Cole Bardin
ECEC-413
2/21/25

Jacobi Solver Report

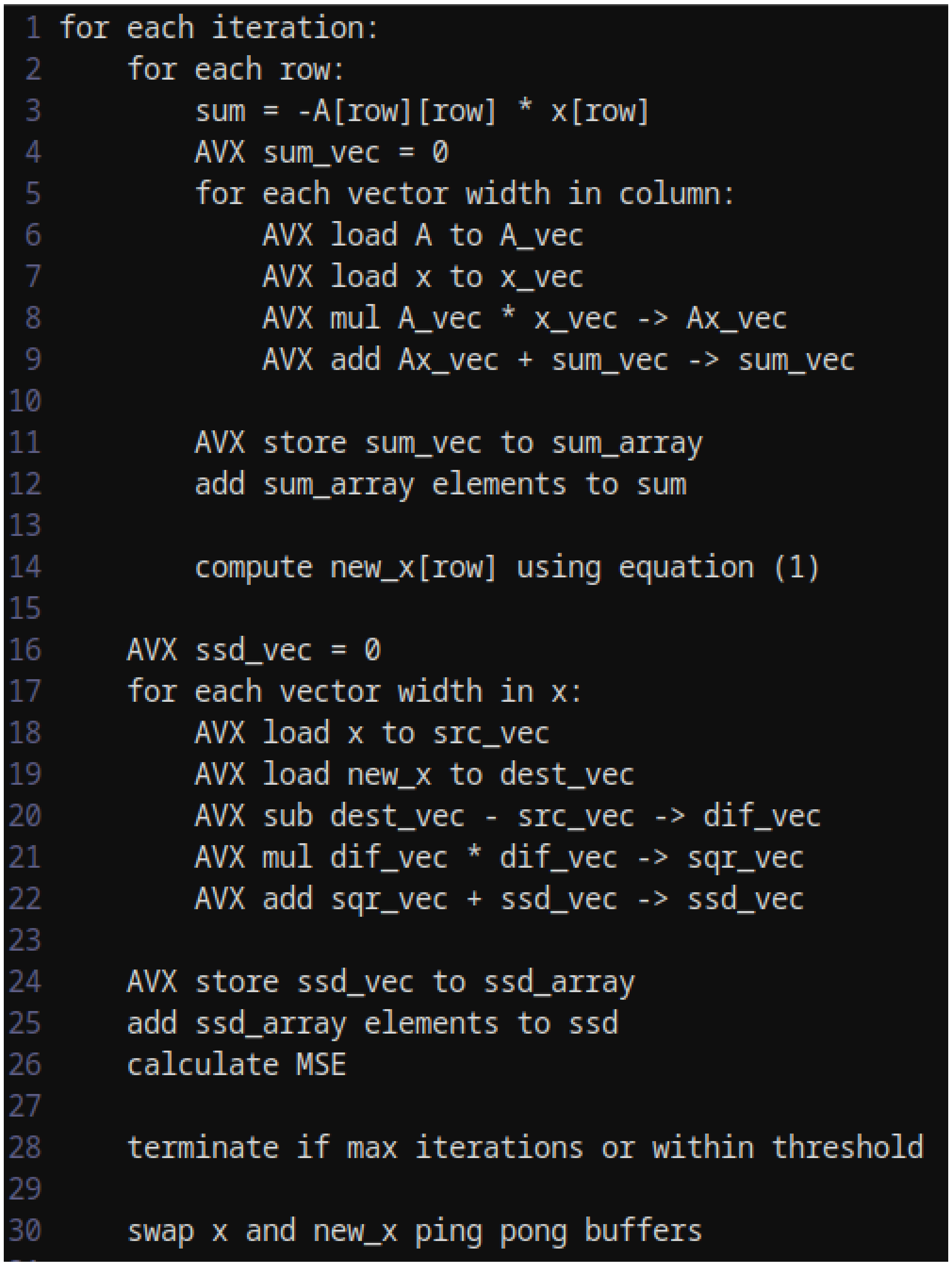
Introduction

The Jacobi method is an iterative algorithm used to solve systems of linear equations, particularly for large, sparse matrices. It is based on decomposing the system matrix into its diagonal, upper triangular, and lower triangular components, and then iteratively updating the solution vector until convergence is achieved. While the method is straightforward and easy to implement, it is often slow to converge, especially for large systems. However, its iterative nature makes it highly suitable for parallelization, as each element of the solution vector can be updated independently during each iteration.

