**Birla Institute of Technology & Science, Pilani**
**Work Integrated Learning Programmes Division**
**Second Semester 2022-2023**

**Mid-Semester Test – Model Solution**
**(EC-2 Regular)**

| | |
|---|---|
| Course No. | : AIML CLZG516 |
| Course Title | : ML System Optimization |
| Nature of Exam | : Open Book |
| Weightage | : 25% |
| Duration | : 2 Hours |
| Date of Exam | : 23/07/2023 (AN) |

No. of Pages = 2
No. of Questions = 4

Note to Students:
1. Please follow all the *Instructions to Candidates* given on the cover page of the answer book.
2. All parts of a question should be answered consecutively. Each answer should start from a fresh page.
3. Assumptions made if any, should be stated clearly at the beginning of your answer.

1. Consider the following equation for gradient descent:

   $w = w - \eta * g(L, D, w)$

   where g is the gradient function, L the loss function, and D, the dataset and $\eta$ denotes the learning rate

   and the corresponding pseudo-code for the mini-batch variant:

```
for i = 1 to num_iter {
      shuffle ( data );
      for batch in get_batches (data, batch_size) {
            grad = eval_gradient ( loss_function , batch , w );
             w = w - learning_rate * grad;
      }
```

   Argue whether the inner loop can be parallelized on a shared memory system:

   - if so, provide the details of a parallel solution (including how shared memory is used) and its scalability.

   - if not, explain the difficulties/limitations in parallelizing and/or scaling the solution.

   **[Expected Time: 30 minutes 6 Marks]**

   **Solution:**

   - The inner loop can be parallelized where each iteration of the loop is executed independently in a data-parallel manner.:

      eval_gradient() is run on all batches in parallel and the weights are eventually updated with the average gradient:

   Pseudo-code:

==============================================================

```
// get_batches splits data into batch_size number of subsets and returns a list of subsets
1. batches = get_batches (data, batch_size)
// 2 is a data-parallel step (i.e., can be implemented as a map)

2. for-each processor $P_j$,     j in 0 .. batch_size-1
// $grad_j$ is local to processor j
  2.1    $grad_j$ = eval_gradient ( loss_function , batch , w );

// outside for-each
// Step 3 (the summation) is a reduction (can be done in parallel)
     3. av_grad = ($\Sigma_j$ $grad_j$) / batch_size

     4. w = w - learning_rate * av_grad;
```

==============================================================

- This is acceptable since the convergence of GD is not impacted by short variations and the order in which the data points are processed (and the gradient updated)

- - Since this is done in a distributed system there is no communication required.

- Step 2 will scale linearly with the number of processors, **p**, as it is data parallel.

- Step 3 will provide a speedup of **p/log(p)** [because of reduction]

Step 4 is an O(1) computation (only arithmetic and assignment).

2. Consider the following equation for gradient descent:

   $w = w - \eta * g(L, D, w)$

   where g is the gradient function, L the loss function, and D, the dataset and $\eta$ denotes the learning rate

   and the corresponding pseudo-code for the mini-batch variant:

```
for i = 1 to num_iter {
    shuffle ( data );
    for batch in get_batches (data, batch_size) {
         grad = eval_gradient ( loss_function , batch , w );
         w = w - learning_rate * grad;
         }
```

Argue that the inner loop can be parallelized on a distributed (memory) system and if so, provide the details of a distributed solution (including how data is distributed) and its scalability. In particular, analyse the impact of communication cost on scalability.

**[Expected Time: 40 minutes                                        9 Marks]**

Solution:

- The inner loop can be parallelized in a distributed memory system where each iteration of the loop and each batch is processed  independently in a data-parallel manner.:

   eval_gradient() is run on all batches in parallel and the weights are eventually updated with the average gradient:

Pseudo-code:

```
=================================================================

// dist_batches splits data into batch_size number of subsets and distributes the subsets into batch_size node
1. batches = dist_batches (data, batch_size)

// 2 is a data-parallel step (i.e., can be implemented as a map)
2. for-each node $P_j$ ,     j in 0 .. batch_size-1
       // $grad_j$ is local to node j
  2.1    $grad_j$ = eval_gradient ( loss_function , batch , w );

// outside for-each
// Step 3 (the summation) is a reduction (can be done in parallel)
    3. av_grad = ($\Sigma_j$ $grad_j$) / batch_size

    4. w = w - learning_rate * av_grad;

=================================================================
```

- This is acceptable since the convergence of GD is not impacted by short variations and the order in which the data points are processed (and the gradient updated)

- - Since this is done in distributed memory, initial data distribution requires communication (and usually, we ignore this cost as one-off)

- Step 2 will scale linearly with the number of nodes, **p**, as it is data parallel.

