer a data value across a network link will typically be greater than the number of cycles required to compute a FFT step (for well-optimized code).

3. **Reduce and Scan** The `workers` module provides an implementation of a generalized reduce.

   (a) (**20 points**) Write an implementation of a generalized scan in erlang This should be a function:

   ```
   scan(W, Leaf1, Leaf2, Combine, Acc0) -> term().
      W = worker_pool(),
      Leaf1 = fun((S::worker_state()) -> term()),
      Combine = fun((L:term(), R:term()) -> term()),
      Leaf2 = fun((S::worker_state(), P:term()) -> worker_state()),
      Acc0 = term()
   ```

   `W`: a worker pool (e.g. created by `workers:create/1`. You're encouraged to use the `wtree` module from
   `http://www.ugrad.cs.ubc.ca/~cs448b/2011-1/src/erl/wtree.erl`
   It provides support for trees of workers, and has an implementation of `reduce` that you can use as a starting point for implementing `scan`.

   `Leaf1`: the `Leaf1` function is executed by each worker process (on it's current process state) to produce a value.

   `Combine`: The values obtained from `Leaf1` are combined using a tree of processes. The value from the `Combine` at the root is the return value from `scan`. Furthermore, the results are propagated back down the tree so that each worker executes `Leaf2(PS, V)` where

   `PS`: is the current process state.

   `V`: is the value from combining all of the `Leaf1` results for all nodes to the "left" of this one.

   `Leaf2`: the `Leaf2` function is executed to update state of the worker process. The parameter `P` is the cumulative value from everything to the "left" of this process.

   `Acc0`: Note that the leftmost worker process doesn't have any results coming from it's left. When `Leaf2` is called in the leftmost worker, it gets `Acc0` as its second argument.

   Solution: See Figures 2 through 4. Note that I've written lots of comments, but not nearly as much code. For a homework solution, you should have enough comments to make it easy to understand how your code is organized and how it works, but you don't need to go into nearly as much detail as I did.

   (b) (**5 points**) Use your generalized scan to make a parallel implementation of the cumulative-sum operation. Here's a sequential version of cumulative sum:

   ```
   cumulative_sum(List) ->
      element(1, lists:mapfoldl(fun(ListElem, Acc) ->
         S = ListElem+Acc,
         {S,S}
      end, 0, List)).
   ```

   Solution: This is very similar to the first test case that I provided for *scan*. Here's my code:

   ```
   cumulative_sum(Workers, Source, Dest) ->
   % Leaf1 computes the sum of the elements in this worker's Source list.
   Leaf1 = fun(PS) -> lists:sum(workers:get(PS, Source)) end,
   % Leaf1 computes a cummulative sum of the elements in this worker's Source list starting
   % with an offset of Acc0. The cummulative sum is assigned to Dest. The scan function calls
   % Leaf2 with Acc0 set to the total of all elements preceding this worker's list.
   Leaf2 = fun(PS, Acc0) ->
   workers:put(PS, Dest, misc:cumsum(Acc0, workers:get(PS, Source)))
   end,
   % Combine computes the total number of elements in two subtrees given the number of elements in each tree.
   Combine = fun(A, B) -> A+B end,
   wtree:scan(Workers, Leaf1, Leaf2, Combine, 0).
   ```

```
%% First, an erldoc description of scan.
%% @spec scan(W, Leaf1, Leaf2, Combine, Acc0) -> term1()
%%    W = worker_pool(),
%%    Leaf1 = fun((ProcState::worker_state) -> term1()),
%%    Leaf2 = fun((ProcState::worker_state, AccIn::term1()) -> worker_state()),
%%    Combine = fun((Left::term1(), Right::term1()) -> term1()),
%%    Acc0 = term1()
%% @doc A generalized scan operation.
%%    The Leaf1() function is applied in each worker process.
%%    The results of these are combined, using a tree, using Combine.
%%    The return value of the scan is the result of applying the Combine function at the root of the tree.
%%    Furthermore, the Leaf2() function is applied in each worker process.
%%    The AccIn argument is the results of the Combine node in the tree.
%%    For the leftmost process, Acc0 is used.
%%    The return value of Leaf2 becomes the state of the worker process.

% Now, I'll describe my implementation.
%    My scan uses five functions:

%      • scan(W, Leaf1, Leaf2, Acc0 -> term()
%          As described above.

%      • scan_dispatch(CPid, LLC)
%          Send the Leaf1, Leaf2, and Combine functions (as the tuple, LLC to the worker)
%          process CPid. The child uses Leaf1 and Combine to compute the cummulative value for its subtree,
%          and sends this value back to its parent. The parent responds with the cummulative value for everything
%          to the left, and the child does the final updates using Leaf2.

%      • scan_work1(ProcState, Children, LLC) -> LeftValues
%          The first phase of the scan. Compute the cummulative value for the subtree rooted at this process.
%          This process handles the current node and all of its left-descendants. This function produces a list of the
%          cummulative values from this node and its left descendants. These values are used in the second phase,
%          scan_work2.

%      • scan_work2(ProcState, Children, LLC, Left, V) -> NewProcState
%          The second phase of the scan. Propagate "left" values down the tree and update the leaf nodes.

%      • scan_receive(Pid, ProcState) -> ReceivedValue
%          Used to receive values and print a debugging message if the time-out fails. I should probably have written a
%          corresponding scan_send function so that the structure and tags for the message transfers were completely
%          encapsulated, but I'd rather get this solution posted than make (and test) tweaks to the code.

% See Figures 3 and 4 for the actual code.
```

