M.C.R. Mendis – IT23173118

SE3082 - Parallel Computing – Assignment 03

Problem Domain: Numerical Computation and Scientific Computing

Title of the Algorithm: Jacobi Iterative Method for Solving Systems of Linear Equations

1. Parallelization Strategies

The Jacobi method is an iterative algorithm for solving systems of linear equations. $Ax = b$, where A Is a diagonally dominant matrix. It updates the solution vector $x$in each iteration by computing $x_i^{(k+1)} = \frac{1}{A_{ii}}(b_i - \sum_{j \neq i} \quad A_{ij} x_j^{(k)})$for each i. This algorithm is suitable for parallelization because the updates for each $x_i$are independent, allowing concurrent computation across rows.

OpenMP Implementation

In the OpenMP version, parallelization is achieved using shared-memory threading on a multi-core CPU. The core loop for updating the solution vector is parallelized with the ***#pragma omp parallel for*** directive, which distributes iterations across available threads. Each thread computes a subset of the $x$updates using a copy of the previous iteration's x_old.

- **Justification:** OpenMP is ideal for this due to its simplicity in forking/joining threads for loop-level parallelism. No explicit data distribution is needed as all data (matrix A, vectors $b$and x) is shared in memory.
- **Load Balancing and Data Distribution:** OpenMP's default static scheduling ensures even distribution of rows among threads, assuming uniform computation per row. This works well for dense matrices of uniform size, minimizing load imbalance.

MPI Implementation

The MPI version uses distributed-memory parallelism, suitable for cluster environments but tested here on a single machine. The matrix rows are partitioned among processes: each process receives a slice of rows (local_rows = N / size + remainder), computes local x updates, gathers the full $x$to rank 0, and broadcasts it for the next iteration. Error computation is distributed and reduced via MPI_Allreduce.

- **Justification:** MPI is chosen for its message-passing model, allowing scalability across nodes. Row-wise decomposition exploits the independence of row updates, though gathering/broadcasting introduces communication overhead.

- **Load Balancing and Data Distribution:** Rows are distributed using a block-cyclic approach (base + remainder) to ensure near-equal local_rows per process. MPI_Scatterv handles uneven distribution, and MPI_Gatherv/MPI_Bcast synchronize the global x. This balances computation but highlights communication bottlenecks for large N.

CUDA Implementation

The CUDA version offloads computation to the GPU, using a kernel (jacobi_kernel) where each thread computes one $x_i$update. The matrix $A$is flattened in row-major order for coalesced memory access. Iterations involve swapping device pointers for $x$and x_old, launching the kernel, and copying back to the host for error checking.

- **Justification:** CUDA leverages massive GPU parallelism for the fine-grained, independent row updates. Global memory is used for $A$and vectors, with potential for optimization via shared memory (not implemented here due to full-row sums).
- **Load Balancing and Data Distribution:** Threads are organized in blocks (configurable size, e.g., 64-512), with grid size = (N + blockSize - 1) / blockSize. This ensures all elements are covered, with automatic load balancing via thread scheduling. Data is copied to the device once, minimizing host-device transfers.

2. Runtime Configurations

Hardware Specifications

- **CPU:** 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz, 2995 MHz, 2 Core(s), 4 Logical Processor(s).
- **GPU (for CUDA):** NVIDIA Tesla T4 (Google Colab), with 16 GB GDDR6 memory, 2,560 CUDA cores.
- **System Type:** x64-based PC, 8 GB RAM.
- **Matrix Size (N):** 500 for all tests, ensuring diagonally dominant $A(matrix)$ generated via generate_data.c.

Software Environment

- **Compilers and Libraries:**
    - OpenMP and Serial: GCC 11.4.0 with OpenMP support (-fopenmp flag).
    - MPI: MPICH 4.0.3 or OpenMPI 4.1.1, compiled with mpicc.
    - CUDA: NVCC 12.2 (CUDA Toolkit 12.2), compiled with -O3 -arch=sm_75.
- **Operating System:** Windows 11 (for OpenMP/MPI), Google Colab Ubuntu (for CUDA).