- Step 3 should ideally provide a speedup of **p/log(p)** with p nodes [because of reduction].

But in a distributed system, reduction involves communication and therefore the actual speedup is **(p/log(p))*(1/(1+r))** where **r** is the ratio of messaging cost to CPU-operation cost i.e. **$r = T_{msg} / T_{add}$**

Thus Step 3 may not scale linear, it is asymptotically near-linear but weighed down by a large constant factor (messaging cost.)

Step 4 is an O(1) computation (only arithmetic and assignment).

=================================================================

3. Consider a collection C of images where you are required to find the image with the lowest number of red pixels and the image with the highest number of blue pixels, assuming that each image is 2400 x 1800 pixels of red, blue, or green. Express your computation as a *map-reduce* solution and calculate the speedup on a distributed systems with N nodes - for each of three different values of N such that N << |C|, N = (|C|), or N >> |C|.

**[Expected Time: 35 minutes 7 Marks]**

**Solution:**

Map-reduce formulation

- Define countBlue(img) to return the number of blue pixels in img.
- Define countRed(img) to return the number of red pixels in img.
- Then the entire computation can be expressed as two steps

1. imgBmax = (reduce max (map countBlue C))
2. imgRmin = (reduce min (map countRed C))

Sequential Time = $(|C|*T_{count}) + |C|-1 = |C|*(T_{count} + 1)$

where

$T_{count}$ is the time take to count (blue or red pixels in a 2400 x 1800 pixel grid.) – sequentially

Speedup with N nodes (Case N==|C|):

This is simple. Each image is processed in one node (for the map stage).

Each of the steps 1 and 2 will take time

$T_{count}$ (*for map stage*) + (N/logN)*(1/1+r) (*for reduce stage*)

= $T_{count}$ + (N/r*logN)

where

- $r = T_{msg} / T_{min}$,
- $T_{msg}$ is the messaging cost, and

- $T_{min}$ is the cost of a CPU operation like min

So, speedup is $T_{seq} / T_{par} = (|C|*(T_{count} + 1)) / (T_{count} + (N/r*\log N))$

$= N*T_{count} / (T_{count} + (N/r*\log N))$

$= N / (1+ N/r*\log N*T_{count})$

<u>Speedup with N nodes (Case N<<|C|)</u>

- Each node processes |C|/N images
- So time taken by each of the two steps will be:
  $T_{count} * |C|/N$ (*for map stage*) **+**
  $(|C|/N) – 1$ (for node-internal min or max done sequentially) **+**
  $(N/r*\log N)$ (*for reduce stage*)
  $=$
  $(|C|/N)*(T_{count} +1) + (N/r*\log N)$
- Speedup(N) = $T_{seq} / T_{par}$ =
  $= (|C|*(T_{count} +1)) / ((|C|/N)*(T_{count} +1) + (N/r*\log N))$

  $= (|C|* T_{count} ) / (|C|/N)*T_{count} + (N/r*\log N))$
  $= |C| / (|C|/N + N/r*\log N*T_{count})$

<u>Speedup with N nodes (Case N>>|C|)</u>

- |C|/N nodes process one image ; Let k = |C|/N
- Each node processes a part of the image i.e. each node processes (2400*1800)/k pixels.
- So time taken by each of the two steps will be:

$(T_{count} / k)$ (*for map stage*) **+**
$(k/r*\log k)$ (for aggregating count by reduce) +
<u>$|C|/r*\log|C|$ (for reduce min) +</u>
$(N/r*\log N)$ (*for reduce stage*)
$= (T_{count} / k) + 1/r (k/\log k + |C|/\log|C| + N/\log N)$
$= T_{count} /k +1/r (|C|/\log|C|)$
Speedup(N) = $T_{seq} / T_{par} = (|C|*(T_{count} +1)) / (T_{count} /k +1/r (|C|/\log|C|))$
$= |C|*T_{count} / (T_{count} /k +1/r (|C|/\log|C|))$
$= |C| / (1/k + 1/r *(|C|/T_{count} * \log|C|)$

4. What are the constraints in obtaining a high speed-up with a software-pipelined version of AdaBoost? Justify your answer.

   **[Expected Time: 15 minutes 3 Marks]**

   **Solution:**

   In AdaBoost every bootstrap stage chooses points from the dataset based on whether the same point was chosen (or not) in the earlier stages.

   Thus if we implement a software pipeline, each pipeline stage will depend on the results of previous pipeline stage i.e. the (j+1st) pipeline stage cannot start before the jth stage is complete.

   This limits the speedup (i.e. reduces the throughput of the pipeline).

=============================END===============================