# Lesson 2-4 Distributed Matrix Multiply

**Matrix Multiply: Basic Definitions**

C ← C + A*B → This is the dot product of row A and column B and accumulating the sum into the output.

The Matrix Multiply as pseudo code:

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

The Time to complete the algorithm is:

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

The Matrix Multiply as Parallel pseudo code:

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

This means each 'row' and 'column' could actually represent a sub matrix.
The third loop is a reduction.

The Matrix Multiply as Parallel pseudo code:

```
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[:])
```

$W(n) = O(n^3)$
$D(n) = O(\log n)$

**A Geometric View**

Using a cube - the rows and columns can be projected onto the cube. The three matrices are areas on the x, y, z planes. If the the three projections intersect - the matrices can be multiplied. The resulting volume is the set of multiplications that need to be done.

According to Loomis and Witney: The volume of I is ….. |I| <= $\sqrt{|sA|\ |sB|\ |sC|}$

If there is no intersection between the surface projections - there will be no multiplications.

**1D Algorithms**

How should the Matrix multiply be accomplished on a distributed machine?

Using Block Row Distribution : this means each node gets n/P rows. (assume the matrices are square and that n is divisible by P).

What is the most work that can be done by a node given the above data distribution?

Since a, b, c have to be multiplied - one of them has to be shifted, convention says 'B' is shifted.

**1-D Block Row Distribution Pseudo Code**

```
let A'[1: n/P] [1:n] = local part of A
        B', C' = same for B, C

let B''[1:n/P][1:n] = temp storage
let r_next ← (RANK + 1) mod P
    r_prev ← (RANK + P -1) mod P
for L ← 0 to P-1 do
        C'[:][:] += A'[:][...L…] . B'[...L…][:]        (...L… is a placeholder for the local indices)
        sendAsync (B' → r_next)          (send the local buffer to the next processor)
        recvAsync (B'' ← r_prev)         (receive from the previous processor)
        wait(*)                                (wait for send and recv to complete)
        swap(B', B'')                          (swap the receive buffer with the compute buffer)
```

**1D Algorithm Cost**

The cost of the algorithm is …

$\tau =$ *time per* "*flop*" (flop means …1 floating point multiply or add)

Total time is …. $T_{comp}(n,P) = 2\tau\ n^3 / P$

**How much time is spent on communication?**

B' is the only data communicated. It's size is n/P words by n columns - so $n^2/P$ *words*.

There are P sends that have to be paid for.
So the total cost of communication is : $\alpha P + \beta n^2$

**1D Algorithm Cost Part 2**
The running time of the algorithm is flops + communication time.

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

Getting more speed from the algorithm:

Here is the pseudo code, rearranging the code can improve the speed.

```
let A'[1: n/P] [1:n] = local part of A
        B', C' = same for B, C

let B''[1:n/P][1:n] = temp storage
let r_next ← (RANK + 1) mod P
    r_prev ← (RANK + P -1) mod P
for L ← 0 to P-1 do
        C'[:][:] += A'[:][...L…] . B'[...L…][:]
        sendAsync (B' → r_next)
        recvAsync (B'' ← r_prev)
        wait(*)
        swap(B', B'')
```

Rearranged pseudo code:

```
let A'[1: n/P] [1:n] = local part of A
        B', C' = same for B, C

let B''[1:n/P][1:n] = temp storage
let r_next ← (RANK + 1) mod P
    r_prev ← (RANK + P -1) mod P
for L ← 0 to P-1 do
        sendAsync (B' → r_next)
        recvAsync (B'' ← r_prev)
        C'[:][:] += A'[:][...L…] . B'[...L…][:]
        wait(*)
        swap(B', B'')
```

The communication is now done BEFORE the computations - gaining a little speed in the process.

This could improve running time by a factor of 2. We can see this as:

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

This uses the fact that : $a + b \leq 2 * max(a, b)$

**Efficiency and the 1D Algorithm**

Recall the running time: $T_{1D,\,overlap}(n;P) = max(2\tau n^3/P,\ \alpha P + \beta n^2)$

Speedup: $S_{1D}(n;P) \equiv T_*(n) / T_{1D}(n;P)$ = $P / max(1,\ 1/2 * \alpha/\tau * P^2/n^3 + 1/2 * \beta/\tau * P/n)$

= $\theta(P)$

Parallel Efficiency = Speedup / P = E(n; P)

**A parallel system is efficient if its parallel efficiency is constant.** This occurs when:
$n = \Omega(P)$

According to the equation … if you double the number of nodes - you have to double the dimension of the problem. But doubling the size of the dimension (since matrices are involved) will lead to quadruple the size of the matrices. The number of flops will increase by a factor of 8.

