CSE 6220: Programming Assignment 3 Report

Reuben Tate and Shasha Liao

April 2020

1 Algorithm Description

Algorithm 1: Jacobi($n \times n$ matrix A, $n \times 1$ column vector b, integer max_iter, float $l_2\_tolerance$, output variable $n \times 1$ column vector x)

`// We assume that` A `is distributed amongst the processor grid and that` b `is distributed amongst the processors in the first column`

1 Set $D := diag(A)$. Note that the diagonal elements of A are going to be along the diagonal of the local matrix of the processors on the diagonal of the processor grid. For each "diagonal" processor, we send its diagonal elements to the processor to the left of it in the first column.

2 Set x to be the all-zeros vector of length n. We will store x in the first column in a distributed fashion. Note that each processor can locally initialize its part of x to 0.

3 **for** $i = 1$ *to* $i = max\_iter$ **do**

4 $w := Ax$. This can be done using distributed matrix-vector multiplication. The result w is distributed amongst the first column of processors in the processor grid.

5 $x := D^{-1}(b - w) + x$. Here the division is componentwise. Note that each processor in the first column has everything it needs to update its portion of x locally (i.e. it just needs to calculate $x_i = \frac{1}{d_i}(b_i - w_i) + x_i$ for the relevant indices i)

6 $l_2 := \|b - w\|$. To calculate this, we first have all processors in the first column of the processor grid compute a "local l_2". That is, if that processor has elements $b_i, b_{i+1}, \ldots, b_j$ and $w_i, w_{i+1}, \ldots, w_j$ (with $i \leq j$), then that processor will first calculate $\sum_{k=i}^{j}(b_k - w_k)^2$. We then run MPI_Reduce with the sum operation along the first column in the processor grid to obtain $\sum_{k=1}^{n}(b_k - w_k)^2$. Taking the square root of this then yields the l_2 norm of $b - w$. To make sure that all the processors know the norm, we broadcast the value (from the top-left processor in the grid) to everyone else.

7 **if** $l_2 < l_2\_tolerance$ **then**

8 exit. Note that the condition above can be calcualted locally after l_2 has been broadcast to everyone.

We will now explain (in words) about the other subroutines. One subroutine is matrix-vector multiplication of Ax to produce b. The algorithm assumes that A is distributed across the processor grid and that b is distribute across the first column of the processor grid. We first run the function `transpose_bcast_vector` to have every processor have the correct portion of x it needs. Then each processor does local matrix-vector multiplication with its local portion of A and the local portion of x it received. We then use MPI_Reduce along each row to do a componentwise sum of all the vectors that were computed during the local matrix-vector multiplication. The results of the reduce are stored in the processors in the first column of the processor grid and together they form the vector b. The correctness of this algorithm was demonstrated in Homework 5.

For the `transpose_bcast_vector` function, we used MPI_Send and MPI_Receive to send the data from the processors in the first column of the processor grid to those in the first row. Then each processor in the

first row of the processor grid broadcasted its data downward to the other processors.

The `distribute_matrix` function was done by first distributing the data from the rank-0 processor to the processors in the first column of the processor grid(that is, if a processor in the first column was in the ith row of the processor grid, then it would be sent the data for *all* the processors in the ith row. Then, for each row, we distribute the data from the leftmost processor to the remaining processors in that row. There was a technical detail when it came to sending the data. During the 2nd round of distributing the data (i.e. distributing the data across the rows), the data to be sent was not contiguous in the buffers' of those processors in the 1st column. However, we observed that it *would* be contiguous if we took the transpose of the matrix. So we implemented a transpose operation and sent the data. After the data was sent, another transpose had to be done to undo the first transpose and to obtain the final local matrix in each processor.

For `distribute-vector` and `gather-vector`, we just used the scatter and gather communication primitives.

2 Data Gathered and Analysis

For every combination of the following parameters below, we ran our algorithm 10 times and took the average runtime.

- n (number of rows): 100, 500, 1000, 5000, 10000
- p (number of processors):
  - 1 (1 node)
  - 4 (1 node)
  - 9 (1 node)
  - 16 (1 node)
  - 25 (1 node)
  - 36 (2 nodes)
  - 49 (2 nodes)
  - 64 (3 nodes)
- d (difficulty): 0.1, 0.5, 0.9

Our group is not fully aware of the hardware implementation of the PACE cluster; however, previous experience showed us that using more than 1 node would cause both an overall increase in time and unstable runtime results (probably due to inter-node communication being more costly than intra-node communication). However, it was impossible to have 64 processors on 1 node in the $p = 64$ case because PACE only supports up to 28 processors per node. When submitting our jobs, we chose the number of nodes to be as small as possible. In the `Plots` section below, the plots but they can be grouped into two types:

- **Type 1**: has n as the x-axis, the runtime or speedup as the y-axis. Each curve corresponds to a different value of p. Each plot corresponds to a fixed value of d
- **Type 2**: has p as the x-axis, the runtime or speedup as the y-axis. Each curve corresponds to a different value of d. Each plot corresponds to a fixed value of n.

