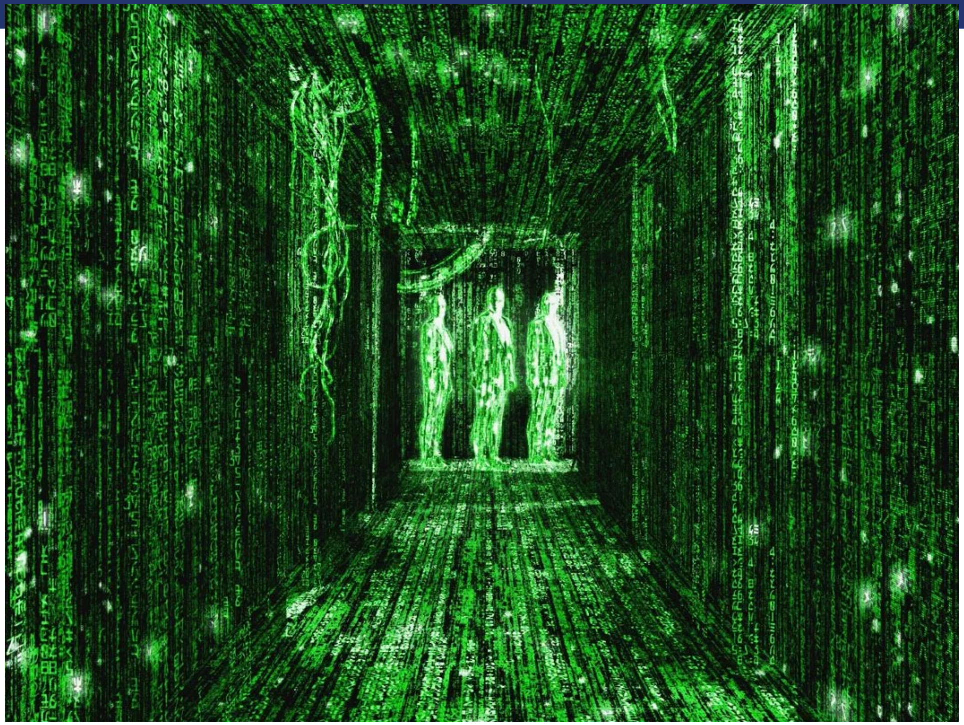
Parallel programming

Matrix Algorithms
in OpenMP and MPI



Today's topic

- Coding seminar
- Goals
  - Practice the theory from the lectures
  - Practice OpenMP and MPI
- 4 Tasks
  - Matrix multiplication (OpenMP)
  - LU factorization (OpenMP)
  - Gauss elimination (MPI)
  - Gauss elimination with cyclic row distribution (MPI)

Matrix multiplication

- Consider 2 matrix A and B and we want matrix C as
  - $C = A \cdot B$
- Matrix multiplication
  - Computational operations: $2n^3$
  - Memory operations: $3n^2$
- Naive algorithm might not be efficient
  - Too many memory operations
  - Cache size is limited
- If we are able to reuse data we can do something better
  - Use **blocks**!

