EE/CSCI 451: Parallel and Distributed Computation

Lecture #6

9/3/2020

Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

USC University of Southern California

Announcements

- HWs
  - PHW2 released Tuesday, due 9/14 AOE
  - HW2 just released today, due 9/10 AOE
- See Lecture 1 for grading policies (for new students)
- Course project
  - Team of 2-3 students
  - Proposal due Oct. 16 (Friday) (Format will be on Piazza later)

Outline

- From last class
  - Synchronous/Asynchronous execution
  - PRAM as a simple synchronous parallel model
  - OpenMP
    - Higher level of programming abstraction
    - Threads based
    - Fork Join
    - Section, Blocks parallelization
    - Loop parallelization

- Today (Chapter 6)
  - Message passing
  - Send and Receive operations
  - Matrix Multiplication (Cannon's algorithm)
  - Communication cost
  - Examples, performance issues

Message Passing Programming Model (1)

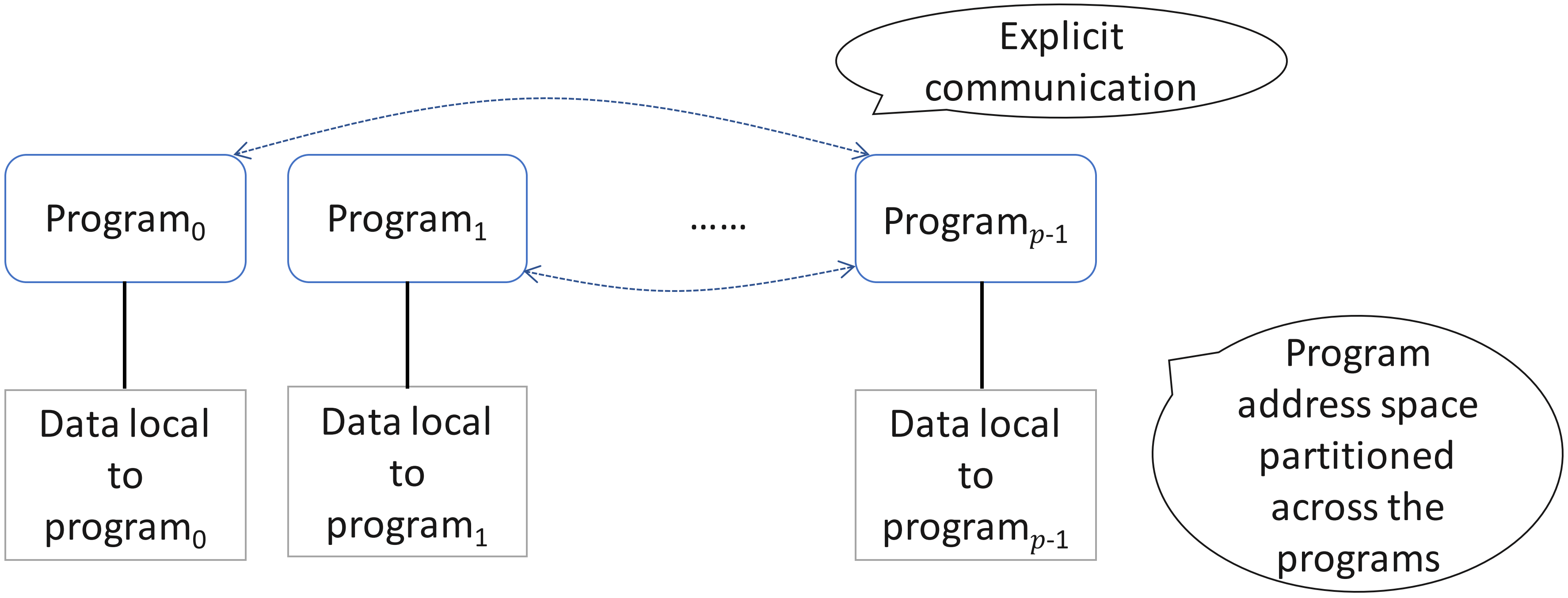
- Message passing
  - One of the oldest parallel programming paradigms
  - Widely used
  - Key features
    - Partition address space

      → local data, remote data
    - Explicit parallelization

      → user is responsible to specify and manage concurrency

Can be challenging

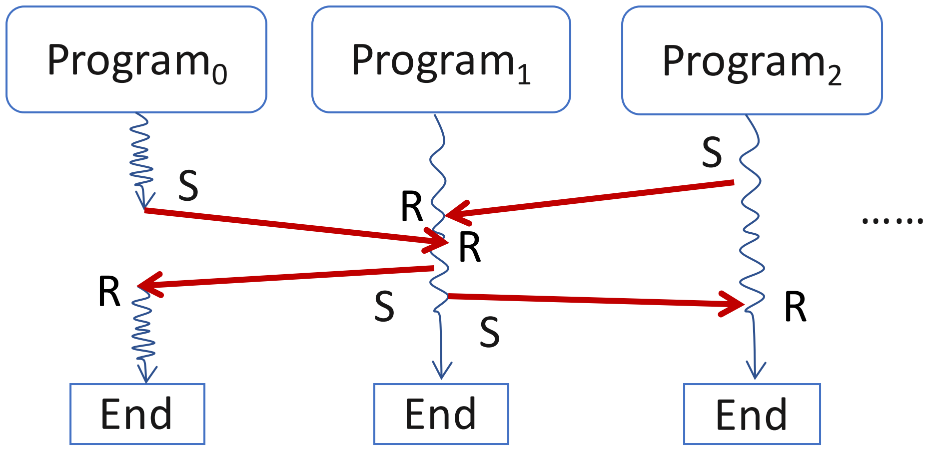
Message Passing Programming Model (2)



Communication - needs coordination among the communicating processes (and the hosts for the two processes)

Message Passing Program (1)

