MPI Assignment 2024:
Parallel Red-Black Successive Over-Relaxation

Student River Vaudrin
Course Coordinators dr. Francesc Verdugo & prof. dr. ir. Henri Bal
Teaching Assistant mr. Mikhail Cassar

1 Introduction

In the optional assignment for the VU course *Programming Large-Scale Parallel Systems*, we, the students, were tasked with implementing and analyzing the Red-Black Successive Over-Relaxation (SOR) algorithm, first introduced by Adams and Ortega [1]. The Red-Black SOR algorithm is used for solving large, sparse linear systems, typically arising from the discretization of partial differential equations (PDEs), such as the Poisson equation. By iteratively updating matrix values based on neighboring elements, it accelerates the convergence of solutions to these linear systems. This algorithm divides the computational domain, often represented as a 2D matrix, into red and black cells (hence the name) and solves the equations in a checkerboard pattern. During each iteration, red cells are updated based on the values of their neighboring black cells, and in the next step, black cells are updated based on the neighboring red cells. This checkerboard update scheme ensures that data dependencies between cells are limited, which allows for parallel updates of non-adjacent cells. This method makes the algorithm highly suitable for parallelization since each process can work on different parts of the grid simultaneously, reducing computational dependencies and improving efficiency. The pseudo-code for an equivalent sequential version of the Red-Black SOR algorithm is shown in Algorithm 1. Note that the Red-Black SOR algorithm typically includes a relaxation factor, denoted as ω, which adjusts the weighting of the new value in the iterative process to speed up convergence. However, in this implementation, we simplify by setting the relaxation factor to a constant value of 0.25.

The algorithm is implemented in Julia, a high-level programming language designed for high-performance numerical analysis and computational science [3]. Julia is particularly well-suited for parallel computing due to its built-in support for multi-threading, distributed computing, and native libraries for linear algebra and numerical methods. Moreover, the parallel communication between processors is managed using the Message Passing Interface (MPI), a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures [5]. MPI enables processes running on different nodes of a distributed memory system to communicate efficiently through message exchanges, ensuring proper synchronization and data sharing.

In this report, I present my implementation of the parallel Red-Black SOR algorithm and compare it to a equivalent sequential implementation provided by the lecturers. The analysis includes measuring the speedup and efficiency of the parallel implementation relative to the sequential one on VU's DAS-5 supercomputer [2]. Additionally, I extend the analysis by measuring the communication and computation overhead and estimating the communication-to-computation ratio on the same system. The results of these analyses will allow us to reason about the performance of the algorithm.

2 Implementation

The function signature we were tasked with implementing is `sor_par!(A, stopdiff, maxiters, comm)`. In this case, A is an $N \times N$ matrix that includes the boundary conditions. We can assume the function will always be called with a number of processes that is a multiple of $N-2$ (excluding boundary rows). The parameters `stopdiff` and

N	$P=2$	$P=4$	$P=5$	$P=8$	**Avg.**
600	0.84	0.64	0.59	0.42	0.62
1200	0.83	0.77	0.69	0.53	0.71
2400	0.85	0.85	0.75	0.66	0.78
4000	0.86	0.82	0.74	0.61	0.76
Avg.	0.85	0.77	0.69	0.56	

Table 1: Efficiency Results.

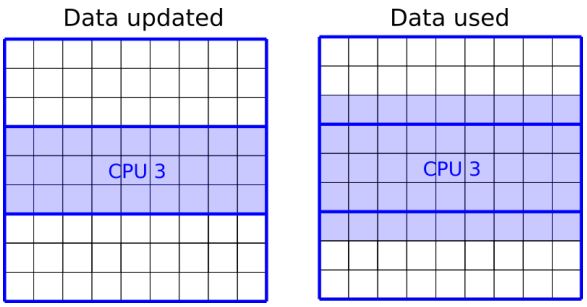

Figure 1: The left image illustrates the portion of the sub-matrix u that is updated during each iteration by process (CPU 3). The right image highlights the entries of u required to perform these updates.

`maxiters` control the stopping criterion and the maximum number of iterations, respectively, while `comm` represents the MPI communication object, which coordinates data exchange between processes. My implementation uses a 1D block row partitioning, where each process is responsible for a subset of consecutive rows while handling all columns of the matrix. This is a form of static load balancing, as the load sizes can be determined in advance based on known matrix dimensions. This guarantees that all processes have an equal share of computations. My parallel Red-Black SOR algorithm is structured into three key phases: the scatter phase, the computation phase, and the gather phase.