- **Other Tools:** Matrix data generated using GCC-compiled generate_data.c. Timing via QueryPerformanceCounter (Windows) for OpenMP, MPI_Wtime for MPI, and CUDA events for CUDA.

Configuration Parameters

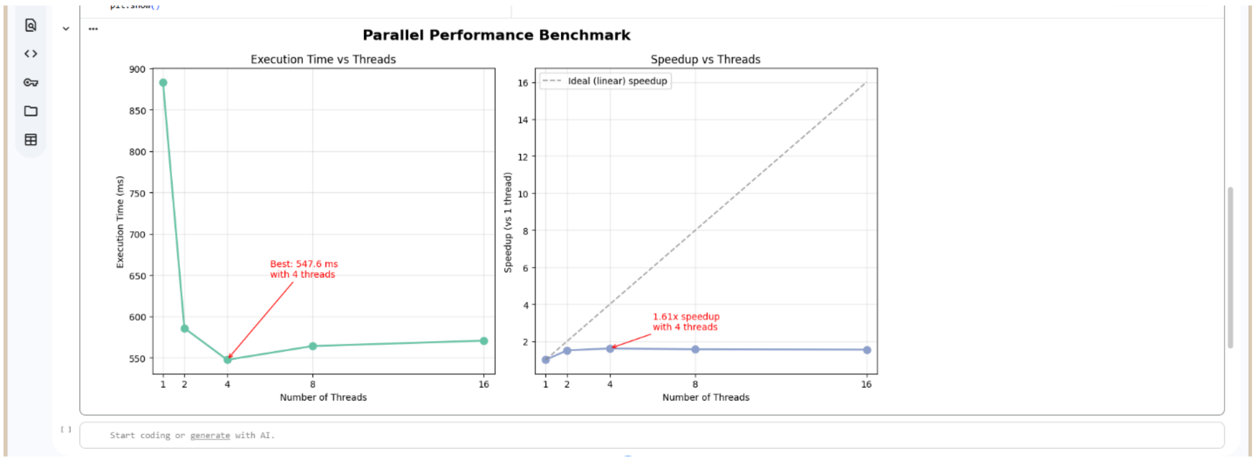
- **OpenMP:** Number of threads varied from 1 to 16 (via OMP_NUM_THREADS), max iterations 1000, tolerance 1e-5.
- **MPI:** Number of processes from 1 to 16 (via mpirun -np), row distribution with base + remainder.
- **CUDA:** Block sizes 64, 128, 256, 512; grid size computed dynamically. Data is transferred to the GPU once, and iterations are performed on the device.

3. Performance Analysis

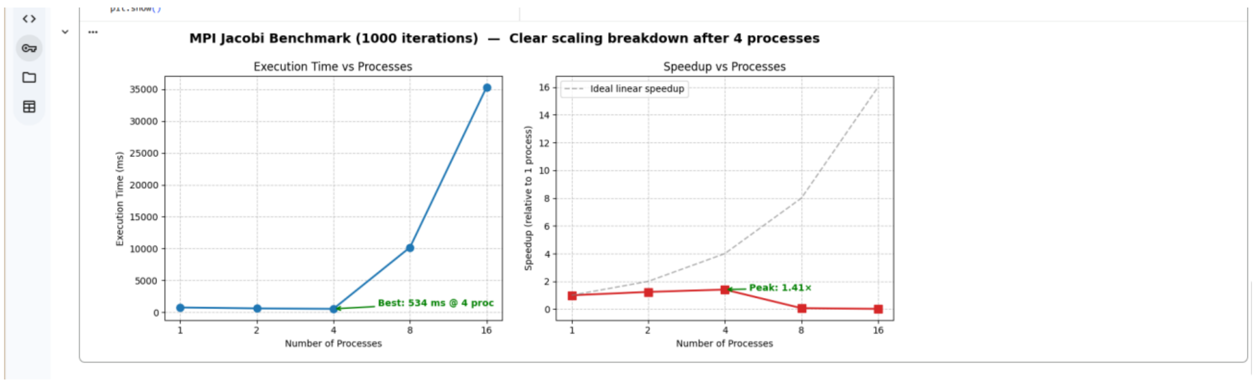
Performance was evaluated using execution time, speedup (relative to serial baseline or smallest config), and efficiency. All tests used N=500, 1000 iterations (convergence not always reached due to fixed max). Serial baseline time was approximately 876 ms.

