CS484: Parallel Programming - Spring 2017
HW3
Due Date : April 25th, 11:59PM
Submission Method : SVN

May 4, 2017

1. **Isoefficiency**

   Compute isoefficiency functions for the following parallel algorithms. Assume a logP communication model where the cost of a message is $\alpha + \beta * m$ where α is the latency of the message, β is the cost of transmitting a byte and m is the message size in bytes. Assume that a process can send/receive a single message at a time.

   (a) Consider a distributed matrix-vector multiplication of an NXN matrix M with an $NX1$ vector Y on a group of P processes. Assume the matrix and vector are already distributed row-wise, where each process contains $\frac{N}{P}$ rows of the matrix, and the vector is distributed in chunks of $\frac{N}{P}$ elements. The figure below explains the data decomposition.

   The algorithm for sequential matrix-vector multiplication is as follows:

```
for (i=0; i<N; i++)
  for(j=0; j<N; j++)
    Z[i] += M[i][j]*Y[j];
```

   Assume M is already distributed row-wise, where each process contains $\frac{N}{P}$ rows of M, and Y and Z are distributed in chunks of $\frac{N}{P}$ elements.

   - Assemble the entire Y vector on each process. (You can do this using an all-to-all algorithm on a hypercube [refer to the lecture notes]).
   - Compute the local chunks of Z by multiplying the rows of the matrix with the vector Y.

   Compute the isoefficiency of this algorithm.

   Solution:

   Assumptions : The matrix is already distributed row-wise across processes, with each process owning $\frac{N}{P}$ rows, each row consisting of N elements. Each process also owns $\frac{N}{P}$ elements of the vectors X and Y.

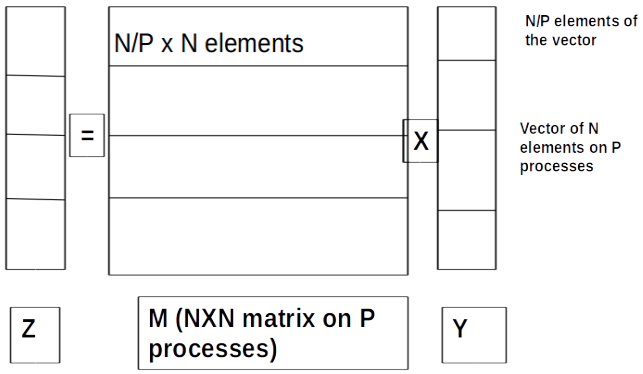

Figure 1:

For the computation, the non-local chunks of vector X should be collected on all processes. This can be done using a broadcast from each process to the other processes in the set or an all-gather. The broadcast can be implemented using separate point-to-point messages or a tree algorithm (discussed in lecture). If each process sends $P-1$ separate messages and this communication can be done in parallel, the total communication cost is the time taken for any process to perform a broadcast.
$W = N^2$ (work done for a sequential matrix-vector multiplication)
Communication cost, $T_c = (P-1) * (\alpha + \beta * \frac{N}{P})$
Computation cost, $T_p = \frac{N^2}{P}$

$$T_t = T_c + T_p$$
$$T_o = P * T_t - W$$
$$P * (P-1) * (\alpha + \beta * \frac{N}{P}) + P * \frac{N^2}{P} - N^2$$
$$W = K * T_o$$
$$N^2 = K * P^2 * (\alpha + \beta * \frac{N}{P})$$

Equating the LHS and RHS separately, using only the second term

$$N^2 = K * P^2 * \beta * \frac{N}{P}$$
$$N = K * P * \beta$$

Substituting for N, we get

$$W = K * P^2 * \beta * \frac{K * P * \beta}{P}$$

$$W = (K * P * \beta)^2$$
$$= O(P^2)$$

We have used the point-point implementation of broadcast, you can also do the same with the tree implementation which sends fewer messages.

(b) What is the isoefficiency of a parallel prefix algorithm for a set of N integers distributed across P processes? Assume $N\%P == 0$.

