Parallel Algorithms and Programming

Parallel Algorithms in Distributed Memory Systems (Part 1)

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

In this lecture

- Performance of parallel distributed algorithms
- Parallel operations on vectors and matrices
    - Matrix-vector multiplication
    - Matrix-matrix multiplication

Introduction

Introduction

Assumptions

- A distributed memory system
    - Processes communicate by send and receiving messages
- A **virtual ring** topology

Main costs to consider

- The cost of running computation
    - floating point operations in our context
- The cost of moving data
    - from the memory to the processors
    - between nodes (over the interconnection network)

Cost of a distributed parallel algorithm

- A computational term:

$$\text{nb of flops} \times \text{time per flop}$$

- A bandwidth term:

$$\text{amount of data moved} \times \frac{1}{bandwidth}$$

- A latency term:

$$\text{nb of messages} \times latency$$

Cost of a distributed parallel algorithm

We should remember that (in general):

$$\text{time per flop} \ll \frac{1}{bandwidth} \ll latency$$

Consequence

- Minimizing communication is important for performance

- It is also important for energy efficiency
    - Moving data from/to DRAM or over the interconnection network are the most energy consuming operations

About memory accesses

Cost of moving data between the memory and the processor

- Can often be ignored
    - Cost mostly hidden by hardware prefetchers

Prefetchers

- Hardware mechanisms to load data in the cache before it is needed
- Based on the memory access pattern of the program
- Works well for regular access patterns
    - It is the case for the programs studied in this lecture

Matrix-vector multiplication

Specification of the problem

We consider:

- A is a matrix of size $n \times n$
- x, y are vectors of size n

We want to compute:

$$y = Ax$$

Specification of the problem

$$\mathrm{Ax} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

Sequential implementation

```
for(i=0; i<n; i++){
    y[i] = 0;
    for(j=0; j<n; j++){
        y[i] = y[i] + A[i][j] * x[j]
    }
}
```

Parallel implementation

Observations

Parallel implementation

Observations

- The computation of each value of y is the scalar product between a row of A and the vector x.
- Each scalar product is independent

Parallel algorithm

Parallel implementation

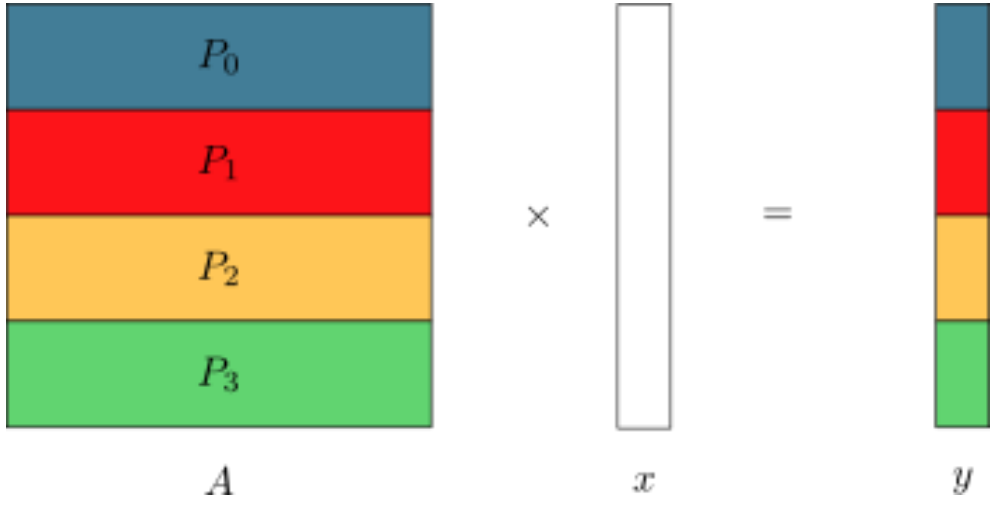
Observations

- The computation of each value of y is the scalar product between a row of A and the vector x.
- Each scalar product is independent

Parallel algorithm

- Distribute the rows of A over p processors
    - Each processor has a copy of x
- Each processor computes $r = \frac{n}{p}$ scalar products in parallel

Parallel implementation



- Distributed matrix-vector multiplication over 4 processors.
- All data with the same color are on the same processor

Improving performance and solving larger problems

Impact of distributing the data and the computation over several nodes

- Improve the execution time (hopefully)

- Solve larger problems
    - Matrix `A` might not fit in the memory of a single node
    - Distributing the rows of matrix `A` over `p` nodes allows storing a much larger matrix in memory

