Open MP Coursework Report

Author: ILKER OZTURK

Date: 15 December 2021

Authorisation: Module Leader

Distribution: All students

School of Computing and Mathematical Sciences



Table of Contents

[1. Comparing of serial Jacobi and Gauss solutions with using different levels of compiler optimization. 4](#_Toc90500093)

[2.Parallel versions of Jacobi and Gauss 10](#_Toc90500094)

[3.Optimizing the parallel code with compile optimization and comparing it with serial code. 18](#_Toc90500097)

[4.Conclusion 21](#_Toc90500098)

Table of figures

[Figure 1: setting the boundaries 4](#_Toc90499946)

[Figure 2:start time of serial 4](#_Toc90499947)

[Figure 3: End time and Calculation 4](#_Toc90499948)

[Figure 4:No Optimization 20x20 problem size Jacobi 5](https://uogcloud-my.sharepoint.com/personal/io6627a_gre_ac_uk/Documents/Desktop/comp1680/Parallelisation_report.docx#_Toc90499949)

[Figure 5:o1\_optimization 20x20 Jacobi 6](#_Toc90499950)

[Figure 6:o2\_Optimization 20x20 Jacobi 6](#_Toc90499951)

[Figure 7:o3\_Optimization 20x20 Jacobi 7](#_Toc90499952)

[Figure 8:-og\_Optimization\_Jacobi 7](#_Toc90499953)

[Figure 9:No optimization 20x20 Gauss 8](#_Toc90499954)

[Figure 10:o1\_optimization 20x20 Gauss 8](#_Toc90499955)

[Figure 11:-o2 optimization 20x20 gauss 9](#_Toc90499956)

[Figure 12:jacobi1thread 13](#_Toc90499957)

[Figure 13:jacobi\_paralel code 20x20 2 thread 14](#_Toc90499958)

[Figure 14:jacobi4thread 14](#_Toc90499959)

[Figure 15jacobi8threads 15](#_Toc90499960)

[Figure 16:Gauss1thread 15](#_Toc90499961)

[Figure 17:gauss2threads 16](#_Toc90499962)

[Figure 18:Gauss4threads 16](#_Toc90499963)

[Figure 19:gauss8threads 17](#_Toc90499964)

[Figure 20:removed while part 17](#_Toc90499965)

[Figure 21:o1 optimization Table 19](#_Toc90499966)

[Figure 22:o2 Optimization table 20](#_Toc90499967)

[Figure 23:o3 optimization table 20](#_Toc90499968)

[Figure 24:ofast optimization table 21](#_Toc90499969)

# Comparing of serial Jacobi and Gauss solutions with using different levels of compiler optimization.

In this section, a performance comparison is made by adapting the limits specified in the coursework to the Jacobi and Gauss serial codes, applying them to the code, and calculating the elapsed time using the gettimeofday() method.

Text

Description automatically generated

Figure 1: setting the boundaries

Boundary conditions set at top 100, bottom 20 , left 30 and right 40 according to requirements in task1.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 2:start time of serial

I also took the start time of the code to calculate the elapsed time.

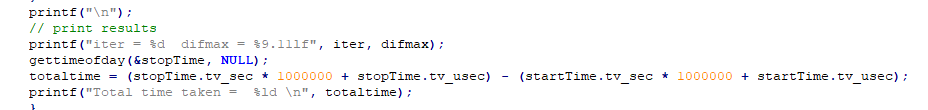


Figure 3: End time and Calculation

In addition, the piece of code in which I obtained the calculation of the elapsed time with these two values by taking the end time with the stopTime variable is shown above.

Then I compared the two methods with problem sets of 10x10 and 20x20 sizes using the optimization levels o0 or -o, -o1,-o2,o3,-ofast.

In addition, the screenshots showing that there is no code with a problem size of 20x20 are shared below.

A picture containing text

Description automatically generatedWith no optimization, iteration number is 718 and total time is 9637.

Figure 4:No Optimization 20x20 problem size Jacobi

A picture containing background pattern

Description automatically generated

Figure 5:o1\_optimization 20x20 Jacobi

With -o1 optimization parameter, iteration number is 718, total time is 8836

Background pattern

Description automatically generated with low confidence

Figure 6:o2\_Optimization 20x20 Jacobi

With -o2 optimization parameter, iteration number is 718, total time is 8827

A picture containing background pattern

Description automatically generated

Figure 7:o3\_Optimization 20x20 Jacobi

Background pattern

Description automatically generated

Figure 8:-og\_Optimization\_Jacobi

I also applied the same process for the Gauss method. Screen straps are shared below to show that the code is not corrupted as the optimization increases in 20x20 problem size for Gauss.

