CSDS 451: Designing High Performant Systems for AI

Lecture 7

9/16/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Parallel Program Analysis (Quick Review)
- Data Parallel Algorithms
  - Sorting
- Task Parallelism

Outline

- Parallel Program Analysis (Quick Review)
- Data Parallel Algorithms
  - Sorting
- Task Parallelism

Speedup

- Improvement in time due to parallelization

$$\text{Speedup} = \frac{\text{Serial time (on a uniprocessor system)}}{\text{Time after parallelization}}$$

- Can be empirical – gives us actual speedup of implementation
- Can be in order notation – helps us in analyze whether a particular parallel algorithm is good - scalable or not

Scalability

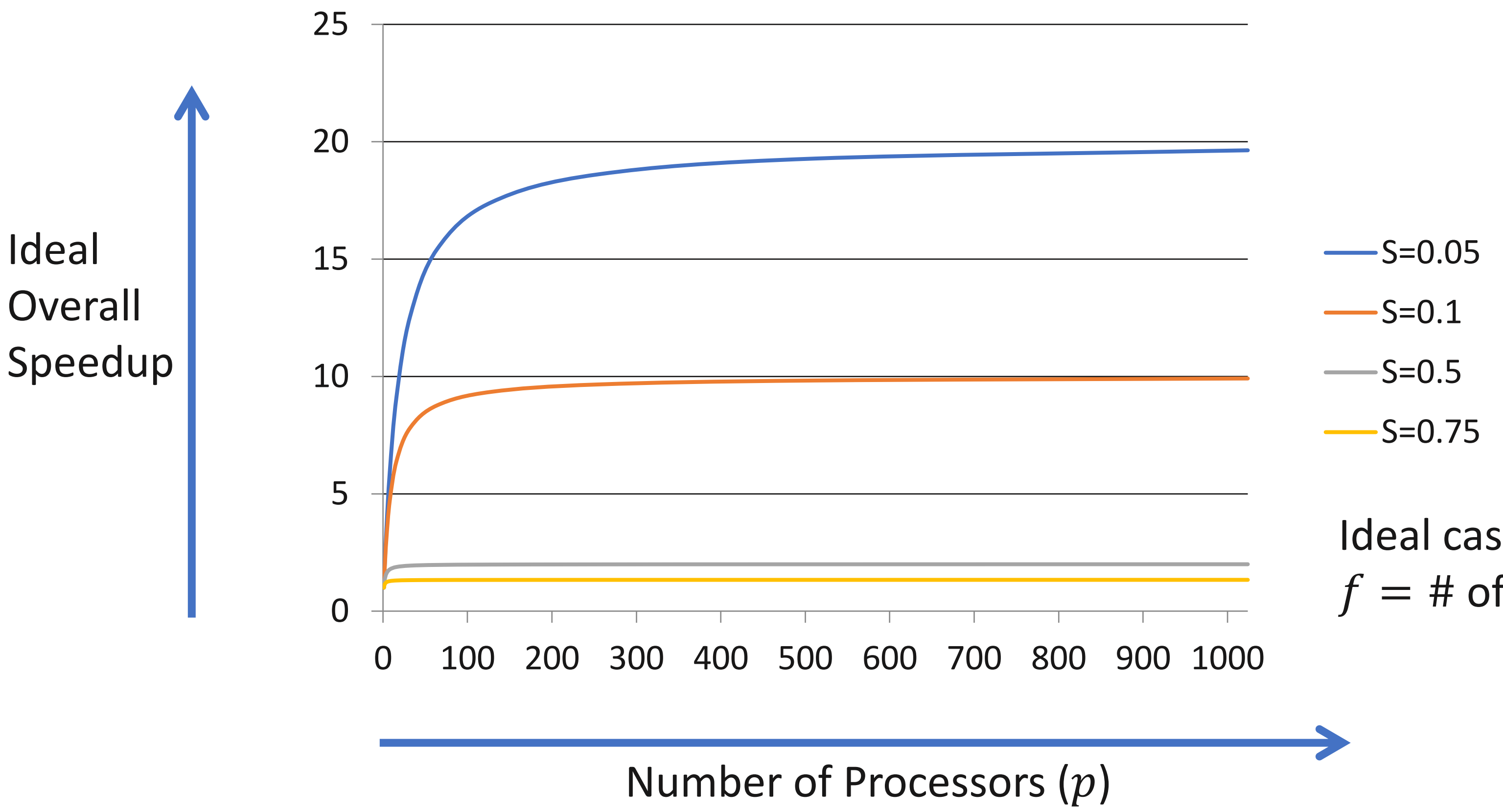
$$\text{Speedup} = \frac{\text{Serial time (on a uniprocessor system)}}{\text{Parallel time using } p \text{ processors}}$$

If speedup = $O(p)$, then it is a **scalable** solution

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Amdahl's Law

$$\text{Overall Speedup} = \frac{1}{S + P/f} = \frac{1}{S + (1-S)/f} \rightarrow \frac{1}{S}$$

