Analysis of the parallelization of solving systems of linear equations using the Gauss-Seidel method

Group Members: George Hope, Naima Ontiveros, Alicja Wagner, Songul Yasin

Task 1: Candidate program for parallelization

Problem chosen/researched: Gauss-Seidel Method

Task 2: Study of the problem prior to its parallelization

Description of the code:

The aim of the code is to check what percentage of the systems of linear equations which are randomly initialized can be solved using the Gauss-Seidel method. The program is based on https://www.sanfoundry.com/c-program-implement-gauss-seidel-method/. The code uses the element-based formula.

The main structures of the code are:

1. A matrices - coefficients of the linear equations
2. b vectors - the product of Ab (absolute terms of the equations)
3. x vectors and y vectors - solution of the calculations.

At the beginning A matrices and b vectors are randomly initialized. Moreover, some numbers are added to the main diagonals of the matrices to try to make them diagonally dominant, which is the convergence condition in the Gauss-Seidel algorithm.

Then each system of equations is solved. The program iterates until it reaches any of the stop conditions which are: achieving the desired precision, achieving the maximum number of iterations or getting a NaN (not a number).

At the end it is checked how many of the systems of equations have converged and the result is printed to the console.

Why it is a good candidate for parallelization:

Gauss-Seidel is a popular iterative method for solving systems of linear equations, particularly in the context of numerical analysis and scientific computing. There is known potential for effective parallelization due to certain properties inherent in the algorithm:

1. Sequential Nature: Gauss-Seidel is sequential in nature[1], where each iteration is based on the results of the previous iteration. However, within each iteration, there is a certain level of parallelism that can be exploited.
2. Local Updates: In each iteration of Gauss-Seidel, the solution for each variable is updated using the latest values of the other variables[2],[3]. This local update property allows the updates for each variable to be calculated in parallel.

3. Data Dependency: Although each iteration has data dependencies because the updates are based on the previous values, the updates for different variables within each iteration can be calculated independently[1], [2]. This property makes parallelization easier.
4. Convergence properties: Gauss-Seidel often converges faster than other iterative methods, potentially allowing for fewer iterations, which in turn reduces the synchronization overhead associated with parallelization[1], [4].

Gauss-Seidel parallelization requires attention to managing data dependencies, properly synchronizing updates, and choosing an efficient parallelization strategy based on available hardware (e.g., shared memory, distributed memory parallelism) in order to maximize performance.

Overall, Gauss-Seidel is a suitable candidate for parallelization due to its inherent sequential parallel structure and local update properties, enabling faster convergence and efficient utilization of modern parallel computing resources.

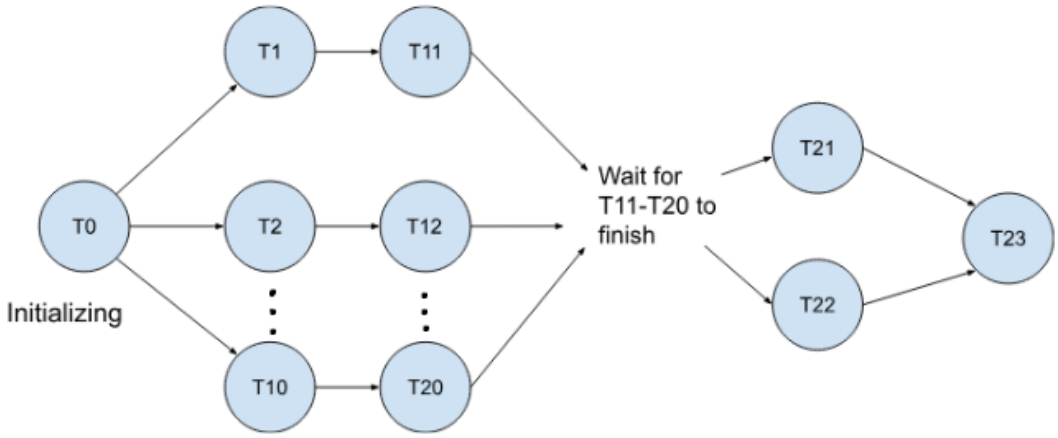
Task 2.1: Inter-Task Dependency Graph

Network of dependencies between tasks: T0 → T1 → T2 → ... Tn , where
T0: Initializing variables
T1-T10: Randomizing Matrices and Vectors
T11-T20: Solving Systems of Linear Equations
T21:Checking Convergence
T22: Counting Converged Tasks
T23: Printing the Counted Tasks

Main tasks/ dependencies:

1. Randomizing Matrices and Vectors:
    - randomize_mat(a[i]) and randomize_vec(b[i]) can be executed in parallel for different values of i because they operate on separate matrices and vectors.
2. Solving Systems of Linear Equations:
    - The loop for (i = 0; i < MAT_NUMBER; i++) iterates through different matrices and vectors, hence each iteration can be considered a separate task.
    - The function if_converges(a[i], b[i]) depends on the values of a[i] and b[i] calculated in the previous step. So, the tasks randomize_mat(a[i]) and randomize_vec(b[i]) need to be completed before if_converges(a[i], b[i]) is executed.