Making the algorithm more generic

Initial distribution of the blocks of x

- An execution scenario:
    - $y = Ax$ followed by $z = By$
    - At the begining of the second computation, blocks of `y` are distributed over the nodes (see previous figure)

A generic matrix-vector multiplication function

- Function that implements $y = Ax$
    - With both `A` and `x` distibuted over the nodes

Algorithm in a virtual ring topology

Basic steps

Algorithm in a virtual ring topology

Basic steps

1. Computation step:
    - Each processor computes a partial result using the elements of x it has in its local memory
2. Communication step:
    - Each processor sends its block of x to its successor and receives from its predecessor in the virtual ring

Algorithm in a virtual ring topology

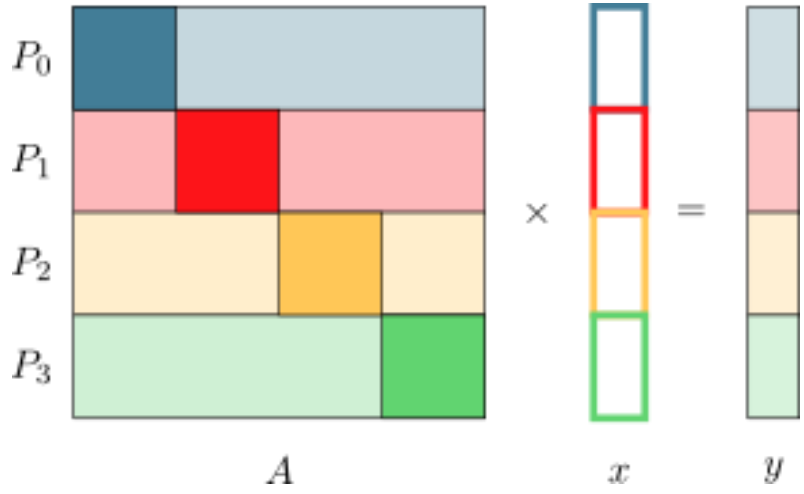
Basic steps

1. Computation step:
    - Each processor computes a partial result using the elements of x it has in its local memory
2. Communication step:
    - Each processor sends its block of x to its successor and receives from its predecessor in the virtual ring

Number of iterations to terminate: p

Algorithm in a virtual ring topology

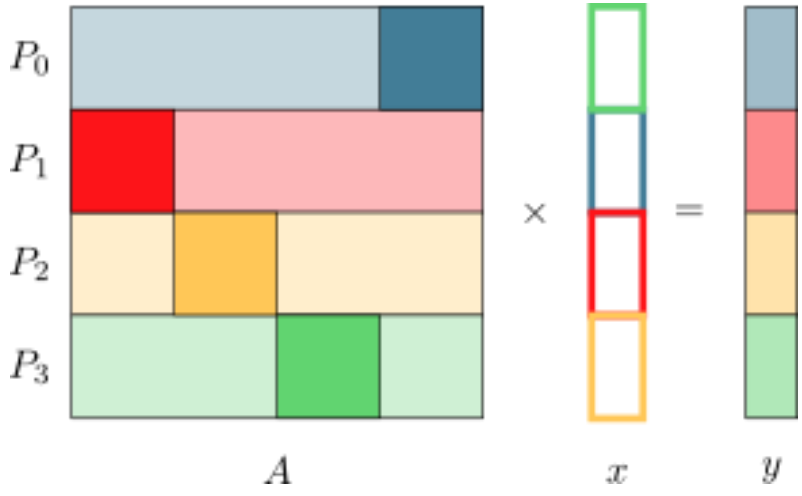
Computation step 0



- Distributed matrix-vector multiplication over 4 processors
  - Each processor stores $\frac{n}{4}$ rows of A
  - Each processor stores $\frac{n}{4}$ values of x.
- Colors for blocks of vector x refer to the initial position of the blocks.

