CME 213

SPRING 2019

Eric Darve

This lecture

- Performance metrics
- Parallel matrix-matrix product

Main debugging techniques

- A whole class would be needed on this topic!
- Use assert: test conditions on variables: equality, inequality, magnitude
- Print out, but asserts are fundamentally better
- Talk to your duck
- Use theoretical results, e.g., convergence rate
- Test against reference code
- Manufactured solution: compare against known solutions
- Incremental changes: test after each change (regression testing)
- Unit/module tests
- Test inputs of increasing difficulty; code coverage
- Simplify problem to minimal buggy example
- Dichotomy, divide-and-conquer
- Ask piazza, and your TAs/instructor.

Performance metrics

Why performance metrics?

Understanding the performance of a code is important:

- to develop efficient code
- understand the bottlenecks of a code
- compare algorithms in a meaningful way, e.g., matrix-vector products using different partitioning schemes for the matrix.

The total runtime $T_p(n)$ can be broken down, generally speaking, into the following categories:

- Local computations
- Data exchange and communication
- Idle time (load imbalance, spurious synchronization)

The basic concept: speed-up

- Define:
  $$T^\star(n)$$
  the optimal (reference) running time with a single process.
- Define:
  $$T_p(n)$$
  the running time with p processes.
- The speed-up is then the ratio:

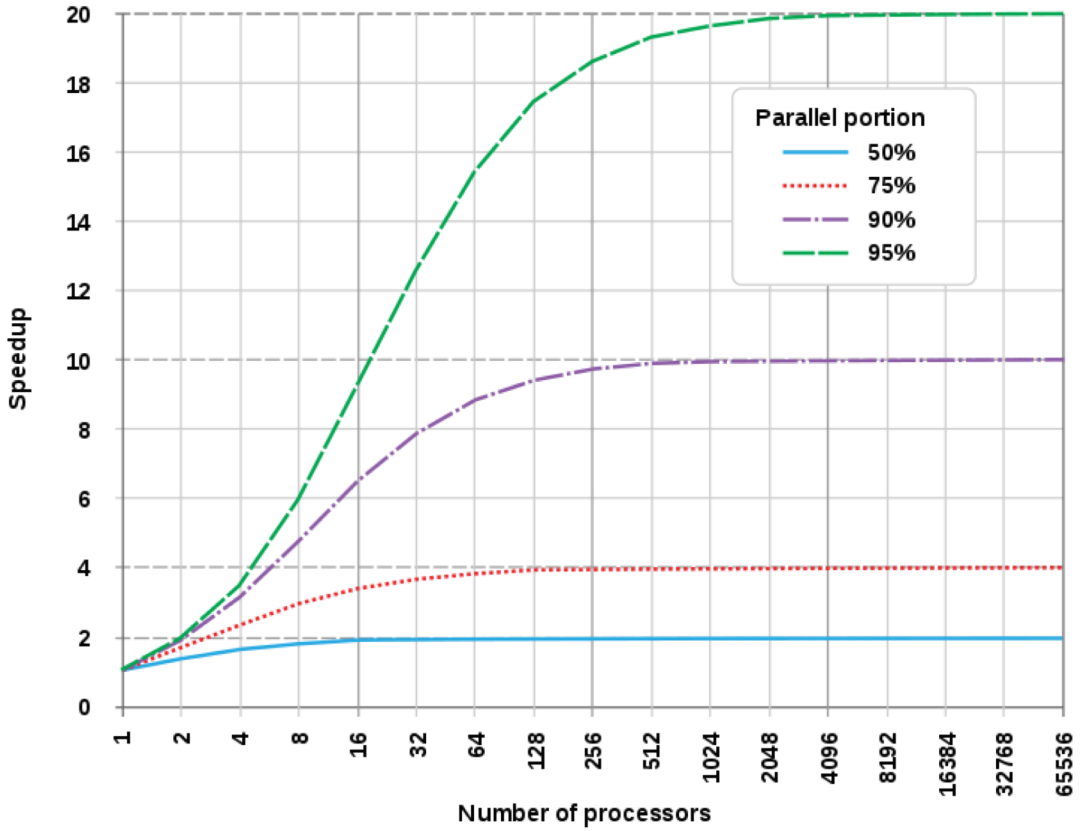
$$S_p(n) = \frac{T^\star(n)}{T_p(n)}$$

- We expect this number to go up as p as we keep increasing the number of processes.

Amdahl's law

If the sequential fraction is f. The speed-up is upper bounded:

$$S_p(n) \leq \frac{T^\star(n)}{fT^\star(n) + ((1-f)/p)\,T^\star(n)} = \frac{1}{f + (1-f)/p} \leq \frac{1}{f}$$



Amdahl's law

- If the sequential fraction is f. The speed-up is upper bounded.

$$S_p(n) \leq \frac{T^\star(n)}{fT^\star(n) + ((1-f)/p)\,T^\star(n)} = \frac{1}{f + (1-f)/p} \leq \frac{1}{f}$$

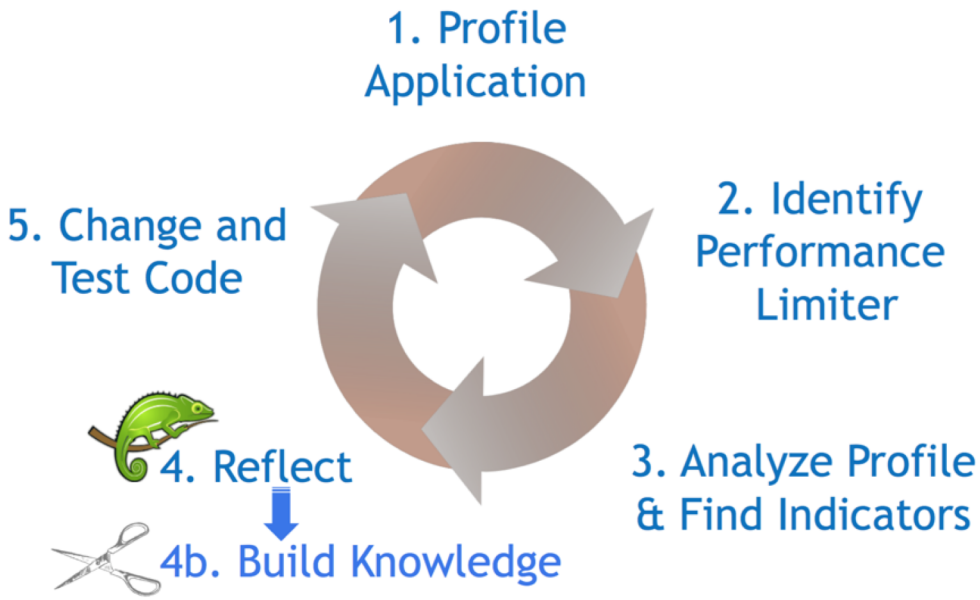
- However we are saved by another fact: f often decreases with n.
- This is why we can still benefit from Peta and Exascale machines with 100,000+ cores.