3. Checking Convergence:
    - The loop for (i = 0; i < MAT_NUMBER; i++) calculates result[i] and doesn't have any dependencies between iterations. This means the iterations of this loop can be executed in parallel.
4. Counting Converged Tasks :
    - After the loop for (i = 0; i < MAT_NUMBER; i++) has completed, the loop for (i = 0; i < MAT_NUMBER; i++) counts how many systems have converged. This loop depends on the previous loop's results.



The image above shows an Inter-Task dependency graph showing how tasks would be run in parallel for Gauss-Seidel.

Task 2.2: Variables

To study the read/write access to each variable in the provided C program and how these variables are used, we will break down the code into different parts:

One part is the global variables. In the program „MAX_ITER“, „MAT_SIZE“, „MAT_NUMBER“ and „EPS“ are global constants, which will be used to set various parameters for the program. These variables are read-only.
In the main-function we have a 3D array „double a[MAT_NUMBER][MAT_SIZE][MAT_SIZE]“ that stores matrices and a 2D array „double b[MAT_NUMBER][MAT_SIZE]“ that stores the vectors. The variables „int i“, „int result[MAT_NUMBER]“, „int converged“ and „time_r“ are used for loop counters and storing results. The variable srand((unsigned) time(&t)) is used to

initialize the random number generator with the current time. The „for“- loop randomizes the matrices and vectors stored in „a" and „b“. This involves both read and write access to these arrays. The next „for“-loop calls the „if_converges”-function and stores the results in the „result“ array. The following „for“-loops calculate the number of matrices that have converged and print the results. These loops primarily use read access to the „result“ array.
In the „if_converges“-function part we have the arrays „double x[MAT_SIZE]“ and „double y[MAT_SIZE]“ which are used for calculating and storing solution vectors. The values „double y_prev“ and „int iter“ are used to store. „int n“, „int i“, „int j“, „int flag“, and „int is_nan“ are used as loop counters and flags. And in the main loop, there is read and write access to the variables in this function, including „y“, „x“, and „is_nan“. The function prints the solution, which involves read access to „y“. Finally, the function returns `0` or `1` based on whether the algorithm converged or not.
The last part the„randomize_mat“ and „randomize_vec“ Functions take in arrays „a“ and „b“, respectively, and write random values into them. The variables within these functions are primarily write-only.

The program can potentially be structured for more efficient execution on a parallel machine by parallelizing the processing of different instances of the „if_converges“- function. For that, we could use multi-threading or other parallel programming techniques to distribute the workload across multiple threads.

Cache problems, such as cache thrashing, may arise due to the frequent access to elements of the arrays („a“, „b“, „x“ and „y“). The data access patterns in the nested loops can lead to cache misses, especially if the data exceeds the cache size. This can result in slower memory access times. To mitigate cache-related issues, you can consider optimizing the memory access patterns, using cache-aware data structures, and potentially using loop tiling or blocking techniques to improve data locality.

