Stefen Pegels sgp62
ECE5720
4 April 2021

Homework 4 Report

1) Description of Algorithm

The purpose of this assignment was to orthogonalize a matrix A, through the use of the Jacobian method of singular value decomposition. To do this, we incorporated MPI message passing to distribute orthogonalization work to multiple processing elements in a ring structure. The algorithm has two main parts: the Givens Matrix rotation and the row passing.

1a) Givens Matrix Rotation

In order to orthogonalize A, each row of A must have a dot product of 0 with all other rows of A. This means for a matrix A with N rows, we need to calculate (N*(N-1))/2 different rotations so that every row is orthogonal to every other row.

To perform a rotation, two rows of matrix A were taken and the dot products of row combinations were placed into the following matrix B:

$$B = \begin{pmatrix} b_{2p,2p} & b_{2p,2p+1} \\ b_{2p+1,2p} & b_{2p+1,2p+1} \end{pmatrix}$$

Where

$$b_{2p,2p} = a_{2p}^{(k)}(a_{2p}^{(k)})^T,\ b_{2p,2p+1} = a_{2p}^{(k)}(a_{2p+1}^{(k)})^T = b_{2p+1,p},\ b_{2p+,2p+1} = a_{2p+1}^{(k)})(a_{2p+1}^{(k)})^T$$

After the 2x2 matrix B was constructed, the Givens matrix G was calculated from the following equality:

$$\begin{pmatrix} c_p & s_p \\ -s_p & c_p \end{pmatrix}^T \begin{pmatrix} b_{2p,2p} & b_{2p,2p+1} \\ b_{2p+1,2p} & b_{2p+1,2p+1} \end{pmatrix} \begin{pmatrix} c_p & s_p \\ -s_p & c_p \end{pmatrix} = \begin{pmatrix} \hat{b}_{2p,2p} & 0 \\ 0 & \hat{b}_{2p+1,2p+1} \end{pmatrix}$$

The operation G^TBG creates a diagonal matrix, and the G matrix represents the sine and cosine of a given angle, so $c^2+s^2=1$ for every Givens matrix. To calculate the givens matrix values s and c, the following root equations are used:

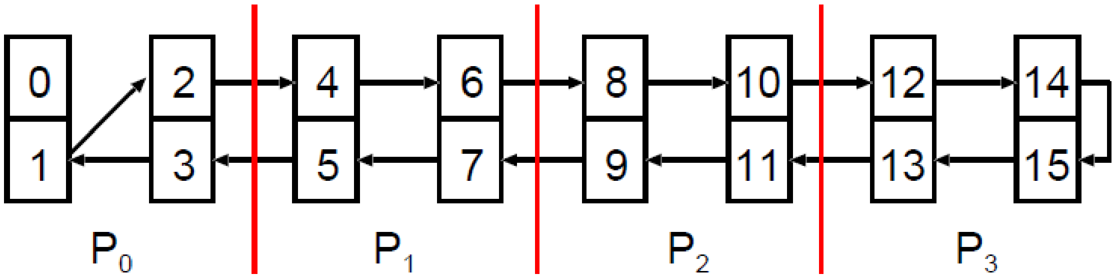
$$\tau = \frac{a_{q,q} - a_{p,p}}{2a_{p,q}},\quad t = -\tau \pm \sqrt{1+\tau^2},$$

$$c = \frac{1}{\sqrt{1+t^2}},\quad s = ct$$

The calculated values of c and s populate the G matrix, which is then transposed and used in the operation G^TA_local, where A_local stores two rows from the larger matrix A. This then orthogonalizes those two rows of A. This rotation algorithm takes place in the *givens_rotate()* function in the code.