Profiling and CUDA computing

- A similar situation arises with CUDA.

1. Profile your code. Kernel 1: 90%
2. Port kernel 1 to GPU. Kernel 1: 10%. Kernel 2 becomes 80% now.

- Remember to iterate between profiling and optimizing your code.



A more appropriate concept: efficiency

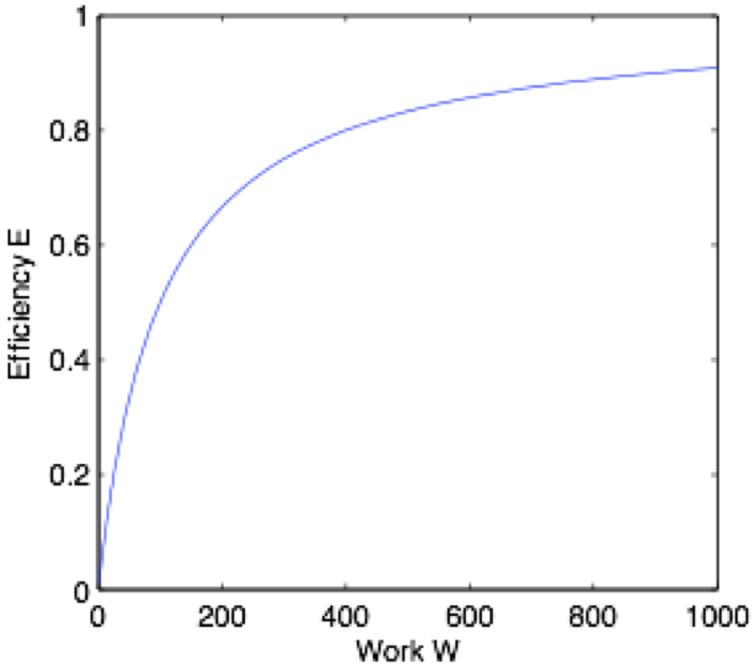
- Ideally, the speed-up is a straight line w.r.t. p.
- It is therefore more convenient to look at the efficiency:

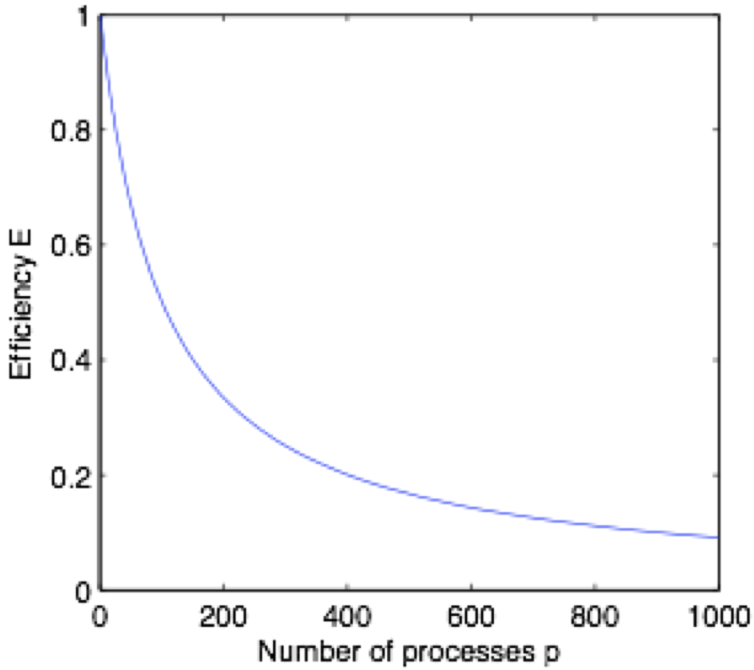
$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^\star(n)}{pT_p(n)}$$

- Ideally, a constant w.r.t. p (easier to read from a plot).
- The maximum value for efficiency is 1 (except in some rare circumstances because of cache effects).

Efficiency plots

Typical behavior of the efficiency w.r.t. n and p.