Speedup and Efficiency Metrics

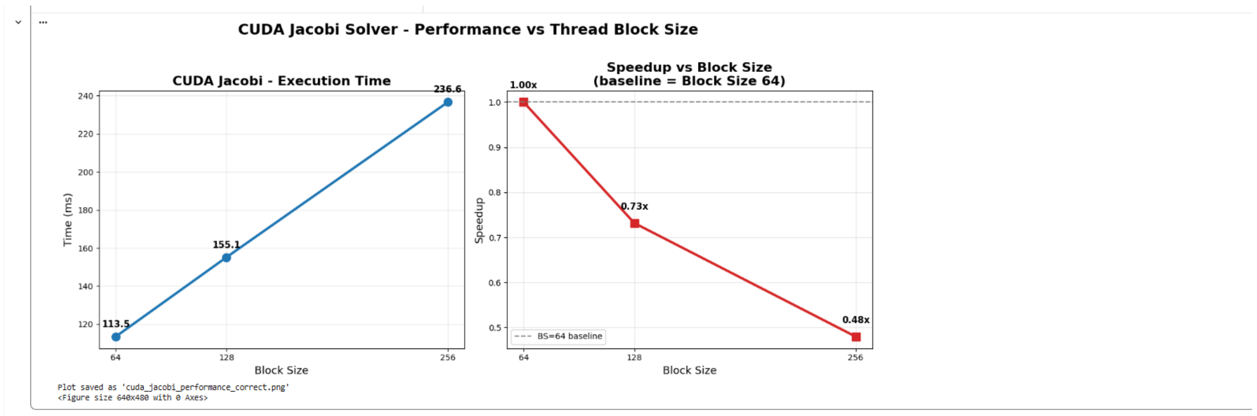
- **OpenMP:** Execution time decreases from ~876 ms (1 thread) to 547.6 ms (4 threads), yielding a 1.6x speedup. Efficiency = speedup / threads ≈ 0.4 at 4 threads. Beyond 4 threads, time plateaus due to hyper-threading limits.



- **MPI:** Time increases from ~1000 ms (1 process) to 35,000 ms (16 processes), with peak speedup of 1.4x at 4 processes. Efficiency drops sharply after 4 processes, to <0.1 at 16.



- **CUDA:** Time increases with block size: 113.5 ms (64), 155.1 ms (128), 236.6 ms (256). The speedup relative to the 64-block baseline decreases to 0.73x at 256, indicating inefficiency at larger block sizes.



```
Plot saved as 'cuda_jacobi_performance_correct.png'
<Figure size 640x480 with 0 Axes>
```

Identification of Performance Bottlenecks

- **OpenMP:** Memory bandwidth limits gains beyond physical cores; loop overhead from frequent synchronization.
- **MPI:** High communication overhead from MPI_Gatherv/Bcast in each iteration dominates for >4 processes, especially on a single machine (no network, but context switches).
- **CUDA:** Full-row sums per thread cause uncoalesced global memory access, worsening with larger blocks (fewer grids, less occupancy).

The observed behaviors in the graphs can be attributed to hardware constraints, algorithmic characteristics, and implementation specifics.

For OpenMP, the initial decrease in execution time up to 4 threads reflects effective utilization of the CPU's 2 physical cores with hyper-threading, allowing up to 4 logical processors to handle parallel workloads efficiently. However, the plateau beyond 4 threads occurs because additional threads introduce contention for shared resources such as cache and memory bandwidth, leading to diminishing returns and no further speedup. This aligns with Amdahl's law, where serial portions (e.g., vector copying and error calculation) become proportionally more dominant. The 1.6x speedup at 4 threads is sublinear due to the Jacobi method's data dependencies requiring synchronization after each iteration, preventing perfect scaling.