Algorithm in a virtual ring topology

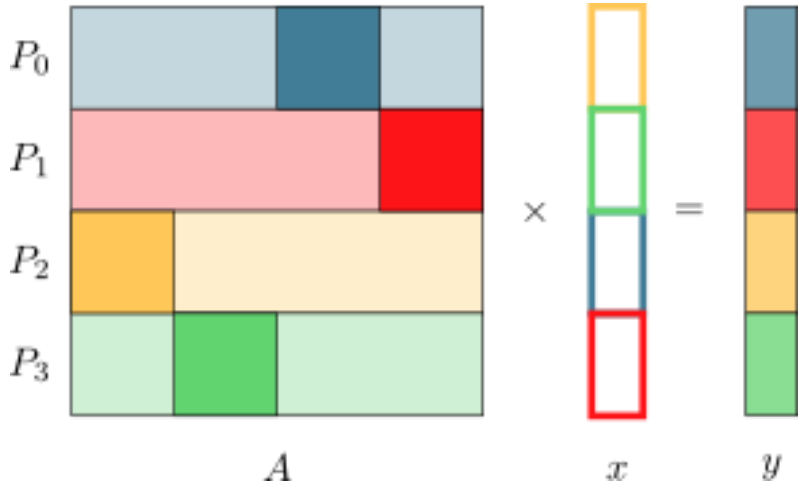
Computation step 1



- Blocks of x have been *shifted* in the virtual ring
    - Focus on the color of the blocks

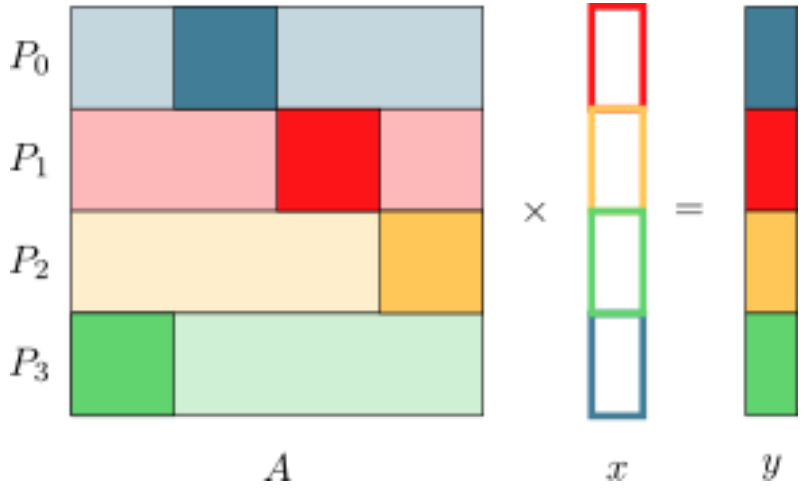
Algorithm in a virtual ring topology

Computation step 2



Algorithm in a virtual ring topology

Computation step 3



- The last round of communication will bring x back into its initial state

Algorithm in pseudo C code

```
double A[r][n]; /* rows of A, assumed to be already initialized*/
double x[r]; /* values of x, assumed to be already initialized*/
double y[r];

int rank = my_rank();
int P = num_procs();

double tempS[r], tempR[r];
```

Algorithm in pseudo C code

```
tempS <- x;  /* copy of the values */
for(step = 0; step < P; step++){
    Send(tempS, (rank+1) % P);
    ||
    Recv(tempR, (rank-1) % P);
    ||
    int block = (rank-step) % P;
    for(i=0; i<r; i++){
        for(j=0; j<r; j++){
          y[i] = y[i] + A[i, r * block + j)] * tempS[j]
        }
    }
    tempS <- tempR;
}
```

Algorithm in pseudo C code

Computing the indexes of elements of `A` to use at each step

- On which block of A, processor `K` should compute at step S?
  - Vector `x` is divided in `p` blocks
  - At step 0, processor `K` computes using block `K` of `x`
  - At step 1, processor `K` computes using block `K-1` of `x`
  - At step 2, processor `K` computes using block `K-2` of `x`

Algorithm in pseudo C code

Computing the indexes of elements of `A` to use at each step

- On which block of A, processor `K` should compute at step S?
    - Vector `x` is divided in `p` blocks
    - At step 0, processor `K` computes using block `K` of `x`
    - At step 1, processor `K` computes using block `K-1` of `x`
    - At step 2, processor `K` computes using block `K-2` of `x`

- Computation of the block index: $block = (rank - step) \mod p$
- Computation of the indexes in a row of A

Algorithm in pseudo C code

Computing the indexes of elements of `A` to use at each step

- On which block of A, processor `K` should compute at step S?
    - Vector `x` is divided in `p` blocks
    - At step 0, processor `K` computes using block `K` of `x`
    - At step 1, processor `K` computes using block `K-1` of `x`
    - At step 2, processor `K` computes using block `K-2` of `x`

- Computation of the block index: $block = (rank - step) \mod p$
- Computation of the indexes in a row of A
    - $r = \frac{n}{p}$ (p is the number of processors)
    - start index: $block \times r$
    - end index: $(block + 1) \times r - 1$

Algorithm in pseudo C code

About the execution in parallel

- Algorithm designed to allow the communication and the computation inside one step to occur in parallel (symbol `||`)
    - Requires to allocate an extra buffer (`tempR`) for the communication
    - The communication in the last iteration allows restoring the initial distribution of the blocks of `x`

Implementation with MPI

- How to overlap communication and computation?