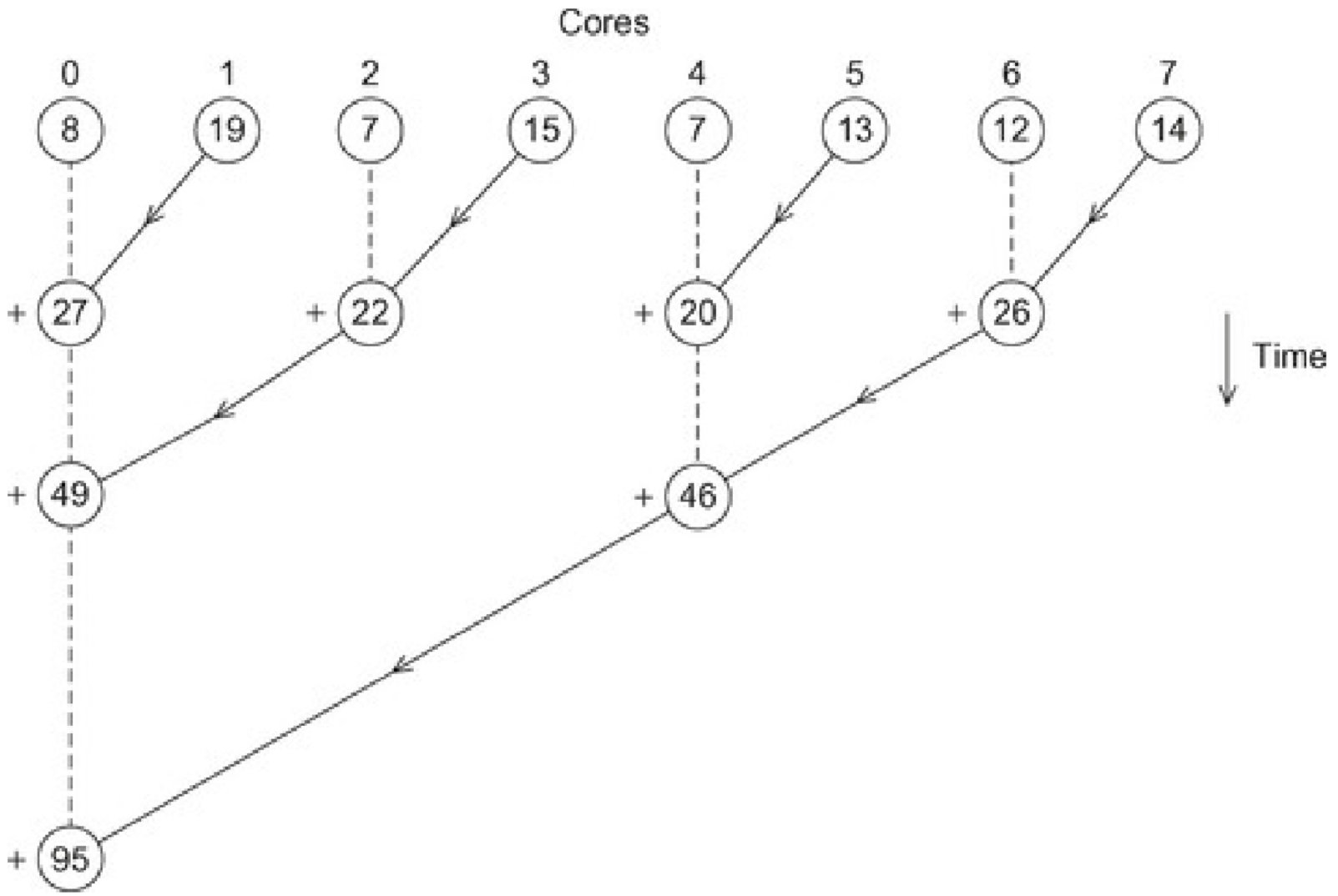
Example 1: dot product

Two-step algorithm:

- Calculate local dot product:

$$\sum_j a_j \; b_j$$

- Use a spanning tree for the final reduction: $\ln_2$ p passes are required.


Cores
0
1
2
3
4
5
6
7
8
19
7
15
7
13
12
14
+ 27
+ 22
+ 20
+ 26
Time
+ 49
+ 46
+ 95

Example 1: dot product

- Total run time with one process:

$$T^{\star}(n) = \alpha n$$

- Total run time in parallel:

$$T_p(n) = \alpha n/p + \beta \ln_2 p$$

Efficiency & Iso-efficiency

- Efficiency:

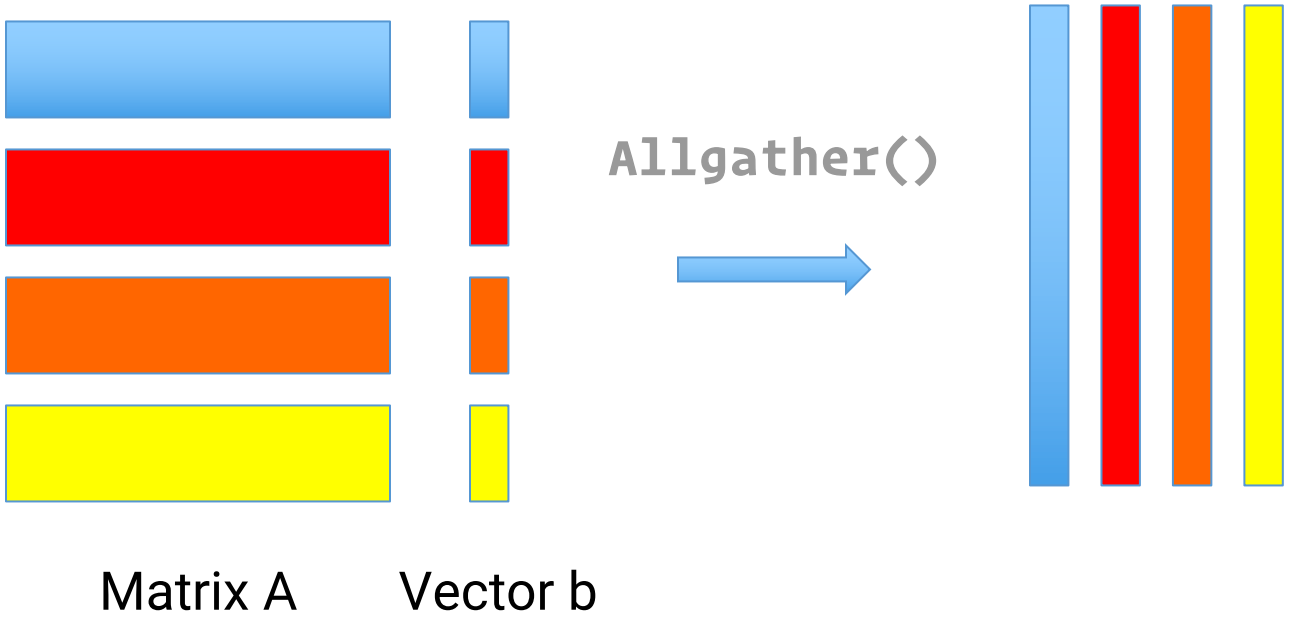
$$E_p(n) = \frac{\alpha n}{\alpha n + \beta\, p \log_2 p} = \frac{1}{1 + (\beta/\alpha)\,(p \log_2 p)/n}$$

- Iso-efficiency means that p increases at a rate such that the efficiency remains constant.
- Dot product:

$$p \log_2 p = \Theta(n) \quad \text{or} \quad p = \Theta(n/\log_2 n)$$

- Good algorithms: p can be increased rapidly at iso-efficiency.

