Parallelization Scheme

1 Question 1: Data Distribution in a 1D Block Split

We perform a 1D (row-wise) block split of an $M \times N$ matrix across P tasks (labeled $0, 1, \ldots, P-1$). Define the split points

$$r_p \;=\; \left\lfloor \frac{p\,M}{P} \right\rfloor, \quad r_{p+1} \;=\; \left\lfloor \frac{(p+1)\,M}{P} \right\rfloor, \quad p = 0, 1, \ldots, P-1.$$

Then Task p handles exactly the rows

$$i \;=\; r_p,\; r_p + 1,\; \ldots,\; (r_{p+1} - 1) \quad \text{for all columns } j = 0, \ldots, N-1.$$

Each task's local submatrix has dimensions

$$(r_{p+1} - r_p) \;\times\; N.$$

When neighboring rows belong to different tasks, those tasks exchange data only for their boundary rows. All other rows are processed independently within each task.

1.1 Which Part of the Matrix Gets Managed by Which Task?

- **Task 0:**
  Rows
  $$i \;=\; r_0 \;=\; 0 \quad \text{through} \quad (r_1 - 1) \;=\; \left\lfloor \frac{M}{P} \right\rfloor - 1,$$
  Columns $j = 0, \ldots, N-1$.
- **Task 1:**
  Rows
  $$i \;=\; r_1 \;=\; \left\lfloor \frac{M}{P} \right\rfloor \quad \text{through} \quad (r_2 - 1) \;=\; \left\lfloor \frac{2\,M}{P} \right\rfloor - 1,$$
  Columns $j = 0, \ldots, N-1$.
- **Task p:**
  Rows
  $$i \;=\; r_p \;=\; \left\lfloor \frac{p\,M}{P} \right\rfloor \quad \text{through} \quad (r_{p+1} - 1) \;=\; \left\lfloor \frac{(p+1)\,M}{P} \right\rfloor - 1,$$
  Columns $j = 0, \ldots, N-1$.
- **Task $P-1$:**
  Rows
  $$i \;=\; r_{P-1} \;=\; \left\lfloor \frac{(P-1)\,M}{P} \right\rfloor \quad \text{through} \quad (\,r_P - 1\,) \;=\; M - 1,$$
  Columns $j = 0, \ldots, N-1$.

These definitions partition the row indices $0, \ldots, M-1$ into P contiguous, non-overlapping intervals. Each task thus owns a single "band" of rows, and communication is required only at the edges of these intervals when data from an adjacent task is needed.

1.2 Visual Representation of the Data Distribution

Below is a centered, monospaced depiction of how the M rows are split among the P tasks. Each horizontal band spans all columns 0 through $N-1$. The notation `r_p ...r_{p+1}-1` denotes the row indices owned by Task p.

```
Columns →  0  ...  N-1
Rows ↓
+-------------------------------------+
|  Task 0: rows r_0 ... r_1 - 1       |
+-------------------------------------+
|  Task 1: rows r_1 ... r_2 - 1       |
+-------------------------------------+
|                                     |
+-------------------------------------+
|  Task p: rows r_p ... r_{p+1} - 1   |
+-------------------------------------+
|                                     |
+-------------------------------------+
|  Task P-1: rows r_{P-1} ... M - 1   |
+-------------------------------------+
```

- Each box represents the contiguous set of row indices assigned to a given task.
- For Task p, the height of its band is $r_{p+1} - r_p$, which is approximately $\frac{M}{P}$ rows.
- Only the top and bottom rows of each band may need to communicate with neighboring tasks; all interior rows remain local to that task.

2 Parallelization Schemes and Termination Problem

Parallelization scheme for the Jacobi method

Distributed Memory (MPI)

Start of Iteration (Immediately)

At the beginning of each iteration, each MPI task:

- Sends its top boundary row to the task above (if such a task exists).
- Sends its bottom boundary row to the task below (if such a task exists).
- Posts nonblocking receives to fetch the rows immediately above and below its assigned block.

