Report

April 2021

1 Algorithm Design

1.1 Vector Distribution

The vector distribution function was implemented by making the 1^{st} processor having all the data distribute the elements of vector one-by-one to all the processors. To handle the case, where number of elements was not divisible by number of processors, the root processor took the extra elements by taking ceil($\frac{n}{\sqrt{p}}$) elements and distributing to other processors floor($\frac{n}{\sqrt{p}}$) elements each.
Runtime :
Computation - $O(\sqrt{p})$
Communication - $O(\tau * \sqrt{p} + \mu * \sqrt{p} * \frac{n}{\sqrt{p}}) = O(\tau\sqrt{p} + \mu n)$

1.2 Matrix Distribution

The matrix distribution function was implemented by making the 1^{st} processor having all the data distribute the **columns** of matrix one-by-one to 1^{st} row of processors. To handle the case, where number of elements was not divisible by number of processors, the root processor took the extra elements by taking ceil($\frac{n}{\sqrt{p}}$) columns and distributing others floor($\frac{n}{\sqrt{p}}$) columns each where $n * n$ is dimension of the matrix.

Likewise, once the columns are distributed, the distribution of **rows** of matrix is carried out in parallel by 1^{st} row of processors. It is carried in the same way as column distribtion with 1^{st} row of processors having ceil($\frac{n}{\sqrt{p}}$) rows and other processors having floor($\frac{n}{\sqrt{p}}$) rows of the matrix.
Essentially, the final distribution of matrix(dimensions of sub-matrices) is as follows:
1^{st} processor = ceil($\frac{n}{\sqrt{p}}$)*ceil($\frac{n}{\sqrt{p}}$)
1^{st} column of processors = floor($\frac{n}{\sqrt{p}}$)*ceil($\frac{n}{\sqrt{p}}$)
1^{st} row of processors = ceil($\frac{n}{\sqrt{p}}$)*floor($\frac{n}{\sqrt{p}}$)
Other processors = floor($\frac{n}{\sqrt{p}}$)*floor($\frac{n}{\sqrt{p}}$)

Runtime :
Computation - $O(\sqrt{p} + \sqrt{p})$ =$O(\sqrt{p})$
Communication - $O(\tau * \sqrt{p} + \mu * \sqrt{p} * \frac{n}{\sqrt{p}} * n)$ + $O(\tau * \sqrt{p} + \mu * \sqrt{p} * \frac{n/\sqrt{p}}{\sqrt{p}} * n)$
$= O(\tau\sqrt{p} + \mu n^2)$

1.3 Gather

The gather function was implement in converse way of vector distribution function with each processor sending their elements to 1^{st} processor.The 1^{st} processor receives the elements and stores it in an array.
Runtime :
Computation - $O(\sqrt{p})$
Communication - $O(\tau * \sqrt{p} + \mu * \sqrt{p} * \frac{n}{\sqrt{p}}) = O(\tau\sqrt{p} + \mu n)$

1.4 Transpose Broadcast Vector

The function takes data with a vector x distributed among the first column of processors and "transposes" it to the remaining processors such that the first column of processors has the first part of x, the second column has the second fragment of x and so on.

This distribution of x was achieved by having the processors in the first column send their portion of x to the corresponding diagonal processor in the same row. Specifically for a processors (i,0) in the first column it sends its data to the processor (i,i) which is along the diagonal of the processor matrix. The size of the data being sent was either ceil($\frac{n}{\sqrt{p}}$) or floor($\frac{n}{\sqrt{p}}$), this size could easily be determined by both the sender and receiver based on the coordinates of the send are receivers in the cartesian topology, thus the message size itself did not need to be sent before the data could be sent.