Example 2: matrix-vector product with 1D partitioning



- Step 1: replicate b on each process: `MPI_Allgather()`
- Step 2: perform product

Summary of communication times

Operation	Hypercube time
One-to-all broadcast All-to-one reduction	$\min\{ (t_s + t_w m) \log p, 2(t_s \log p + t_w m) \}$
All-to-all broadcast All-to-all reduction	$t_s \log p + t_w m (p-1)$
All-reduce	$\min\{ (t_s + t_w m) \log p, 2(t_s \log p + t_w m) \}$
Scatter, Gather	$t_s \log p + t_w m (p-1)$
All-to-all personalized	$(t_s + t_w m) (p-1)$
Circular shift	$t_s + t_w m$

- m: size of message
- p: number of processes
- ts: latency
- tw: reciprocal bandwidth

Example 2: matrix-vector product with 1D partitioning

- Serial running time:

$$T^{\star}(n) = \alpha n^2$$

- Parallel running time (comp + comm):

$$T_p(n) = \alpha n^2/p + \beta \ln p + \gamma n$$

- Efficiency:

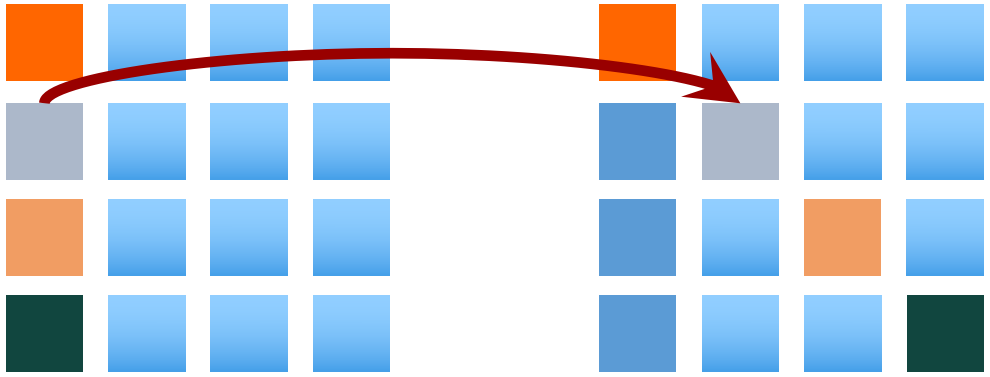
$$E_p(n) = \frac{1}{1 + (\beta/\alpha)(p \ln p)/n^2 + (\gamma/\alpha)p/n}$$

- Iso-efficiency requires:

$$p = \Theta(n)$$

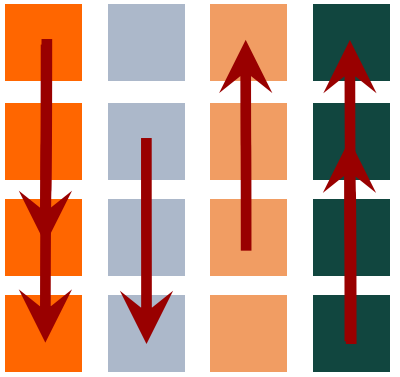
- Ideally, p increases roughly like n^2 (the amount of work).

Example 2: matrix-vector product with 2D partitioning

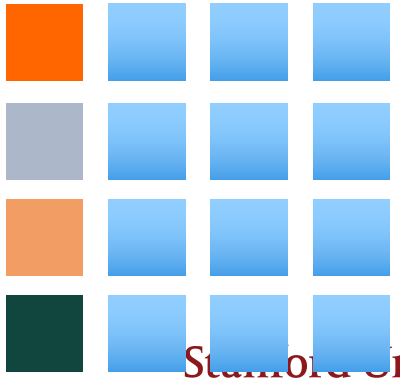


First column contains b

Send b to the diagonal processes



Send b down each column.



Example 2: matrix-vector product with 2D partitioning

- Computation:

$$\alpha n^2/p$$

- Send b to diagonal processes:

$$\beta + \gamma\, n/\sqrt{p}$$

- Broadcast b in each column:

$$(\beta + \gamma\, n/\sqrt{p}) \ln \sqrt{p}$$

- Reduction across columns (same running time as broadcast because these operations are dual of one another):

$$(\beta + \gamma\, n/\sqrt{p}) \ln \sqrt{p}$$

Operation	Hypercube time
One-to-all broadcast All-to-one reduction	$\min\{ (t_s + t_w m) \log p, 2(t_s \log p + t_w m) \}$

Iso-efficiency

- With the previous results:

$$E_p(n) = \frac{1}{1 + (\beta/\alpha)(p \ln p)/n^2 + (\gamma/\alpha)(p^{1/2} \ln p)/n}$$

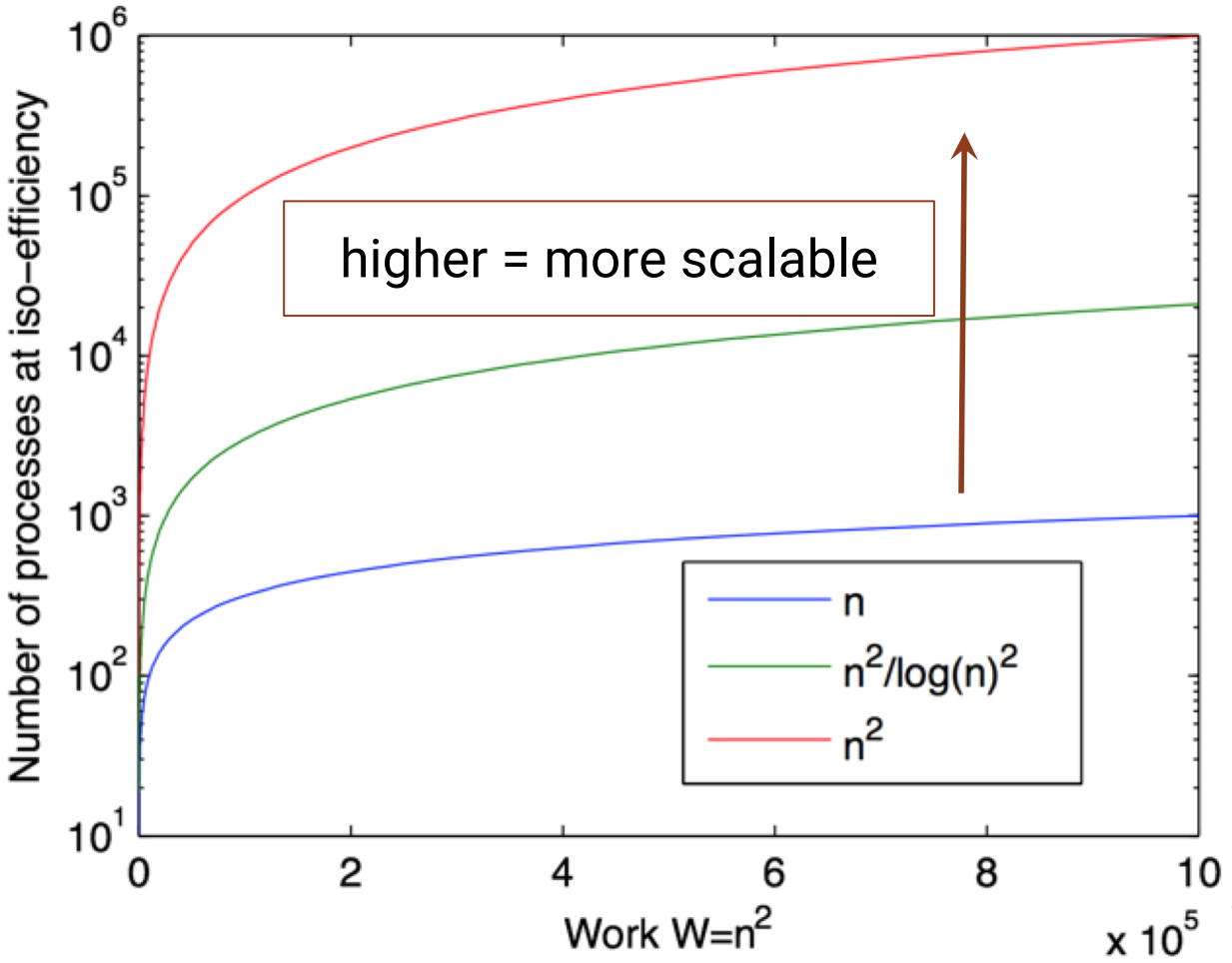
- Iso-efficiency:

$$p = \Theta\left(n^2/(\ln n)^2\right) \gg \Theta(n$$

- This is much better than with the 1D partitioning.
- We can increase p more rapidly at iso-efficiency.
- Another interpretation is that for a given number of processes, this scheme is faster. Practically, it has less communication.

Iso-efficiency plots

- Plot p as a function of n at iso-efficiency.
- Larger values of p are better (improved scalability).



Matrix-matrix products

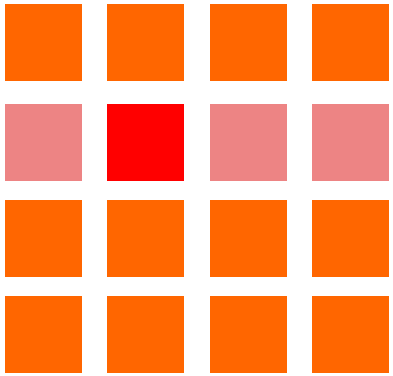
Matrix-matrix products

Algorithm in pseudo-code:

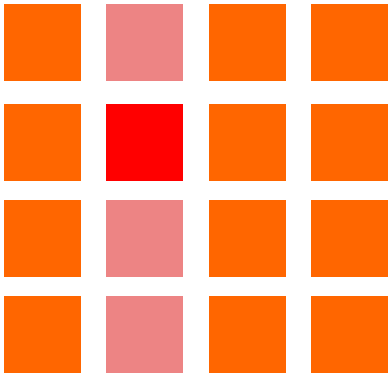
```
for i=0:n-1 do
   for j=0:n-1 do
    C(i,j) = 0;
    for k=0:n-1
        C(i,j) += A(i,k) * B(k,j);
    end
   end
end
```

Naïve block operations

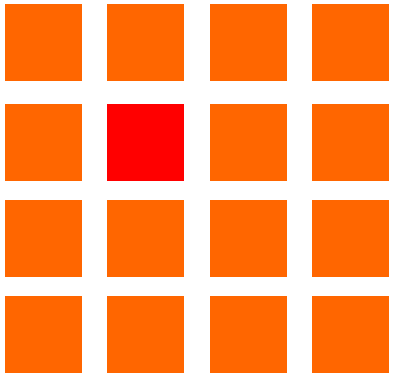
- Algorithm proceeds by doing block operations.



Matrix A



Matrix B



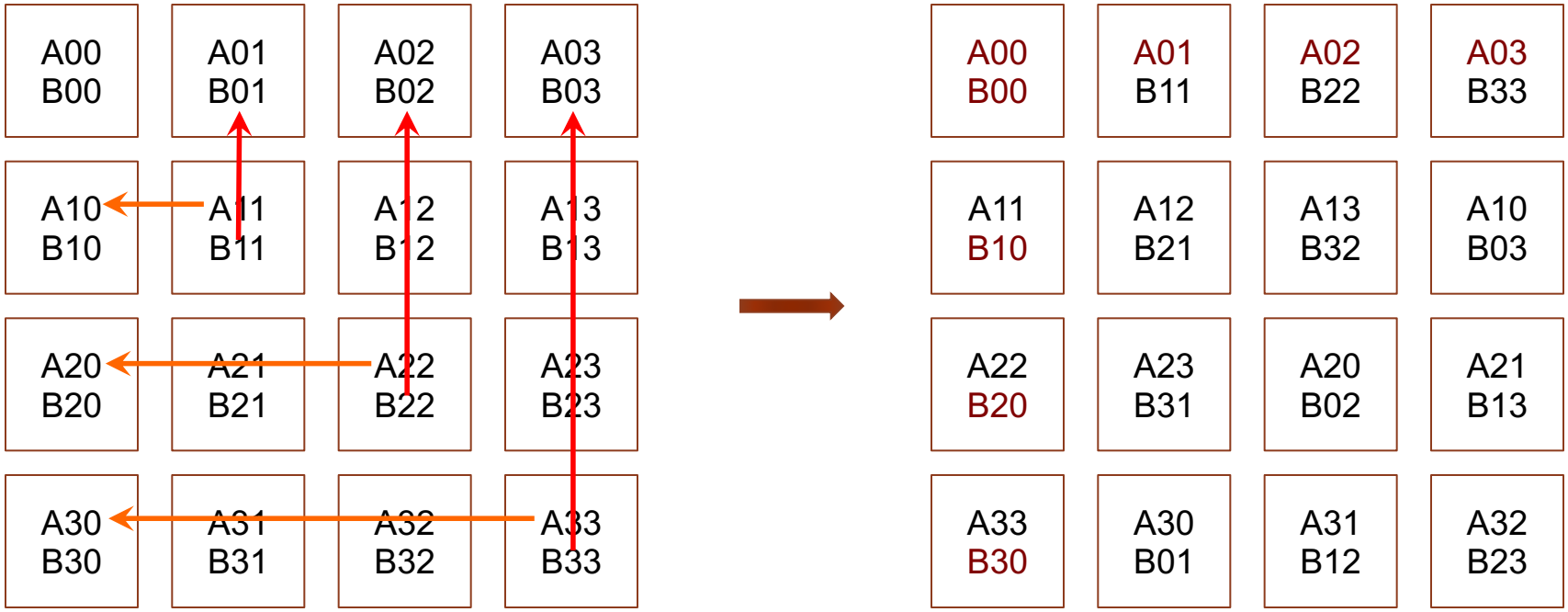
Matrix C

- For p processes, we create p blocks of size $n/p^{1/2}$.
- Simple approach:
  - all-to-all broadcast in each row of A
  - all-to-all broadcast in each column of B
  - Perform calculation with local data on each process