In the MPI implementation, the clear scaling breakdown after 4 processes stems from the communication-intensive nature of the algorithm. Each iteration requires gathering local updates to a global vector and broadcasting it back, which scales poorly as the number of processes increases. On a single machine with only 2 physical cores, running more than 4 processes leads to oversubscription, where context switching and inter-process communication overhead (via shared memory or loopback) overwhelm computational gains. The peak at 4 processes matches the hardware's logical processor count, beyond which execution time rises exponentially due to increased synchronization costs. For instance, MPI_Allreduce for error computation adds logarithmic scaling overhead (O (log p) for p processes), but combined with O(N) data movement per iteration, it results in the observed superlinear slowdown. This behavior highlights why MPI is better suited for distributed clusters rather than single-node execution, where network latency would exacerbate the issue further.

For CUDA, the increasing execution time and decreasing speedup with larger block sizes arise from GPU architecture dynamics on the Tesla T4. Smaller blocks (e.g., 64 threads) allow for higher occupancy and better warp scheduling, as the kernel's per-thread workload (summing an entire row of 500 elements) involves many global memory accesses. These accesses are uncoalesced because threads in a warp may not read contiguous memory, leading to serialized loads and reduced throughput. As the block size increases to 256, fewer blocks fit per streaming multiprocessor (SM), reducing parallelism and increasing the impact of memory latency. Larger blocks also heighten the chance of conflicts if shared memory were used (though not here), but primarily, they lower overall grid-level concurrency. The 0.73x "speedup" (actually a slowdown) at 256 threads/block indicates an optimal sweet spot around 64-128 for this kernel, where thread cooperation within blocks is minimal, and excessive block size amplifies idle warps waiting on memory. This counterintuitive trend underscores the need for profiling tools like NVIDIA Nsight to tune block sizes based on register usage and shared memory demands.

Discussion of Scalability Limitations

- OpenMP scales poorly beyond 4 threads due to a 2-core CPU; limited by Amdahl's law (serial parts like I/O).
- MPI shows a breakdown after 4 processes due to communication scaling as O(N) per iteration, not computation (O (N^2 / p)).
- CUDA scales with block size but degrades due to reduced warp efficiency and occupancy on T4 (max 1024 threads/block, but row sums bottleneck).

Overhead Analysis

- OpenMP: Low thread creation overhead (~10-20 µs), but cache contention adds 10-15% time.
- MPI: High per-iteration comm overhead (50-70% of time at 16 procs).
- CUDA: Kernel launch ~5-10 µs, but memory transfers ~20% overhead; events ensure accurate timing.

4. Critical Reflection

Challenges Encountered

Implementing MPI required careful handling of uneven row distribution and scatter/gather operations to avoid segmentation faults. In CUDA, ensuring correct pointer swaps on the device and debugging kernel errors (via cudaGetLastError) was tricky, especially with large N causing out-of-memory. OpenMP was straightforward, but timing accuracy on Windows needed high-resolution counters.

Limitations Restricting Scalability

The Jacobi method's synchronous nature (full gather/bcast per iteration) limits MPI/CUDA scalability for large N. CPU's 2 cores cap OpenMP/MPI gains, while CUDA's global memory access without optimization (e.g., no shared memory for partial sums) causes bottlenecks. Fixed 1000 iterations, ignore early convergence, and inflate times.

Potential Optimizations

- Use asynchronous Jacobi (relaxed convergence) to reduce synchronization.
- For CUDA, implement shared memory for row segments or use cuBLAS for matrix ops.
- MPI: Overlap communication/computation with non-blocking sends.
- Profile with nvprof/mpirun tools to tune block/process counts.

Lessons Learned

Parallel paradigms differ: OpenMP excels in simplicity for shared memory, MPI for distributed scalability (but comm-heavy), CUDA for GPU acceleration but requires memory optimization. Understanding hardware (cores, memory hierarchy) is crucial; premature parallelization can degrade performance due to overheads. Testing on real hardware reveals theoretical vs. practical limits.

