Programming Assignment Report 2

Jacobian Iteration

Premise:

The Jacobian iteration is a technique that solves linear combinations of square matrices (number of unknowns = number of equations). The following formula takes Ax = B and solves for x through repeated iterations till x converges.

Approach:

The approach is to implement a serial version of the algorithm first. Then make additions to it such that an openMP version(s) can be created. Finally, by varying iterations, vector size and threads, we can measure metrics such as speedup and efficiency of the openMP algorithm(s). The following is the train of thought when adapting the serial version of the algorithm for openMP versions.

Setup:

1. When initializing mock matric A and vector B, for loops are used which can be parallelized using a "omp parallel for" command.
2. The time it took to setup the matrices for each method is documented but not used as a performance metric since this is specifically optimizing the iteration algorithm.

OpenMP (Reduction)

1. Add a "omp parallel for reduction (+:summation)" command to the inner for loop such that summation can be aggregated when running in parallel.
2. Optimize other for loops in the algorithm (swap) although it did not interfere with the execution times in the long run

OpenMP (Collapse)

1. Create a "perfectly nested" for loop such that the "omp parallel for collapse(2)" command can be used
    a. This means everything that is being executed must be inside the inner for loop
    b. The formula was adjusted to make this possible, but some time was added to setup time which is negligible.
2. Optimize other for loops in the algorithm (swap) although it did not interfere with the execution times in the long run and did slow down (so they were removed).

Insight Gained and Problems:

The biggest problem was adjusting the element based Jacobean formula such that everything was done in the inner for loop for collapse to work. The initialize Jacobean formula was the following:

$$X_{new\ i} = 1/a_{i,i}(b_i - \text{summation}(a_{i,j} * x_{old\ i} \text{ where } i \neq j \text{ and } i = 1....n))$$

To make way for the collapse command, the summation, difference and multiplication should be done in one loop. Therefore, the $1/a_{ii}$ was distributed and the temporary holder for x_{new} was initialized with b_i. The difference was distributed into the summation as such:

$$X_{new\ i} = b_i/a_{i,i} - 1/a_{i,i} * a_{i,j} * x_{old,i} - \ldots - 1/a_{i+(n-1),i+(n-1)} * a_{i+(n-1),j} * x_{old\ i+(n-1)}$$

$$\text{where } i \neq j \text{ and } i = 1....n$$

Another problem was the incorrect results were being returned when trying to initially parallelize the serial algorithm since private and shared data was being messed with. This issue was solved through intensive debugging and insight into the openMP library was gained to work on this task.

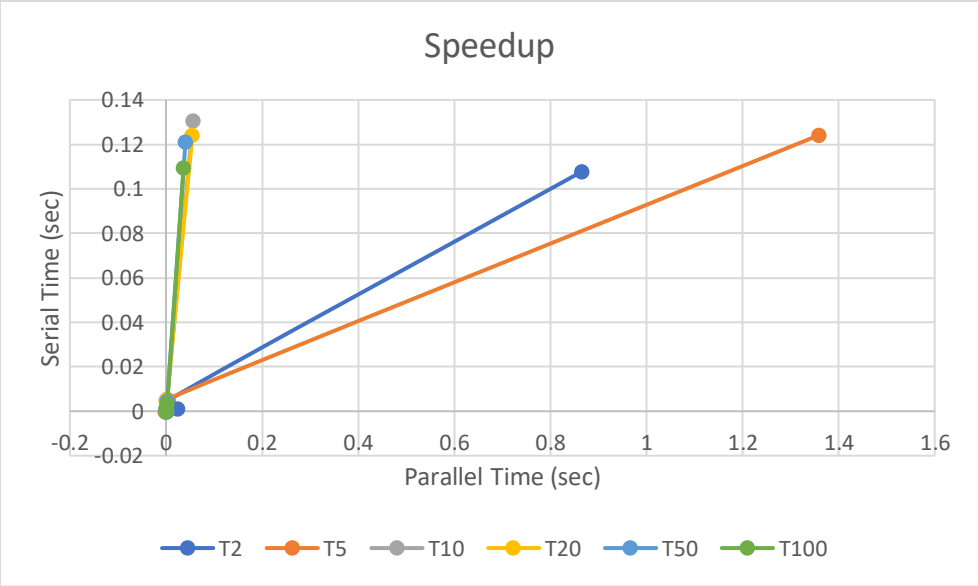
Optimizations:

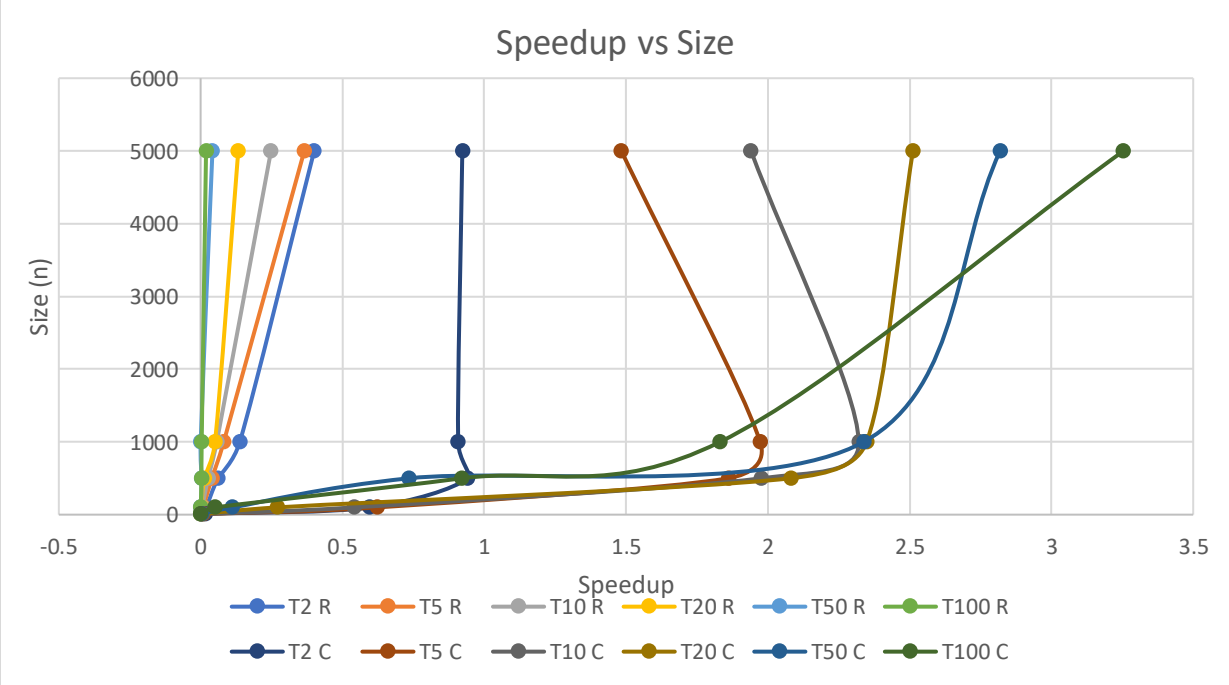
1. As mentioned before, the reduction and collapse(2) commands were added separately to create two different algorithms for better performance than the serial algorithm. Two metrics were measured to ensure the optimizations were successful and see which the better points at specific points was. The two metrics used was speedup and efficiency.

2. To also ensure the correct results were being calculated, a test 3x3 matrix (and vector) was introduced. Serial, openMPreduce and openMPcollapse were tested on this matrix to see matching results.

Performance Results & Analysis:

- Speedup = $T_{serial}/T_{parallel}$

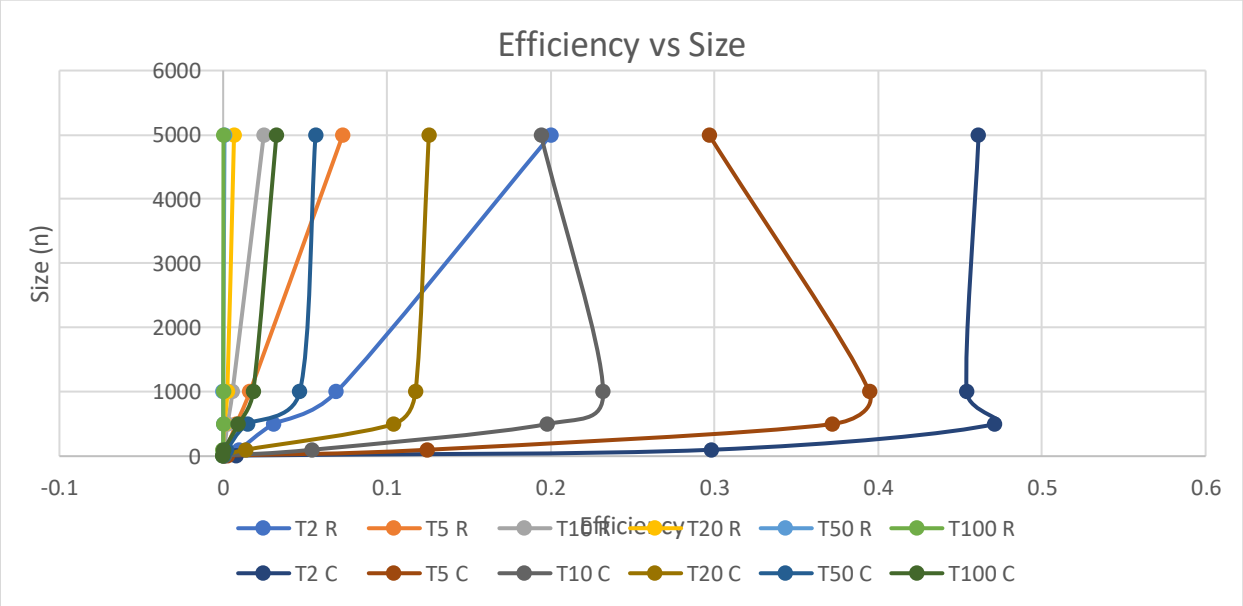




The above graph shows the speedup metric being measured over different matrix sizes and number of threads. The last independent variable in the formula was iterations but further tests suggested number of iterations had no impact on speedup. Thus, this variable was left alone. Different matrix sizes such as 10, 100, 500, 1000 and 5000 were used. Different number of threads such as 2, 5, 10, 20, 50 and 100 were used. In the legend of the graph above, the R signifies the reduction algorithm and C signifies the collapse algorithm. As seen, the collapse

