Distributed Sum on a 2D mesh using MPI

Alessandro Febretti

Introduction

For this homework we had to compute the sum of a set of random integers using a map reduce approach on a mesh topology. The algorithm would read integers from an input file, run on 1, 2, 4, 8 or 16 Nodes, and compute time and speedup wrt. Sequential computation on the head node.

Assumptions

The mesh structure would be the following depending on node number:

- P = 2: 2x1 mesh
- P = 4: 2x2 mesh
- P = 8: 4x2 mesh
- P = 16: 4x4 mesh

For the P = 16 mesh, some processing resources would be allocated to multiple cores on the same node, since we were allowed 8 nodes max on the argo cluster.

Instead of having separate files with random numbers for $N=2^{10}$, 2^{13}, 2^{16} etc. I created a single file with 2^{20} random numbers. The distributed program would use only a subset of this file when running with less numbers as input. I also decided to test the algorithm with 2^{16}, 2^{18}, 2^{20} input numbers, since a sequential sum would run very fast without using a large enough input. Result data types were adapted to avoid overflow errors.

Algorithm Design

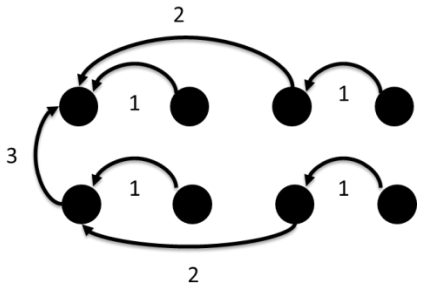

Figure 1. Reduction steps on a 2x3 mesh

Since the final result was needed on the head node (0, 0) only, the algorithm would perform a standard recursive halving on rows first, and then perform a recursive halving on the first column, to bring the final result on (0, 0) (see Figure 1).

So, if N is the input data size, P the number of processors and W, H the mesh width and height respectively (so that P = WxH). The initial computation time is $\Theta(N/P)$ and the number of communication rounds is $\Theta(\log(W) + \log(H)) = \Theta(\log(WH)) = \Theta(\log(P))$. This is assuming that all message transmissions have the same time cost (no collisions, no multi-hop). The speedup should be $\Theta(P)$ for $N >> P$ but we also have to take into account the different cost of the reduction operation (i.e. integer sum) vs. the cost of transmitting messages. If transmitting messages takes

much longer than a reduction operation, distributing the operation may bring no advantage, or actually be more time consuming than the corresponding sequential computation.

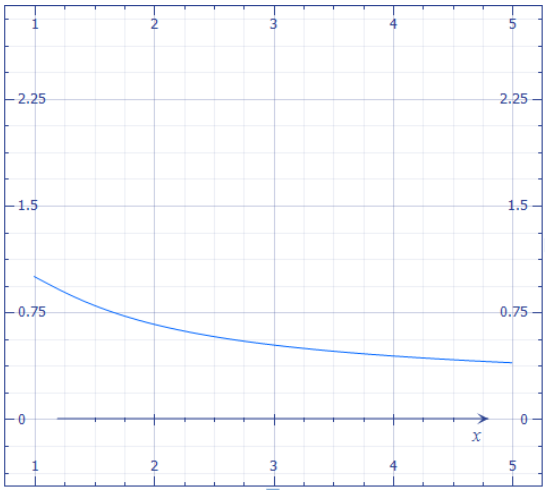

Figure 2 . Example of negative speedup (computed using formula described above) x = Nodes, y=speedup. If the cost of sequential reduction is the same as cost of transmitting one message, speedup is negative.