While Waiting for Neighbor Data

- The task computes updates for all interior rows in its block (those rows not on the top or bottom edge). These interior updates depend only on values already local to the task.

After Neighbor Rows Arrive

- The task updates its top boundary row using:
  - Values from the received row above ("ghost row")
  - Local left/right neighbors in the top boundary row.

- The task updates its bottom boundary row using:
  - Values from the received row below ("ghost row")
  - Local left/right neighbors in the bottom boundary row.

End of Iteration

Once both interior and boundary rows are updated:

- The task computes its local convergence metric (e.g., maximum difference or residual over its assigned rows).
- The task participates in a global reduction (e.g., `MPI_Allreduce`) to check if the overall convergence criterion has been met.

Shared Memory (OpenMP / Pthreads)

Iteration Begins

- Each thread updates interior rows in its region first, using values from the shared array that were written in the previous iteration.

Barrier Ensures Data Visibility

A synchronization barrier follows the interior updates, ensuring:

- All interior updates by all threads are visible to any thread that will update boundary rows.
- At this point, the rows immediately above and below each thread's region are fully updated from the previous iteration.

After Barrier: Boundary Updates

- Each thread updates its top boundary row by reading the row above from the shared array.
- Each thread updates its bottom boundary row by reading the row below from the shared array.
- If multiple threads might write adjacent boundary cells simultaneously, locks or atomic operations prevent data races.

Before Next Iteration

A second barrier ensures:

- All boundary updates are complete.
- Every thread sees consistent data when the next iteration begins.

What data does the task require from its neighbors and when will the data exchange take place?

Data Required

In each iteration, a task (or thread) needs:

- The row immediately above its assigned block, in order to update its top boundary row.
- The row immediately below its assigned block, in order to update its bottom boundary row.

MPI: Timing of Data Exchange

- At the very start of the iteration, the task posts nonblocking sends of its current top/bottom boundary rows to neighbors above/below, and posts nonblocking receives for those same neighbor rows.
- Boundary updates occur only after the incoming neighbor rows have arrived (i.e., after completing the interior-row computation and the corresponding `MPI_Wait`/`MPI_Test` for each receive).

Shared Memory: Timing of Data Exchange

- Neighbor rows are already in the shared array (from the previous iteration).
- A synchronization barrier ensures that the threads responsible for those neighbor rows have updated them before any thread reads them for boundary updates.
- Thus, explicit messaging is not required; the barrier enforces correct ordering.

Variables and Data Access by Task

Local Block of the Matrix

- Each task (MPI) or thread (OpenMP/Pthreads) maintains:
  - A local copy of all cells in its assigned block of rows (current-iteration values).
  - A buffer (or double-buffer) to store next-iteration values for those same rows.

Ghost-Row Buffers (MPI Only)

- Two temporary arrays of length N:
  - `topGhost` holds the fetched row immediately above the task's block.
  - `bottomGhost` holds the fetched row immediately below the task's block.

Interior Updates

- For any cell not on the top or bottom edge of the block:
  - The task uses values from its own local array: left neighbor, right neighbor, and the rows immediately above and below (all within the block).
  - No communication is required for interior cells.

Boundary Updates

- **Top Boundary Cells:**
  - Need values from `topGhost` (fetched row above) plus local left/right neighbors.
  - In shared memory, these values are read directly from the shared global array (after the barrier).
- **Bottom Boundary Cells:**
  - Need values from `bottomGhost` (fetched row below) plus local left/right neighbors.
  - In shared memory, these values are read directly from the shared global array (after the barrier).

Convergence Tracking

- Each task computes a local convergence metric (e.g., maximum difference over all its cells).
- In MPI: tasks perform a collective reduction (`MPI_Allreduce`) to determine global convergence.

- In shared memory: threads use a shared-memory reduction or atomic updates to track the global convergence metric.

Neighbor Identification

- **MPI:**
  - Each task knows the rank of the neighbor above $(p-1)$ and below $(p+1)$.
  - These ranks are used to post sends/receives for ghost rows.
- **Shared Memory:**
  - Threads implicitly know which rows they own (by index).
  - No explicit neighbor IDs are needed; row indices determine which thread will update a given neighboring row.