Most General Model: Asynchronous

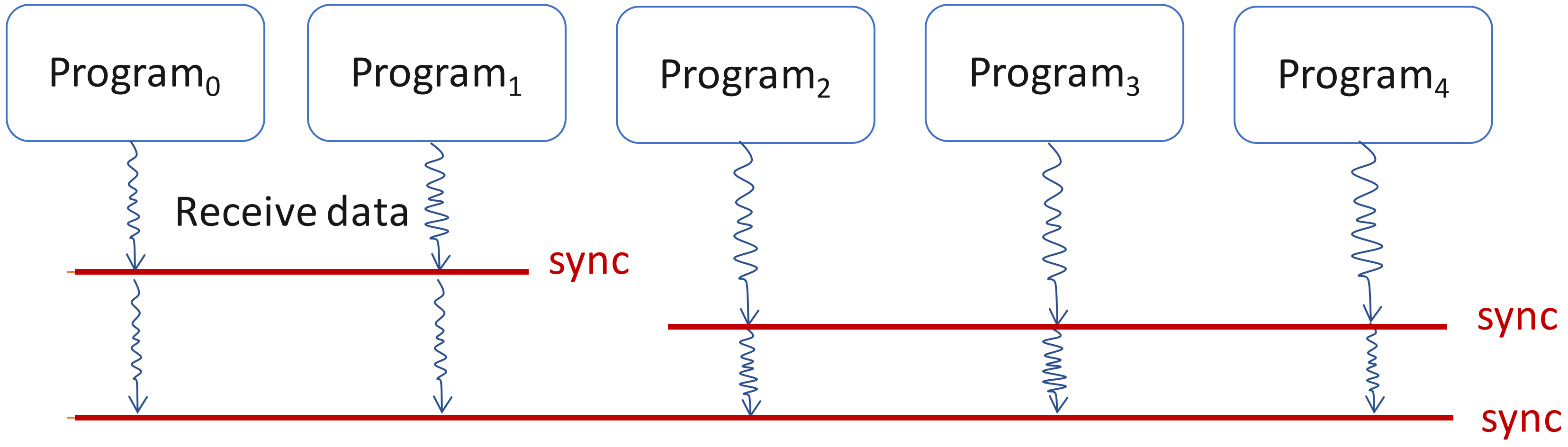


S = send
R = receive

- No structure with respect to instructions, interactions
- No global clock
- Execution is asynchronous
- Programs $0,1, \dots, p-1$ can be all distinct
- Hard to write/debug

Message Passing Program (2)

Loosely synchronous



Some structure

Easier to reason about than asynchronous execution model

Message Passing Program (3)

SPMD (Single Program Multiple Data)

- Code is same in all the processes except for initialization
- Restrictive model, easy to write and debug
- Widely used

In all 3 cases (models of concurrency)

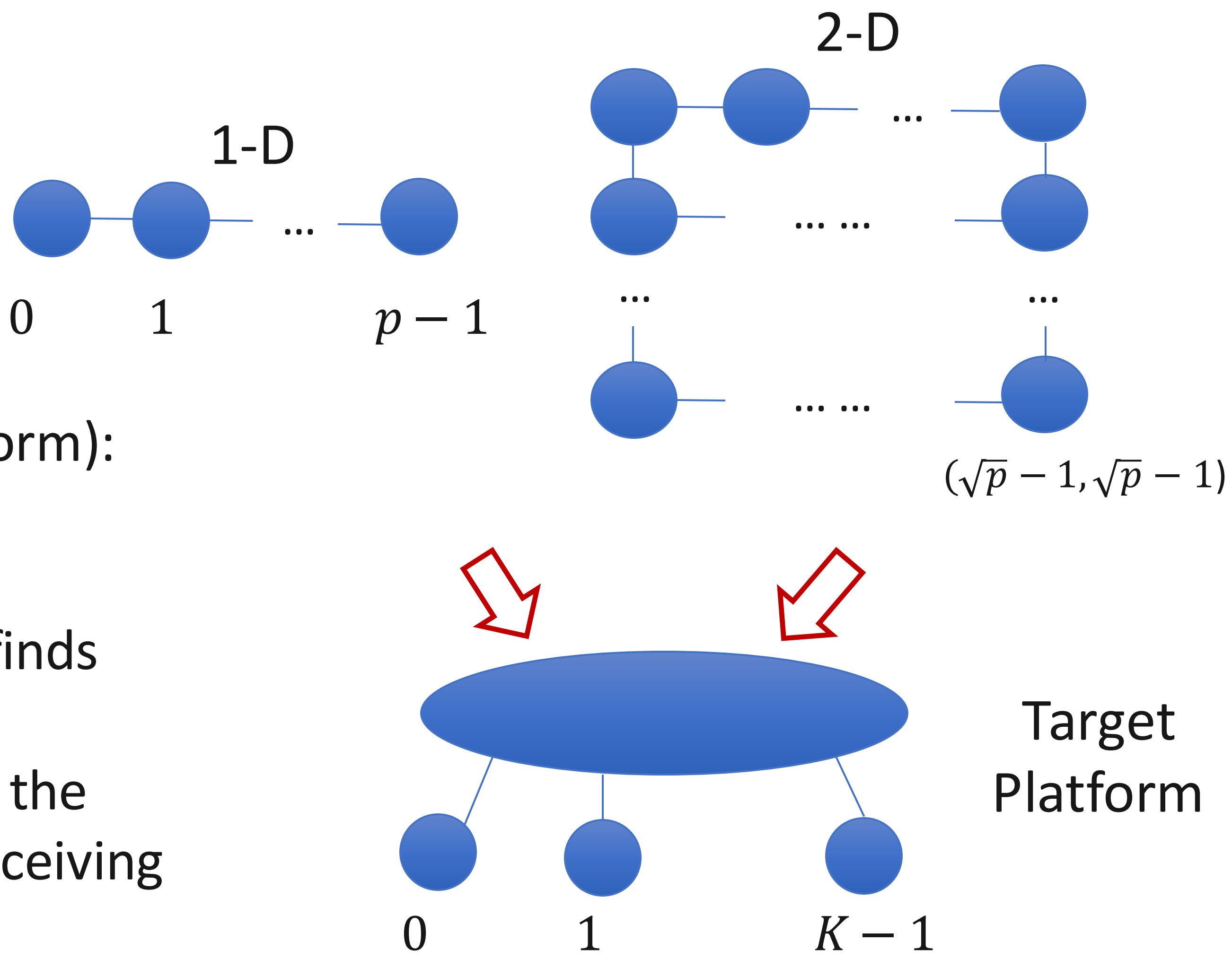
Correctness

Irrespective of the rate of execution of each program, should produce the correct result for every input data (problem instance)

Message Passing Program Specification

User specifies:
- Processes
- Process layout
- Data layout

Embedding (into target platform):
- Specified by
  - User or
  - MPI system software finds the most appropriate mapping that reduces the cost of sending and receiving messages



Send and Receive (1)

Send and Receive operations

P_0

Send (sendbuf,
size,
dest)

data

size

Destination process ID

P_1

Receive (
recvbuf,
size,
source)

Source process ID

- Send data from process 0 to process 1 (processor 0 and processor 1)
- Sent data = data at the **beginning** of the execution of Send
- Send and Receive should be matched (for ex. use process IDs)
- Complications may arise due to the way the software and hardware implement the operation

Send and Receive (2)

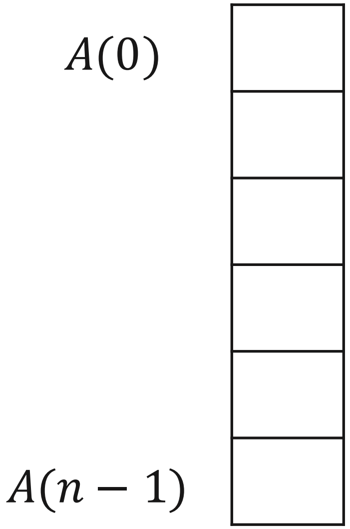
Issues

- What data is sent?
- Buffered?
- Sending process: wait until completion of communication?
- Overheads at sender, at receiver

