Dense Matrix Algorithms

KUMAR Chapter 8
Overview

1. Matrix-vector Multiplication
2. Matrix-matrix Multiplication
Utility of Matrix Algorithms

Applied in several numerical and non-numerical contexts:

- 3D image calculations
- Solving (linear) equations
- Simulations of physical systems

E.g.: makes the basis of LINPACK, a software library for performing numerical linear algebra, by using the BLAS (Basic Linear Algebra Subprograms) libraries for performing basic vector and matrix operations.
Dense versus Sparse Matrices

**Dense matrices**: have no or few known zero entries

**Sparse matrices**: are populated primarily with zeros

- often appear in science or engineering when solving partial differential equations
- easily compressed
- very large sparse matrices are impossible to manipulate with the standard algorithms, due to memory limitations
- special versions of the algorithms are necessary and are more efficient
Matrix – Vector Multiplication

\[ C = A \times V \]

\[ C_i = \sum_{k=1}^{n} A_{ik} \cdot V_k \quad (i:1..n) \]

\[ T_{seq} = \delta_{mm} \cdot n^2 \]

for (i=0; i<n; i++) {
    C[i] = 0;
    for (k=0; k<n; k++) {
        C[i] += A[i, k] * V[k];
    }
}

\( n \): number of elements in the vector, number of rows and columns of the matrix
Matrix/Vector Partitioning?

- **Row-wise block striping**
  - $p_1$
  - $p_2$
  - $p_3$
  - $p_4$

- **Column-wise block striping**
  - $p_1$
  - $p_2$
  - $p_3$
  - $p_4$

- **Checkerboard partitioning**
  - $p_1$
  - $p_2$
  - $p_3$
  - $p_4$
  - $p_5$
  - $p_6$
  - $p_7$
  - $p_8$
  - $p_9$

- **Vector partitioning**
  - $p_1$
  - $p_2$
  - $p_3$
  - $p_4$
Data & results distributed

- In parallel applications, data remains distributed while operations (such as vector-matrix operations) are performed on them.

- In the following we assume that the data is already distributed among the processors
Parallel Matrix Multiplication Version 1

\((n=p)\)

Figure 8.1 Multiplication of an \(n \times n\) matrix with an \(n \times 1\) vector using rowwise block 1-D partitioning. For the one-row-per-process case, \(p = n\).
Parallel M x V Multiplication
Version 1  \( p < n \)

\( \frac{n}{p} \) rows of matrix and \( \frac{n}{p} \) elements of vector per processor
Parallel M x V Multiplication Version 2

\( n^2 = p \)

(a) Initial data distribution and communication steps to align the vector along the diagonal

(b) One-to-all broadcast of portions of the vector along process columns

checkboard partitioning

(c) All-to-one reduction of partial results

(d) Final distribution of the result vector

Figure 8.2 Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, \( p = n^2 \) if the matrix size is \( n \times n \).
Parallel M x V Multiplication
Version 2  \( p < n^2 \)

\((n/\sqrt{p}) \times (n/\sqrt{p})\) blocks of matrix and \(n/\sqrt{p}\) elements of vector per processor
Overview

1. Matrix-vector Multiplication

2. Matrix-matrix Multiplication
Matrix Multiplication

\[ C = A \times B \]

\[ C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj} \quad (i, j: 1..n) \]

\[ T_s = \delta_{mm} \cdot n^3 \]

```c
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        C[i,j]=0;
        for (k=0; k<n; k++){
            C[i,j]+=A[i, k]*B[k,j];
        }
    }}
```
MxM: one-step version

- B is sent to all processors
- Computation in 1 step
- Same amount of communication
Alternate shift-compute version

- Algorithm alternates p computation and communication steps
- Computation step: each processor multiplies its A submatrix with its B submatrix, resulting in a submatrix of C. The black circles indicate the step in which each submatrix is computed.
- After multiplication: processor sends its B submatrix to the next processor and receives one from the preceding processor. The communication forms a circular shift operation.
Parallel MxM: Execution Profile

Speedup = 2.55  Efficiency = 85%
Theoretical Analysis of Matrix Multiplication

- Computation time
  \[ T_{work}^i = \frac{n^3}{p} \delta_{mm} \]

- Communication time (slave)
  \[ T_{comm}^i = (p+1)t_s + (1 + \frac{2}{p}).n^2.t_w \]

- Explains parameter dependence of performance:
  - Total overhead = O(n^2.p)
  - Computation = O(n^3/p)
  - Overhead ratio = O(p/n)
  - Ratio computation/communication = 2n/p
Parameter Dependence of Matrix Multiplication

$n$: work size, here: matrix size
V3: Cannon’s algorithm

Checkerboard partitioning
Elements of A and B Needed to Compute a Process’s Portion of C

Algorithm 1 & 2

Cannon’s Algorithm

Why faster? ....
Cannon’s parallel $M \times M$ Communication steps on 16 processes
Initially, blocks Need to Be Aligned

Each triangle represents a matrix block

Only same-color triangles should be multiplied
Rearrange Blocks

Block $A_{ij}$ cycles left $i$ positions

Block $B_{ij}$ cycles up $j$ positions
Consider Process $P_{1,2}$

Step 1:
Computation & shift
Consider Process $P_{1,2}$

Step 2
Consider Process $P_{1,2}$

Step 3
Consider Process $P_{1,2}$

Step 4
Complexity Analysis

- Algorithm has $\sqrt{p}$ iterations
- During each iteration process multiplies two $(n / \sqrt{p}) \times (n / \sqrt{p})$ matrices: $\Theta(n^3 / p^{3/2})$
- **Computational complexity**: $\Theta(n^3 / p)$ [the same]
- During each iteration process sends and receives two blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$
- **Communication complexity**: $\Theta(n^2/ \sqrt{p})$ [lower!]
Efficient Interconnection Networks for Cannon’s MxM?

- line
- ring
- star
- tree
- mesh
- hypercube
- wraparound mesh
Memory need of each processor

As a function of $n$, $p$ and $b$ (the number of bytes per element)

- Sequential algorithm:
- One-step algorithm:
- Alternate algorithm:
- Cannon’s algorithm:
Special MPI functions

- **MxV version 2**: Reduce operation
- **Cannon**: Shift operation through a SendRecv_replace call

**Submatrix sending:**
- Consist of equally spaced blocks
- `DataType.Vector(int count, int blocklength, int stride, Datatype oldtype)`
Initially, matrices are copied to GPU
- If they are not still in memory from previous matrix operations, keep pointers in CPU to the data on the GPU

Every thread computes 1 element of C.

Not enough memory to put all data in shared memory (16K)

On one multiprocessor, 1 block of threads computes 1 block of the C matrix

Iteratively copy A row blocks and B column blocks to shared memory.

Result: 200x speedup, 50x if compared to quadcore

Example: GPUmat toolbox in Matlab