After the diagonal processors have received the message from the first column processors, we then create a column sub communicator for each column we set the rank or processor (i,j) to have rank $(i - j)mod(n)$, this sets each processor on the diagonal to have rank 0 and gives for remaining processors we give them a rank equal to the number of rows that they are below the diagonal, with wraparound. This makes it very easy to then for each column sub-communicator broadcast the fragment of x from the root processor (which is now the diagonal processor) to all the other processors in the same column. After the x fragments are received the we undo the column sub-communicator and return to the usual Cartesian topology. Also it should be noted that each of these column wise broadcast involves only $\sqrt{p}$ processors.

Runtime:
Computation - $O(log(\sqrt{p})) = O(log(p))$
Communication - $O(\tau log(\sqrt{p}) + \mu \frac{n}{\sqrt{p}} log(\sqrt{p})) = O(\tau log(p) + \mu \frac{n}{\sqrt{p}} log(p))$

1.5 Distributed Matrix Multiplication

We will assume that the data has already been distributed according to the matrix and vector distribution functions outlined earlier. So we have the matrix A block distributed across all the p processors and the vector x block distributed among the first column of processors. Thus the first step is to use the Transpose Broadcast Vector function to distribute this vector x appropriately to all the processors (i.e. so that each processor has $O(\frac{n}{\sqrt{p}})$ elements of x).

Now that all the data is distributed each processor has the local A matrix and a local vector x which we can compute the local matrix multiplication on, giving a local y. We can then perform a sum reduction to the first column of processors to form the actual y vector (i.e. column wise sum the local y's and leave result in first column of processors), this reduction requires sending messages of size $O(\frac{n}{\sqrt{p}})$. (We ignore the part were we gather y to processor 0, since this isn't necessary if we want to perform successive matrix multiplications starting with distributed data).

The local computation time of this algorithm is dominated by performing the local matrix vector multiplication between a matrix of size $O(\frac{n^2}{p})$ and a vector of size $O(\frac{n}{\sqrt{p}})$. Each row of this matrix vector multiplication requires $O(\frac{n}{\sqrt{p}})$ work and we have $O(\frac{n}{\sqrt{p}})$ rows, giving a total local computation complexity of $O(\frac{n^2}{p})$. We also have a small amount of computation that occurs from performing Transpose Broadcast Vector and Reduce both of which give a computation time of $O(log(p))$ and communication time of $O(\tau log(p) + \mu \frac{n}{\sqrt{p}} log(p))$.

Runtime:
Computation - $O(\frac{n^2}{p} + log(p))$
Communication - $O(\tau log(p) + \mu \frac{n}{\sqrt{p}} log(p))$

1.6 Distributed Jacobi Iteration

We will again assume with start with data distributed in the same way as the data was distributed for matrix multiplication (to get the computation and runtimes without distributed data simply add on the complexities from Matrix and Vector distribution functions).

The first step is to then block distribute the diagonal of A to the first column of processors. We can make the observation that the diagonal elements of A are only on the diagonal processors, namely the diagonal of the diagonal processors is the diagonal of A. Thus each diagonal processor sends a message of size $O(\frac{n}{\sqrt{p}})$ to the corresponding processor in the first column. We then make a local copy of A with the diagonal elements set to 0, we call this matrix R. We also create a vector x of size n on the first column of processors, this vector is already block distributed since we can just make each piece locally on the first column processors.

Now we perform the Jacobi iteration as follows:

1. $w = Rx$, using distributed matrix multiplication
2. $x = (b - w)/D$, on first column only
3. $w = Ax$, using distributed matrix multiplication
4. $L2 = ||b - w||$, use L2 norm function
5. Broadcast L2 to all processors (message size of 1)
6. if L2 is less than a threshold terminate the algorithm, if not repeat, but if the number of iterations reaches 100 also terminate

The complexity of a single iteration of the loop is mostly dominated by the complexity of the matrix multiplication and thus has almost the same complexity. The thing that is different is we have a $O(\tau\sqrt{p})$ instead of $O(\tau log(p))$ due to the L2 norm calculation. Thus each iteration has Computation time $O(\frac{n^2}{p} + log(p))$ and Communication time $O(\tau\sqrt{p} + \mu\frac{n}{\sqrt{p}}log(p))$. Since the number of iterations of the algorithm is bounded by 100, the algorithm has a whole has the same complexity as a single iteration.

