**Parallelization of Gauss-Seidel**

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

**Task 1**:

Problem chosen/researched: Gauss-Seidel Method

**Task 2:**

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.

**Task 2.1: Inter-Task Dependency Graph**

Network of dependencies between tasks: T1 → T2 → T3 → ... Tn , where

T1: Randomizing Matrices and Vectors

T2: Solving Systems of Linear Equations

T3: Checking Convergence

T4: Counting Converged 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, so 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.

**Task 2.2**

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 and analyze each section individually:

One part is the global variables. In the Programm „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 the 3D array „double a[MAT\_NUMBER][MAT\_SIZE][MAT\_SIZE]“ that stores matrices and the 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.

In summary, variables like „a“ and „b“ in the „main“ function are used to store data and have both read and write access. Variables within the „if\_converges“ function are used for calculations and control flow, with a mix of read and write access. The global constants are used for configuration and are read-only. Other variables are used for loop control and intermediate storage and have read and write access as required by the logic of the program.

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**

**Task 2.4**

**Task 2.5**

**Task 2.6**

**Task 4:**

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, tp(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:

tseq(T1) = 4 hours

tp(2) = 3.33 hours

sp(2) = tseq(T1) / tp(2) = 4/3.33 = 1.2, which is less than p=2 so we are happy.

𝜂p(2) = sp(2) / p = 1.2/2 = 0.6, i.e. an efficiency of 60%.

Assuming we are using T2 for the base sequential time:

tseq(T2) = 20 hours

tp(2) = 3.33 hours

sp(2) = tseq(T2) / tp(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.

𝜂p(2) = sp(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, tseq = 4 hours.

tp(2) = max{0.5\*4, 0.5\*4} = 2 hours

sp(2) = tseq / tp(2) = 4/2 = 2, which is equal to p, indicating perfect parallelisation.

𝜂p(2) = sp(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, tseq = 20 hours.

tp(2) = max{0.5\*20, 0.5\*20} = 10 hours

sp(2) = tseq / tp(2) = 20/10 = 2, which is equal to p, indicating perfect parallelisation.

𝜂p(2) = sp(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, tp(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:

tseq(T1) = 4 hours

tp(3) = 2.86 hours

sp(3) = tseq(T1) / tp(3) = 4/2.86 = 1.4, which is less than p=3 so we are happy.

𝜂p(3) = sp(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:

tseq(T2 or T3) = 20 hours

tp(3) = 2.86 hours

sp(3) = tseq(T2 or T3) / tp(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.

𝜂p(3) = sp(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 optimising 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.

**Task 5:**

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[3],[2]. 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.

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, MPIcan 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