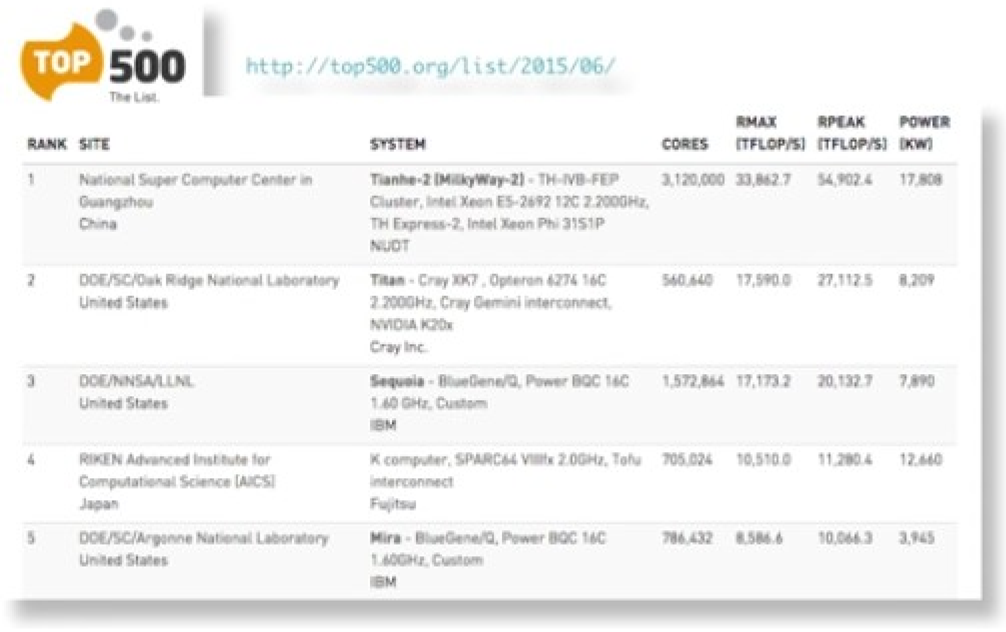
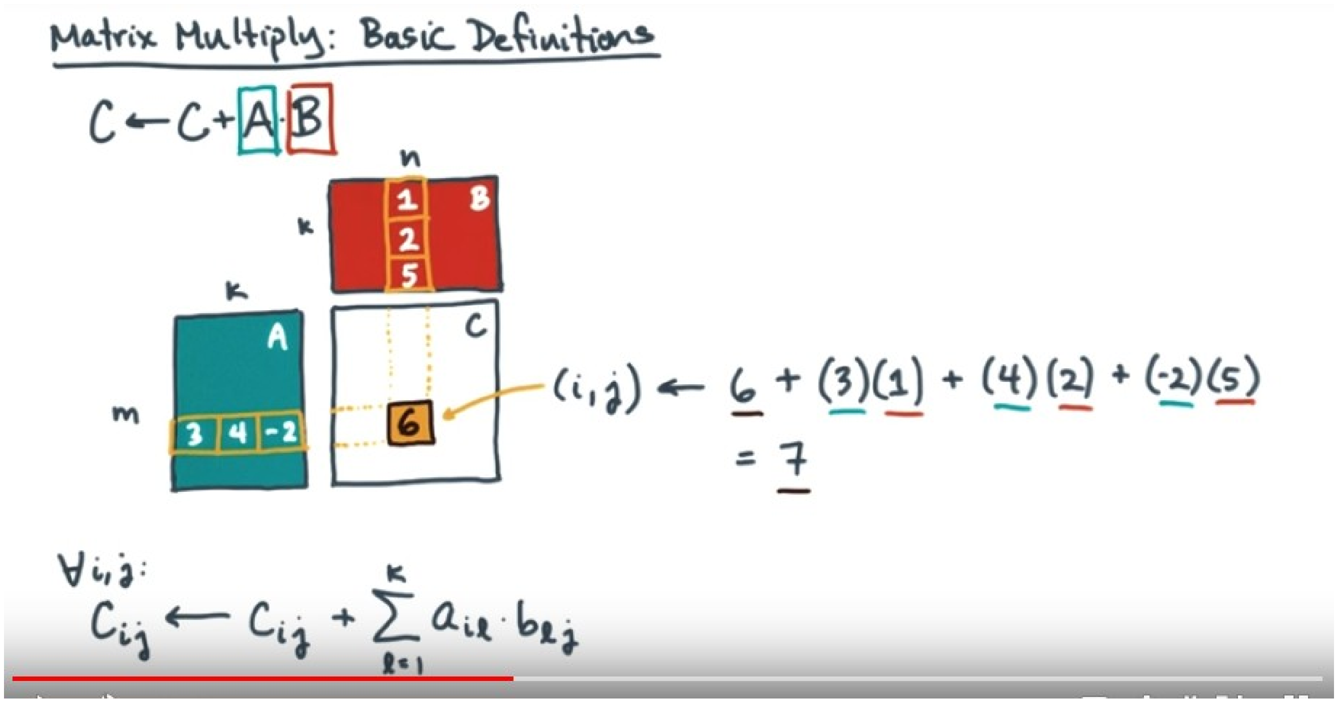


http://top500.org/list/2015/06/

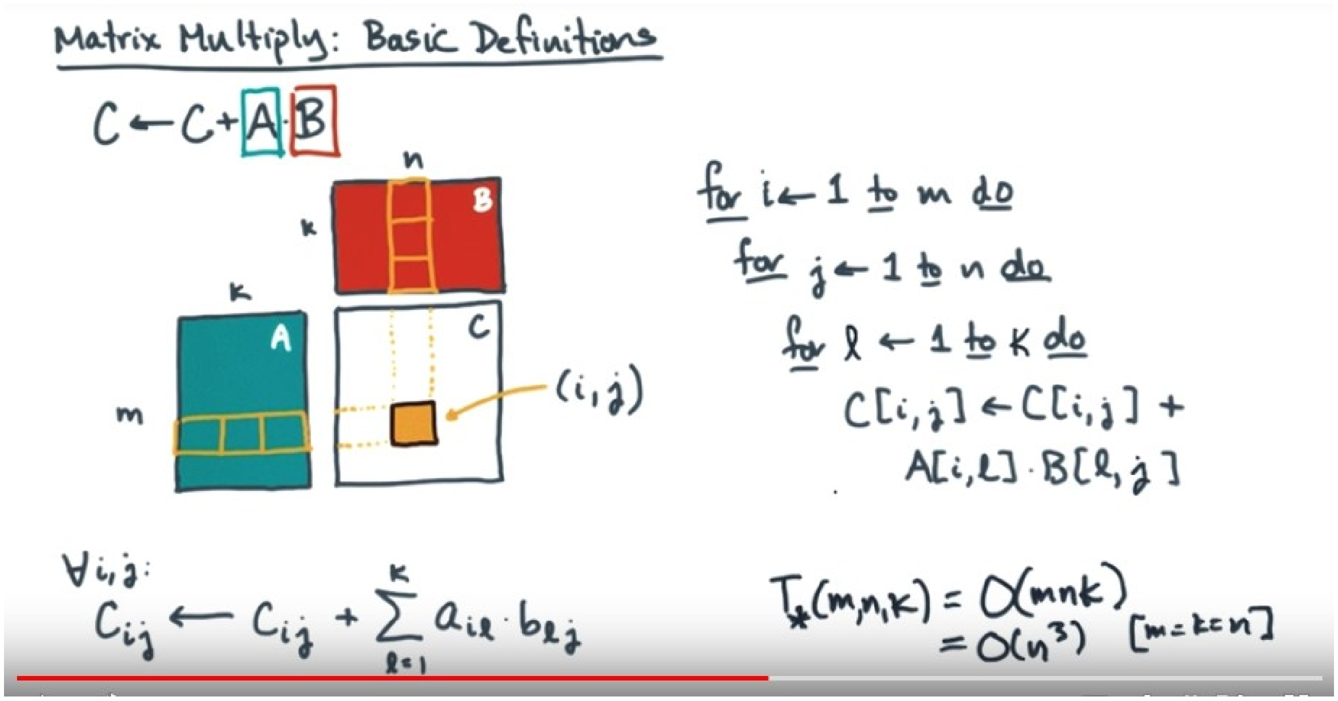
RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

1. This lesson is about distributed memory algorithms for multiplying matrices. Matrix multiply appears in lots of applications, from simulations in quantum computational chemistry, to backpropagation in neural networks. And when the HPC community ranks the worlds supercomputers for its top 500 list, it does so using a benchmark that, to first order, is basically a really big matrix multiply. Now that's not the only reason to study it. The other one is that it's not so hard to analyze and that makes it a great starting point for practicing some basic analysis techniques.


Matrix Multiply: Basic Definitions
$C \leftarrow C + AB$
n
k
B
1
2
5
k
A
C
m
3
4
-2
6
$(i,j) \leftarrow 6 + (3)(1) + (4)(2) + (-2)(5) = 7$
$\forall i,j: \; c_{ij} \leftarrow c_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$

2. A matrix multiply is the following operation. Given two input matrices, A and B, and given a third matrix C, perform the update, A times B + C, and overwrite C. Schematically, I like to draw matrix multiplies this way. Matrix A has m rows and k columns. Matrix B has k rows and n columns, and the

matrix C has m rows, same as A, and n columns, same as B. The matrix update formula may be interpreted as follows. Consider some output element Cij. This scalar update formula at the bottom tells you how to update it. Essentially, you compute a dot product between a row of a and a column of b. The dot product comes from multiplying corresponding elements of the two vectors and then accumulating them into the output. Let's look at an example. Suppose row i of A and column j of B have these values. And let's further suppose that Cij has this initial value. The dot product is the sum of the element-wise products. So in this case, we take the element-wise products 3 times 1, 4 times 2, -2 times 5. Add them all together, accumulate them with the initial value of C, which was 6, and we'd get 7.



Matrix Multiply: Basic Definitions

$C \leftarrow C + \boxed{A} \cdot \boxed{B}$

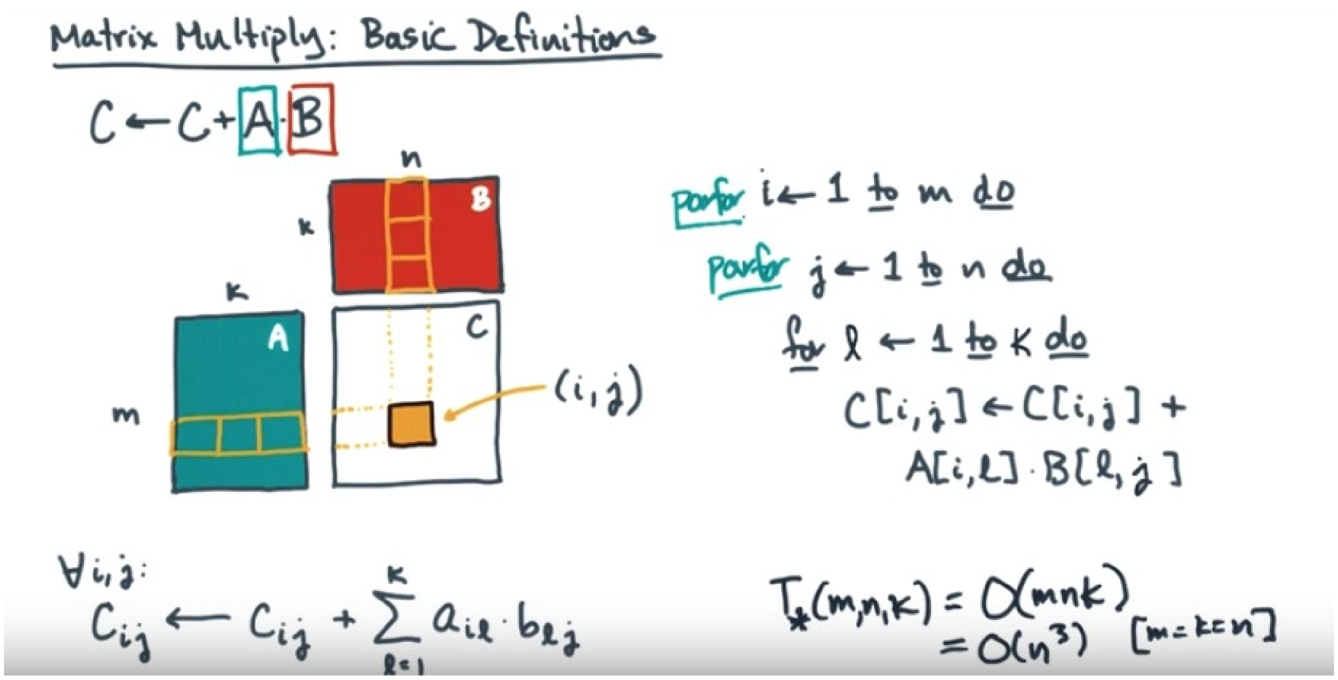
```
for i ← 1 to m do
  for j ← 1 to n do
    for l ← 1 to k do
      C[i,j] ← C[i,j] +
          A[i,l] · B[l,j]
```

$\forall i,j:$

$$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

$$T_*(m,n,k) = O(mnk)$$
$$= O(n^3) \quad [m = k = n]$$

Here's what a matrix multiply looks like as pseudocode. It's just three nested loops. This form makes it easy to see that the algorithm takes time O of m times n times k. And of course, when all the dimensions are equal, that's big O of, say, n cubed.



Matrix Multiply: Basic Definitions

$C \leftarrow C + \boxed{A} \cdot \boxed{B}$

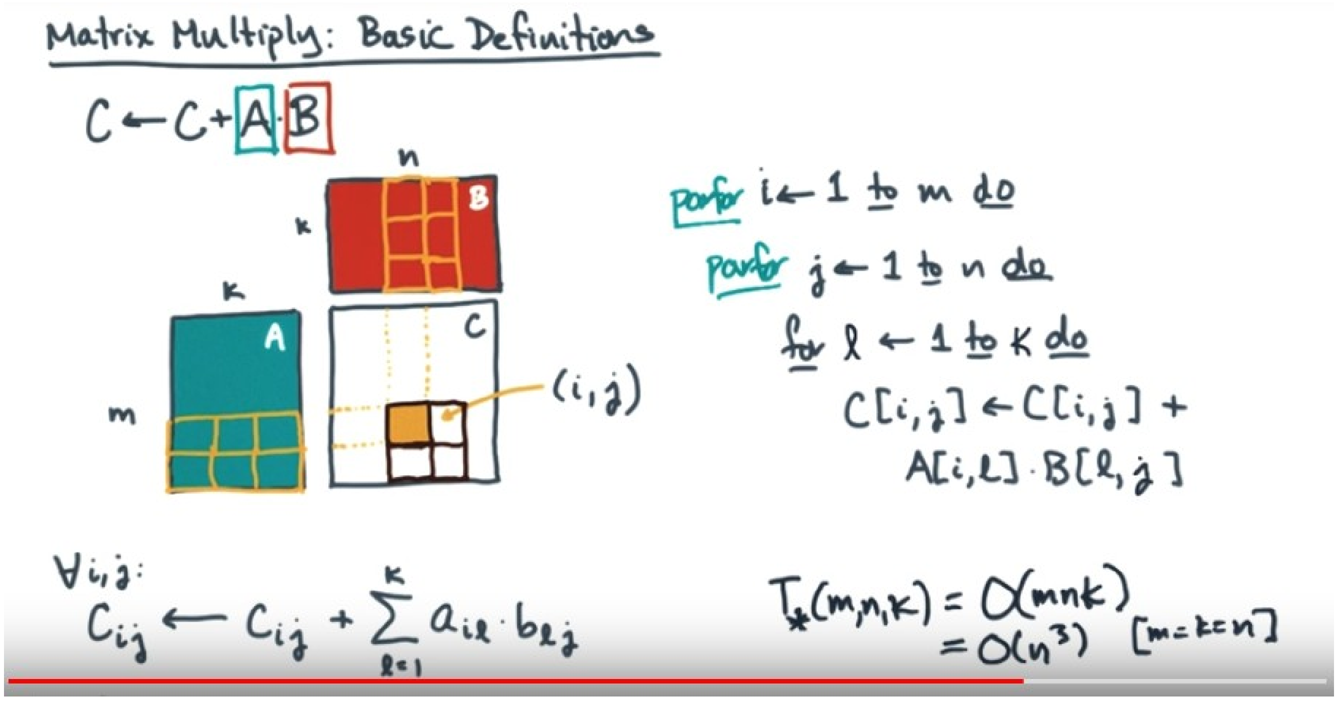
```
parfor i ← 1 to m do
  parfor j ← 1 to n do
    for l ← 1 to k do
      C[i,j] ← C[i,j] +
               A[i,l]·B[l,j]
```

$\forall i,j:$

$$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

$$T_*(m,n,k) = O(mnk) = O(n^3) \quad [m=k=n]$$

So, where's the parallelism? Both the picture and the code suggests that all the output elements are independent of one another. So, if you were to parallelize this code for a PRAM type system, the first easy thing you would do is to replace the outermost for loops with parallel for loops. 意思就是上圖左邊的 不同的 dot product 可以在不同 node 上完成.



Matrix Multiply: Basic Definitions

$C \leftarrow C + \boxed{A} \cdot \boxed{B}$

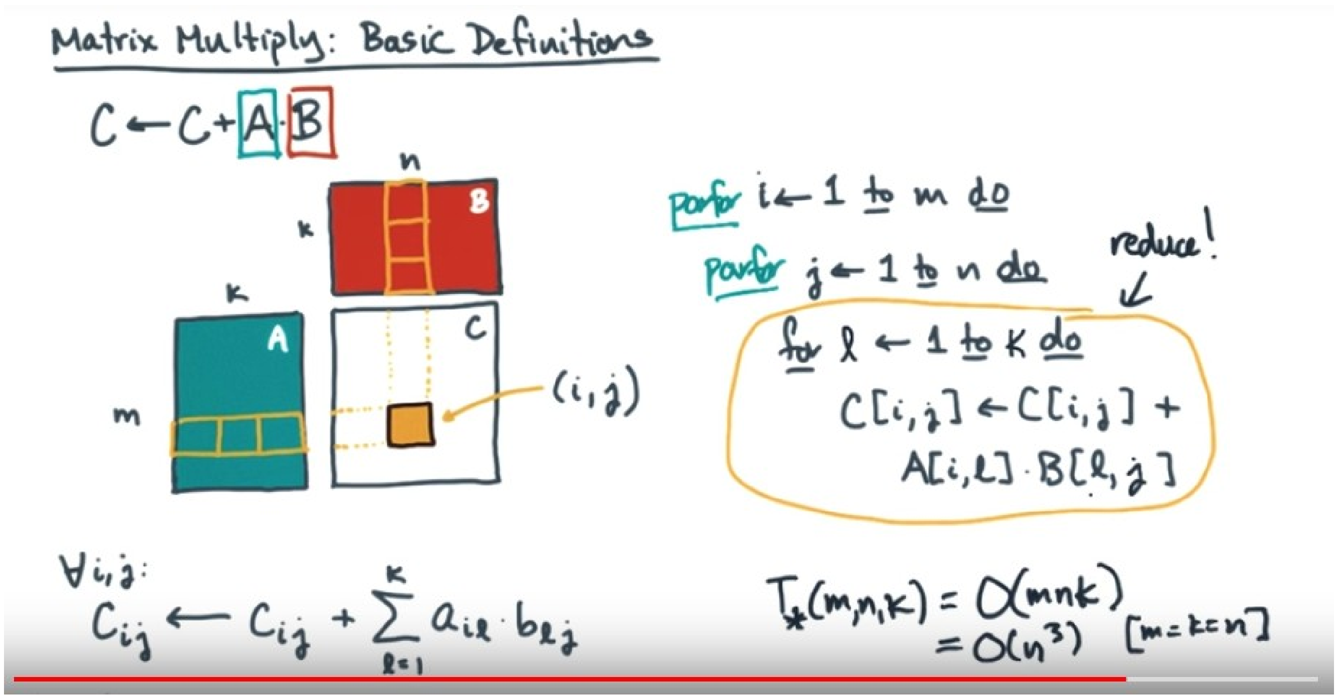
```
parfor i ← 1 to m do
  parfor j ← 1 to n do
    for l ← 1 to k do
      C[i,j] ← C[i,j] +
               A[i,l]·B[l,j]
```

$\forall i,j:$

$$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

$$T_*(m,n,k) = O(mnk) = O(n^3) \quad [m=k=n]$$

In fact, that gives you another useful fact about matrix multiply. Suppose you want to compute a group of elements of C. Then you can do a group update by doing a sub matrix multiply or block matrix multiply. For example, this block would depend on this block of rows, and this block of columns. Put another way, everywhere you see a matrix element in this picture, you could instead imagine not a single element, but an entire submatrix. The computation is essentially the same with respect to blocks.



Matrix Multiply: Basic Definitions

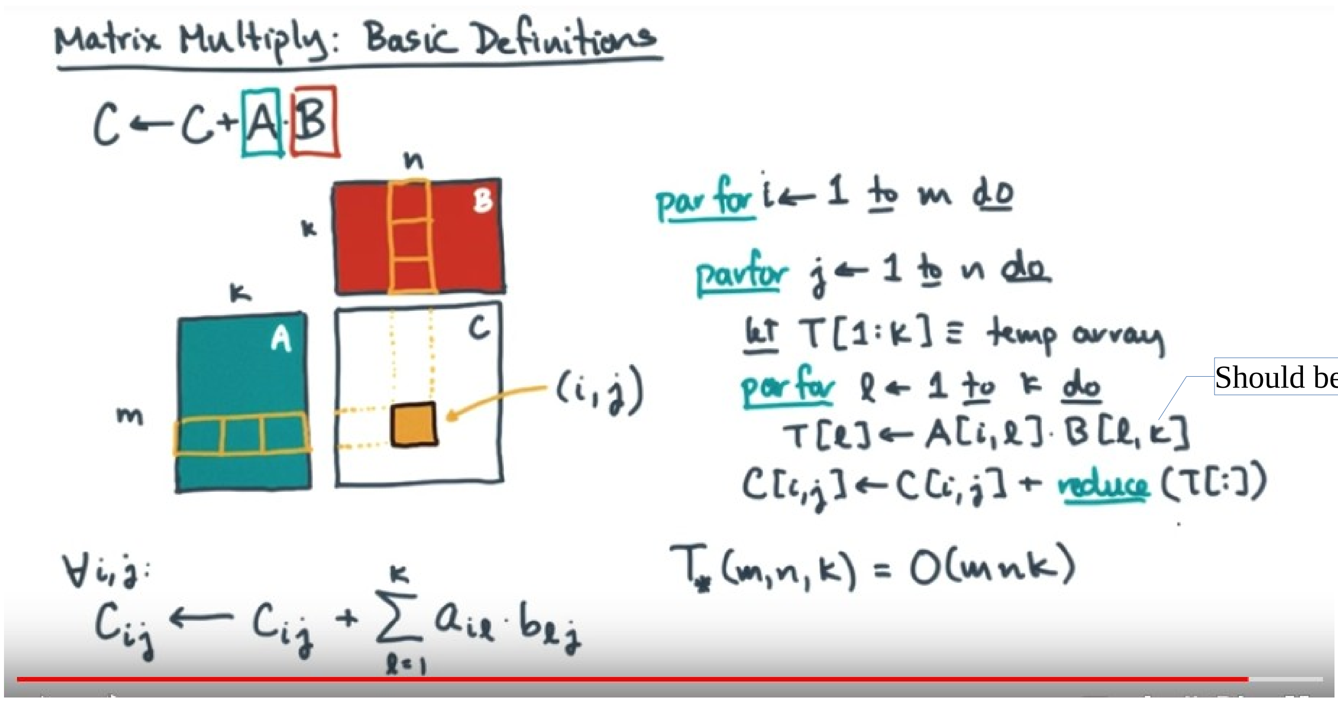
$C \leftarrow C + \boxed{A} \cdot \boxed{B}$

```
parfor i ← 1 to m do
  parfor j ← 1 to n do
    for l ← 1 to k do
      C[i,j] ← C[i,j] +
               A[i,l]·B[l,j]
```

$\forall i,j:$
$$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

$T_*(m,n,k) = O(mnk)$
$= O(n^3)$ $[m=k=n]$

Okay, back to parallelism. What about this innermost loop? I hope you recognize that it's basically a reduction. Notice that the element-wise products within the loop are independent. Once you've computed them all, you just add them all up.



Matrix Multiply: Basic Definitions

$C \leftarrow C + \boxed{A} \cdot \boxed{B}$

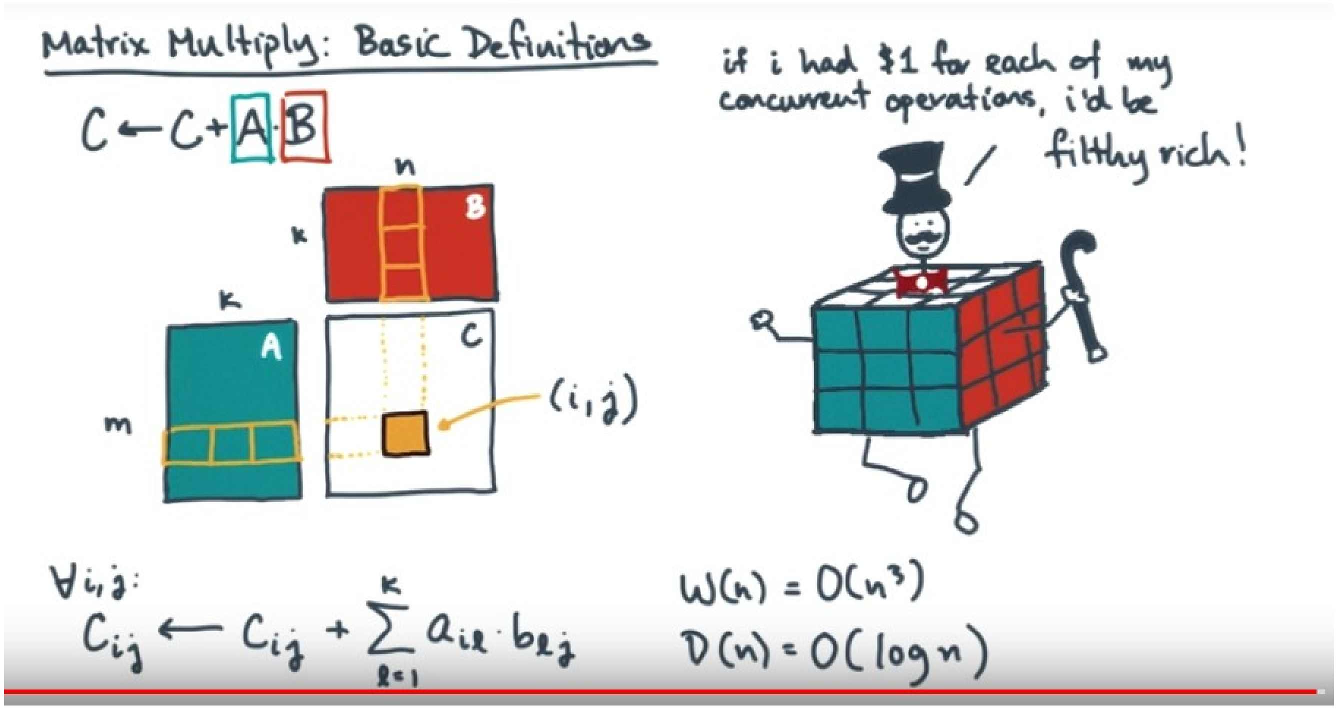
```
parfor i ← 1 to m do
  parfor j ← 1 to n do
    let T[1:k] ≡ temp array
    parfor l ← 1 to k do
      T[l] ← A[i,l]·B[l,k]
    C[i,j] ← C[i,j] + reduce(T[:])
```

Should be j

$T_*(m,n,k) = O(mnk)$

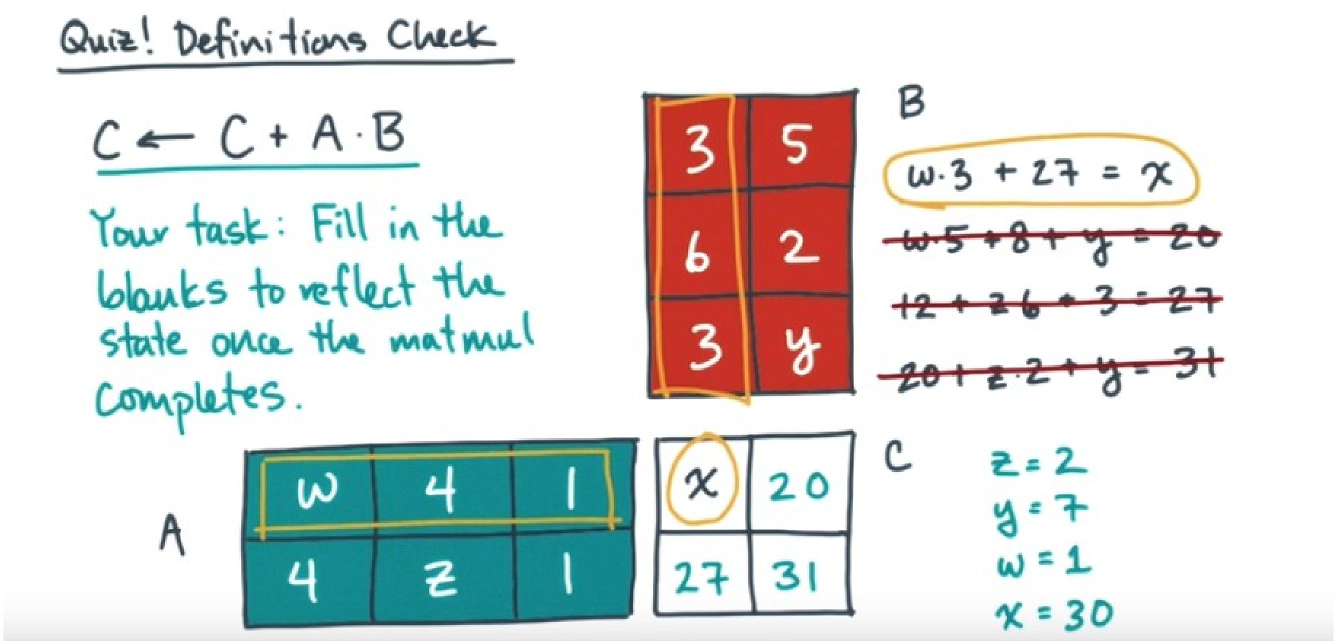
$\forall i,j:$
$$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

So we can replace the inner-most l loop with the following. So there's a temporary array, a new loop to compute the independent products, and then a reduction.