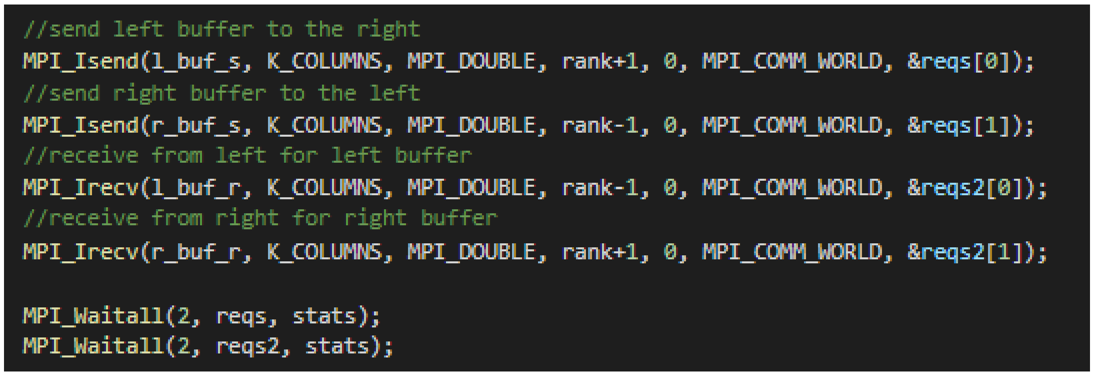
1b) Row Passing

Each iteration of the Jacobian algorithm, a sweep, comprises a for loop that runs for 2 times the number of processing elements. This is because each processing element operates on two rows, so we need that many iterations to operate on every combination of rows. The communication structure follows this ring pattern:



In this pattern, every PE sends and receives nodes from its neighbors, with node 0 and the last node having special behavior. Node 0 never moves row 0 so that row 0 can interact with every other row, and the final row on the right moves one spot instead of two. Each iteration of the loop, a node passes one row and receives another from each side, so that it operates on two different rows from before. After 2*npes loops, every row will have been paired with every other row in exactly one node, so that they are all orthogonal. This comprises a full sweep.

This message passing logic is found in the main for loop of *sgp62_jacobi.c*, where *l_buf_s* denotes the buffer to send a row to the left, *l_buf_r* denotes the buffer to receive a row from the left, *r_buf_s* denotes the buffer to send a node to the right, and *r_buf_r* denotes the buffer to receive a node from the right. MPI_send and MPI_recv are used with the following logic to send and receive rows. For a processing element not on the edges:



```
//send left buffer to the right
MPI_Isend(l_buf_s, K_COLUMNS, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, &reqs[0]);
//send right buffer to the left
MPI_Isend(r_buf_s, K_COLUMNS, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &reqs[1]);
//receive from left for left buffer
MPI_Irecv(l_buf_r, K_COLUMNS, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD, &reqs2[0]);
//receive from right for right buffer
MPI_Irecv(r_buf_r, K_COLUMNS, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD, &reqs2[1]);

MPI_Waitall(2, reqs, stats);
MPI_Waitall(2, reqs2, stats);
```

MPI_Waitall is used at the end of a message pass to ensure no nodes get ahead of the others.

After exchanging rows, each node copies its two rows into *A_local* and performs the Givens rotation on its pair of rows.

At the end of each iteration, error checking is performed by computing the sum of the squares of the non diagonal elements of $A * A^T$. These values are collected and summed with MPI_Allreduce, so that each PE knows if it should stop executing before MAX_ITERS is reached.

Thus the control flow for the algorithm looks like this:

