Q1 Poisson Solver with RMA-based Ghost Exchange

Implementation of a 2D Poisson equation solver using a 5-point stencil finite difference method with a 2D processor decomposition.
The key feature is the implementation of ghost/halo exchanges using MPI's Remote Memory Access (RMA) operations instead of traditional message passing.

Features

- 2D domain decomposition for parallel processing
- Three ghost/halo exchange methods:
  1. Traditional MPI message passing (Send/Recv)
  2. RMA with MPI_Win_fence synchronization
  3. RMA with general active target synchronization (MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait)
- Jacobi iteration for solving the Poisson equation
- Performance timing for comparing different communication methods

Compiling

To compile the program:

```
make
```

Or manually with:

```
mpicc -o poisson_solver_rma poisson_solver_rma.c -lm
```

Usage

Run the program with:

```
mpiexec -n <num_processes> ./poisson_solver_rma <grid_size>
[max_iterations] [exchange_method]
```

Where:

- `<num_processes>` : Number of MPI processes (should ideally be a perfect square for good domain decomposition)
- `<grid_size>` : Size of the global grid (N×N)
- `[max_iterations]` : Maximum number of Jacobi iterations (default: 10000)
- `[exchange_method]` : Method for ghost exchange:
  - 0: Traditional MPI message passing (default)
  - 1: RMA with MPI_Win_fence synchronization
  - 2: RMA with general active target synchronization

How to Run the Code

Run with MPI message passing:

```
make run
```

Run with RMA fence synchronization:

```
make run_fence
```

Run with RMA general active target synchronization:

```
make run_gats
```

Run all tests with a smaller grid for quick comparison:

```
make test_all
```

Expected Output

For each run, we see output similar to:

```
Iteration 100, Residual: 1.234567e-04
...
Solution converged after X iterations with residual Y.YYYYYYe-ZZ
Grid size: 100 x 100, Processor grid: 2 x 2
Exchange method: N (METHOD_NAME)
Setup time: 0.XXXX s
Solve time: Y.YYYY s
Total time: Z.ZZZZ s
```

Understanding the Output

The program will output:

- Number of iterations required for convergence
- Final residual error
- Grid size and processor grid configuration
- Exchange method used
- Setup time, solve time, and total execution time

Technical Details

2D Domain Decomposition

The global NxN grid is divided among MPI processes in a 2D Cartesian topology. Each process is responsible for computing a local portion of the grid and exchanging ghost/halo cells with its neighbors.

Ghost/Halo Exchange Methods

1. **MPI Message Passing**:
   - Uses non-blocking MPI_Isend and MPI_Irecv operations
   - Requires explicit message passing between processes
2. **RMA with MPI_Win_fence**:
   - Uses one-sided communication with MPI_Put operations
   - Synchronization is performed using MPI_Win_fence calls
   - Simpler to implement but potentially less efficient
3. **RMA with General Active Target Synchronization**:
   - Uses MPI_Win_start, MPI_Win_complete, MPI_Win_post, and MPI_Win_wait
   - Allows more fine-grained control over synchronization
   - Potentially more efficient by reducing global synchronization

Jacobi Iteration

The program uses Jacobi iteration to solve the discretized Poisson equation:

- Each grid point is updated based on its four neighbors and the source term
- Convergence is determined by calculating the global residual error

Efficiency Comparison

Generally, RMA operations can be more efficient than traditional message passing in certain communication patterns, the traditional point-to-point communication (Send/Recv) outperformed the one-sided communication methods for this particular problem size and configuration. This might be unexpected, but its performance depends heavily on the specific implementation, hardware, and problem characteristics.

All methods achieved identical numerical results. The solution converged after 1000 iterations with a final residual of 2.300506e+00
Performance varied between methods:

Standard MPI: 0.0057s solve time (fastest)
RMA fence: 0.0188s solve time (slowest)
RMA general active target: 0.0174s solve time

Performance Comparison:

Traditional MPI Send/Recv was about 3.3× faster than the RMA methods
The RMA general active target was slightly faster than RMA fence

Q2 Read Matrix and do a Matrix-Vector Product

Part 1: Reading Matrix and Vector with MPI-IO

1. **Matrix File (mat-d20-b5-p4.bin):**
   - **Format:**
     - First 4 bytes: integer 20 (dimension of Matrix).
     - Subsequent data: 4 block columns, each containing four 5x5 blocks (100 doubles total per block column).
   - **Reading:**

    - Each process reads its block column starting at offset 4 + rank *100* sizeof(double).

2. **Vector File (x-d20.txt.bin):**
    - **Format:**
        - First 4 bytes: integer 20 (vector length).
        - Subsequent data: 20 doubles.
    - **Reading:**
        - Each process reads 5 elements starting at offset 4 + rank *5* sizeof(double).

Part 2: Matrix-Vector Product

- **Local Computation:**
    - Each process multiplies its 20x5 block column with its 5-element vector segment.
- **Global Sum:**
    - Use `MPI_Reduce` to sum local results into the final 20-element vector on rank 0.

Verification

1. **Compile and Run:**
    - Use `mpicc -o ax ax.c` and run with 4 processes.
2. **Compare with Python/Matlab:**
    - Construct the full matrix and vector from the given data, compute `Ax`, and ensure the results match.

README Instructions

- **Compile:** `make` (ensure a Makefile is provided).
- **Run:** `mpirun -np 4 ./ax`.
- **Output:** The result vector is printed by rank 0. Compare with reference implementation. Ax = [235.0 252.0 282.0 316.0 194.0 226.0 263.0 189.0 210.0 192.0 130.0 147.0 163.0 188.0 254.0 245.0 191.0 187.0 197.0 221.0]

Python Code for Verification

```
import numpy as np
```

```python

def read_matrix_file(filename):
    """Read the matrix file in the specified block column format."""
    with open(filename, 'rb') as f:
        dim = np.fromfile(f, dtype=np.int32, count=1)[0]
        data = np.fromfile(f, dtype=np.float64)

    # Reshape into 4 block columns (each 20x5 = 100 elements)
    block_columns = []
    for bc in range(4):
        # Extract data for this block column
        bc_data = data[bc*100 : (bc+1)*100]
        # Split into four 5x5 blocks and vertically stack
        blocks = [bc_data[i*25 : (i+1)*25].reshape(5, 5) for i in
range(4)]
        block_col = np.vstack(blocks)  # Shape: 20x5
        block_columns.append(block_col)

    # Horizontally stack block columns to form 20x20 matrix
    return np.hstack(block_columns)

def read_vector_file(filename):
    """Read the vector file."""
    with open(filename, 'rb') as f:
        dim = np.fromfile(f, dtype=np.int32, count=1)[0]
        x = np.fromfile(f, dtype=np.float64)
    return x

# Read data
A = read_matrix_file('mat-d20-b5-p4.bin')
x = read_vector_file('x-d20.txt.bin')

# Compute Ax
Ax = A @ x

print("Result using Python:")
print(Ax)
```

Result using Python: Ax=[235. 252. 282. 316. 194. 226. 263. 189. 210. 192. 130. 147. 163. 188. 254. 245. 191. 187. 197. 221.]
and our C **Output:** result vector is printed by rank 0 are same.
Ax = [235.0 252.0 282.0 316.0 194.0 226.0 263.0 189.0 210.0 192.0 130.0 147.0 163.0 188.0 254.0 245.0 191.0 187.0 197.0 221.0]