In the scatter phase, the initial communication ensures that all processes are set up with the correct data for the computation phase. This phase starts with broadcasting the matrix dimensions using `MPI.Bcast!`. Broadcasting is used here to ensure that all processes know the global problem size N. It is more efficient than individual send-receive operations, as its a single communication point, which reduces the complexity of synchronization between the master and the worker processes. Once the matrix size is shared, we calculate the number of rows each worker must process, referred to as the `load`, by computing $\frac{N-2}{P}$, where P is the number of processes. We need to exclude the boundary rows from this calculation because they do not participate in the interior updates. Next, the matrix A is distributed among the worker processes using `MPI.Scatter!`. Each process is assigned a portion of matrix A called u, with dimensions $(\texttt{load}+2) \times N$, since one row above and one row below the load are needed for computing the boundary values during updates. In my implementation, I use `MPI.Scatter!` to efficiently distribute A across the processes, avoiding the overhead of multiple sends from the master. Each worker receives an amount of rows equal to the `load`, with dimensions $\texttt{load} \times N$. For this, both the send and receive buffers are transposed due to MPI's handling of rows and columns. The boundary rows for each node are received during the computation phase, but the boundary rows for the master and the last process (`rank=P-1`) are handled during the scatter phase. This is because these rows remain uncharged during the computation phase. The first row of the master's u is the first row of A, which it already knows. Subsequently, the last process receives the last row of u (or last of A) directly from the master using `MPI.Send` and `MPI.Recv!`.

The computation phase executes the core Red-Black SOR algorithm. The algorithm's outer loops—those iterating over the iterations and the two color sweeps (red and black)—are sequential because updates depend on the previous iteration's values. However, the loops over the rows and columns within each process's subdomain (i.e., u) are parallelizable. Before these updates can occur, the boundary rows of u between neighboring processes must be exchanged using `MPI.Sendrecv!`. The choice of `MPI.Sendrecv!` minimizes deadlock risks because it handles both the send and receive operations simultaneously. Each process with `rank>0` (i.e., not the master) send their second row of u (the first row containing updated values) and receive their top boundary row of u from the previous process. Similarly, each rank that is not `rank=P-1` (the last process) send their second-to-last row of u (the last row containing updated values) and receive their bottom boundary row of u from the next process. The data dependencies for a single process can be seen in Figure 1. After updating the values of both red and black cells, `MPI.Allreduce!` is used to find the global maximum difference between the old and new values $\max \|A^{k+1}ij - A^k ij\|$ across all processes. This is done to check if the stopping criterion, $\max \|A^{k+1}ij - A^k ij\| < \epsilon$, is met.

The gather phase involves combining the interior rows of all of the different u matrices from each process into the final result matrix A. This is done using `MPI.Gather!`, which assembles the submatrices u from each process into the full matrix. Similar to the scatter phase, both the send and receive buffers are transposed, ensuring correct assembly of the matrix. And again `MPI.Gather!` is used because it allows the master to efficiently collect the results without multiple point-to-point communications.

N	P	`stopdiff`	`maxiters`	`matrix_id`
[600, 1200, 2400, 4000]	[1, 2, 4, 5, 8]	$10e^{-20}$	1000	3

Table 2: Experimental Set-Up Parameters.

3 Experimental Set-up

To evaluate the performance of my parallel Red-Black SOR algorithm, I measured the execution time of the provided sequential implementation on DAS-5 using the Julia function `@elapsed`. In parallel, I tested my implementation on the same system, varying the number of processes and matrix sizes. Each configuration was run five consecutive times, and the fastest execution times were used to calculate both speedup and efficiency [6]. Speedup is defined as $S_P = \frac{T_1}{T_P}$, where T_1 is the best execution time of the sequential implementation, and T_P is the best execution time using P processes. Efficiency, defined as $\frac{S_P}{P}$, measures the utilization of computational resources. It indicates how effectively the algorithm scales as more processors are added. Efficiency is a critical metric because it reflects the diminishing returns of adding more processors—while speedup shows how quickly the problem is solved, efficiency highlights whether the added processes are fully utilized or if overhead (e.g., communication) is hindering performance. The parameter values used in the experiments are listed in Table 2. For the initial guess matrix, referred to as `matrix_id` $= 3$, all boundary values were set as $A^0_{ij} = \frac{(i+j)}{N}$. Results are presented by showing the efficiency and best-case execution time for each combination of matrix size N and number of processes P.