Adding Using Message Passing (1)

Start with adding on PRAM

Output $= \sum_{i=0}^{n-1} A(i)$ in $A(0)$



Adding using Message Passing (2)

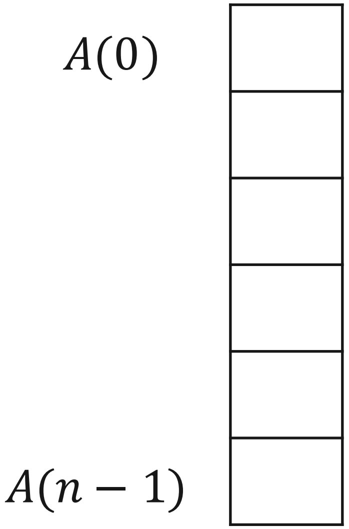
- **PRAM** Algorithm (from before)

Program in processor $j, 0 \leq j \leq n-1$

1. Do $i = 0$ to $\log_2 n - 1$
2. If $j = k \cdot 2^{i+1}$, for some $k \in N$
   $$A(j) \leftarrow A(j) + A(j + 2^i)$$
3. end

Note:

A is **shared** among all the processors (in case of PRAM)
Synchronous operation [For ex. all the processors execute instruction 2 during the same cycle, $\log_2 n$ time]
N = set of natural numbers = {0, 1, …}
Parallel time = $O(\log n)$ cycles



Adding using Message Passing (3)

- Message Passing Algorithm (SPMD model)

Program in process $j, 0 \leq j \leq n-1$

1. Do $i = 0$ to $\log_2 n - 1$
2. If $j = k \cdot 2^{i+1} + 2^i$, for some $k \in N$
3. **Send $A(j)$ to process $j - 2^i$**
4. Else if $j = k \cdot 2^{i+1}$, for some $k \in N$
5. **Receive $A(j + 2^i)$ from process $j + 2^i$**
6. $A(j) \leftarrow A(j) + A(j + 2^i)$
7. End
8. **Barrier**
9. End

2^i distance communication

Note:
$A(j)$ is local to process j
N = set of natural numbers = {0, 1, …}
Parallel time = $O(\log n)$ iterations

Adding using Message Passing (4)

- Communication between processes
  - Power of 2 connections (Communication between processes whose IDs differ by 1, 2, 4,…)
  - e.g. Hypercube

Total amount of communication = $O(n)$

Total number of communication steps = log n

MM using Message Passing (1)

Cannon's algorithm

$\boldsymbol{C} \leftarrow \boldsymbol{A} \times \boldsymbol{B}$

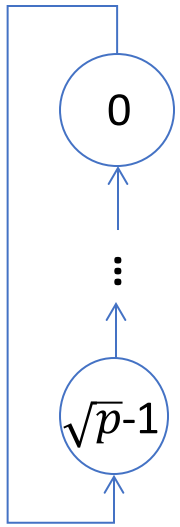
- $\boldsymbol{n} \times \boldsymbol{n}$ matrices
- $\sqrt{\boldsymbol{p}} \times \sqrt{\boldsymbol{p}}$ processors (processes), P_{ij} $0 \leq i, j < \sqrt{p},\ 1 \leq \sqrt{p} \leq n$
- Processor $\boldsymbol{P}_{i,j}$ assigned to $\boldsymbol{A}_{i,j}$, $\boldsymbol{B}_{i,j}$, $\boldsymbol{C}_{i,j}$ (this data **local** to the processor)

(i, j)th block of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

MM using Message Passing (2)

Circular left shift

Circular up shift



MM using Message Passing (3)

Initial data alignment

For $\boldsymbol{A}$:

i^{th} row – circular left shift by i $(0 \leq i < \sqrt{p})$

For $\boldsymbol{B}$:

j^{th} column – circular up shift by j $\left(0 \leq j < \sqrt{p}\right)$

4×4 matrix

4×4 processor array

$\boldsymbol{A_{0,0}}$ $\boldsymbol{B_{0,0}}$	$\boldsymbol{A_{0,1}}$ $\boldsymbol{B_{1,1}}$	$\boldsymbol{A_{0,2}}$ $\boldsymbol{B_{2,2}}$	$\boldsymbol{A_{0,3}}$ $\boldsymbol{B_{3,3}}$
$\boldsymbol{A_{1,1}}$ $\boldsymbol{B_{1,0}}$	$\boldsymbol{A_{1,2}}$ $\boldsymbol{B_{2,1}}$	$\boldsymbol{A_{1,3}}$ $\boldsymbol{B_{3,2}}$	$\boldsymbol{A_{1,0}}$ $\boldsymbol{B_{0,3}}$
$\boldsymbol{A_{2,2}}$ $\boldsymbol{B_{2,0}}$	$\boldsymbol{A_{2,3}}$ $\boldsymbol{B_{3,1}}$	$\boldsymbol{A_{2,0}}$ $\boldsymbol{B_{0,2}}$	$\boldsymbol{A_{2,1}}$ $\boldsymbol{B_{1,3}}$
$\boldsymbol{A_{3,3}}$ $\boldsymbol{B_{3,0}}$	$\boldsymbol{A_{3,0}}$ $\boldsymbol{B_{0,1}}$	$\boldsymbol{A_{3,1}}$ $\boldsymbol{B_{1,2}}$	$\boldsymbol{A_{3,2}}$ $\boldsymbol{B_{2,3}}$

A and B after initial alignment

MM using Message Passing (4)

Parallel algorithm (global view)

1. Initial data alignment
2. Repeat $\sqrt{p}$ times

Super step

- All processors $P_{i,j}$ perform $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ matrix multiplication in parallel using **local** data
- In parallel for all i, j

  Processor $P_{i,j}$:

  circular left shift A $(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}})$ by 1 position in each row
- In parallel for all i, j

  Processor $P_{i,j}$:

  circular up shift B $(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}})$ by 1 position in each col

End

Data alignment using message passing

(permutation in each row and each col)

Note:

A, B, C are partitioned : $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ matrices, **local** to each processor

MM using Message Passing (5)

Parallel algorithm (local view from $P_{i,j}$)

Repeat $\sqrt{p}$ times

Super step

- $C \leftarrow C + A \times B$ → $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ matrix multiplication
- $A \leftarrow$ read from right neighbor from $\{i, (j+1) \bmod \sqrt{p}\}$
- $B \leftarrow$ read from neighbor below from $\{(i+1) \bmod \sqrt{p}, j\}$

