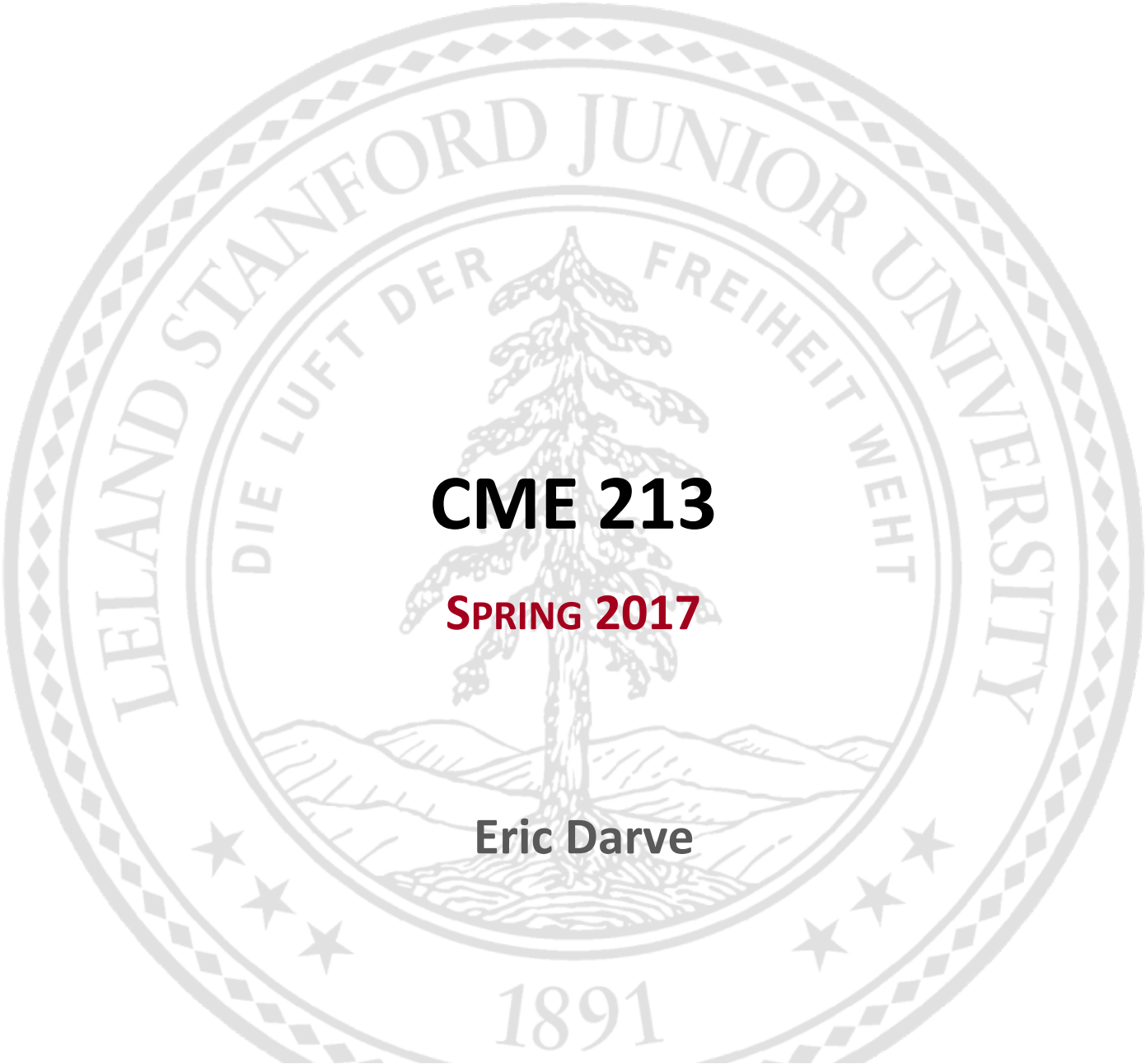


CME 213

SPRING 2017

Eric Darve

Main debugging techniques

A whole class would be needed on this topic!

1. **Use assert: test conditions on variables: equality, inequality, magnitude**
2. **Print out, but asserts are fundamentally better**
3. **Talk to your duck**
4. **Use theoretical results, e.g., convergence rate**
5. **Test against reference code**
6. **Manufactured solution: compare against known solutions**
7. **Incremental changes: test after each change (regression testing)**
8. **Unit/module tests**
9. **Test inputs of increasing difficulty; code coverage**
10. **Simplify problem to minimal buggy example**
11. **Dichotomy, divide-and-conquer**
12. **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

- This quantity measures how much faster the code runs because we are using many processes.
- 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

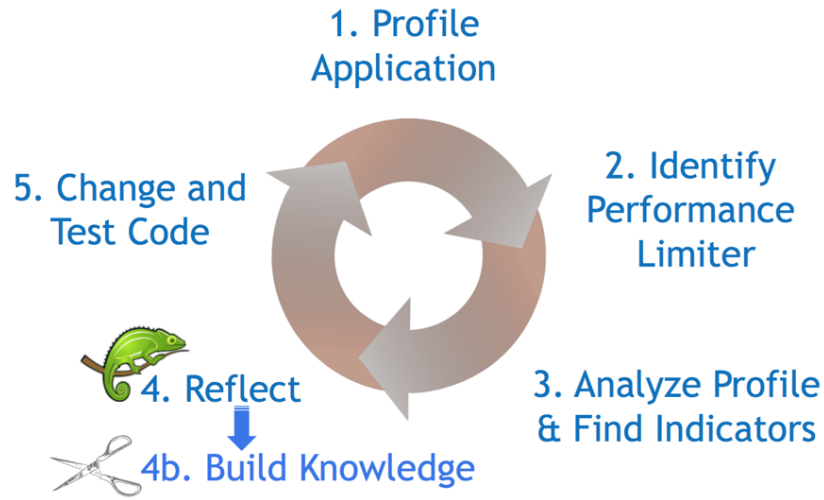
- Although this is a crude model, it can provide a general sense of what speed-up can be achieved. This is a good measure to understand how much of the code one should try to parallelize.
- Assume that a fraction f of the code is executed sequentially. Then the speed-up is given by:

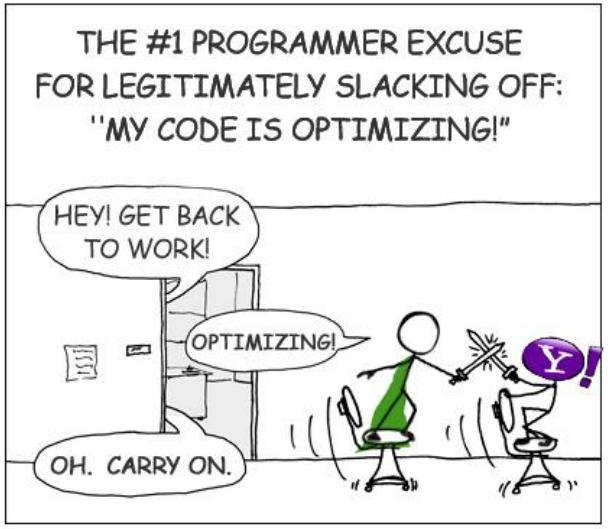
$$S_p(n) \leq \frac{T^\star(n)}{fT^\star(n) + ((1-f)/p)\,T^\star(n)} = \frac{1}{f + (1-f)/p} \leq \frac{1}{f}$$

- This means that the speed-up has an upper bound. The efficiency must go to 0 eventually. In most cases, this statement is true as p increases.
- However we are saved by another result: f often decreases with n, that is the fraction of parallel work increases 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. Further optimizing kernel 1 will no longer yield any benefit.**
- **Remember to iterate between profiling and optimizing your code.**
- **Always, focus on the longest kernel as given by your profiler.**





A MORE APPROPRIATE CONCEPT: EFFICIENCY

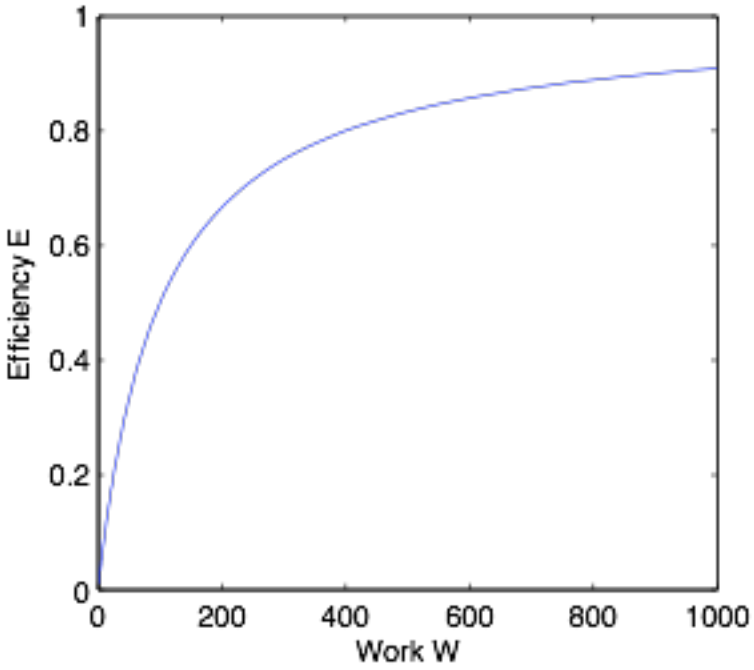
- The previous definition has a problem. As p increases, we want to know whether the speed-up scales as p or not.
- This might be difficult to assess from a plot. Ideally, the speed-up is a straight line.
- It is therefore more convenient to look at the efficiency:

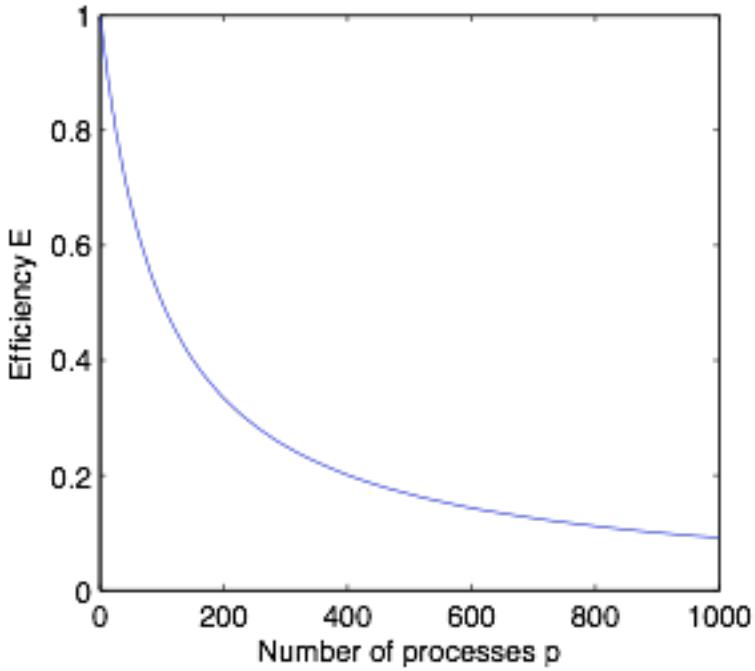
$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^\star(n)}{pT_p(n)}$$

- Ideally that quantity is simply a constant as p increases. That is 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 as the number of processes increases or as the problem size (amount of computational work to perform) increases.