Approach

There are two types of parallelism that can be applied to the Jacobi method: coarse and fine grained parallelism. Fine grained parallelism is implementing vector instructions to improve the iterative summation process. Coarse grained parallelism is utilizing pthreads to perform calculations on multiple rows simultaneously. Two alternative methods to the serial Jacobi solver implementation were developed, one that uses fine grained parallelism and one that uses both coarse and fine grained parallelism.

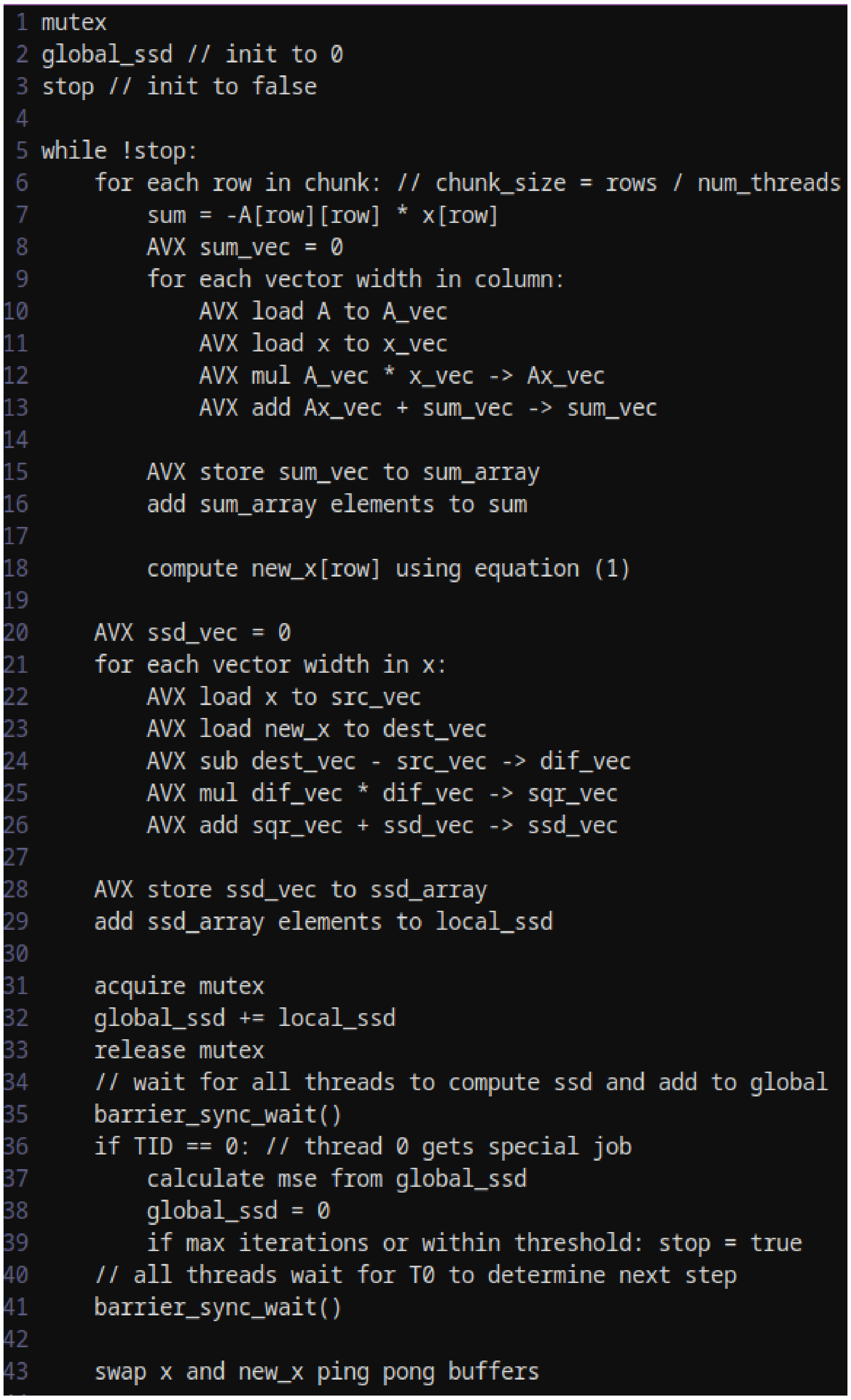
To implement fine grained parallelism through the use of vector instructions, the program uses AVX. There are two places where AVX instructions can be utilized to speed up the arithmetic. AVX instructions can be used to parallelize the summation of the column elements multiplied by the x values when calculating the new x value. AVX instructions can also be used to compute the sum of squared differences between the old and new solution. Pseudo code can be seen below:



```
for each iteration:
    for each row:
        sum = -A[row][row] * x[row]
        AVX sum_vec = 0
        for each vector width in column:
            AVX load A to A_vec
            AVX load x to x_vec
            AVX mul A_vec * x_vec -> Ax_vec
            AVX add Ax_vec + sum_vec -> sum_vec

        AVX store sum_vec to sum_array
        add sum_array elements to sum

        compute new_x[row] using equation (1)

    AVX ssd_vec = 0
    for each vector width in x:
        AVX load x to src_vec
        AVX load new_x to dest_vec
        AVX sub dest_vec - src_vec -> dif_vec
        AVX mul dif_vec * dif_vec -> sqr_vec
        AVX add sqr_vec + ssd_vec -> ssd_vec

    AVX store ssd_vec to ssd_array
    add ssd_array elements to ssd
    calculate MSE

    terminate if max iterations or within threshold

    swap x and new_x ping pong buffers
```