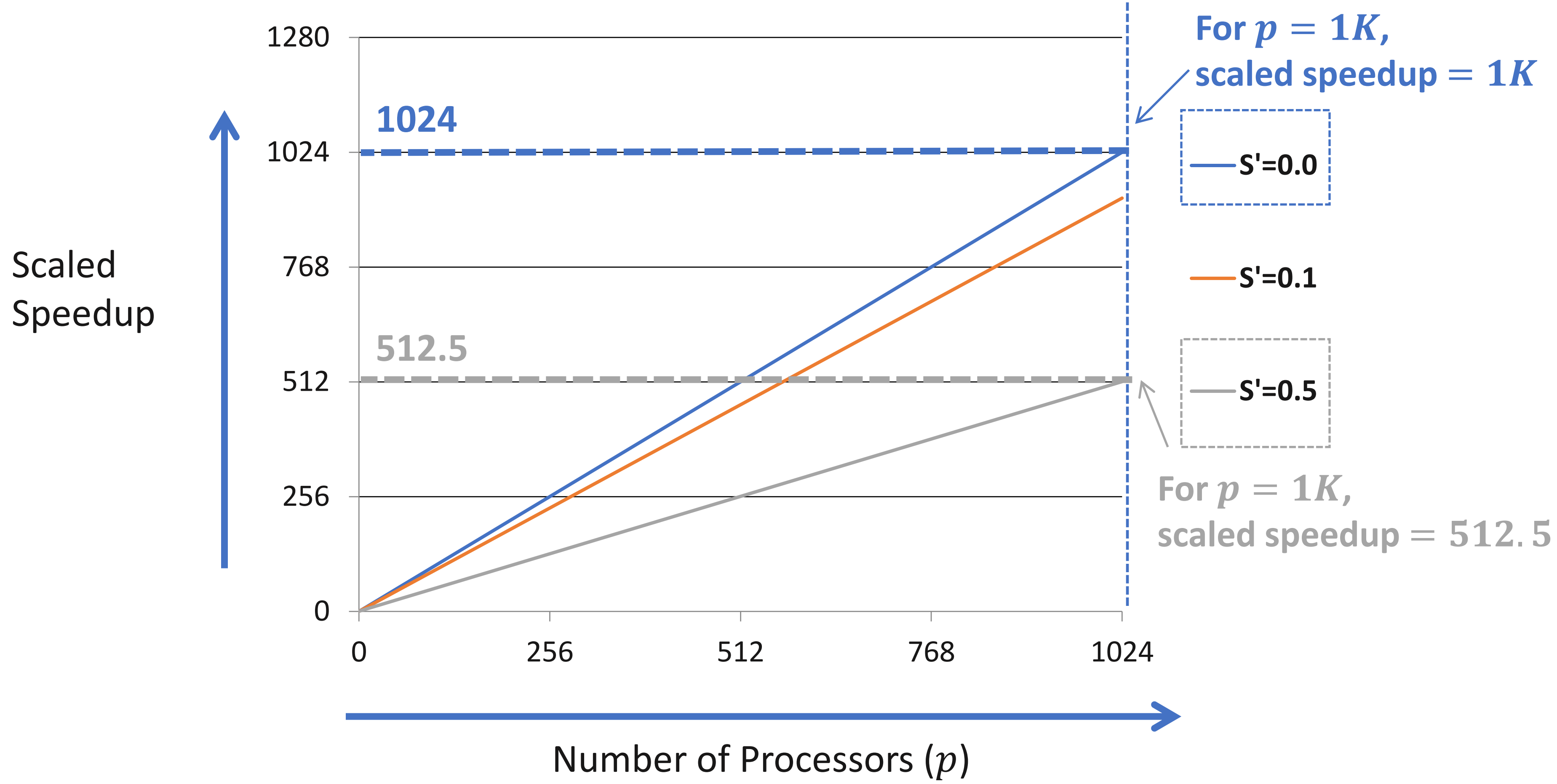


Ideal case
f = # of processors

Scaled Speedup (Gustafson's Law) (1)

- Amdahl's Law — Serial portion of code limits performance
  (As we use more processors)
- As we use more processors
  → we use more data
  → e.g., more fine grained model
- E.g. Processing $N \times N$ image
  Using p processors
  As we increase p we usually increase image size

Scaled Speedup (Gustafson’s Law) (2)



Strong or Weak Scaling

- Strong Scaling
  - Governed by Amdahl's Law
  - The number of processors is **increased** while the problem size **remains constant**
  - Results in a **reduced** workload per processor

- Weak Scaling
  - Governed by Gustafson's Law
  - **Both** the number of processors and the problem size are **increased**
  - Results in a **constant** workload per processor

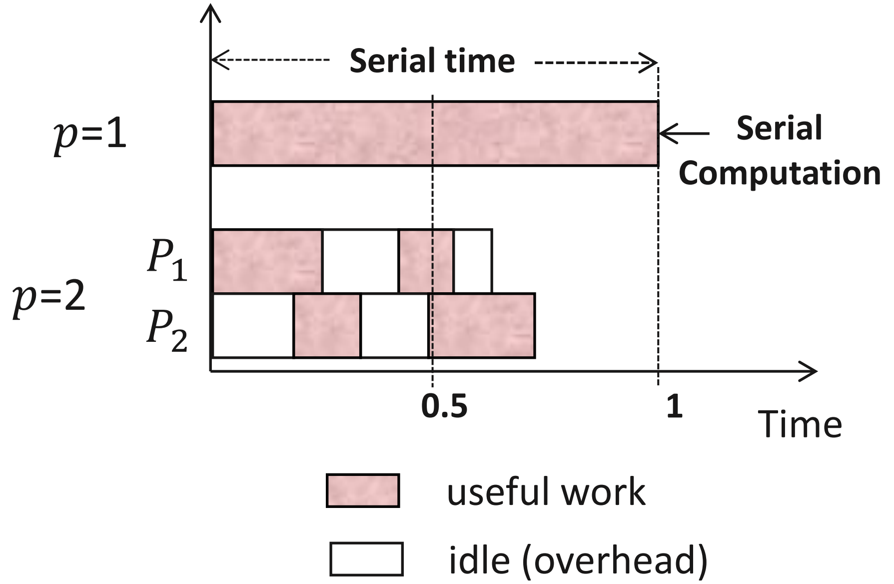
Performance (1)

Efficiency

- Question: If we use p processors, is speedup = p ?

- Efficiency $\triangleq$ Fraction of time a processor is usefully employed during the computation

Typical execution of a program on a parallel machine

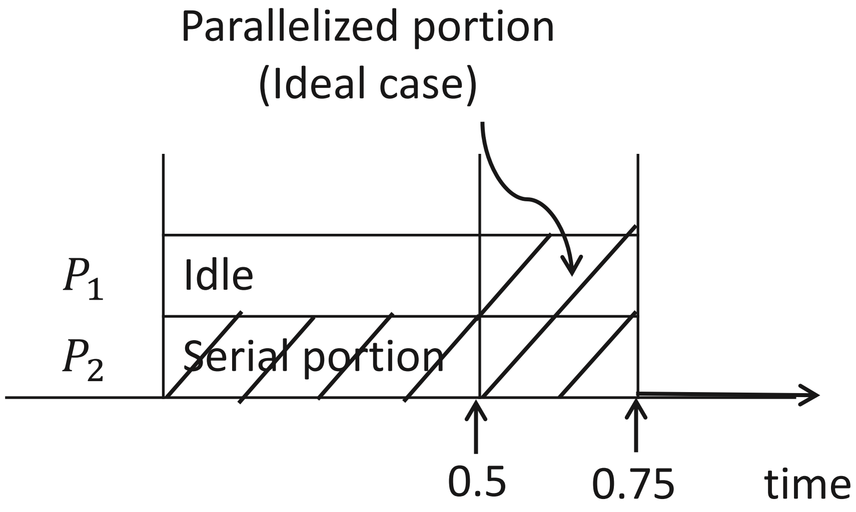


- $E =$ Speedup / # of processors used
  - E is the average efficiency over all the processors
  - Efficiency of each processor can be different from the average value

Performance (2)

Ex. $S = 0.5$
$P = 0.5$

2 processor system



$$\text{Speedup} = \frac{1}{0.75} = {}^{4}/_{3}$$

$$\text{(Average) Efficiency} = \frac{{}^{4}/_{3}}{2} = {}^{2}/_{3}$$

$$\text{Efficiency of } P_1 = \frac{0.25}{0.75} = {}^{1}/_{3}$$

$$\text{Efficiency of } P_2 = \frac{0.75}{0.75} = 1$$

$$\text{(Average) Efficiency} = \frac{{}^{1}/_{3}+1}{2} = 2/3$$

Performance (3)

- Cost = Total amount of work done by a parallel system

  = Parallel Execution Time x Number of Processors

  = $T_p \times p$

- Cost is also called Processor Time Product

- COST OPTIMAL (or WORK OPTIMAL) Parallel Algorithm
  - Total work done = Serial Complexity of the problem

Blocked Matrix Multiplication

- Serial Algorithm - $O(n^3)$

- Parallel Algorithm:
- Number of steps n/b
- Work done by each processor in each step: $O(b)$

Total Number of Processors

- Cost of the algorithm: $O\left(n^2 \times \left[\frac{n}{b} \times b\right]\right) = O(n^2 \times n) = O(n^3)$

Work done in each processor

- Cost of the algorithm ~ serial complexity. So work optimal.

Sum of Elements in an Array

- Serial Algorithm: $O(p)$

- Parallel Algorithm:
- Number of steps - $O(\log_2 p)$
- Number of processors in each step - $O(p)$
  - Note: We can also consider only active processors to come up with a different analysis.

