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.

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

   (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 $[2, \ldots 1024]$. 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.

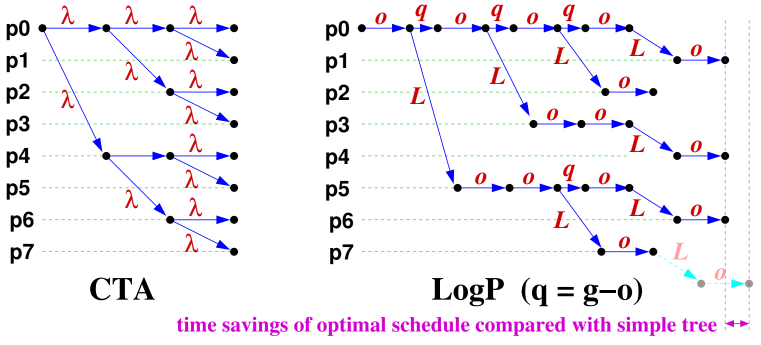

Figure 1: Two Broadcast Trees

2. **Computation vs. Communication (25 points)**
Consider implementing the FFT algorithm for N points using P processors. Assume that the P processors are arranges 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?

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

   (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?

   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.

(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)).
```

(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 form 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]
```