What follows is the pseudocode for the algorithm:

```
N = total data
P = total nodes
p = this node index [0, P]
(MW, MH) = the mesh width and height
(pi, pj) = this node index in the mesh ([0, MW], [0, MH])

data = data array (N elements)

// Compute local sum
Sum = sum N/P elements of data, starting from index N/P*p

// Row reduction
rounds = log2(MW)
For k in [1: rounds]
        // Interval between nodes involved in this round
        I = 2^(k - 1)
        // if pi is divisible by interval, we take part in this round
        If(mod(pi, I) = 0)
                        // Check if pi is divisible by next round interval
                        // If it is, this node is involved in next round of reduction
                        // This means this node should be a RECEIVING node this round
                        // Since sending nodes will not take part in next computation round
                        // due to recursive halving.
                       If(mod(pi, I*2) = 0) Sum += receive sum from node(pi + I, pj)
                       Else send my sum to node(pi - I, pj)

// Column reduction on first column only
If(pi = 0)
    // This computation follows same principle as row reduction, but acts on column indices
    Rounds = log2(MH)
    For k in [1:rounds]
        I = 2^(k - 1)
        If(mod(pj, I) = 0)
             If(mod(pj, I*2) = 0) Sum += receive sum from node(pi, pj + I)
             Else send my sum to node(pi, pj - I)

// If we are node(0, 0) print the final result.
If(pi = 0 and pj = 0)
   Print Sum
```

Implementation

The algorithm was implemented using MPI, with Recv and Send primitives only. As mentioned in assumptions, the executable would accept a text file as input containing 2^{20} random numbers, and an additional parameter indicating how many numbers would be read back and involved in the computation. The algorithm was run for P=1, 2, 4, 8,16 nodes and for N=2^{16}, 2^{18}, 2^{20}.

The launcher script is called `run_all`. This convenience script queues multiple versions of the computation on the cluster. For instance, running

```
./run_all 18
```

Will queue computations on 1,2,4,8,16 nodes for N=2^{18}. Output for each computation is written in a separate file. This simplifies regenerating the full results dataset.

An important observation is that the algorithm was queued to run on **local cores** whenever possible: each machine in the argo cluster has 4 cores, so 4 instances of the algorithm can be run on the same machine. Running on the same machine reduces message passing times. This is the node / core allocation used for this project:

- P = 1: 1 node, 1 core
- P = 2: 1 node, 2 cores
- P = 4: 1 node, 4 cores
- P = 8: 2 nodes, 4 cores
- P = 16: 4 nodes, 4 cores

Given this allocation, I expected the time and speedup to show two different trends when going from single node execution (P=1, 2, 4) to multi-node execution (P=8,16)

The algorithm was also implemented in two modes:

- With chunk distribution: the head node would send the needed data to every processor
- With direct file access: every processor would read its section of data directly from the source file

Results

The plots in the following figures show speedup trends (P = 1 was considered speedup 1) for various data sizes, for both chunk distribution and direct file access modes. The full data is included after the plots. And is also allegated as an excel spreadsheet and pivot chart. Figure 3 shows how for N=16 the data is not big enough to compensate for message passing costs. N=18 and N=20 get speedups, although they are more marked when the other instances of the process are running on a single node. For direct data access, speedup trends seem to be less consistent (Figure 4). Figures 5 and 6 compare speedup and time averages for the two distribution modes. Chunk distribution speedups are close to consistent to direct access (except for N=4), but chunk distribution runs generally faster: reading the file once and streaming the data works better than reading in for every single node. This can probably be explained by access concurrency issues.