End

MM using Message Passing (6)

Cannon's algorithm

Illustration
(4×4 matrix,
4×4 processor array)

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,0}$ $B_{0,2}$	$A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,0}$ $B_{0,1}$	$A_{3,1}$ $B_{1,2}$	$A_{3,2}$ $B_{2,3}$

- Initial alignment

Super step 0
- Compute using local data
- Circular left shift $\boldsymbol{A}$
- Circular up shift $\boldsymbol{B}$

- Note: It is easy to visualize/understand using *p=n*

MM using Message Passing (7)

Cannon's algorithm

$A_{0,1}$ $B_{1,0}$	$A_{0,2}$ $B_{2,1}$	$A_{0,3}$ $B_{3,2}$	$A_{0,0}$ $B_{0,3}$
$A_{1,2}$ $B_{2,0}$	$A_{1,3}$ $B_{3,1}$	$A_{1,0}$ $B_{0,2}$	$A_{1,1}$ $B_{1,3}$
$A_{2,3}$ $B_{3,0}$	$A_{2,0}$ $B_{0,1}$	$A_{2,1}$ $B_{1,2}$	$A_{2,2}$ $B_{2,3}$
$A_{3,0}$ $B_{0,0}$	$A_{3,1}$ $B_{1,1}$	$A_{3,2}$ $B_{2,2}$	$A_{3,3}$ $B_{3,3}$

- Initial alignment

Super step 0

- Compute using local data
- Circular left shift $\boldsymbol{A}$
- Circular up shift $\boldsymbol{B}$

Super step 1

- Compute using local data
- Circular left shift $\boldsymbol{A}$
- Circular up shift $\boldsymbol{B}$

MM using Message Passing (8)

Cannon's algorithm

$A_{0,2}$ $B_{2,0}$	$A_{0,3}$ $B_{3,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$	$A_{1,0}$ $B_{0,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$	$A_{2,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$	$A_{3,2}$ $B_{2,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,0}$ $B_{0,3}$

- Initial alignment

Super step 0
- Compute using local data
- Circular left shift $\boldsymbol{A}$
- Circular up shift $\boldsymbol{B}$

Super step 1
- Compute using local data
- Circular left shift $\boldsymbol{A}$
- Circular up shift $\boldsymbol{B}$

Super step 2
- Compute using local data
- Circular left shift $\boldsymbol{A}$
- Circular up shift $\boldsymbol{B}$

MM using Message Passing (9)

Cannon's algorithm

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

- Initial alignment

Super step 0
  - Compute using local data
  - Circular left shift $\boldsymbol{A}$
  - Circular up shift $\boldsymbol{B}$

Super step 1
  - Compute using local data
  - Circular left shift $\boldsymbol{A}$
  - Circular up shift $\boldsymbol{B}$

Super step 2
  - Compute using local data
  - Circular left shift $\boldsymbol{A}$
  - Circular up shift $\boldsymbol{B}$

- Super step 3
  - Compute using local data

MM using Message Passing (10)

Performance analysis

- Total number of multiply and add operations in each super step (in each PE):

$$(\frac{n}{\sqrt{p}})^3 \text{ multiplications and } (\frac{n}{\sqrt{p}})^3 \text{ additions}$$

- Total number of super steps: $\sqrt{p}$
- Total number of operations (over all the PEs):

$$(\frac{n}{\sqrt{p}})^3 \times \sqrt{p} \times \left(\sqrt{p} \times \sqrt{p}\right) = n^3 \text{ multiplications}$$

$$(\frac{n}{\sqrt{p}})^3 \times \boxed{\sqrt{p}} \times \boxed{\left(\sqrt{p} \times \sqrt{p}\right)} = n^3 \text{ additions}$$

Number of super steps

Number of processors

- Total amount of data communicated (data received) over all the PEs=

$$\boxed{p} \cdot (2\frac{n}{\sqrt{p}}\frac{n}{\sqrt{p}}) \cdot \boxed{\sqrt{p}} = O(n^2 \cdot \sqrt{p})$$

Number of processes

Number of super steps

MM using Shared Variable (1)

$$C \leftarrow A \times B$$

$n \times n$

Each thread(i, j) is responsible to update $C(i, j)$, $0 \leq i, j < n$

A and B are shared variables

MM using Shared Variable (2)

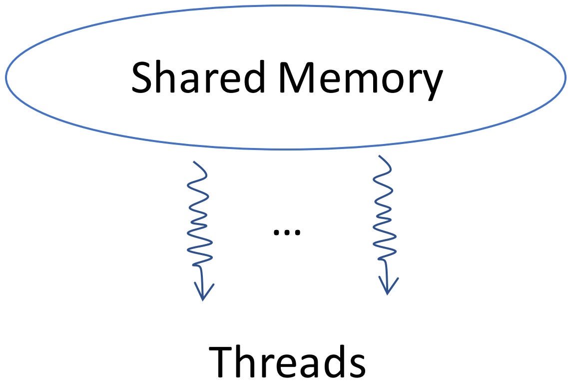
Thread$(\boldsymbol{i}, \boldsymbol{j})$

$C(i,j) \leftarrow 0$

Do k from 0 to $n-1$

$$C(i,j) \leftarrow C(i,j) + A(i,k) * B(k,j)$$

End



MM using Message Passing (1)

$$\underbrace{C}_{n \times n} \leftarrow A \times B$$

n processes

Data Layout

Process i i^{th} row of A, i^{th} col of B, $0 \leq i \leq n$
$2n$ elements/process

Process i Compute i^{th} row of C, $0 \leq i \leq n$

MM using Message Passing (2)

Process $\boldsymbol{i}$

Do $j = 0$ to $n - 1$

"Broadcast"

Received Data

If $i = j$, transmit column j of B to all processes

Receive j^{th} column of B from process j

$C(i, j) \leftarrow$ Inner product $[i^{th}\ row\ of\ A\ \times\ j^{th} column\ of\ B]$

Local Data

End

MM using Message Passing (3)