- Cost of the Algorithm - $O(p \log_2 p)$

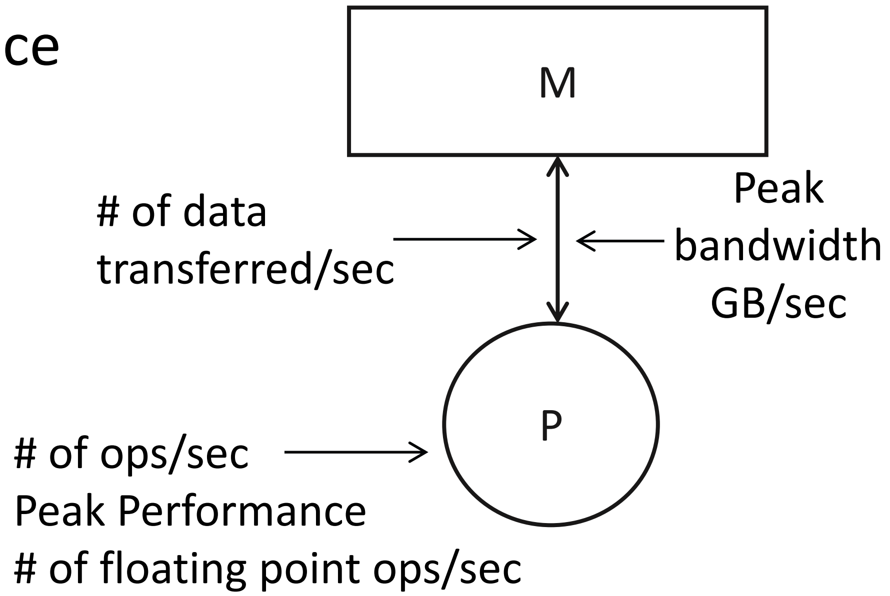
- Cost of the algorithm !~ serial complexity. So not work optimal (assuming GPU model). It is work optimal if we only consider active processor in the analysis.

Roofline Model

- **In Practice:** Processor Performance ≫ Memory Performance

- Processor may idle waiting for data

$$\text{Machine Balance} = \frac{Peak\ Performance\ (Flops/sec)}{Peak\ Bandwidth\ (GB/sec)}$$



The Roofline Model
Samuel Williams, Lawrence Berkeley National Lab
https://escholarship.org/content/qt0qs3s709/qt0qs3s709.pdf

Roofline Model

- Arithmetic Intensity (AI) $=$ For every fetched data (from memory) how many ops make use of them

$$= \frac{\#\ of\ ops\ performed}{Amount\ of\ data\ moved\ from\ external\ memory}$$

- Algorithm Dependent

- Example

  AI of vector inner product $= \frac{2n}{2n} = O(1)$

  AI of Dense matrix multiplication $= \frac{2n^3}{2n^2} = O(n)$

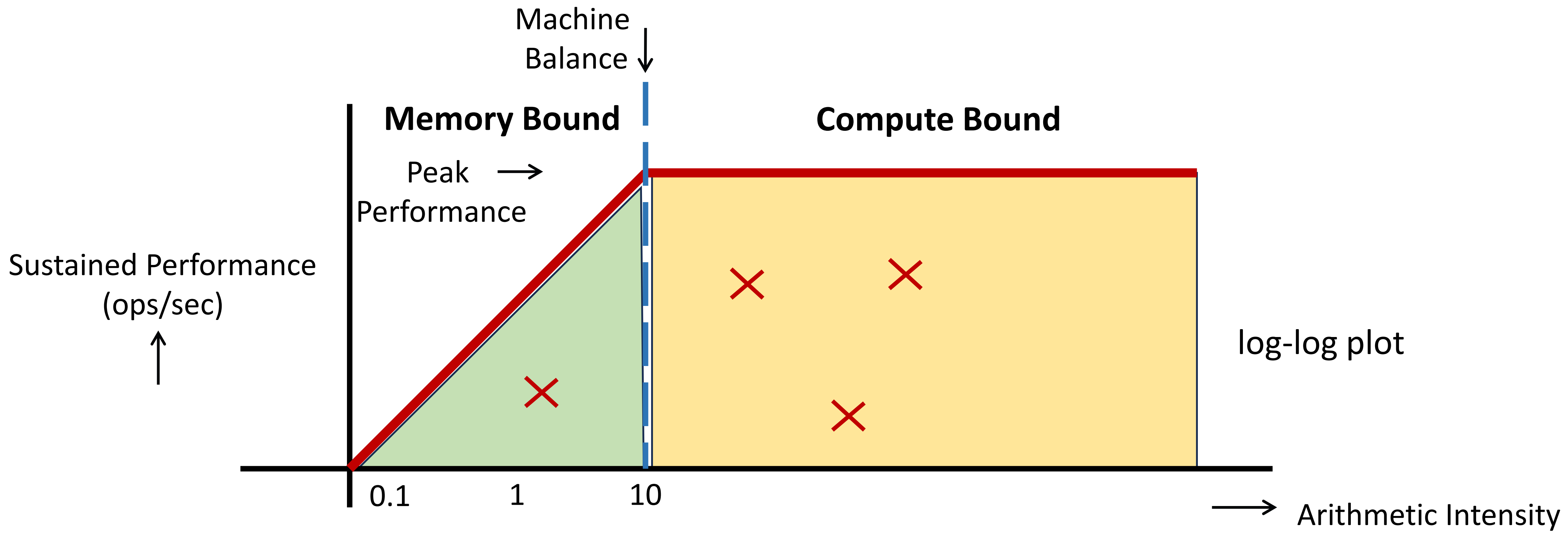
Roofline Model

- Understand system bottlenecks – Computation vs Memory
- High level model using few parameters (ignores ISA etc.)
- Can be used to understand and evaluate an application performance on a target
- Can be used to make architectural decisions for custom accelerators

The Roofline Model
Samuel Williams, Lawrence Berkeley National Lab
https://escholarship.org/content/qt0qs3s709/qt0qs3s709.pdf