In addition to measuring efficiency, I also estimated the communication-to-computation ratio (CCR) of my implementation, which provides insight into how well the algorithm balances communication overhead with actual computation [4]. This ratio is calculated as $CCR = \frac{T_{comm}}{T_{comp}}$, where T_{comm} is the total time spent on communication, and T_{comp} is the total time spent on computation. A higher CCR indicates that the algorithm spends a significant portion of time in communication rather than computation, which may explain reduced efficiency for certain configurations. If the communication time dominates computation, it suggests that increasing the number of processes will lead to inefficiencies, as more time is spent synchronizing data between processes rather than performing useful work. On the other hand, a lower CCR indicates a more computation-bound workload, where parallel resources are being used more efficiently. To gather the communication and computation timings, I used `MPI.Wtime` to mark the start and end of different code blocks. Communication times include operations such as broadcasting N, scattering matrix A, exchanging boundary rows of u, performing global reductions to check the stopping condition, and gathering the results. The transpositions necessary for the scatter and gather operations are also included. Computation times refer to the inner loops updating the red and black cells in the matrix. These timings are collected from all processes, and `MPI.Reduce` is used to aggregate the measurements into two final values: total communication time and total computation time across all processes. This experiment was also repeated for five consecutive runs on DAS-5, using the same parameter values as in the efficiency experiment (Table 2). Results will show the best-case CCR, alongside the corresponding communication and computation times for each configuration of N and P.

Before presenting the results of these experiments, I will first outline my implementation's theoretical complexity in terms of the number of processes P and the matrix size N:

- **Communication**: During each iteration, each process (except the first and last) sends and receives two entire rows, leading to a communication cost of $2N$. Therefore, the communication complexity is $O(2N)$.
- **Computation**: In each iteration, all the interior values of the sub-matrix assigned to each process are updated. Given that each process handles $\frac{N}{P}$ rows, the total number of values updated per process is $\frac{N}{P} \times N$, resulting in a computational complexity of $O\left(\frac{N^2}{P}\right)$.
- **CCR**: The ratio of communication to computation is $\frac{2N}{\frac{N^2}{P}} = \frac{2P}{N}$. This simplifies to $O\left(\frac{P}{N}\right)$, indicating that when $P \ll N$, the communication overhead becomes negligible. This suggests that the implementation has the potential to scale well for large problem sizes.

4 Discussion

The results for efficiency and best-case execution times are presented in Tables 1 and 3. As expected, increasing the problem size results in longer execution times on average, regardless of the number of processes. However, distributing the workload across multiple processes consistently reduces the execution time, as parallelization allows for more computations to be completed simultaneously. While this confirms that the parallel implementation works as intended, it does not necessarily demonstrate its efficiency or scalability. The efficiency results reveal a decline

in average efficiency as the number of processes increases. This behavior is typical in parallel algorithms due to communication overhead, which grows faster than the computation time as more processes are introduced. The increased communication between processes adds significant overhead, causing diminishing returns in speedup. This explains why, in some cases, the parallel implementation performs worse than the sequential one, which does not suffer from communication costs. However, efficiency tends to improve with larger problem sizes. In larger problems, the computation time increases more rapidly than communication time, meaning that each process performs more computations relative to the amount of data exchanged. As a result, the negative impact of communication overhead is mitigated, and the parallel implementation becomes more efficient. This trend highlights the importance of problem size when determining the viability of parallelization; larger problems benefit more from parallel execution than smaller ones.

The results from the second experiment, focusing on the CCR, are summarized in Tables 4, 5 and 6. As anticipated, communication time increases with the number of processes and the size of the problem. More processes require more data exchanges between them, while larger problems involve transmitting larger chunks of data, leading to higher communication overhead. In contrast, computation time remains relatively stable as the number of processes increases. This is because the total number of computations required for a given problem size remains constant, but they are distributed across more processes. The increase in computation time is only observed when the problem size itself grows, as larger matrices naturally require more operations. The CCR results indicate that, for all configurations, the ratio is less than 1, meaning that most of the execution time is spent on computation rather than communication. This suggests that the implementation scales well, with communication overhead being relatively small compared to the total computation time. However, a closer look reveals that the average CCR increases with the number of processes. This occurs because the computation time remains roughly constant while the communication time grows with additional processes, leading to a higher proportion of time spent on communication. On the other hand, the average CCR decreases as the problem size increases. This is because larger problems demand more computations per process, and as a result, the computation time grows faster than communication time. This trend indicates that the parallel algorithm becomes more efficient with larger problem sizes, as the growing computation workload helps to mask the communication overhead.

N	$P = 1$	$P = 2$	$P = 4$	$P = 5$	$P = 8$	**Avg.**
600	1.11	0.66	0.43	0.37	0.33	0.58
1200	4.44	2.68	1.44	1.29	1.04	2.18
2400	20.42	11.96	6.03	5.46	3.87	9.55
4000	56.95	32.99	17.34	15.44	11.57	26.86
Avg.	20.73	12.07	6.31	5.64	4.02	

Table 3: Best-Case Execution Time (in seconds).