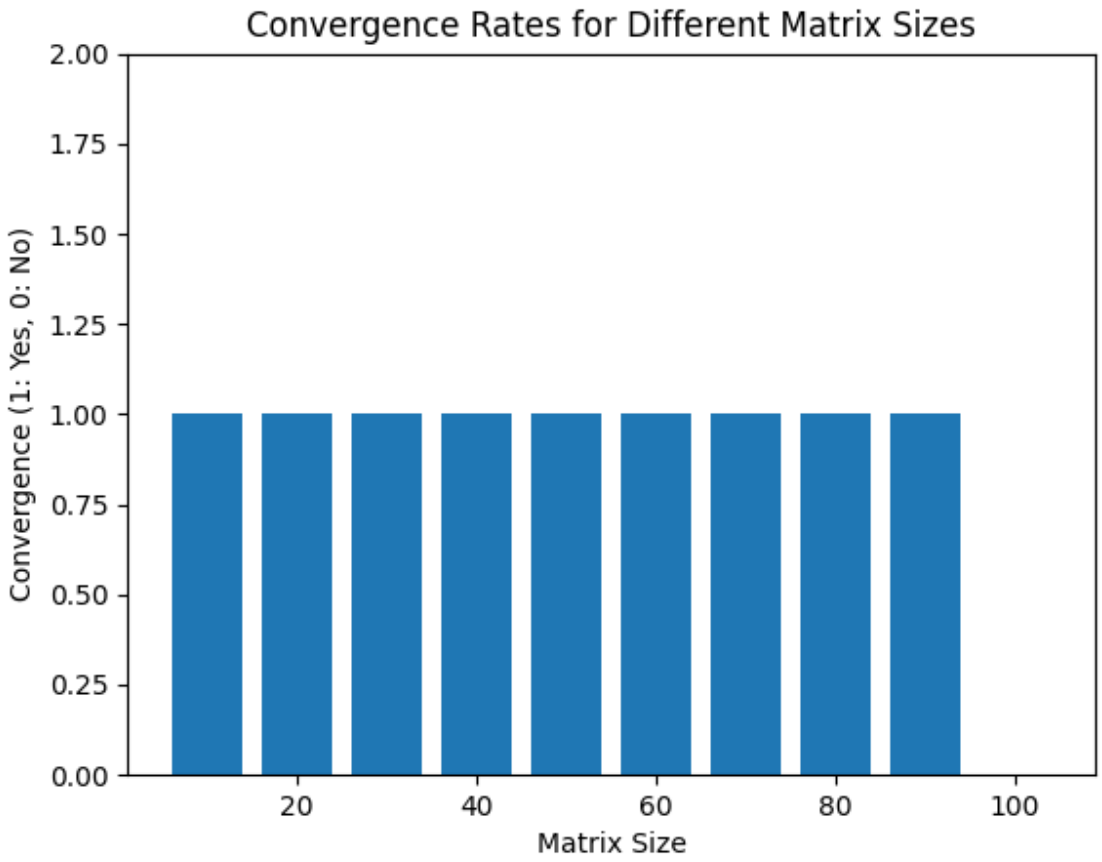
In summary, the potential for parallelism exists in processing different instances of linear equation solving, but the inner loop within „if_converges“ remains inherently sequential. Cache performance could be optimized by focusing on improving data locality and reducing cache misses.

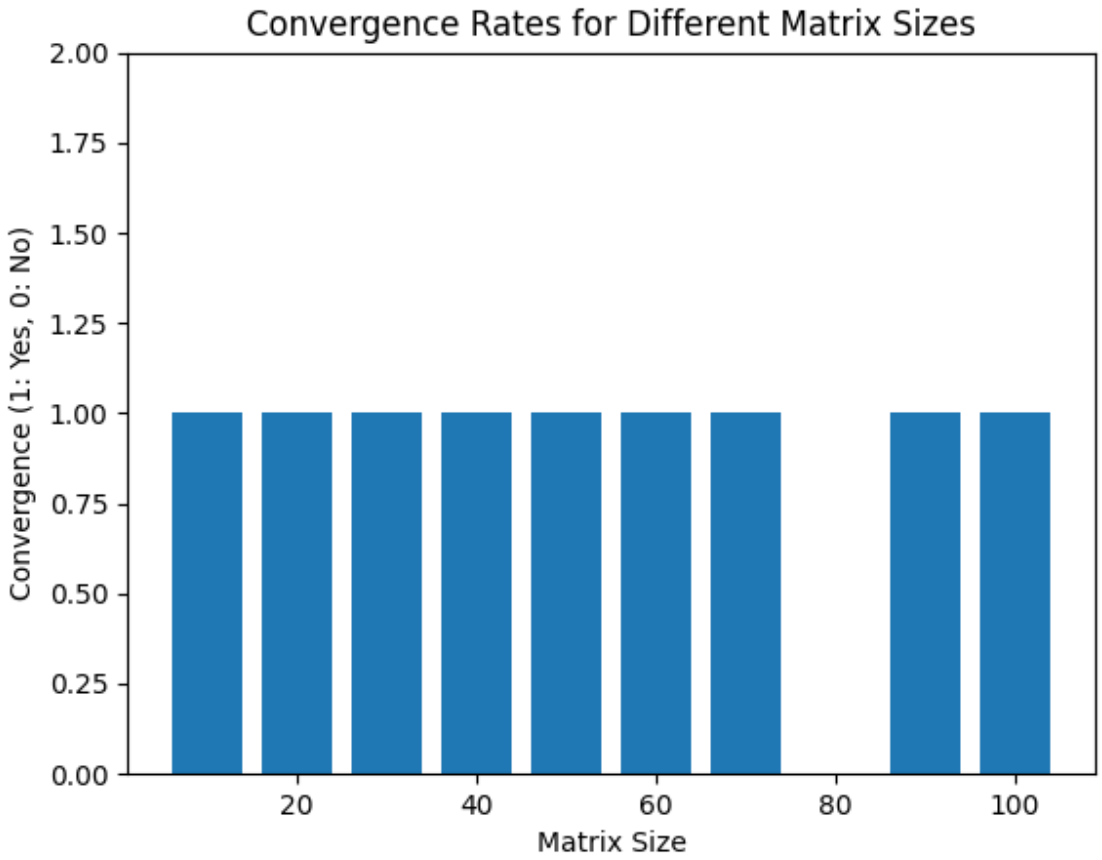
Task 2.3: Load variation study

Multiple factors can affect the performance of the problem, such as the matrix size (MAT_SIZE), number of matrices (MAT_NUMBER), max iterations (MAX_ITER), and epsilon (EPS). When increased or decreased, they can all potentially affect the computational load and overall performance of the program.