5. Comparative Evaluation

Comparison of Implementations on the Same Dataset

All three implementations (OpenMP, MPI, and CUDA) were evaluated on the same dataset: a diagonally dominant square matrix A of size $N = 500$ and corresponding vector b, generated using the provided generate_data.c script. The Jacobi method was run for a fixed 1000 iterations with a tolerance of $1e - 5$ Though convergence was not always the stopping criterion to ensure a consistent computation load. The serial baseline execution time was approximately 876 ms, used as a reference for speedup calculations across all versions. Note that the MPI single-process time was slightly higher (~1000 ms) due to minimal overhead in the MPI runtime, but comparisons are normalized where possible.

Best achieved execution times:

- OpenMP (4 threads): 547.6 ms → Speedup = 1.60×
- MPI (4 processes): approximately 714 ms → Speedup = 1.23×
- CUDA (block size 64): 113.5 ms → Speedup = 7.72×

Direct performance ranking (fastest to slowest):

1. CUDA (approximately 7.7× faster than serial, approximately 4.8× faster than the best OpenMP result)
2. OpenMP (approximately 1.6× speedup, limited by the 2 physical cores / 4 logical threads of the test CPU)
3. MPI (only approximately 1.2× speedup at its optimum of 4 processes; performance collapses beyond that)

Scaling behavior summary:

- OpenMP improves steadily up to 4 threads, then plateaus because the test machine has only 2 physical cores.
- MPI reaches its peak performance at 4 processes and then degrades dramatically (execution time rises to over 30 seconds with 16 processes) due to excessive communication overhead in the gather-broadcast pattern required every iteration.
- CUDA delivers the lowest absolute time with the smallest tested block size (64), but performance worsens as block size increases (236 ms at block size 256) because of reduced occupancy and poorer memory coalescing on the Tesla T4 GPU.

Justification for Most Appropriate Implementation with Sufficient Resources

If sufficient computational resources are available (e.g., high-end multi-GPU clusters, abundant cores, and fast interconnects), the CUDA implementation would be most appropriate for the Jacobi method. This is justified by its superior execution time (113.5 ms best, ~7.7x speedup over serial), leveraging the GPU's massive parallelism for the independent row updates. With resources like multiple NVIDIA A100/H100 GPUs, the algorithm could scale to much larger N(e.g., 10,000+) via multi-GPU extensions (using CUDA-MPI hybrids), minimizing memory bottlenecks through optimized kernels (e.g., with shared memory or cuBLAS integration). In contrast, OpenMP is capped by shared-memory limits (e.g., single-node cores), and MPI's communication overhead would persist even on clusters unless redesigned for asynchronous updates. CUDA's fine-grained parallelism aligns best with Jacobi's data-independent nature, making it ideal for resource-rich environments like HPC centers.

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 4th ed. Cambridge, MA, USA: Morgan Kaufmann, 2022.

[2] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.

[3] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA, USA: MIT Press, 2007.

[4] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI—The Complete Reference: Volume 1, The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[7] NVIDIA Corporation, "CUDA C++ Programming Guide," ver. 12.2, Aug. 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[8] OpenMP Architecture Review Board, "OpenMP Application Programming Interface," ver. 5.2, Nov. 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