Figure 1. Pseudo code for fine grained parallelism with AVX instructions

While this approach can improve runtime performance, there is room for improvement. Each row is an independent calculation that this method is performing sequentially. Pthreads can be used to parallelize these row operations through chunking. However, it is critical that proper thread synchronization is performed as all rows must be iterated over before calculating the total sum of squared differences and evaluating the performance. Pseudo code for this approach can be seen below:



```
mutex
global_ssd // init to 0
stop // init to false

while !stop:
    for each row in chunk: // chunk_size = rows / num_threads
        sum = -A[row][row] * x[row]
        AVX sum_vec = 0
        for each vector width in column:
            AVX load A to A_vec
            AVX load x to x_vec
            AVX mul A_vec * x_vec -> Ax_vec
            AVX add Ax_vec + sum_vec -> sum_vec

        AVX store sum_vec to sum_array
        add sum_array elements to sum

        compute new_x[row] using equation (1)

    AVX ssd_vec = 0
    for each vector width in x:
        AVX load x to src_vec
        AVX load new_x to dest_vec
        AVX sub dest_vec - src_vec -> dif_vec
        AVX mul dif_vec * dif_vec -> sqr_vec
        AVX add sqr_vec + ssd_vec -> ssd_vec

    AVX store ssd_vec to ssd_array
    add ssd_array elements to local_ssd

    acquire mutex
    global_ssd += local_ssd
    release mutex
    // wait for all threads to compute ssd and add to global
    barrier_sync_wait()
    if TID == 0: // thread 0 gets special job
        calculate mse from global_ssd
        global_ssd = 0
        if max iterations or within threshold: stop = true
    // all threads wait for T0 to determine next step
    barrier_sync_wait()

    swap x and new_x ping pong buffers
```

Figure 2. Pseudo code for coarse and fine grained parallelized thread body logic

This approach, however, requires that the number of rows/columns can evenly be divided by the number of threads times the vector width. If not, then significantly more logic must be implemented.

Results

Implementing these approaches, the program can be run for matrix sizes 512x512, 1024x1024, 2048x2048 with thread counts of 2, 4, 8, 16, 32. The program was run on XUNIL-05 with 12-cores and 24-threads across 2 Intel Xeon E5-2620s with 48MB of RAM running CentOS 7 with Linux Kernel 3.10.