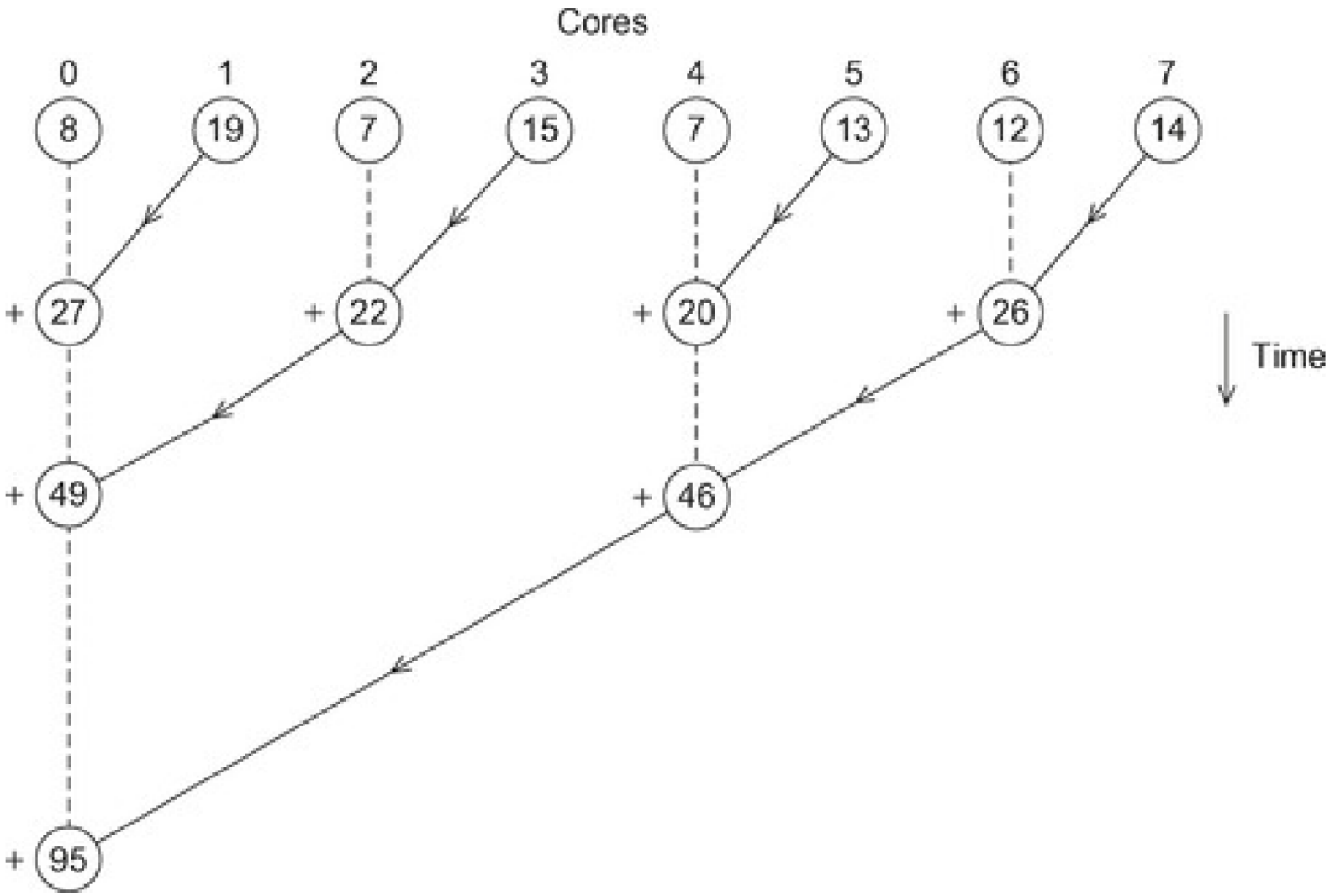
EXAMPLE 1: DOT PRODUCT

Two-step algorithm:

1. **Calculate local dot product:**

$$\sum_j a_j \; b_j$$

1. **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:

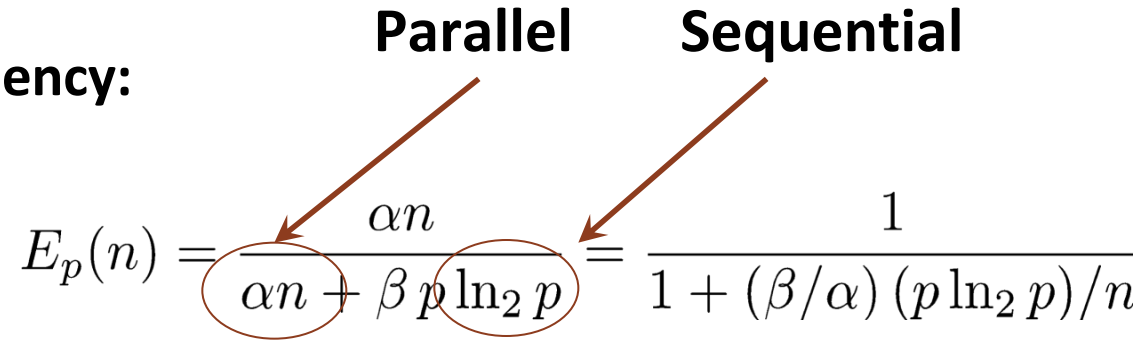
$$T^{\star}(n) = \alpha n$$

Total run time in parallel:

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

EFFICIENCY

- **Efficiency:**



- **The efficiency can be maintained provided that we do not scale p faster than:**

$$p \ln_2 p = \Theta(n) \quad \text{or} \quad p = \Theta(n/\ln_2 n)$$

- **This means that p cannot increase as fast as $\Theta(n)$.**
- **Iso-efficiency means that p increases at a rate such that the efficiency remains constant.**
- **This is very important. Good algorithms are such that p can be increased rapidly at iso-efficiency.**

EXAMPLE 2: MATRIX-VECTOR PRODUCT WITH 1D PARTITIONING

- **In that case, we can model the serial running time as:**

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

- **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:**

The iso-efficiency is

$\Theta(n)$

$\Theta(n^2 / \ln n)$

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

EXAMPLE 2: MATRIX-VECTOR PRODUCT WITH 1D PARTITIONING

- **In that case, we can model the serial running time as:**

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

- **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:**

$$p = \Theta(n)$$

- **This is actually not very good. We expect that p should increase like n^2 roughly, because this is the amount of work to do.**

EXAMPLE 2: MATRIX-VECTOR PRODUCT WITH 2D PARTITIONING

- **Recall that we previously said that 2D partitioning is better. Let's see if theory supports our claim.**
- **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):**