Matrix Multiply: Basic Definitions
$C \leftarrow C + \boxed{A} \cdot \boxed{B}$
n
k
B
k
A
m
C
(i, j)
if i had $1 for each of my concurrent operations, i'd be filthy rich!
$\forall i, j:$
$C_{ij} \leftarrow C_{ij} + \sum_{\ell=1}^{k} a_{i\ell} \cdot b_{\ell j}$
$W(n) = O(n^3)$
$D(n) = O(\log n)$

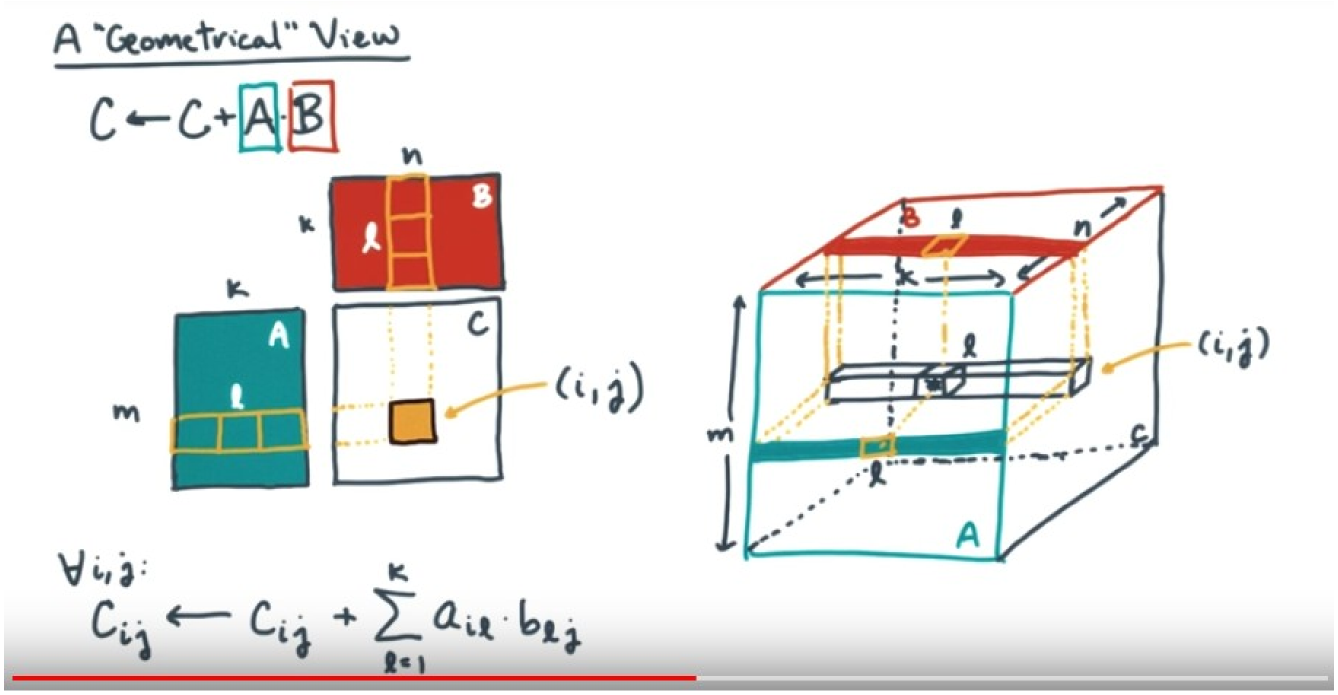
Now if all the dimensions are equal to little n, then the total work is n cubed, and the span is just log n. So, I hope you find a matrix multiply to be easy to state and rich with available parallelism.

3. Let's check to see if you understand the basic definition of a matrix multiply. Here's an A, B and C with four missing values. I want you to determine the missing values. You can assume that initially, C was set to all zeros. So, what I want you to compute is the state of the matrices once the matrix multiply is finished.


Quiz! Definitions Check
$C \leftarrow C + A \cdot B$
Your task: Fill in the blanks to reflect the state once the matmul completes.
B
3 5
6 2
3 y
$w \cdot 3 + 27 = x$
~~$w \cdot 5 + 8 + y = 20$~~
~~$12 + 26 + 3 = 27$~~
~~$20 + z \cdot 2 + y = 31$~~
A
w 4 1
4 z 1
C
x 20
27 31
z = 2
y = 7
w = 1
x = 30

(本 quiz 太弱, 不用看)

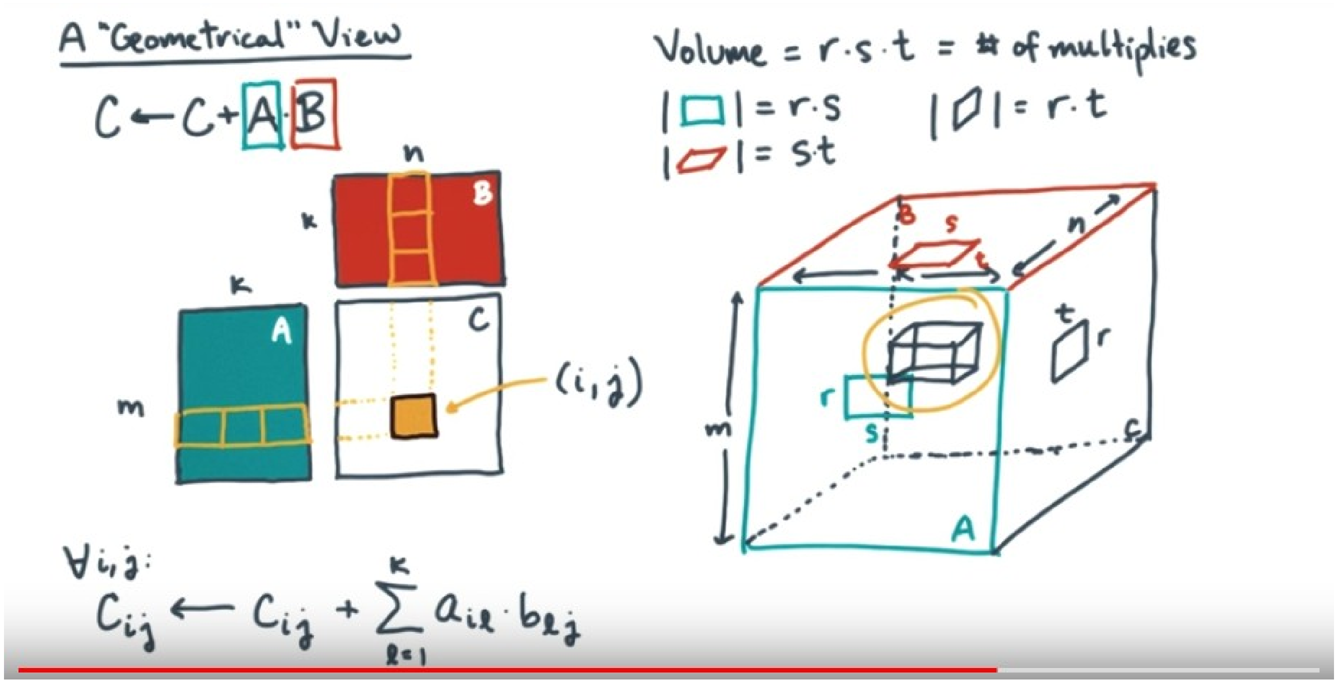
4. Here's the answer. Let's see where it comes from. Remember that each output is computed from a dot product. So, for example, let's take this output. There are two missing values, so let's give them some arbitrary symbols. How about w and x? The dot product is just this equation, basically, the element-wise products summed together equaling the output, and we can repeat this for each of the other output elements. For each blank, I introduced a new symbol. As you can see, we have four equations and four unknowns, so we should be able to find a solution. Let's start with the third equation, which only has one unknown. Solving for z, we get the value 2. We can then plug this into the fourth equation. Solving for y, we get 7, so that eliminates the fourth equation. Since we know y, we can solve the second equation for w. You'll get w equals 1. Eliminating the second leaves us just the first. You'll find x equals 30.



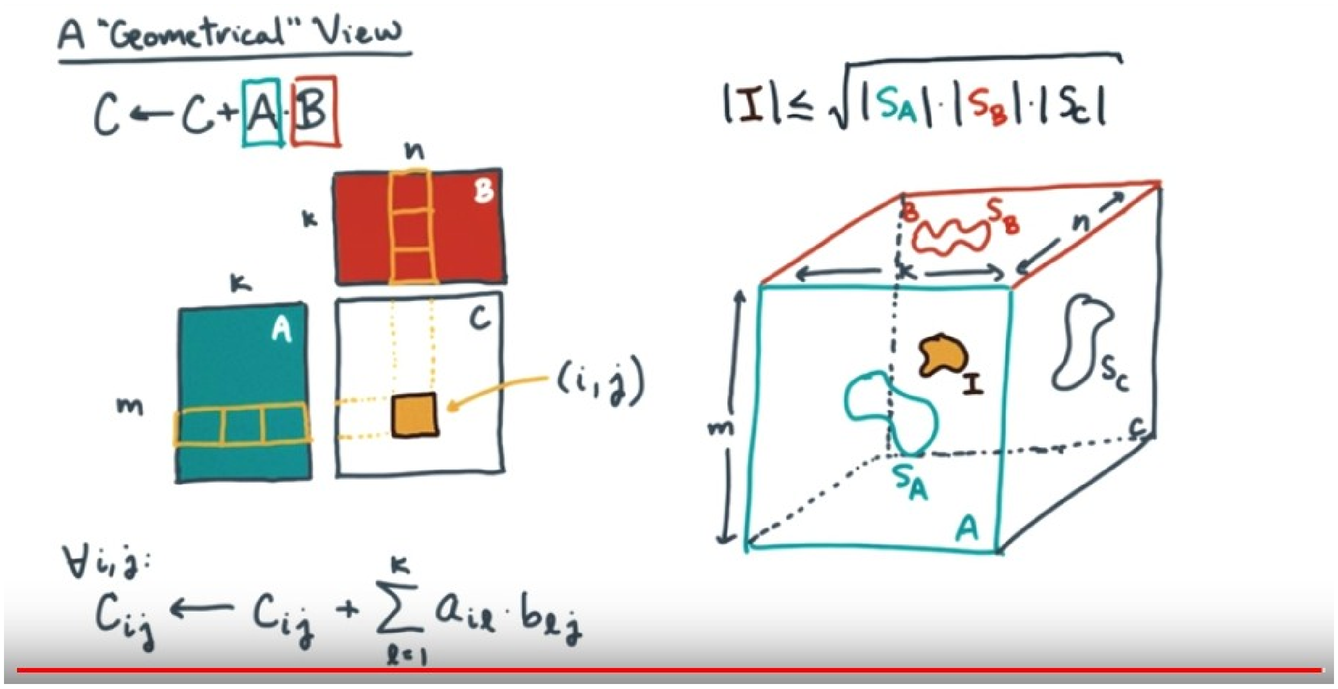
A "Geometrical" View
$C \leftarrow C + \boxed{A} \cdot \boxed{B}$
n
k
B
l
k
A
C
m
l
(i, j)
$\forall i,j:$
$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$
B
l
n
k
l
(i, j)
m
C
l
A

將左圖中的 A 和 B 按進去, 就成了右邊的立體圖了.

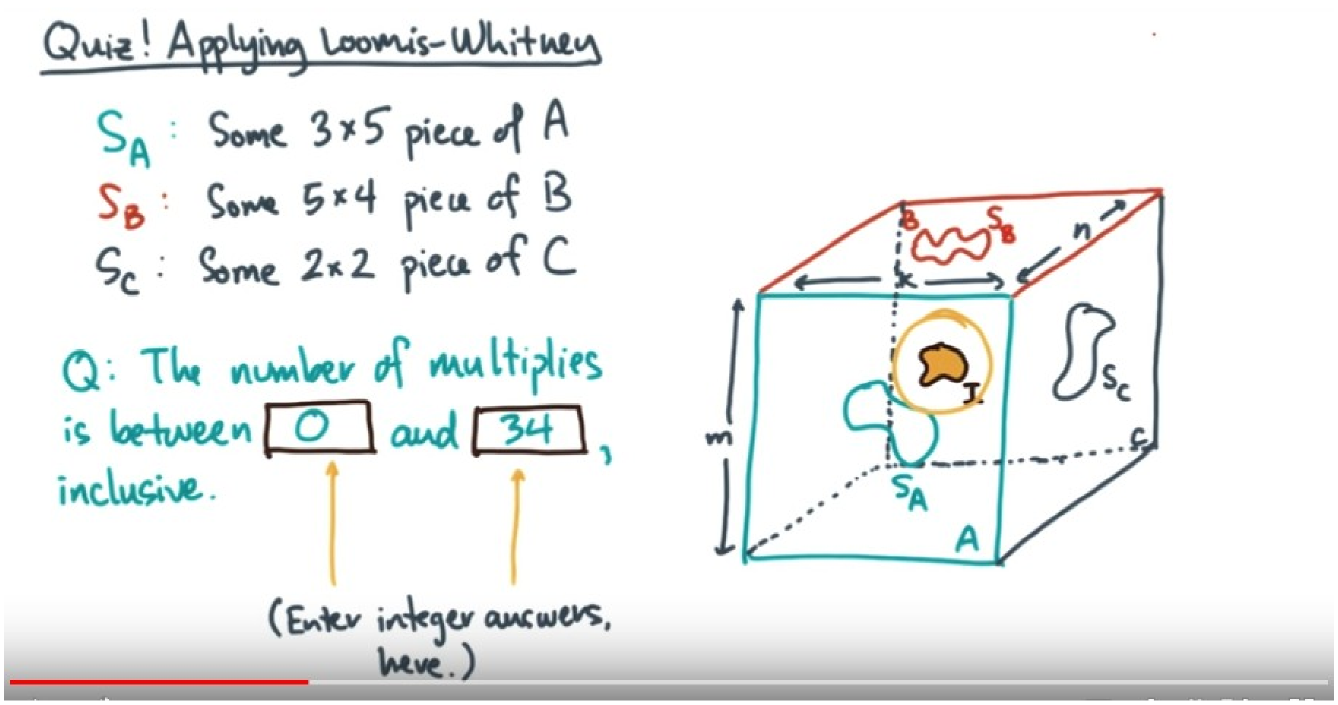
5. While we're on the subject of this sort of picture of a matrix multiply, and I mean the picture on the left, I want to mention a related geometric way of thinking about how a matrix multiply works. This geometric view will give you a handy fact which you'll need later when I ask you, what's the minimum amount of communication a matrix multiply needs? Oh, the intrigue. The intrigue, oh! Starting with this initial picture, I want you to imagine a cuboid where the faces are the matrix operands. Notice how the edges match the same way the dimensions need to match. For example, where A has k columns, and B has k rows. There's a matching edge. The same holds for the columns of B and the corresponding edge of the cube. Columns of B, and corresponding edge. Recall that an element i, j of C depends on row i of A and column j of B. In the cube, there's also an element i, j of C. The dependents on A and B is a line that cuts through here. And if line is a projection of a row from A and a column from B. Now each of these scalar multiplications is like a point inside this cube.


A "Geometrical" View
$C \leftarrow C + \boxed{A} \cdot \boxed{B}$
n
k
B
k
A
C
m
(i, j)
$\forall i, j:$
$c_{ij} \leftarrow c_{ij} + \sum_{\ell=1}^{k} a_{i\ell} \cdot b_{\ell j}$
Volume = r·s·t = # of multiplies
|▭| = r·s
|▱| = s·t
|▱| = r·t
B
s
t
n
k
t
r
r
s
m
C
A

Let's think about what that means. Suppose I pick a set of matching rectangular blocks from A, B and C. So here I picked an r by s block of A, a matching s by t block of B, and of course there's the corresponding matching output which is r by t. If they're perfectly aligned, then they define a volume of the interior of the cube. In other words, this volume is the set of (number of) all multiplications you need to do in order to update this output block with these two surface sub-blocks. How many multiplies is that? Well, it's just the volume of the interior, so that's r * s * t. In terms of the sizes of the sub blocks it's just the square root of their product.


A "Geometrical" View
$C \leftarrow C + \boxed{A} \cdot \boxed{B}$
n
k
B
k
A
C
m
(i, j)
$\forall i, j:$
$c_{ij} \leftarrow c_{ij} + \sum_{\ell=1}^{k} a_{i\ell} \cdot b_{\ell j}$
$|I| \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$
B
S_B
n
k
I
S_C
m
C
S_A
A

In fact, there's a general result from the geometry of discrete lattices like this one, for any integer cube of point, suppose I give you a subset of its surfaces. Let's call them SA, SB, and SC. Now there may be some blob of intersection in the interior, we'll call that I. The theorem is this, the volume of I is at most the square root of the size of A, times the size of B, times the size of C. This is a theorem due to Loomis and Whitney in 1949 and a sighted in the instructors notes.



Quiz! Applying Loomis-Whitney

S_A : Some 3×5 piece of A

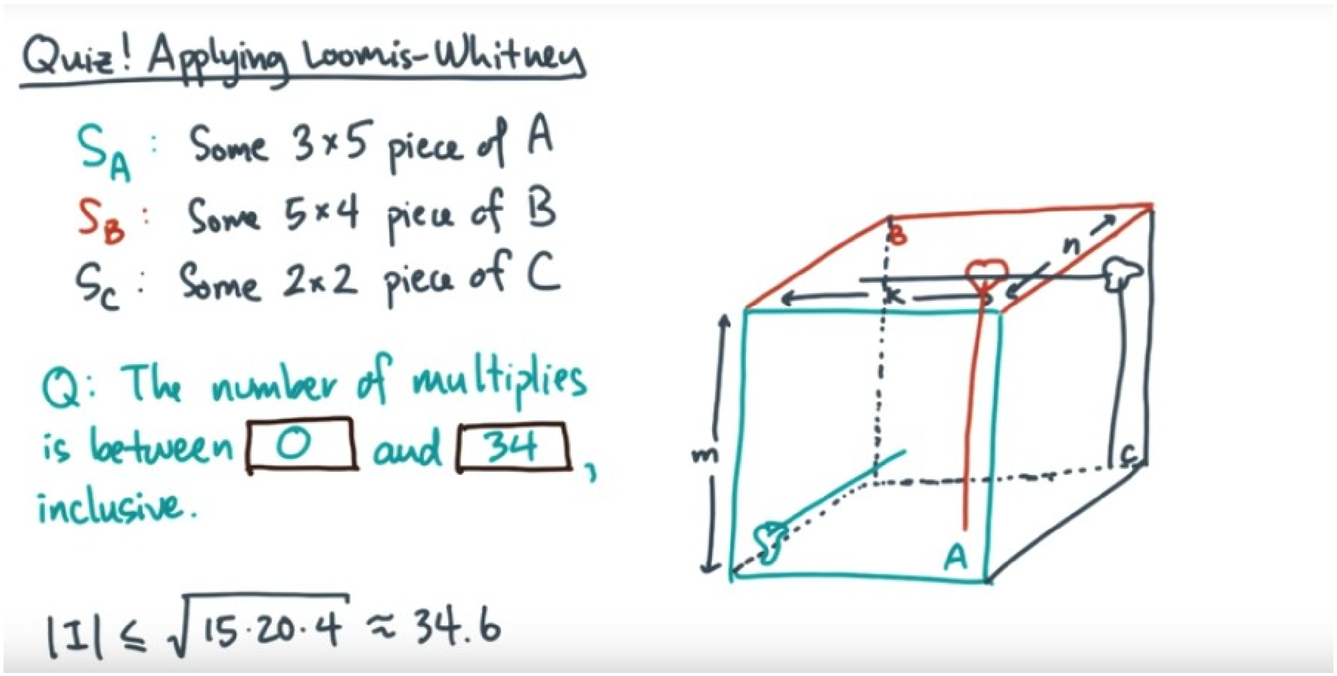
S_B : Some 5×4 piece of B

S_C : Some 2×2 piece of C

Q: The number of multiplies is between $\boxed{0}$ and $\boxed{34}$, inclusive.

(Enter integer answers, here.)

6. Let's reinforce the Loomis-Whitney Theorem. Suppose I assign you part of a matrix multiply. That is, let's say I give you a piece of A, which is 3 x 5, a piece of B which is 5 x 4, and a piece of C which is 2 x 2. Here is my question. What is the minimum and maximum number of multiplies you could possibly do? I want you to fill in these blanks with a suitable integer.



Quiz! Applying Loomis-Whitney

S_A : Some 3×5 piece of A

S_B : Some 5×4 piece of B

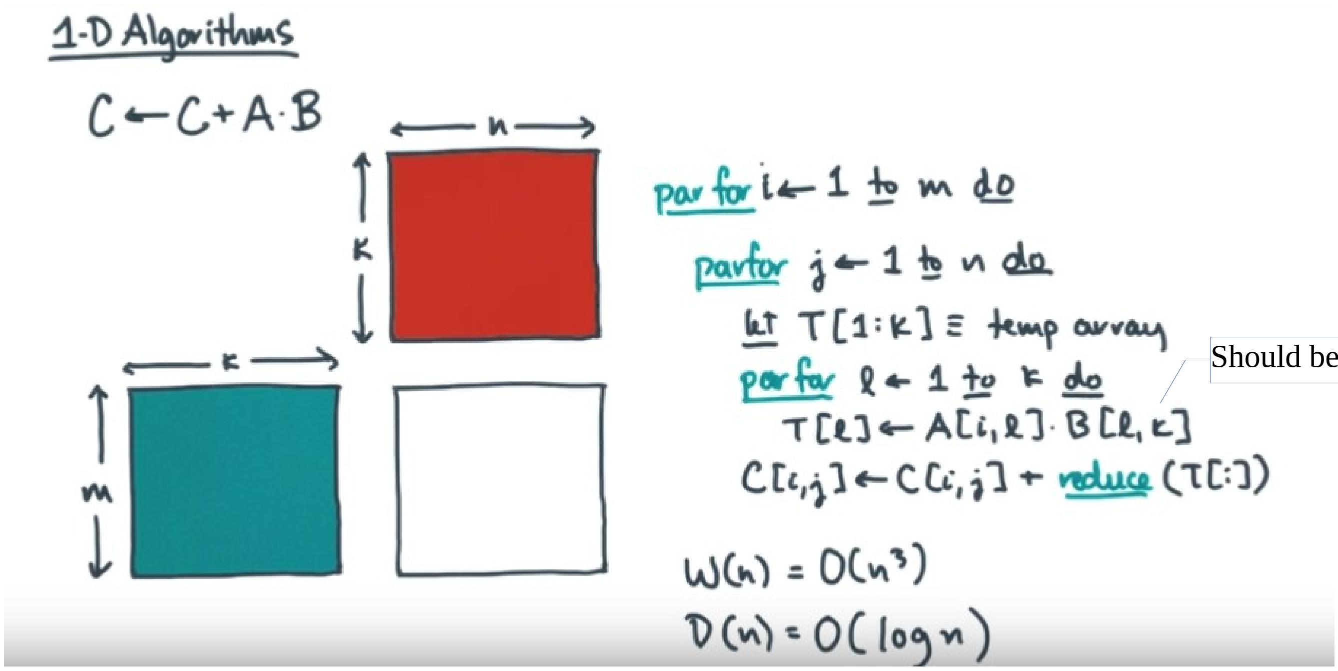
S_C : Some 2×2 piece of C

Q: The number of multiplies is between $\boxed{0}$ and $\boxed{34}$, inclusive.

$$|I| \leq \sqrt{15 \cdot 20 \cdot 4} \approx 34.6$$

7. Here's the answer. The minimum possible value is zero and the maximum is 34. The minimum is

zero because there might not be any intersection among the elements of the blobs. For example, here are three blobs that are in opposite corners of their respective faces. And although the perspective is totally messed up hopefully you can kind of believe that. They don't intersect. The maximum would occur under some best case alignment of the three operands. The Loomis Whitney theorem gives us an upper bound. If you take the square root of the product and the sizes, you'll get about 34.



1-D Algorithms

$C \leftarrow C + A \cdot B$

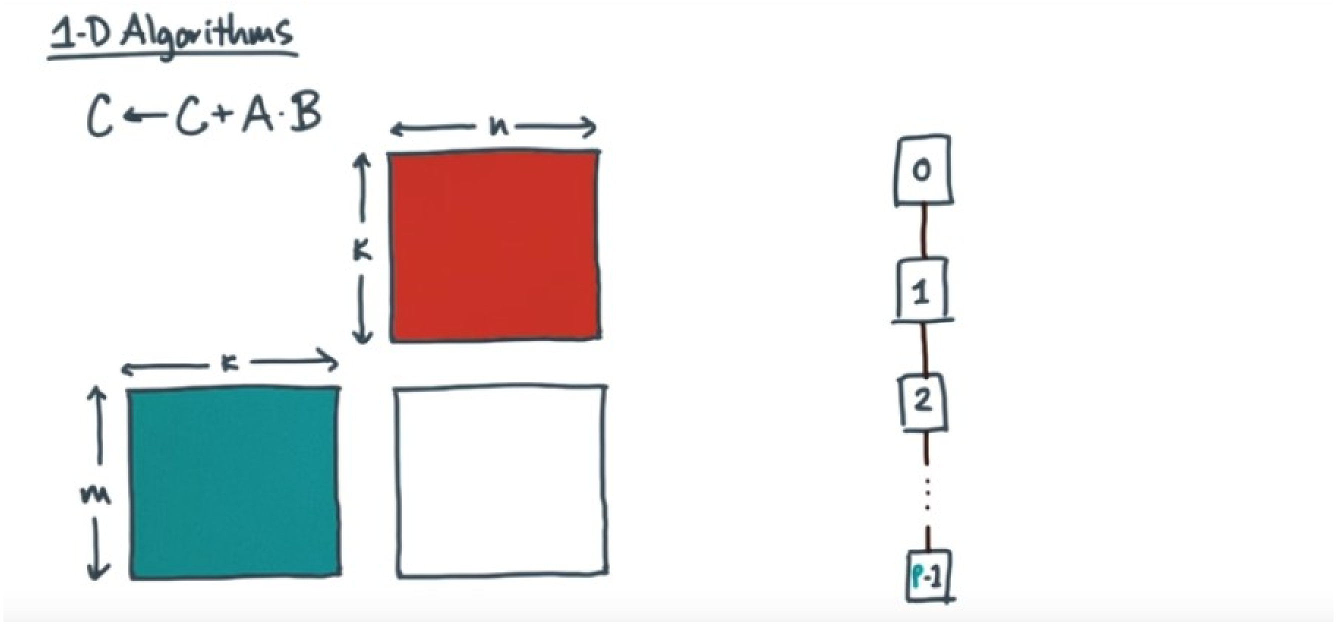
```
par for i ← 1 to m do
  parfor j ← 1 to n do
    let T[1:k] ≡ temp array
    parfor ℓ ← 1 to k do
      T[ℓ] ← A[i,ℓ]·B[ℓ,k]
    C[i,j] ← C[i,j] + reduce(T[:])
```

Should be j

$W(n) = O(n^3)$

$D(n) = O(\log n)$

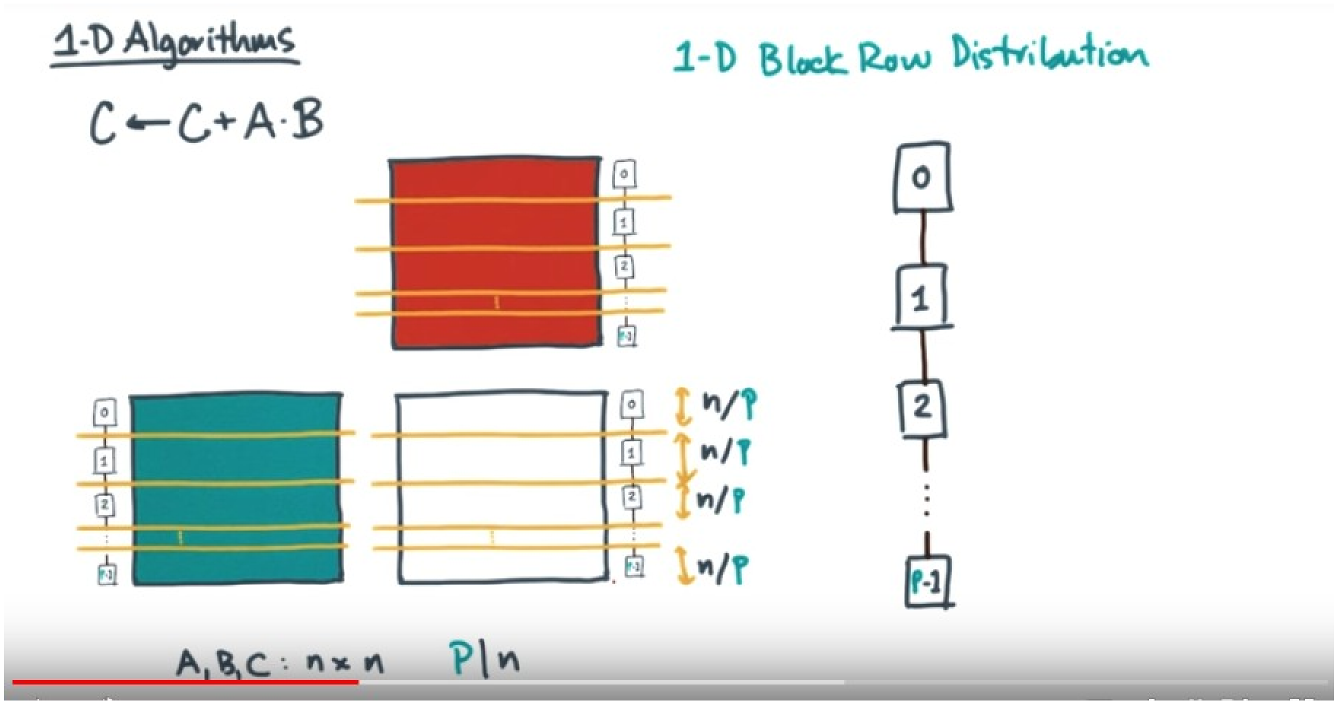
8. So a multi-threaded DAG algorithm for matrix multiply is easy enough to write down as we did before.