Transmit column j of B to all processes

Do $l = 0$ to $n - 1$

Send column j of B from process j to process l

Send the data one by one

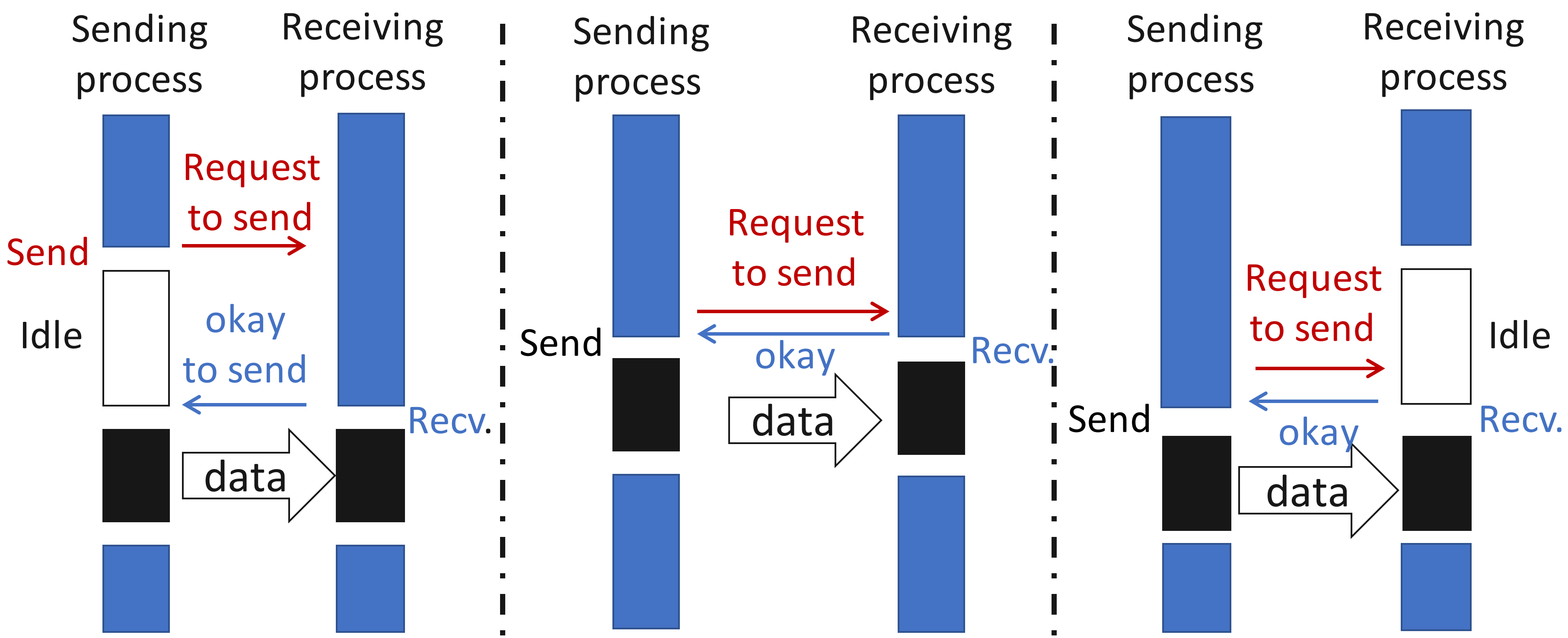
Blocking Send/Receive

Blocking semantics

- Data sent = data at the time the Send command was initiated
- To ensure correctness, block the send operation till some condition to ensure semantics of Send
- **Blocking non-buffered Send**
  - Block sending process
  - Send request to receiving process
  - Wait for receiving process to acknowledge (matched receive operation)
  - Upon receiving acknowledgement, start the transfer
  - No buffers are used for data to be sent

Blocking Send/Receive

Idling overheads



(a) Sender comes first; idling at sender

(b) Sender and Receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at Receiver

Deadlock (1)

Example (1)

blocked

P_0

1. send($\&a$, 1,1);
2. receive($\&b$, 1,1);

P_1

1. send($\&a$, 1,0);
2. receive($\&b$, 1,0);

Deadlocks are very easy in blocking protocols

Deadlock (2)

Example (2)

If myId = even

Send

Receive

If myId = odd

Receive

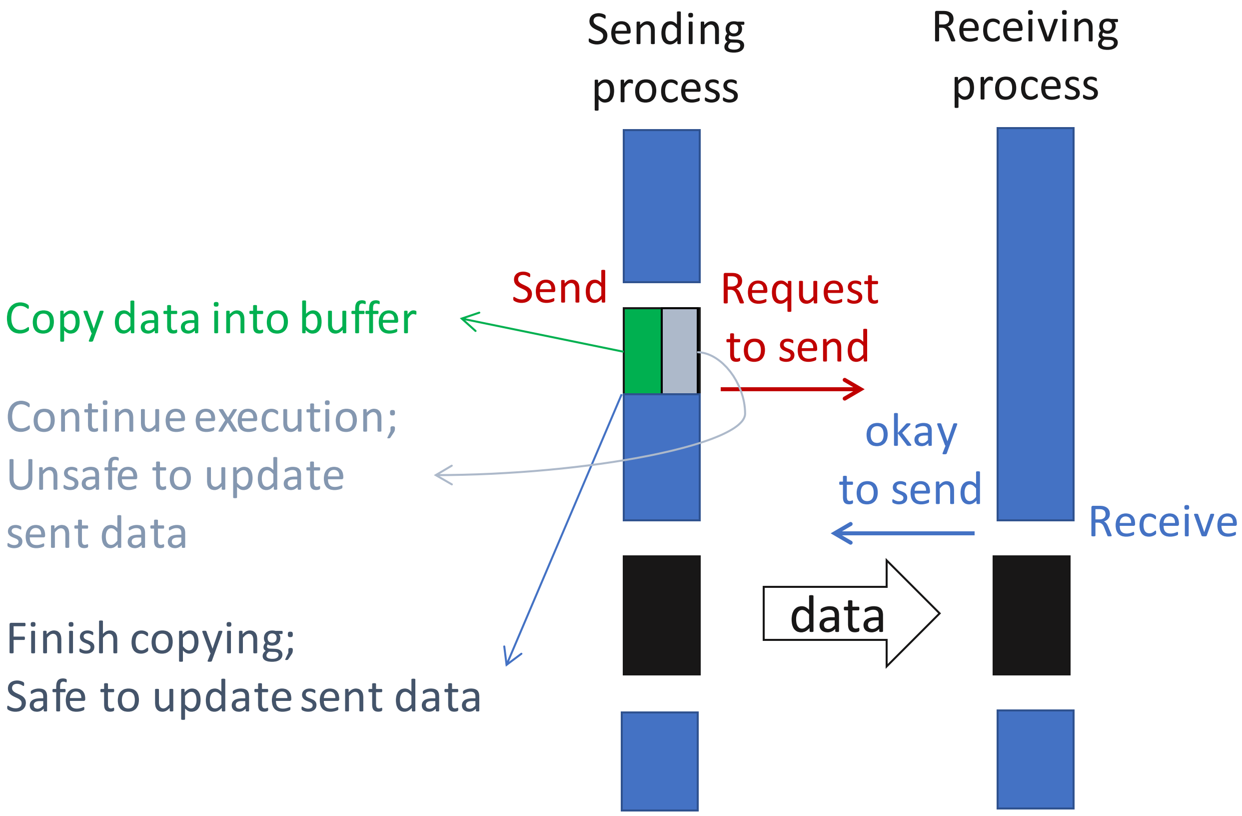
Send

P_0	P_1
Send	Receive
Receive	Send

Non-blocking Send/Receive (1)