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

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:**

The iso-efficiency is

$\Theta(n)$

$\Theta(n^2)$

$\Theta(n^2 / \ln n)$

$\Theta(n^2 / (\ln n)^2)$

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

ISO-EFFICIENCY

- **With the previous results:**

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

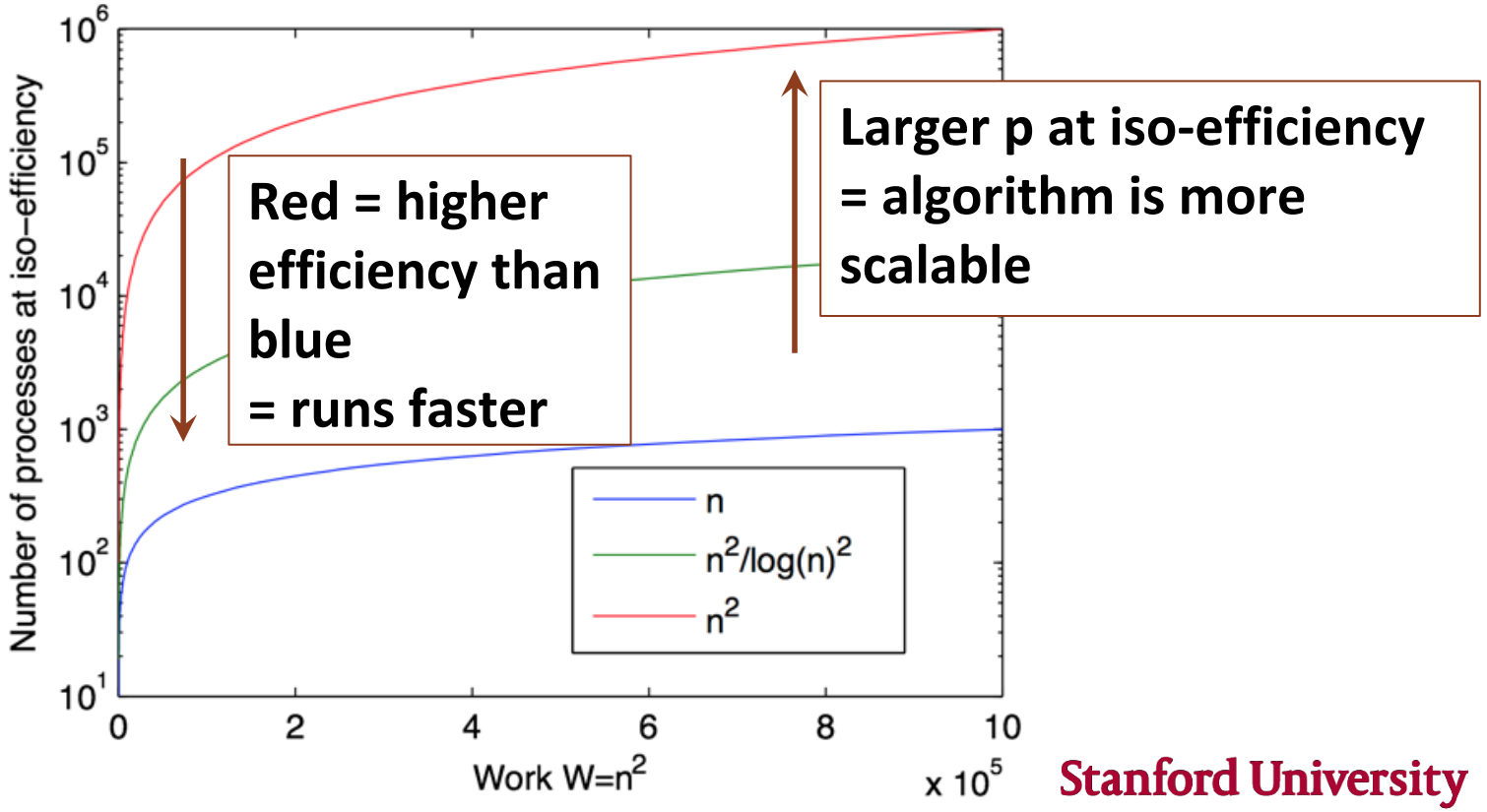
- **Iso-efficiency:**

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

- **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

- We plot for the two algorithms the value of p as a function of n such that iso-efficiency is maintained.
- Larger values of p are better.
- This means improved scalability.



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**
- **t_s: latency**
- **t_w: bandwidth**

MATRIX-MATRIX PRODUCTS

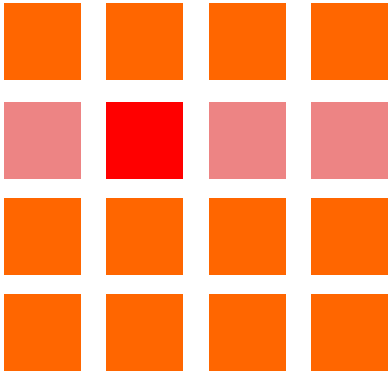
MATRIX-MATRIX PRODUCTS

Algorithm in pseudo-code:

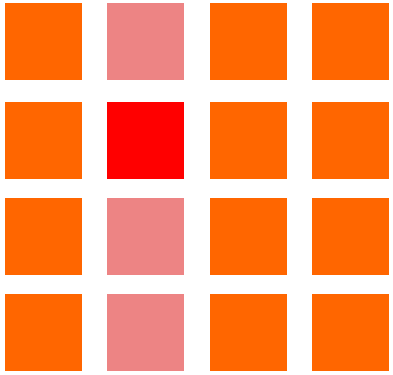
```
for i=0:n-1 do
      for j=0:n-1 do
            C(i,j) = 0;
            for k=0:n-1
                  C(i,j) += A(i,k) * B(k,j);
            end
      end
end
```