In this case, we tested how changing "MAT_SIZE" affects performance. Once the results were plotted, as the matrix size increased, the convergence rate was 1 throughout the x-axis. When it got to 100 matrices, there was no convergence.

The graph below shows the Efficiency vs Matrix Size. If the convergence rate is 1, then it will result in a solution of the matrix, otherwise, no matter how many iterations are done, there is no progress and the efficiency has decreased. We can see from the graph that at a size of 100, the Gauss-Seidel method is not efficient.





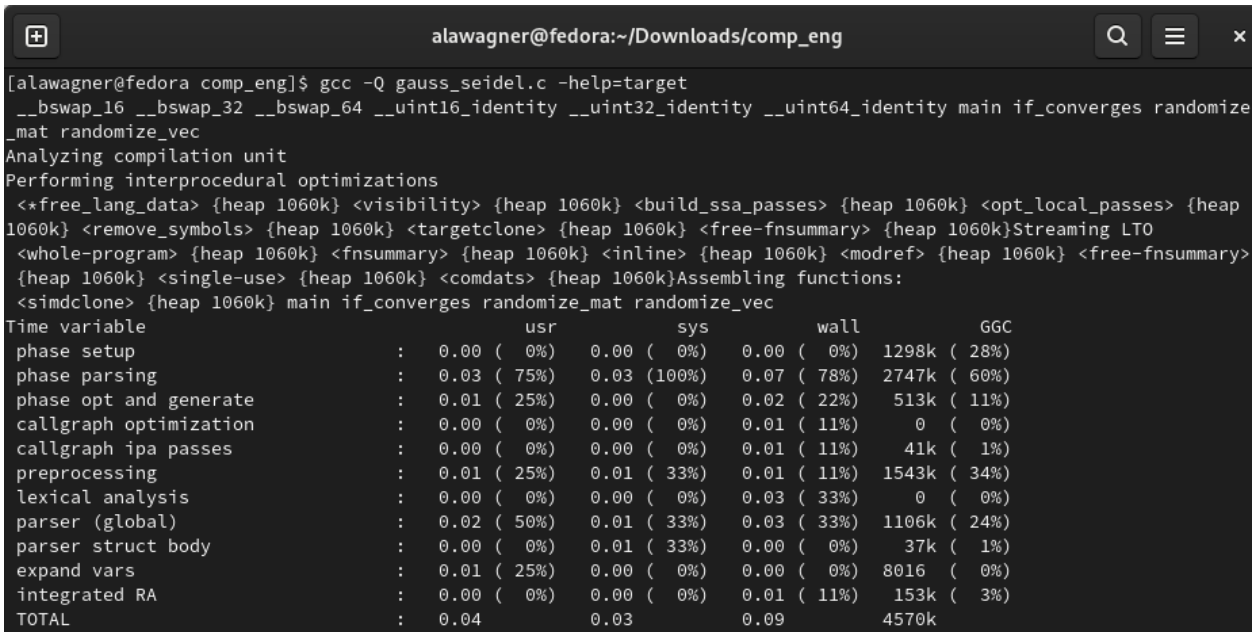
Task 2.4: "Tuning" of compilation parameters

Compiler options:

The GCC compiler can be run with various options that provide additional analysis and optimization information.