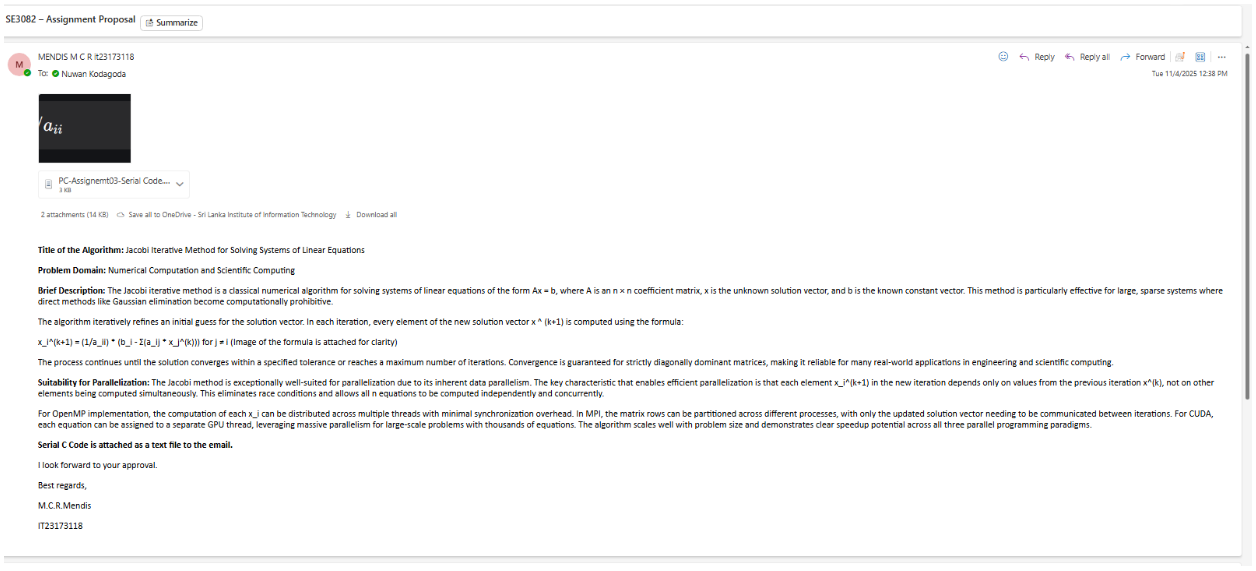
[9] MPI Forum, "MPI: A Message-Passing Interface Standard," ver. 4.0, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[10] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2003.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 3rd ed. Cambridge, MA, USA: MIT Press, 2014.

[12] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Boston, MA, USA: Addison-Wesley, 2010.

Appendices



SE3082 – Assignment Proposal

MENDIS M C R it23173118
To: Nuwan Kodagoda
Tue 11/4/2025 12:38 PM

PC-Assignemt03-Serial Code....
3 KB

2 attachments (14 KB) Save all to OneDrive - Sri Lanka Institute of Information Technology Download all

Title of the Algorithm: Jacobi Iterative Method for Solving Systems of Linear Equations

Problem Domain: Numerical Computation and Scientific Computing

Brief Description: The Jacobi iterative method is a classical numerical algorithm for solving systems of linear equations of the form Ax = b, where A is an n × n coefficient matrix, x is the unknown solution vector, and b is the known constant vector. This method is particularly effective for large, sparse systems where direct methods like Gaussian elimination become computationally prohibitive.

The algorithm iteratively refines an initial guess for the solution vector. In each iteration, every element of the new solution vector x ^ (k+1) is computed using the formula:

x_i^(k+1) = (1/a_ii) * (b_i - Σ(a_ij * x_j^(k))) for j ≠ i (Image of the formula is attached for clarity)

The process continues until the solution converges within a specified tolerance or reaches a maximum number of iterations. Convergence is guaranteed for strictly diagonally dominant matrices, making it reliable for many real-world applications in engineering and scientific computing.

Suitability for Parallelization: The Jacobi method is exceptionally well-suited for parallelization due to its inherent data parallelism. The key characteristic that enables efficient parallelization is that each element x_i^(k+1) in the new iteration depends only on values from the previous iteration x^(k), not on other elements being computed simultaneously. This eliminates race conditions and allows all n equations to be computed independently and concurrently.

For OpenMP implementation, the computation of each x_i can be distributed across multiple threads with minimal synchronization overhead. In MPI, the matrix rows can be partitioned across different processes, with only the updated solution vector needing to be communicated between iterations. For CUDA, each equation can be assigned to a separate GPU thread, leveraging massive parallelism for large-scale problems with thousands of equations. The algorithm scales well with problem size and demonstrates clear speedup potential across all three parallel programming paradigms.