Naïve block operations

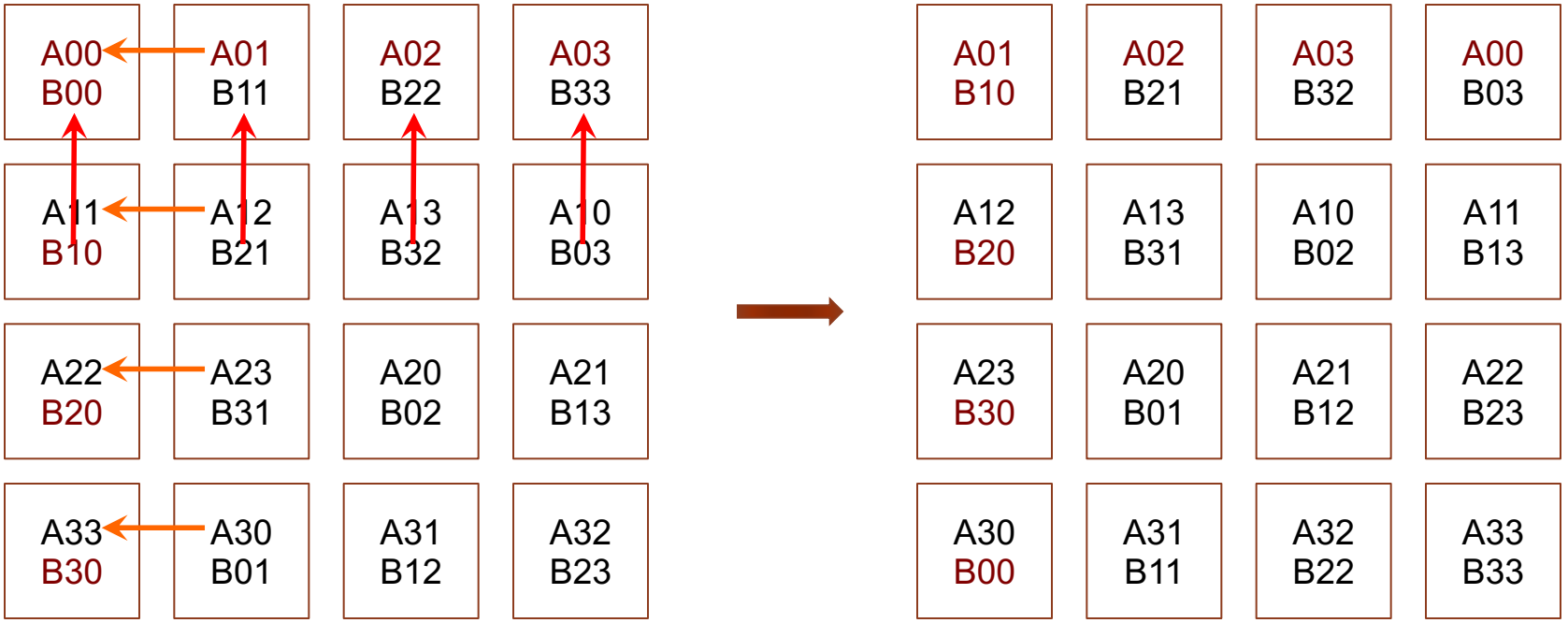
- Two issues:
  - iso-efficiency: $p = \Theta(n^2)$ (ideally, O(n^3)).
  - memory footprint: $\frac{2n^2}{\sqrt{p}}$. At iso-efficiency: $\Theta(n)$
- Cannon's algorithm allows reducing the memory footprint.
- The Dekel-Nassimi-Sahni (DNS) algorithm allows improving the scalability significantly.

Cannon's algorithm: Communication steps



- After the first communication step, each process has data to perform its first block multiplication.
- A key point is that, from then on, only a simple communication with neighbors is required at each step.

First shift



- B blocks are shifted up while A blocks are shifted left.
- Each process has now the next two blocks required for the product.
- A similar second and third shifts are required to complete the calculation: shift B up and A left.

Cannon's algorithm

- Memory footprint: $\frac{2n^2}{p}$. At iso-efficiency: $\Theta(1)$

- Every process never stores more than one block of A and B.

- The trick is to start with the right alignment.

- The blocks of A are rotated inside each row while the blocks of B are rotated inside each column.

Cannon's algorithm

- MPI code: `mmm/`

- With this algorithm, processes store only 2 blocks at a time.
- Cost of communication is slightly different from naïve algorithm but in the end the running times are comparable.
- The iso-efficiency curve is very close to the other algorithm with p scaling as n^2.
- Cannon's main feature is the reduced memory footprint. This is important as MPI codes often need a lot of memory.

Scalability: increasing the number of processes

```
Dimension of matrix: 1536, block size: 1536, number of procs along both dims: 1 1
The calculation took 16.068481 seconds; p x runtime = 16.068481

Dimension of matrix: 1536, block size: 768, number of procs along both dims: 2 2
The calculation took 4.514676 seconds; p x runtime = 18.058703

Dimension of matrix: 1536, block size: 512, number of procs along both dims: 3 3
The calculation took 2.262061 seconds; p x runtime = 20.358550

Dimension of matrix: 1536, block size: 384, number of procs along both dims: 4 4
The calculation took 2.193327 seconds; p x runtime = 35.093235
```

Dekel-Nassimi-Sahni algorithm

Dekel-Nassimi-Sahni algorithm

- The number of operations is $O(n^3)$. Therefore, we should be able to use up to about $p = O(n^3)$ processes. The DNS algorithm achieves this.