Runtime:
Computation - $O(\frac{n^2}{p} + log(p))$
Communication - $O(\tau\sqrt{p} + \mu\frac{n}{\sqrt{p}}log(p))$

1.7 L2 norm calculation

The L2 norm calculation function was implemented in same way as the gather function. The key difference is in gather function, the processors are sending their vector elements to 1^{st} processor while in L2 norm calculation function, the processors are sending the local L2 norm calculated.
Runtime :
Computation - $O(p)$
Communication - $O(\tau * \sqrt{p} + \mu * \sqrt{p} * 1) = O(\tau\sqrt{p} + \mu\sqrt{p})$

2 Performance Analysis

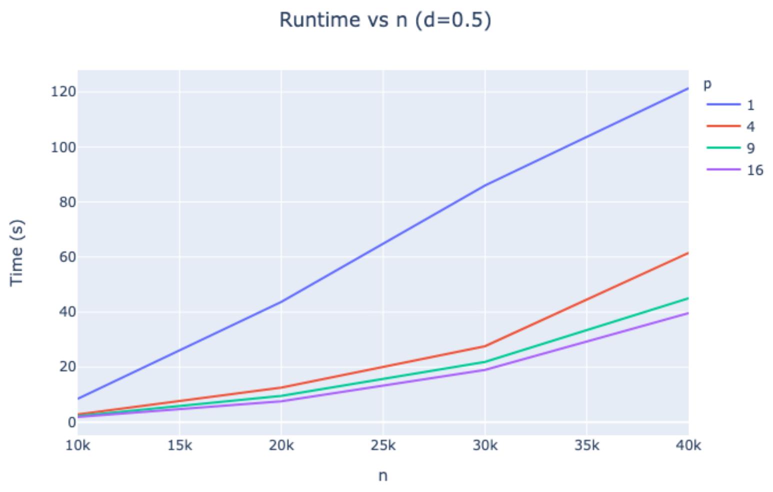
d=0.5

	P1	P4	P9	P16	P36	P64
n	Time (s)					
10000	8.47731	2.78904	2.15813	1.86609	9.69663	157.409
20000	43.6845	12.5086	9.58047	7.56447	-	-
30000	85.9854	27.5939	21.8752	18.9645	-	-
40000	121.305	61.5316	44.9935	39.5796	-	-

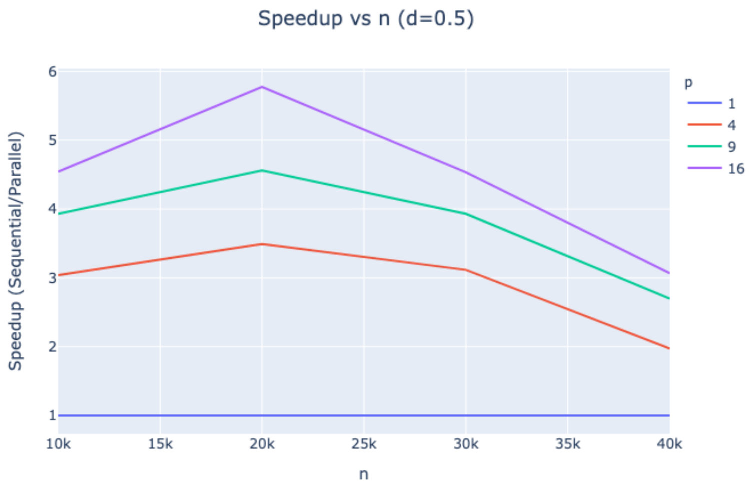
Observation - The run times for p = 36 and p = 64 are very likely incorrect as they are worse than even the sequential code, thus we suspect their was an issue with running the algorithm for large p on the cluster. Since this data seems erroneous we did not include them in the plots below.