Algorithm in pseudo C code

About the execution in parallel

- Algorithm designed to allow the communication and the computation inside one step to occur in parallel (symbol `||`)
    - Requires to allocate an extra buffer (`tempR`) for the communication
    - The communication in the last iteration allows restoring the initial distribution of the blocks of `x`

Implementation with MPI

- How to overlap communication and computation?
    - Use of MPI_Isend() and MPI_Irecv(), plus MPI_Wait()

Execution time of the algorithm

Parameters

- `p` processors
- matrix of size $n \times n$
- $r = \frac{n}{p}$

Model parameters

- `L`: the latency of a network communication
- `B`: the bandwidth of a network communication
- `w`: the computation time for one basic unit of work
    - = two floating point operations in this algorithm

Execution time of the algorithm

Execution time of the algorithm

- Time for a communication: $T_{comm} = L + \frac{r}{B}$
    - We assume that time to receive of message = time to send a message

Execution time of the algorithm

- Time for a communication: $T_{comm} = L + \frac{r}{B}$
    - We assume that time to receive of message = time to send a message
- Time for computation in one step: $T_{comp} = r^2 \times w$
    - Computation on a block of size $r \times r$

Total time

Execution time of the algorithm

- Time for a communication: $T_{comm} = L + \frac{r}{B}$
    - We assume that time to receive of message = time to send a message
- Time for computation in one step: $T_{comp} = r^2 \times w$
    - Computation on a block of size $r \times r$

Total time

- p iterations
- Computation occur in parallel with communication

Execution time of the algorithm

- Time for a communication: $T_{comm} = L + \frac{r}{B}$
    - We assume that time to receive of message = time to send a message
- Time for computation in one step: $T_{comp} = r^2 \times w$
    - Computation on a block of size $r \times r$

Total time

- p iterations
- Computation occur in parallel with communication

$$T_{MV}(p) = p \times \max(T_{comp}, T_{comm})$$

$$T_{MV}(p) = p \times \max(r^2 \times w, L + \frac{r}{B}) = \max(\frac{n^2}{p} \times w, p \times L + \frac{n}{B})$$

Performance for large matrices

When `n` becomes large, the execution time is asymptotically equal to:

$$T_{MV}(p) = \frac{n^2}{p} \times w$$

Performance for large matrices

When `n` becomes large, the execution time is asymptotically equal to:

$$T_{MV}(p) = \frac{n^2}{p} \times w$$

Speedup of p with p processors

Distribution of the data

- **Case 1**: The rows of `A` are already distributed between the processors
    - `A` is the result of a previous computation
    - `A` is loaded from a file on network file system
        - Parallel IO libraries (e.g., MPI-IO) allow processors to load data in parellel

- **Case 2**: The data are initially on one node and should be aggregated on a single node at the end of the computation

Distribution of the data

- **Case 1**: The rows of `A` are already distributed between the processors
    - `A` is the result of a previous computation
    - `A` is loaded from a file on network file system
        - Parallel IO libraries (e.g., MPI-IO) allow processors to load data in parellel

- **Case 2**: The data are initially on one node and should be aggregated on a single node at the end of the computation
    - Use of MPI_Scatter() and MPI_Gather() functions