If you can't (or don't) double the dimensions you will see diminishing returns in increasing the parallelism.

Isoefficiency Function is: $n = \Omega(P)$ → the value of P that n must satisfy to have constant parallel efficiency.

Temporary Storage: the B'' needs temporary storage. The 3 matrices and 1 temp. matrix.

Temporary Storage → $M(n;P) = (3 + 1)\, n/P * n = 4\, n^2/P$

**IsoEfficiency**

E(n;P) = S(n;P)/P = $T_*$(n) / P*T(n;P)

Parallel Cost = P*T(n;P) = $1/((1+P/\tau)log\ P\ +\ (\alpha/\tau)(\ log\ P)/n$
$\tau =$ *time per scalar add*

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

Notice: there is no setting of n that will make E(n;P) a constant.

$1/((1+P/\tau)log\ P$ → goes to infinity as n increases.

**A 2D Algorithm SUMMA**

Begins with a 2D distribution of the matrix operands. Each node is responsible for updating the part of the 'C' matrix that it owns.

The SUMMA algorithm begins by integrating strips of width and height l.
So:

```
for l ← 1 to n/S do
        broadcast(horizontal strip, (owner))
        broadcast(vertical strip,(owner))
```

The run time:
Assume: nxn matrices, $\sqrt{P}$ $\sqrt{P}$ mesh, $\sqrt{P}$*and s both divisible by n.*

$T_{SUMMA}(n;P,s)\ =\ n/s\ *\ (2\tau\ *(n^2s/P)\ +\ T_{net}(n;P,s)$
$=\ 2\tau n^3/P\ +\ T_{net}(n;P,s)$

**SUMMA Communication Time**

$T_{net}\ =\ O(\alpha * n/s\ *\ log\ P\ +\ \beta * n^2/\sqrt{P} * log\ P)$ → Tree

$T_{net}\ =\ O(\alpha * n/s\ *\ P\ +\ \beta * n^2/\sqrt{P})$ → Bucket

**Efficiency of 2D SUMMA**

The 2D SUMMA is more scalable than the 1D Block.

Note: ‘s’ is the width of the strip, it is also the tuning parameter of the algorithm.

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

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

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

The bucket is slightly worse than the tree, it trades a higher latency cost for a lower communication cost.

**SUMMA Memory**
The amount of memory needed for SUMMA is:

$M_{SUMMA} = 3 * n^2/P + 2 * s * n/\sqrt{P})$

$> 4 * n^2/P$ when $s > 1/2 * n/\sqrt{P}$

A smaller ‘s’ increases latency time.

If s is at it’s maximum value, the SUMMA algorithm might need 5 times $\sqrt{P}$ amount of storage.

**A Lower Bound on Communication**

lower bound - the number of words a node MUST communicate.

each phase sends and receives exactly ‘m’ words.

$S_A, S_B, S_c \rightarrow$ the set of unique elements of each matrix seen in this phase.

$Max\ \#\ multiplies\ per\ phase \leq \sqrt{|S_A| * |S_B| * |S_C|} \leq 2 * \sqrt{2} * M^{3/2}$

$L \geq \#\ full\ Phases \geq [W/max\ \#\ multiplies\ per\ phase]_{\text{FLOOR}}$

$L \geq W/(2\sqrt{2} * M^{3/2}$

# words communicated by 1 node $\geq (\#\ full\ phases) * M$

# words communicated by 1 node = $\Omega(n^2/\sqrt{P})$

**A Lower Bound on Communication**

$T_{net}(n;P) = \Omega(\alpha * \sqrt{P} + \beta * n^2/\sqrt{P})$

$T_{\text{SUMMA, net}}(n; P,s) = \left\{ \begin{array}{ll} \alpha * n/s * \log P + \beta * n^2/\sqrt{P} * \log P & (tree) \\ \alpha * n/s * \sqrt{P} + \beta * n^2/\sqrt{P} & \textit{(bucket)} \end{array} \right\}$

(assume: $1 \leq s \leq n/\sqrt{P}$)

$T_{Lower}(n;P) = \Omega(\alpha\sqrt{P} + \beta * n/\sqrt{P})$ assume: $M=\theta(n^2/P)$

Cannon's