Below is a listing of various trends/observations regarding the plots.
Analysis of Type 1 Plots:

- Everything else being fixed, as n grows, the runtime also usually grows (for obvious reasons).

- Looking at runtime vs n, the curve for $p = 16$ seems to behave very differently. We are unsure of why this occurs.
- We see that an increase in the number of processors is beneficial for large n but not so much for small n. Although asymptotically, we expect more processors to give us better performance, when n is small, the communication initialize cost τ might be prohibitively expensive compared to everything else.
- For all values of p, the speedup is eventually greater than 1 for n large enough (with larger p's tending to have larger speedups). For all values of p, it seems that the speedup is less than 1 for very small n (e.g. $n = 100$).

Analysis of Type 2 Plots:

- With a few exceptions, it seems that as the difficulty increases, then so does the runtime as expected.
- When $n = 10000$, it seems that n is large enough where we see the phenomena where more processors leads to reduced runtimes. There is a notable exception at $p = 32$ processors. We hypothesize that this may be due to the fact that this is the first value of p where we use 2 nodes on the cluster (instead of 1 node).

Another thing that makes the analysis difficult is that from a theoretical point of view, we don't know how the number of iterations of the algorithm depends on n without knowing about the convergence rate of Jacobi a priori. The overall runtime is going to be determined by both the number of iterations and the amount of time it takes for each iteration. A more detailed analysis would be possible if we had recorded the number of iterations when running our jobs and then plotted the number of iterations as well as the average amount of time per iteration.

Another issue that there are two termination conditions: the max iteration condition and the l_2 condition.

Lastly, another thing to consider is that as n increases, since we fix the l_2 termination value, the x value is forced to become more and more accurate (in the l_1 or componentwise sense) before we are allowed to terminate. Thus, its possible we could have seen different results if we had an l_1 termination condition instead or if the l_2 termination condition was dependent on n.

3 Ideas to Improve Performance

- One improvement to performance that we did actually implement was to not bother calculating $R := A - D$ in the algorithm description provided in the template source code. By doing this, we save memory (since we don't need to store R) and time (since we don't need to initialize R). Let $w' := Rx$ denote the old definition of w (as provided in the template source code), and let $w = Ax$ be the new definition of w as seen in our algorithm description above. Then,

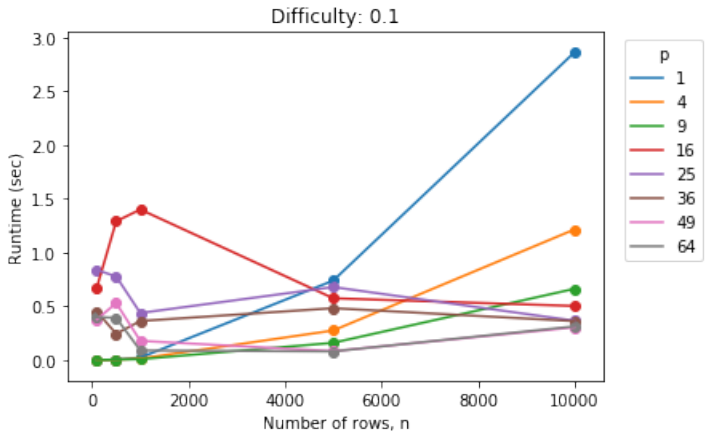
  $$x \leftarrow D^{-1}(b - w') = D^{-1}(b - Rx) = D^{-1}(b - (A - D)x) = D^{-1}(b - Ax) + x = D^{-1}(b - w) + x$$

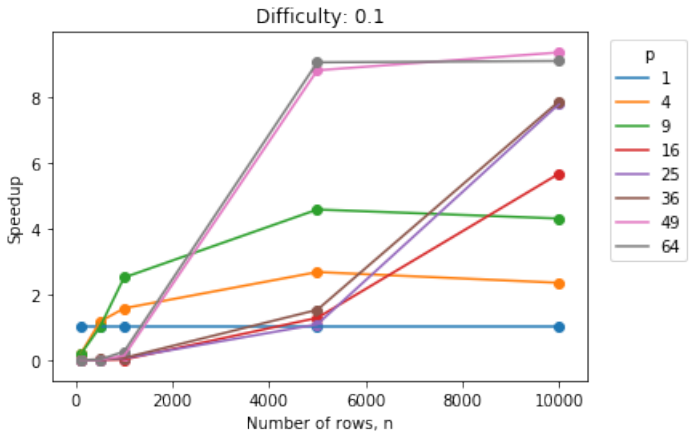
  which corresponds to the update step seen in the our algorithm description above
- Our group isn't very skilled at C/C++ so although we tried our best to be aware of memory issues (e.g. memory leaks, etc), there still might be spots in the code where memory usage could be improved.
- The function could be improved for special cases, for example if A is symmetric or if the input matrix or vector is sparse.
- Since x is initialized to the zero vector, we know that at the first iteration, we have that $w = Rx = R(0) = 0$. Thus, we can save a small amount of time by not actually doing this matrix-vector multiplication since we know the result will be zero.