Matrix-matrix multiplication

Specification of the problem

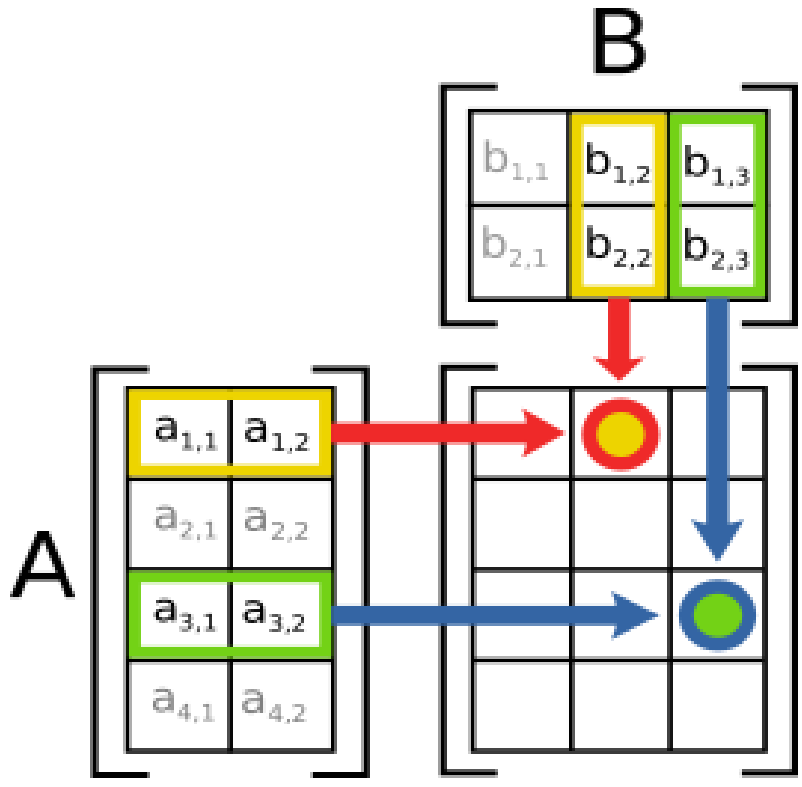
We consider:

- A, B, C are matrices of size $n \times n$

We want to compute:

$$C = A \times B$$

Specification of the problem



$$C_{3,3} = A_{3,1} \times B_{1,3} + A_{3,2} \times B_{2,3}$$

Source: Wikipedia (Bilou)