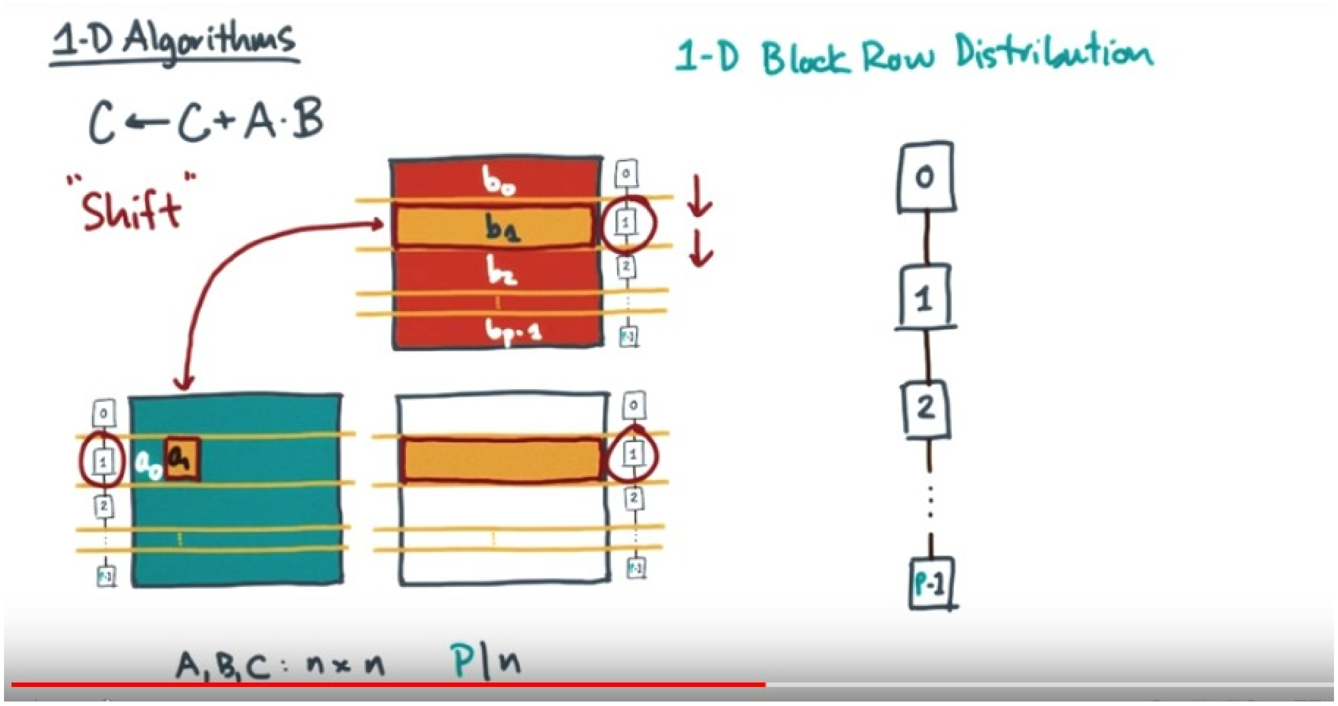
1-D Algorithms

$C \leftarrow C + A \cdot B$

What should you do for a distributed memory machine? Let's start with the case of a linear network having P nodes.



For simplicity, let's also assume that the matrixes are squared n by n objects and that P divides n. The first question is, how should you distribute the matrix operands? There are lots of potential choices. I want you to start with the following convention, distribute the operands by block row in an identical fashion. That means you give each node n/P consecutive rows of each matrix operand. Let me draw a picture. For each node, I've assigned n/P consecutive rows. Notice I've used the same distribution for each matrix. While it's not strictly necessary to do so, it simplifies the design of practical interfaces to the matrix objects, because essentially you can always assume they have the same distribution.

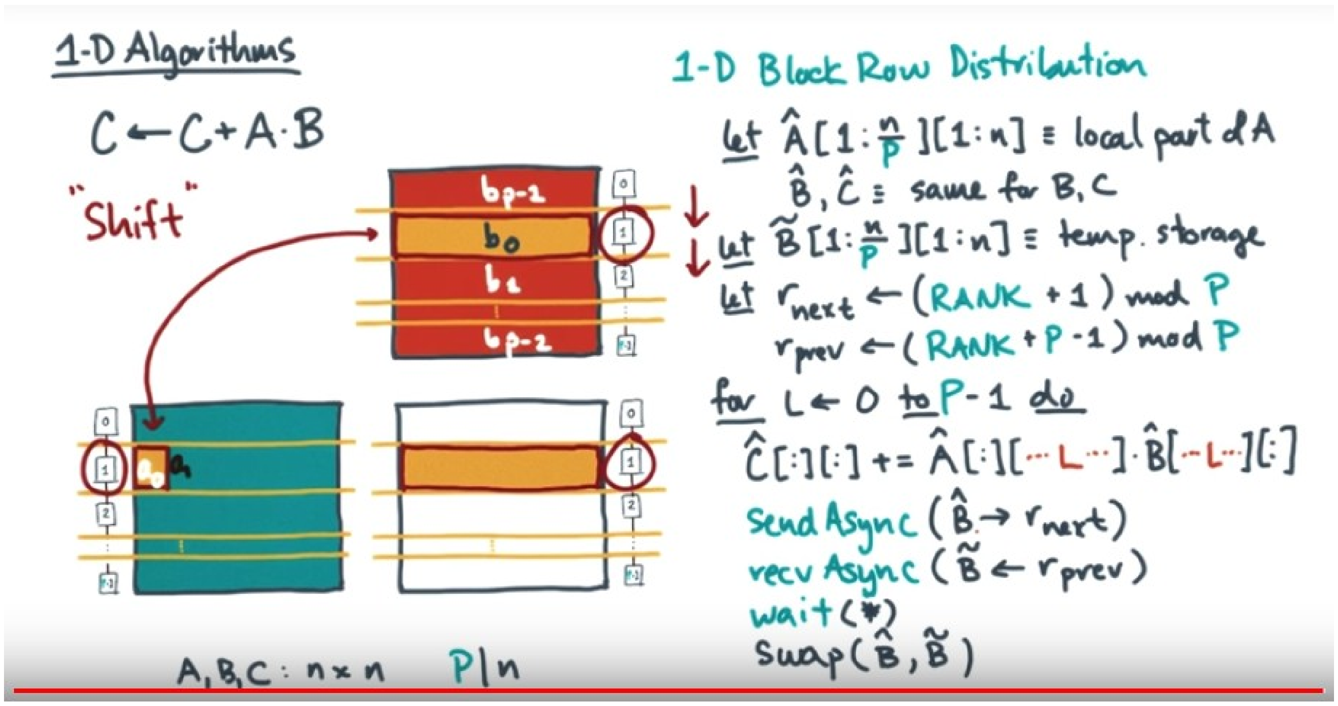


看後面算法之前, 先看我寫的: 上圖中, 我們的目的是得到 白色 matrix 中的那條黃帶(稱為小白)
將 綠色 matrix 中紅圈對到的那一行(即 a0, a1, ...)稱為小綠. 將小綠豎起來(稱為小豎綠).
小豎綠 點乘 紅色 matrix 中的左起第一列, 就得到小白中的左起第一個元素.
小豎綠 點乘 紅色 matrix 中的左起第二列, 就得到小白中的左起第二個元素.
…...
這樣就算完了(得到了完整的小白).

但由於 紅色 matrix 中的每一行是存在不同電腦中的, 故實際上是這麼算的:
小豎綠的最上第一個元素(a0) 跟 紅色 matrix 第一行所有元素(b0)乘一遍, 存在一個數組(E1[....])中
小豎綠的最上第二個元素 (a1)跟 紅色 matrix 第二行所有元素(b1)乘一遍, 存在一個數組(E2[....])中
…
最後 E1[...] + E2[...] + ... 就是要求的小白.

故第一次算的時候(即第一個電腦)需要 b0 行, 第二次算的時候(即第二個電腦)需要 b1 行...
所以文中才想出了往下 shift 的想法.

Now consider some process, what is the most work it can do given this data distribution? In the best case it could update it's entire block row of C using the entire block row of B and one sub block of A. It could not for instance do anything with a sub block of A that's say over here or over here (即那個 sub block 的左邊或右邊). Why not? Well recall the definition of matrix multiply. To fully update this block row of C, you need to multiply this little block of A by this entire block row of B and then this block A by this entire block row of B, and so on (想一想白色 matrix 中的黃帶的最左邊的元素是怎麼算出來的即知). So on. But, given this distribution, this process doesn't have access to this block row labeled b0. So to help the algorithm make progress, you're going to need to shuffle blocks around. Now again, there are many choices. One convention is in order compute strategy in which you keep the block of C in place. You also keep A in place. If you do that, then you have to move B around. Since we're on a linear network, one strategy is to shift the block rows of B. For example, if I were to shift B downwards (即 B 中每一行都往下移一行, 最下一行移到最上面去), then I would get the following result.

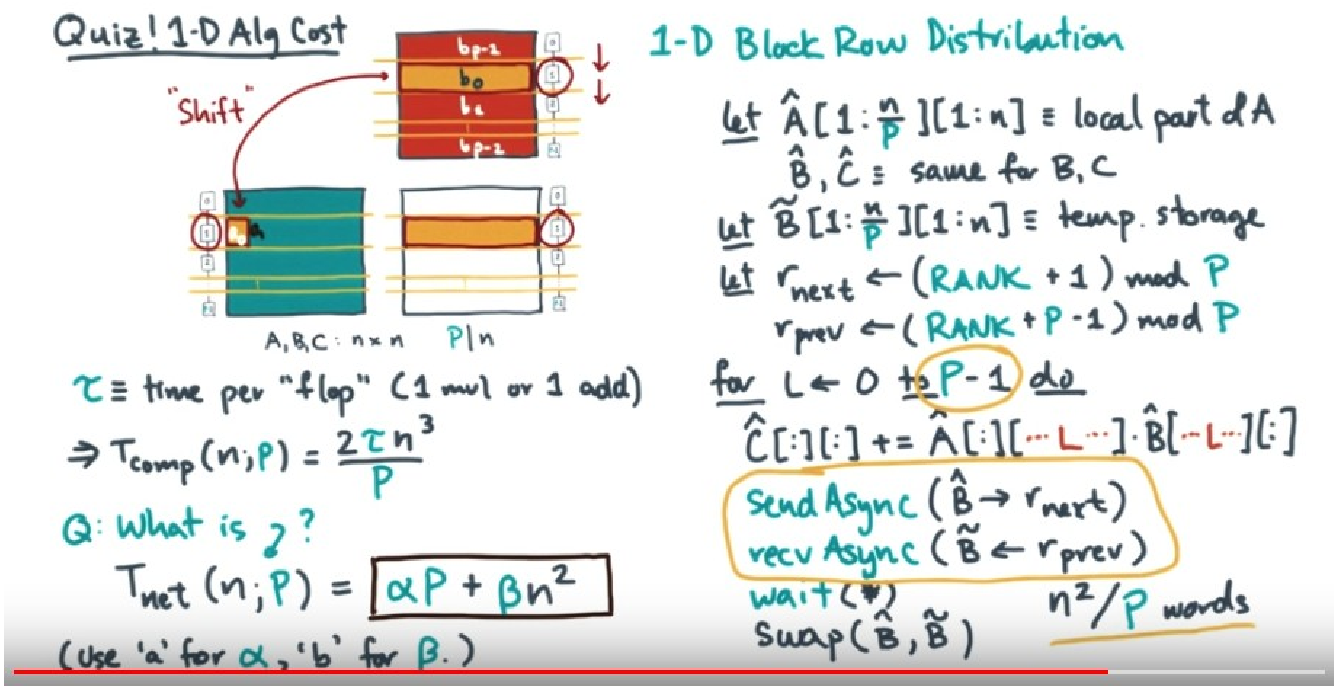


1-D Block Row Distribution

```
let $\hat{A}[1:\frac{n}{P}][1:n]$ ≡ local part of A
    $\hat{B}, \hat{C}$ ≡ same for B, C
let $\tilde{B}[1:\frac{n}{P}][1:n]$ ≡ temp. storage
let $r_{next} \leftarrow$ (RANK + 1) mod P
    $r_{prev} \leftarrow$ (RANK + P - 1) mod P
for L ← 0 to P-1 do
    $\hat{C}[:][:]$ += $\hat{A}[:][\cdots L \cdots] \cdot \hat{B}[\cdots L \cdots][:]$
    sendAsync($\hat{B} \rightarrow r_{next}$)
    recvAsync($\tilde{B} \leftarrow r_{prev}$)
    wait(*)
    swap($\hat{B}, \tilde{B}$)
```

B^和 B~都是當前 node 中的數組.

Now that this node has b0, it can do a partial update with the block a0. If you keep circularly shifting, eventually, every node will see all of B. That's a total of P shifts, which by the way, leaves B in its original distribution once you're done. Very convenient. Let's write down some pseudo code to express the algorithm. We'll let A hat, B hat, and C hat represent the local parts of A, B, and C. Notice that they're of size n over P by n. Now to implement the shift we're going to need to send the block that we have and we're going to need some space to receive a new block row of B. So here I've declared another little temporary storage matrix B twiddle. These two variables compute the neighbor (node) below me, r next, and the neighbor above me, r previous. The loop iterates over circular shifts. We can start by doing a local matrix multiply on the data we have. To keep the pseudo-code simple I've used placeholders, where you would compute the appropriate indices of the local variables as a function of L. Following the partial update is the communication. Step 1 is to send the local buffer to the next processor. Step 2 is to receive from the previous processor. Then we wait for these two communication operations to complete. So, that ends one circular shift operation. And, finally, before moving on to the next round, we just swap the temporary buffer used to receive with the current local buffer used to compute. Easy breezy.



Quiz! 1-D Alg Cost

$\tau \equiv$ time per "flop" (1 mul or 1 add)

$\Rightarrow T_{comp}(n;P) = \frac{2\tau n^3}{P}$

Q: What is ?

$T_{net}(n;P) = \boxed{\alpha P + \beta n^2}$

(Use 'a' for α, 'b' for β.)

1-D Block Row Distribution

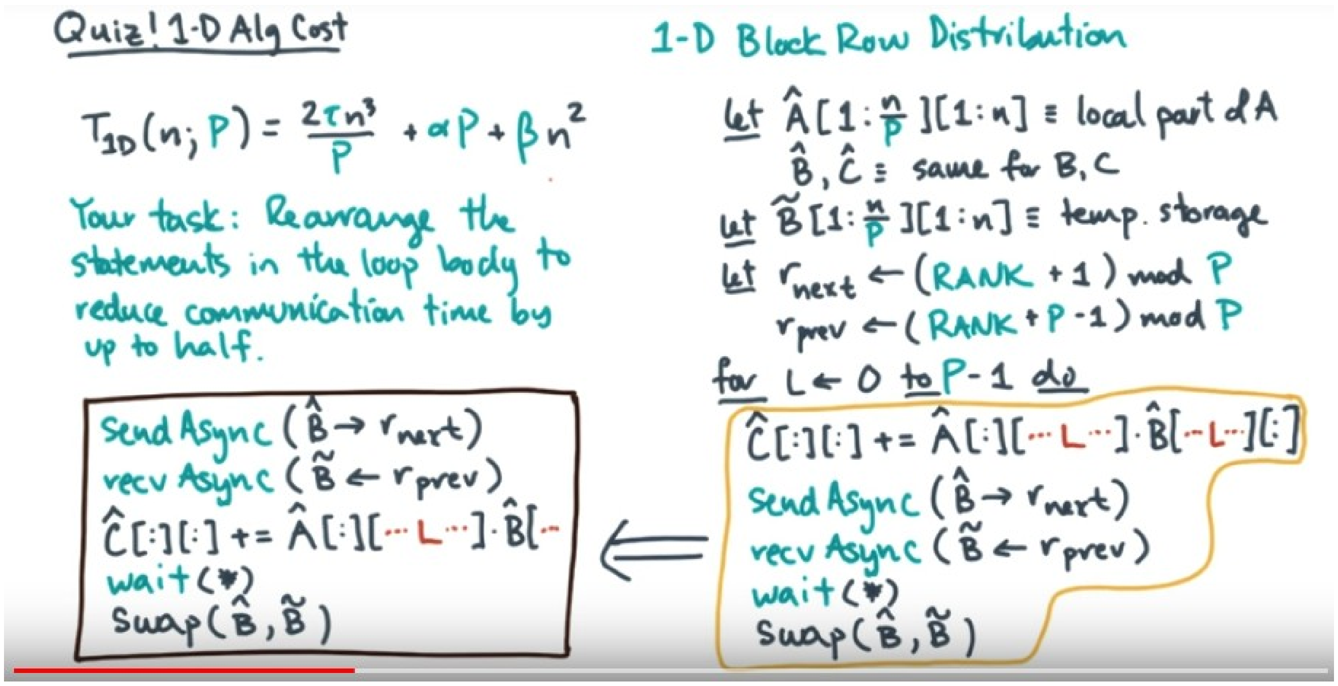
```
let $\hat{A}[1:\frac{n}{P}][1:n] \equiv$ local part of A
    $\hat{B}, \hat{C} \equiv$ same for B, C
let $\tilde{B}[1:\frac{n}{P}][1:n] \equiv$ temp. storage
let $r_{next} \leftarrow (RANK + 1) \bmod P$
    $r_{prev} \leftarrow (RANK + P - 1) \bmod P$
for $L \leftarrow 0$ to $P-1$ do
    $\hat{C}[:][:] \mathrel{+}= \hat{A}[:][\cdots L \cdots] \cdot \hat{B}[\cdots L \cdots][:]$
    sendAsync($\hat{B} \rightarrow r_{next}$)
    recvAsync($\tilde{B} \leftarrow r_{prev}$)
    wait(*)
    swap($\hat{B}, \tilde{B}$)
```

n^2/P words

由 Lesson 2-1 知:
The time to send a message if it contains n words is: T(n) = α + β n
而此處一個 message 有 n^2/P words, 故 T(n) = α + β n^2/P
而總共要 send P 次, 故總的時間是 T(n; P) = P * T(n) = α P + β n^2

9. Here's the 1-D Block Row algorithm for matrix multiply. Let tau be the time per floating point operation, or flop for short. In this case flop is a shorthand for say a floating point multiply or a floating point add. It does not mean a terrible movie. So if tau is the time per flop, then the total time this algorithm spends doing multiplies and adds is 2 tau n cubed / p (2 是因為有 both multiply 和 add). Pause to convince yourself that this is true before you move on. Here's my question. What is the time spent on communication? I want you to type your answer symbolically here. As usual, assume our alpha beta model of communication. Type a little a for alpha, and little b for beta.

10. Here's how to get the solution. In any iteration of the for loop the only data communicated is B hat. It's size is n over p words by n columns. So that's n squared over p words in total. Now there are p rounds of communication because of the loop bound. Therefore, you have to pay for p sends each of sides n squared over p. So that would be a total communication time of alpha times p times beta times n squared.



Quiz! 1-D Alg Cost

$T_{1D}(n;P) = \frac{2\tau n^3}{P} + \alpha P + \beta n^2$

Your task: Rearrange the statements in the loop body to reduce communication time by up to half.

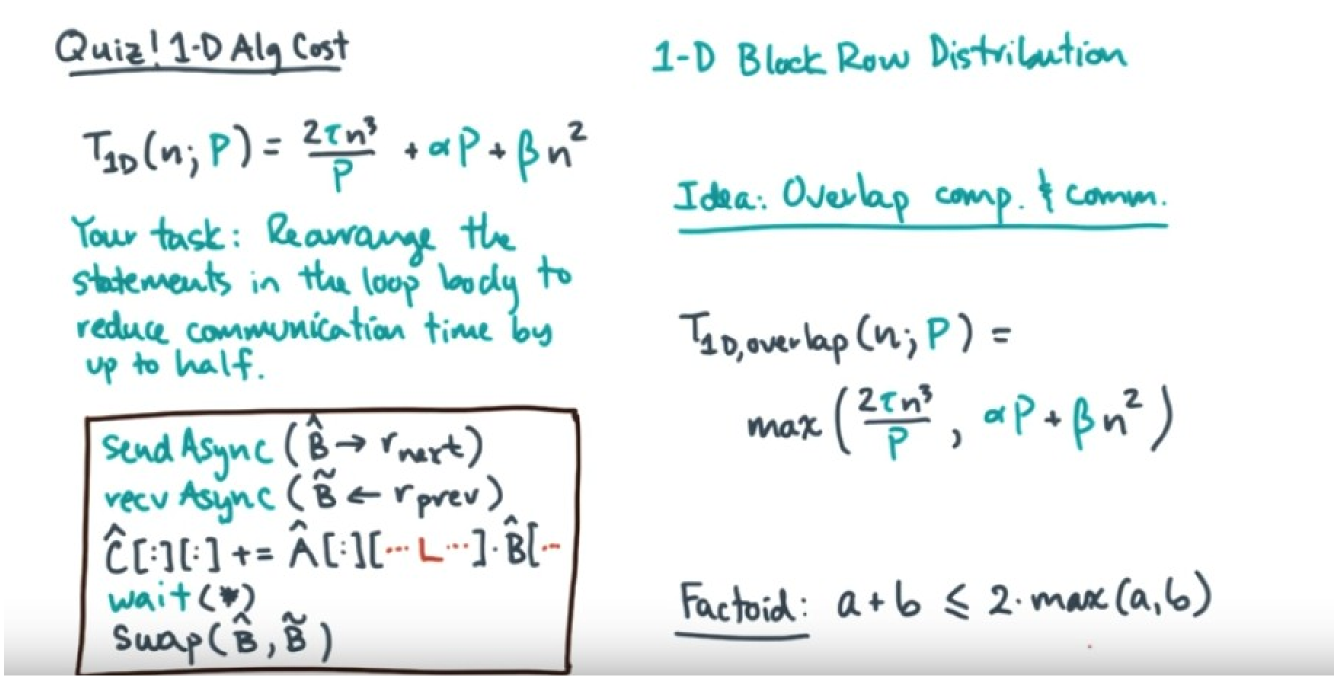
```
SendAsync($\hat{B} \to r_{next}$)
recvAsync($\tilde{B} \leftarrow r_{prev}$)
$\hat{C}[:][:] \mathrel{+}= \hat{A}[:][\cdots L \cdots] \cdot \hat{B}[\cdots$
wait(*)
Swap($\hat{B}, \tilde{B}$)
```

1-D Block Row Distribution

```
let $\hat{A}[1:\frac{n}{P}][1:n] \equiv$ local part of A
    $\hat{B}, \hat{C} \equiv$ same for B, C
let $\tilde{B}[1:\frac{n}{P}][1:n] \equiv$ temp. storage
let $r_{next} \leftarrow$ (RANK + 1) mod P
    $r_{prev} \leftarrow$ (RANK + P - 1) mod P
for L ← 0 to P-1 do
    $\hat{C}[:][:] \mathrel{+}= \hat{A}[:][\cdots L \cdots] \cdot \hat{B}[\cdots L \cdots][:]$
    SendAsync($\hat{B} \to r_{next}$)
    recvAsync($\tilde{B} \leftarrow r_{prev}$)
    wait(*)
    Swap($\hat{B}, \tilde{B}$)
```

11. Recall the 1-D block row algorithm and its pseudocode. You know that it's running time is (2 tau n cubed) / P, for flops, + alpha P + beta n squared, for communication. In fact, you can make this code a little bit faster. All you have to do is rearrange the statements of the loop body. Notice, I said rearrange the statements of the loop body. You don't have to actually change the statements themselves. In the best case, the rearrangement I have in mind might improve the performance by up to a factor of 2 (後面會說為何是 2, 以及為何是 up to), relative to this running time. Your task is to show me how. I will pre-fill this text box with these statements, and I want you to rearrange them to get this possible factor of 2 improvement in speed.

12. Here's the solution I had in mind. Now this running time charges you separately for flop time and communication time, but, in fact, doing the local multiply and doing the sends and receives are completely independent. So a natural idea is to try to overlap them. That suggests that I start the communication. While the communication is pending, do the matrix multiply and then wait for the communication to complete. So that's it, overlap, computation and communication.



Quiz! 1-D Alg Cost

$T_{1D}(n;P) = \frac{2\tau n^3}{P} + \alpha P + \beta n^2$

Your task: Rearrange the statements in the loop body to reduce communication time by up to half.

```
sendAsync ($\hat{B} \rightarrow r_{next}$)
recvAsync ($\tilde{B} \leftarrow r_{prev}$)
$\hat{C}$[:][:] += $\hat{A}$[:][$\cdots L \cdots$]$\cdot\hat{B}$[$\cdots$
wait(*)
swap($\hat{B}$, $\tilde{B}$)
```

1-D Block Row Distribution

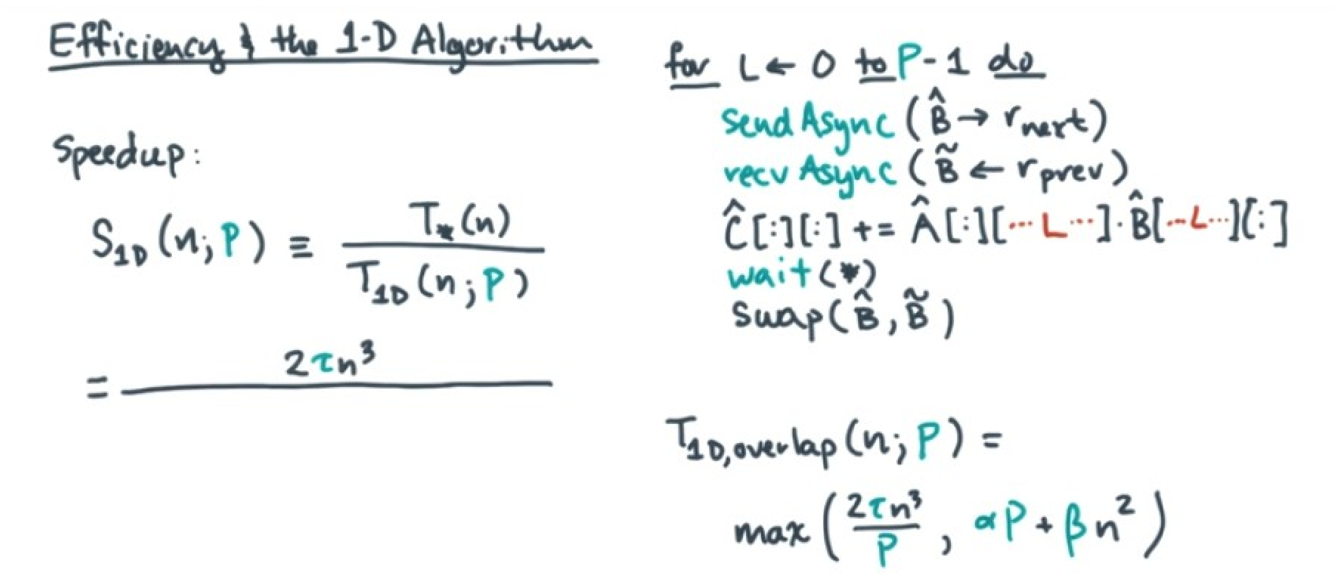
Idea: Overlap comp. & comm.

$T_{1D,overlap}(n;P) = \max\left(\frac{2\tau n^3}{P}, \alpha P + \beta n^2\right)$

Factoid: $a + b \leq 2 \cdot \max(a, b)$

factoid: 仿真陳述

Now, why do I say that might improve the running time by up to a factor of two? Well, if you overlap computation and communication, then the running time should be the max of the time to do the flops against the time it takes to communicate. This is a factor of 2 faster because of an algebraic fact. Namely, the sum of a and b is always less than or equal to twice the max of a and b. That's why I said **up** to a factor of two.



Efficiency & the 1-D Algorithm

Speedup:

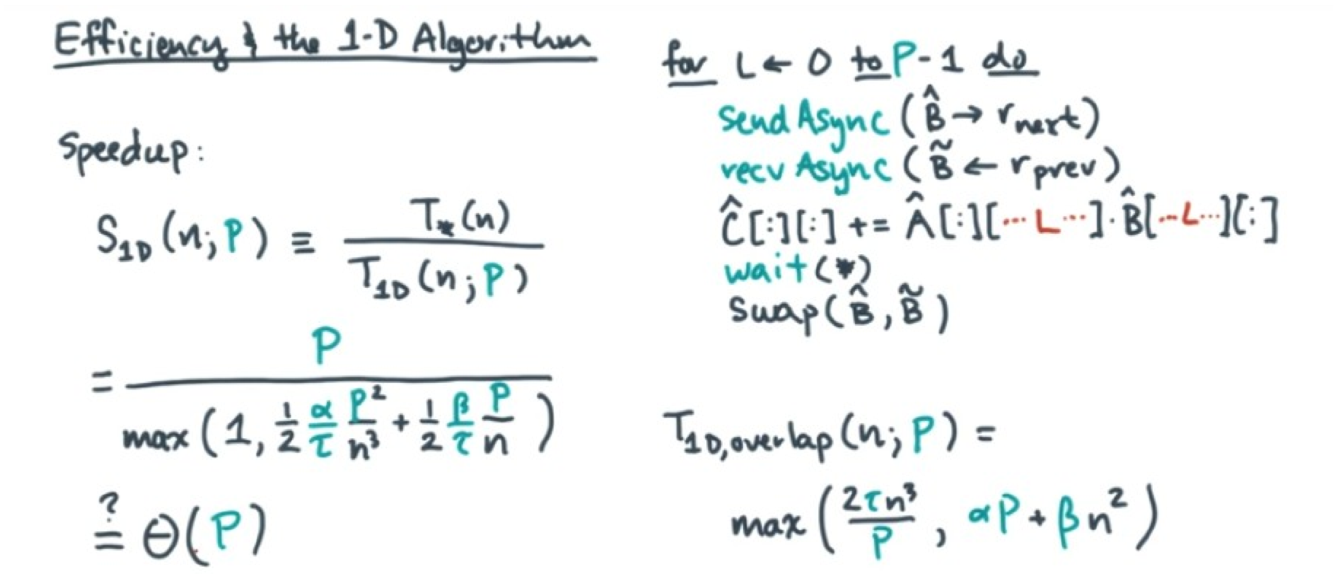
$S_{1D}(n;P) \equiv \frac{T_*(n)}{T_{1D}(n;P)}$

$= \frac{2\tau n^3}{\quad}$

```
for L ← 0 to P-1 do
    sendAsync ($\hat{B} \rightarrow r_{next}$)
    recvAsync ($\tilde{B} \leftarrow r_{prev}$)
    $\hat{C}$[:][:] += $\hat{A}$[:][$\cdots L \cdots$]$\cdot\hat{B}$[$\cdots L \cdots$][:]
    wait(*)
    swap($\hat{B}$, $\tilde{B}$)
```

$T_{1D,overlap}(n;P) = \max\left(\frac{2\tau n^3}{P}, \alpha P + \beta n^2\right)$

13. So there's one D block row algorithm. Is it good or bad? Start by recalling the running time. Now, it's the first algorithm we've talked about in a long lesson, so it has to be bad in some way. Let's pretend to be more analytical. Start by computing the speed up. The best sequential algorithm pays only for flops so its time is just two times tau times n cubed.



