Project 3 - Jacobi Method - Kevin Dougherty

Kevin Dougherty

May 2025

1 Introduction

I have chosen to implement the Jacobi Iterative Method from linear algebra. It is a method to solve a linear equation of the form $Ax = B$ where in each step we update our initial guess until we are within some bounded error. The basic algorithm is as follows:

```
k = 0
while convergence not reached do
    for i := 1 step until n do
        sigma = 0
        for j := 1 step until n do
            if j != i then
                sigma = sigma + aij xj(k)
            end
        end
        xi(k+1) = (bi - sigma) / aii
    end
    increment k
end
```

Listing 1: Jacobi Iterative Method

In each iteration, a calculation is performed on the updated X. If the desired convergence is achieved, the function outputs X. The desired convergence is typically calculated using the spectral radius, but other options can be used, such as the L1/L2 norm, or the sum of the absolute difference between respective values between iterations. For the purposes of this project there is a flag to designate which convergence to use (an implementation of the spectral radius, L1 norm, and absolute difference are all provided), otherwise it will default to the spectral radius.

Another note about the algorithm is convergence is not guaranteed unless the matrix is strictly diagonally dominant. For a matrix to be strictly diagonally dominant it must satisfy the following property:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

As a result, I have ensured this is always the case in the randomized testing by setting $a_{ii} = \sum |a_{ij}| + \epsilon$ where ϵ is some random integer between 1 and 10.

2 Executing the Benchmarks

To execute the benchmark program, I've provided a simply python script that can be run simply by 'python benchmark.py' in the benchmark directory. Alternatively, to post a job on the Peanut Cluster, this can be done by running 'sbatch benchmark.sh' in the benchmark directory. My final submission has set the python benchmarking file to run with a matrix size of 1024, and rows per task (for work-steal) of 1, 2, 3, and 4. These can easily be altered to achieve the additional speedup graphs I've provided in the results section. To run the Go benchmarking file there are six different flags that can be specified for the: size of the matrix, rows per task in the work-steal implementation, number of threads to use, the desired convergence, the max number of iterations, and the mode (sequential, bsp, or worksteal). The program will then print out the result vector as well as the final time in seconds it took to run.

3 Bulk Synchronous Parallel Implementation

In Bulk Synchronous Parallel (BSP), there are some number of workers operating independently on a subset of the data, and when a worker reaches the end of the stage, it must wait for the rest of the workers to finish their respective stages so they can synchronize their results and move on to the next stage.

The Jacobi Iterative Method requires a number of iterations to be run and stopped either when the desired convergence has been achieved, or the max number of iterations has been reached. As a result, each iteration of the method spawns the required number of go routines to process that iteration. For example, on the first iteration, two go routines are spawned, and we wait until they finish to calculate the convergence from the previous iteration. On the next iteration, two new go routines are spawned and the process is repeated.

BSP is an ideal choice for running the Jacobi Method because it introduces little overhead to the system. There is no load balancing to consider with this type of problem because all threads are mostly doing the same amount of work (assuming a mostly uniform matrix and vector, i.e. not a sparse matrix). Also as a result of the even distribution of work, BSP allows for a higher throughput since all threads are fully utilized. Latency will be relatively low for each iteration, but will be increased slightly for barrier synchronization, though this is a predictable outcome. The synchronization among threads is the one area that can be improved in the BSP implementation. Because this is simply a calculation of convergence between two vectors, this can also technically be parallelized. This is the exact kind of problem that would easily map to a GPU and be performed in nearly constant time, but the overhead of calling the GPU may result in increased runtime.

4 Work Steal Implementation

In the work-steal implementation, there is a double-ended lock-free queue implemented with a circular array per the textbook. It allows for getting and setting (putting), pushing/popping an item from the bottom, and stealing an item from the top. These operations are done with the compare and swap mechanism as this is essentially the alternative to a locking mechanism.