Block matrix multiplication

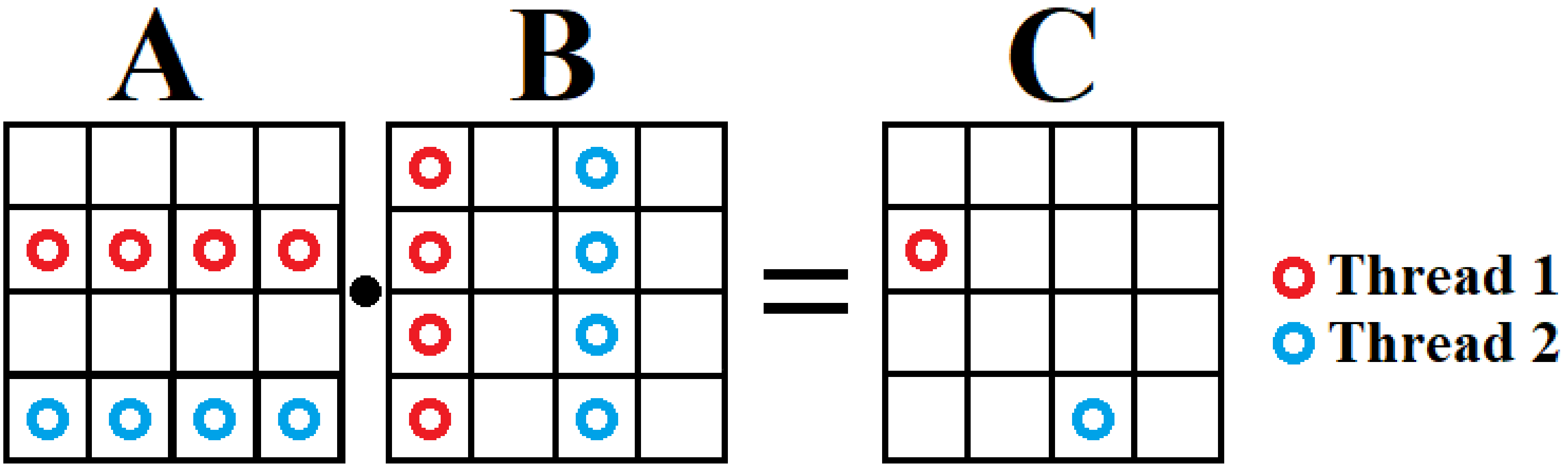
- We can divide A into blocks of row and B into block of columns
  - If rows and columns are too large, they won’t fit in the cache!
- Divide A and B into blocks of size $b \times b$

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$$

- Then $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31}$
  - Each $A_{ij} \cdot B_{ji}$ operation has $\mathbf{2b^2}$ memory operations and $\mathbf{2b^3}$ computational operations
- Chose *b* so that entire block can fit into the cache!

Parallel block matrix multiplication

- Using block matrix multiplication
- Use task to parallelize the algorithm
  - Beware of race conditions
  - Beware of correct data sharing among threads



Matrix Multiplication

`MatrixMultiplication.cpp`

- Open provided template and fill empty functions according to guidelines

LU Factorization

- LU factorization of matrix A
  - $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$
  - $\mathbf{L}$ is lower triangular matrix
  - $\mathbf{U}$ is upper triangular matrix
- Usefull for solving linear equations
  - $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$
  - $\mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b}$
  - $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ => get vector using backward triangular substitution
  - $\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$ => get vector using backward triangular substitution
- How we get L and U matrixes?
  - Gaussian ellimination
- Complexity:
  - Data are $\mathbf{O}(\mathbf{n}^2)$
  - Number of computations $\mathbf{O}(\mathbf{n}^3)$

LU Factorization example

Initialize $L = \mathrm{I}$ and $U = A$

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 2 & 3\\ 3 & 1 & 4\\ 5 & 3 & 1\end{pmatrix}$$

$R_2 \leftarrow R_2 - 3.R_1$

$R_3 \leftarrow R_3 - 5.R_1$

$$\begin{pmatrix}1 & 0 & 0\\ 3 & 1 & 0\\ 5 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 2 & 3\\ 3 & -5 & -5\\ 5 & -7 & -14\end{pmatrix}$$

$R_3 \leftarrow R_3 - 2{,}5.R_2$

$$\begin{pmatrix}1 & 0 & 0\\ 3 & 1 & 0\\ 5 & 2{,}5 & 1\end{pmatrix}\begin{pmatrix}1 & 2 & 3\\ 0 & -5 & -5\\ 0 & 0 & -7\end{pmatrix}$$

$$L = \begin{pmatrix}1 & 0 & 0\\ 3 & 1 & 0\\ 5 & 2{,}5 & 1\end{pmatrix}$$

$$U = \begin{pmatrix}1 & 2 & 3\\ 0 & -5 & -5\\ 0 & 0 & -7\end{pmatrix}$$

Block LU Factorization

- Block algorithm will be effiecient
  - Block distribution of matrixes:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} * \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix}$$