algorithm did better in terms of speedup than the reduction algorithm. The 1 on the x-axis signifies the point which the parallel implementation is faster than the serial algorithm. As seen, as the size of the matrix increases and the number of threads increase, the higher the speedup. At around 5 threads and n between 500 and 1000, the parallel implementation overtakes the serial implementation (only collapse). Furthermore, there seems to be an exponential relationship where the size of matrix starts to matter less as the number of threads increase.

- Efficiency = Speedup/#of threads



The graph above shows the relationship between the different sizes of the square matrix against the efficiency of the parallel algorithm. As thread sizes become greater than the matrix sizes, the parallel implementation becomes more efficient. Once again, the parallel algorithms are exponential as the matrix size overtakes the thread count.

All in all, both the reduction and collapse implementations have an exponential growth with respect to the matrix size and speedup. The collapse implementation is far superior to the reduction algorithm in terms of speedup and efficiency. Finally, the number of iterations has no impact on speedup and efficiency.

Output:

Running the Program: n(size) and iter(iterations)

```
$ gcc -o ans -fopenmp jacobian.c

Jithin@DESKTOP-C1KE8PM ~/OneDrive - hawk.iit.edu/Lenovo/School/2019 Fall/CS 546/Programming Assignment/2

$ export OMP_NUM_THREADS=100

Jithin@DESKTOP-C1KE8PM ~/OneDrive - hawk.iit.edu/Lenovo/School/2019 Fall/CS 546/Programming Assignment/2

$ ./ans 5000 2
```

Sample Ouptput:

```
Creating Matrix & Vector via Serial...

Do you want to print A matrix and b vector? (Y/...): N
Skipping printing...

Running Serial Jacobian...

Do you want to print the resulting x values? (Y/...): N
Skipping printing...
Freeing Matrix...

Creating Matrix & Vector via OpenMP...

Running Reduction OpenMP Jacobian...

Do you want to print the resulting x values? (Y/...): N
Skipping printing...
Running Collapse OpenMP Jacobian...

Do you want to print the resulting x values? (Y/...): N
Skipping printing...

-----------------------------------------

RESULTS:

2 iterations & 5000 vector size

Setup Time:

Serial Jacobian: 0.119476 seconds
OpenMP Reduction Jacobian: 0.128068 seconds
OpenMP Collapse Jacobian: 0.127997 seconds

Calculation Time:

Serial Jacobian: 0.244104 seconds
OpenMP Reduction Jacobian: 12.277022 seconds
OpenMP Collapse Jacobian: 0.064941 seconds

SpeedUp:

OpenMP Reduction Jacobian: 0.019883 seconds
OpenMP Collapse Jacobian: 3.758858 seconds

Efficiency:

OpenMP Reduction Jacobian: 0.000199 seconds
OpenMP Collapse Jacobian: 0.037589 seconds
```

Testing of known coverging matrix:

```
Jithin@DESKTOP-C1KE8PM ~/OneDrive - hawk.iit.edu/Lenovo/School/2019 Fall/CS 546/Programming Assignment/2

$ gcc -o ans -fopenmp jacobian.c

Jithin@DESKTOP-C1KE8PM ~/OneDrive - hawk.iit.edu/Lenovo/School/2019 Fall/CS 546/Programming Assignment/2

$ ./ans 5000 2
Do you want to use converging 3x3 matrix and vecor (Y/...): Y
Do you want to print A matrix and b vector? (Y/...): Y

Printing Matrix A:
[ 6.000000, 2.000000, 1.000000]
[ 4.000000, 10.000000, 2.000000]
[ 3.000000, 4.000000, 14.000000]

Printing Vector b:
[3.000000, 4.000000, 2.000000]

Running Serial Jacobian...

Do you want to print the resulting x values? (Y/...): Y

x[0] = 0.342857
x[1] = 0.171429
x[2] = -0.078571

Running Reduction OpenMP Jacobian...

Do you want to print the resulting x values? (Y/...): Y

x[0] = 0.342857
x[1] = 0.171429
x[2] = -0.078571

Running Collapse OpenMP Jacobian...

Do you want to print the resulting x values? (Y/...): Y

x[0] = 0.342857
x[1] = 0.171429
x[2] = -0.078571


----------------------------------------
```

Attachment:

jacobian.c	The c file with all three implementations - Two integer command line inputs are needed: n & iteration - Otherwise defaults to n = 5 & iterations =100 - Test matrix is added and can be used when running the program (Will be asked)
data.csv	The excel file with the data used for the plots