Efficiency & the 1-D Algorithm

Speedup:

$$S_{1D}(n;P) \equiv \frac{T_*(n)}{T_{1D}(n;P)}$$

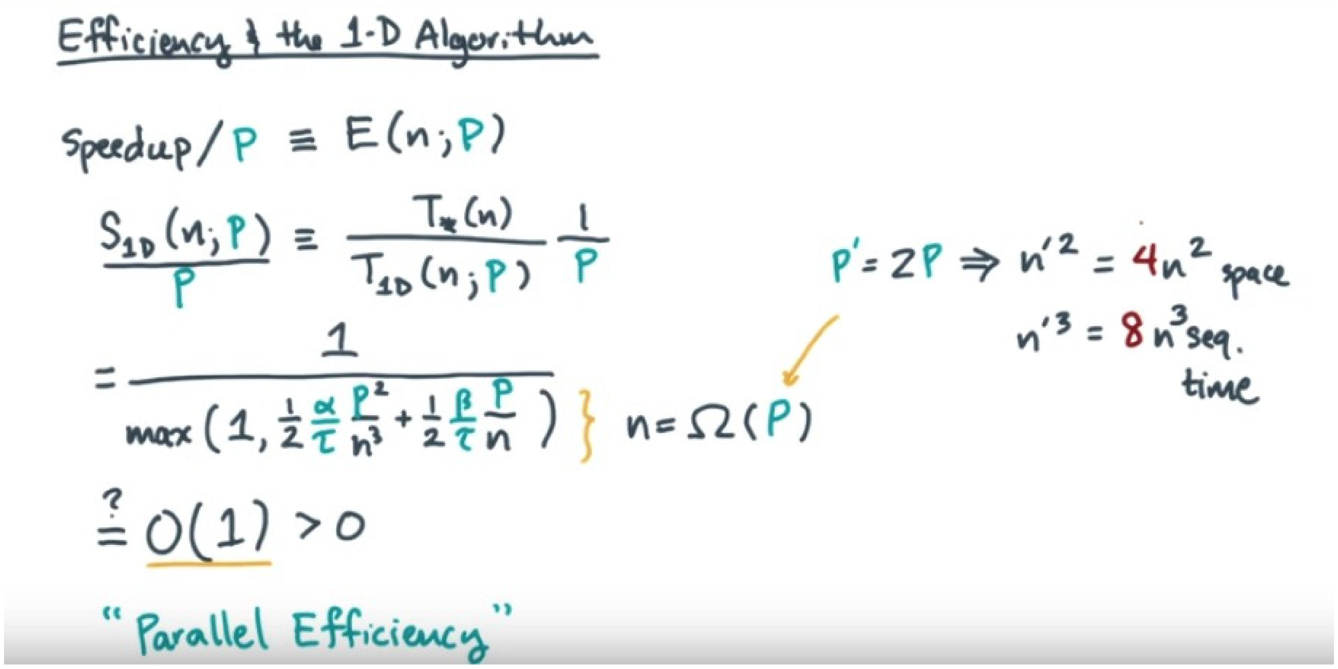
$$= \frac{P}{\max\left(1, \frac{1}{2}\frac{\alpha}{\tau}\frac{P^2}{n^3} + \frac{1}{2}\frac{\beta}{\tau}\frac{P}{n}\right)}$$

$$\overset{?}{=} \Theta(P)$$

```
for L ← 0 to P-1 do
    sendAsync($(\hat{B} \rightarrow r_{next})$)
    recvAsync($(\tilde{B} \leftarrow r_{prev})$)
    $\hat{C}[:][:] \mathrel{+}= \hat{A}[:][\cdots L \cdots] \cdot \hat{B}[\cdots L \cdots][:]$
    wait(*)
    swap($(\hat{B}, \tilde{B})$)
```

$$T_{1D,overlap}(n;P) = \max\left(\frac{2\tau n^3}{P}, \alpha P + \beta n^2\right)$$

Next let's plug in the parallel time for the 1D block row algorithm. Let me massage it into a different form for you. Now this form makes it easy to see how the speed up compares to our ideal, which would be something proportional to p.



Efficiency & the 1-D Algorithm

$$\text{Speedup}/P \equiv E(n;P)$$

$$\frac{S_{1D}(n;P)}{P} \equiv \frac{T_*(n)}{T_{1D}(n;P)}\frac{1}{P}$$

$$= \left.\frac{1}{\max\left(1, \frac{1}{2}\frac{\alpha}{\tau}\frac{P^2}{n^3} + \frac{1}{2}\frac{\beta}{\tau}\frac{P}{n}\right)}\right\} n = \Omega(P)$$

$$\overset{?}{=} \underline{O(1)} > 0$$

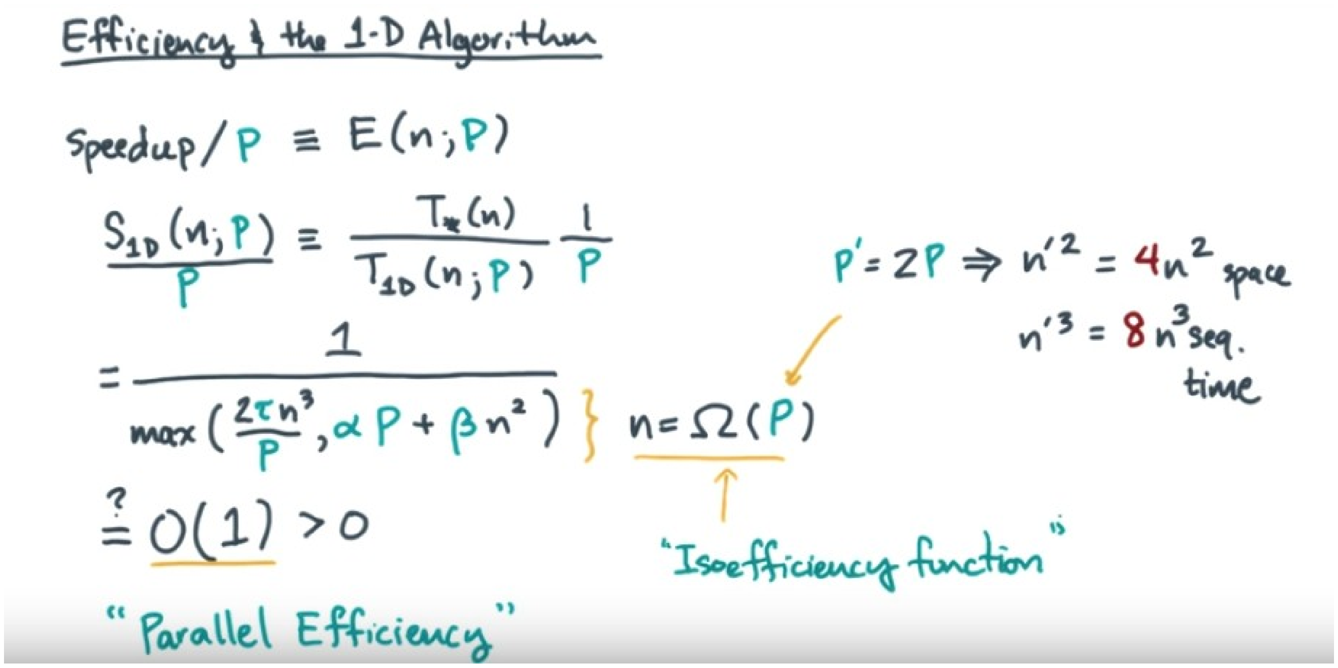
"Parallel Efficiency"

$$P' = 2P \Rightarrow n'^2 = 4n^2 \text{ space}$$

$$n'^3 = 8n^3 \text{ seq. time}$$

From CLRS:
f(n) = Θ(g(n)): c_2 g(n) < f(n) < c_1 g(n)
f(n) = O(g(n)): f(n) < c g(n)
f(n) = Ω(g(n)): f(n) > c g(n)
記憶: xiao 中有 o

That is, if you want to know when the speed up divided by p is a constant. In fact, this concept of speed up divided by p has a special name. It's called **parallel efficiency**, which I'll denote by capital E. So a parallel system is efficient if its parallel efficiency is a constant. And higher constants are better since they correspond to higher fractions of linear speed-up. Now in order to have constant efficiency you need the denominator to be a constant. So when will the denominator approach a constant? In this case when N is big omega of P. Now think about what it means to be efficient in this sense. Let's say you want to double the number of nodes In order to solve the problem faster. Then this condition you've just arrived says you also have to double the problem dimension. But if you double the problem dimension, the size of the matrices will quadruple because they're N by N. Even worse, the number of flops will have to increase by a factor of 8. In other words, if you double the problem dimension, you have to double the amount of memory you need per node and you have to spend more time doing flops. Dang, that's huge. Yes, thank you, that's hilarious Tiny. Put another way, if you don't or can't double the dimension, then you'll quickly see diminishing returns as you increase the parallelism of the system.



Efficiency & the 1-D Algorithm

$$\text{Speedup}/P \equiv E(n;P)$$

$$\frac{S_{1D}(n;P)}{P} \equiv \frac{T_*(n)}{T_{1D}(n;P)} \frac{1}{P}$$

$$= \frac{1}{\max\left(\frac{2\tau n^3}{P}, \alpha P + \beta n^2\right)} \Big\} \; \underline{n = \Omega(P)}$$

$$\overset{?}{=} \underline{O(1)} > 0$$

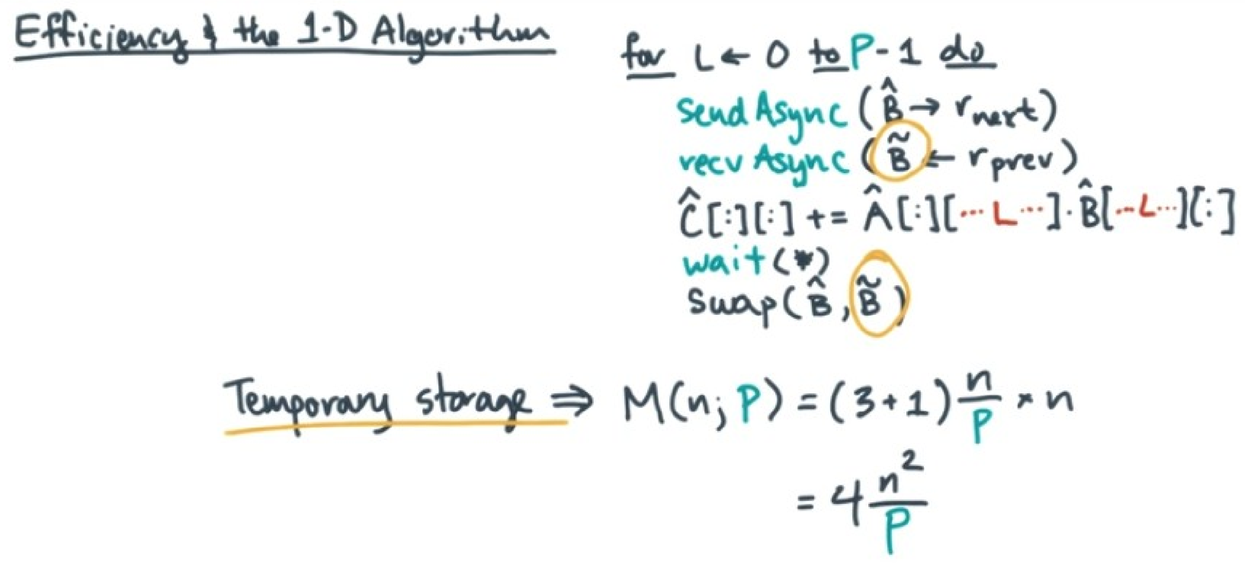
"Parallel Efficiency"

"Isoefficiency function"

$$P' = 2P \Rightarrow n'^2 = 4n^2 \text{ space}$$

$$n'^3 = 8n^3 \text{ seq. time}$$

By the way, this function or this relationship between n and P has a special name. It's called the isoefficiency function. That is, the **isoefficiency** function is the function of p that n has to satisfy in order to have constant parallel efficiency.



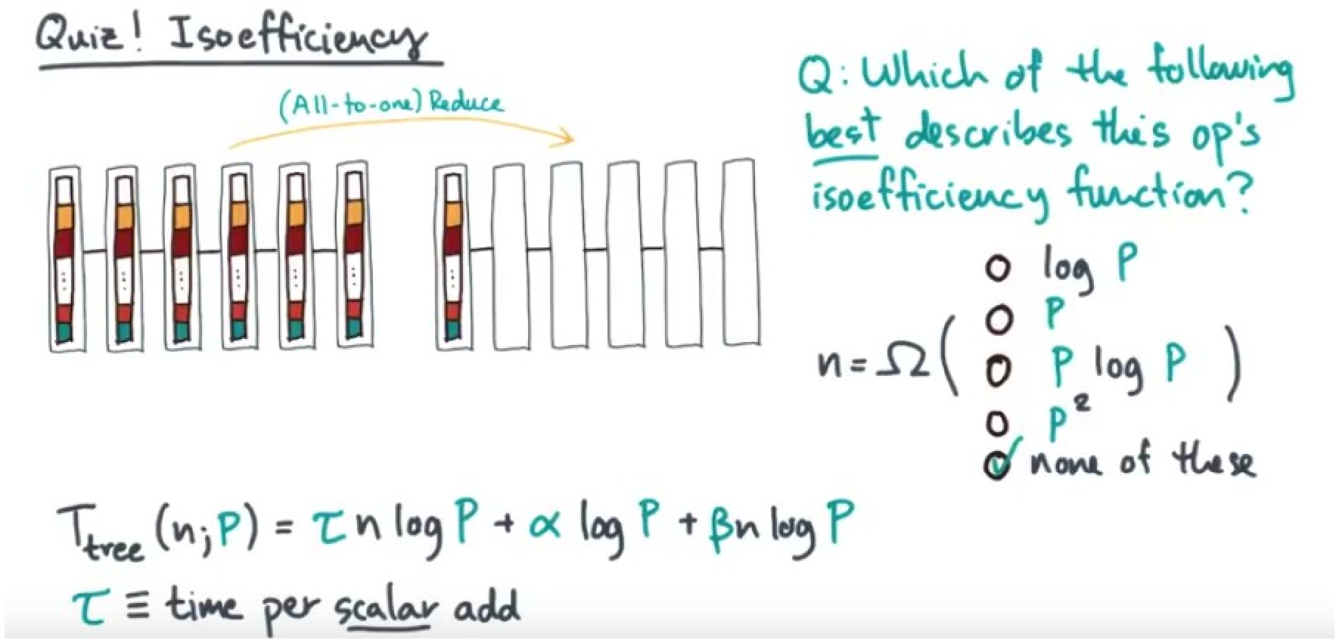
Efficiency & the 1-D Algorithm

```
for $L \leftarrow 0$ to $P-1$ do
    sendAsync($\hat{B} \rightarrow r_{next}$)
    recvAsync($\tilde{B} \leftarrow r_{prev}$)
    $\hat{C}[:][:] \mathrel{+}= \hat{A}[:][\cdots L \cdots] \cdot \hat{B}[\cdots L \cdots][:]$
    wait(*)
    swap($\hat{B}$, $\tilde{B}$)
```

Temporary storage $\Rightarrow$ $M(n;P) = (3+1)\frac{n}{P} \times n$

$$= 4\frac{n^2}{P}$$

Now there's one other consideration, which is the amount of temporary storage space. Remember, you need temporary storage for these B twiddles. So, you have to store the three local matrix upper ends (A^, B^, C^) plus the B twiddle. That gives you four times n squared / p. Okay. So, that gives us a pretty complete analysis of the time, speed up, efficiency and storage requirements of the one D block row algorithm. Good job, you.



Quiz! Isoefficiency

Q: Which of the following best describes this op's isoefficiency function?

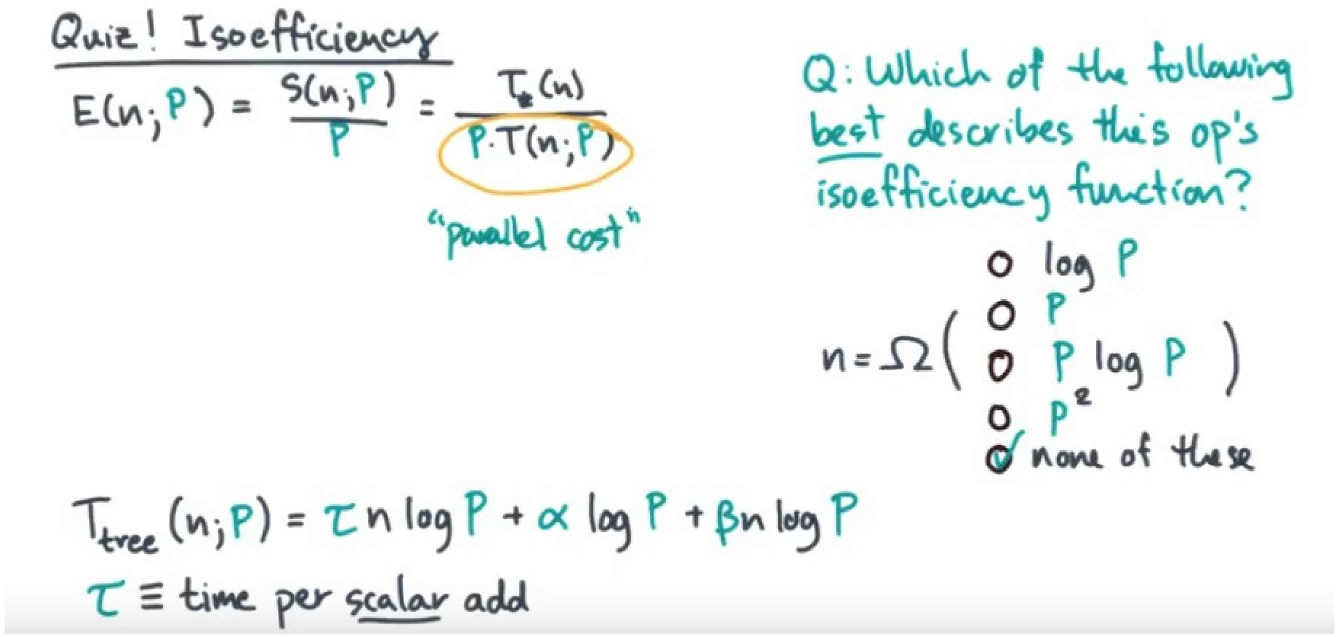
$n = \Omega($
- ○ $\log P$
- ○ P
- ○ $P \log P$
- ○ P^2
- [✓ none of these]

$)$

$$T_{tree}(n;P) = \tau n \log P + \alpha \log P + \beta n \log P$$

$\tau \equiv$ time per scalar add

14. Consider an-all-to one vector reduction. Now suppose tau is the time per scalar addition. Assuming a tree-based scheme this is the time to reduce an input of size N on a linear network. Here is my question. Which of the following best describes the isoefficiency function of a tree-based all-to-one reduce? Here are your choices.



Quiz! Isoefficiency

$$E(n;P) = \frac{S(n;P)}{P} = \frac{T_*(n)}{\boxed{P\cdot T(n;P)}}$$

"parallel cost"

Q: Which of the following best describes this op's isoefficiency function?

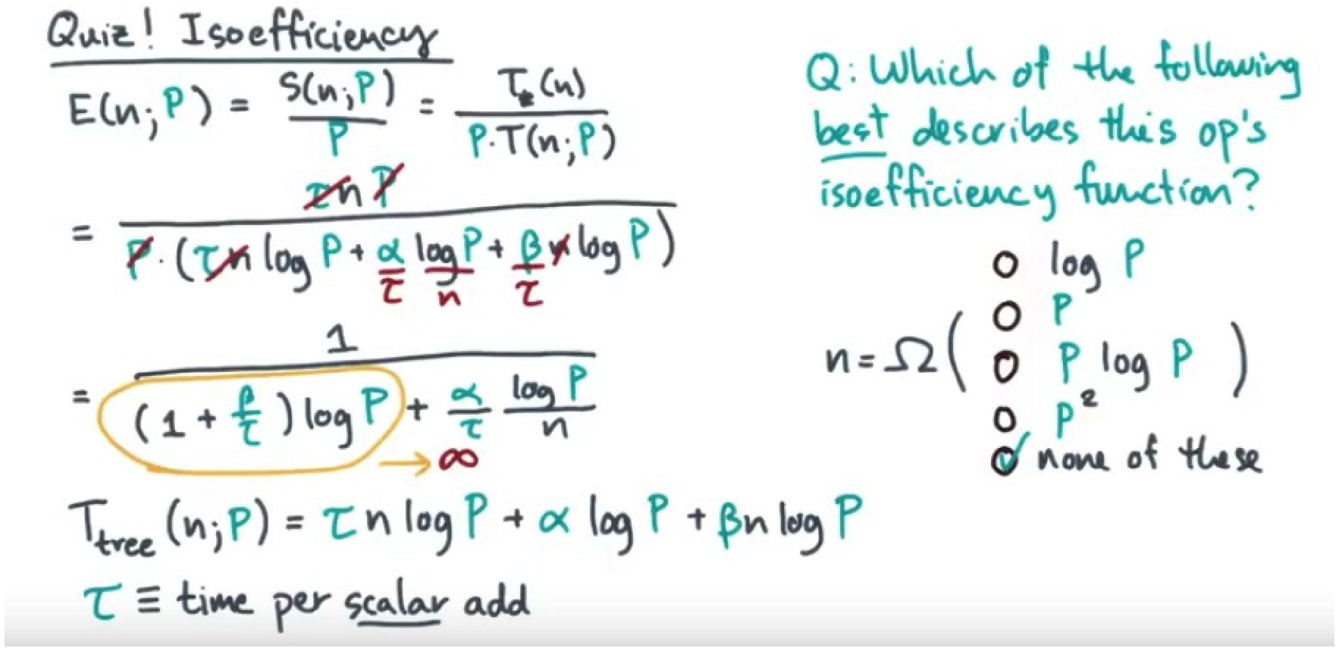
$n = \Omega($
- ○ $\log P$
- ○ P
- ○ $P \log P$
- ○ P^2
- ✓ none of these

$)$

$$T_{tree}(n;P) = \tau n \log P + \alpha \log P + \beta n \log P$$

$\tau \equiv$ time per scalar add

15. The answer is none of these. Start by writing down the efficiency of the algorithm. Remember that's speed up divided by P. By the way, if you expand it, you get this expression. This denominator has a special name. It's called the parallel cost.



Quiz! Isoefficiency

$$E(n;P) = \frac{S(n;P)}{P} = \frac{T_*(n)}{P\cdot T(n;P)}$$

$$= \frac{\cancel{\tau n}\cancel{P}}{\cancel{P}\cdot\left(\cancel{\tau n}\log P + \frac{\alpha}{\tau}\frac{\log P}{n} + \frac{\beta}{\tau}\cancel{n}\log P\right)}$$

$$= \frac{1}{\underbrace{\left(1+\frac{\beta}{\tau}\right)\log P}_{\to\infty} + \frac{\alpha}{\tau}\frac{\log P}{n}}$$

Q: Which of the following best describes this op's isoefficiency function?

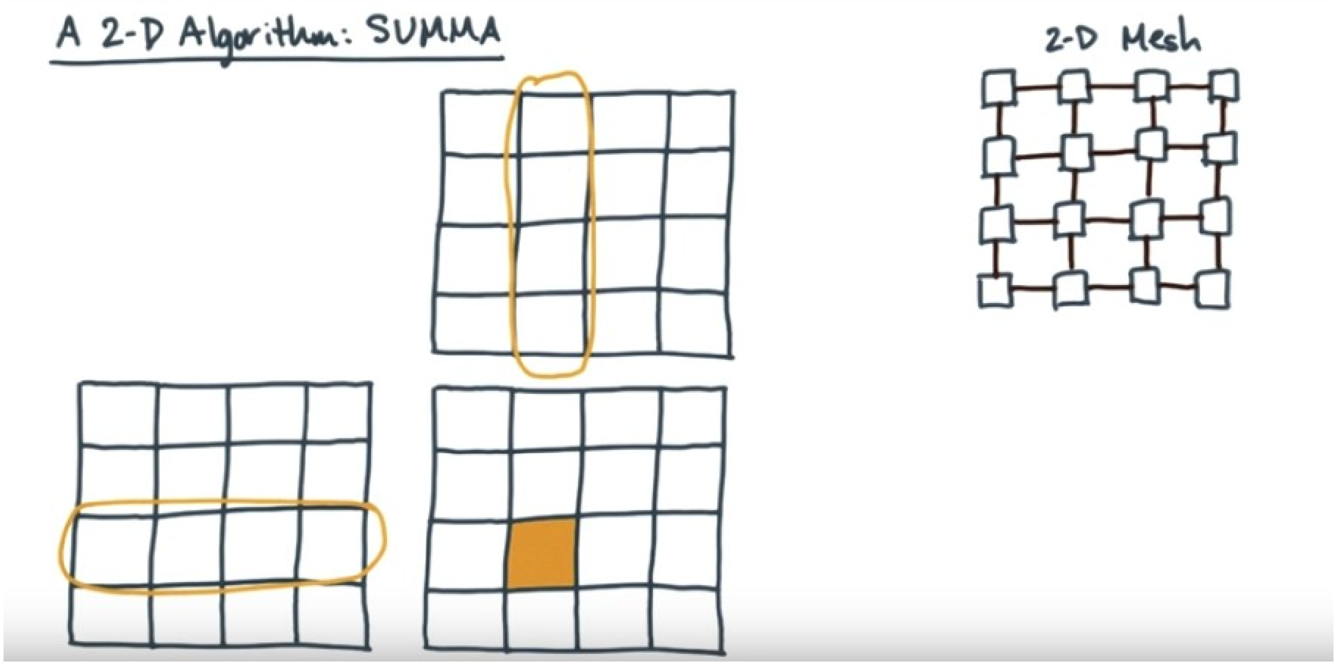
$n = \Omega($
- ○ $\log P$
- ○ P
- ○ $P \log P$
- ○ P^2
- ✓ none of these

$)$

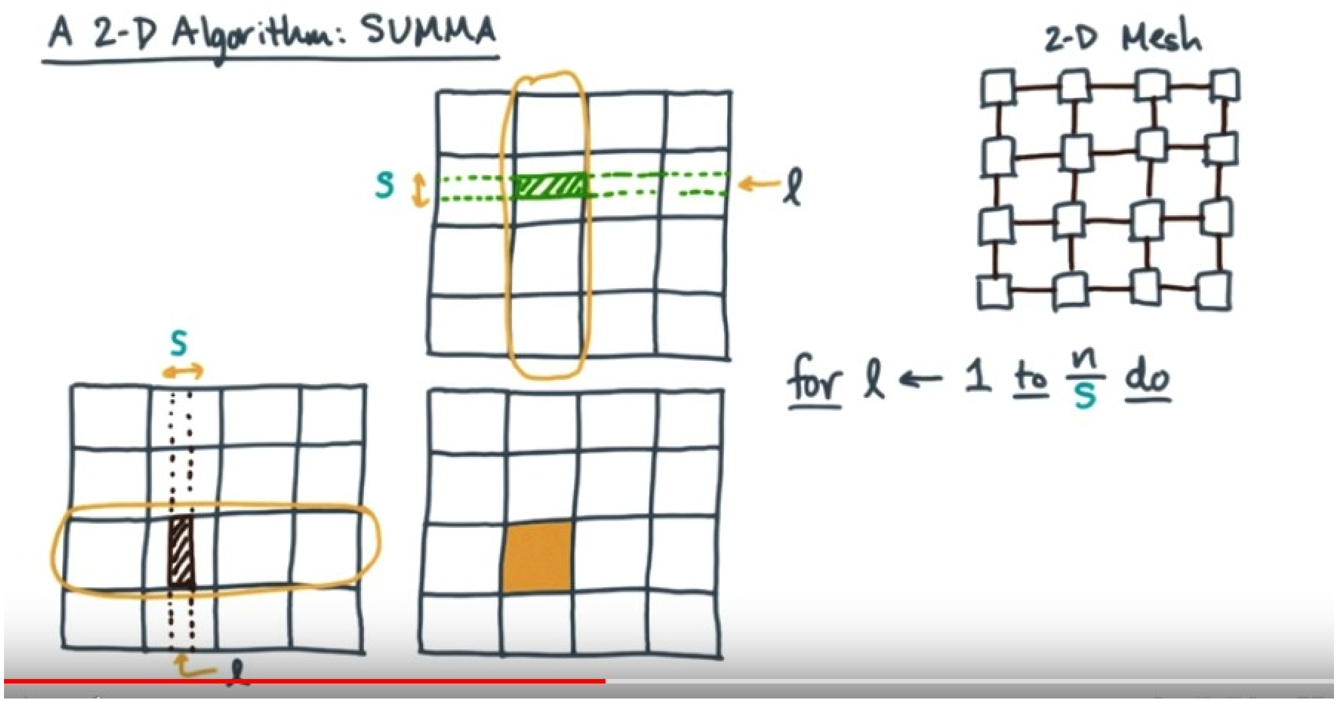
$$T_{tree}(n;P) = \tau n \log P + \alpha \log P + \beta n \log P$$

$\tau \equiv$ time per scalar add

So let's plug in these two times, T star and T of n P. And let's do a little algebra. In this case, I'm going to divide the numerator and denominator by tao times n times P and you should get something like this. Notice that there is no setting of n that will make E of n P a constant. The problem is this first term which goes to infinity as P increases. In other words, the efficiency will always tend to zero. Here's a thought question for you. Is a bucketing or pipeline scheme possible? And if so would it fix the problem? Hmm.