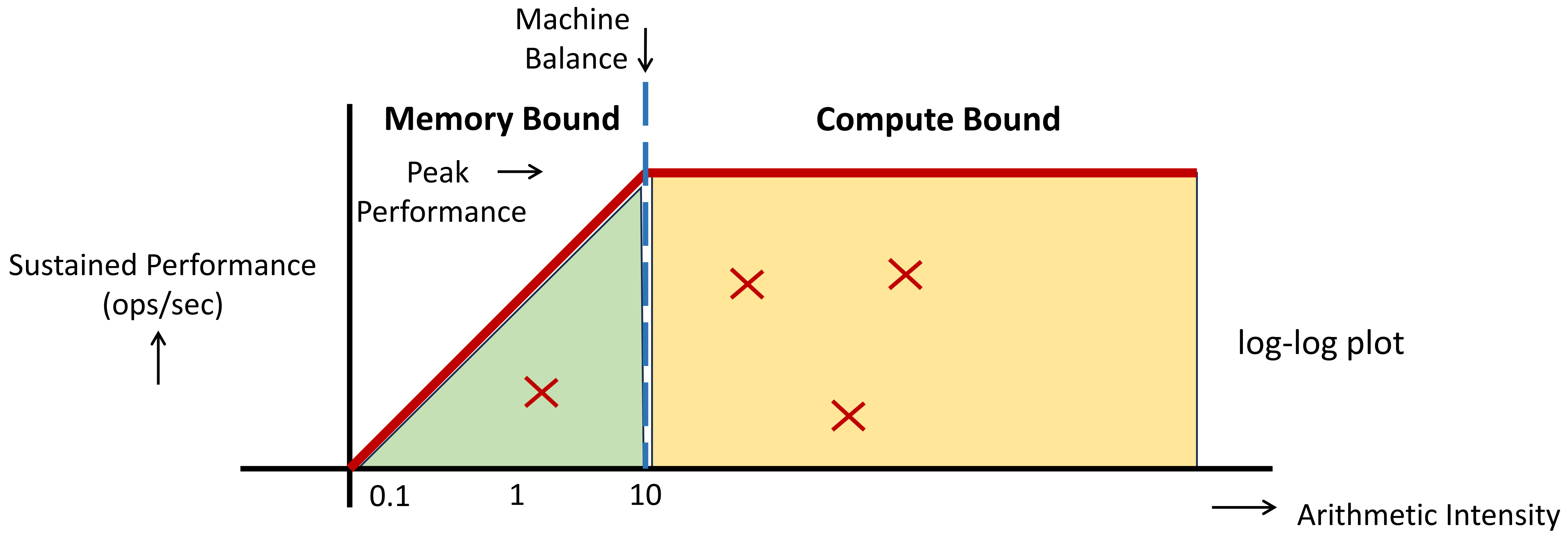
Compute Bound vs Memory Bound

Roofline Model (for a given platform)



Compute Bound vs Memory Bound

Roofline Model (for a given platform)



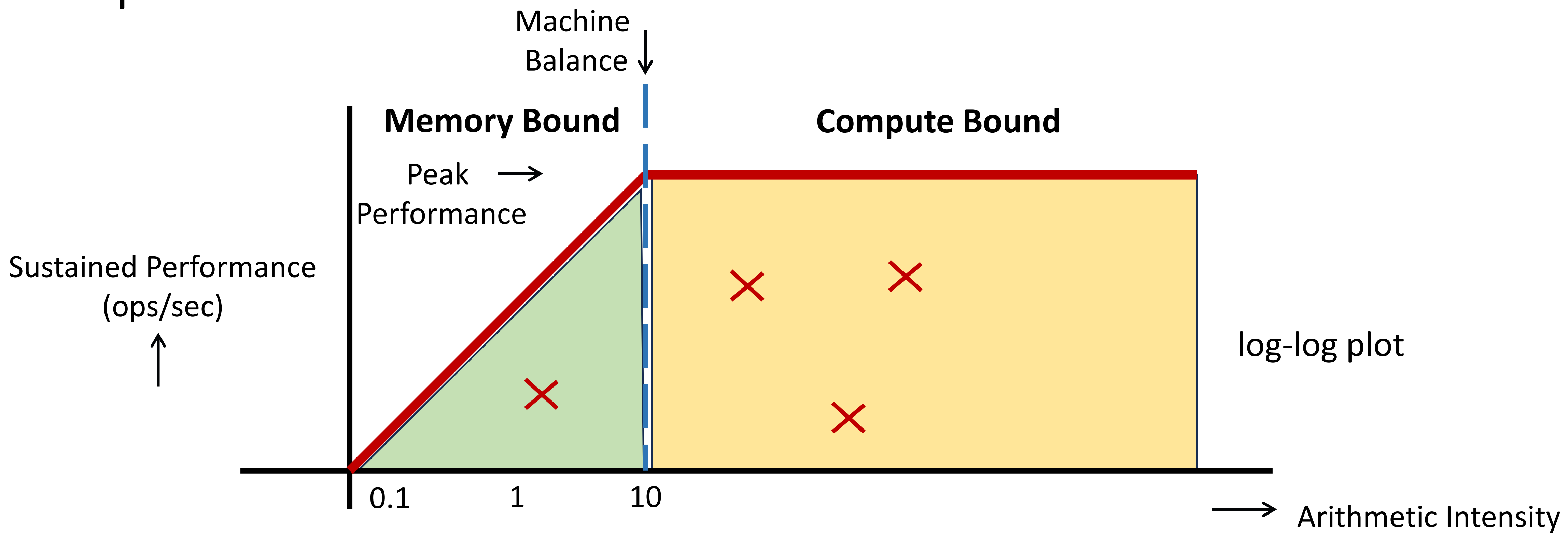
- Bold Red line denotes the maximum achievable performance for a given arithmetic intensity
- Memory bound region: maximum achievable performance bounded by memory bandwidth
- Compute bound region: maximum achievable performance bounded by compute capability

Compute Bound vs Memory Bound Roofline Model (for a given platform)

- Compute Bound: Maximum Performance is limited by available compute resources
  - if we add more compute resources, the performance may improve (for a given bandwidth)
  - Useful in designing accelerators. Not a focus of this class

- Memory Bound: Maximum Performance is limited by memory bandwidth
  - Reorder computations to more efficiently utilize bandwidth
  - In other words, improve arithmetic intensity to move right on x axis, which also improves y axis

Roofline Model – What about actual performance?

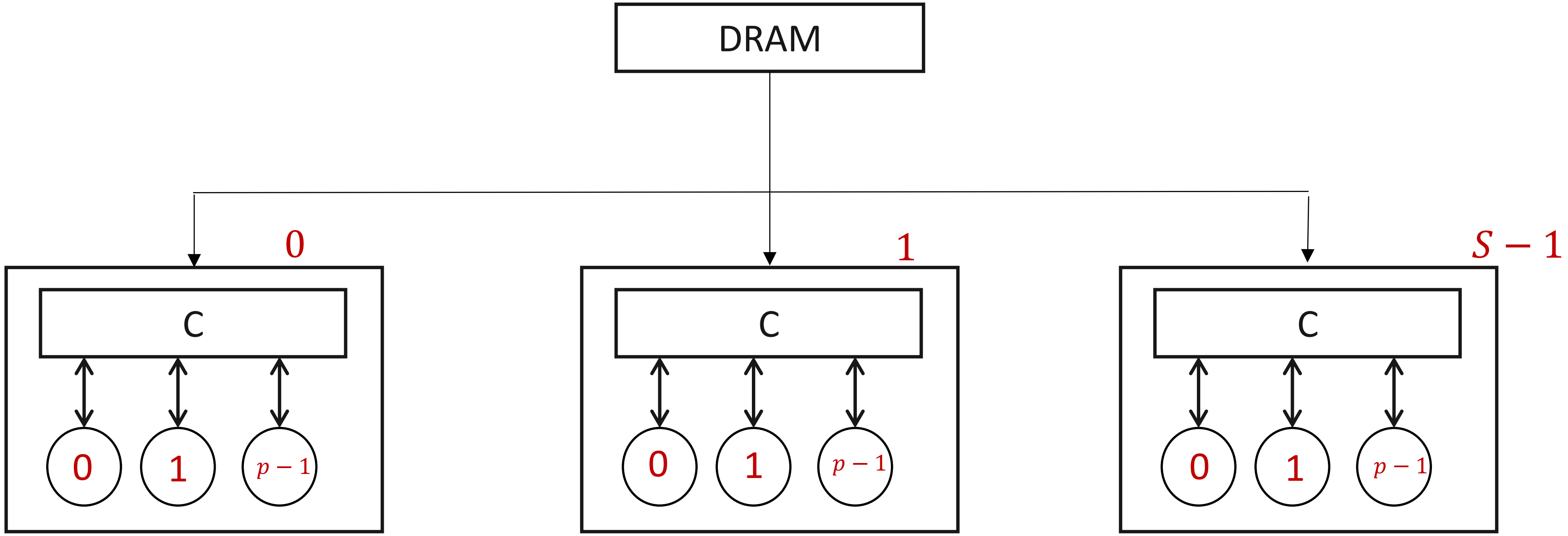


- For a fixed AI: improving performance (going up) requires better parallelism – Data and Task Parallel Techniques
- Once the Red bold line is reached
  - Green Region: Write better algorithm to improve arithmetic intensity
  - Yellow Region: No algorithmic solution, need to add more computation power