Sequential implementation

```
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] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

Parallel implementation

Same assumptions as before

- `p` processes
- A *virtual ring* topology

Basic idea of the algorithm

- Same idea as for matrix-vector multiplication

Algorithm in a virtual ring topology

Description of the algorithm

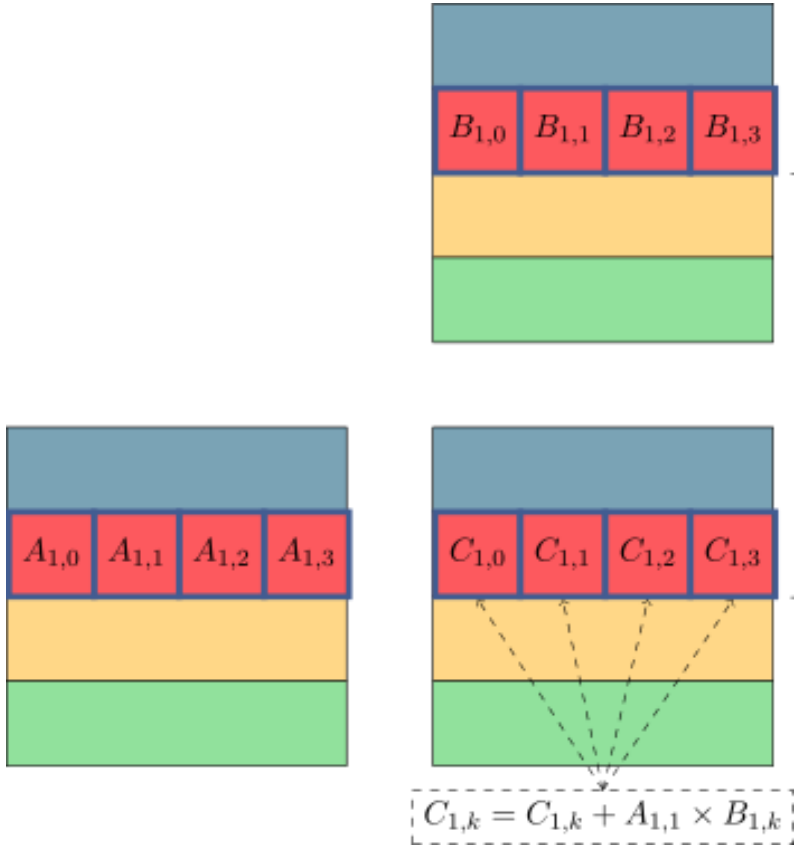
- The 3 matrices are distributed over the processes based on the rows.
    - Each process stores $r = \frac{n}{p}$ rows of each matrix.

- In each iteration:
    1. Each processor runs a partial matrix-matrix multiplication based on the data that are available locally.
    2. Each processor sends r rows of B to the next processor and receives r rows of B from the previous processor

The algorithm requires p iterations to terminate.

Algorithm in a virtual ring topology

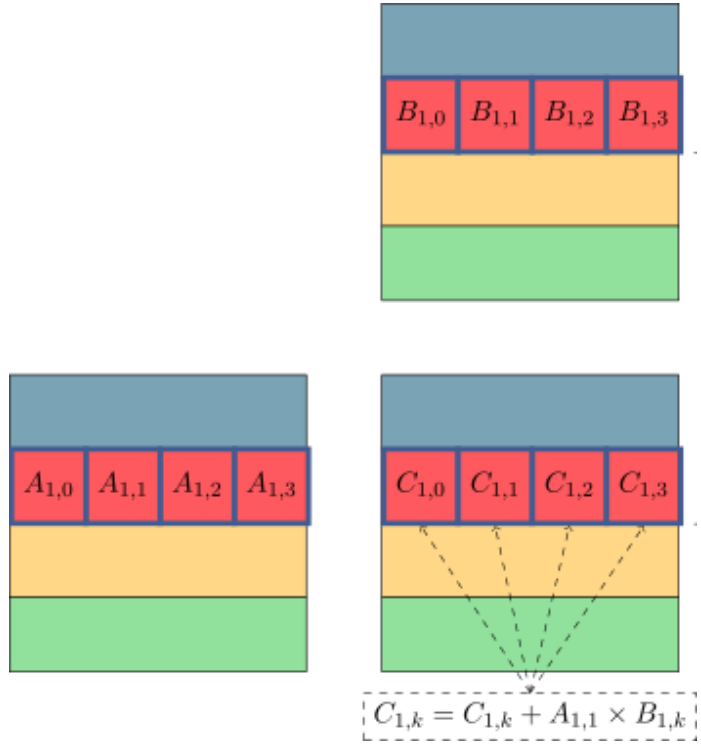
Computation step 0