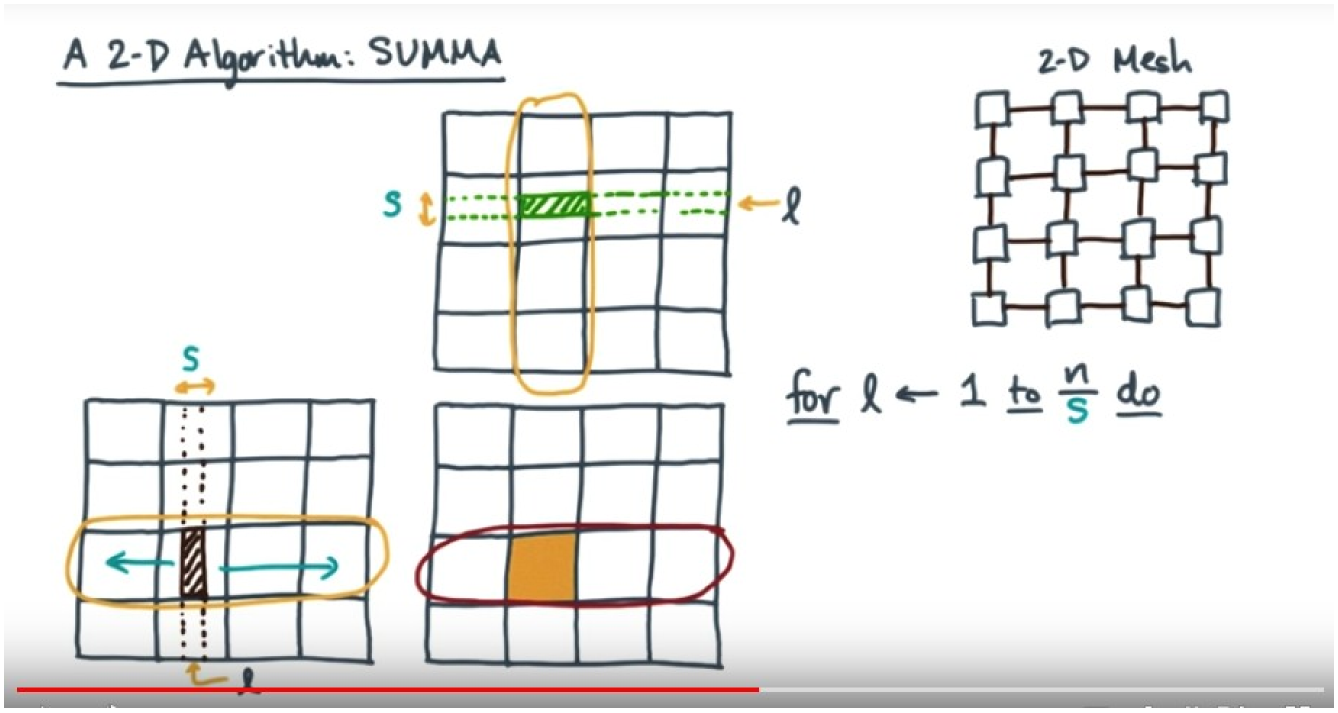


16. So what about a 2-D Algorithm for a 2-D mesh or torus? In fact, somehow intuitively, a 2-D mesh or torus should be a better match for matrix multiply given the fact that matrixes are in fact themselves two dimensional. You can't keep me on a linear network, weeee. Here's an elegant algorithm called the summa algorithm, that's SUMMA (Scalable Universal Matrix Multiply) not sumo. We'll put a reference to a paper on summa in the instructor's notes. SUMMA starts with a 2-D distribution of the matrix operands. Each node is responsible for updating a part of the C matrix that it owns. For example, consider this node. You know that eventually the owner of this block (黃色實心 block) needs to see all of this block row (黃色空心橫的 block) and all of this block column (黃色空心豎的 block) .



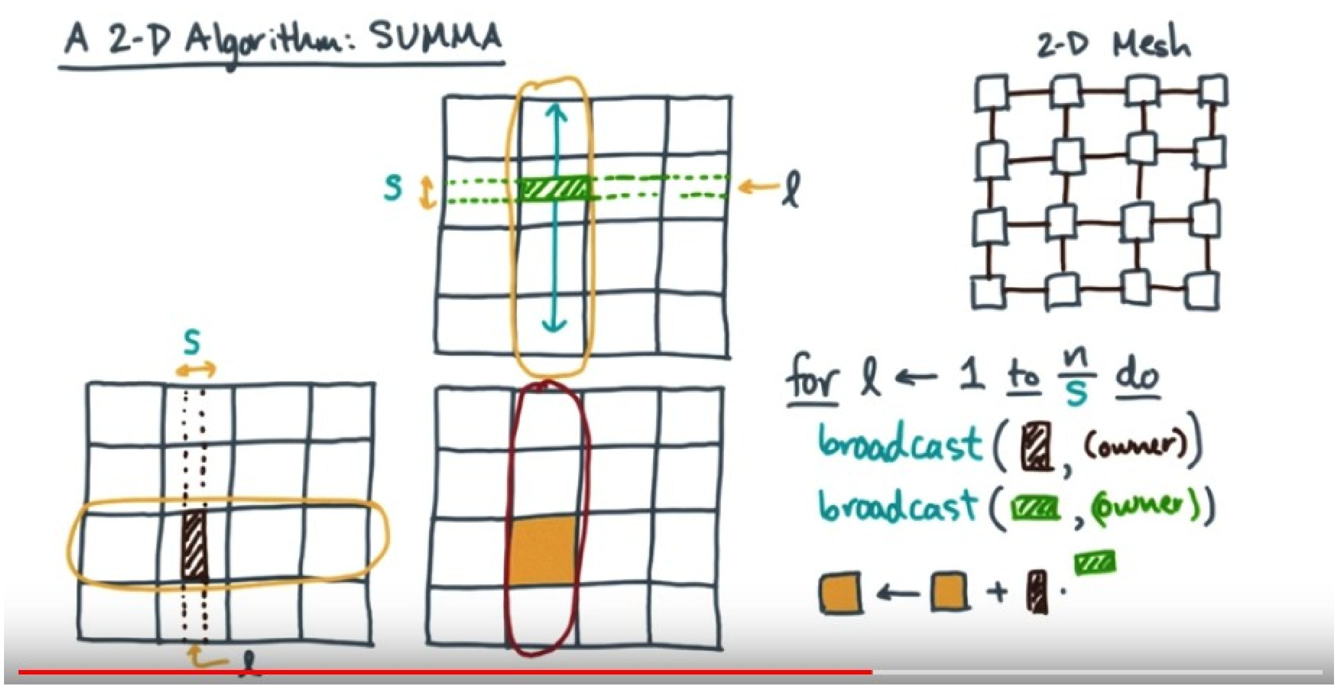
The SUMMA algorithm accomplishes this as follows. First, it iterates over vertical strips (黑色虛線豎

的帶子) of A and the corresponding horizontal strips (黑色虛線橫的帶子) of B. At each step, let's denote the current strip by the index little l. Now the strip can have any width in A or height in B. Let's denote the strip dimension by little s. Algorithmically, you can think of all processes synchronously executing a for loop that iterates over these strips.

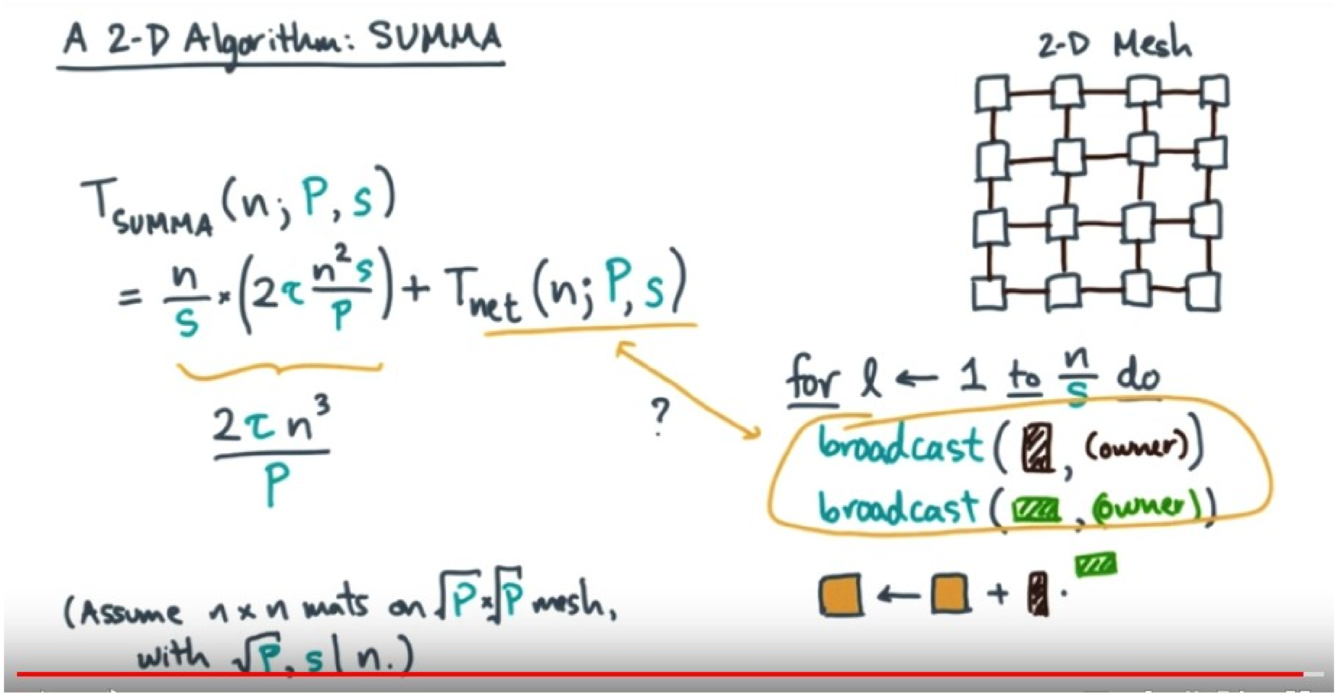


n 是 matrix 的行數或列數

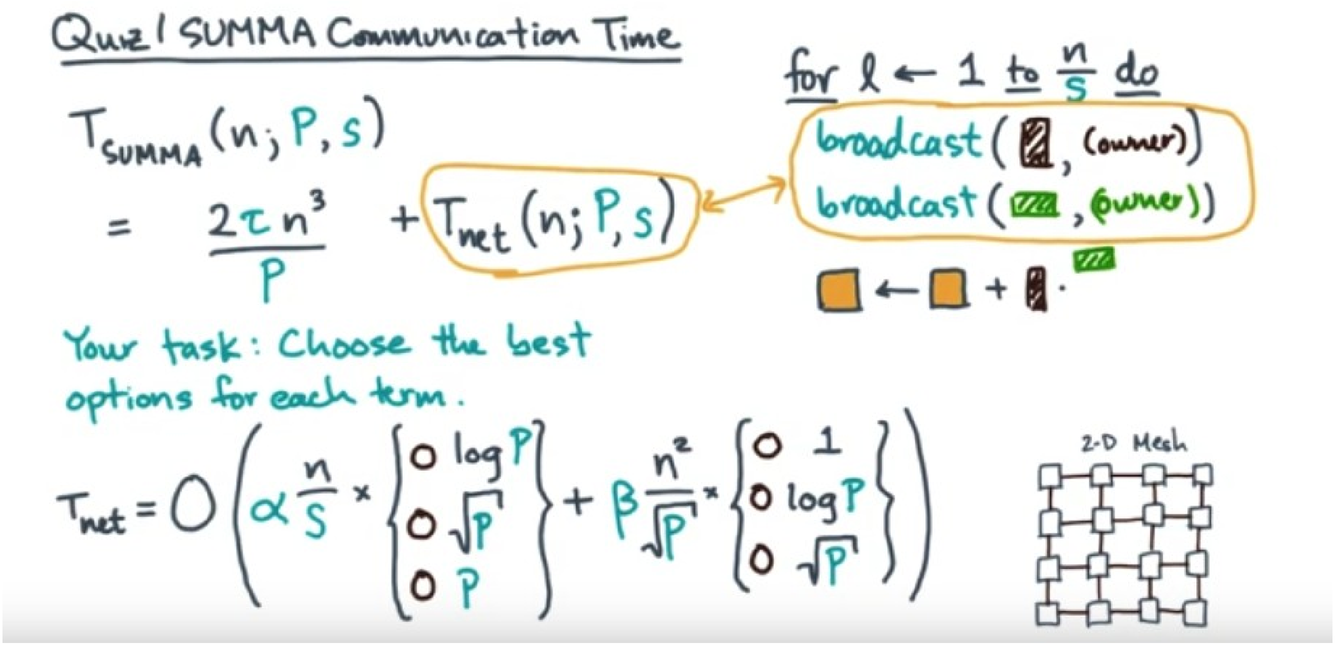
Now in order for all nodes in this block row (黃色空心橫的 block) or this block column (黃色空心豎的 block) to proceed, every node needs to see the current strip. That means the owners of the current strip need to send a copy to all relevant nodes. For example, this strip (← 和 → 之間的那個黑色實心小豎的) of A needs to go to all processes in its block row (黃色空心橫的 block, 注意實際上是 2-D mesh 中與其對應位置的電腦或 node). The owner of the strip can simply do a broadcast within the block row (黃色空心橫的 block, 注意實際上是 2-D mesh 中與其對應位置的電腦或 node).



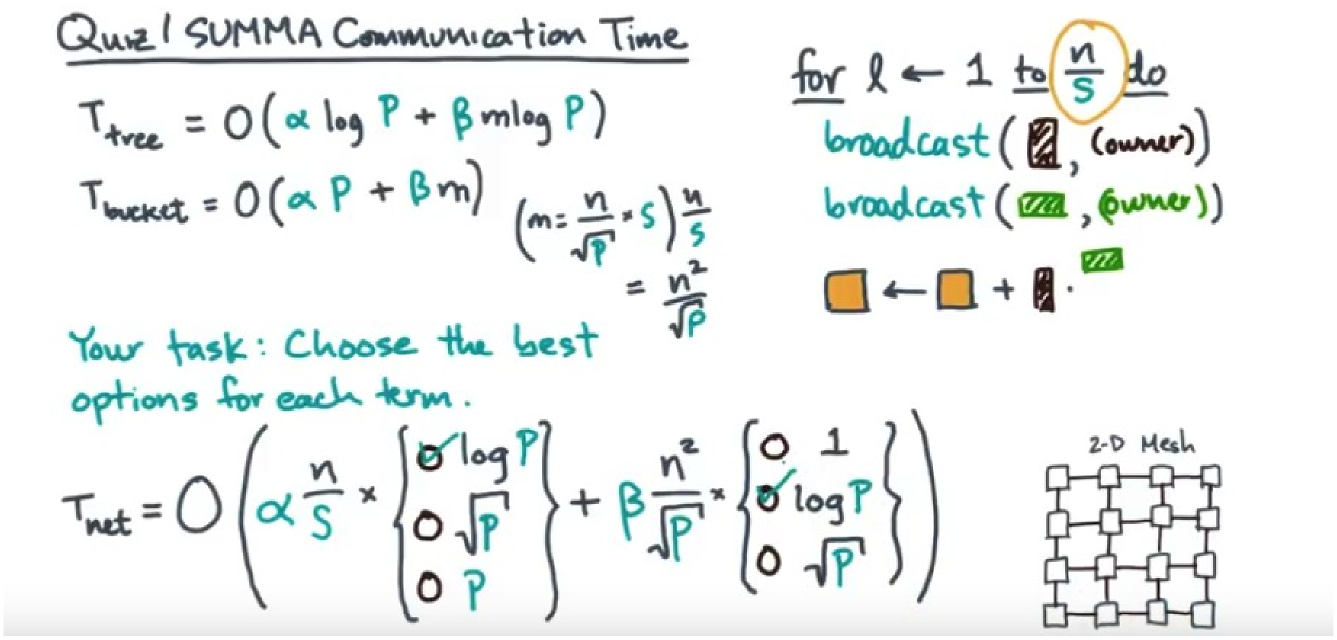
Now a similar thing should happen in the block column (黃色空心竪的 block, 注意實際上是 2-D mesh 中與其對應位置的電腦或 node). 為何要這樣? 因為這樣才能保證 C 中黃色實心 block 中既有黑色實心 block, 又有綠色實心 block. Once both strips have arrived, each node can do a local update.


A 2-D Algorithm: SUMMA
2-D Mesh
$T_{SUMMA}(n; P, s)$
$= \underbrace{\frac{n}{s} \cdot \left(2\tau \frac{n^2 s}{P}\right)}_{\frac{2\tau n^3}{P}} + T_{net}(n; P, s)$
?
for l ← 1 to n/s do
broadcast(, (owner))
broadcast(, owner))
(Assume n × n mats on √P × √P mesh, with √P, s | n.)

So what's the running time of this scheme? To keep things simple let's assume square matrices and a square process grid. And though it's not hard to generalize let's also assume that the square root of P which is the dimension of the mesh and little s the strip size, both divide the matrix dimension n. First observe that there are n/s iterations of the loop, and each iteration does the same thing. Now for the cost of each iteration. Let's start with the update step. Each of these strips is (n/sqrt(p)) * s (因為 s 是 stip 的寬, n/sqrt(p)是 strip 的長). We can use that fact to compute the number of flops for this local update step (2τ n^2 s / p 中: 2 是因為有 both multiply 和 add, 然後是右下角的黑的乘綠的, 共乘 n/sqrt(p)次, 而每次乘 是兩個長度為 s 的小向量點乘). This is the cost of the update step, assuming that each flop costs tau. Multiplying this out, the total flop time is 2 tau n cubed / p. What about the communication time? The communication time is essentially just the time for these two broadcasts. Mm, I wonder how much time that costs?


Quiz / SUMMA Communication Time
$T_{SUMMA}(n; P, s) = \frac{2\tau n^3}{P} + T_{net}(n; P, s)$
for $\ell \leftarrow 1$ to $\frac{n}{s}$ do
broadcast(▨ , (owner))
broadcast(▨ , owner))
■ ← ■ + ▨ · ▨
Your task: Choose the best options for each term.
$T_{net} = O\left(\alpha \frac{n}{s} \times \begin{Bmatrix} \circ \ \log P \\ \circ \ \sqrt{P} \\ \circ \ P \end{Bmatrix} + \beta \frac{n^2}{\sqrt{P}} \times \begin{Bmatrix} \circ \ 1 \\ \circ \ \log P \\ \circ \ \sqrt{P} \end{Bmatrix}\right)$
2-D Mesh

17. Let's do a little quiz where you calculate the communication time of the summa algorithm. You argued previously that the running time to do all the flops is 2n cubed over P. But what about communication? Here's my question. Which of the following estimates are plausible estimates of the overall communication running time? For the alpha term, I want you to choose one of these. For the beta term, I want you to choose the matching one of these. Now, I put a big O in front of this so that you can ignore constants and just focus on asymptotics. Now, there may be more than one correct pairing, so the key is, based on your choice for one of them choose something that's plausible for the other one.



Quiz: SUMMA Communication Time

$T_{tree} = O(\alpha \log P + \beta m \log P)$

$T_{bucket} = O(\alpha P + \beta m)$

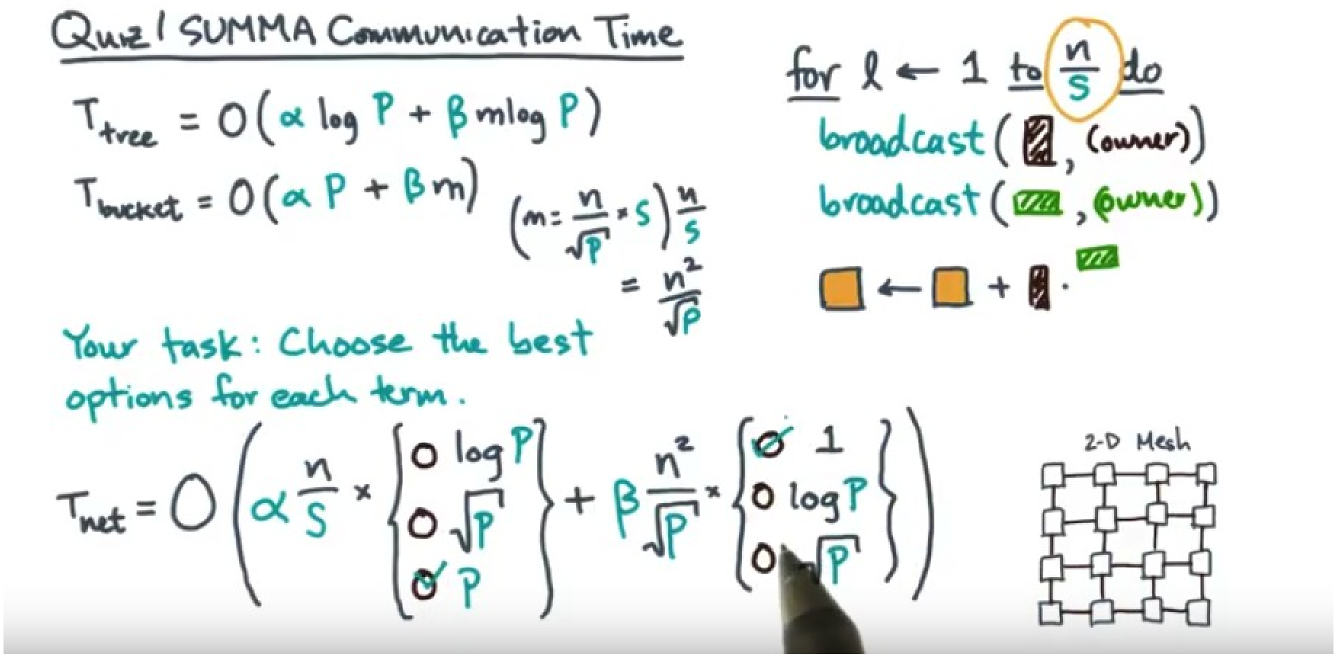
$\left(m = \frac{n}{\sqrt{P}} \cdot s\right) \frac{n}{s} = \frac{n^2}{\sqrt{P}}$

```
for l ← 1 to n/s do
    broadcast(▨, (owner))
    broadcast(▨, (owner))
    ■ ← ■ + ▨ · ▨
```

Your task: Choose the best options for each term.

$T_{net} = O\left(\alpha \frac{n}{s} \times \begin{Bmatrix} ✓ \log P \\ \sqrt{P} \\ P \end{Bmatrix} + \beta \frac{n^2}{\sqrt{P}} \times \begin{Bmatrix} 1 \\ ✓ \log P \\ \sqrt{P} \end{Bmatrix}\right)$

18. The communication all comes from the two broadcast operations. Their cost depends on what you assume about the cost of a broadcast. Two schemes you've derived on the tree based algorithm and the pipeline and the bucket algorithm. And you recall these were the communication times for those two implementations. The little m in these formulas denotes the size of the message. Now for this question, little m corresponds to the size of these little blocks. How large are they? Recall that the dimensions are n over root P by s. Now to get the total volume of communication, remember that we execute this pair of broadcasts, n over s times. Thus the total communication volume we have to pay for is n/s times the size of each message. That, it turns out is n squared over square root of P. So assuming a choice of these two implementations, here are two possible correct pairings. If you assume the tree based algorithm, then a correct pair would be log P and log P.



Quiz: SUMMA Communication Time

$T_{tree} = O(\alpha \log P + \beta m \log P)$

$T_{bucket} = O(\alpha P + \beta m)$

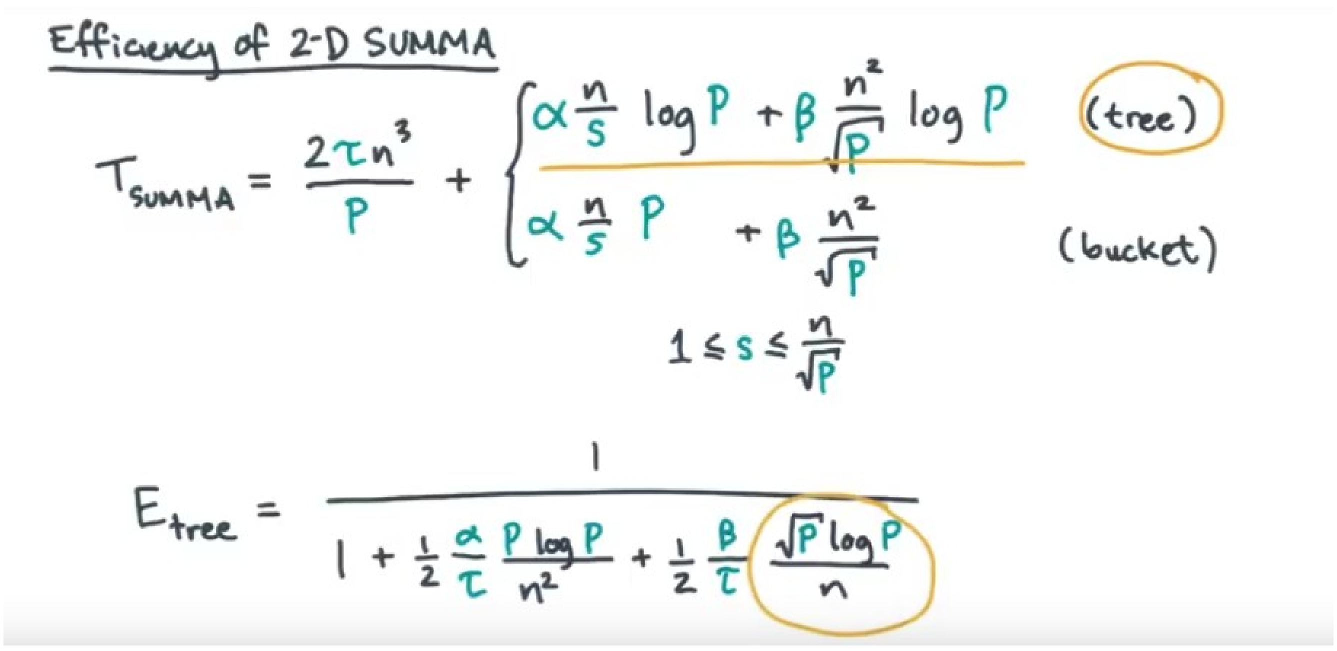
$\left(m = \frac{n}{\sqrt{P}} \cdot s\right) \frac{n}{s} = \frac{n^2}{\sqrt{P}}$

```
for l ← 1 to n/s do
    broadcast(▨, (owner))
    broadcast(▨, (owner))
    ■ ← ■ + ▨ · ▨
```

Your task: Choose the best options for each term.

$T_{net} = O\left(\alpha \frac{n}{s} \times \begin{Bmatrix} \log P \\ \sqrt{P} \\ ✓ P \end{Bmatrix} + \beta \frac{n^2}{\sqrt{P}} \times \begin{Bmatrix} ✓ 1 \\ \log P \\ \sqrt{P} \end{Bmatrix}\right)$

If, on the other hand, you assume the bucketing based algorithm, then a correct pairing would be P and 1.



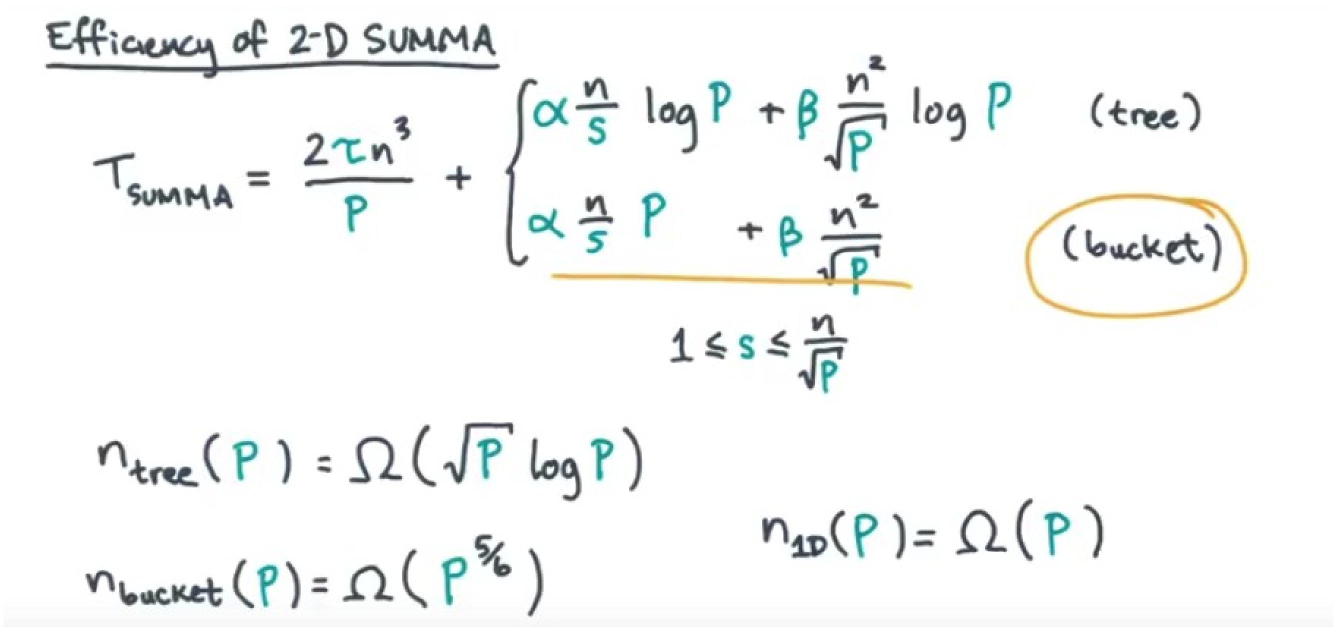
Efficiency of 2-D SUMMA

$$T_{\text{SUMMA}} = \frac{2\tau n^3}{P} + \begin{cases} \underline{\alpha \frac{n}{s} \log P + \beta \frac{n^2}{\sqrt{P}} \log P} & \boxed{\text{(tree)}} \\ \alpha \frac{n}{s} P + \beta \frac{n^2}{\sqrt{P}} & \text{(bucket)} \end{cases}$$

$$1 \leq s \leq \frac{n}{\sqrt{P}}$$

$$E_{\text{tree}} = \frac{1}{1 + \frac{1}{2} \frac{\alpha}{\tau} \frac{P \log P}{n^2} + \frac{1}{2} \frac{\beta}{\tau} \boxed{\frac{\sqrt{P} \log P}{n}}}$$

19. So is the 2-D summa scheme intrinsically more scaleable than the 1-D block row scheme? The answer is quite possibly. First note that the strip width is a tuning perimeter of the algorithm. Its range is between 1 and N over square root of P. Again, that's assuming square operands and a square process grid. To analyze the efficiency, let's start by assuming the tree based broadcast implementation. If you compute the efficiency, you should find that it's equal to this. Now if you stare at this long enough, one of two things will happen. One is you'll glaze over. The other is being observant, you'll see that this factor (黃圈中的) determines the isoefficiency.