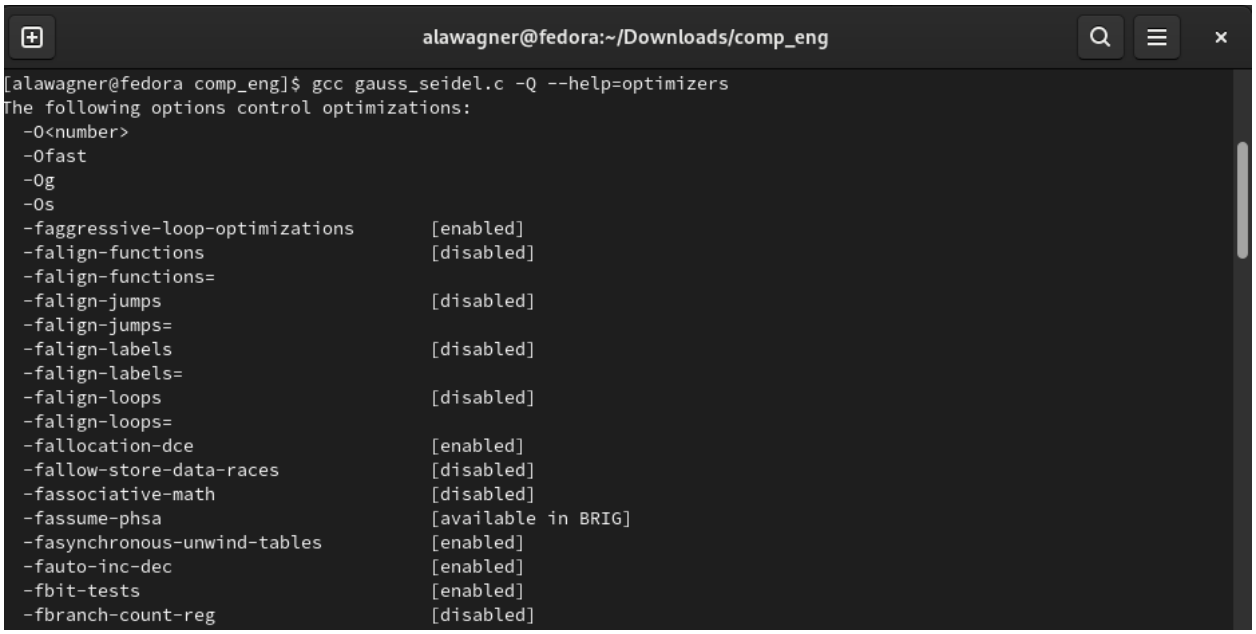
One of them is “gcc -Q -help=target”. When you use -Q followed by a specific argument like -help=target, GCC provides information related to the target architecture and code generation. Below there is an example of the output of that command which was obtained without any changes of the parameters and compilation options - the compiler itself decided which options and parameters to use.

The output is specific to GCC and provides detailed information about the various compilation phases and their respective timings. It's often used for diagnostic and profiling purposes to analyze how the compiler is processing your code and to identify potential areas for improvement in terms of compilation speed and efficiency.



These timings can help you understand which phases of the compilation process are taking the most time. In the above output, the “Parsing” and “Parser” phases appear to be the most time-consuming, followed by “Preprocessing” and “Opt and Generate.”

Another option is “gcc -Q --help=optimizers”. The last part of the command tells GCC to provide information about its optimization options. It will display a list of available optimization flags and indicate which ones are enabled. Below we can see the first lines of the output of this command, also executed without any changes of the parameters and compilation options.



Analyzing the outputs we can understand how to utilize compiler optimizations effectively for our specific code and target platform.

Tuning of compilation parameters:
Compiling the code using different parameters can significantly optimize the performance of an application. Our application includes randomization, making the execution time different each time. Therefore, the times shown below are averages from 100 executions of the program.

The basic time (without specifying any flags): **0,2507 seconds**.
Example: gcc gauss_seidel.c -O3 [-o linear_solver]

Parameter	**Description**	**Enhanced time [s]**
-ffast-math	Enables aggressive math optimizations, which can significantly improve the performance of mathematical operations. However, it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions.	0,2257
-funroll-loops	Unrolling loops can reduce loop overhead and improve performance.	0,2422
-finline-functions	Inlining functions can eliminate function call overhead, which is beneficial for small, frequently called functions.	0,2441
-march=native	Instructs the compiler to generate machine code specifically tailored to the native instruction set architecture of the host machine where the code will be executed.	0,2232
-ffp-contract=fast	Instructs the compiler to aggressively optimize floating-point expressions by contracting them. Floating-point contraction is a form of optimization that combines multiple floating-point operations into a single operation whenever possible.	0,2334
-ftree-vectorize	Enables loop vectorization, which is essential for improving performance in numerical code with loops.	0,2267
-O3	-O<number>: The -O option enables different levels of optimization, where <number> can be from 0 (no optimization) through 3 (highest optimization). With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. Typically, using -O3 provides significant performance improvements. It turns on several optimization flags.	0,0860

There are multiple flags that can optimize the performance of the program. In our case the most beneficial ones would be those that improve numerical computations and loop operations. Enabling single flags, however, does not give the best results. The best option is to use many of them at once, for example by applying the -O3 option.