Outline

- Parallel Program Analysis (Quick Review)
- Data Parallel Algorithms
  - Sorting
- Task Parallelism

Modeling GPU Architectures



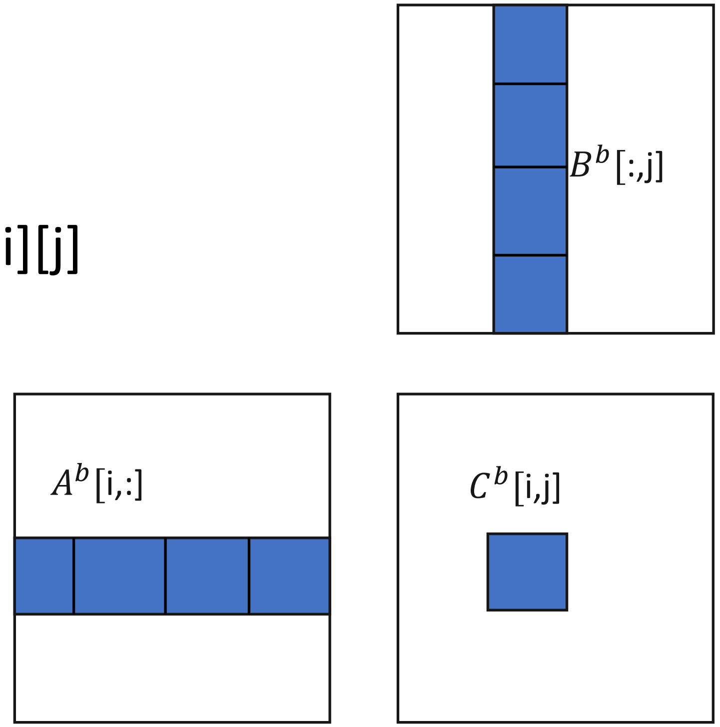
- Number of Blocks can be (in fact should be) greater than the number of SMPs
- Number of threads per block need not be equal to the number of compute cores per SMP
  - It is good to have it as a multiple though
- We will discuss these considerations when we discuss GPU programming later

- For now, assume the simple model above.
  - S Blocks
  - p Threads per block

Block Matrix Multiplication

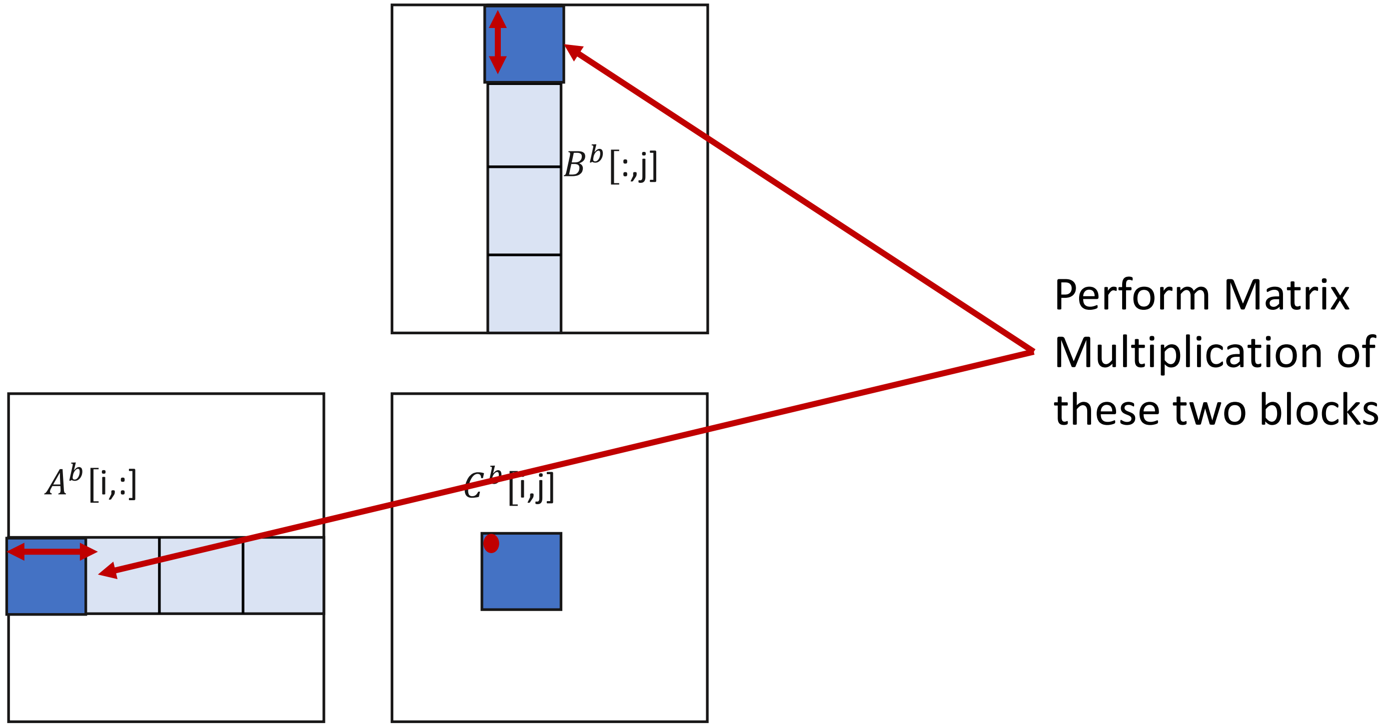
- Key Idea: Partition matrices A/B/C into $b \times b$ size blocks
- Blocks denoted as $A^b[1{:}\frac{N}{b}][1{:}\frac{N}{b}]$/ $B^b[1{:}\frac{N}{b}][1{:}\frac{N}{b}]$/ $C^b[1{:}\frac{N}{b}][1{:}\frac{N}{b}]$

- $C^b[i][j]$ = BlockedMM($A^b[i][:]$, $B^b[:][j]$)
- For $k = 0$ to N/b
  - Fetch $A^b[i][k]$, $B^b[k][j]$
  - Matrix Multiply $A^b[i][k]$, $B^b[k][j]$ and Accumulate $C^b[i][j]$