Efficiency of 2-D SUMMA

$$T_{\text{SUMMA}} = \frac{2\tau n^3}{P} + \begin{cases} \alpha \frac{n}{s} \log P + \beta \frac{n^2}{\sqrt{P}} \log P & \text{(tree)} \\ \underline{\alpha \frac{n}{s} P + \beta \frac{n^2}{\sqrt{P}}} & \boxed{\text{(bucket)}} \end{cases}$$

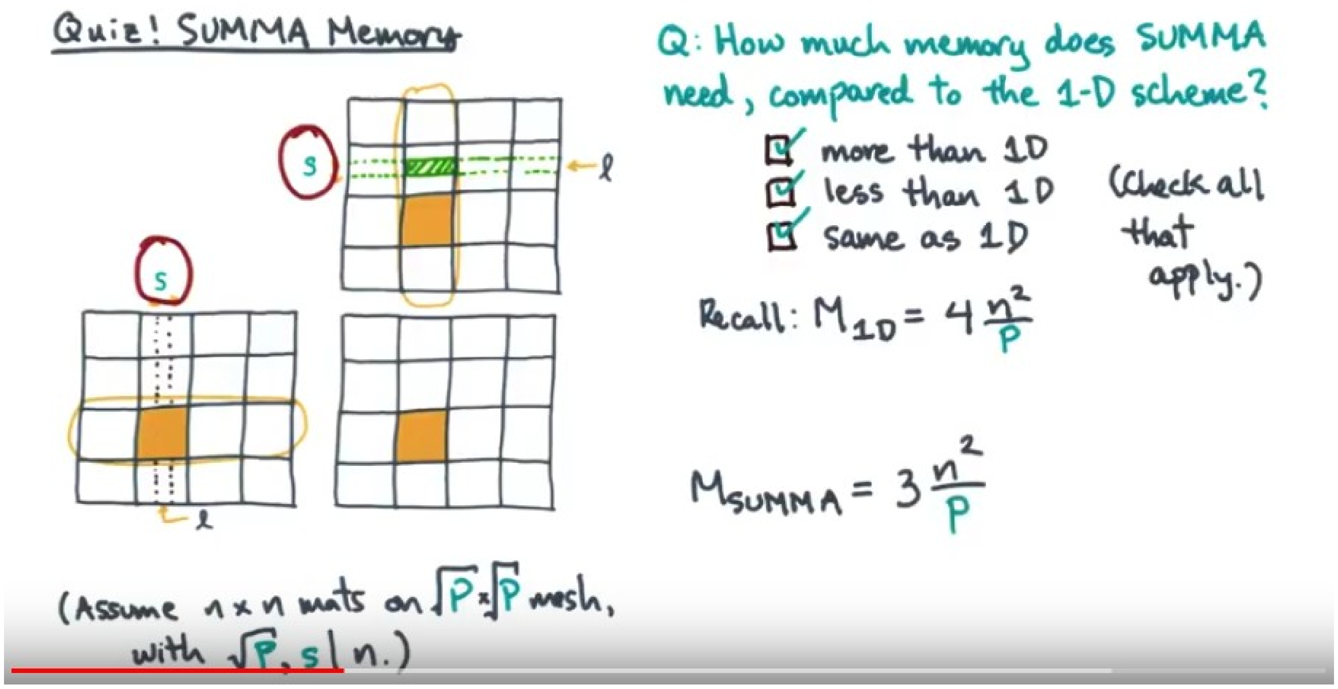
$$1 \leq s \leq \frac{n}{\sqrt{P}}$$

$$n_{\text{tree}}(P) = \Omega(\sqrt{P} \log P)$$

$$n_{\text{bucket}}(P) = \Omega(P^{5/6})$$

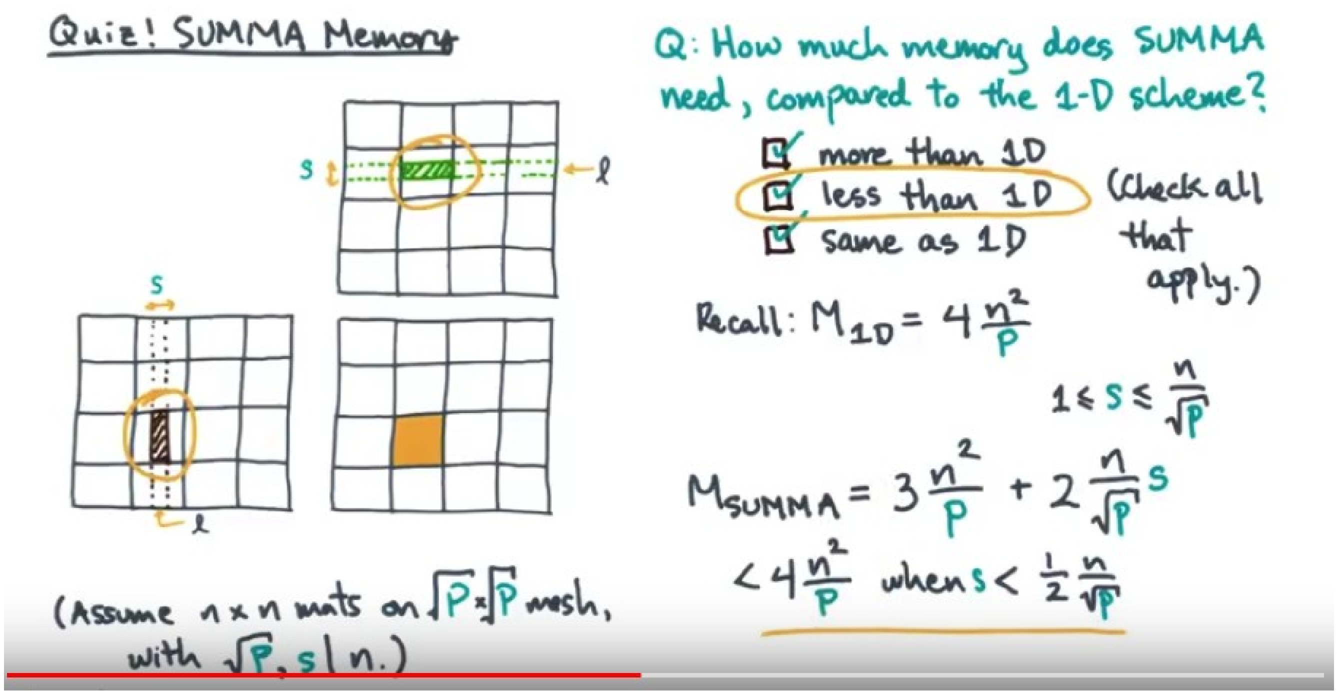
$$n_{1D}(P) = \Omega(P)$$

In particular the isoefficiency function is as follows. Let's compare this to the 1-D algorithm. The isoefficiency function of the 2-D scheme is asymptotically lower than the 1-D scheme. That means the 2-D scheme is intrinsically more scalable. You don't need to increase the problem size as fast as you need to with the 1-D scheme. Now you can of course do the same exercise with a bucketing scheme. In that case you'll find a slightly different ISO efficiency function. Asymptotically, this is a little bit worse than the tree based scheme. If you work it out you'll find that it comes from trading a lower communication volume with a higher latency cost.


Quiz! SUMMA Memory
s
l
s
l
Q: How much memory does SUMMA need, compared to the 1-D scheme?
☑ more than 1D
☑ less than 1D
☑ same as 1D
(Check all that apply.)
Recall: $M_{1D} = 4\frac{n^2}{P}$
$M_{SUMMA} = 3\frac{n^2}{P}$
(Assume $n \times n$ mats on $\sqrt{P} \times \sqrt{P}$ mesh, with $\sqrt{P}, s \mid n$.)

20. Here's a quick follow-up question about the SUMMA algorithm. Compared to the 1-D scheme, does SUMMA require more memory, the same amount of memory, or less memory? By way of reminder, remember the 1-D scheme's memory requirement was basically 4 n squared over P. You'll remember that came from having to store a piece of A, a piece of B, and a piece of C as well as an extra buffer for messaging (故 4). Now, if you think the answer depends on something, you can check more than one of the options that you think are possible.

21. As you might have anticipated from my hint, the answer is of course that it depends. So it could be more, it could be less or it could be the same. It all boils down to how you choice the value of the tuning parameter, little s. To see how, let's write down an expression for the storage. First, each node has to store one block of A, B and C. So that of course gives us three n squared over P.



Quiz! SUMMA Memory

Q: How much memory does SUMMA need, compared to the 1-D scheme?

- ☑ more than 1D
- [✓ less than 1D]
- ☑ same as 1D

(Check all that apply.)

Recall: $M_{1D} = 4\frac{n^2}{P}$

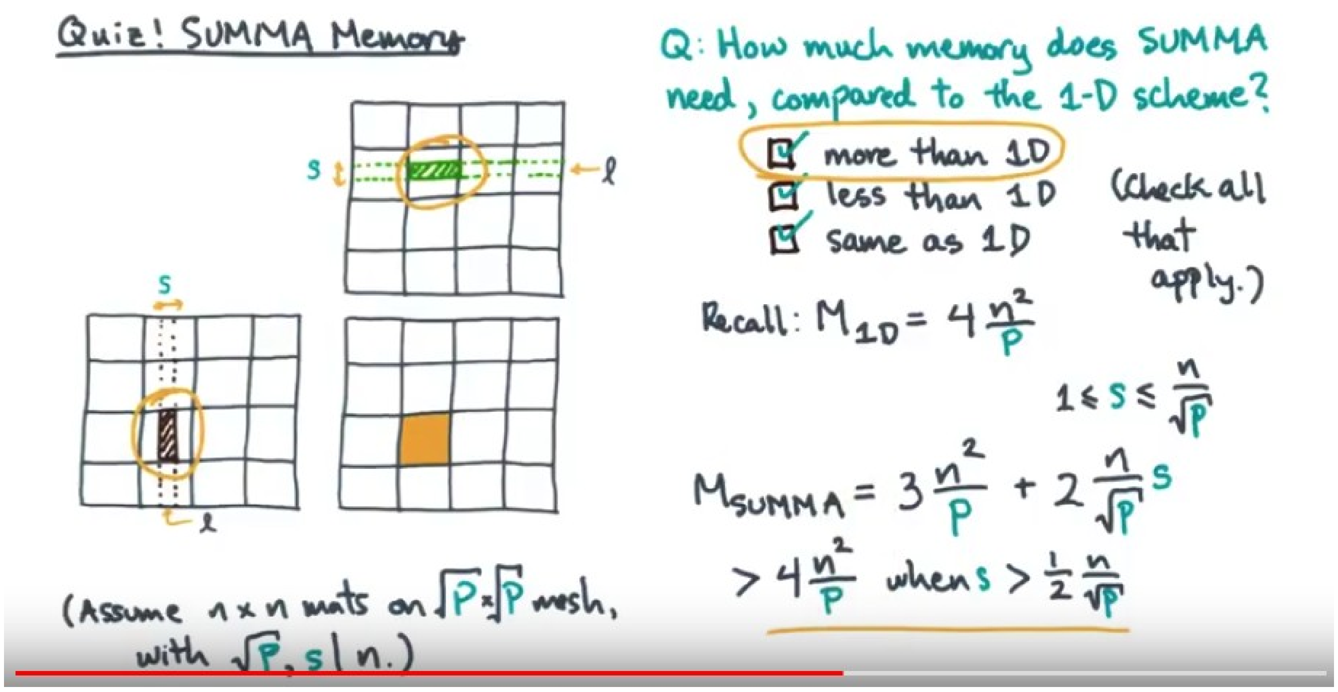
$1 \le s \le \frac{n}{\sqrt{P}}$

$$M_{SUMMA} = 3\frac{n^2}{P} + 2\frac{n}{\sqrt{P}}s$$

$$< 4\frac{n^2}{P} \text{ when } s < \frac{1}{2}\frac{n}{\sqrt{P}}$$

(Assume $n \times n$ mats on $\sqrt{P} \times \sqrt{P}$ mesh, with $\sqrt{P}, s \mid n$.)

Now you also need buffers for the two broadcasted strips. The amount of storage depends on the strip width, little s. So here are the cases. S has to go between one and N over root P, as you'll remember. When S is less than one half N over root P, then the total storage is less than four N squared over P, meaning less than the 1D algorithm. Remember that the problem with a smaller s is it increases the latency time.



Quiz! SUMMA Memory

Q: How much memory does SUMMA need, compared to the 1-D scheme?

- [✓ more than 1D]
- ☑ less than 1D
- ☑ same as 1D

(Check all that apply.)

Recall: $M_{1D} = 4\frac{n^2}{P}$

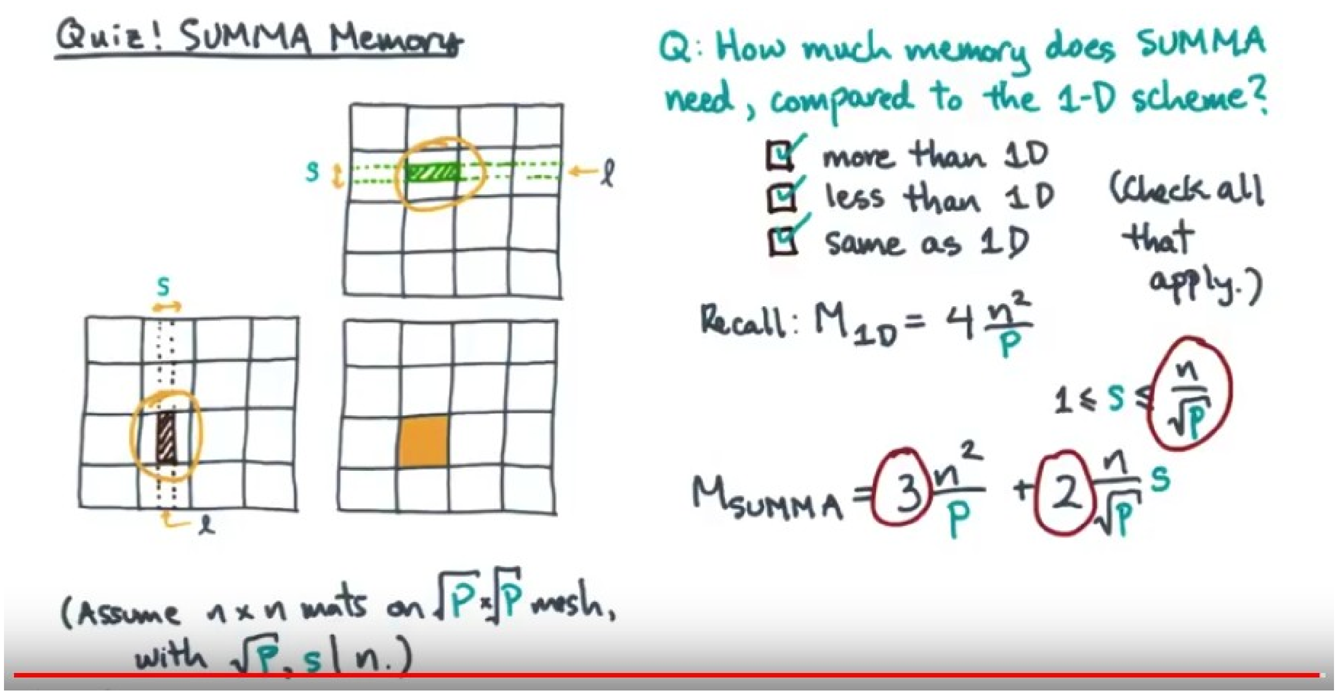
$1 \le s \le \frac{n}{\sqrt{P}}$

$$M_{SUMMA} = 3\frac{n^2}{P} + 2\frac{n}{\sqrt{P}}s$$

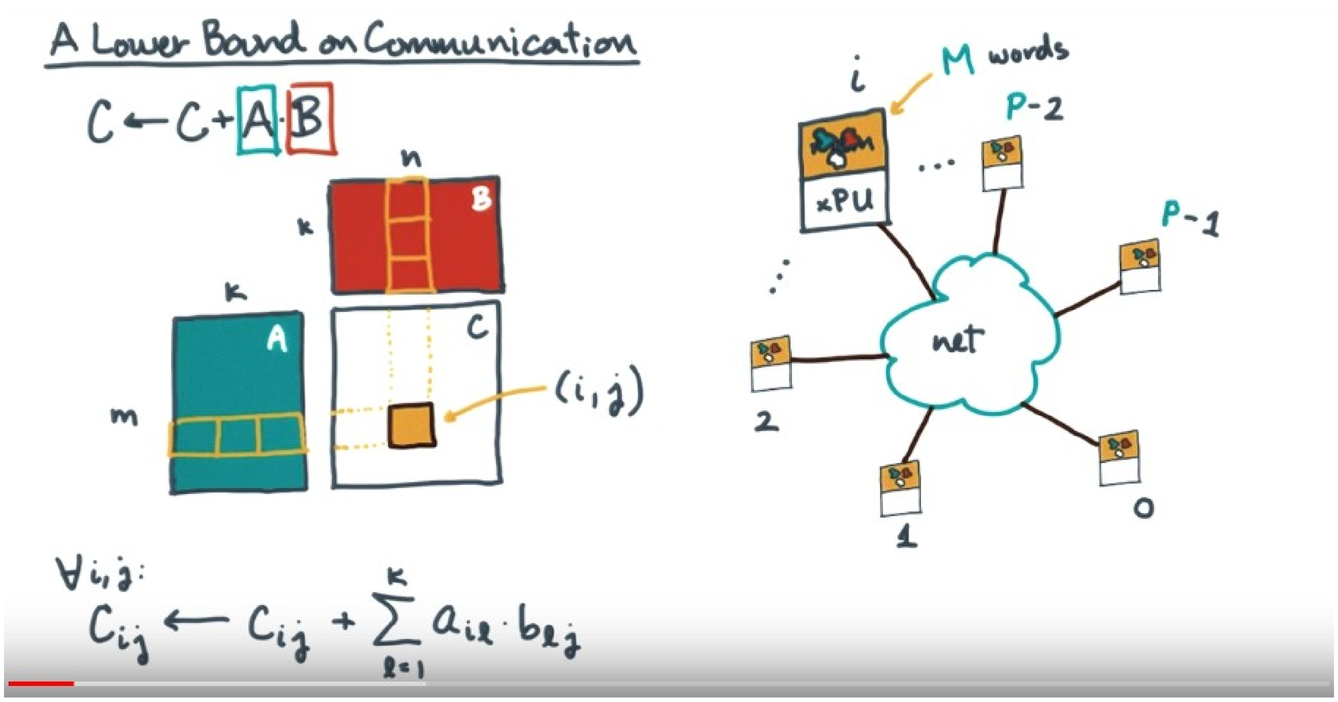
$$> 4\frac{n^2}{P} \text{ when } s > \frac{1}{2}\frac{n}{\sqrt{P}}$$

(Assume $n \times n$ mats on $\sqrt{P} \times \sqrt{P}$ mesh, with $\sqrt{P}, s \mid n$.)

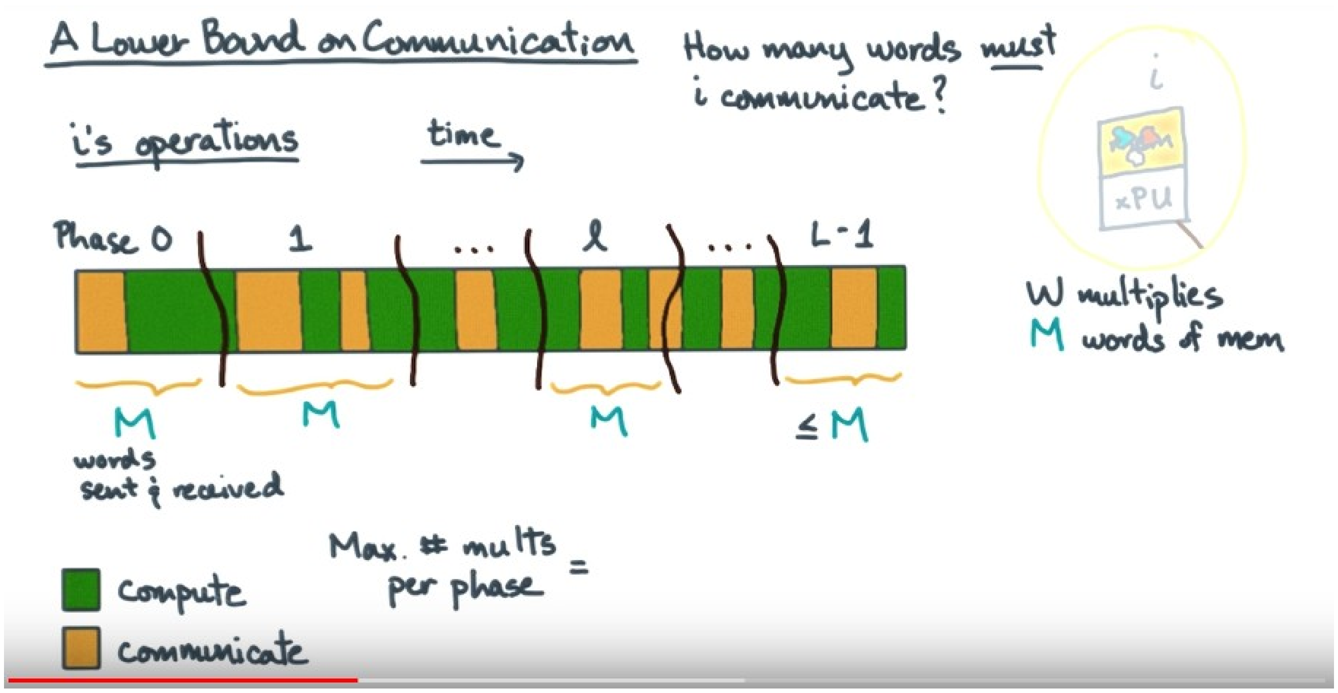
Now if instead s is greater than 1/2 n over root P then of course the storage will be greater than 4 n squared over P or more than the 1D algorithm.


Quiz! SUMMA Memory
Q: How much memory does SUMMA need, compared to the 1-D scheme?
☑ more than 1D
☑ less than 1D
☑ same as 1D
(Check all that apply.)
Recall: $M_{1D} = 4\frac{n^2}{P}$
$1 \le s \le \frac{n}{\sqrt{P}}$
$M_{SUMMA} = 3\frac{n^2}{P} + 2\frac{n}{\sqrt{P}}s$
s
l
(Assume n × n mats on $\sqrt{P} \times \sqrt{P}$ mesh, with $\sqrt{P}, s \mid n$.)

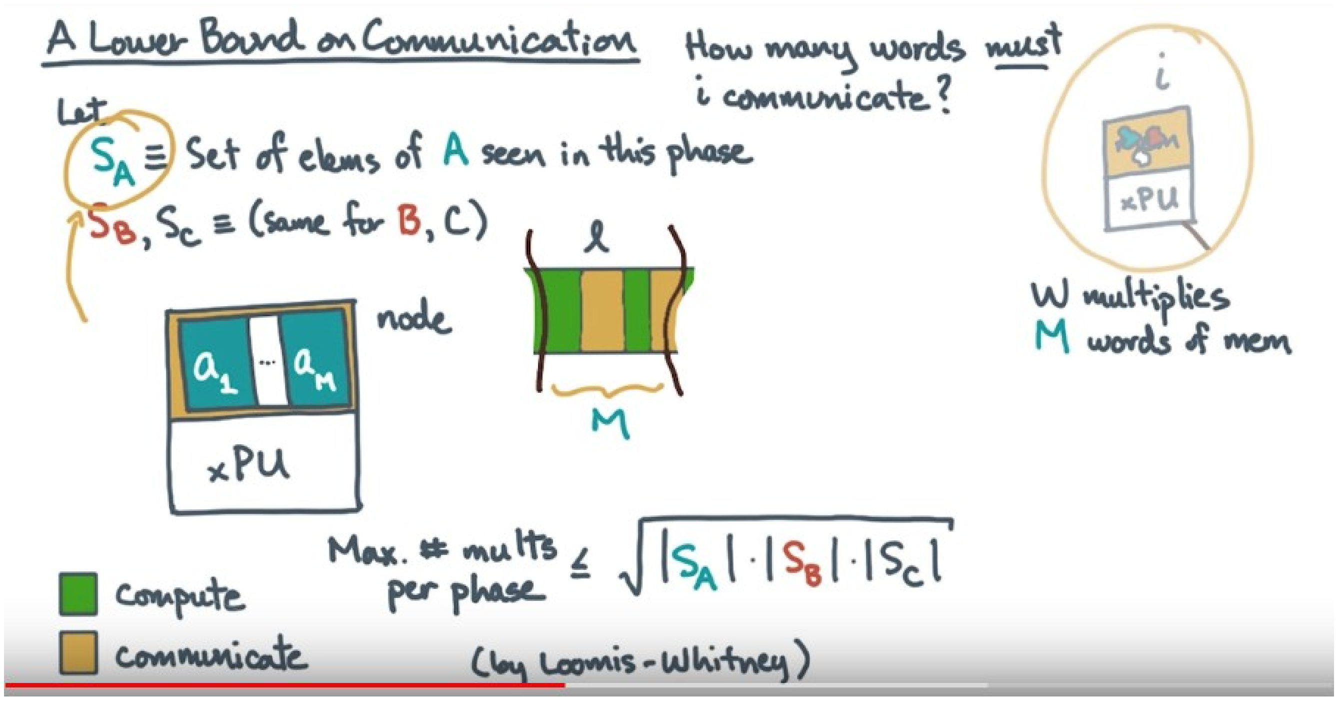
In fact what happens when s is its maximum value. In that case the SUMMA algorithm might need five times n squared over P worth of storage compared to four times for the 1D algorithm. The interesting bit of course, is that the strip width serves as a tuning parameter which adjusts the relevant importance of the latency term, the bandwidth term, the overall scaling and the memory requirement. It's the overall simplicity of a summa algorithm, combined with this tuning parameter that lets you trade off time and storage. That made summa the gold standard for dense linear algebra at least for the better part of the last 20 years or so.


A Lower Bound on Communication
$C \leftarrow C + A \cdot B$
n
k
B
k
A
C
m
(i, j)
$\forall i, j$:
$C_{ij} \leftarrow C_{ij} + \sum_{l=1}^{k} a_{il} \cdot b_{lj}$
i
M words
xPU
P-2
P-1
net
2
1
0

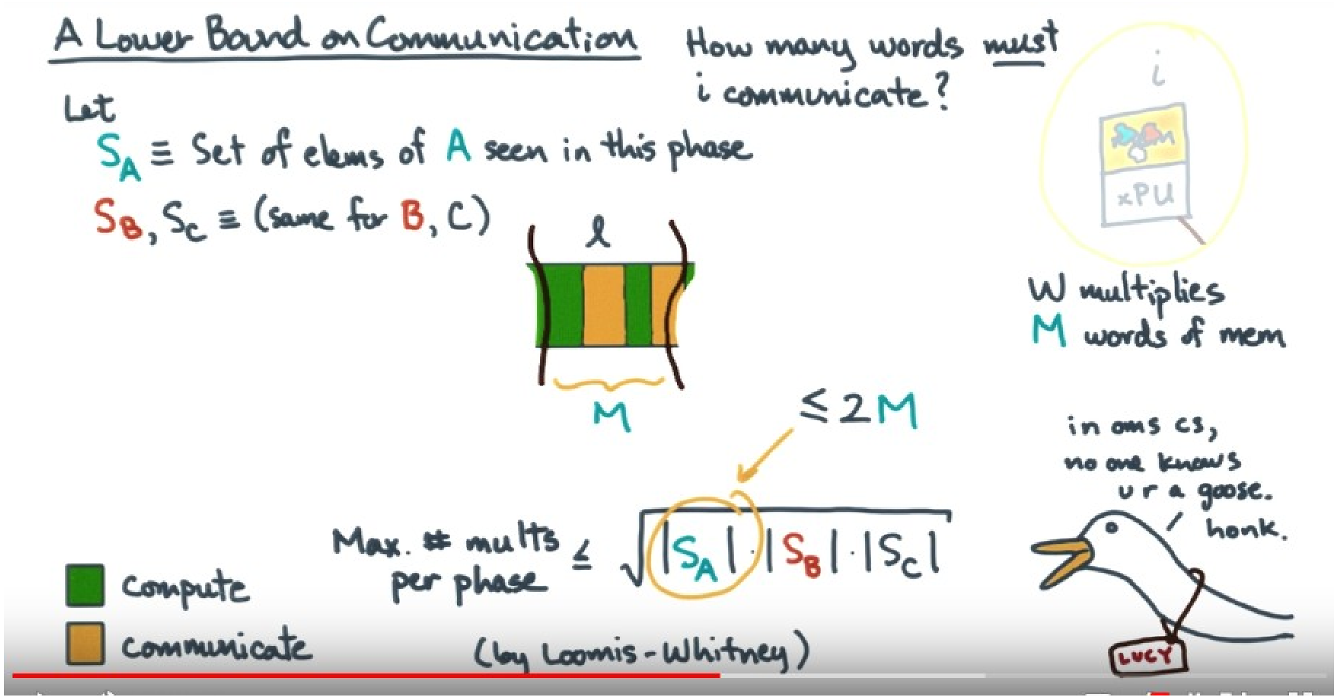
22. So the Summa algorithm seems pretty good, but how close is it to the best possible scheme? Imagine a machine with P nodes connected by some topology, further suppose each node has M words of main memory. Now let's say you take the matrix operands and you chop them up into little bits and distribute the little bits across all the nodes. What I want you to do now, is pick one of the nodes, zoom in on it and study its possible behavior.



Now recall that each node has M words of main memory. Let's further suppose that during the entire distributed matrix multiply, this particular node does W multiplications. We don't know what W is, but you'll determine that later. Now let me pose the following question to you. How many words must this node send or receive? To figure this out, let's start by imagining that the word performs its work as a sequence of computation and communication steps. We'll visualize what the node does by a hypothetical timeline, like this one. So time precedes horizontally. And the node alternates between phases of, say, communication followed by some computation, followed by some more communication, followed by computation, and so on. Now let's divide this timeline into L distinct phases. A phase is special. We’ll assume, in each phase, that every node sends and receives exactly M words. The exception is the last phase. In last phase, you might send or receive fewer than M words. The reason is simply that the total number of words sent and received might not be divisible by L. Here's a question. What is the largest number of multiplies that each phase can possibly do?