- Fast send/receive (reduced overhead)
- Let the programmer manage semantic correctness
- Send:
  - Perform simple initiation, setup
  - Return control immediately
- User should **not** alter data immediately after issuing Send. However, user can do other (useful) operations
- Status information available for user to check
  - Example: check-status

Non-blocking Send/Receive (2)



Summary of Blocking Send

Non-buffered

- Data sent = data at the time the Send command was initiated
- Issue send request and block sending process
- Start data transfer after receiving acknowledgement from receiving process
- Return control to sending process after communication completion
  - Eg. Receiving process has received the entire data

Summary of Non-Blocking Send

- Data sent = data at the time the Send command was initiated
- Copy data into send buffer then return control to sending process immediately
- User can alter sent data after they have been copied into buffer

Additional Materials in Textbook

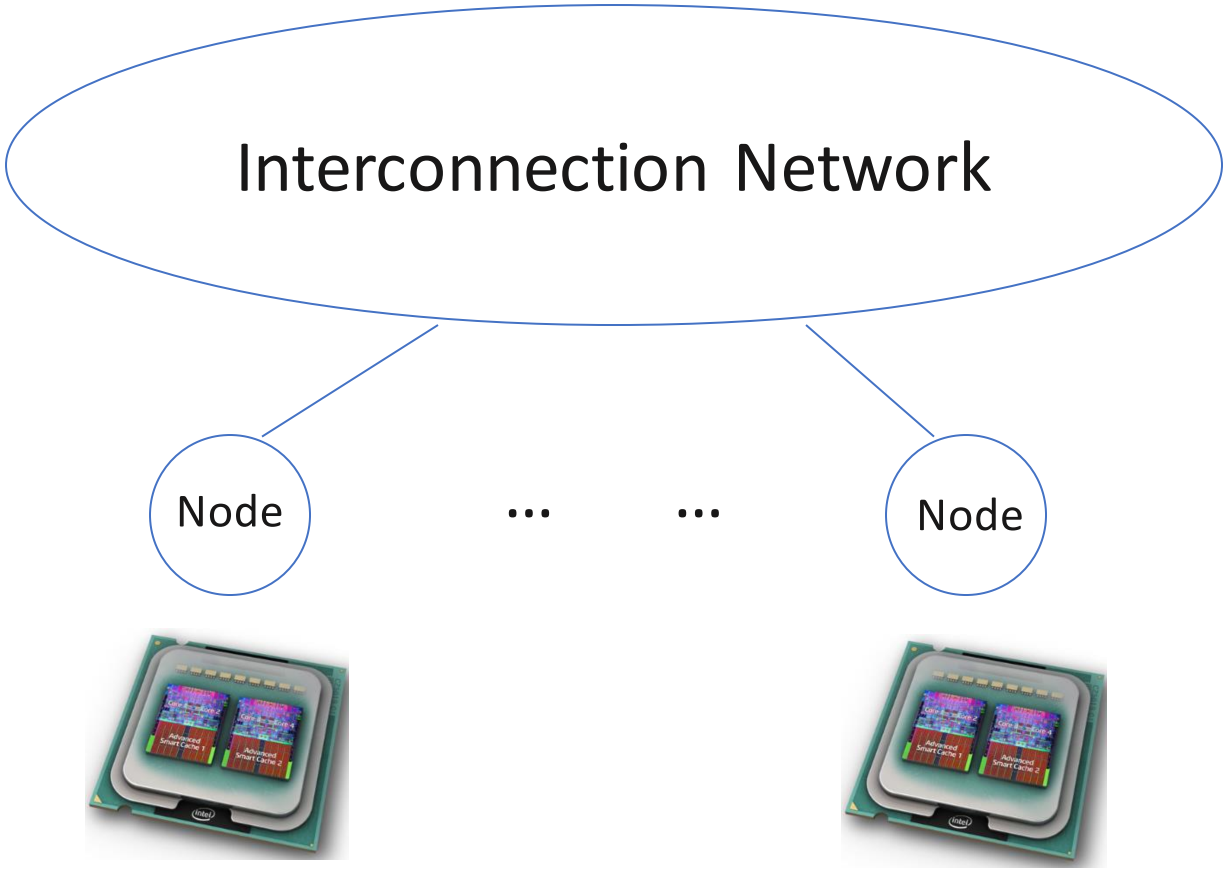
- These are **not** required for this class
  - Non-blocking non-buffered send/receive operations with communication hardware support
  - Non-blocking non-buffered send/receive operations without communication hardware support

OpenMP or MPI ?

MPI →

OpenMP →

Multicore
Shared-Memory



Summary

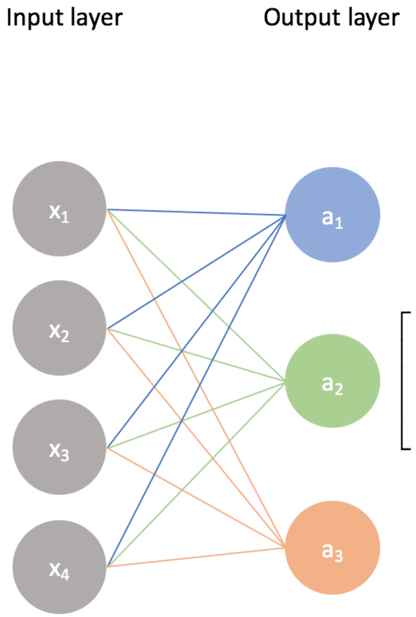
- Send and Receive operations
  - Blocking / non-blocking
  - Issues
    - Overhead
    - Performance
    - Correctness
    - Deadlock
  - Data layout, communication, coordination—user responsibility

41

Further Reading

Applications of MM: Neural Networks (1)

- A neural network takes input data, multiplies them with a weight matrix and applies an activation function.