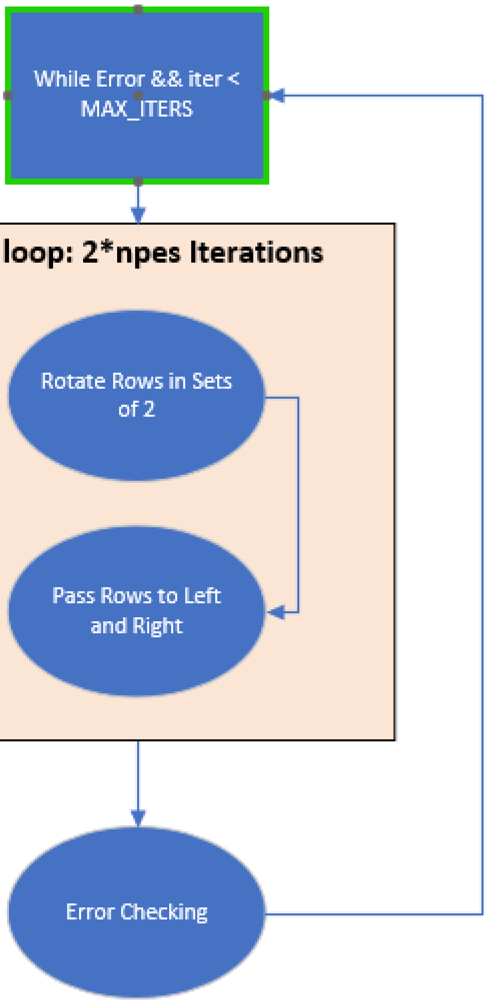

While Error && iter < MAX_ITERS
For loop: 2*npes Iterations
Rotate Rows in Sets of 2
Pass Rows to Left and Right
Error Checking

Figure 1: *Control Flow Diagram for Jacobian Algorithm*

2) Discussion of Results

The Jacobian algorithm was benchmarked at different matrix dimensions: 128, 512, and 1024, with processing element benchmarks of 8, 32, and 64. Note that on the ecelinux servers, a node has 16 cores, so a node could only work on 16 tasks at once. The following graph of timing was obtained:

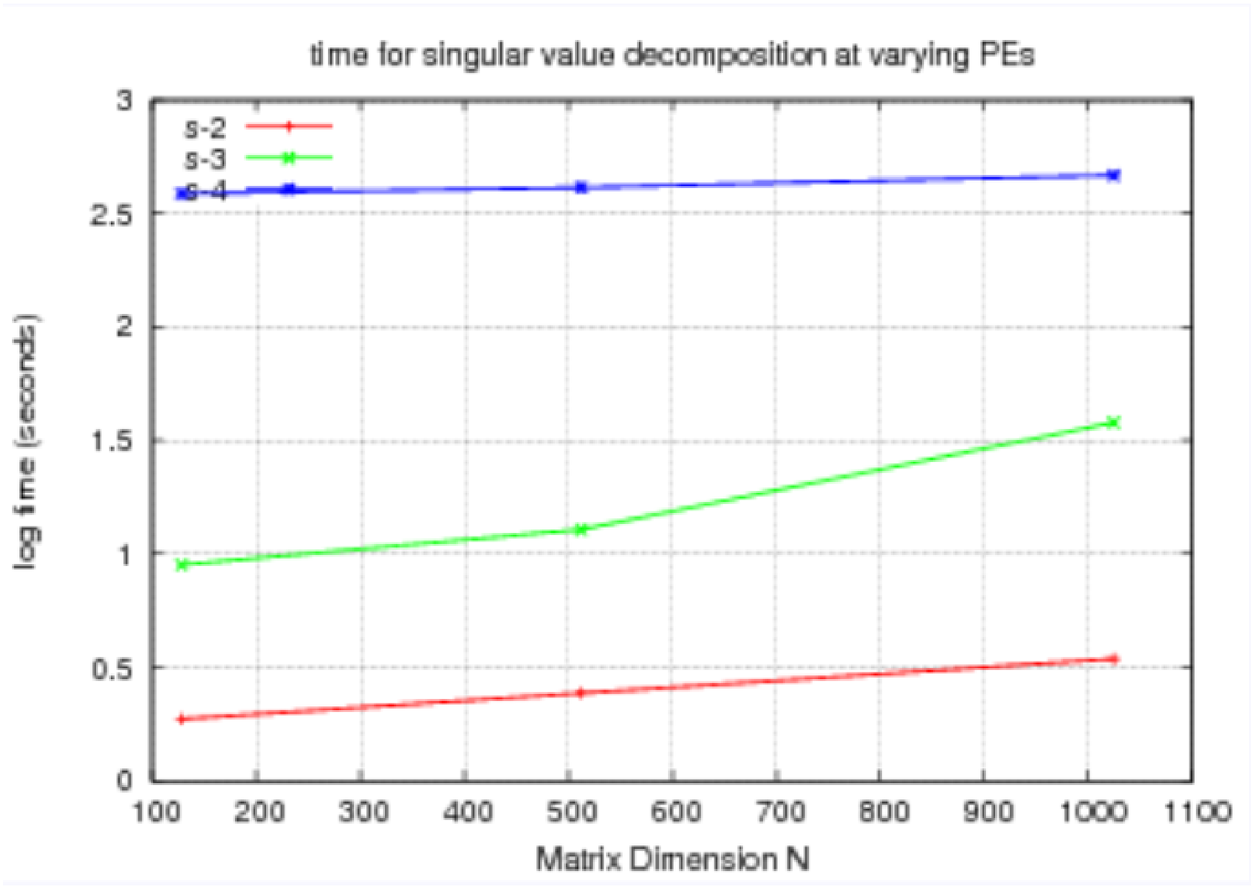


Figure 2: *Timing Graph for Jacobian Algorithm: Dimension vs log execution time*

Note that in the above image, Red = 8 PEs, Green = 32 PEs, and Blue = 64 PEs. Also, for a given dimension n, the amount of columns K_COLUMNS was decreased to preserve the total number of elements in the matrix between each processing element benchmark.

From the graph, higher PEs had the opposite effect of improvement through parallelization: 64 PEs had the slowest execution time compared to the other two combinations. While in other assignments (see HW3: nbody report), increased parallelization resulted in substantial speedups, in this assignment the communication overhead with MPI outweighed possible parallelization improvements. To analyze communication time, a ping pong benchmark was performed, where a char array of varying sizes was sent back and forth over nodes and timing was measured.

Non Blocking Bandwidth Test

The algorithm of the nonblocking test, found in *sgp62_nonblock.c* constituted the following:

1) Assign task pairs based on n_tasks so that they are equidistant from the median task
2) For tasks in the first half
    a) Send and receive a char array of varying sizes, starting at 1 and ending at 1000000 elements, multiplying by 10 each time

    b) Repeat ping pong for 20 repetitions, taking average time
    c) Change array size and take new timing results
3) Gather all results in root, creating an array of timings to analyze

The results for ping pong timing per byte are shown in the graph below: Note that testing with a blocking version of the algorithm yielded nearly identical results, thus they are not shown for clarity.