A Lower Bound on Communication
How many words must i communicate?
Let
$S_A \equiv$ Set of elems of A seen in this phase
$S_B, S_C \equiv$ (same for B, C)
node
a_1 … a_M
xPU
ℓ
M
i
xPU
W multiplies
M words of mem
Max. # mults per phase $\le \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$
(by Loomis-Whitney)
🟩 compute
🟧 communicate

To figure this out, let's consider one of the phases. Let S sub A be the number of unique elements of A seen during this phase. And similarly, let's let S sub B, and S sub C, be the number of unique elements of B and C that we see during this phase respectively. By Loomis-Whitney, you can compute the maximum number of multiplies performed in this phase given S sub A, S sub B, and S sub C. So all we need to figure out now is, how big are these sets? Let's start by considering just S of A. Now imagine the node and the state of it's memory. It's possible that at the very beginning of the phase, the memory is full of only elements from A. And of course, because the memory capacity is M, there can't be more than M such words. Now remember, that by definition the phase performs exactly M sends or receives. So let's think about what that might mean. It's possible that during the phase, all of these words are used to do computation. It's also possible that all M of these words, at some point, get pushed out. And when I said pushed out, I don't necessarily mean being communicated, I just mean we get rid of them and make space for other words.



A Lower Bound on Communication

How many words must i communicate?

Let

$S_A \equiv$ Set of elems of A seen in this phase

$S_B, S_C \equiv$ (same for B, C)

Max. # mults per phase $\leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$

(by Loomis-Whitney)

W multiplies

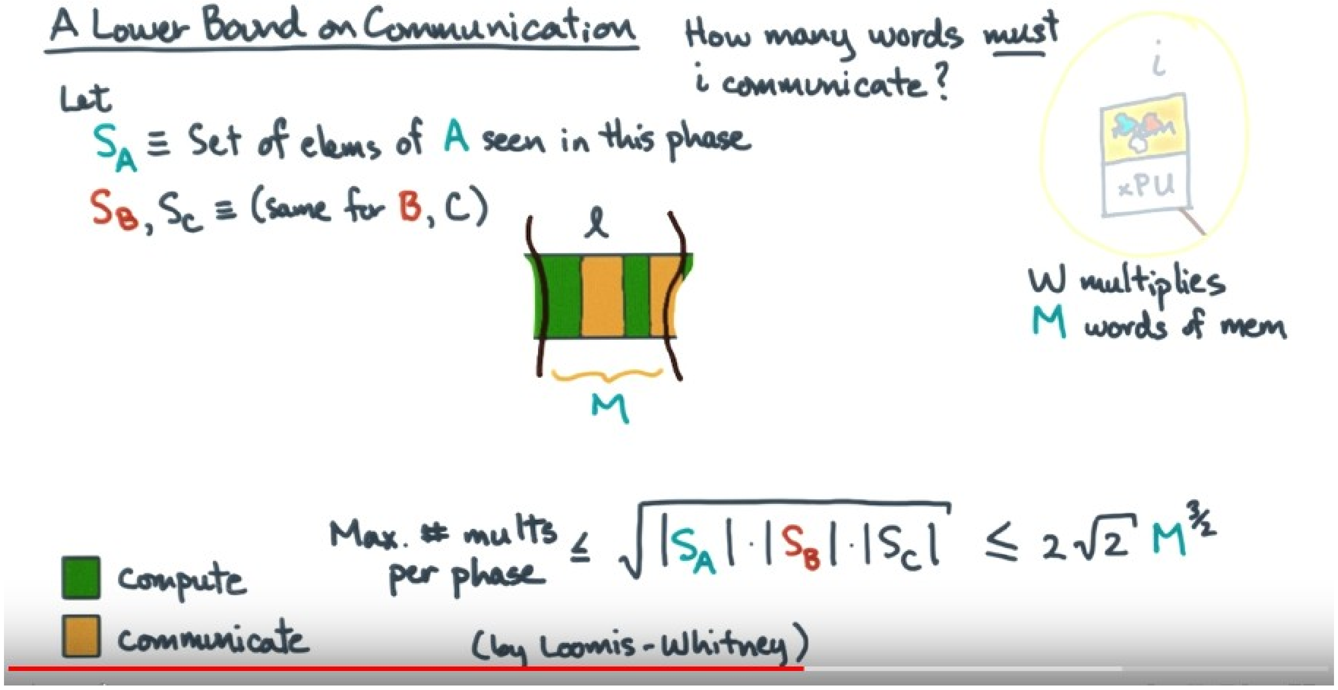
M words of mem

in oms cs, no one knows u r a goose. honk.

🟩 compute

🟧 communicate

It's then possible that all these communications bring in a whole new set of elements of A. These are M completely brand new set of elements and it's possible that before the end of the phase, we use all of them to form more multiplications. So what have we said so far about the capacity of S sub A? Basically, we just said that it's possible, during this phase, that we might have seen up to 2 times M elements from A. Now remember, this is just an upper bound. And I realize it might seem a little loosey goosey, as far as bounds go. Honk. Now even if this bound a bit loose, remember, we're just after a bound, so there's nothing incorrect about this reasoning. It just might not be as precise as it could be.



A Lower Bound on Communication

How many words must i communicate?

Let

$S_A \equiv$ Set of elems of A seen in this phase

$S_B, S_C \equiv$ (same for B, C)

Max. # mults per phase $\leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|} \leq 2\sqrt{2}\, M^{3/2}$

(by Loomis-Whitney)

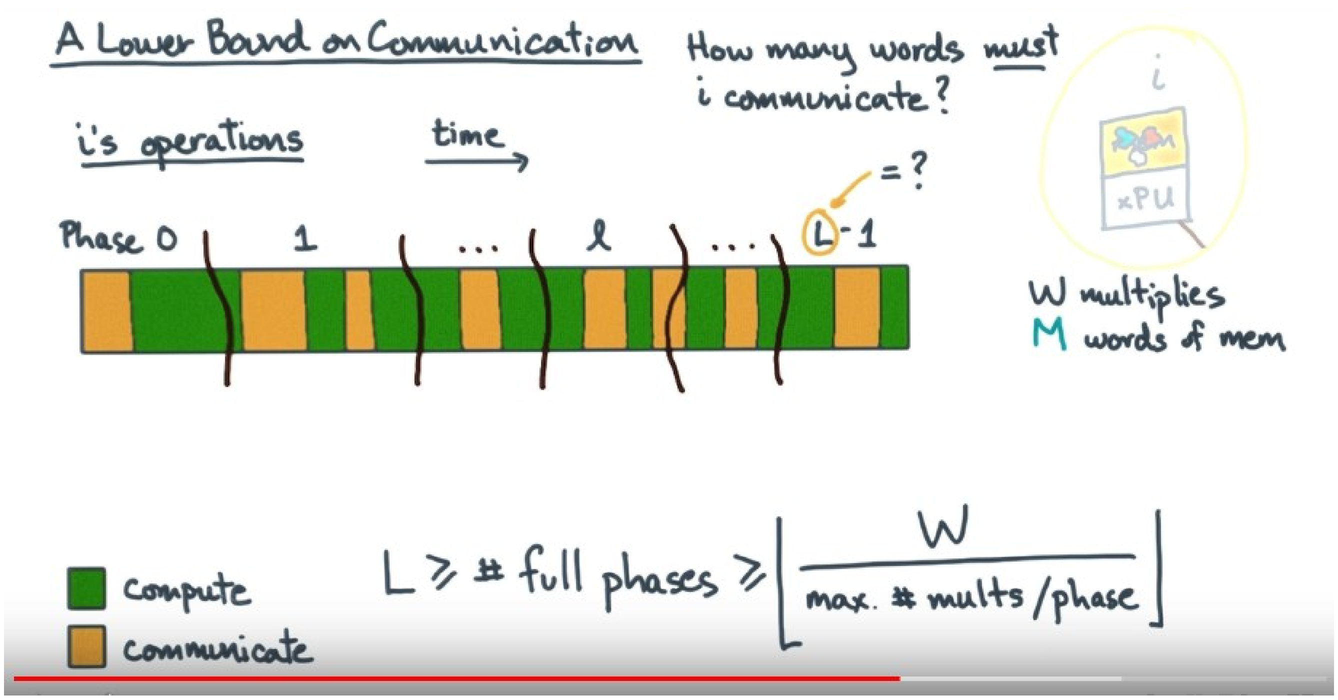
W multiplies

M words of mem

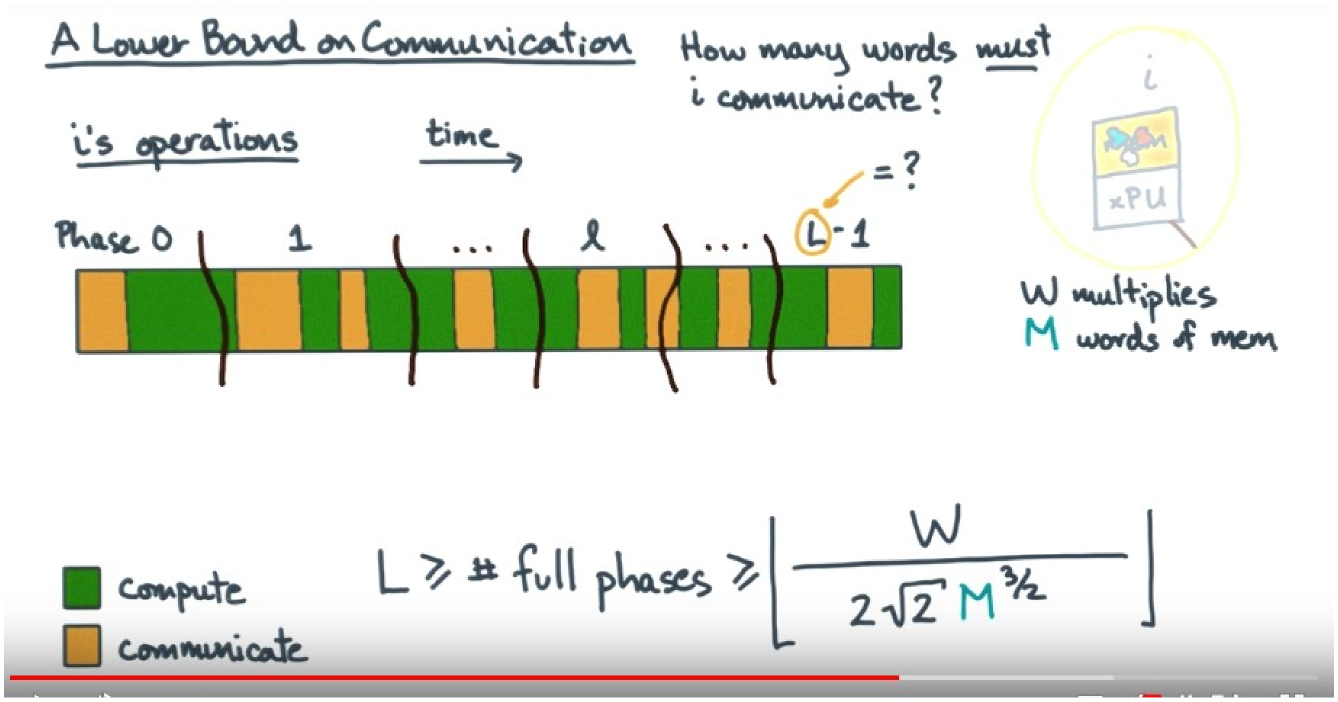
🟩 compute

🟧 communicate

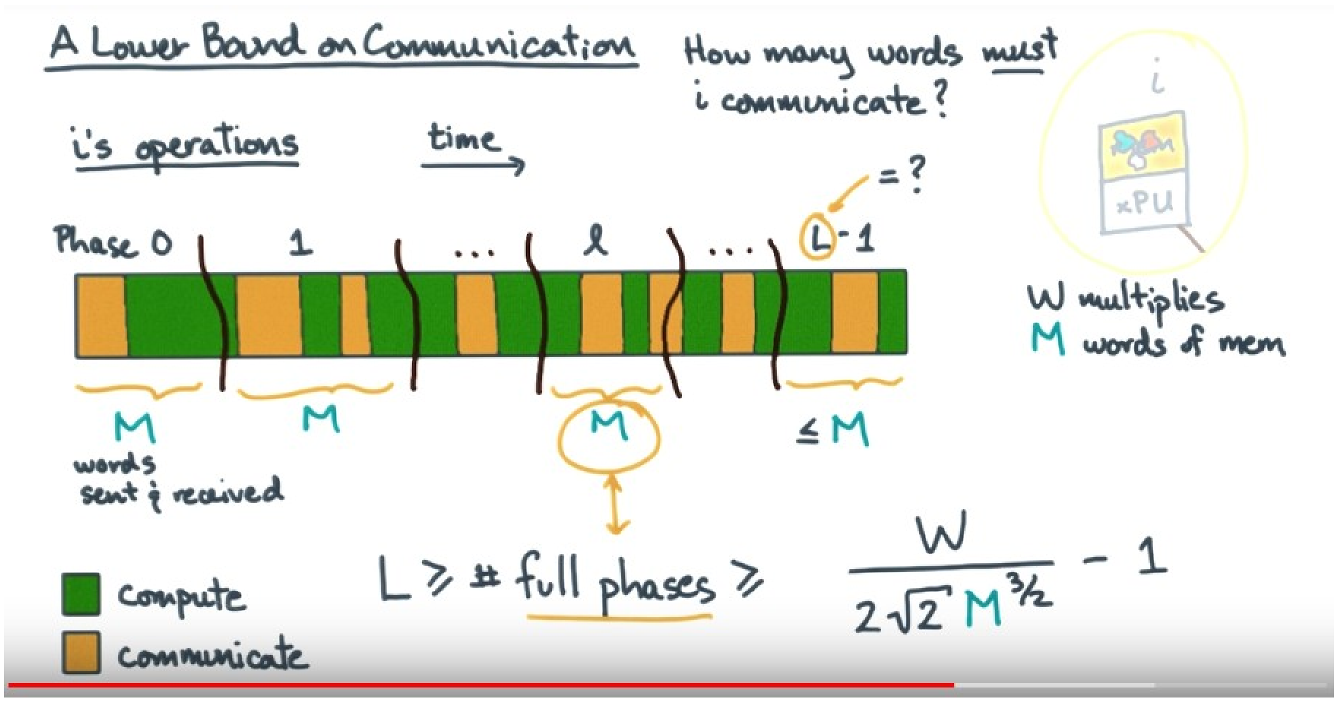
Now by similar reasoning, you can find that the maximum size of S B and S C are also 2 times M. When you plug all that in you'll get 2 root 2, times M to the three-halves. You might want to stop and check that you agree. What we've just said is that the maximum number of multiplies that might have occurred during this phase is 2 times root 2, times M to the three-halves.


A Lower Bound on Communication
How many words must i communicate?
i's operations
time
Phase 0
1
...
l
...
L-1
=?
i
xPU
W multiplies
M words of mem
compute
communicate
$L \geq \#\text{ full phases} \geq \left\lfloor \frac{W}{\text{max. \# mults / phase}} \right\rfloor$

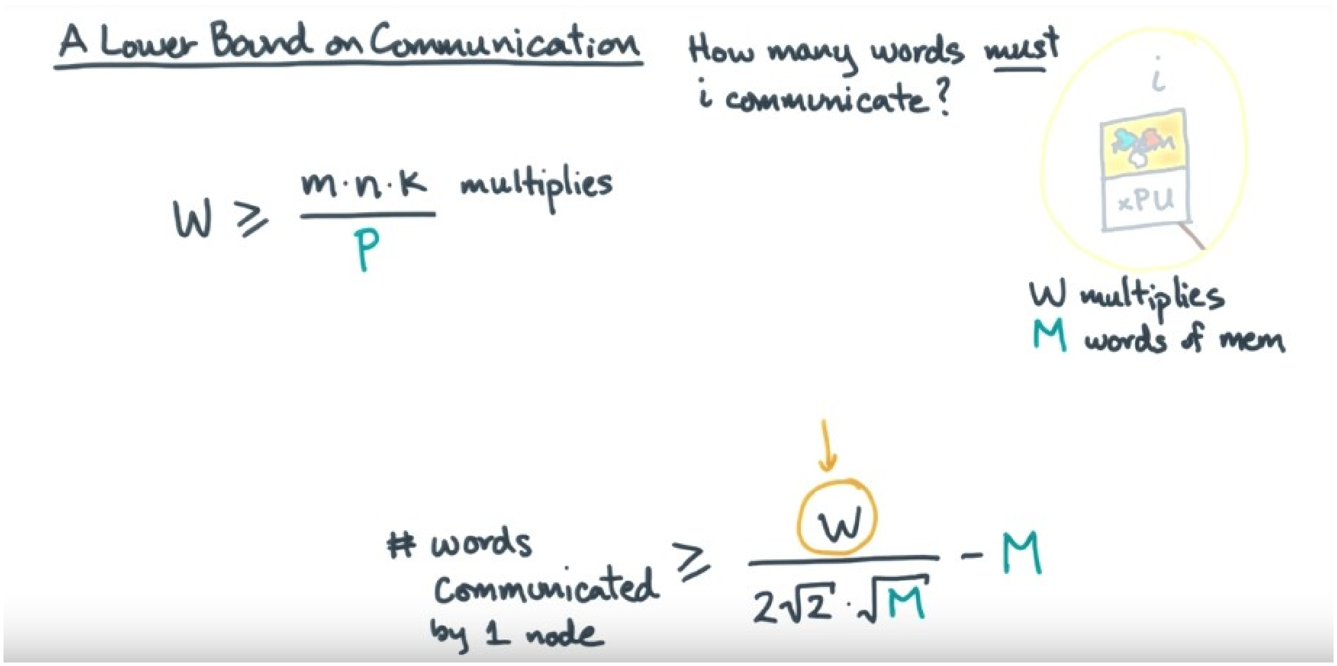
A natural follow-up question is, how many phases are there? Let's go back to the timeline. I told you this node did a total of W multiplies. Remember, that every phase does at most this number of multiplies. So what might L be? Well, L has to be at least equal to the number of full phases. And the number of full phases has to be at least the number of multiplies that the node does, divided by the maximum number of multiplies per phase.


A Lower Bound on Communication
How many words must i communicate?
i's operations
time
=?
Phase 0
1
...
l
...
L-1
i
xPU
W multiplies
M words of mem
compute
communicate
$L \geqslant$ # full phases $\geqslant \left\lfloor \frac{W}{2\sqrt{2}\,M^{3/2}} \right\rfloor$

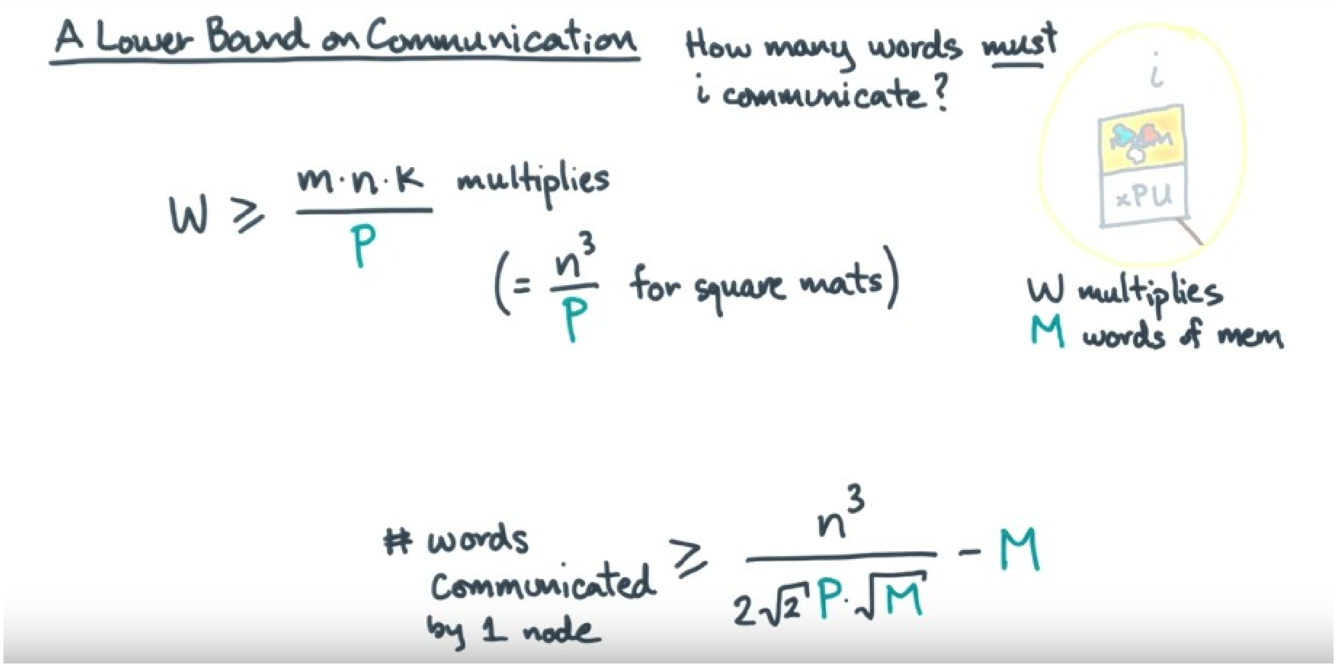
And of course, you just computed that, so we can plug that in.


A Lower Bound on Communication
How many words must i communicate?
i's operations
time
=?
Phase 0
1
...
l
...
L-1
i
xPU
W multiplies
M words of mem
M
words sent & received
M
M
≤M
compute
communicate
$L \geqslant$ # full phases $\geqslant \frac{W}{2\sqrt{2}\,M^{3/2}} - 1$

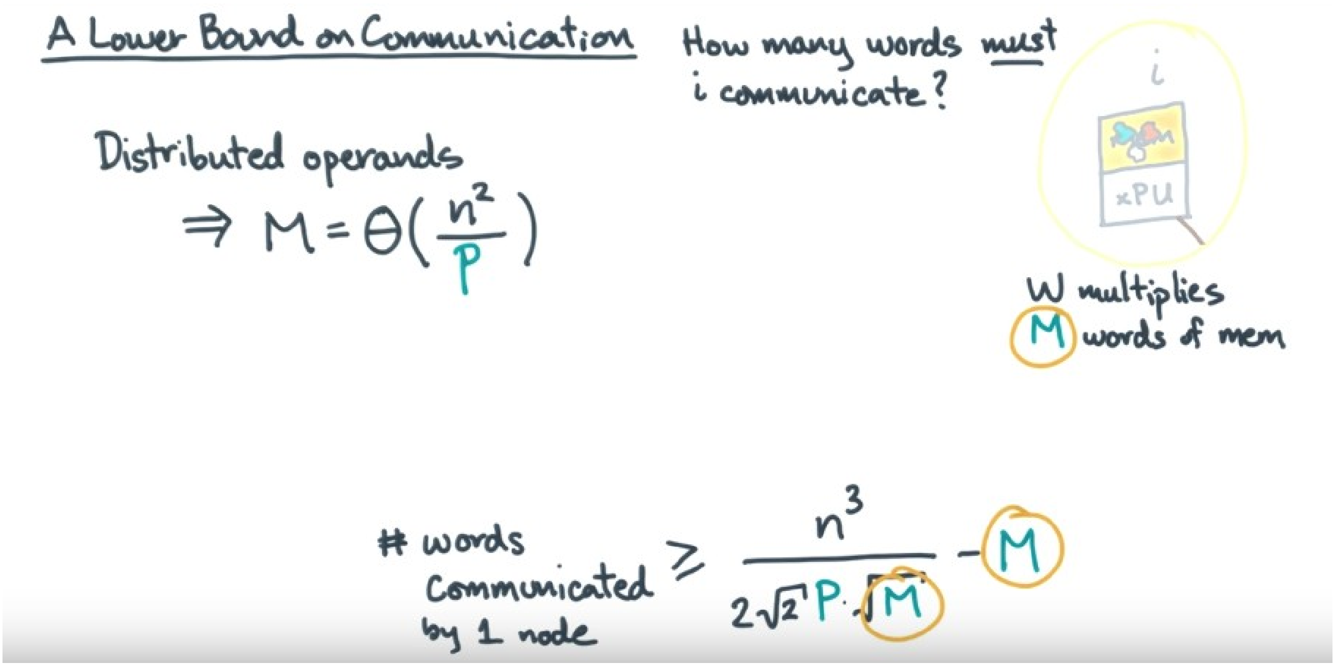
And then, of course, get rid of the floors, leaving you with this.


A Lower Bound on Communication
How many words must i communicate?
$W \geq \frac{m \cdot n \cdot K}{P}$ multiplies
i
xPU
W multiplies
M words of mem
# words communicated by 1 node $\geq \frac{W}{2\sqrt{2} \cdot \sqrt{M}} - M$

Now remember, that every full phase does exactly M transfers. That means you can get a lower bound on the total number of transfers, which would just be the number of full phases times M. So the number of words communicated by at least one node, has to be at least the number of full phases times M. Let's plug all this stuff in. Okay, whew, you're almost there. Now what about W? If the matrix operands had been of size mby k, k by n, and m by n, then you know the total number of multiplies is just mnk. Now, if that's a total number of multiplies, and all the processes are executing in parallel, then at least one node has to execute mnk over P multiplications. That is, there's some node for which W is at least mnk over P.


A Lower Bound on Communication
How many words must i communicate?
$W \geq \frac{m \cdot n \cdot K}{P}$ multiplies
$(= \frac{n^3}{P}$ for square mats$)$
i
xPU
W multiplies
M words of mem
# words communicated by 1 node $\geq \frac{n^3}{2\sqrt{2} P \cdot \sqrt{M}} - M$

Let's plug that in. What you now have is a lower bound on the volume of communication by at least 1 node. Let's simplify this a little bit for the square case. That would be n cubed over P multiplies, per node.



A Lower Bound on Communication

How many words *must* i communicate?

i

xPU

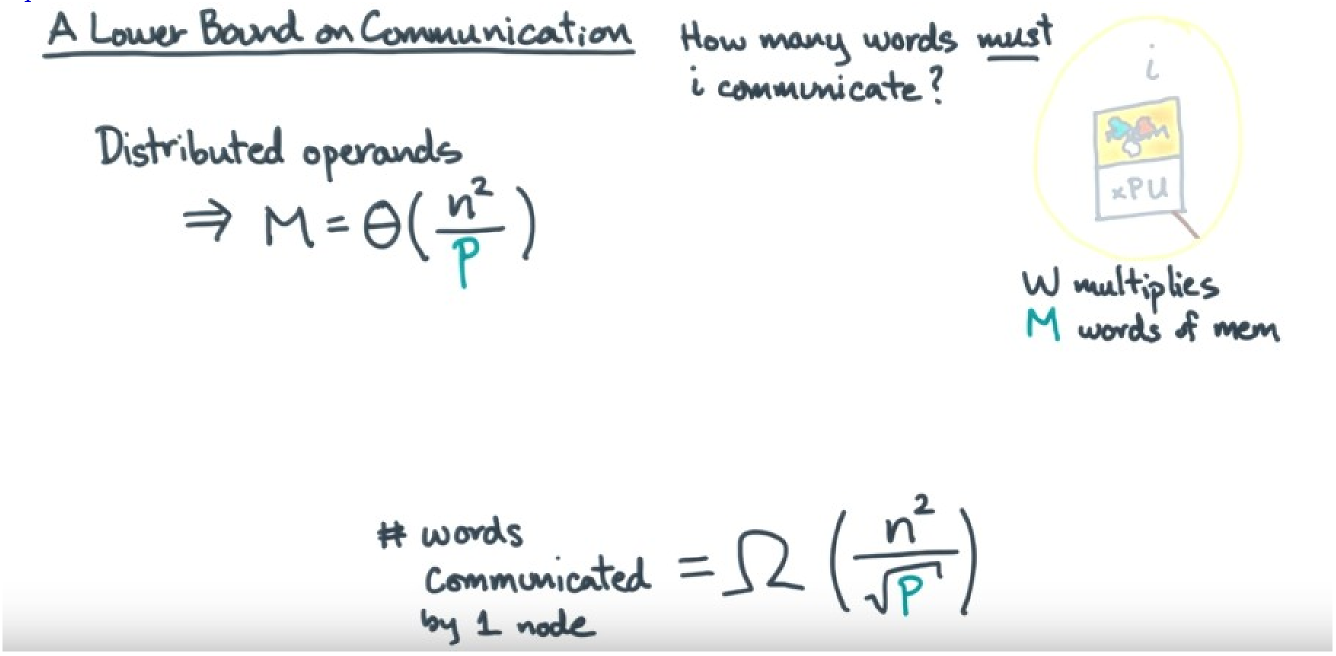
W multiplies M words of mem

Distributed operands

$$\Rightarrow M = \Theta\left(\frac{n^2}{P}\right)$$

$$\#\text{ words communicated by 1 node} \geq \frac{n^3}{2\sqrt{2}\,P \cdot \sqrt{M}} - M$$

Now there's just one more piece of useful information which is the value of M. Remember, that we distributed the matrices, evenly, over all the nodes. So that means M has to be proportional to n squared over P.



