Sparse Matrix Multiplication

Alok Ranjan
241010011

Department of Aerospace Engineering
IIT Kanpur

April 21, 2025

Serial: Core Mathematics

For matrices A and B, compute:

$$C_{ij} = \sum_{k=1}^{80} A_{ik} \times B_{kj}$$

Basic Optimization

Skip when $A_{ik} = 0$:

$$C_{ij} = \sum_{\substack{k=1 \\ A_{ik} \neq 0}}^{80} A_{ik} \times B_{kj}$$

- Operations Saved: 70% (from 512,000 to 153,600)
- Key Idea: Check before multiplying

Serial Code Explained

```
for (int i = 0; i < 80; i++) {          // For each output row
    for (int k = 0; k < 80; k++) {      // For each element
        if (A[i][k] != 0) {              //  zero-check
            for (int j = 0; j < 80; j++) {
                C[i][j] += A[i][k] * B[k][j];
}}}}
```

- **Loop Order**: i→k→j optimal for row-major memory
- **Why This Works**: Skips entire inner loops when $A_{ik} = 0$

OpenMP: Parallel Mathematics

Extend serial approach by distributing rows:

$$C_{\text{thread}} = \sum_{\text{assigned rows}} A_{\text{thread}} \times B$$

Implementation

- Divide A's rows equally among threads
- Each thread computes its portion independently

OpenMP Code

```
#pragma omp parallel for  //  parallel directive
for (int i = 0; i < 80; i++) {
    // Same serial logic:
    if (A[i][k] != 0) {           //  zero-check
        for (int j = 0; j < 80; j++) {
            C[i][j] += A[i][k] * B[k][j];
}}}
```

- How Distribution Works:
  - Thread 0: rows 0-19
  - Thread 1: rows 20-39
  - etc.
- Why Safe: Threads write to different $C[i]$ rows

MPI: Data Distribution Mechanics

Scattering Matrix A:

- `MPI_Scatter` splits A by rows
- Example (4 processes):
  - Process 0: rows 0-19
  - Process 1: rows 20-39
  - Process 2: rows 40-59
  - Process 3: rows 60-79
- Each gets `localA[20][80]`

Broadcasting Matrix B:

- `MPI_Bcast` sends full B to all
- Each process gets `B[80][80]`
- Why? All need full B for multiplication

MPI: Computation & Gathering

Local Computation:

- Each process calculates:

$$\texttt{localC} = \texttt{localA} \times B$$

- Uses same zero-skipping as serial
- Works on its 20-row chunk

Gathering Results:

- `MPI_Gather` collects `localC` chunks
- Reassembles full C at Rank 0
- Like puzzle pieces coming together

MPI Code

```c
// Rank 0 divides A
MPI_Scatter(A, 20*80, MPI_INT, localA, 20*80, MPI_INT, 0,
    MPI_COMM_WORLD);

// All get full B
MPI_Bcast(B, 80*80, MPI_INT, 0, MPI_COMM_WORLD);

//  local computation
for (i = 0; i < 20; i++) {
    if (localA[i][k] != 0) {  //  zero-check
        for (j = 0; j < 80; j++) {
            localC[i][j] += localA[i][k] * B[k][j];
}}}

// Rank 0 collects results
MPI_Gather(localC, 20*80, MPI_INT, C, 20*80, MPI_INT, 0,
    MPI_COMM_WORLD);
```

Execution Time Comparison

Implementation	2 Threads	4 Threads	8 Threads
Serial		0.0066 s	
OpenMP	0.002713 s	0.001528 s	0.004866 s
MPI	0.002752 s	0.002797 s	0.000853 s

Table: Execution times for different parallel implementations

- MPI with 8 threads showed the best performance (0.000853 s)
- OpenMP performance degraded with 8 threads (0.004866 s)
- Serial implementation was the slowest (0.0066 s)

Parallel and Serial Ax=b linear Solvers

Implementing Jacobi, Gradient Descent and PETSc

Jacobi Method - Algorithm

Mathematical Formulation

Update each variable using previous iteration values:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