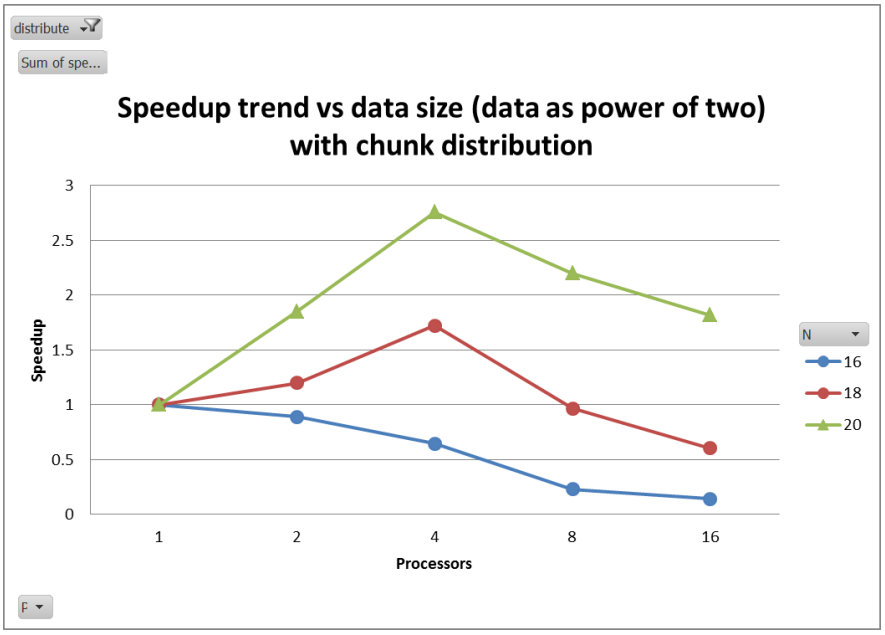

Figure 3. Speedup trend vs data size with chunk distribution

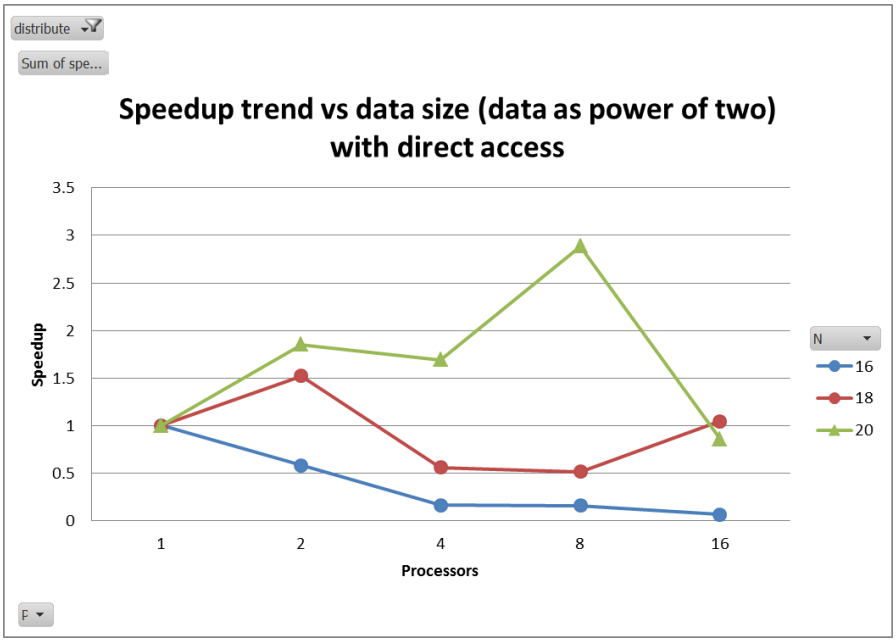

Figure 4. Speedup trend vs data size with direct access

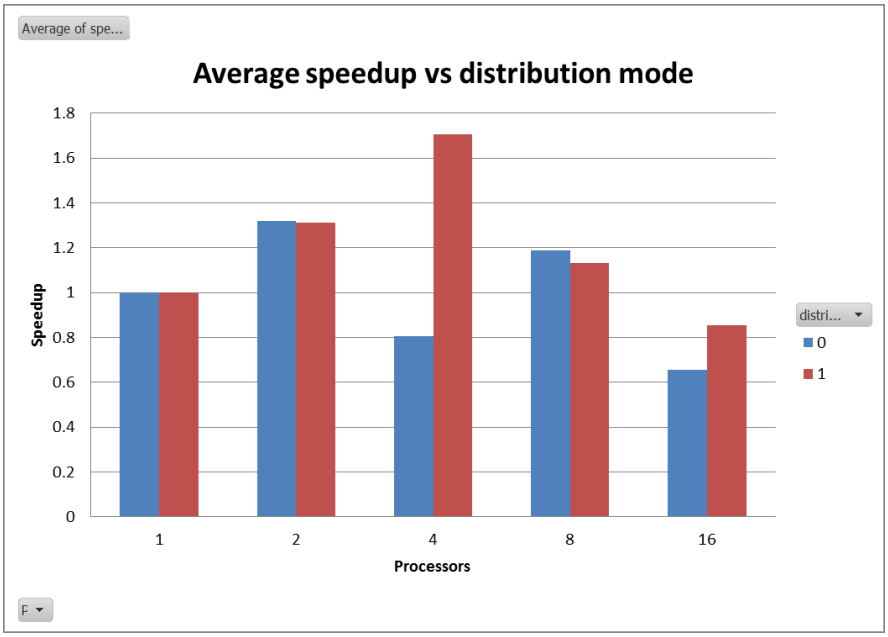

Figure 5. Average speedup vs distribution mode. 1 = chunk distribution. 0 = direct access

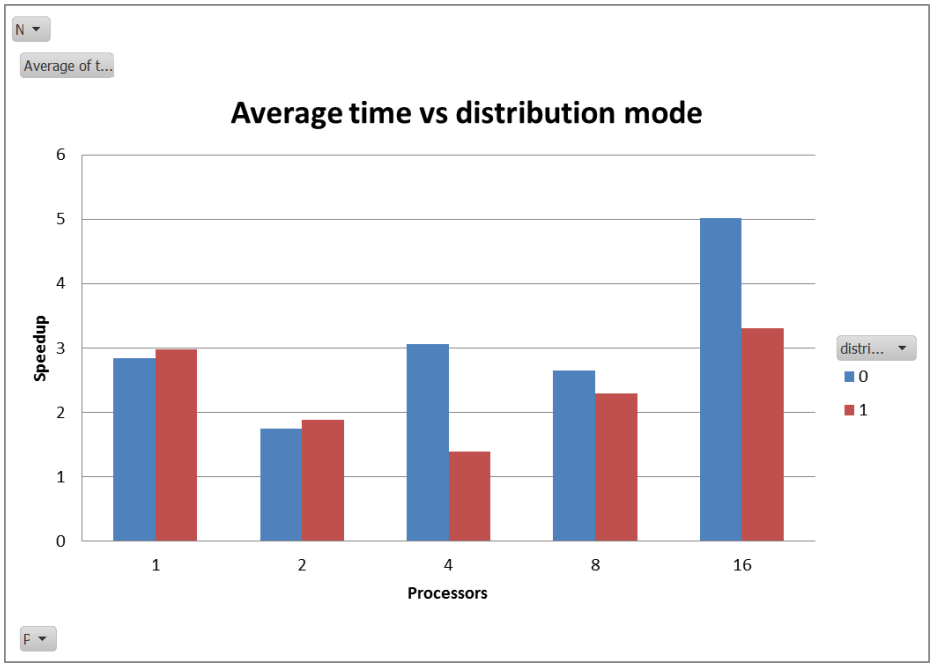

Figure 6. Average time vs distribution mode. 1 = chunk distribution. 0 = direct access

P	N	time	speedup	distribute
1	16	0.413	1	1
2	16	0.465	0.888	1
4	16	0.642	0.643	1
8	16	1.813	0.228	1
16	16	2.964	0.139	1
1	18	2.024	1	1
2	18	1.686	1.2	1
4	18	1.176	1.721	1
8	18	2.095	0.966	1
16	18	3.358	0.603	1
1	20	6.508	1	1
2	20	3.512	1.853	1
4	20	2.361	2.756	1
8	20	2.96	2.199	1
16	20	3.583	1.816	1
1	16	0.408	1	0
2	16	0.697	0.585	0
4	16	2.475	0.165	0
8	16	2.555	0.16	0
16	16	5.959	0.068	0
1	18	1.623	1	0
2	18	1.064	1.523	0
4	18	2.89	0.561	0
8	18	3.135	0.517	0
16	18	1.551	1.045	0
1	20	6.475	1	0
2	20	3.495	1.853	0
4	20	3.83	1.691	0
8	20	2.245	2.884	0
16	20	7.55	0.858	0