Parallel and Distributed Algorithms and Programs
TP n°4 - Parallel Matrix-Matrix Multiplication using Cannon's Algorithm

Oguz Kaya
oguz.kaya@ens-lyon.fr

Pierre Pradic
pierre.pradic@ens-lyon.fr

14/12/2016

In this session, we will be dealing with the implementation of the parallel matrix-matrix multiplication of two matrices using Cannon's algorithm. The advantage of Cannon's algorithm over SUMMA is that it incurs significantly less communication, and is very well suited for 2D communication networks. For simplicity, in this exercise, we will only be multiplying $N \times N$ matrices A and B to obtain another $N \times N$ matrix C. We will use $P = p \times p$ processors, and assume for simplicity that p divides N. In Cannon's algorithm, the matrices A, B, and C are split into $p \times p$ submatrices each of size $(N/p) \times (N/p)$. For instance, for $p = 2$, A is split into submatrices A_{11}, A_{12}, A_{21}, and A_{22} of size $N/2 \times N/2$ each. In general, each process with index (i, j) (or p_{ij}) holds the corresponding submatrices A_{ij}, B_{ij}, and C_{ij} initially. The goal of p_{ij} is to compute the final result for the submatrix C_{ij}.

The Cannon's method for matrix multiplication is given in Algorithm 1. The algorithm uses the procedure

`SHIFT(matrix, shiftSize, shiftDirection)`

that shifts the `matrix` by `shiftSize`-many processors in the given `shiftDirection` (which is `'up'` or `'left'`). For instance, if the statement SHIFT(A, `2, 'left'`) is called at each process, then each process p_{ij} would receive the local matrix A from the process $p_{i,(j-2 \bmod p)}$, and send its matrix to the process $p_{i,(j+2 \bmod p)}$. Figure 1 represents the shift operations (including the initial shift) involved in the Cannon's algorithm.

Algorithm 1 Cannon's parallel matrix-matrix multiplication algorithm

Input: A, B, C: Matrices of size $N \times N$
$P = p \times p$: The number of processes available
Output: $C = AB$ is computed.

1: Distribute matrices so that the process p_{ij} owns the matrices A_{ij}, B_{ij}, and C_{ij}.
2: At each process p_{ij}, SHIFT(A, `i, 'left'`)
3: At each process p_{ij}, SHIFT(B, `j, 'up'`)
4: At each process p_{ij}, initialize $A \leftarrow A_{ij}$ and $B \leftarrow B_{ij}$.
5: **for** $k = 1 \dots p$ **do** ▶ Do at each process p_{ij}
6: $C_{ij} = C_{ij} + AB$.
7: SHIFT(A, `1, 'left'`)
8: SHIFT(B, `1, 'up'`)

As usual, we provide two codes `cannon-skeleton.c` and `cannon-solution.c` for this exercise. You are supposed to provide your implementations in the given function skeletons in `cannon-solution.c`.

Question 1

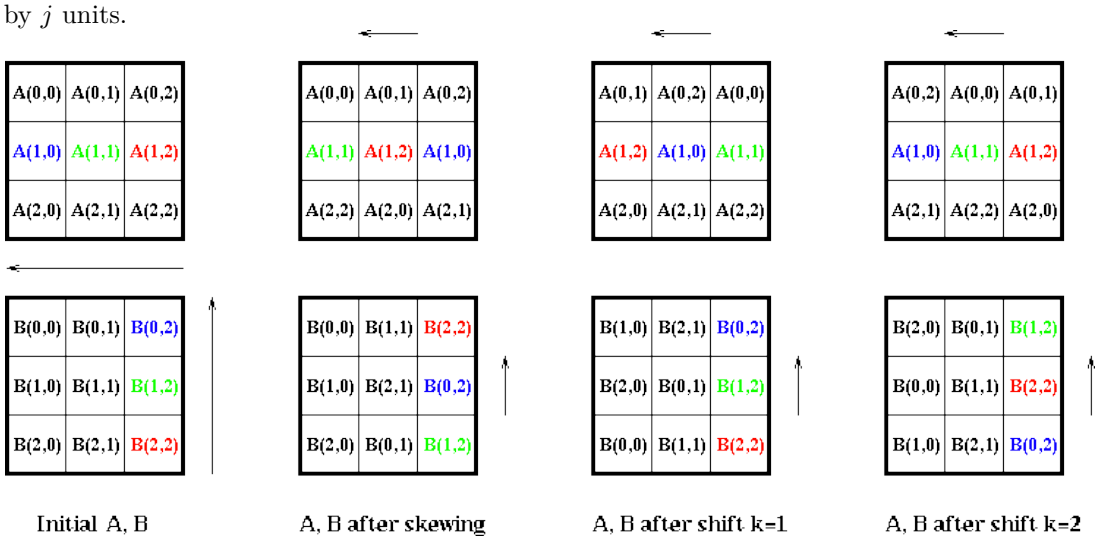
a) Similar to the previous TP, at each process create the local matrices `Aloc`, `Bloc`, and `Cloc` of size N/p. Make sure to initialize `Aloc` and `Bloc` randomly, and `Cloc` with zeros. Compute also the row index and the column index of each process.

b) Implement the given SHIFT operator that shifts the matrices among processes in the given direction by the given amount. Make sure you use `MPI_Isend` for sends, and `MPI_Recv` for receives to prevent deadlocks.

c) Using the `Shift` function, implement the Cannon's algorithm. Make sure to test it using different number of processors, and different matrix sizes. Use the provided function

```
multiplyMatrix(double *a, double *b, double *c, int m, int k, int n)
```

Figure 1: Shift operation in the Cannon's algorithm. "Skewing" corresponds to the initial shift where the row/column j is shifted by j units.



for local matrix-matrix multiplies.

d) Measure the performance of your implementation using SMPI for $N = 1024$ and P up to 64 (using a 8×8 processor grid). How well does your algorithm scale? Compare the performance to the SUMMA implementation that we did in the previous TP. Try to change the network bandwidth, and see when the algorithm starts to lose scalability.

Make sure to backup all your implementations as they might be useful later on!