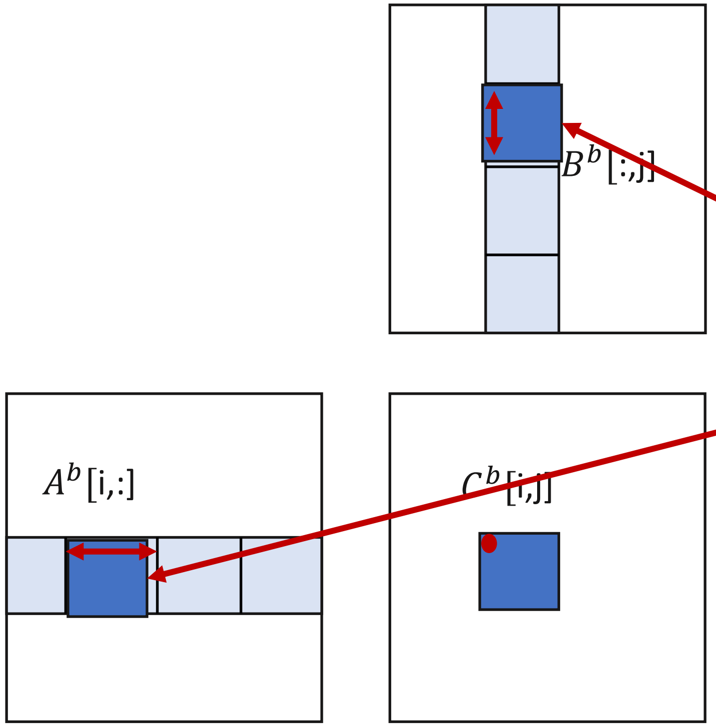
Block Matrix Multiplication

SMP: BID_i, BID_j



Block Matrix Multiplication

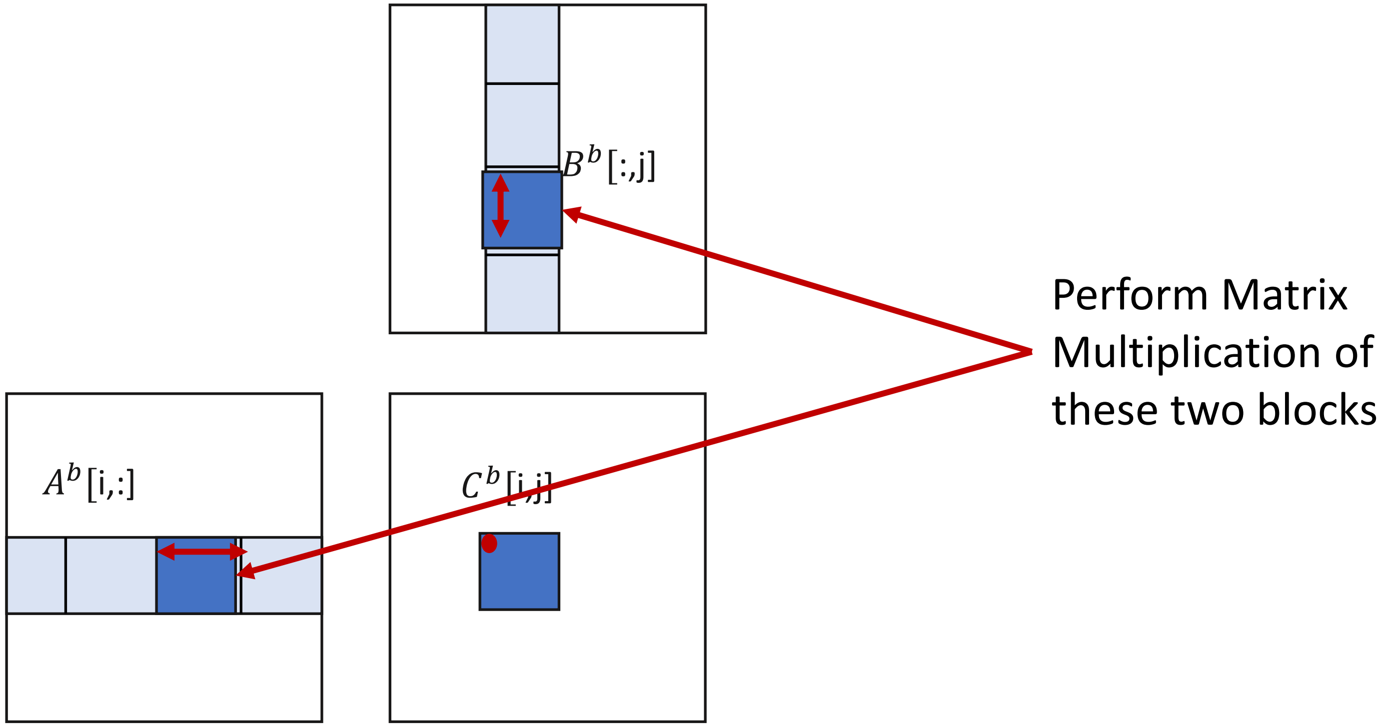
SMP: BID_i, BID_j



Perform Matrix Multiplication of these two blocks

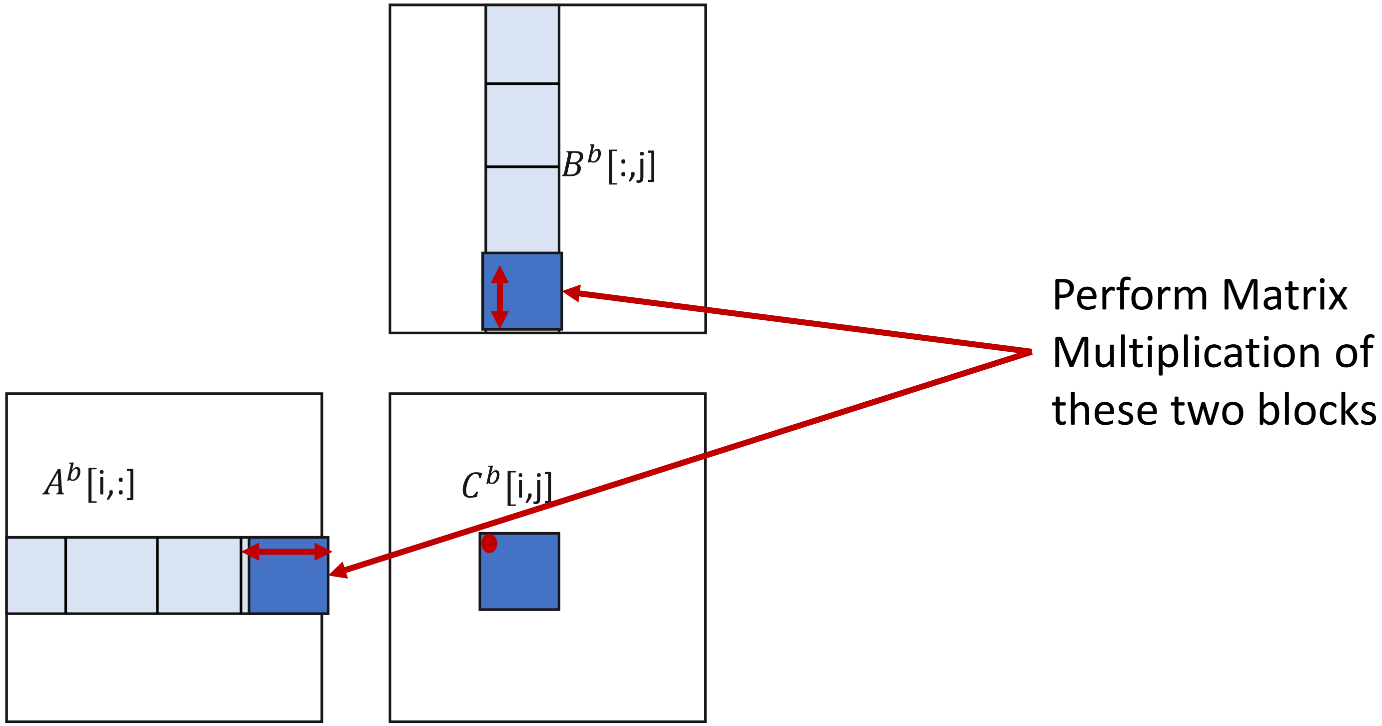
Block Matrix Multiplication

SMP: BID_i, BID_j



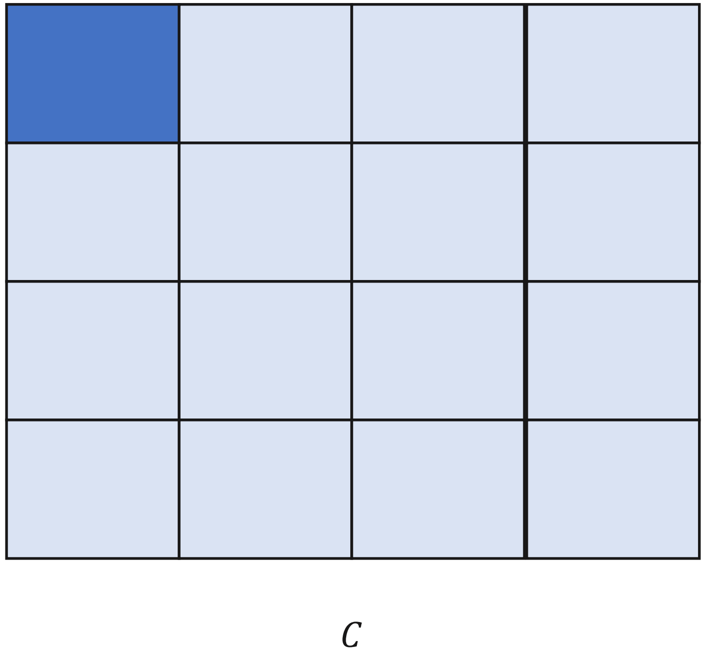
Block Matrix Multiplication

SMP: BID_i, BID_j



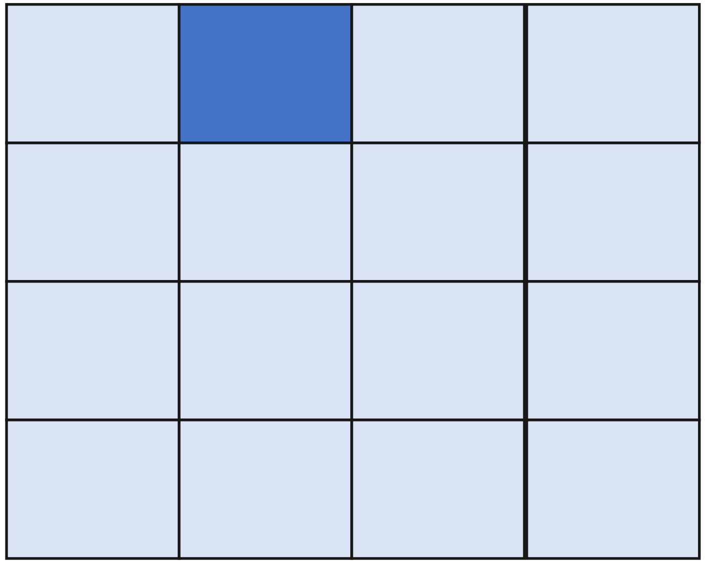
Block Matrix Multiplication

SMP: $0{,}0$



Block Matrix Multiplication

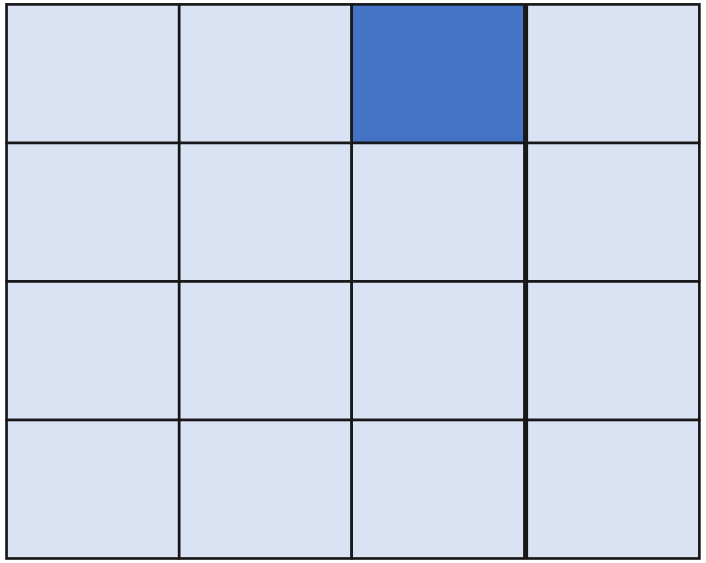
SMP: 0,1



C

Block Matrix Multiplication

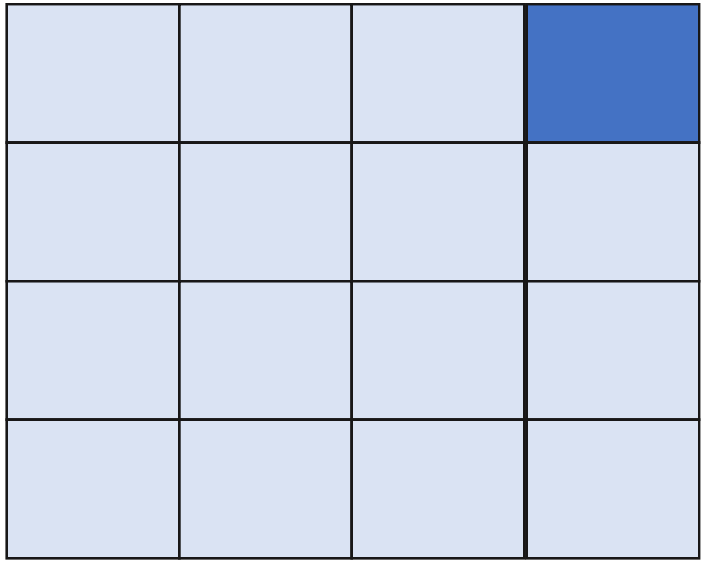
SMP: 0,2



C

Block Matrix Multiplication