- In this algorithm, each process P_{ijk} calculates one product

```
A(i,k) * B(k,j)
```

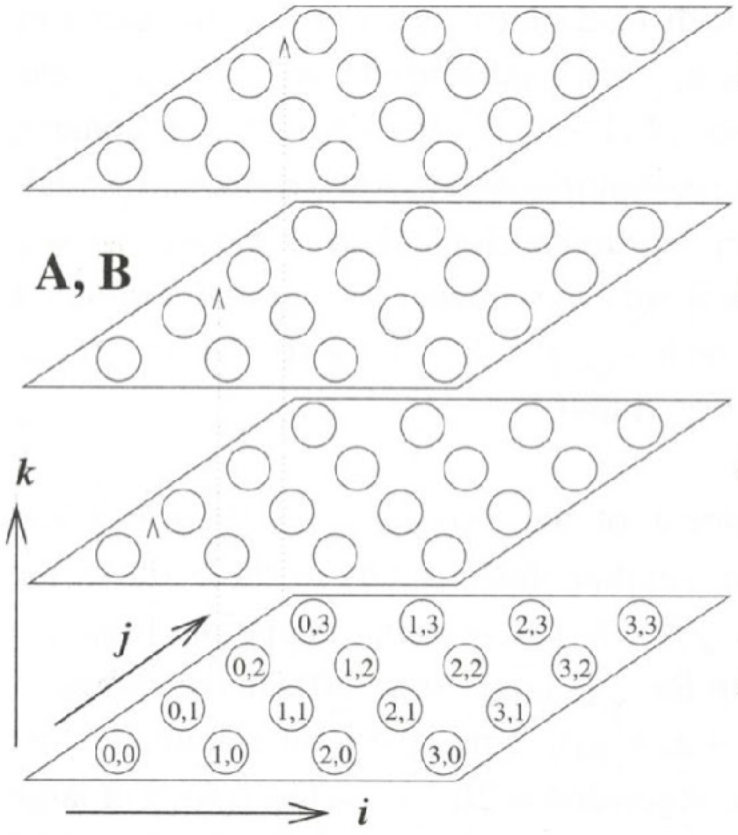
  where these are matrix blocks.

Initial distribution of data

- We use $O(n^3)$ processes.
- Initially, the processes (i,j,0) hold the data for A and B.

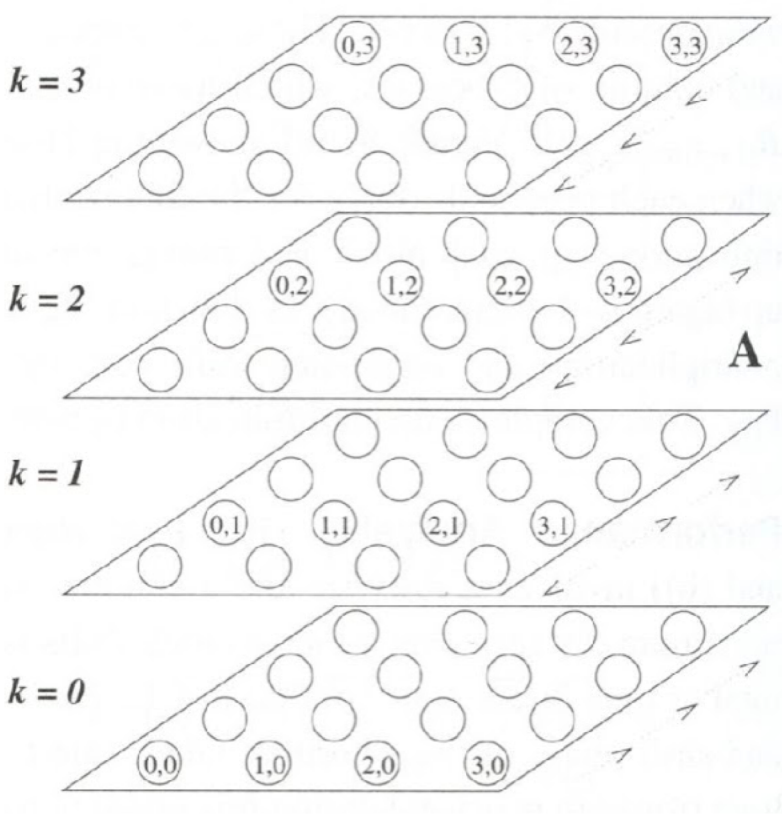
- We want to compute
  `C(i,j) += A(i,k) * B(k,j)`

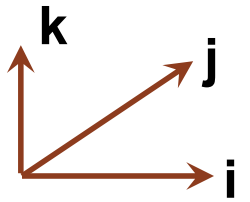
- Process (i,j,k) will do this multiplication.
- Then, a reduction over k is used to get the final C(i,j).



Step 1

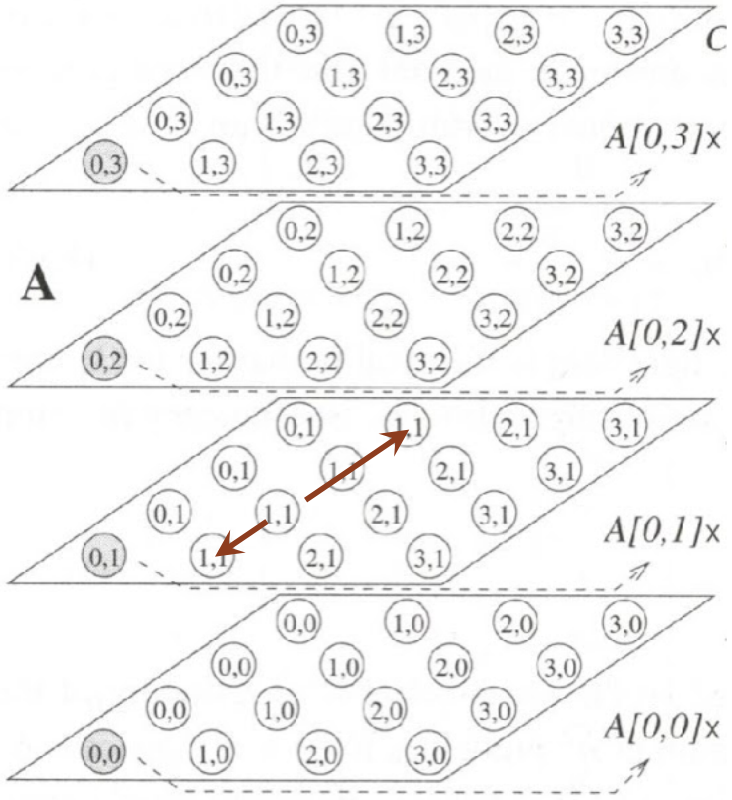
- Let's focus on A. B is similar.
- We need to propagate the blocks of A such that process (i,j,k) has A(i,k).
- So process (i,j,0) (has A(i,j)) needs to send its block to all processes (i,*,j).
- The first communication is a Send from (i,j,0) to (i,j,j).
- Do this for all i and j.