3 Parallelization Scheme for the Gauss–Seidel Method

Distributed Memory (MPI)

Processing Order

Gauss–Seidel updates use the most recent available values immediately. Each task proceeds row by row within its assigned block of rows.

Start of Iteration (Topmost Row in Block)

- To update the first row of its block, Task p must receive the updated bottom boundary row from Task $(p-1)$.
- Task p posts a receive for this data and blocks until it arrives before computing its top boundary cell.

Communication During Iteration

For each row i in Task p's block, from top to bottom:

1. Ensure the updated row $(i-1)$ is available. If i is the first row, receive from Task $(p-1)$; otherwise, use the already-updated local row $(i-1)$.
2. Compute the updated values for row i using:
   - Left/right neighbors in row i.
   - The already-updated value of row $(i-1)$.
   - The old (or, if already updated, the new) value of row $(i+1)$.
3. Immediately send the updated row i to Task $(p+1)$ (if one exists) so that the neighbor can use it as soon as it needs its top boundary.

End of Iteration (Bottommost Row)

- After updating and sending the bottommost row to Task $(p+1)$, Task p computes its local convergence metric.
- Since all rows in the block have been updated using the freshest data, Task p then participates in a global reduction (e.g., `MPI_Allreduce`) to check for convergence.

Shared Memory (OpenMP / Pthreads)

Row-by-Row Dependencies

Gauss–Seidel requires that, before updating row i, the value of row $(i-1)$ has already been updated. Threads coordinate this dependency via row-level locks or flags.

Lock or Flag for Each Row

- Each row i in the global array has an associated lock or atomic flag.
- To update row i, Thread p first acquires (or checks) the lock/flag for row $(i-1)$ to ensure it has been updated.
- Once row $(i-1)$ is available, Thread p updates row i and then releases (or sets) the lock/flag for row i so that the thread responsible for row $(i+1)$ can proceed.

Avoiding Global Barriers

- No global synchronization barrier is needed at each row. Instead, row-level locks or flags enforce the correct ordering.
- After finishing all rows in its block, Thread p computes its local convergence metric, then participates in a shared-memory reduction (or performs an atomic update) to decide if overall convergence has been met.

Neighbor Data Requirements and Exchange Timing

Neighbor Data Needed for Each Row i

- The updated version of row $(i-1)$ (the row immediately above) is required to compute row i. For the first row of the block, in MPI this comes via a blocking receive from Task $(p-1)$; in shared memory, it comes from the shared array after the corresponding row-level lock is set.

Exchange Timing (MPI)

- For the topmost row of Task p's block, a receive must be posted and awaited before performing any update.
- After computing any row i, Task p immediately sends the new row i to Task $(p+1)$ so that when Task $(p+1)$ reaches its boundary, it has the correct up-to-date values.

Exchange Timing (Shared Memory)

- To update row i, a thread must wait until row $(i-1)$'s lock/flag indicates it has been updated.
- No explicit messaging is used; row-level synchronization ensures the correct ordering.

Variables and Data Access

Local and Neighbor Rows

To update row i in Task p's block:

- Access local left and right neighbors in row $(i, j \pm 1)$.
- Read the already-updated value at row $(i-1)$ (from a received buffer in MPI or directly from the shared array after synchronization).
- Read the old or already-updated value at row $(i+1)$, depending on whether that row was processed earlier.

Temporary Send Buffers (MPI Only)

- After updating row i, Task p copies the entire row into a send buffer and posts a non-blocking send to Task $(p+1)$. No long-lived ghost arrays are needed—each row is sent immediately after being computed.

Row-Level Locks or Flags (Shared Memory)

- Each row i has an associated lock or atomic flag:
  - When a thread has updated row i, it releases (or sets) the lock/flag for row i, which allows the thread responsible for row $(i+1)$ to proceed.