Figure 2: Generalized Scan (part 1)

```
% See Figure 2 for overview comments.

% scan(Workers, Leaf1, Leaf2, Combine, Acc0)
%    Dispatch the scan to the first worker (i.e. the root of the tree). Send that worker Acc0 as the value of “everything
%    to the left”. Return the cummulative value for the entire tree.

scan([W0 | _], Leaf1, Leaf2, Combine, Acc0) ->
   scan_dispatch(W0, {Leaf1, Leaf2, Combine}),
   W0 ! {'$wtree$', scan, self(), Acc0},
   scan_receive(W0, []);
scan([], _L1, _L2, _C, _A0) -> ok.   % empty worker pool, nothing to do

% scan_dispatch(CPid, LLC)
%    Dispatch the scan to the first worker (i.e. the root of the tree).
%    We send the process a function that invokes scan_work1 to perform the first phase of the scan.
%    scan_work1 returns a list of values, the head of which is the cummulative value for our subtree. We send this value
%    to our parent. We then wait to recieve the cummulative value for everything to the left of our subtree, and then send it back
%    down our tree (along with the list of cummulative values for our left-descendants) for the second phase of the scan.
%    Note that this list of cummulative values from our subtrees provides the storage of intermediate values needed between
%    the two phases of the scan.

scan_dispatch(CPid, LLC) ->
   CPid ! fun(PS) ->
      V = scan_work1(PS, children(PS), LLC),
      parent(PS) ! {'$wtree$', scan, self(), hd(V) },
      Left = scan_receive(parent(PS), PS),
      scan_work2(PS, children(PS), LLC, Left, tl(V))
   end.

% scan_work1(ProcState, Workers, LLC): first phase of the scan
%    We invoke scan_dispatch on our right subtree and then make a recursive call to ourself to handle our
%    left subtree. We then receive a value from the process for the right subtree, and use Combine to
%    combine the value from the two subtrees. We *prepend* this combined value to the list of values that
%    was produced by the recursive call. This records the intermediate values that scan_work2 will
%    need for the second phase of the scan.
%    Note: each worker is represented by a tuple of the form {Nleft, RootRight} where Nleft is the number of
%    workers in the left subtree, and RootRight is the pid of the root process for the right subtree.
%    For this function, we only RootRight and we discard Nleft.

scan_work1(ProcState, [{_, RootRight} | CT], LLC) ->
   Combine = element(3, LLC),
   scan_dispatch(RootRight, LLC),
   Left = scan_work1(ProcState, CT, LLC),
   Right = scan_receive(RootRight, ProcState),
   [ Combine(hd(Left), Right) | Left ];
scan_work1(ProcState, [], LLC) -> % No more children, just handle our oown node.
   Leaf1 = element(1, LLC),
   [Leaf1(ProcState)].
```

Figure 3: Generalized Scan (part 2)

```
% See Figure 2 for overview comments.

% scan_work2(ProcState, Workers, LLC, Left, Values): second phase of the scan
%     Update our left and right subtrees using the appropriate cummulative vvalues.
%     In more detail, Left, is the cummulative value of everything to the left of our subtree. We pass this
%     in a recursive call to ourself to perform the update of our left subtree. The first element of Values, VH,
%     is the cummulative value of our left subtree. We use the Combine function to combine VH with Left
%     to produce the cummulative value for everything to the left of our right subtree. We send this combined value
%     to the process at the root of our right subtree.

scan_work2(ProcState, [{_, RootRight} | CT], LLC, Left, [VH | VT]) ->
   Combine = element(3, LLC),
   RootRight !  {'$wtree$', scan, self(), Combine(Left, VH)},
   scan_work2(ProcState, CT, LLC, Left, VT);
scan_work2(ProcState, [], LLC, Left, []) -> % no children, update our own state
   Leaf2 = element(2, LLC),
   Leaf2(ProcState, Left).   % Return the value from Leaf2. This becomes our new process state.

% scan_receive(Pid, ProcState): receive a message from Pid.
%     Wait for a message from Pid with the tag for a scan; then return the value from that message.
%     If no message is received after debug_timeout, print an error message.

scan_receive(Pid, ProcState) ->
   receive
      {'$wtree$', scan, Pid, V} -> V
      after debug_timeout(ProcState) ->
         misc:msg_dump("wring:scan",
            [io_lib:format("'$wtree$', scan, ~w, V", [Pid])])
   end.
```