NAÏVE BLOCK OPERATIONS

- **Algorithm proceeds by doing block operations.**



Matrix A



Matrix B

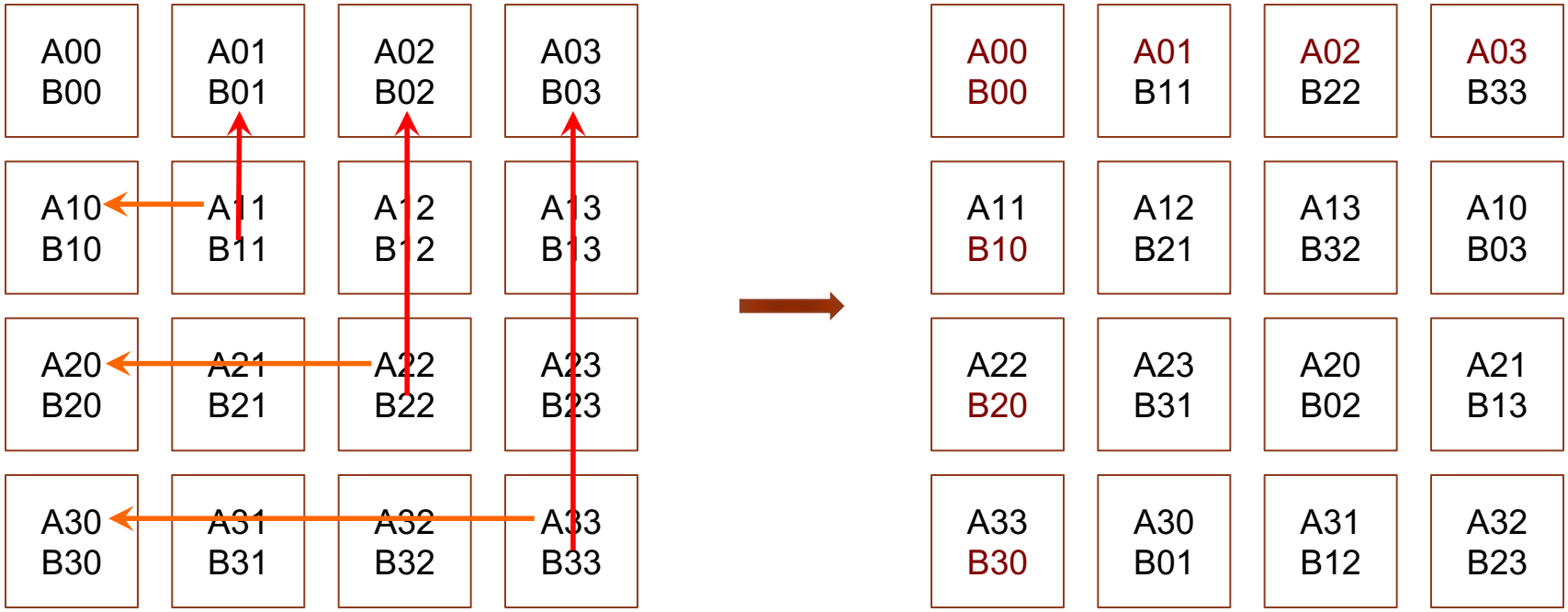
- **For p processes, we create p blocks of size $n/p^{1/2}$.**
- **Simple approach:**
  - **all-to-all broadcast in each row of A**
  - **all-to-all broadcast in each column of B**
  - **Perform calculation with local data on each process**
- **Iso-efficiency:**

$$p = \Theta(n^2)$$

CANNON'S ALGORITHM

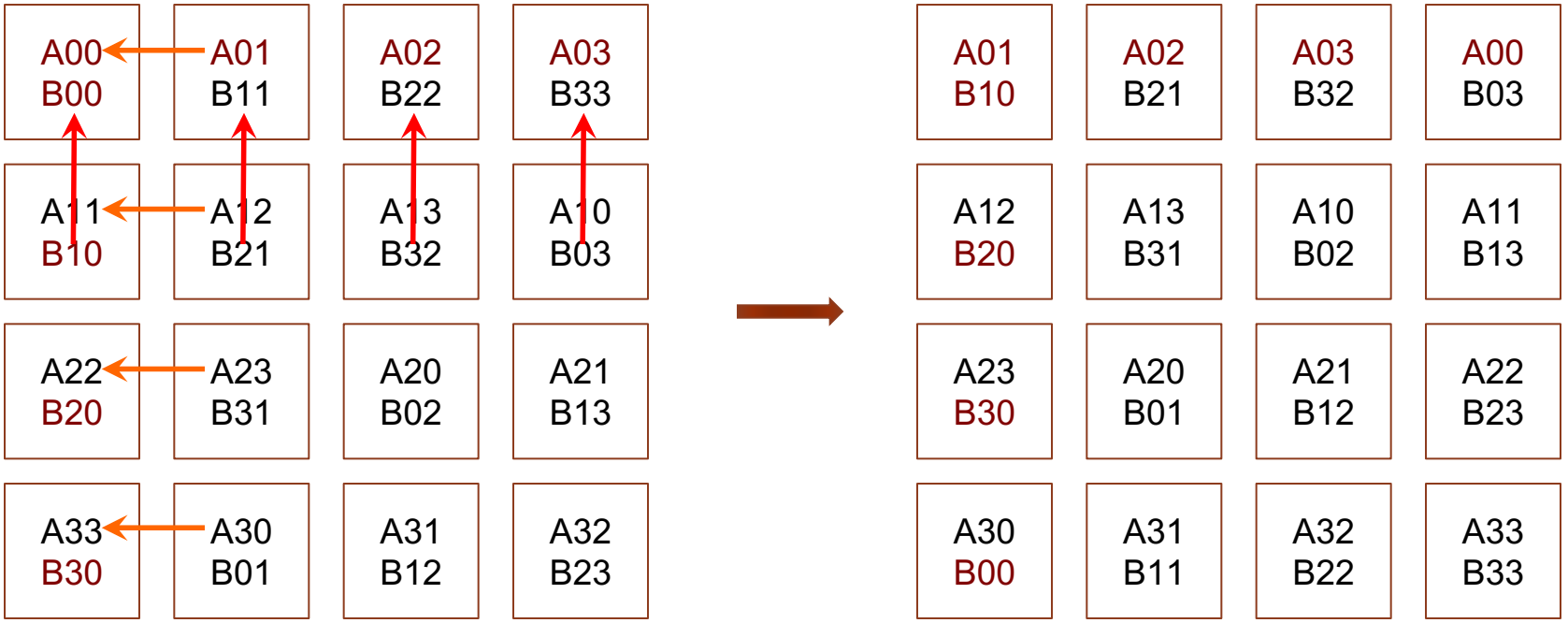
- There are two issues with this simple algorithm:
  - We should be able to increase p closer to n^3 at iso-efficiency.
  - This algorithm requires a lot of memory since a process needs to store an entire block row of A and block column of B.
- Cannon's algorithm allows reducing the memory footprint.
- It works by cleverly shuffling the blocks of A and B such that each process never stores more than one block of A and B.
- The blocks of A are rotated inside each row while the blocks of B are rotated inside each column.
- The trick is to start with the right alignment.
- We will see later that the Dekel-Nassimi-Sahni (DNS) algorithm allows improving the scalability significantly.