A simple neural network

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow[\text{activation}]{} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

https://www.jeremyjordan.me/intro-to-neural-networks/

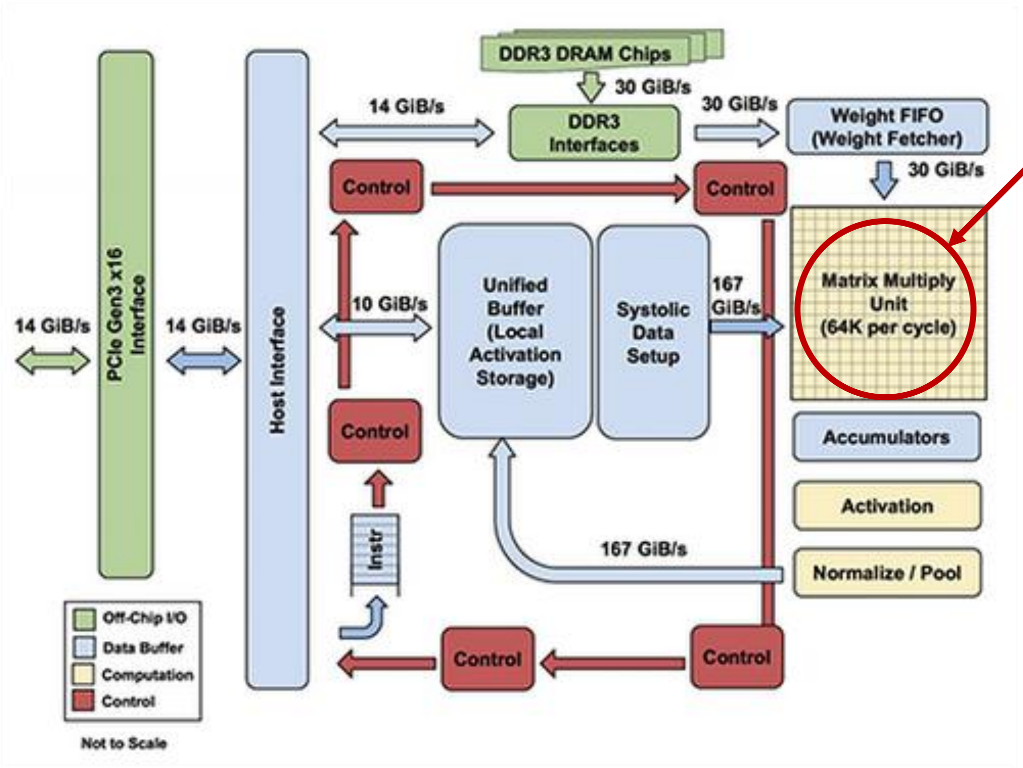
Applications of MM: Neural Networks (2)

- Operating on a batch of inputs, sequence of multiplications and additions can be written as a **matrix multiplication**.

- Even when working with much more complex neural network model architectures, multiplying matrices is often the **most computationally intensive part** of running a trained model.
  - For example - a Convolution layer can be equivalently expressed as MM

https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

Acceleration of MM: Tensor Processing Unit (TPU) (1)



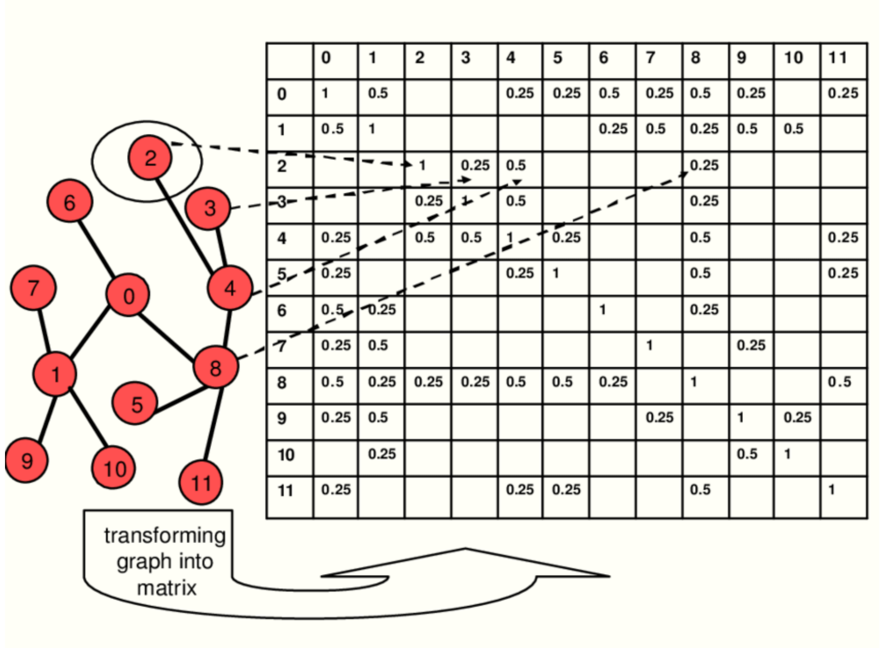
TPU Block Diagram

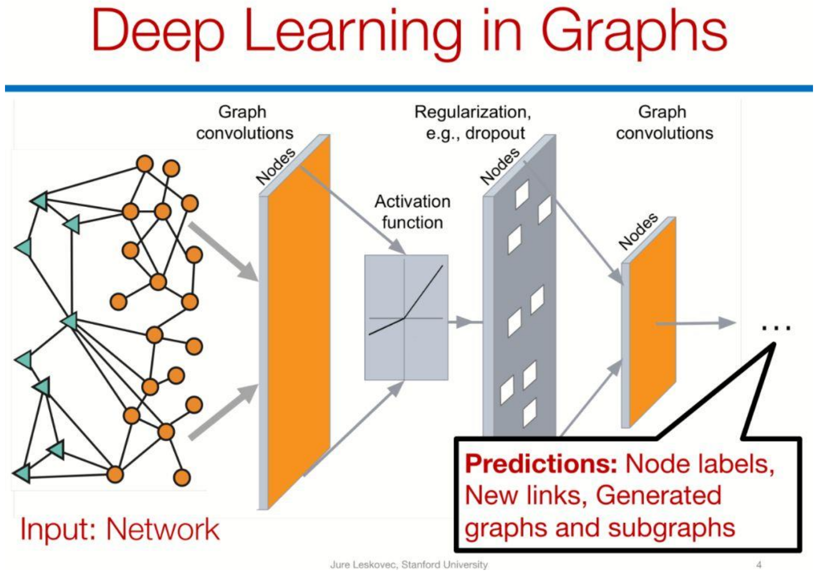
Matrix Multiplier Unit (MXU): 65,536 8-bit multiply-and-add units for matrix operations

https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

Applications of MM: Graph Analytics (1)

- Graph is represented using (usually sparse) Adjacency Matrix
- SpMV and SpMM dominates
  - For example, Page Rank – Google website search algorithm – performs iterative SpMV
    - User-graph Size: 3.5 billions nodes
  - Recommendation System – AliGraph on Taobao E-Commerce platform: information propagation (SpMM) and transformation (MM)
    - User-product graph size: 6.7 billion nodes





Chen, Peng, et al. "Finding scientific gems with Google's PageRank algorithm." *Journal of Informetrics* 1.1 (2007): 8-15.

Zhu, Rong, et al. "Aligraph: A comprehensive graph neural network platform." *arXiv preprint arXiv:1902.08730* (2019).

