Assignment 2

Uppsala University
Parallel and Distributed Programming (1TD070)
Spring '18

Moustafa Aboushady

August 9, 2020

Problem Description

Given two dense square matrices $A = \{a_{i,j}\}$, $B = \{b_{i,j}\}$, $A, B \in \boldsymbol{R^{nxn}}$, $i, j = 1, 2, ..., n$. Let $C = AB$, $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$.

$$A \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & .. & .. & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a_{2,2} & .. & .. & a_{1,n-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & .. & .. & a_{2,n-1} \\ .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. \\ a_{n,0} & a_{n,1} & a_{n,2} & .. & .. & a_{n-1,n-1} \end{bmatrix} \times B \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & .. & .. & b_{0,n-1} \\ b_{1,0} & b_{1,1} & b_{2,2} & .. & .. & b_{1,n-1} \\ b_{2,0} & b_{2,1} & b_{2,2} & .. & .. & b_{2,n-1} \\ .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. \\ b_{n,0} & b_{n,1} & b_{n,2} & .. & .. & b_{n-1,n-1} \end{bmatrix}$$

$$= C \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & .. & .. & c_{0,n-1} \\ c_{1,0} & c_{1,1} & c_{2,2} & .. & .. & c_{1,n-1} \\ c_{2,0} & c_{2,1} & c_{2,2} & .. & .. & c_{2,n-1} \\ .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. \\ c_{n,0} & c_{n,1} & c_{n,2} & .. & .. & c_{n-1,n-1} \end{bmatrix}$$

Implementation

As the problem of Matrix-Matrix multiplication, consists of the dot product of an A's row and a B's column to get one element of C, as follows : $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$. It can be thought of as a repeated Matrix-Vector multiplication, and that is the approach i took in my implementation, by following these steps:

- Read A and B, then save them in a 1-d array representations.
- Scatter each row of A over all processes "Block partitioning".
- transpose B, but since it is saved as 1-d array, the transposition was done with the following equation:

Algorithm 1: *transpose B*

Result: b_t
initialize b : a 1-d array of length n*n,
  $counter = 0$
  for i = 0, 1, ... , n-1 **do**
    for j=0, 1, ... , n-1 **do**
      $b_t[j + counter] = b[j * n + i]$
    end for
    $counter = n$
  end for
  return

This will do the transposition as follows: Let $b = [0, 1, 2, 4, 5, 6, 7, 8]$, $i = 0, j = 0, counter = 0, n = 3$, we get $b_t[0 + 0] = b[0 * 3 + 0] = b[0] = 0$, the next iteration would be $i = 0, j = 1, counter = 0, n = 4$, and we will get, $b_t[1 + 0] = b[1 * 3 + 0] = b[3] = 4$, and so on. Next step is:

- broadcast first row of b_t over all processes "block partitioning", as in algorithm 2.
- Perform the matrix-vector multiplication.
- broadcast the second row of b_t, and repeat the process of sending the next row of b_t, then do the multiplication for n times.

Algorithm 2: *get_next_row B*

Result: *local_b*
initialize b_t : a 1-d array of length n*n,
  $b\_row\_counter = 0$
  for i = 0, 1, ... , n-1 **do**
    $local\_b[i] = b_t[i + (b\_row\_counter * local\_n)]$
  end for
  $b\_row\_counter = (b\_row\_counter + 1)\%n$
  return

Using the same 1-d array from the last example $[0, 1, 2, 3, 4, 5, 6, 7, 8]$, it's thought of begin divided into n parts, we send each part as explained in the previous steps, but to get the right part from b_t to send each time, we use the $b\_row\_counter$ variable. When $b\_row\_counter = 0$, we get the first part $[0, 1, 2]$, and when $b\_row\_counter = 1$, we get the second part $[3, 4, 5]$, and so on. The result for each iteration of the multiplication is saved in $local\_c$, after all the multiplication is done, we gather all $local\_c$ into C.

My reason for transposing B, was to also use block partitioning, since in MPI it's the easiest to use, tried to minimize the communication by using collective communication with only using Scatter, Broadcast, and Gather throughout the program, that gives us a tree-based hierarchical communication that takes log(P), with P is the number of processes used. However, the down side I see with this implementation, is the heavy computation work left to be done by process 0, but at the end, local computations are by far less expensive than unnecessary communications.

Numerical Experiments

Numerical experiments were run with the input data available at UPPMAX, we used 3600, 5716, and 7488 order of matrices, on 1, 2, 4, 8, and 16 processes. Table shows the run times in seconds of each experiment.

	Order of Matrix		
Comm Size	3600	5716	7488
1	146.02	639.67	1579.34
2	90.2	325	769.16
4	46.00	179.47	398.76
8	23.26	83.77	204.99
16	11.88	46.99	104.27

Table 1: Experiments' Run Times in seconds

Results

From table 2 and figure 1, we can see that the algorithm achieves near linear speedup, where we have the speedups getting near, $S(n,p) \approx p$, especially where $p = 5716$ *and* 7488 and for all values of n. Our worst case is where $n = 3600$, however the speedup values aren't that small compared to the other results. For efficiencies, we see in table 3 for large p, and small n, we get linear parallel efficiency, for large p and large n, we get near linear parallel efficiency, and for large p and small n is somewhat far from linear.

	Order of Matrix		
Comm Size	3600	5716	7488
1	1.0	1.0	1.0
2	1.61	1.96	2.0
4	3.17	3.56	3.96
8	6.27	7.64	7.7
16	12.29	13.61	15.14

Table 2: Experiments' Speedups in seconds

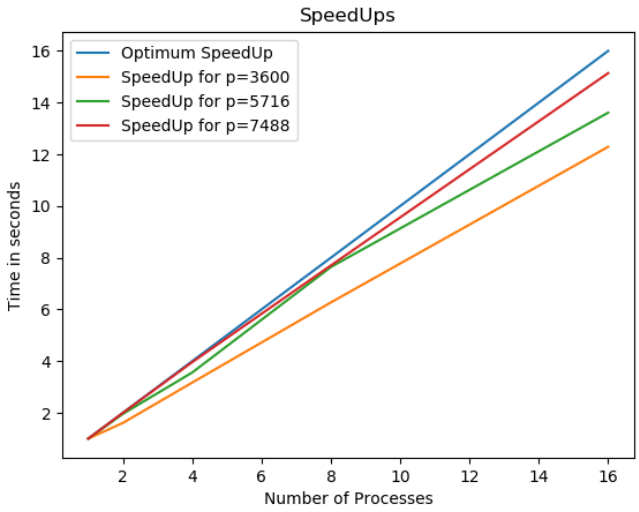

Figure 1: Comparison between the optimal/linear Speedup and the speedup we got for p = [3600, 5617, 7488]

	Order of Matrix		
Comm Size	3600	5716	7488
1	1.0	1.0	1.0
2	0.8	0.98	1.0
4	0.79	0.89	0.92
8	0.78	0.95	0.99
16	0.76	0.85	0.94

Table 3: Experiments' Efficiencies in seconds

Discussion

We see from how the algorithm achieved from the speedups, and efficiencies, that it might be considered a **weak scalability** algorithm, since the efficiency isn't constant regardless of the problem size, that is as n increases and $p \geq 2$, the efficiencies are decreasing, except for one case where p is doubled from 4 to 8, we see that we get an increase of efficiencies. Yet, it can also be considered a **strong scalability** algorithm, since for example when we double p from 8 to 16 for the same $n = 3600$, the efficiency only decreases from 0.78 to 0.76, and that is a very small decrease.