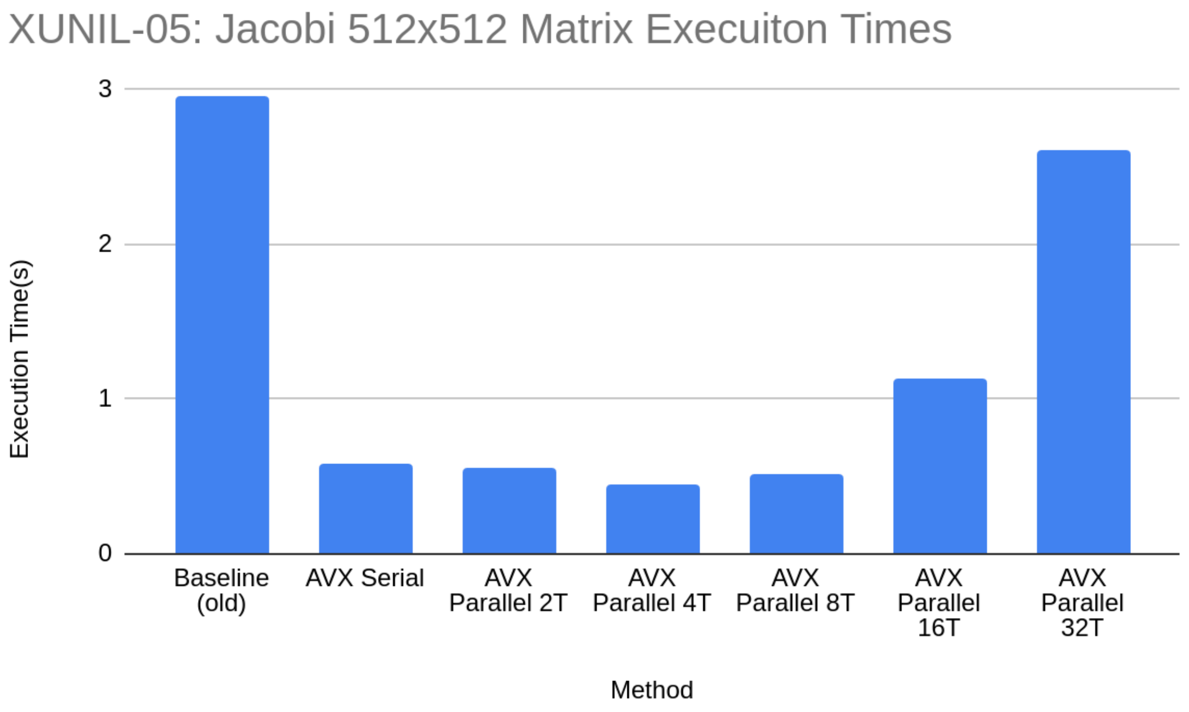


Figure 3. Execution time results for 512x512 matrix for baseline, AVX serial, and multi-threaded AVX methods