Auto vectorization:

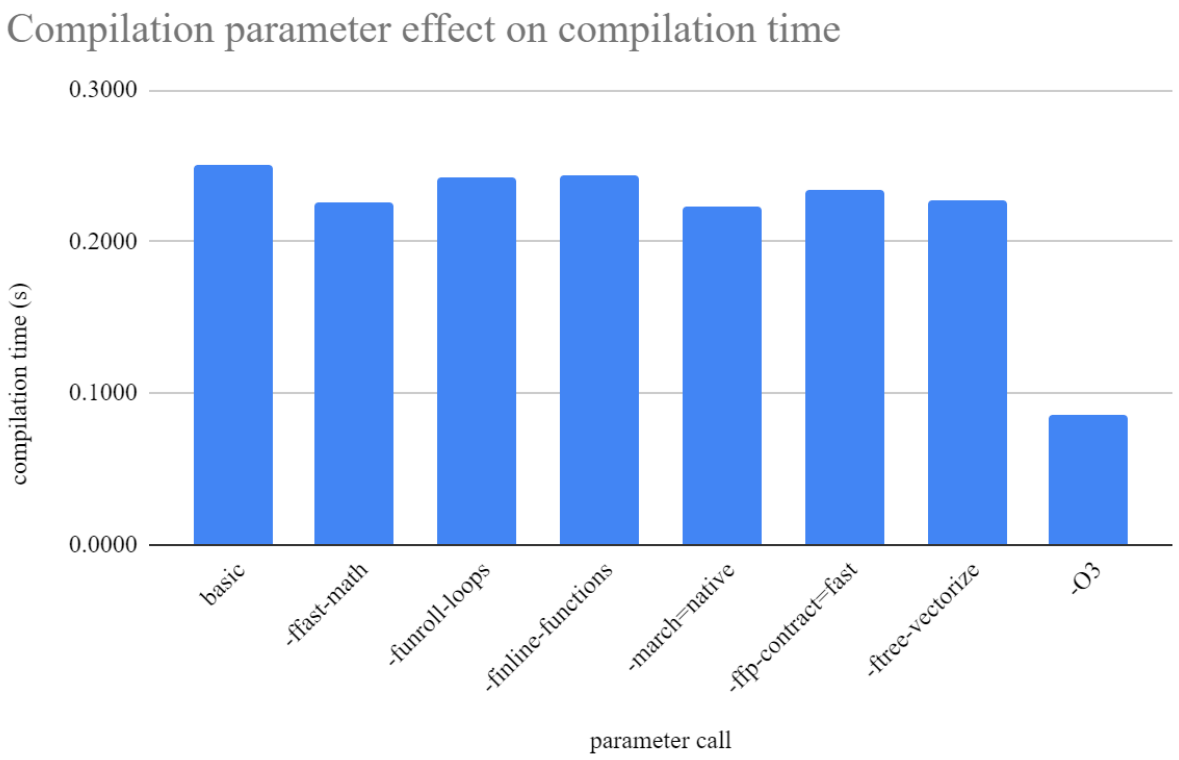
The command “objdump -j .text -D program.elf” is used to disassemble and display the content of the ".text" section of the "program.elf" binary file. The ".text" section typically contains the executable code of the program. By running this command, you can view the assembly language instructions that make up the program's binary executable, allowing you to inspect and analyze the low-level instructions that the program consists of.

By running “objdump -j .text -D linear_solver” we can see which assembly instructions are used in our code. The SIMD instructions often use XMM registers which allow simultaneous operations on multiple variables and are widely used in parallel computing. Examples of such operations which appear in our code are: movsd (Move or Merge Scalar Double Precision Floating-Point Value), divsd (Divide Scalar Double Precision Floating-Point Value), mulsd (Multiply Scalar Double Precision Floating-Point Value), ucomisd (Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS). Below there is a fragment of disassembled “if_converges” function which is responsible for solving a system of linear equations using Gauss-Seidel method.

```
226   401384:       45 31 c9                xor    %r9d,%r9d
227   401387:       45 31 e4                xor    %r12d,%r12d
228   40138a:       f2 0f 10 17             movsd  (%rdi),%xmm2
229   40138e:       f2 41 0f 10 0c d3       movsd  (%r11,%rdx,8),%xmm1
230   401394:       31 c0                   xor    %eax,%eax
231   401396:       f2 0f 10 19             movsd  (%rcx),%xmm3
232   40139a:       f2 0f 5e ca             divsd  %xmm2,%xmm1
233   40139e:       f2 0f 11 09             movsd  %xmm1,(%rcx)
234   4013a2:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
235   4013a8:       39 d0                   cmp    %edx,%eax
236   4013aa:       74 17                   je     4013c3 <if_converges+0x93>
237   4013ac:       f2 0f 10 04 c6          movsd  (%rsi,%rax,8),%xmm0
238   4013b1:       f2 0f 5e c2             divsd  %xmm2,%xmm0
239   4013b5:       f2 41 0f 59 04 c0       mulsd  (%r8,%rax,8),%xmm0
240   4013bb:       f2 0f 5c c8             subsd  %xmm0,%xmm1
241   4013bf:       f2 0f 11 09             movsd  %xmm1,(%rcx)
242   4013c3:       48 83 c0 01             add    $0x1,%rax
243   4013c7:       48 83 f8 64             cmp    $0x64,%rax
244   4013cb:       75 db                   jne    4013a8 <if_converges+0x78>
245   4013cd:       66 0f 28 c1             movapd %xmm1,%xmm0
246   4013d1:       66 0f 2e c9             ucomisd %xmm1,%xmm1
247   4013d5:       f2 41 0f 11 0c d0       movsd  %xmm1,(%r8,%rdx,8)
248   4013db:       f2 0f 5c c3             subsd  %xmm3,%xmm0
249   4013df:       45 0f 4a cd             cmovp  %r13d,%r9d
250   4013e3:       66 0f 2e c4             ucomisd %xmm4,%xmm0
251   4013e7:       45 0f 47 e5             cmova  %r13d,%r12d
252   4013eb:       48 83 c2 01             add    $0x1,%rdx
```