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.

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

Dimension of matrix: 1536, block size: 256, number of procs along both dims: 6 6

The calculation took 0.472822 seconds; p x runtime = 17.021599

Dimension of matrix: 1536, block size: 192, number of procs along both dims: 8 8

The calculation took 0.174466 seconds; p x runtime = 11.165817

SCALABILITY: FIXED NUMBER OF PROCESSES

```
Dimension of matrix: 1536, block size: 512
Number of procs along both dims: 3 3

1 node   The calculation took 2.090732 seconds
p x runtime = 18.816589

2 nodes  The calculation took 1.783496 seconds
p x runtime = 16.051465

3 nodes  The calculation took 1.705040 seconds
p x runtime = 15.345360

5 nodes  The calculation took 1.671645 seconds
p x runtime = 15.044806

9 nodes  The calculation took 1.651060 seconds
p x runtime = 14.859539
```

at is the main advantage of Can compared to the previous algorithm?

Less flops

Less memory

Less communication

Fewer messages

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

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.**

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

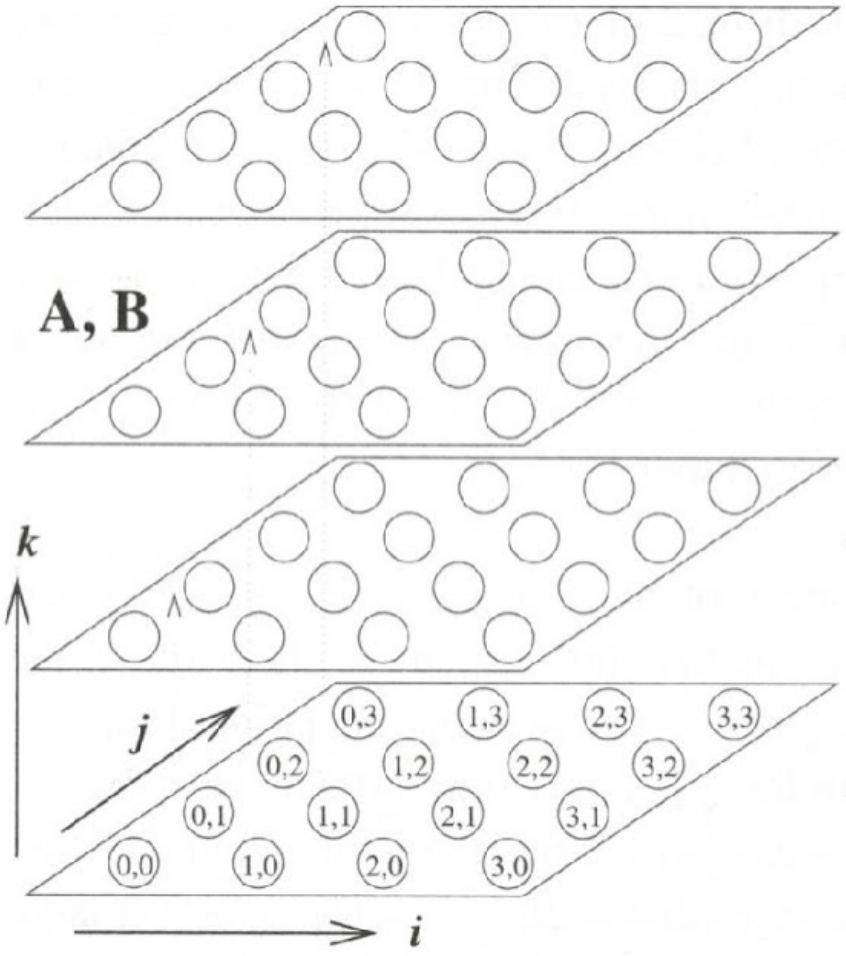
  $A(i,k) * B(k,j)$

  where these are matrix blocks.