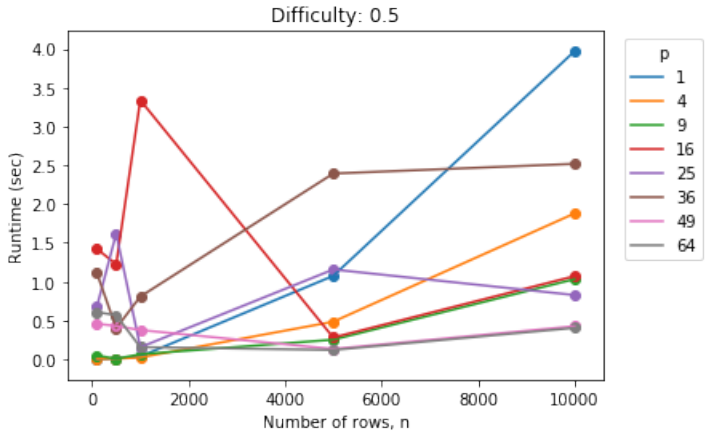
4 Plots

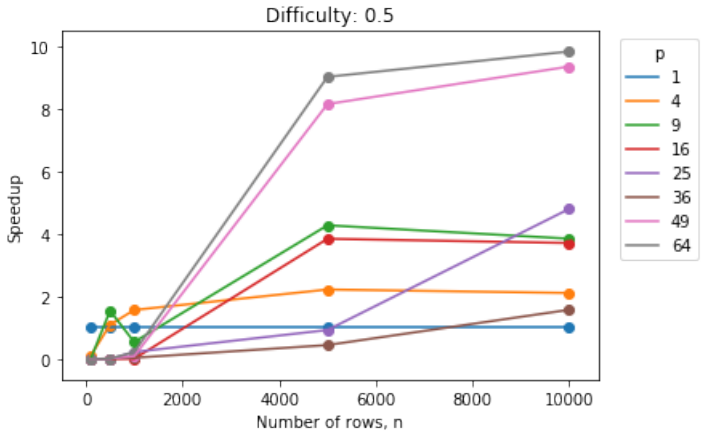
Please see section 2 for how these plots are constructed.