The actual work-steal algorithm consists of a stealer (worker, but we already used this function name in the BSP implementation), and the work-steal main function. The stealer will either pop an item from the bottom if it is able, or if the pop bottom function returns false, it will attempt to steal an item from the top. When it successfully steals an item from either end of the queue, it will assign the item to a task type and perform the slice operation defined in the BSP code. The main work-steal function is similar in structure to the main BSP function. It essentially adds items to the queue, and spawns go routines that call the stealer function where the work is performed.

If there were load-balancing in the Jacobi Method, the work-steal implementation would correct for it, however, there is no load balancing, so the queueing system only adds more overhead. As a result of all of the threads checking for work unnecessarily, the latency is increased. The throughput of the work-steal implementation can be increased by making it more coarse-grained. This is shown in the results section below where as the number of rows processed per thread increases, the performance of the work-steal implementation also increases. The effects do become marginal as the number of rows per thread increases, however.

5 Results

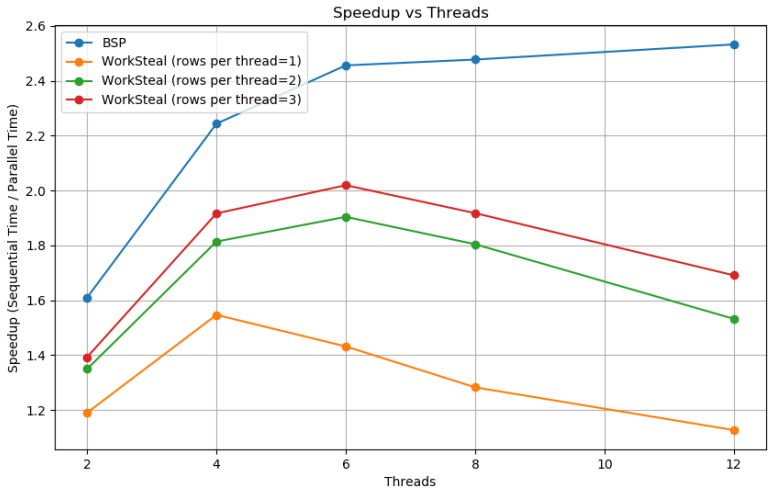
In all three charts, the BSP algorithm was superior to the work-stealing implementation no matter the matrix size of the problem, and no matter the number of rows per thread used to effectively tile the matrix. This is likely because the work-steal implementation added unnecessary overhead to the problem. A work-steal implementation is best used for dynamic problems that are more random in nature. The Jacobi Method, on the other hand, is constantly updating the same values repeatedly. BSP has less overhead and statically divides the work between threads without calls to steal work from another thread. Because all threads are more or less doing the exact same amount of work in matrix multiplication, there is no need for the additional overhead calls to steal work from another thread.

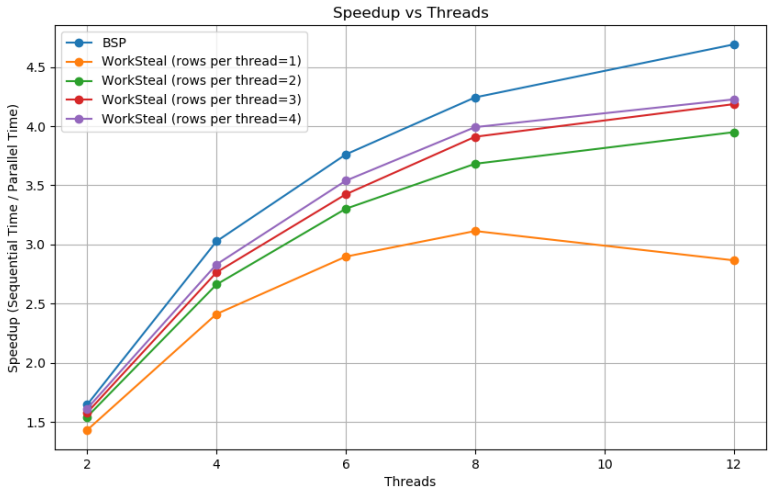
The graphs also display good weak scaling; as the problem size increases, the parallel implementations perform better and better in relation to the sequential implementation. When using a matrix of size 512, the max speedup for the BSP is about 2.5x, and for work-steal about 2x. When using a matrix of size 2048, however, the max speedup for BSP is over 5x, and for work-steal just over 4.5x. BSP does not appear to have hit its plateau yet in the case of a matrix with