INITIAL DISTRIBUTION OF DATA

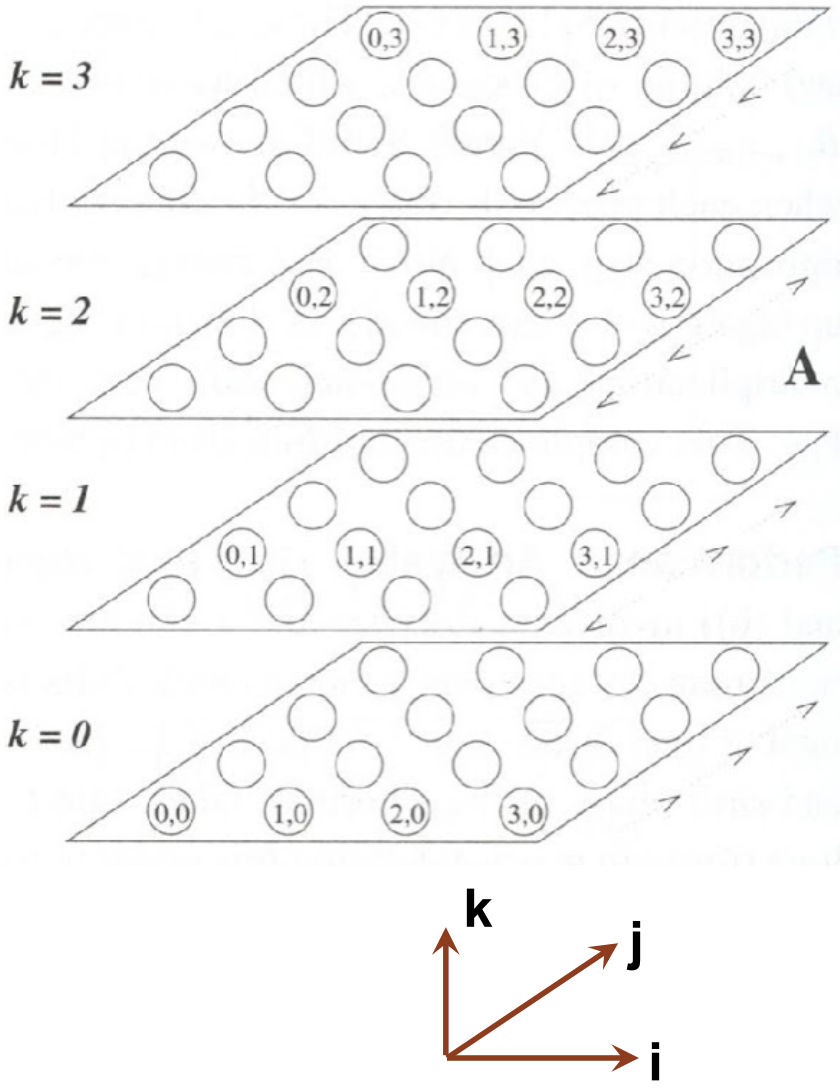
- We use $p=n^3$ processes.
- We want to compute

  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).
- Initially, the processes (i,j,0) hold the data for A and B.



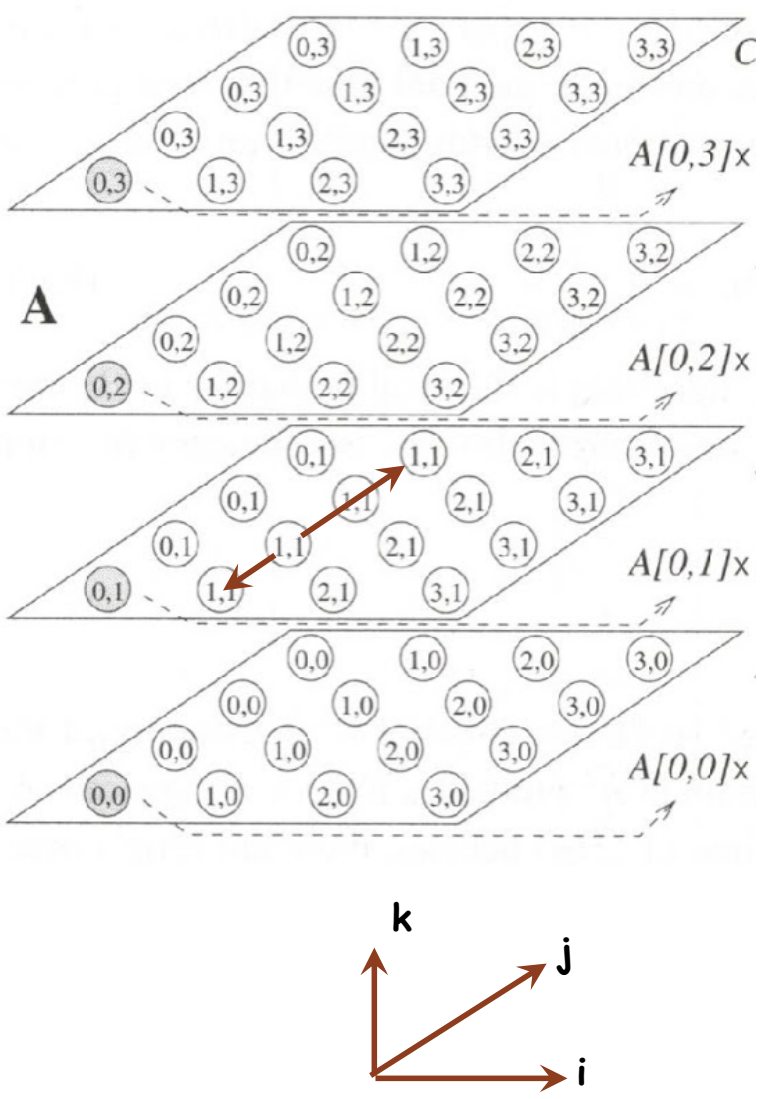
STEP 1

- 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.