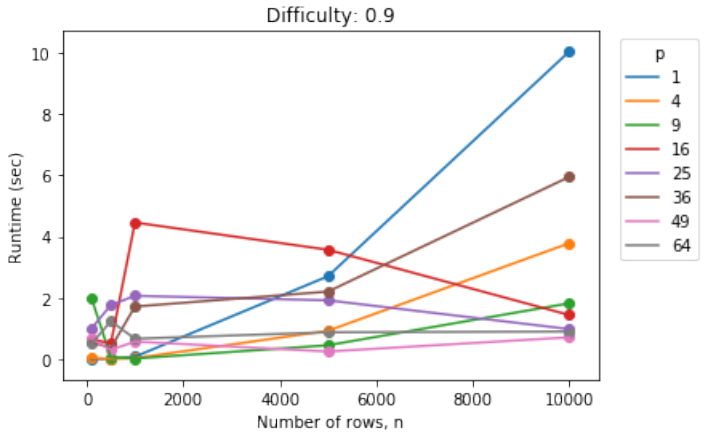
Type 1 Plots

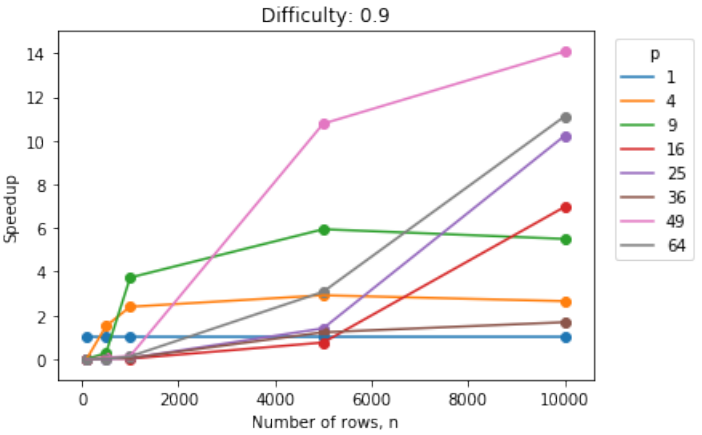




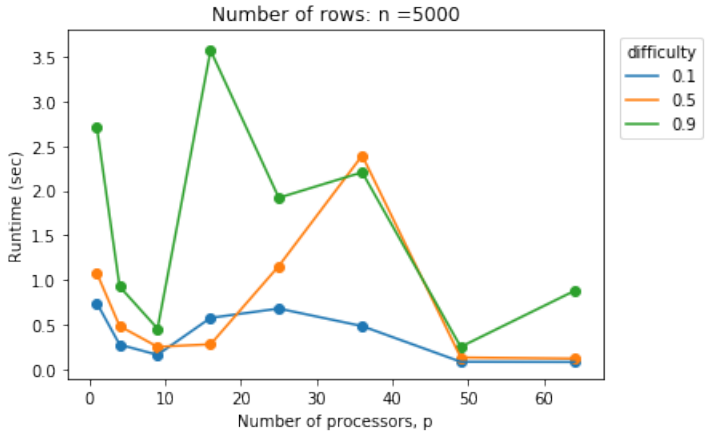


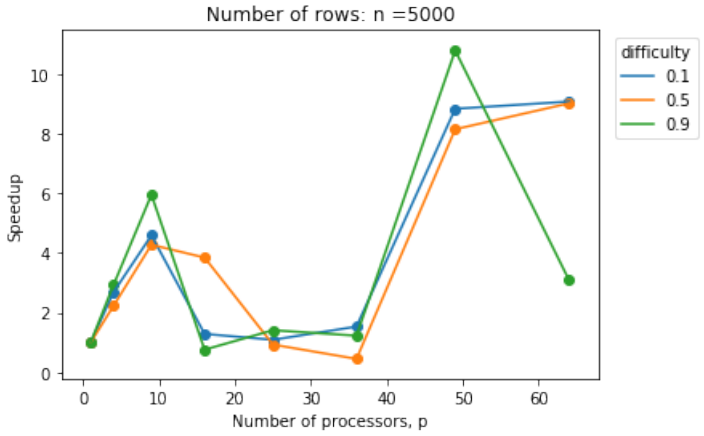


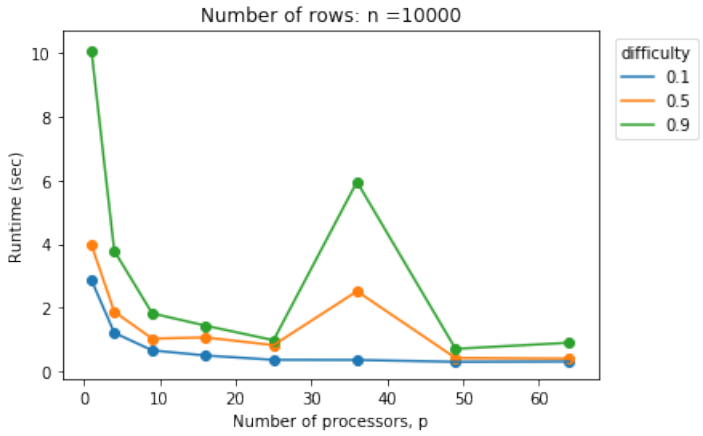


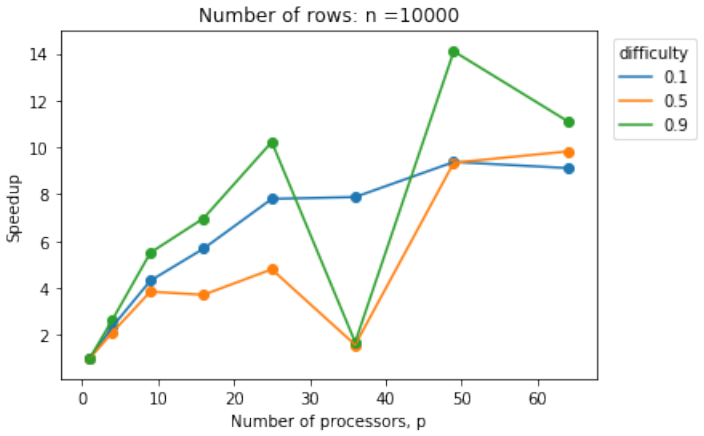


Type 2 Plots


Number of rows: n =5000
Runtime (sec)
Number of processors, p
difficulty
0.1
0.5
0.9


Number of rows: n =5000
Speedup
Number of processors, p
difficulty
0.1
0.5
0.9


Number of rows: n =10000
Runtime (sec)
Number of processors, p
difficulty
0.1
0.5
0.9


Number of rows: n =10000
Speedup
Number of processors, p
difficulty
0.1
0.5
0.9