N	$P = 2$	$P = 4$	$P = 5$	$P = 8$	**Avg.**
600	0.04	0.27	0.35	0.77	0.38
1200	0.02	0.12	0.20	0.43	0.19
2400	0.02	0.08	0.18	0.26	0.14
4000	0.01	0.09	0.14	0.27	0.13
Avg.	0.02	0.14	0.22	0.43	

Table 4: Best-Case CCR

N	$P = 2$	$P = 4$	$P = 5$	$P = 8$	**Avg.**
600	0.06	0.37	0.51	1.23	0.54
1200	0.14	0.63	1.16	2.66	1.15
2400	0.41	1.90	4.41	6.81	3.38
4000	0.92	6.15	9.62	19.97	9.17
Avg.	0.38	2.26	3.93	7.67	

Table 5: Best-Case Communication Time (in seconds).

N	$P = 2$	$P = 4$	$P = 5$	$P = 8$	**Avg.**
600	1.46	1.41	1.47	1.60	1.49
1200	5.58	5.43	5.69	6.15	5.71
2400	25.10	23.45	24.25	26.08	24.72
4000	76.10	67.74	70.12	75.11	72.27
Avg.	27.06	24.51	25.38	27.24	

Table 6: Best-Case Computation Time (in seconds).

5 Conclusion

In summary, the experiments demonstrate that my parallel Red-Black SOR algorithm performs efficiently, particularly for larger problem sizes when a smaller number of processes is used. In these scenarios, the CCR is consistently below 0.10, indicating that the majority of the execution time is devoted to computation. The best efficiency achieved, as high as 0.86, correlates with the lowest CCR values, demonstrating that minimizing communication overhead and/or increasing the problem size significantly enhances performance. Which is in alignment with the theoretical complexity analysis. Thus, the results also emphasize the need to consider both problem size and process count carefully when optimizing for performance in parallel systems.

I spent a total of four days, working nine hours per day, on implementing the parallel version of this algorithm. It's safe to say that I've learned a lot from the assignment, particularly as it was my first time working extensively with Julia, I must admit, I like it quite a bit. But also my first time using the MPI protocol, prior to this course, I had only vaguely heard of MPI. So with all this being new to me, I learned a lot of new syntax, functions and logic. But most importantly, I learned to always be mindful of communication overhead when designing parallel algorithms.

References

[1] L. M. Adams and J. M. Ortega. "A Multi-Color SOR Method for Parallel Computation". In: *IEEE Computer Society* (1982), pp. 53–56.

[2] H. Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer (Long Beach, Calif.)* 49.5 (2016), pp. 54–63.

[3] J. Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[4] M. Crovella et al. "Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction". In: *IEEE Parallel and Distributed Processing* (1992), pp. 238–245.

[5] J. J Dongarra et al. "A Message Passing Standard for MPP and Workstations". In: *Communications of the ACM* 39.7 (1996), pp. 84–90.

[6] Wikipedia: The Free Encyclopaedia. *Analysis of parallel algorithms*. 2024. URL: `https://en.wikipedia.org/wiki/Analysis_of_parallel_algorithms`.

Algorithm 1 Black-Red SOR Algorithm Using Matrix A (No Relaxation Factor).

Input: Matrix A, tolerance ϵ, maximum iterations max_iter
Output: Updated matrix A
procedure sor_seq!(A, ϵ, max_iter)
    Initialize A^0 (initial guess)
    for $k = 0$ to $max\_iter - 1$ **do**
        for each color c in {black, red} **do**
            if c is black **then**
                for each node (i, j) such that $(i + j)$ is even **do** ▷ Black node condition
                    $A_{ij}^{k+1} \leftarrow 0.25 \cdot (A_{i+1,j}^{k} + A_{i-1,j}^{k} + A_{i,j+1}^{k} + A_{i,j-1}^{k})$
                end for
            else
                for each node (i, j) such that $(i + j)$ is odd **do** ▷ Red node condition
                    $A_{ij}^{k+1} \leftarrow 0.25 \cdot (A_{i+1,j}^{k+1} + A_{i-1,j}^{k+1} + A_{i,j+1}^{k+1} + A_{i,j-1}^{k+1})$
                end for
            end if
        end for
        Check convergence: $\|A^{k+1} - A^k\| < \epsilon$
        if converged **then**
            break
        end if
    end for
    return A
end procedure