- The figure assumes 4 processors (4 colors)
    - The figure focuses on the computation run by processor P_1.
- The notation $A_{l,k}$ refers to the **block** k in the part of matrix A stored by processor l

Algorithm in a virtual ring topology

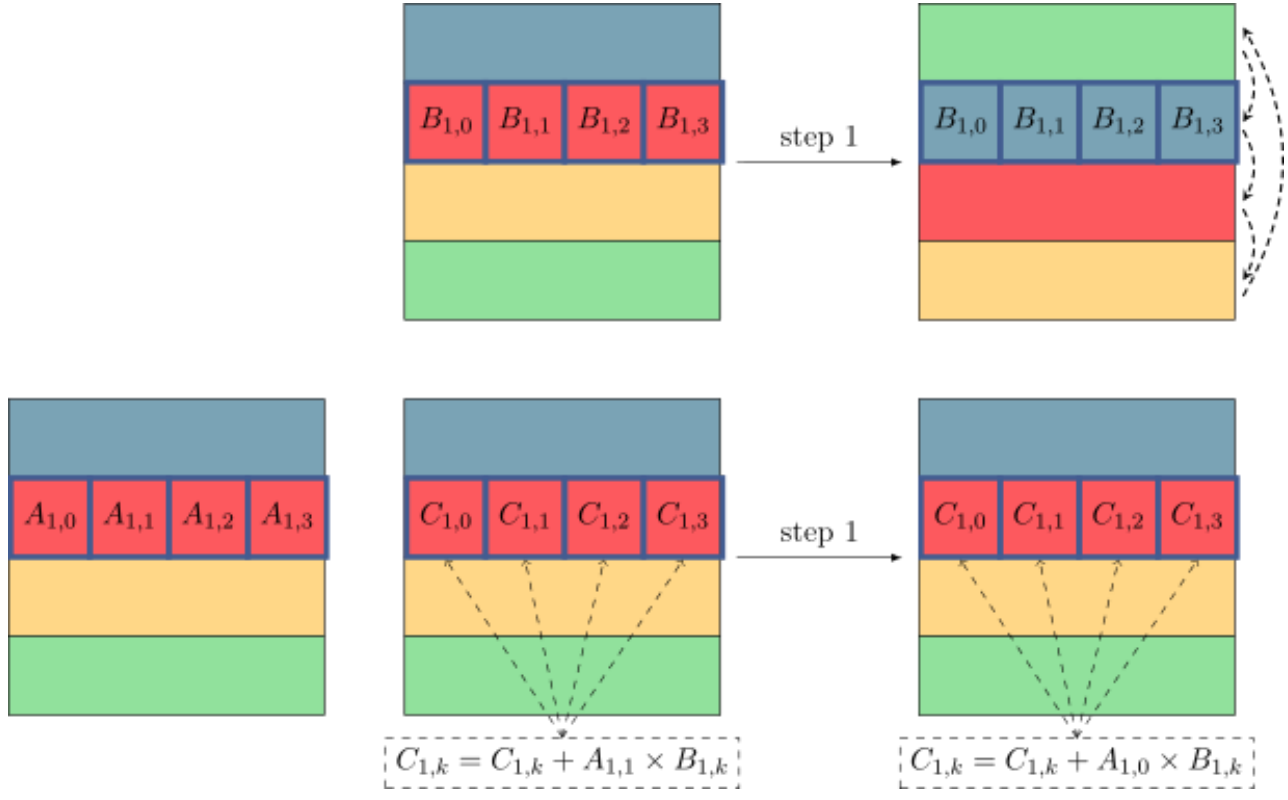
Computation step 0



- In step 0, processor P_k has rows $[r \times k]$ to $[r \times (k+1) - 1]$ of B
  - P_1 has rows r to $2r - 1$ of B
- P_k can only compute based on elements $[r \times k]$ to $[r \times (k+1) - 1]$ of A in each row (block $A_{k,k}$)

Algorithm in a virtual ring topology

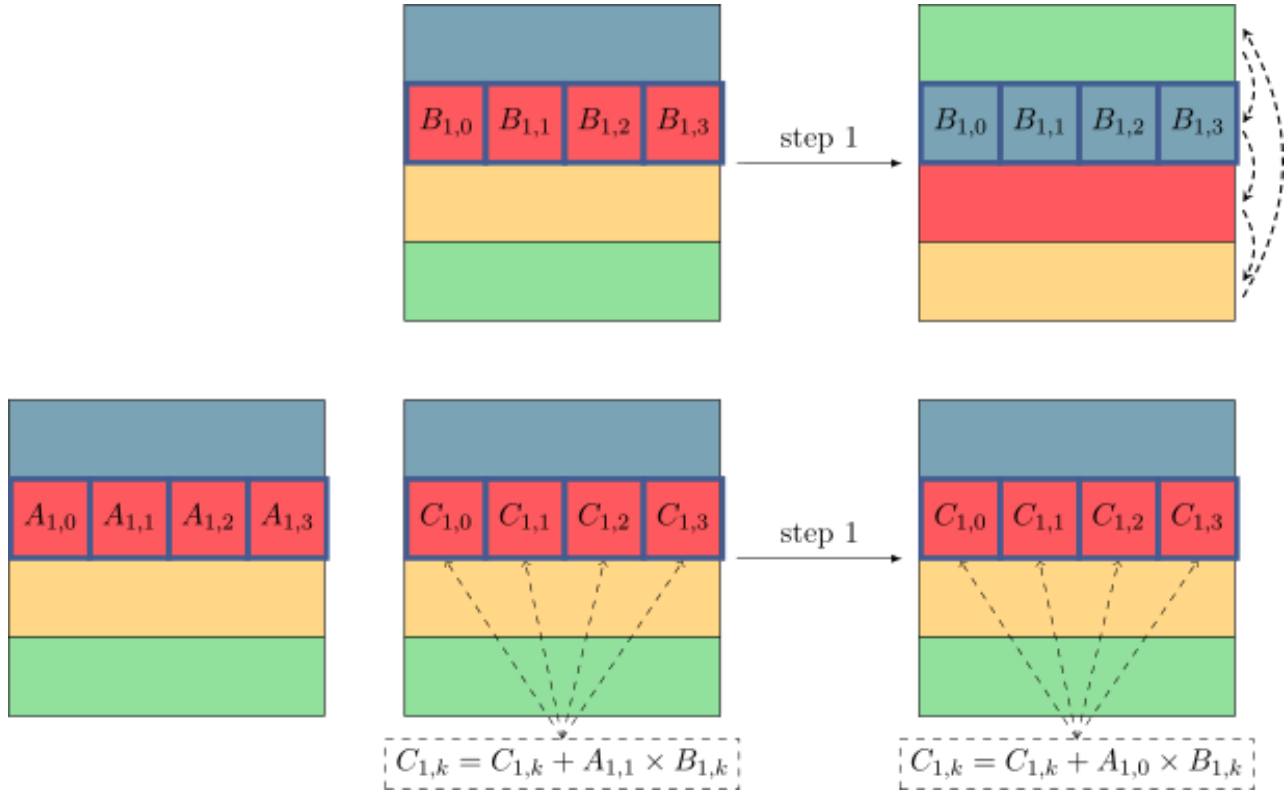
Computation step 1



- During step 0, processors exchange their rows of `B` in the virtual ring
- Processors keep the same rows of A during the whole algorithm

Algorithm in a virtual ring topology

Computation step 1



- In step 1, processor P_k has rows $[r \times (k-1)]$ to $[r \times k - 1]$ of B
  - P_1 has rows 0 to $r - 1$ of B

Algorithm in pseudo C code