Algorithm 1 Jacobi Iteration

```
1: Initialize $x^{(0)} = 0$
2: for $k = 1$ to max_iterations do
3:     for $i = 1$ to $n$ do
4:         $\sigma \leftarrow \sum_{j \neq i} a_{ij} x_j^{(k)}$
5:         $x_i^{(k+1)} \leftarrow (b_i - \sigma)/a_{ii}$
6:     end for
7:     if $\|x^{(k+1)} - x^{(k)}\| < \epsilon$ then break
8:
```

Jacobi - Serial Code

```
// Main Jacobi loop
for (k = 0; k < MAX_ITER; k++) {
    for (i = 0; i < N; i++) {
        sum = 0;
        for (j = 0; j < N; j++) {
            if (j != i) sum += A[i][j] * x[j];
        }
        x_new[i] = (b[i] - sum) / A[i][i];
    }
    error = compute_error(x_new, x);
    if (error < TOL) break;
    // Update for next iteration
    for (i = 0; i < N; i++) x[i] = x_new[i];
}
```

- Time: 0.000094 sec (16×16 matrix)
- Pros: Simple, easy to parallelize

Gradient Descent - Algorithm

Key Steps

1. Compute residual: $r = b - Ax$
2. Calculate step size: $\alpha = \frac{r^T r}{r^T A r}$
3. Update solution: $x = x + \alpha r$

Algorithm 2 Gradient Descent

```
1: $x \leftarrow 0$
2: repeat
3:     $r \leftarrow b - Ax$
4:     $\alpha \leftarrow (r^T r)/(r^T A r)$
5:     $x \leftarrow x + \alpha r$
6: until $\|r\| < \epsilon$ =0
```

Gradient Descent - Serial Code

```c
for (iter = 0; iter < MAX_ITER; iter++) {
    // Compute residual r = b - A*x
    mat_vec_mult(A, x, Ax);
    vector_sub(b, Ax, r);

    // Compute step size
    dot_rr = vector_dot(r, r);
    mat_vec_mult(A, r, Ar);
    alpha = dot_rr / vector_dot(r, Ar);

    // Update solution
    vector_add_scaled(x, alpha, r, x);

    if (sqrt(dot_rr) < TOL) break;
}
```

- Time: 0.000604 sec (16×16 matrix)
- Pros: Faster convergence for large systems

PETSc - Key Components

Matrix Setup

- `MatCreate()`: Creates matrix
- `MATAIJ`: Sparse format
- Preallocation: Critical for performance

Solver Setup

- `KSPCreate()`: Solver context
- `KSPCG`: Conjugate Gradient
- `KSPSetTolerances()`: Convergence

Why These Choices?

- Efficient for sparse systems
- Robust convergence
- Parallel-ready design

PETSc - Code Implementation

```c
// 1. Matrix Setup
MatCreate(PETSC_COMM_WORLD, &A);
MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, N, N);
MatSetType(A, MATAIJ); // Sparse format
MatMPIAIJSetPreallocation(A, 5, NULL, 5, NULL);

// 2. Vector Setup
VecCreate(PETSC_COMM_WORLD, &x);
VecSetSizes(x, PETSC_DECIDE, N);
VecDuplicate(x, &b);

// 3. Solver Configuration
KSPCreate(PETSC_COMM_WORLD, &ksp);
KSPSetType(ksp, KSPCG); // Conjugate Gradient
KSPSetTolerances(ksp, 1e-6, PETSC_DEFAULT, PETSC_DEFAULT,
    1000);

// 4. Solve System
KSPSetOperators(ksp, A, A);
KSPSolve(ksp, b, x);
```

PETSc - Behind the Scenes

- **Matrix Assembly**:
  - PETSc distributes matrix across MPI processes
  - Each process owns portion of rows
- **Solver Workflow**:
  - Builds Krylov subspace (CG: conjugate directions)
  - Applies preconditioner (default: block Jacobi)
  - Checks convergence criteria
- **Parallel Communication**:
- MPI handles inter-process data exchange
- Overlaps computation/communication

Execution Results

Implementation	Time (s)
Serial Jacobi	0.000094
Serial Gradient	0.000604
PETSc Jacobi	–
PETSc Gradient	–

- Jacobi: Fastest for small matrices
- Gradient: Better theoretical scaling
- PETSc: Implementation in progress

Thank You!