Acceleration of MM: Tensor Processing Unit (TPU) (2)

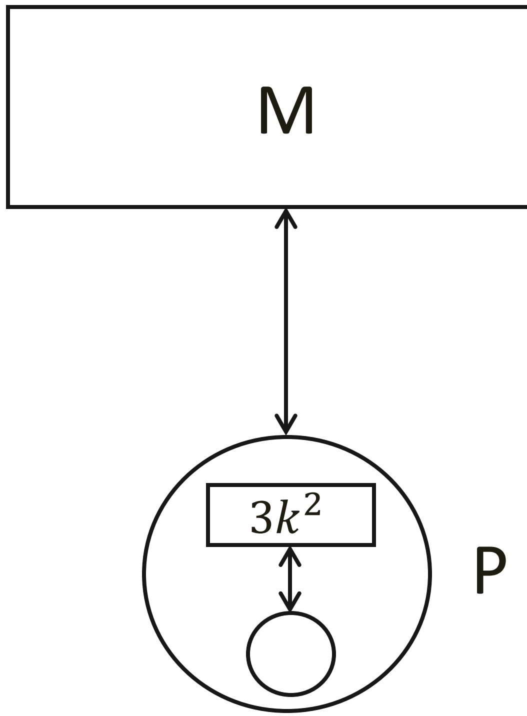
- In short, neural networks require **massive amount of multiplications and additions** between data and parameters.
- Organize these multiplications and additions as **matrix multiplication**.
- The problem TPU solves is how to **perform large matrix multiplication quickly** with less power consumption.

https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning

Backup Slides

Impact of Cache on Processor-Memory Traffic (1)

- $n \times n$ matrix multiplication $C \leftarrow A \times B$
- Cache (on chip memory) size = $3k^2 \ll n^2$
- **Processor-Memory (P-M) Traffic = Total data communicated between $\boldsymbol{P}$ and $\boldsymbol{M}$**



Impact of Cache on Processor-Memory Traffic (2)

- Suppose cache size = $O(1)$
  Straightforward matrix multiplication

Repeat for all i, j [Repeat n times [Read $A(i,k), B(k,j)$
$C(i,j) \leftarrow C(i,j) + A(i,k) * B(k,j)$]
write $C(i,j)$ in M]

Total data communicated between P and M

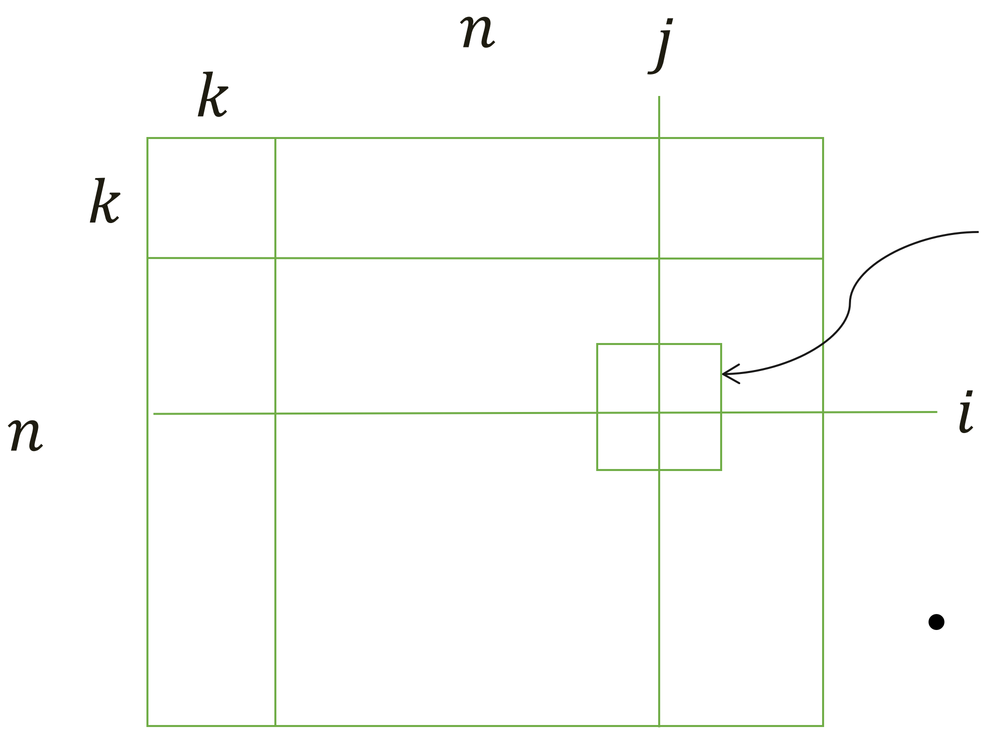
Read operation ⇢ $n^2(2 \cdot n) = 2n^3$

Also, n^2 write operations $[C(i,j)]$ // can be ignored

Data Reuse = 1

Impact of Cache on Processor-Memory Traffic (3)

Given Cache size $3k^2$, choose block size = $k \times k$



Notation:
$C^{ij} = (i, j)^{th}$ block of size $k \times k$

$$0 \leq i, j < \frac{n}{k}$$

- Total number of blocks = $\frac{n^2}{k^2}$

Impact of Cache on Processor-Memory Traffic (4)

Do $i = 0$ to $\frac{n}{k} - 1$

Do $j = 0$ to $\frac{n}{k} - 1$

Compute C^{ij}

$C^{ij} \leftarrow 0$

Do $l = 0$ to $\frac{n}{k} - 1$

Read A^{ij}, B^{lj} from M

$C^{ij} \leftarrow C^{ij} + A^{il} \otimes B^{lj}$

P-M traffic = $2 \cdot k^2$

$\otimes$ = $k \times k$ matrix multiplication ← 3 level nested loop

Total number of operations = $2 \cdot k^3$ – **Data Reuse = k**

Total number of nested loops = 6

Total number of operations performed = $\left(\frac{n}{k}\right)\left(\frac{n}{k}\right)\left(\frac{n}{k}\right) \cdot 2k^3 = 2n^3$

Impact of Cache on Processor-Memory Traffic (5)

$$\text{Total data communicated} = \left(\frac{n}{k}\right)\left(\frac{n}{k}\right)\left(\frac{n}{k}\right) \cdot 2k^2$$

$$= \frac{2n^3}{k}$$

Traffic with
Cache = $O(1)$

$$= \frac{2n^3}{\sqrt{k^2}}$$

$$\propto \frac{\text{Total data communicated in naive MM}}{\sqrt{Cache\ Size}}$$

Impact of Cache on Processor-Memory Traffic (6)

Check

- $k = 1$: naive MM $\quad P\text{-}M$ traffic $\propto n^3$
- $k = n$: read complete A, B matrices once $\quad P\text{-}M$ traffic $\propto n^3 \left(\frac{n^3}{\sqrt{n^2}}\right)$

- On a desktop platform, can improve performance by 100X!
- Idea applicable to p process parallel machine (shared memory, message passing)