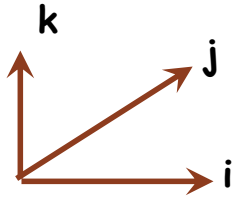




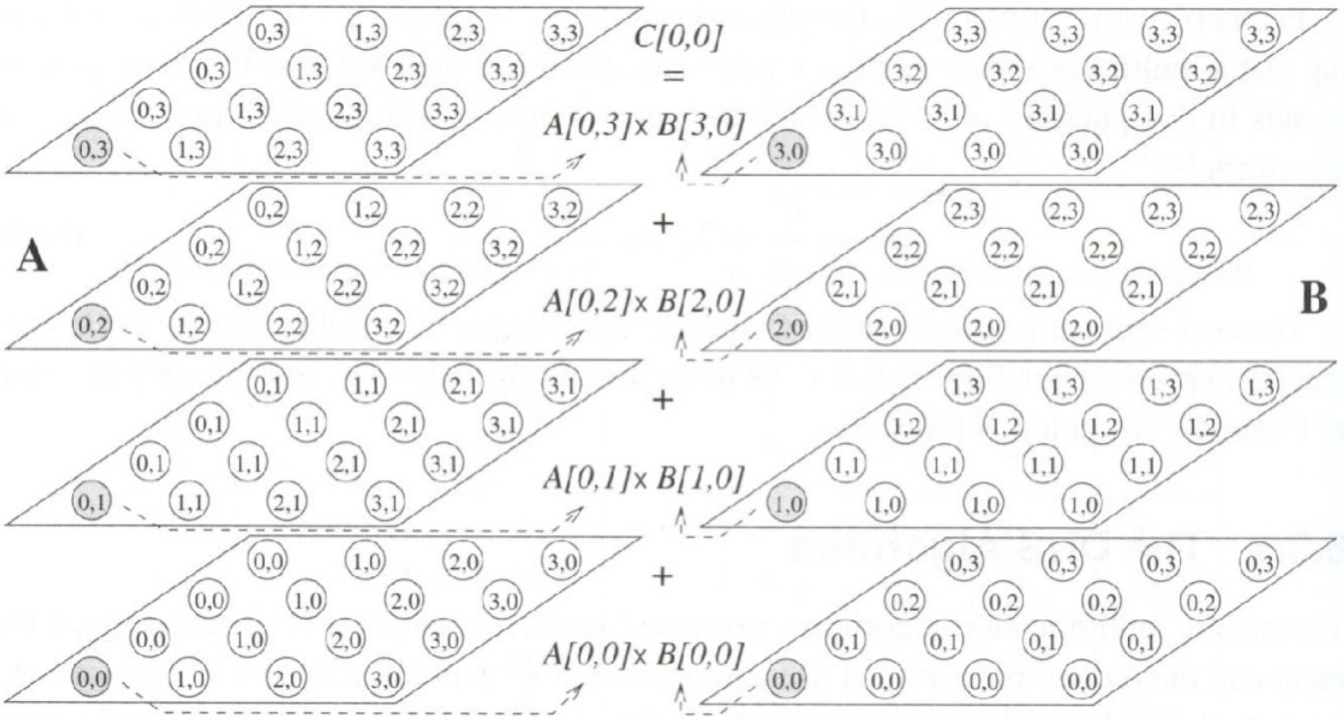
Step 2

- We now do a broadcast inside each column.
- (i,j,j) broadcasts to all (i,*,j).
- A broadcast with a communicator is used for this.





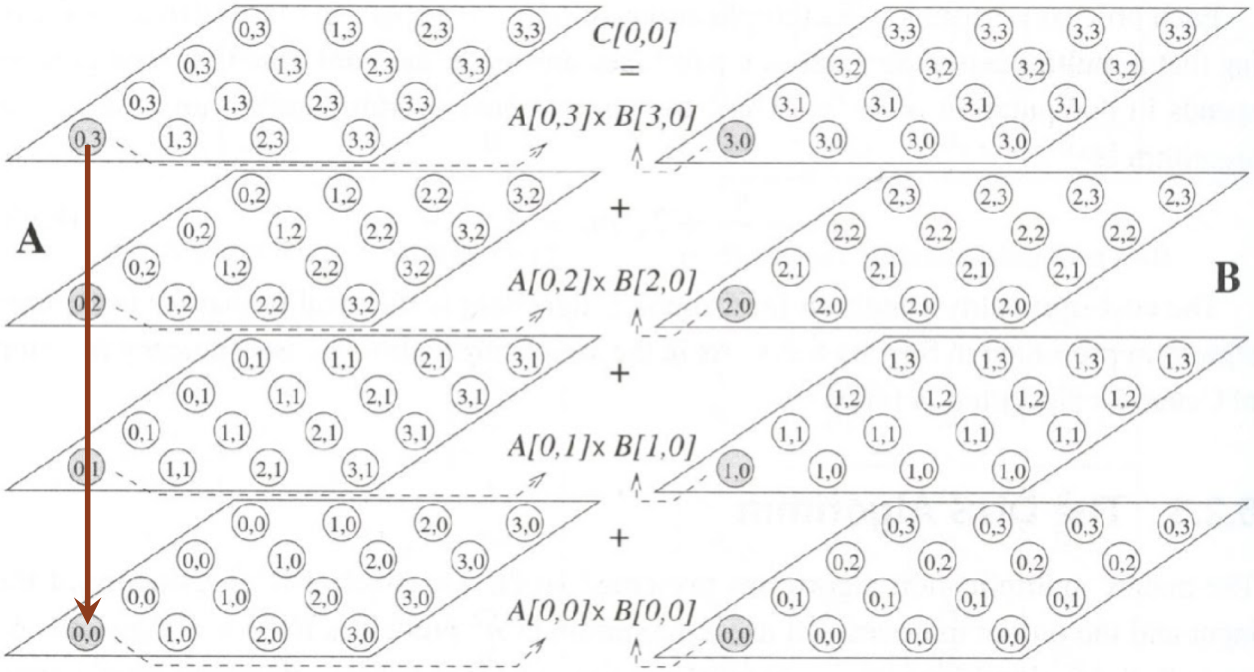
Step 3



- At last, we can compute something!
- Process (i,j,k) has A(i,k) and B(k,j).

Step 4

Reduction along k



- We finally do a reduction inside each vertical column (index k).
- Process (i,j,0) has C(i,j) in the end.

Why is DNS a better algorithm?

- The iso-efficiency is $p = \Theta\left(n^3/(\ln n)^3\right)$
- Memory footprint: $\frac{2n^2}{p^{2/3}}$. At iso-efficiency: $\Theta(\ln^2(n))$
- Because DNS is able to use larger blocks, fewer communications are required.
- The algorithm is therefore more efficient (has better scalability).
- Communication = size^2. Computation = size^3. Large blocks are favorable.