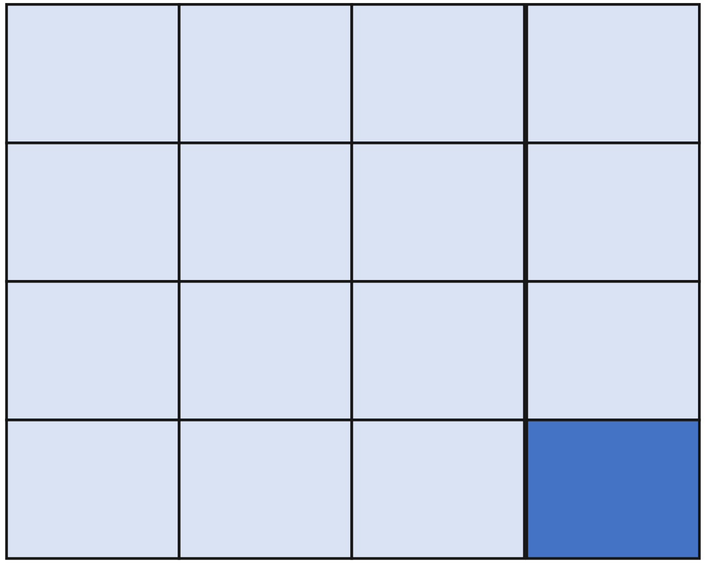
SMP: 0,3



C

Block Matrix Multiplication

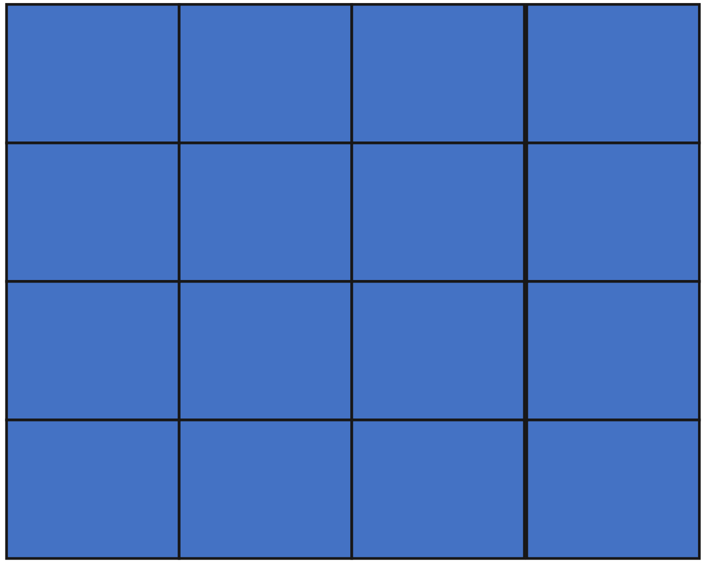
SMP: ? , ?



C

Block Matrix Multiplication

SMP: 0,0
SMP: 0,1
SMP: 0,2
SMP: 0,3
SMP: 1,0
SMP: 1,1
SMP: 1,2
SMP: 1,3
SMP: 2,0
SMP: 2,1
SMP: 2,2
SMP: 2,3
SMP: 3,0
SMP: 3,1
SMP: 3,2
SMP: 3,3



C

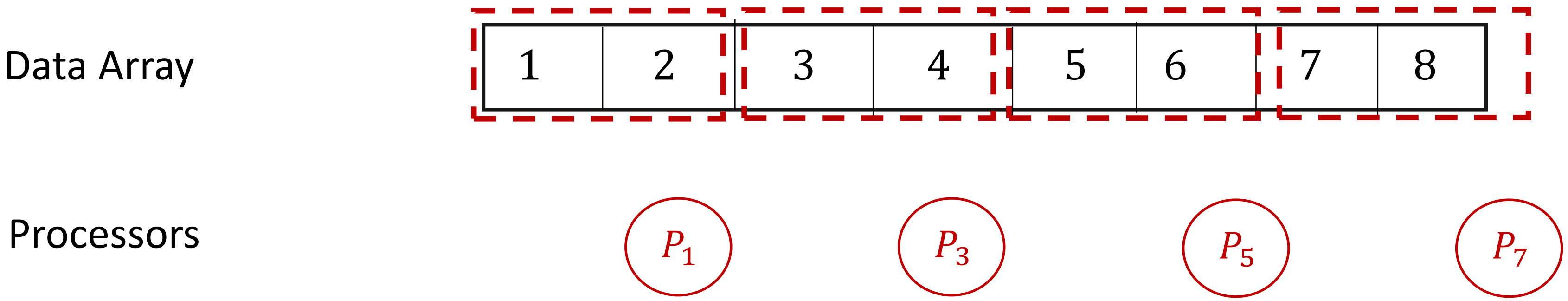
In parallel produce all the blocks of C

- No conflicts. Each SMP can work independently
- In practice, number of blocks can be greater than # SMPs available on the hardware
  - Hardware performs time multiplexing, hidden from programmer,
  - We will discuss in our HIP programming class

Sum of an Array