Figure 4: Generalized Scan (part 3)

```
%% See Figure 6 for the definitions of a_leaf1, a_leaf2, and a_combine functions.

% n_ascend(W, Source, Dest)
%     W is a tree of workers.
%     Determine the number of ascending elements preceding each element of the distributed list with key Source.
%     Store the result using key Dest.
n_ascend(W, Source, Dest) ->
   Leaf1 = fun(PS) -> a_leaf1(workers:get(PS, Source)) end,
   Combine = fun(V1, V2) -> a_combine(V1, V2) end,
   Leaf2 = fun(PS, V) ->
      workers:put(PS, Dest, a_leaf2(V, workers:get(PS, Source)))
   end,
   Acc0 = {true, 0, [], []},
   wtree:scan(W, Leaf1, Leaf2, Combine, Acc0).
```

Figure 5: The `n_ascend` function

(c) (**5 points**) Use your generalized scan to make a parallel implementation of a function that for each element of a list, it computes the number elements preceding that element that are a non-descending sequence that includes the current element.

```
n_ascend(List) ->
   element(1, lists:mapfoldl(fun(ListElem, Acc) ->
      case Acc of
         {Prev, N} when Prev =< ListElem -> {N+1, {ListElem, N+1}};
         _ -> {0, {ListElem, 0}}
      end
   end, start, List)).
```

For example,

```
1> n_ascend([0,1,4,9,2,3,5,-1,-2,6,14,2]).
[0,1,2,3,0,1,2,0,0,1,2,0]
```

Solution: See Figures 5 and 6.

```
%% The n_ascend function is a bit more involved that the cummulative_sum function. I wrote stand-alone functions
%% (instead of in-line lambda expressions) for each of the components of the scan. This makes the code easier to comment
%% and easier to test.

% a_leaf1(List, AccIn = {AllAscend, N_ascend, First, Last}) -> AccOut
%     This is our main "accumulator" function. The accumulator tuple has four components:
%         • AllAscend: true if all elements of subtree are in ascending order.
%         • N_Ascend: the length of the ascending sequence at the right end of this subtree (may be 0)
%         • First: the leftmost of this subtree – [] if this subtree is empty.
%         • Last: the rightmost element of this subtree – [] if this subtree is empty.
%     a_leaf1 traverses this workers list, setting AllAscend to false if it ever sees a descending element,
%     and updating N_ascend and Last as it goes.
a_leaf1([], V) -> V;
a_leaf1([H | T], {AllAscend, N_ascend, First, Last}) when H >= Last ->
   a_leaf1(T, {AllAscend, N_ascend+1, First, H});
a_leaf1([H | T], _) -> a_leaf1(T, {false, 0, H, H}).
% a_leaf1(List) set the initial accumulator and call a_leaf/2 (if List is not empty).
a_leaf1([]) -> {true, 0, [], []};
a_leaf1([H | T]) -> a_leaf1(T, {true, 1, H, H}).

% a_combine(V1, V2) combine "accumulators" from two subtrees.
a_combine(V1, {true, 0, [], []}) -> V1; % if V2 is empty, ignore it
a_combine({true, 0, [], []}, V2) -> V2; % if V1 is empty, ignore it
% if AllAscend is false for V2, we can pretty much ignore V1
a_combine({_, _, First1, _}, {false, N_ascend2, _, Last2}) ->
   { false, N_ascend2, First1, Last2 };
% if Last1 > First2, we can pretty much ignore V1
a_combine({_, _, First1, Last1}, {true, N_ascend2, First2, Last2})
   when (Last1 > First2) -> {false, N_ascend2, First1, Last2};
% In the final case, the right subtree is in ascending order, and the last element of the left subtree
%     is less than the first element of the right subtree. In this case, the ascending sequence at the right end spans
%     the right subtree and includes all or part of the left subtree.
a_combine({AllAscend1, N_ascend1, First1, _}, {true, N_ascend2, _, Last2}) ->
   {AllAscend1, N_ascend1 + N_ascend2, First1, Last2}.

% a_leaf2(LeftAcc, List) -> AList
%     Determine the number of ascending elements preceding each element of List.
%     LeftAcc is the "Accumulator" (as described for a_leaf1) for the subtree on our left.
a_leaf2({_, N_ascend0, _, Last0}, List) ->
   element(1, lists:mapfoldl(fun(E, N0, L0) ->
      if
         L0 == [] -> {0, {0, E}}; % nothing before this element
         L0 =< E -> {N0+1, {N0+1, E}}; % E is an ascending element
         L0 > E -> {0, {0, E}}; % E is a descending element
         true -> % one of the previous cases should always match
            io:format("huh?  E = ~w, N0 = ~w, L0 = ~w~n", [E, N0, L0]),
            {0, E}
      end
   end, {N_ascend0, Last0}, List)).
```

Figure 6: Helper functions for n_ascend