```
double A[r][n]; /* rows of A, assumed to be already initialized*/
double B[r][n]; /* rows of B, assumed to be already initialized*/
double C[r][n];

int rank = my_rank();
int P = num_procs();

double tempS[r][n], tempR[r][n];
```

Algorithm in pseudo C code

```
tempS <- B;  /* copy of the values */
for(step = 0; step < P; step++){
    Send(tempS, (rank+1) % P);
    ||
    Recv(tempR, (rank-1) % P);
    ||
    int block = (rank-step) % P;
    for(l=0; l<P; l++){
        for(i=0; i<r; i++){
            for(j=0; j<r; j++){
                for(k=0; k<r; k++){
                    C[i][l * r + j] = C[i][l * r + j]
                    + A[i][block * r + k] * tempS[k][l * r + j]$;
    }}}}
    tempS <- tempR;
```

The additional index (l) is introduced to iterate over the blocks of B and C.

Execution time of the algorithm

Parameters

- `p` processors
- matrix of size $n \times n$
- $r = \frac{n}{p}$

Model parameters

- `L`: the latency of a network communication
- `B`: the bandwidth of a network communication
- `w`: the computation time for one basic unit of work
    - = two floating point operations also in this algorithm

Execution time of the algorithm

Execution time of the algorithm

- Time for a communication: $T_{comm} = L + \frac{n \times r}{B}$
    - Processors send and receive r rows of B

Execution time of the algorithm

- Time for a communication: $T_{comm} = L + \frac{n \times r}{B}$
    - Processors send and receive r rows of B
- Time for computation in one step: $T_{comp} = n \times r^2 \times w$
    - Computation on a block of size $n \times r$ with r operations for each value

Total time

- p iterations
- Computation occur in parallel with communication

$$T_{MM}(p) = p \times \max(T_{comp}, T_{comm})$$

$$T_{MM}(p) = p \times \max(n \times r^2 \times w, L + \frac{n \times r}{B}) = \max(\frac{n^3}{p} \times w, p \times L + \frac{n^2}{B})$$

Performance for large matrices

When `n` becomes large, the execution time is asymptotically equal to:

$$T_{MM}(p) = \frac{n^3}{p} \times w$$

Performance for large matrices

When `n` becomes large, the execution time is asymptotically equal to:

$$T_{MM}(p) = \frac{n^3}{p} \times w$$

Speedup of p with p processors

About Bulk communication

Observation:

- The parallel matrix-matrix multiplication could be implemented as n matrix-vector multiplications (one for each column of B)

Total time

$$T_{MM-alt}(p) = n \times T_{MV}(p)$$

$$T_{MM-alt}(p) = \max(\frac{n^3}{p} \times w, n \times p \times L + \frac{n^2}{B})$$

About Bulk communication

T_{MM} **vs** T_{MM-alt}

$$T_{MM}(p) = \max(\frac{n^3}{p} \times w, p \times L + \frac{n^2}{B})$$

$$T_{MM-alt}(p) = \max(\frac{n^3}{p} \times w, n \times p \times L + \frac{n^2}{B})$$

About Bulk communication

T_{MM} vs T_{MM-alt}

$$T_{MM}(p) = \max(\frac{n^3}{p} \times w, p \times L + \frac{n^2}{B})$$

$$T_{MM-alt}(p) = \max(\frac{n^3}{p} \times w, n \times p \times L + \frac{n^2}{B})$$

- The asymptotic efficiency is the same

- The latency term is multiplied by n with the alternative version
  - Can have a big impact on performance in practice
  - Relates to the fact that instead of exchanging blocks of matrices, processors would exchange blocks of vectors

Sending data in bulk

- General technique to reduce the cost of communicating over the network in parallel algorithms
  - Reduces the impact of latency.

References

- Section 4.1 and 4.2 of the book "Parallel Algorithms" (by Casanova, Robert, and Legrand).