Solution : Each process has N/P elements. Parallel prefix can be computed by first performing a local prefix sum and then a prefix sum on the last value at each process. We can use a binary tree algorithm for parallel prefix. After this the prefix values are added to the local prefix values. Assuming integers with size = 4 bytes ; $W = N$ (linear work to compute prefix sum sequentially)

$$Computationcost T_c = \frac{N}{P}$$

local prefix-sum

$$Communicationcost T_p = logP * (\alpha + 4 * \beta)$$

$$T_t = T_c + T_p$$
$$T_o = P * T_t - W$$
$$T_o = P * \frac{N}{P} + P * logP * (\alpha + 4 * \beta) - N$$
$$T_o = P * logP * (\alpha + 4 * \beta)$$
$$W = K * T_o$$
$$N = K * P * logP * (\alpha + 4 * \beta)$$
$$W = O(P * logP)$$

2. **Topologies**

   (a) Suppose you have a network arranged in a 3D mesh topology with size 30x40x50 on the x, y, and z axes respectively.

      i. What is the diameter and degree of the above topology?
      Solution: diameter = (30-1)+(40-1)+(50-1) = 117; degree = 3 (at most one neighbor on each axis);

      ii. Assuming a link bandwidth of 4 GB/s, what is the bisection bandwidth of this network? Explain your answer.
      Solution: Because link bandwidth is uniform, we minimize bisection bandwidth by bisecting the network along the largest dimension (z=50), getting 30x40 = 1200 nodes. The bisection bandwidth is thus 1200*4GB/s = 4800 GB/s.

      iii. Suppose now the bandwidth along each axis is different: links parallel to the x axis have a bandwidth of 2 GB / s, while the remaining links have a bandwidth of 4 GB /s. What is the bisection bandwidth of this new network?
      Solution: By bisecting the network along the x-axis (x=30), we get 40x50 = 2000 nodes. Though there are more nodes to travel through, the new bandwidth is only 2000*2GB/s = 4000 GB/s, smaller than before.

   (b) Suppose you have to map a 200x300 2D grid of processes onto the above 30x40x50 network; i.e., for each process p, p[i,j] communicates only with p[i-1,j], p[i+1,j], p[i,j-1], and p[i,j+1]. Describe a scheme for assigning processes to nodes; your scheme should assign process[i,j] to node[x,y,z] such that the communicating processes are on nearby nodes as much as possible.

   Solution: Many creative solutions are possible; though there is no "perfect" solution in the sense that any solution will unavoidably have some processes that are neighbors logically but not physically, good mappings will achieve direct physical connections between as many logical neighbors as possible.

3. **Sample Sort**

   This problem will guide you step by step towards writing a parallel sorting program. For this problem, youll implement Histogram sort with sampling (with one round of sampling/histogramming) in MPI. Refer to lecture slides for the algorithm.

   A skeleton code has been provided for this problem. You should use this skeleton. However you are free to change any part of the code.

   - Every processor populates its local data (my_data) with random values. Then it sorts its local data using a sequential sorting algorithm. You can use STLs sorting function for this step.
   - Every processor randomly selects some number of sample keys from its local data. The number of samples chosen on each local processor is called the oversampling ratio.The default oversampling ratio is set to 20 in the skeleton code.Samples from all processors should be gathered at processor 0 (root). You should use MPI_Gather for this step.

- Processor 0 sequentially sorts the gathered keys (also called probes), using a sequential sorting algorithm. This code has been provided with the skeleton code. Note that the code adds a maxkey at the end of the probes. Processor 0 broadcasts the sorted probes to all processors. You should use MPI_Bcast for this step.
- Each processor counts how many keys it has between consecutive probes. Loop over the probes, and find how many keys are in each range. To achieve this, first compute the rank of each probe in the local input (using binary search) to obtain cumulative counts. Do a one pass over cumulative counts to change it to local counts. Reduce the array of resultant counts at the root using an addition operation. This will give how many keys in the entire input are between two consecutive probes. The reduced counts is also called a histogram. You should use MPI_Reduce for this step.
- Processor 0 selects $p-1$ splitter keys from the probes and the histogram using the findSplitters function provided in utils.cpp.
- Processor 0 broadcasts p splitters (including the redundant last key). You should use MPI_Bcast for this step. Using the splitters, each processor figures out where to send nonlocal data. In particular, it should send a key that lies in [splitter[i-1], splitter[i]) to process i.
- Each processor conveys to every other processor how much data it is going to send to that processor. You can should use MPI_Alltoall or point-point messages for this step. Get data to the right processors. You can use MPI_Alltoallv or point-point messages for this step.
- Locally sort all received data using a sequential sorting algorithm. The skeleton code computes initial and final checksum of the data after sorting. If your code runs correctly, the initial and final checksums are likely to match.

Experiments :

- For large problem sizes, vary the oversampling ratio and observe the effect on maximum load on a processor (load on a processor is stored in bucketCounts array). Vary the oversampling ratio and observe the effect on the maximum load on any process.
- For a given problem size, vary the number of processes, plot a graph of the execution time of sample sorting with increasing number of processes.

What to submit: Turn in the entire code. Make sure that your code compiles using the makefile provided. Write a document (preferrably pdf) with the graphs and any additional observations you may have made regarding sample sort.