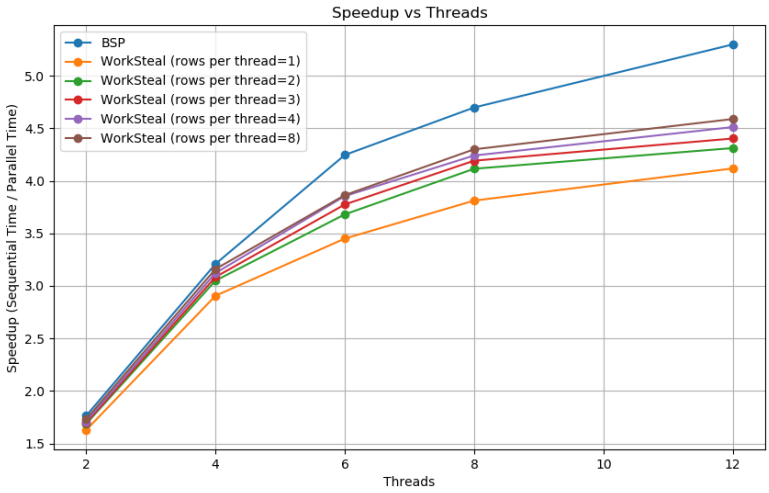
size 2048 in terms of the number of threads. This implies that a larger number of threads for the larger problem size would still increase the speedup relative to parallel. The fact that the speedup increases as the problem size increases is due to the fraction of the program that is spent in the parallel processing stage increasing, thus resulting in a greater speedup over the sequential implementation. The third graph adds an additional number of rows per thread line to demonstrate the marginal impact the rows per thread will have beyond a certain point for the work-steal implementation even for larger problem sizes.

One area for improvement would be the convergence calculation. This can technically be parallelized as well, but it serves as a barrier calculation to synchronize all of the threads after completing one iteration. This is the only area that is not parallelized in my code. Other hotspots, however, would be mostly in relation to memory bandwidth, and cache performance. There are a lot more reads and writes in the Jacobi Method than there are calculations, so as the number of threads increases, the speedup will plateau as a result of memory bandwidth being saturated. Particularly in relation to the work-steal implementation, the cache performance will also degrade as stealers are constantly attempting to find new work to do.

The main challenge I faced doing this assignment was deciding how to slice the matrix into pieces. While there exists a tiling method for this that is more efficient, I decided to simply slice the matrices by rows so that I could clearly demonstrate a difference in the granularity of tasks for the work-steal implementation. This was a focus of mine when thinking about how to implement the work-steal algorithm because I had originally assumed that a more fine-grained implementation would perform better, but was surprised at first to see the opposite trend. Ultimately, it makes sense that a more coarse-grained approach would work better for the work-steal implementation because it reduces the cache coherency issues and the overhead costs since there are less tasks to worry about.


Speedup vs Threads
BSP
WorkSteal (rows per thread=1)
WorkSteal (rows per thread=2)
WorkSteal (rows per thread=3)
Speedup (Sequential Time / Parallel Time)
Threads
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
2
4
6
8
10
12


Speedup vs Threads
BSP
WorkSteal (rows per thread=1)
WorkSteal (rows per thread=2)
WorkSteal (rows per thread=3)
WorkSteal (rows per thread=4)
Speedup (Sequential Time / Parallel Time)
Threads
4.5
4.0
3.5
3.0
2.5
2.0
1.5
2
4
6
8
10
12



6 Sources

- Jacobi Method
- Random Arrays in Go
- Flags in Go
- Art of Multiprocessor Programming (387-388)