A Lower Bound on Communication

How many words *must* i communicate?

i

xPU

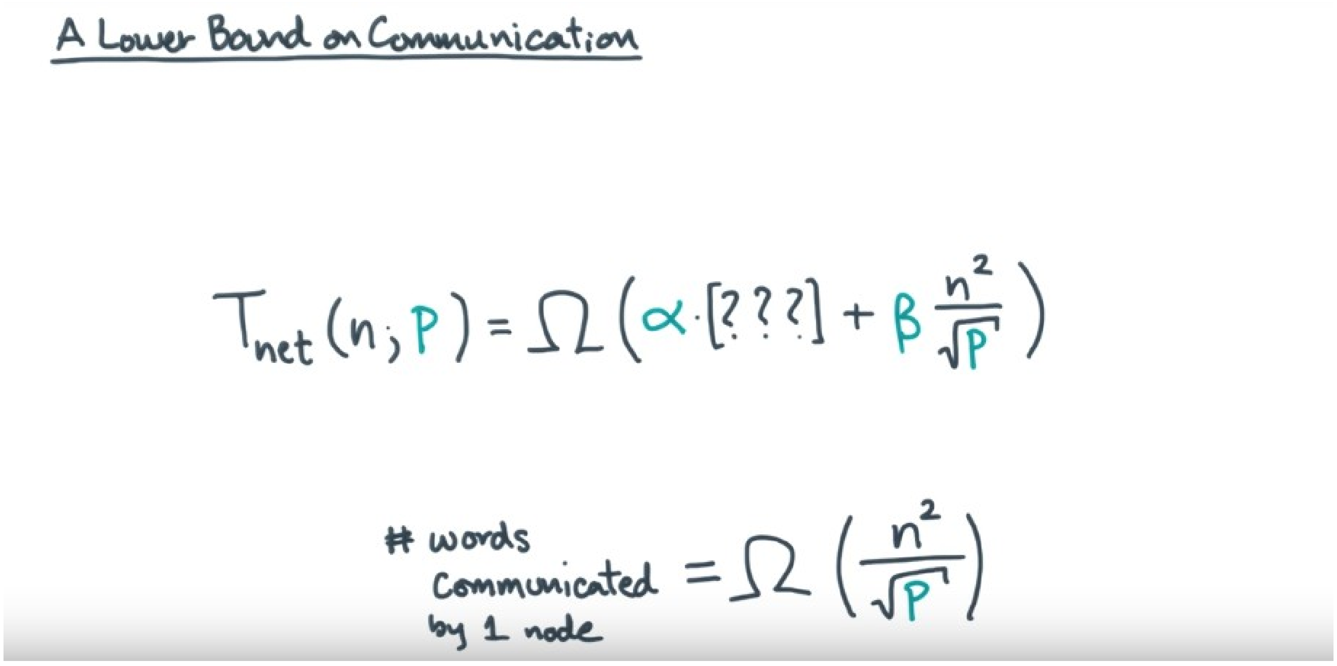
W multiplies M words of mem

Distributed operands

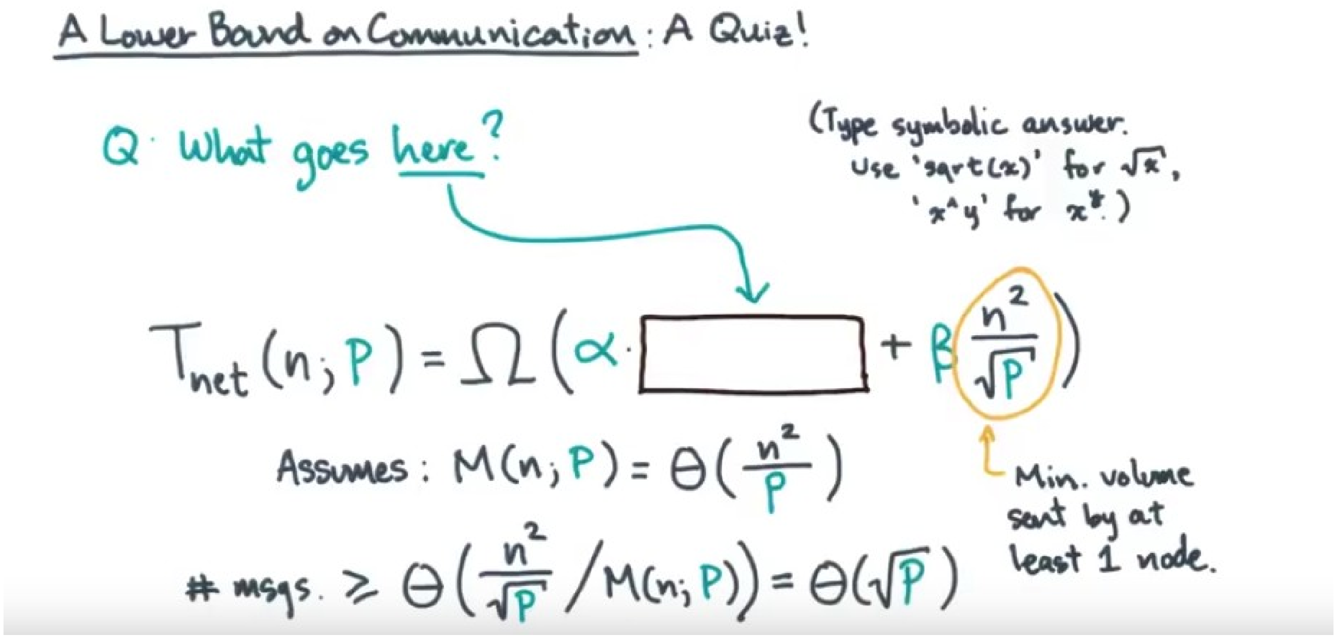
$$\Rightarrow M = \Theta\left(\frac{n^2}{P}\right)$$

$$\#\text{ words communicated by 1 node} = \Omega\left(\frac{n^2}{\sqrt{P}}\right)$$

So if you plugged that in and simplify, you'll get this. So to summarize, at least one node sends and receives n squared over root P words.


A Lower Bound on Communication
$T_{net}(n;P) = \Omega\left(\alpha \cdot [???] + \beta \frac{n^2}{\sqrt{P}}\right)$
# words communicated by 1 node $= \Omega\left(\frac{n^2}{\sqrt{P}}\right)$

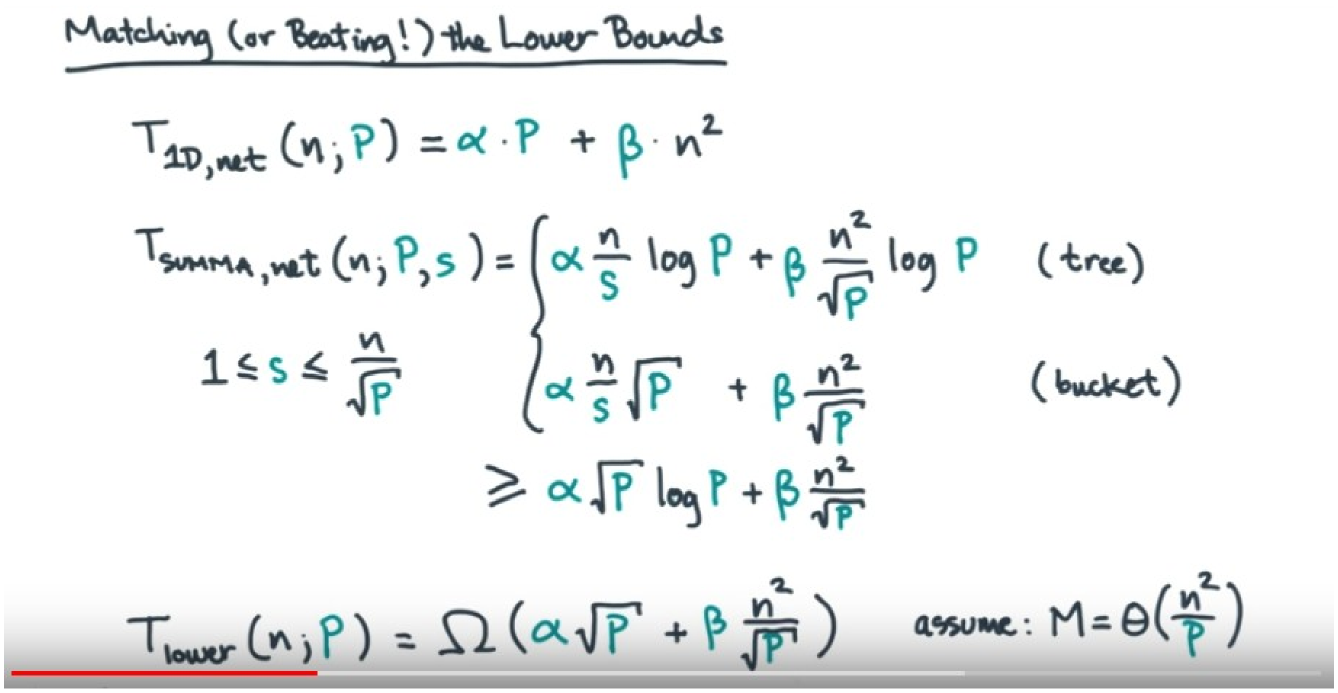
Let's connect this back to the big picture of this segment. This analysis is about the volume of communication seen by at least one processor. In terms of the time to do communication, it allows you to conclude that the bandwidth term should have this form. At least one node sends this much data. And it can only send it at a rate of beta. That's because beta is the minimum time per word. And your lower bound tells you the minimum number of words that at least one node must have sent. So that gives us a lower bound on this beta term. That's super cool. The only question now, is what is this?


A Lower Bound on Communication : A Quiz!
Q: What goes here?
(Type symbolic answer. Use 'sqrt(x)' for $\sqrt{x}$, 'x^y' for x^y.)
$T_{net}(n;P) = \Omega\left(\alpha \cdot \boxed{} + \beta \frac{n^2}{\sqrt{P}}\right)$
Assumes: $M(n;P) = \Theta\left(\frac{n^2}{P}\right)$
# msgs. $\gtrsim \Theta\left(\frac{n^2}{\sqrt{P}} / M(n;P)\right) = \Theta(\sqrt{P})$
Min. volume sent by at least 1 node.

23. So far, you've shown the following lower bound on the running time of matrix multiplied with respect to communication. It assumes square matrices and, furthermore, that every node holds an equal part of a matrix operation. Given this information, can you figure out what goes here? Put another way, can you determine a lower bound on the number of messages that a node must send? I want you

type your answer symbolically. You can use our usual shortcuts like square root of x for root x, or x carat y for x to the y.

24. Remember the main idea of the lower bound analysis. You computed the minimum volume of data sent by at least one node. So if a node sends n squared over root p words, how many messages is that? Well remember every node has a memory of size M, so that means the largest message that a node can send is M. That means the number of messages is at least the total volume divided by M. So simplifying, you'll get this. Since that's the minimum number of messages that the node can send, that becomes the alpha term.



Matching (or Beating!) the Lower Bounds

$$T_{1D,net}(n;P) = \alpha \cdot P + \beta \cdot n^2$$

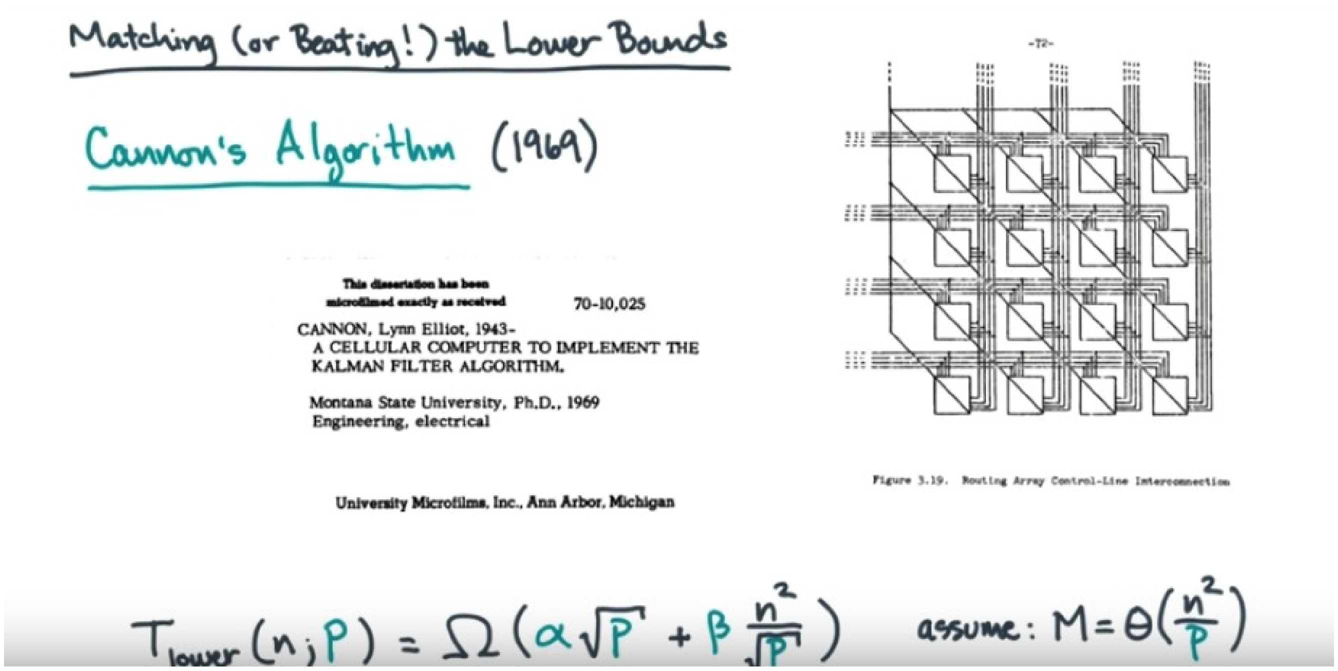
$$T_{SUMMA,net}(n;P,s) = \begin{cases} \alpha \frac{n}{s} \log P + \beta \frac{n^2}{\sqrt{P}} \log P & \text{(tree)} \\ \alpha \frac{n}{s} \sqrt{P} + \beta \frac{n^2}{\sqrt{P}} & \text{(bucket)} \end{cases}$$

$$1 \le s \le \frac{n}{\sqrt{P}}$$

$$\ge \alpha \sqrt{P} \log P + \beta \frac{n^2}{\sqrt{P}}$$

$$T_{lower}(n;P) = \Omega\left(\alpha \sqrt{P} + \beta \frac{n^2}{\sqrt{P}}\right) \qquad \text{assume: } M = \Theta\left(\frac{n^2}{P}\right)$$

25. Let's recap where things stand by connecting the idea of the lower bound to the one d-block row algorithm and the two d-suma algorithm. First, there's the one d algorithm, who's communication time is this. Then, there's the two d-suma algorithm. Remember, there are two variants depending on whether you assume a broadcast is tree based or bucket based. Also remember that the suma algorithm has a tuning parameter, a little s. At the maximum value of s, a lower bound on the communication time would be this. So what I've done is taken the maximum value of s, plugged it into these two parameters, and just taken the best case latency and bandwidth bounds. This way we don't have to fuss with tree based versus bucket based analyses. So, these are times for concrete algorithms. Then, there's the lower bound on communication. So, relative to the lower bounds, SUMA looks pretty good. Matches in the beta term, and is just off by a little bit in the alpha term. So, can you do even better than SUMA?



Matching (or Beating!) the Lower Bounds

Cannon's Algorithm (1969)

This dissertation has been microfilmed exactly as received 70-10,025

CANNON, Lynn Elliot, 1943-
A CELLULAR COMPUTER TO IMPLEMENT THE KALMAN FILTER ALGORITHM.

Montana State University, Ph.D., 1969
Engineering, electrical

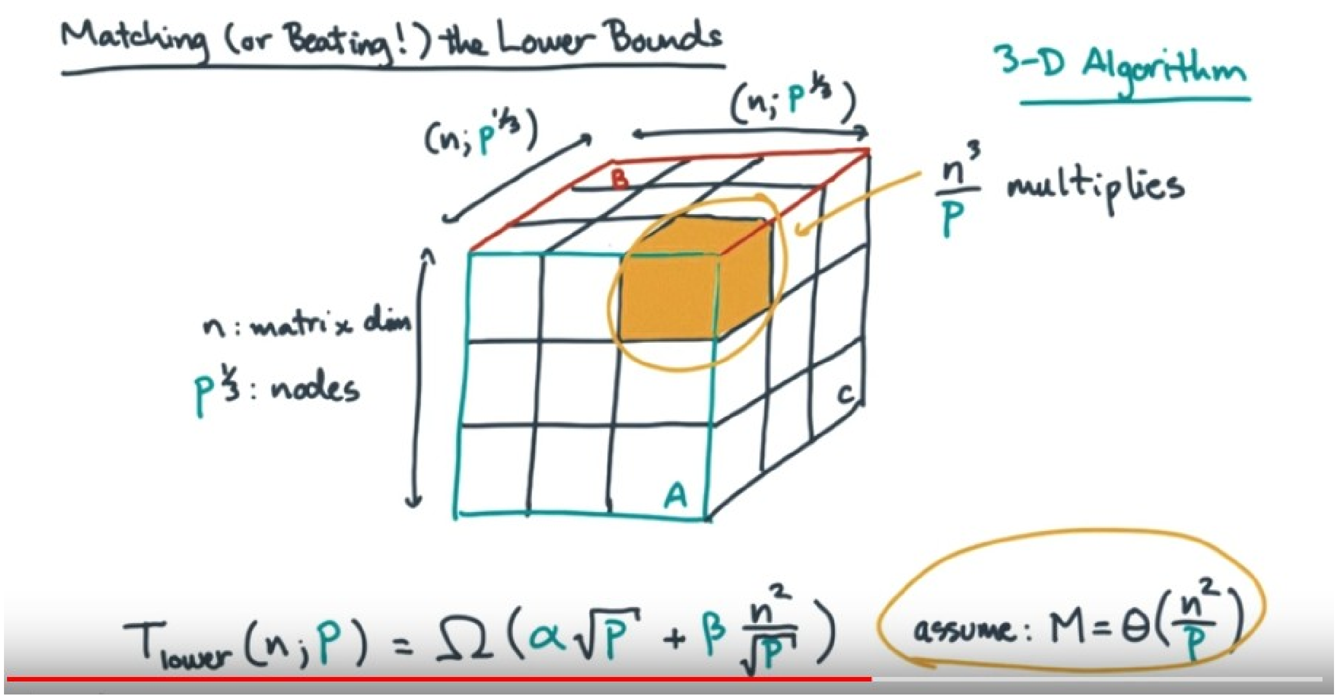
University Microfilms, Inc., Ann Arbor, Michigan

-72-

Figure 3.19. Routing Array Control-Line Interconnection

$$T_{lower}(n; P) = \Omega\left(\alpha\sqrt{P} + \beta\frac{n^2}{\sqrt{P}}\right) \qquad \text{assume: } M = \Theta\left(\frac{n^2}{P}\right)$$

In fact, you can, using an algorithm called Cannon's algorithm. It has a communication time that exactly matches the lower bound. It's an impressive algorithm in the sense that it predates the lower bound analysis. In fact, Cannon developed the algorithm as part of its PhD dissertation in 1969, groovy baby. I wish Mike Myers was here. Unfortunately, Cannon's Algorithm suffers from a few restrictions that make it not quite as easy and practical to implement compared to the SUMMA algorithm. Anyway, if you want more details head to the instructor's notes.

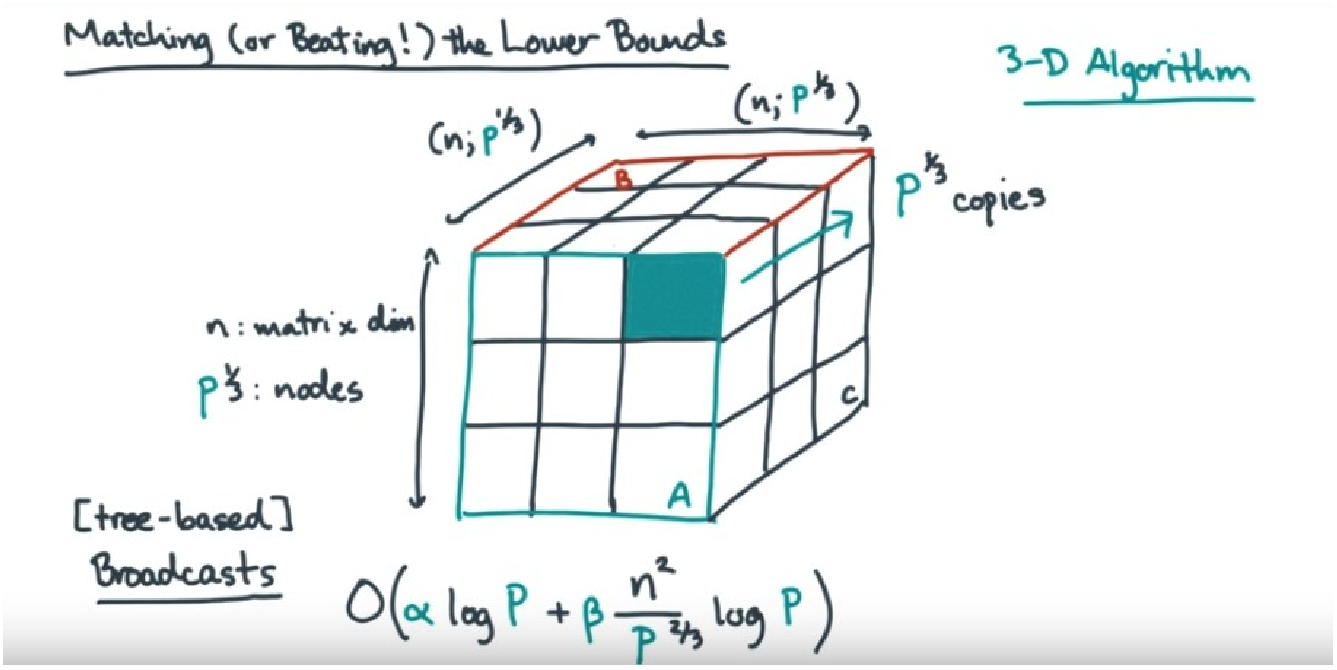


Matching (or Beating!) the Lower Bounds

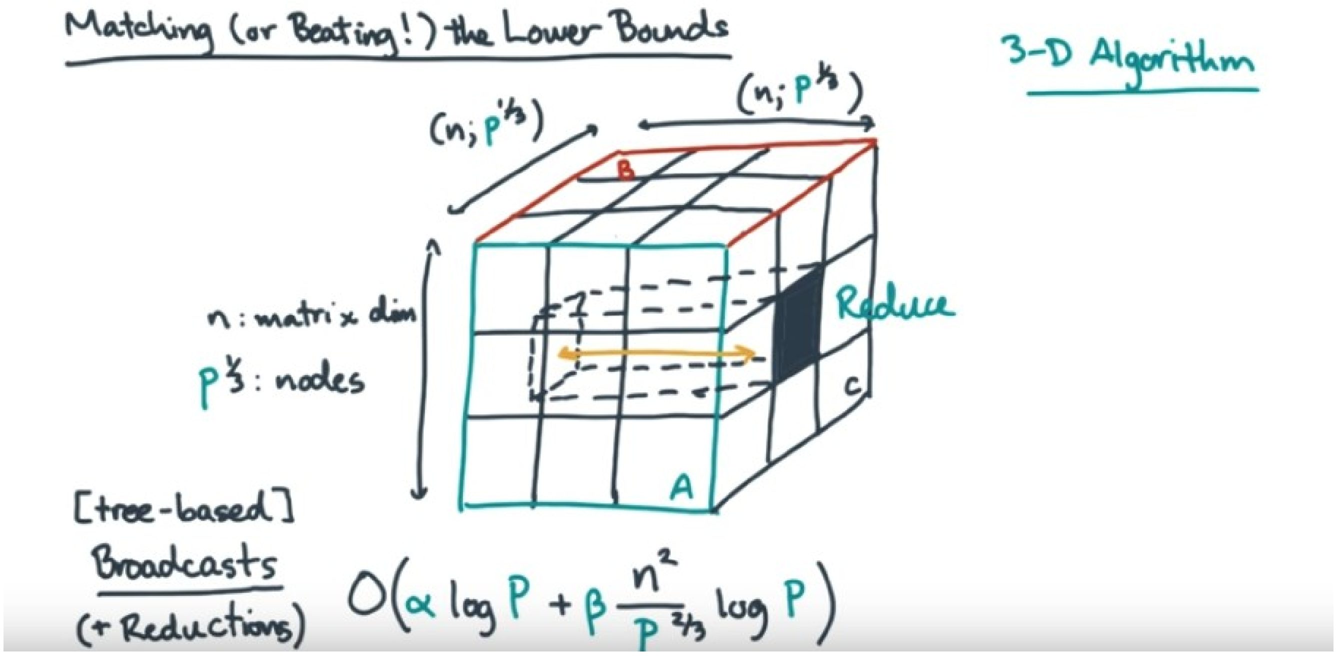
3-D Algorithm

$$T_{lower}(n; P) = \Omega\left(\alpha\sqrt{P} + \beta\frac{n^2}{\sqrt{P}}\right) \qquad \text{assume: } M = \Theta\left(\frac{n^2}{P}\right)$$

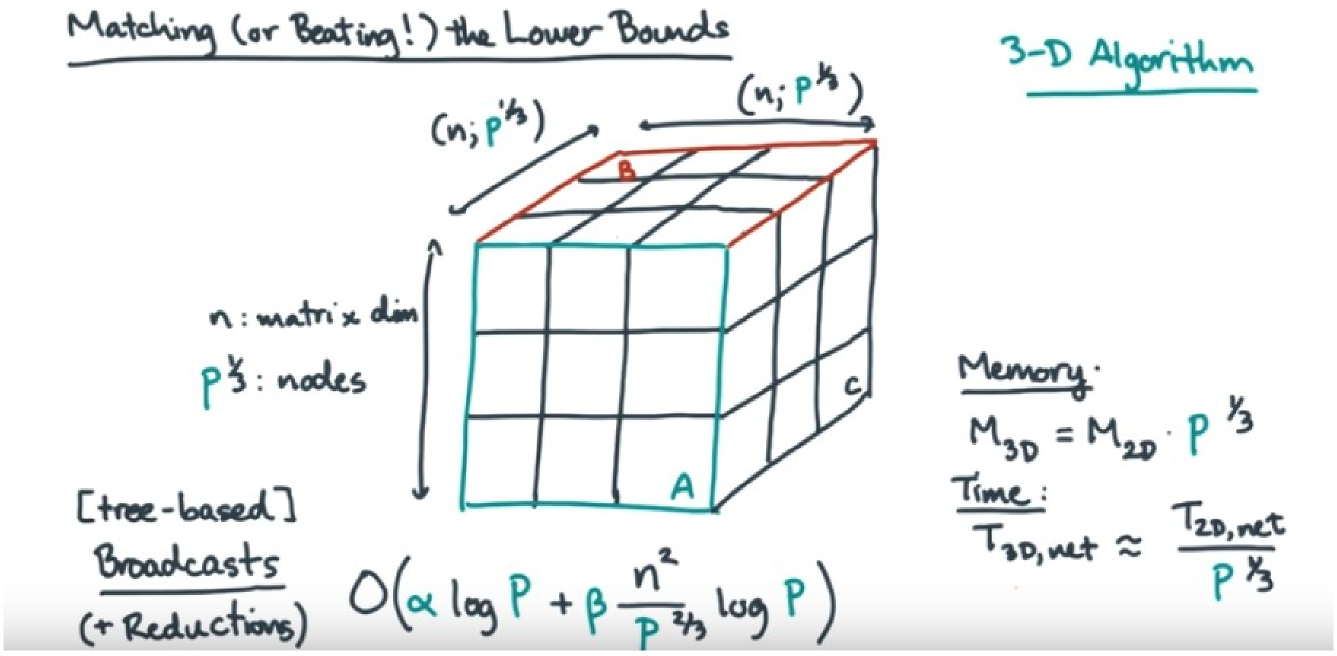
Now what about beating the lower bound? The lower bound analysis makes a critical assumption. Can you spot it? The assumption is this one. That is, you assumed you only had enough memory on each node to store n squared over p data. Recall the cube of multiplications concept. This assumption is akin to distributing surfaces of the cubes across nodes. So a good question is whether having more memory would let you replicate some data, and thereby reduce communication. One example of such a scheme is a three dimensional algorithm. It says rather than distributing the surface is why not distribute the volume. That is suppose you took the nodes and arranged in them in a 3-D mesh, instead of a 2-D mesh. So suppose you have capital P nodes now put them in a 3D mesh of size cube root of p on a side. You could then assign chunks of this n cubed work to each node. Each chunk consists of n cubed over p multiplications. Ideally you'd like to set up this scheme in such a way that every node can update its cube concurrently. Now to do that you're going to need to have cube root of P copies of each submatrix.



For instance, consider these three processes. So you either need to predistribute copies of the matrices, or depending on how the operands are allocated, do some kind of broadcast. Here's what such a broadcast might cost you. In this case, i've assumed a tree-based scheme, and of course, you'd incur the same cost to ++broadcast b and c. Now, once all the upper ends are replicated, you can do local multiplications, and depending on where you want the final results to be, you might need to combine the copies of C.



You can, of course, do that with a reduction, and since reduce is the dual of broadcast, the overall communication cost still matches the broadcast cost. At least asymptotically. You could also do an all reduce to get replicated results within a block column.


Matching (or Beating!) the Lower Bounds
3-D Algorithm
$(n; P^{1/3})$
$(n; P^{1/3})$
B
n : matrix dim
$P^{1/3}$: nodes
C
A
[tree-based]
Broadcasts
(+ Reductions)
$O\left(\alpha \log P + \beta \frac{n^2}{P^{2/3}} \log P\right)$
Memory:
$M_{3D} = M_{2D} \cdot P^{1/3}$
Time:
$T_{3D,net} \approx \frac{T_{2D,net}}{P^{1/3}}$

So let's compare this to our 2-D lower bound. If you have enough memory to replicate by a factor of cube root of P, you can get a cube root of P reduction in the communication time. So what if you don't have this much storage? Well, you could imagine a hybrid or 2.5-D scheme. Rather than full replication, you could use partial replication. That could reduce the memory requirement, but perhaps at a cost of a slightly increased communication time relative to the 3-D case. But wait. You know what? Let's save the two point five d scheme for another time. Like, an exam! Oh, no, not an exam!

26. This point you're probably sick of matrix multiply, but I can't say enough what an important computational primitive it is to study. Now I say that, not only because I think it has important applications, it's also because you can use it to see a lot of different analysis techniques. For instance, in this lesson you saw different algorithmic approaches like 1-D versus 2-D versus 3-D distributions. As well as a way to argue about communication lower bounds. It won't always be easy to do those sorts of analyses on other algorithms, but I hope you'll try. Another reason to think about matrix multiply is that for better or for worse, the machines we build today, if you're a wee bit cynical like me, are tuned for it. For instance, you should have noticed something. When the problem is big enough, there's a lot more work than communication. That means matrix multiply ought to scale well as systems and problems get bigger. It's in that sense that things like the top 500 list, which measure peak capability, might not translate to more communication-intensive applications. Now speaking of big, and I admit this fact as a bit of a non sequitur. Did you know that a Rubix cube has 43 quintillion 252 quadrillion 3 trillion 274 billion 489 million 856 thousand possible configurations? Try telling people that fact at your next cocktail party, and I'm sure you'll be a really big hit. Just like me, because I'm a huge hit at cocktail parties with my numbers and my constants out to ten digits and stuff. Very cool.