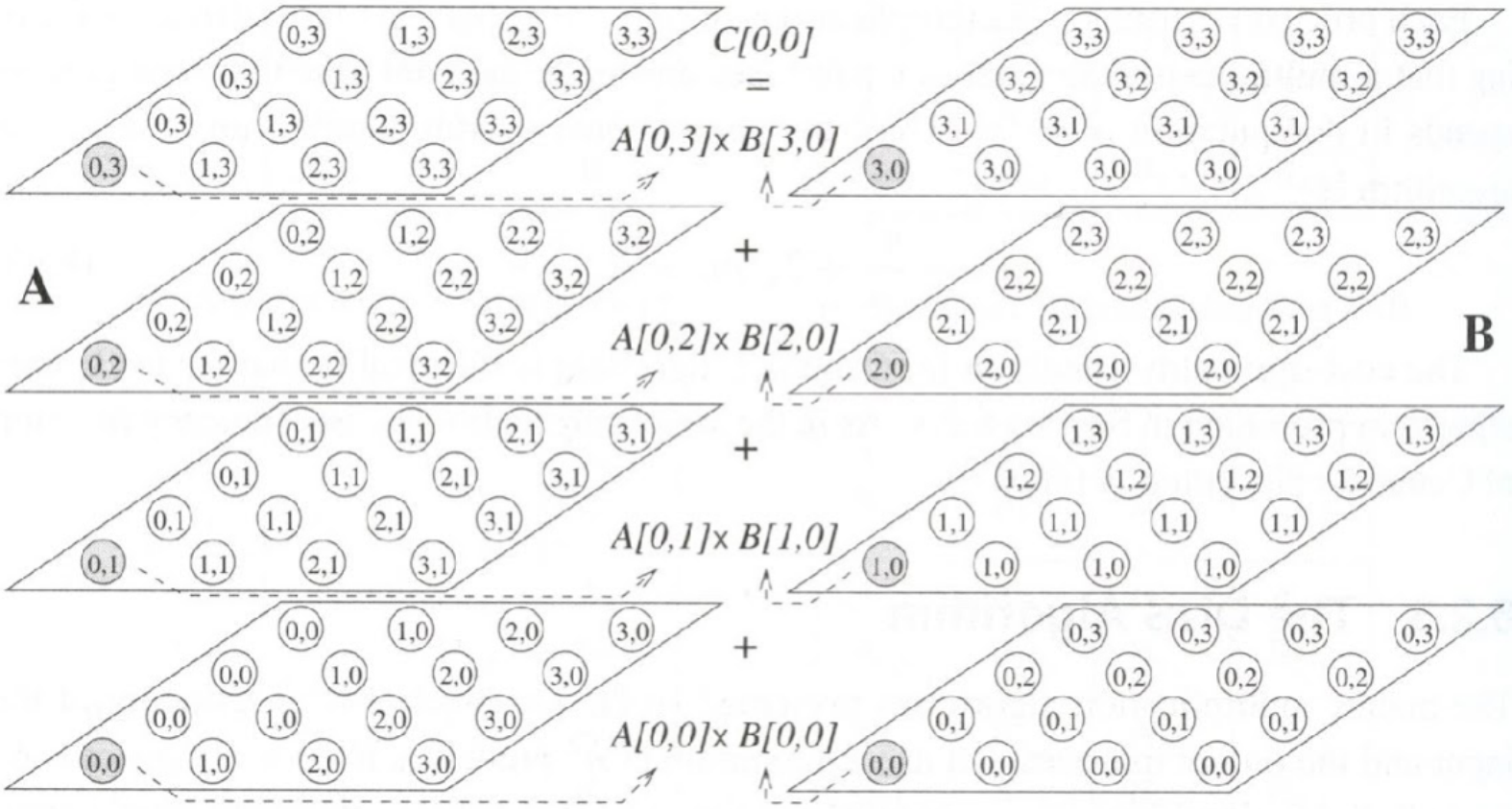


STEP 2

- 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.



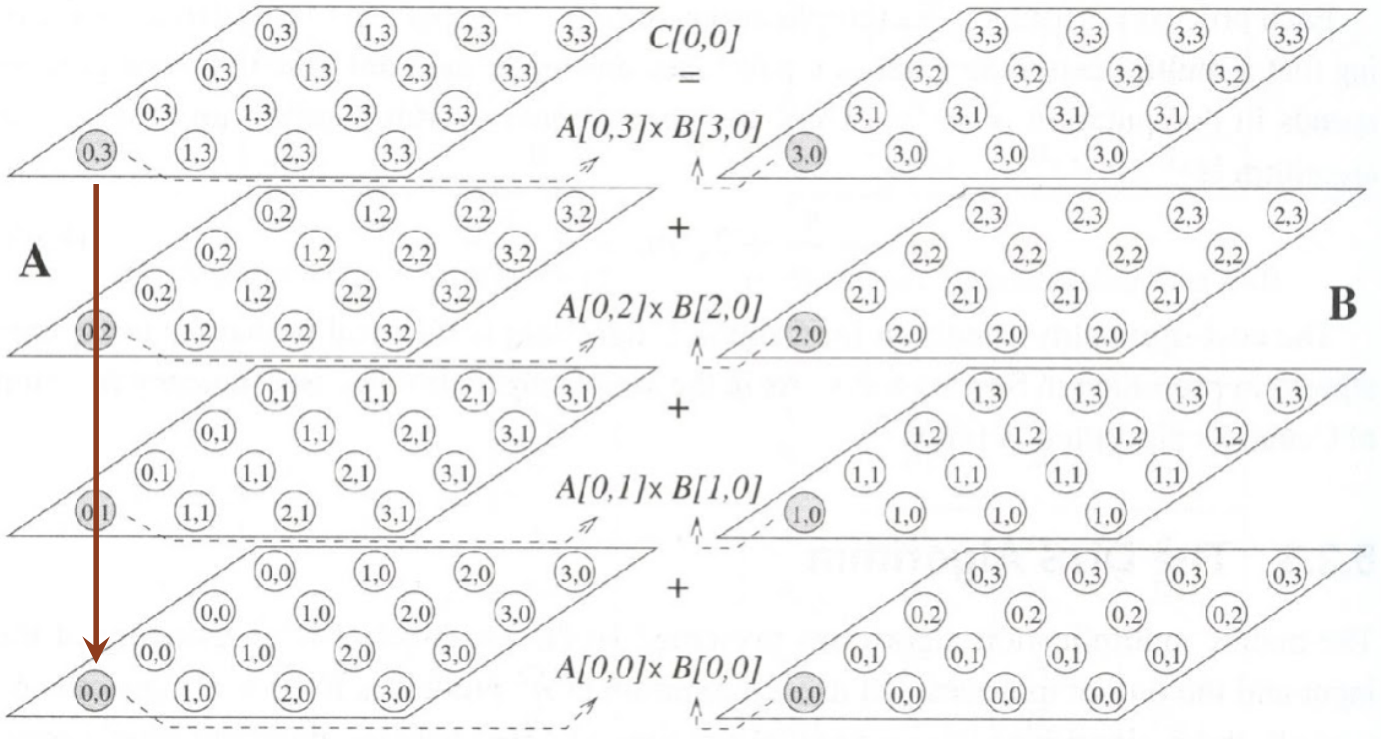
STEP 3



- 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**

$$p = \Theta\left(n^3/(\ln n)^3\right)$$

- **In Cannon, $p = O(n^2)$, whereas in DNS, $p = O(n^3/(\ln n)^3)$.**
- **What does it mean that the iso-efficiency curve of DNS is better than Cannon?**
- **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.**