Background pattern

Description automatically generated

Figure 9:No optimization 20x20 Gauss

With no optimization parameter, iteration number is 387, total time is 5518

A picture containing background pattern

Description automatically generated

Figure 10:o1\_optimization 20x20 Gauss

With -o1 optimization parameter, iteration number is 387, total time is 2606

Background pattern

Description automatically generated

Figure 11:-o2 optimization 20x20 gauss

With -o2 optimization iteration number is 387, total time is 2616

Background pattern

Description automatically generated

With -o3 optimization iteration number is 387, total time is 2602

For the convenience of following these values, the table below has been created.

Method Size opt\_parameter Iteration No Time

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Jacobi | 10x10 | -o | 227 | 976 |
| Gauss | 10x10 | -o | 121 | 612 |
| Jacobi | 20x20 | -o | 718 | 9637 |
| Gauss | 20x20 | -o | 387 | 5518 |
| Jacobi | 30x30 | -o | 1417 | 38983 |
| Gauss | 30x30 | -o | 770 | 23021 |
| Jacobi | 10x10 | -o1 | 227 | 907 |
| Gauss | 10x10 | -o1 | 121 | 589 |
| Jacobi | 20x20 | -o1 | 718 | 8836 |
| Gauss | 20x20 | -o1 | 387 | 5518 |
| Jacobi | 30x30 | -o1 | 1417 | 37735 |
| Gauss | 30x30 | -o1 | 770 | 20350 |
| Jacobi | 10x10 | -o2 | 227 | 896 |
| Gauss | 10x10 | -o2 | 121 | 585 |
| Jacobi | 20x20 | -o2 | 718 | 8827 |
| Gauss | 20x20 | -o2 | 387 | 2606 |
| Jacobi | 30x30 | -o2 | 1417 | 36986 |
| Gauss | 30x30 | -o2 | 770 | 20350 |
| Jacobi | 10x10 | -o3 | 227 | 892 |
| Gauss | 10x10 | -o3 | 121 | 566 |
| Jacobi | 20x20 | -o3 | 718 | 8707 |
| Gauss | 20x20 | -o3 | 387 | 2602 |
| Jacobi | 30x30 | -o3 | 1417 | 36986 |
| Gauss | 30x30 | -ofast | 770 | 20102 |
| Jacobi | 30x30 | -o3 | 1417 | 30435 |
| Gauss | 30x30 | -ofast | 770 | 19028 |

When I examine the figures in the table in detail, it is observed that the most effective optimization is -ofast, but it will cause corruption in the code in large-size problems. In general, I observe a steady increase in performance from non-optimization to -o3, but these improvements are theoretically expected to increase as the problem gets bigger. Since I have observed that both 100x100 size and -ofast is broken when I tries first one and after that it started to work, the point to be considered here is to determine that the optimization does not affect the accuracy of the result.

## 2.Parallel versions of Jacobi and Gauss

In this part, the serial gaussian and Jacobi codes were parallelized, and their performances were observed.

In this section, I made some changes on the serial Jacobi code.

Text

Description automatically generated with medium confidence

I parallelised the first initialization array as default shared method which equally divides the workload to threads. OMP supports inner loops and this inner loop is not corrupting the OMP standards.

Text

Description automatically generated

Here, I added the critical keyword before the if condition to prevent each thread from updating the priv\_max value at the same time, because its simultaneous changes will cause the data to suffocate and the code to give a wrong output. Also, the omp\_get\_wtime result in the code was returning in seconds, I updated the code so that I could see it as microseconds.

Text, letter

Description automatically generated

In the section below, it is mentioned how the gaussian code is parallelized.

Text

Description automatically generated

# In the same way as in Jacobi, parallelization was made in the assign part. I used static and default as shared among the threads and I think it is more correct to do so, because it seems more logical for our algorithm to distribute the task of threads equally.

Text

Description automatically generated

Similarly, to Jacobi, I added the critical keyword before the if condition to prevent each thread from updating the priv\_max value at the same time, because its simultaneous changes will cause the data to suffocate and the code to give a wrong output. Also, the omp\_get\_wtime result in the code was returning in seconds, I updated the code so that I could see it as microseconds like I did in the Jacobi. When I check the outputs of the code, I observe that there is no data loss in any way. For this reason, it is observed that the code is successfully parallelized, although it may not be at the most perfect level.

Below is a screenshot showing the 20x20 size of the Jacobi code working without corruption for 1,2,4,8 threads..