  - Blocks $\mathbf{A}_{00}, \mathbf{L}_{00}$ and $\mathbf{U}_{00}$ are of size $b \times b$
  - Blocks $\mathbf{A}_{10}, \mathbf{L}_{10}$ and $\mathbf{U}_{10}$ are of size $(n-b) \times b$
  - Blocks $\mathbf{A}_{01}, \mathbf{L}_{01}$ and $\mathbf{U}_{01}$ are of size $b \times (n-b)$
  - Blocks $\mathbf{A}_{11}, \mathbf{L}_{11}$ and $\mathbf{U}_{11}$ are of size $(n-b) \times (n-b)$
- It holds:
  - $\mathbf{L}_{00} \cdot \mathbf{U}_{00} = \mathbf{A}_{00}$
  - $\mathbf{L}_{10} \cdot \mathbf{U}_{00} = \mathbf{A}_{10}$
  - $\mathbf{L}_{00} \cdot \mathbf{U}_{01} = \mathbf{A}_{01}$
  - $\mathbf{L}_{10} \cdot \mathbf{U}_{01} + \mathbf{L}_{11} \cdot \mathbf{U}_{11} = \mathbf{A}_{11}$

Parallel LU Factorization

1. Compute $\mathbf{L}_{00}$ and $\mathbf{U}_{00}$
   - Factorizing $\mathbf{A}_{00} = \mathbf{L}_{00} \cdot \mathbf{U}_{00}$
2. Compute $\mathbf{U}_{01}$
   - $\mathbf{A}_{01} = \mathbf{L}_{00} \cdot \mathbf{U}_{01}$
   - $\mathbf{U}_{01}$ is full matrix and $\mathbf{L}_{00}$ is triangular matrix => Triangular solve
3. Compute $\mathbf{L}_{10}$
   - $\mathbf{A}_{10} = \mathbf{L}_{10} \cdot \mathbf{U}_{00}$
   - $\mathbf{U}_{00}$ is triangluar matrix and $\mathbf{L}_{10}$ is full matrix => Triangular solve
4. Update $\mathbf{A}'_{11}$ of $\mathbf{A}_{11}$ is set to
   - $\mathbf{L}_{11} \cdot \mathbf{U}_{11} = \mathbf{A}_{11} - \mathbf{L}_{10} \cdot \mathbf{U}_{01} = \mathbf{A}'_{11}$
   - $\mathbf{L}_{10} \cdot \mathbf{U}_{01}$ is matrix multiplication that can be done in parallel
5. Recursively solve $\mathbf{A}'_{11} = \mathbf{L}_{10} \cdot \mathbf{U}_{01}$

LU Decomposition

`LUDecomposition.cpp`

- Open provided template and implement parallel LU factorization of matrix A using OpenMP

Gauss Elimination

- Usefull for solving system of linear equations
  - Row reduction
  - Can also be used to compute the rank of a matrix, the determinant of a square matrix, and the inverse of an invertible matrix
- Sequence of row operations
  - Multiplying a row by a nonzero number
  - Adding a multiple of one row to another row
- Each row operation need **pivot** row that defines the multiplying coefficients