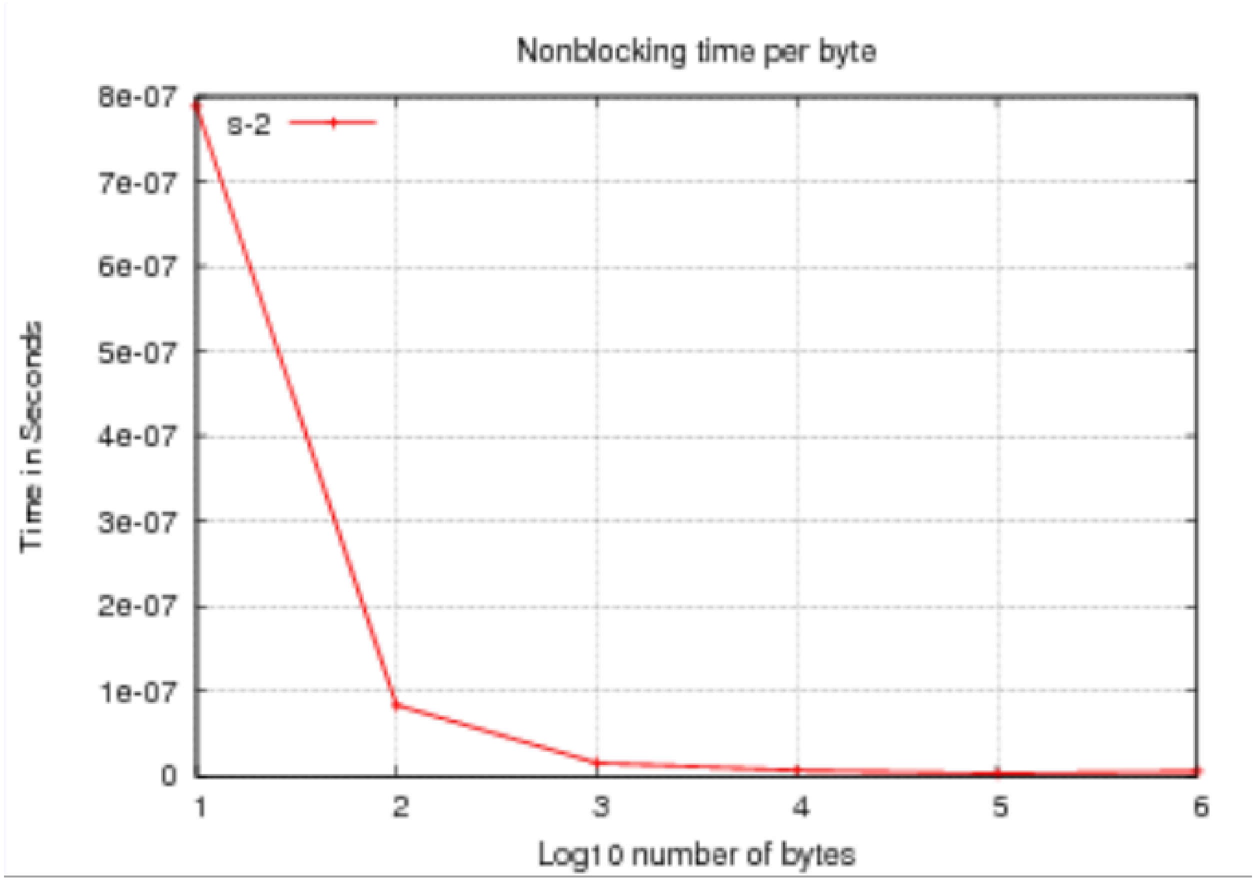


Figure 3:*Timing for nonblocking bandwidth test, for varying byte message lengths.*

From the graph, it appears that the smaller arrays took much longer **per byte** to send. This seems to indicate that there is some kind of non negligible start up time associated with an MPI message transfer, that is **more significant** for shorter lengths of messages sent.

If we apply this timing logic to the Jacobian results, it would make sense that larger numbers of PEs would suffer more from the communication overhead, since they are performing more transactions with other PEs, rather than queueing up rows within their own PE in a sequential fashion. Also notable is the fact that for a send and receive in the jacobian algorithm, the maximum row length tested was 1024, much smaller than the maximum value used in the nonblocking example.

Thus we come to the conclusion that communication overhead for OpenMPI is a significant factor in timing results, and can outweigh the effects of parallelization in some cases.

To improve my algorithm, I would most likely experiment with other MPI row passing structures, possibly creating queues of rows that aren't being operated on to maybe diminish time losses due to message passing.

3) Notable Instructions for Executing Files

For my files, the arguments MAX_ITERS and K_COLUMNS at the start of the file are changeable to benchmark the code in multiple ways. To run, compile the job by linking the -lm math library (for use with square root calculations), otherwise compile with mpi and sbatch to the SLURM scheduler as normal. Note that graphing scripts for the timings were written in another location and not included with the final submission; they modeled gnuplot scripts from hw3_nbody.