n=20,000

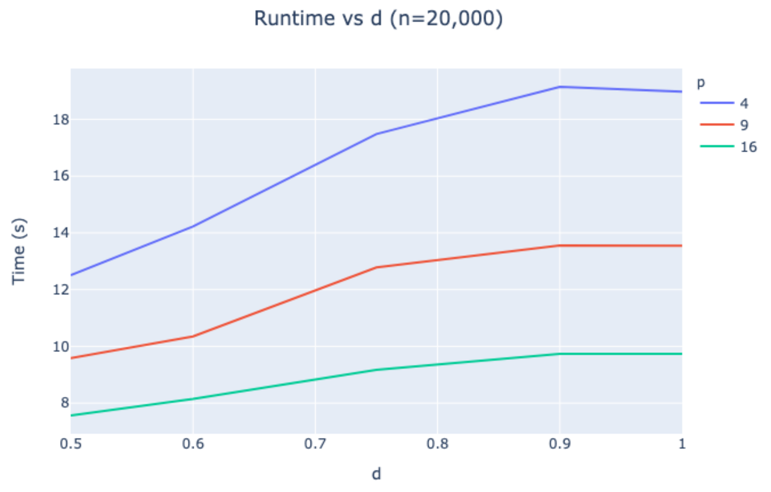
	P1	P4	P9	P16
d	Time (s)			
0.5	43.6845	12.5086	9.58047	7.56447
0.6	45.758	14.2247	10.3497	8.14397
0.75	73.7069	17.4849	12.7826	9.17506
0.9	99.6344	19.1482	13.5579	9.73624
1	141.763	18.9785	13.5443	9.7336



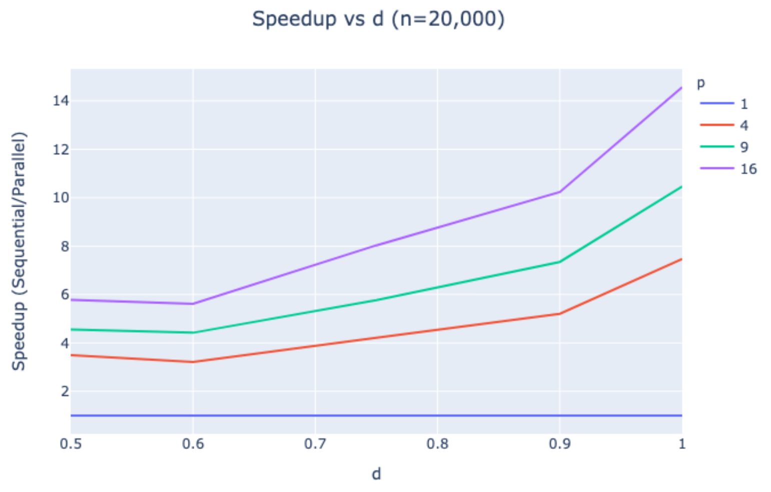
Observation - We observe that the as n is increased, there is almost linear relationship of run-time with respect to n. As expected, we observe that the run-time decreases as p is increased. For lower n values, the communication run-time dominates,thereby nullifying the computation speedup obtained by increasing p. As n is increased, the computation speedup dominates the communication slowdown, so the gap between run-time increases as n is increased.