Gauss Elimination Pseudocode

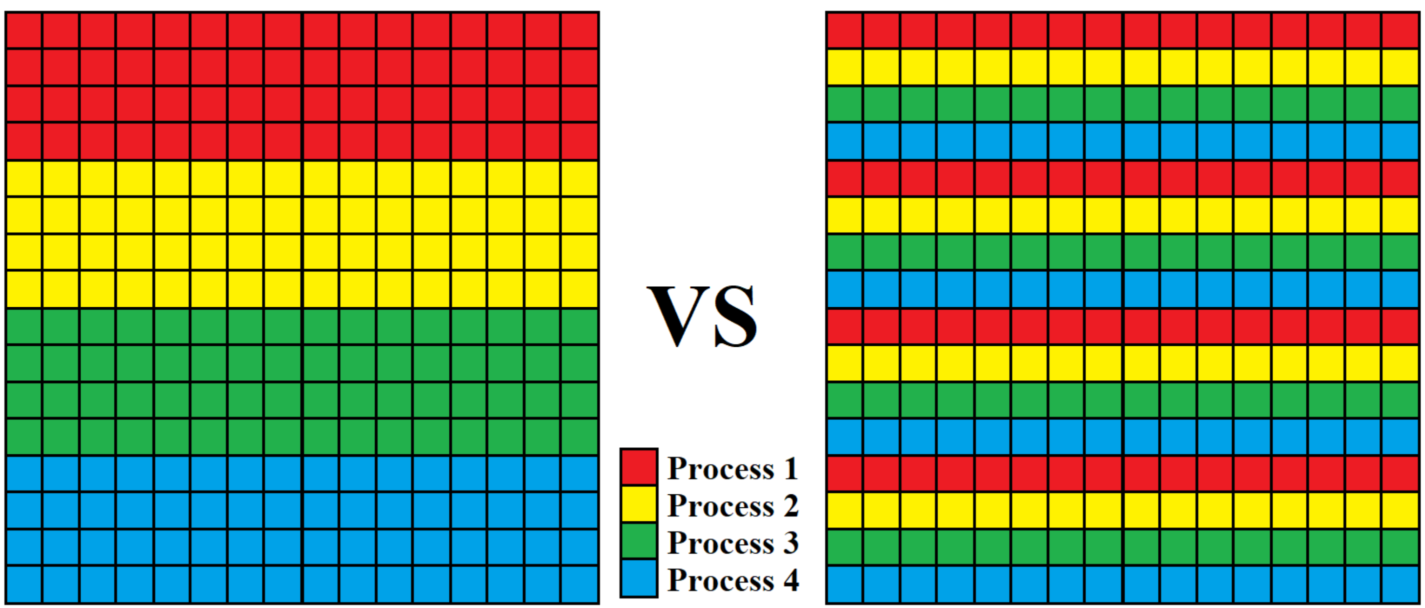
```
for k = 1 to (n-1)
    for i = (k+1) to n
        factor = A(i, k) / A(k, k)
        for j = k to n
            A(i, j) = A(i, j) - factor * A(k, j)
        end for
        b(i) = b(i) - factor * b(k)
    end for
end for


for i = n to (step-1)
    x(i) = b(i)
    for j = i+1 to n
        x(i) = x(i) - A(i, j) * x(j)
    end for
    x(i) = x(i) / A(i, i)
end for
```

Distributed Gauss Elimination

1. Scatter the matrix rows
2. For each iteration select pivot row
3. Processor with pivot row in k-th iteration perform operation on pivot row to get 1 at k-th position
4. Processor with pivot row broadcasts the pivot row
5. Perform row reduction for rows under the pivot row to get 0 at k-th position
6. Repeat steps 2.-5. until get to last row
7. Gather the updated rows at processor 0

Row distribution for distributed Gauss Elimination

- Using naive distribution may not be the efficient method
  - After process update all its rows, it won‘t do any work
- Using cyclic row distribution
  - More efficient (processes will be working almost until the end)



Gauss Elimination

`GaussEliminationBlock.cpp`

- Open provided template and implement parallel Gauss Elimination with **block row distribution** using MPI. Follow provided guidelines.

`GaussEliminationCyclic.cpp`

- Open provided template and implement parallel Gauss Elimination with **cyclic row distribution** using MPI. Follow provided guidelines.