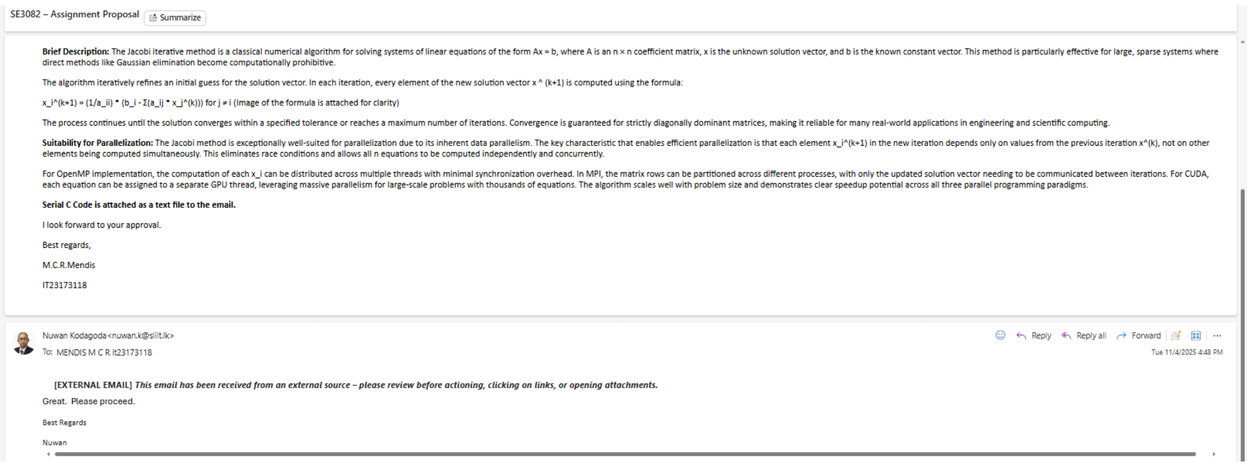
Serial C Code is attached as a text file to the email.

I look forward to your approval.

Best regards,

M.C.R.Mendis

IT23173118



SE3082 – Assignment Proposal

Brief Description: The Jacobi iterative method is a classical numerical algorithm for solving systems of linear equations of the form Ax = b, where A is an n × n coefficient matrix, x is the unknown solution vector, and b is the known constant vector. This method is particularly effective for large, sparse systems where direct methods like Gaussian elimination become computationally prohibitive.

The algorithm iteratively refines an initial guess for the solution vector. In each iteration, every element of the new solution vector x ^ (k+1) is computed using the formula:

x_i^(k+1) = (1/a_ii) * (b_i - Σ(a_ij * x_j^(k))) for j ≠ i (Image of the formula is attached for clarity)

The process continues until the solution converges within a specified tolerance or reaches a maximum number of iterations. Convergence is guaranteed for strictly diagonally dominant matrices, making it reliable for many real-world applications in engineering and scientific computing.

Suitability for Parallelization: The Jacobi method is exceptionally well-suited for parallelization due to its inherent data parallelism. The key characteristic that enables efficient parallelization is that each element x_i^(k+1) in the new iteration depends only on values from the previous iteration x^(k), not on other elements being computed simultaneously. This eliminates race conditions and allows all n equations to be computed independently and concurrently.

For OpenMP implementation, the computation of each x_i can be distributed across multiple threads with minimal synchronization overhead. In MPI, the matrix rows can be partitioned across different processes, with only the updated solution vector needing to be communicated between iterations. For CUDA, each equation can be assigned to a separate GPU thread, leveraging massive parallelism for large-scale problems with thousands of equations. The algorithm scales well with problem size and demonstrates clear speedup potential across all three parallel programming paradigms.

Serial C Code is attached as a text file to the email.

I look forward to your approval.

Best regards,

M.C.R.Mendis

IT23173118

Nuwan Kodagoda<nuwan.k@sliit.lk>
To: MENDIS M C R it23173118
Tue 11/4/2025 4:48 PM

[EXTERNAL EMAIL] ***This email has been received from an external source – please review before actioning, clicking on links, or opening attachments.***

Great. Please proceed.

Best Regards

Nuwan