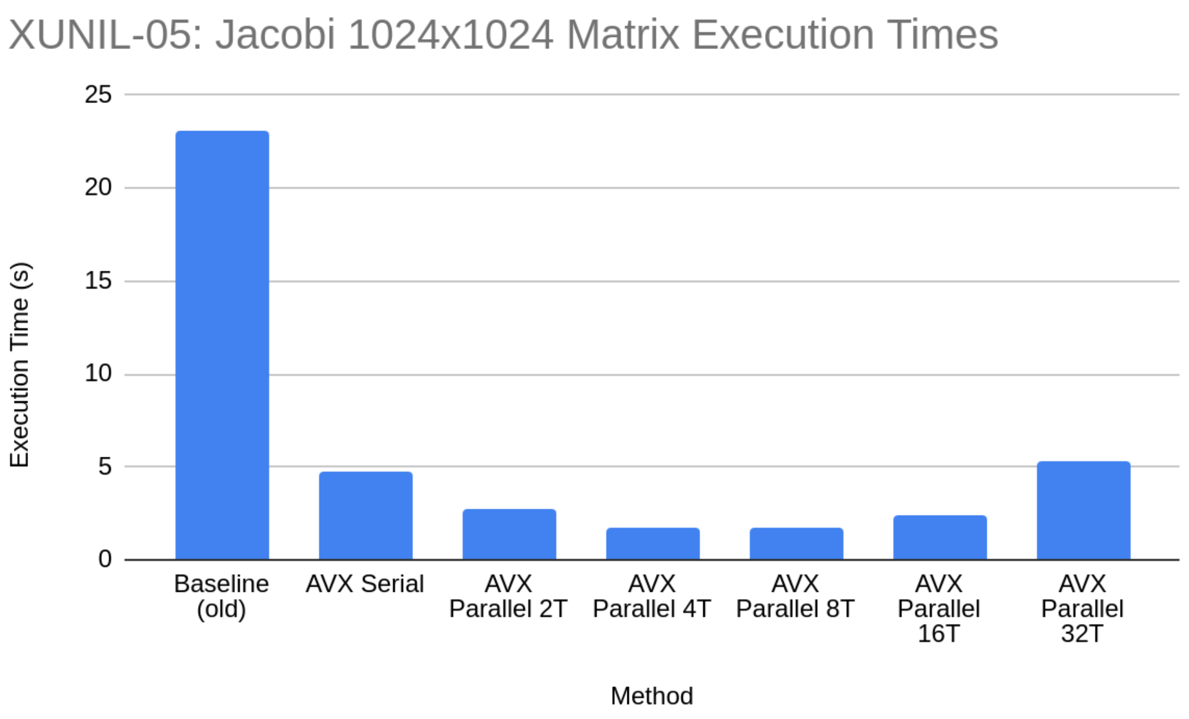


Figure 4. Execution time results for 1024x1024 matrix for baseline, AVX serial, and multi-threaded AVX methods

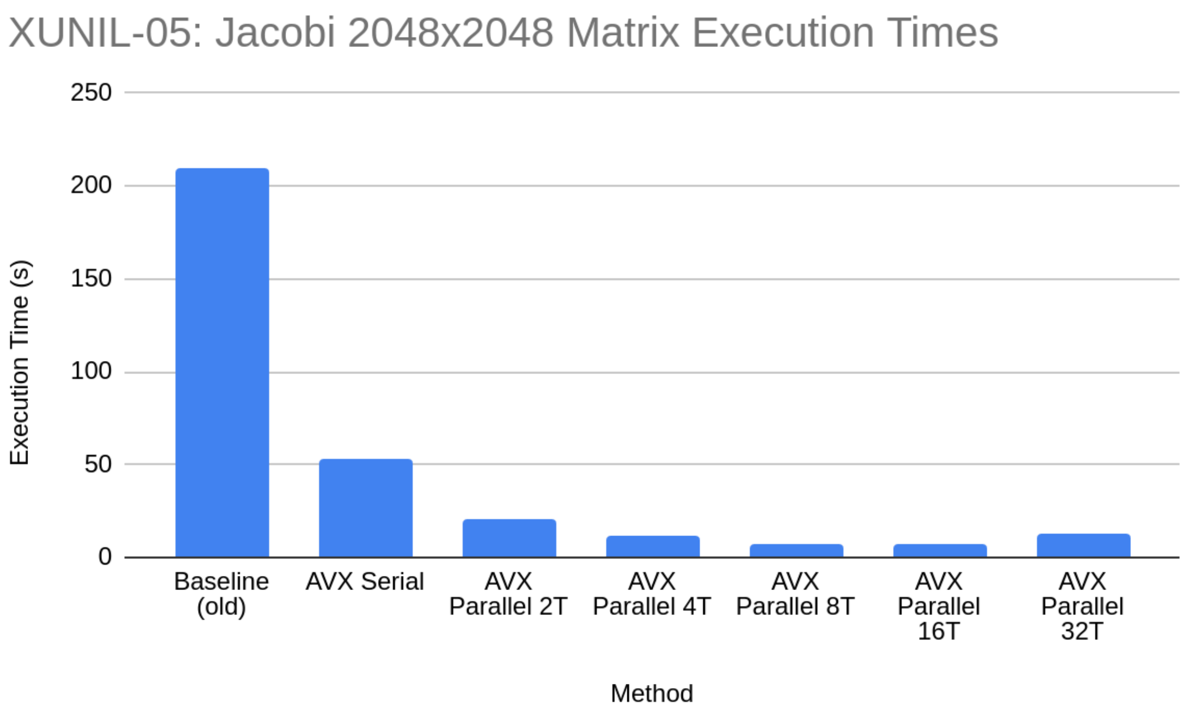


Figure 5. Execution time results for 2048x2048 matrix for baseline, AVX serial, and multi-threaded AVX methods

Conclusion

Utilizing fine grained and coarse grained parallelism can provide significant performance improvements in terms of reducing runtime for the Jacobi Solver compared to the serial baseline. This performance improvement increases in proportion to the size of the matrix. While increasing the number of threads can improve execution time, using too many threads can significantly hinder the performance improvements as the overhead for thread synchronization increases. Also, all cases with 32 threads performed worse than 16 threads. This is because the platform used only has 12 cores/24 threads. Therefore, it could not run all 32 threads simultaneously and the parallelism was lost. As the size of the matrix increased, this performance loss was decreased but with small matrices and excessive number of threads, there is little to no performance gain. This shows how it is critical to adjust how parallelism is implemented depending on the platform and computation intensity.