Convergence Metric

- Each task/thread tracks the maximum change while updating rows in its block.
- After processing all rows, it participates in:
  - MPI: A global reduction (e.g., `MPI_Allreduce`).
  - Shared Memory: A shared-memory reduction or atomic update to determine if the global convergence criterion is satisfied.

Neighbor Identification

- MPI:
  - Task p sends to rank $(p+1)$ and receives from rank $(p-1)$ row by row.
- Shared Memory:
  - A thread infers which lock/flag corresponds to row $(i-1)$ or row $(i+1)$ based on the row index—no explicit neighbor ID is needed.

4 Discussion of the Termination Problem

Four termination scenarios to consider:

1. **Jacobi with a fixed number of iterations.**

   - Each task stops immediately after completing the last predetermined iteration. No convergence check is needed; tasks simply cease computation.

2. **Jacobi with precision-based (residual/norm) stopping.**

   - After each iteration, every task computes its local residual or maximum change. Tasks then perform a collective reduction (MPI_Allreduce or shared-memory reduction) to obtain the global norm.
   - As soon as the global norm falls below the tolerance, all tasks stop. Because the reduction is blocking, every task will have completed that iteration and will exit together.

3. **Gauß–Seidel with a fixed number of iterations.**

   - Tasks perform row-by-row updates in each iteration. Once a task has updated all assigned rows for the final iteration, it stops immediately. No additional checks or communication beyond inter-row exchanges are required.

4. **Gauß–Seidel with precision-based (residual/norm) stopping.**

   - As each task finishes updating its final row in iteration k, it computes its local norm.
   - Tasks then participate in a collective reduction to compute the global norm.
   - When the global norm is below tolerance, every task halts. All tasks must reach and complete the reduction for iteration k before any can exit.

There are four cases to consider: termination based on a fixed number of iterations and termination based on precision, each for Jacobi and Gauß–Seidel.

2.1 At which time will a task notice that the termination condition has been reached and it can stop its calculations?

- **Fixed number of iterations:** Each task knows in advance the total iteration count. It simply stops after completing its assigned work for the final iteration and does not need to check any values. No further communication is required once the final iteration's updates are done.
- **Precision-based (residual or norm threshold):**
  - After each iteration's updates, each task computes its local convergence metric (e.g., the maximum change or residual over its block).
  - In MPI, tasks then participate in a collective reduction (e.g., MPI_Allreduce) to compute the global convergence measure.
    - A task notices the condition when the reduction returns a value below the tolerance. At that moment—immediately after the reduction—the task can stop further iterations.
    - However, because all tasks must call the reduction, every task will perform at least one reduction call per iteration until the final one. Once any task detects convergence, it can set a shared flag (or rely on the return value of the reduction) to skip further updates.
  - In shared memory, tasks (threads) write their local metric into a shared location (e.g., using atomic updates to track the global maximum). A barrier or another synchronization point ensures that every thread sees the aggregated global metric. When that metric is below the tolerance, all threads stop.

2.2 In which iteration will the other tasks be at the time that a task notices the termination condition has been reached?

- **Fixed number of iterations:** All tasks finish exactly at the same iteration number. No task is 'faster' or 'slower', since iteration counts are predetermined.
- **Precision-based termination:**
  - In MPI, tasks perform their local updates for iteration k, then immediately participate in the reduction. When the reduction completes, all tasks receive the global norm at the same time (assuming a blocking collective). Thus, every task is synchronized at the reduction call and learns of convergence concurrently. At that point, all tasks have completed the iteration k.
  - In a non-blocking or asynchronous scheme (rare), some tasks might finish their local computations slightly later, but they cannot proceed to iteration $k+1$ until the reduction from iteration k is complete. Therefore, whenever a task 'notices' convergence, all other tasks are also in iteration k and at the same reduction barrier.
  - Shared memory: Threads compute local norms, then synchronize (e.g., via a barrier) before checking the global metric. Once the aggregated metric is below the threshold, all threads exit from the iteration loop together. No thread can be ahead by a full iteration because they synchronize at each check.