Task 2.5: Analyzing compiler parameters and options

As we can see from the first section, the tuning of certain compiler parameters can result in a more optimized compilation.



Most of the parameters didn't provide a significant time improvement, however, the -O3 call made a very significant difference. This is because the -O3 call allows the compiler to determine which flags to call, and it can call many at once, which is not the case with our other tests. It is mainly this ability, to combine the power of multiple calls, that allows the -O3 call to produce better results.

Task 2.6: Performance of the application

The followed the key parameters that can be varied to study their impact on application performance:

- Changing "MAT_SIZE" directly affects the complexity of the linear system to be solved. As the size increases, the number of calculations required increases.
- Changing "MAX_ITER" will affect the number of iterations the algorithm performs to converge and possibly also the execution time.
- Changing "MAT_NUMBER" affects the number of systems solved in each run.

By changing these parameters, differences in the acceleration of the application can be observed. Here is an analysis of how each parameter can affect performance:

- Increasing the "MAT_SIZE" results in more calculations per iteration, which results in longer calculation times.
  Reducing the "MAT_SIZE" reduces the computational load and can lead to faster execution times.
- Increasing "MAX_ITER" may result in longer execution times, especially if convergence is slow on certain systems.
  Reducing "MAX_ITER" can reduce overall execution time, but could affect accuracy if the convergence criterion is not met.
- A larger "MAT_NUMBER" results in more calculations and possibly longer execution times.
  A smaller "MAT_NUMBER" could speed up the application.

By changing these parameters individually and in combination, one can observe how execution times vary.

Task 4: big.LITTLE processor

Part 1:

To determine the parallelised time, we must examine the flow rates of the taps. Suppose we have a water tank of volume 100L. The first tap (T1) takes 4 hours to fill up the tank, therefore, its flow rate is 100L/4h = 25 L/h. The second tap (T2) takes 20 hours to fill up the tank, therefore, its flow rate is 100L/20h = 5 L/h. When the two taps work in parallel, their flow rates combine such that the parallelised flow rate is 25+5 = 30 L/h. At this flow rate, it will take the two taps (100L) / (30L/h) = 3.33 hours to fill the 100L tank when working together. Therefore, $t_p(2)$=3.33 hours

Because there are two different taps with different flow rates, we can answer this question using two different sequential times.

Assuming we are using T1 for the base sequential time:
$t_{seq}(T1) = 4$ hours
$t_p(2) = 3.33$ hours
$s_p(2) = t_{seq}(T1) / t_p(2) = 4/3.33 = 1.2$, which is less than p=2 so we are happy.
$\eta_p(2) = s_p(2) / p = 1.2/2 = 0.6$, i.e. an efficiency of 60%.

Assuming we are using T2 for the base sequential time:
$t_{seq}(T2) = 20$ hours
$t_p(2) = 3.33$ hours

$s_p(2) = t_{seq}(T2) / t_p(2) = 20/3.33 = 6$, which is more than p=2, which is technically a fault, but it is allowed in this instance because we are using two taps (analogous to processors) that have different speeds.
$\eta_p(2) = s_p(2) / p = 6/2 = 3$, i.e. an efficiency of 300%. This is obviously 'impossible', but again it is allowed in this instance due to the differing performances of the taps.

Part 2:
This question is much easier than the previous part because both taps have the same flow rate. Each tap takes 4 hours to fill the tank, hence, $t_{seq} = 4$ hours.
$t_p(2) = \max\{0.5*4, 0.5*4\} = 2$ hours
$s_p(2) = t_{seq} / t_p(2) = 4/2 = 2$, which is equal to p, indicating perfect parallelisation.
$\eta_p(2) = s_p(2) / p = 2/2 = 1$, i.e. 100% efficiency.