Text

Description automatically generated with low confidence

Figure 12:jacobi1thread

# A picture containing background pattern Description automatically generated

Figure 13:jacobi\_paralel code 20x20 2 thread

Text

Description automatically generated with medium confidence

Figure 14:jacobi4thread

Text

Description automatically generated

Figure 15jacobi8threads

Below is a screenshot showing the 20x20 size of the Gauss code working without corruption.

Background pattern

Description automatically generated

Figure 16:Gauss1thread

Background pattern

Description automatically generated

Figure 17:gauss2threads

A picture containing text

Description automatically generated

Figure 18:Gauss4threads

Text

Description automatically generated

Figure 19:gauss8threads

|  |  |
| --- | --- |
| Method | Time(ms) |
| Gauss 1 thread | 587 |
| Gauss 2 threads | 830 |
| Gauss 4 threads | 1387 |
| Gauss 8 threads | 180506 |
| Jacobi 1 thread | 704 |
| Jacobi 2 threads | 900 |
| Jacobi 4 threads | 1743 |
| Jacobi 8 threads | 200342 |

Performance is not the issue in that step, so I did not care about the environment and performance while compiling and executing.

I also removed the while loop in gauss code to just run the code in 1 iteration. This code version also added to code documents.

Background pattern

Description automatically generated with low confidence

Figure 20:removed while part

|  |  |
| --- | --- |
| Threads | Time(ms) |
| 1 Thread | 44 |
| 2 threads | 169 |

When running the gaussian code with only 1 iteration, it is observed that very few of the values are updated and it takes a very short time as it is indicated in the table, as it is completed with only one iteration. When run with 2 threads, it is easily understood that the table has slightly more appropriate values than a single thread, but it is not a correct table.

# 3.Optimizing the parallel code with compile optimization and comparing it with serial code.

First, the code will be run with the -o1,-o2,-o3 and -ofast compiler optimization parameters, and the result will be compared with the serial results.

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Optimization parameter | Thread number | Time(ms) |
|  |  |  |  |
| Serial Jacobi | -o1 | - | 1.314.450 |
| Parallel Jacobi | -o1 | 2 | 25.773 |
| Parallel Jacobi | -o1 | 4 | 23.539 |
| Parallel Jacobi  Serial Jacobi | -o1  -o2 | 8  - | 23.543  989.654 |
|  |  |  |  |
| Parallel Jacobi | -o2 | 2 | 19.983 |
| Parallel Jacobi  Serial Jacobi | -o2  -o3 | 4  - | 17.184  640.104 |
| Parallel Jacobi | -o3 | 2 | 22.959 |
| Parallel Jacobi | -o3 | 4 | 19.603 |
| Parallel Jacobi  Serial Jacobi  Parallel Jacobi  Parallel Jacobi  Parallel Jacobi  Serial Gauss  Parallel Gauss  Parallel Gauss  Parallel Gauss  Serial Gauss  Parallel Gauss  Parallel Gauss  Parallel Gauss  Serial Gauss  Parallel Gauss  Parallel Gauss  Parallel Gauss  Serial Gauss  Parallel Gauss  Parallel Gauss  Parallel Gauss | -o3  -ofast  -ofast  -ofast  -ofast  -o1  -o1  -o1  -o1  -o2  -o2  -o2  -o2  -o3  -o3  -o3  -o3  -ofast  -ofast  -ofast  -ofast | 8  -  2  4  8  -  2  4  8  -  2  4  8  -  2  4  8  -  2  4  8 | 17.023  400.234  21.743  18.895  16.341  819.125  24.683  22.582  20.384  695.043  23.544  22.072  19.981  620.764  23.502  18.043  15.965  384.983  22.984  17.692  13.459 |

Figure 21:o1 optimization Table

Figure 22:o2 Optimization table

Figure 23:o3 optimization table

Figure 24:ofast optimization table

The tables above are divided according to the optimization parameters and since the serial ones are single thread, the same value is entered in the three columns.

# 4.Conclusion

It is seen that the serial codes to be deduced from the above tests and the explanations made in the codes provide a lot of advantages in large data sets, and the optimization that can be made without causing the code to break down is very beneficial on performance, especially in for problems large data sets. I observe that the Gauss method is faster than Jacobi both in serial and parallel, because it can complete this process with fewer iterations. It is not the full gaussian method used here, the algorithm applied here is the adapted form of gaussian to jacobi and it also works more effectively. The conclusion to be drawn from here is that it is seen that the parallelization made by properly planning can have very large effects on the performance. In this coursework, it can be seen that serial code sets have been successfully converted to parallelization using OpenMP's basic parallelisation rules. It has been learned in detail by me that when faced with very large data in the future, these probes can solve much more effectively if their algorithms are suitable for this.