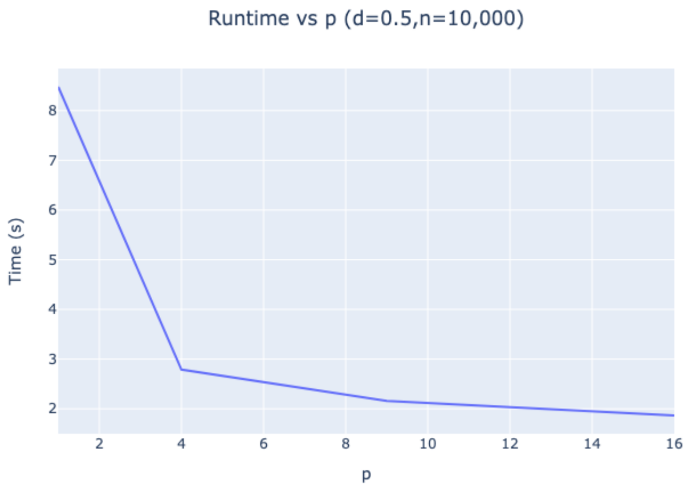
Observation - We see an interesting feature in the above graph, namely that the speedups of the parallel algorithm increases from 10,000 to 20,000, then for larger n the speedup always decreases. It is not immediately obvious why this would be the case, but it could be possible that this decreasing slope in the speedup is a result of the fact that as we increase n we significantly increase the amount of data that we have to distribute from the root processor to all the other processors. So even though the computation might be completed significantly faster for say p=16 compared to p=1 (i.e. sequential) the added cost of distributing a larger amount of data dramatically decreases the benefit of running the code in parallel.



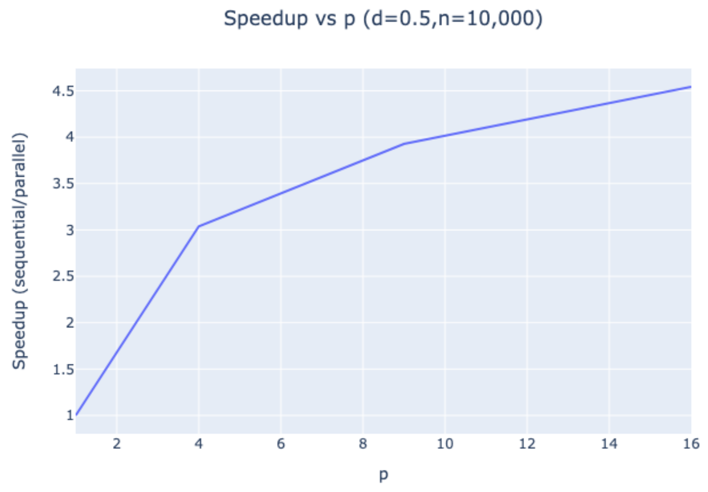
Observation - We observe that the run-time increases as we increase d keeping constant p. This is in-line with expectation that as d is increased, the Jacobi method takes more iterations to converge. As d is increased, after $d > 0.8$, the run time plateaus due to early termination of Jacobi method due to maximum iteration limit.



Observation - We observe that the speedup increases as we increase d keeping constant p. This is in-line with expectation that as d is increased, the Jacobi method takes more iterations to converge. As d is increased, after $d > 0.8$, the run time plateaus due to early termination of Jacobi method due to maximum iteration limit. The maximum iteration limiter is not implemented for sequential Jacobi, therefore there is an abrupt increase after $d > 0.8$, as the sequential Jacobi is converging to the solution and parallel Jacobi is terminating early, thus keeping the run-time of parallel implementation to be almost constant.



Observation - As expected we can see that the run time always decreases as we add on more processors, however we can see that the magnitude of the slope decreases rapidly. Thus even if we were to increase p to a much larger number we might not expect to see a significantly faster run time. In particular the largest gains are seen for p = 4, but for p larger than 4 we only see a minor decrease in the run time.



Observation - These results are in agreement with the runtime plot. Namely that we see the largest increase in speedup for p = 4 and as we increase p the slope tends to decrease thus for each processor we add we are getting less and less of a benefit out of it. In terms of efficiency we can see that the parallel algorithm is most efficient for p = 4 with an efficiency of roughly $\frac{3}{4}$ which is near the optimum of 1 and for larger p the efficiency would drop off rapidly since each new processor being added doesn't provide much benefit to the speed of the parallel algorithm.