Part 3:
This question is very similar to the previous part as both taps also have the same flow rate. Each tap takes 20 hours to fill the tank, hence, $t_{seq} = 20$ hours.
$t_p(2) = \max\{0.5*20, 0.5*20\} = 10$ hours
$s_p(2) = t_{seq} / t_p(2) = 20/10 = 2$, which is equal to p, indicating perfect parallelisation.
$\eta_p(2) = s_p(2) / p = 2/2 = 1$, i.e. 100% efficiency.

Part 4:
To determine the parallelised time, we must examine the flow rates of the taps like we did in part 1. Suppose we have a water tank of volume 100L. The first tap (T1) takes 4 hours to fill up the tank, therefore, its flow rate is 100L/4h = 25 L/h. The second tap (T2) and third tap (T3) each take 20 hours to fill up the tank. Therefore, their flow rates are 100L/20h = 5 L/h. When the three taps work in parallel, their flow rates combine such that the parallelised flow rate is 25+5+5 = 35 L/h. At this flow rate, when working together, it will take the three taps (100L) / (35L/h) = 2.86 hours to fill the 100L tank. Therefore, $t_p(3)$=2.86 hours

Because there are three different taps with two different flow rates, we can answer this question using two different sequential times.

Assuming we are using T1 for the base sequential time:
$t_{seq}(T1) = 4$ hours
$t_p(3) = 2.86$ hours
$s_p(3) = t_{seq}(T1) / t_p(3) = 4/2.86 = 1.4$, which is less than p=3 so we are happy.
$\eta_p(3) = s_p(3) / p = 1.4/3 = 0.467$, i.e. an efficiency of 46.7%.

Assuming we are using either T2 or T3 for the base sequential time:
t_{seq}(T2 or T3) = 20 hours

$t_p(3)$ = 2.86 hours
$s_p(3) = t_{seq}$(T2 or T3) / $t_p(3)$ = 20/2.86 = 7, which is more than p=3, which is technically a fault, but it is allowed in this instance because we are using three taps that don't all have the same speeds.
$\eta_p(3) = s_p(3) / p = 7/3 = 2.33$, i.e. an efficiency of 233%. This is obviously 'impossible', but again, it is allowed in this instance due to the differing performances of the taps.

Part 5:
As technology develops, the user demand for greater device performance is rising much quicker than the rate of development of battery technology, yet users expect both performance and battery life to increase over time. To solve this issue, and to allow performance to increase while optimizing power use, the big.LITTLE processor was introduced. The big.LITTLE processor is a heterogenous combination of two processors, each with different strengths. The 'big' processor is a maximum performance processor, that is used only when necessary to tackle a task with more power. The 'LITTLE' processor is a less powerful processor designed to tackle less demanding tasks while optimising energy consumption. In this way, users can experience better maximum power output while not compromising the battery life too much. The previous parts of this task 4 present questions that deal with the 'paralellisation' of taps with different flow rates. This is analogous to the paralellisation of multiple processors that have different processing powers, ie. a heterogenous combination of processors.

Parallel architectures suitable for the execution:

To execute this program efficiently in parallel, you can consider using several parallel computing architectures:

- Multithreading (OpenMP): The "if_converges" function can be parallelized with OpenMP. OpenMP is a popular choice for loop parallelization and can distribute the workload across multiple CPU cores. pragmas allow you to specify which loops should be parallelized, and OpenMP takes care of thread management.
- CUDA (for GPU): With access to a GPU, the calculation can be shifted to the GPU using CUDA. CUDA allows you to parallelize the computation on the GPU, which can provide significant speedup for certain types of numerical problems[3].
- MPI (Message Passing Interface): With access to a cluster of machines, MPI can be used to parallelize the execution of the program across multiple nodes. MPI allows for distributed memory[4].
- Heterogeneous Computing (CPU + GPU); The combination of CPU and GPU computing can be used for hybrid parallelism. Multi-threading can be used for CPU computations and CUDA for GPU computations[3].

Bibliography

[1] Ahmadi, A. ; Manganiello, F. ; et al.: “A Parallel Jacobi-Embedded Gauss-Seidel Method”, IEEE
Transactions on Parallel and Distributed Systems, vol. 32 no. 6, pages 1452-1464, 2021

[2] Al-Towaiq, Mohammad H.: “Parallel Implementation of the Gauss-Seidel Algorithm on k-Ary n-Cube Machine“, Applied Mathematics, vol. 04 no. 01, pages 177-182, 2013

[3] Koester, D.P. ; Ranka, S. ; et al.: „A Parallel Gauss-Seidel Algorithm for Sparse Power System Matrices,“ Proceedings of Supercomputing, pages 184-193, 1994

[4]Zhang, Yi ; Parker, David ; et al.: „A Wavefront Parallelisation of CTMC Solution using MTBDDs“, IEEE Computer Society Press, 2005

[5] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[6] https://www.sanfoundry.com/c-program-implement-gauss-seidel-method/