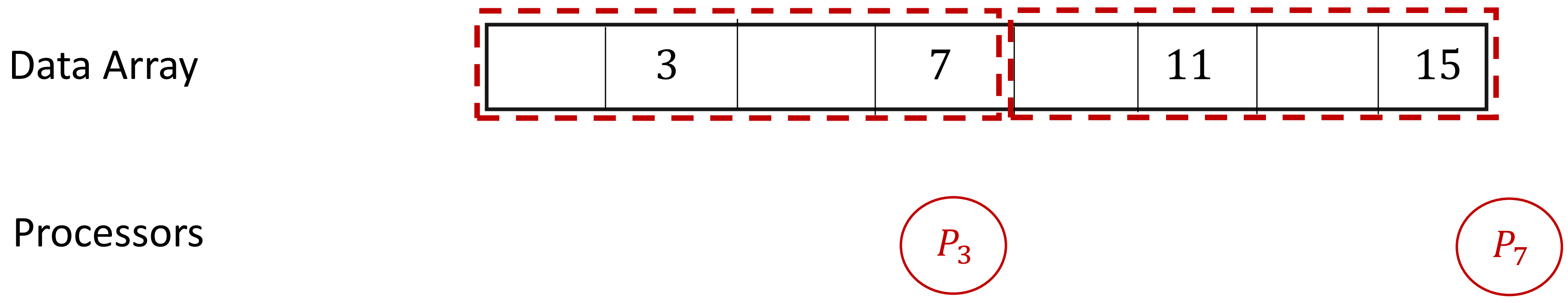
- Block MM – Output Partitioning

- Sum of an Array – Input Partitioning



Sum of an Array

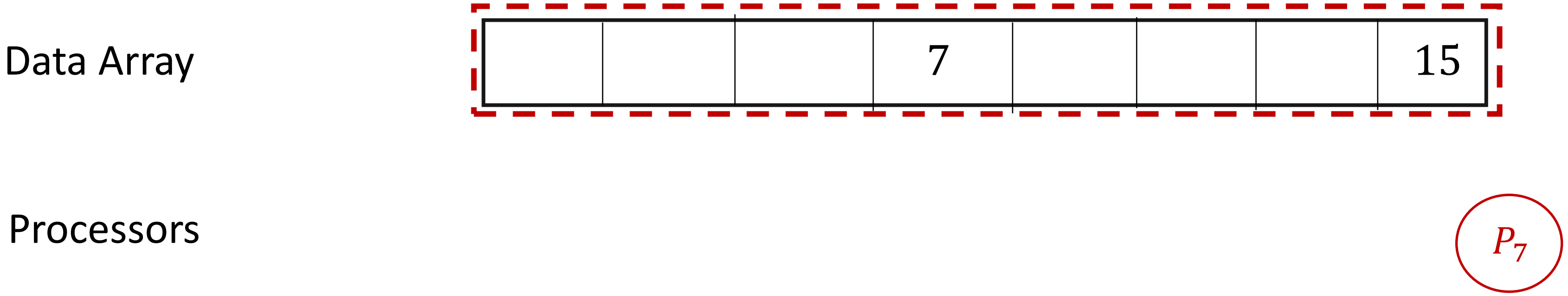
- Block MM – Output Partitioning
- Sum of an Array – Input Partitioning



Sum of an Array

- Block MM – Output Partitioning

- Sum of an Array – Input Partitioning



Parallel Sorting

- Sort a list of elements in ascending (or descending) order

- Input: [8,7,6,5,4,3,2,1]
- Sorting Order: Ascending
- Output: [1,2,3,4,5,6,7,8]

Parallel Sorting

Input:

8	7	6	5	4	3	2	1

Output:

1	2	3	4	5	6	7	8

Can I do conflict-free output partitioning based data parallelism?

Parallel Sorting

Input:

8	7	6	5	4	3	2	1

Output:

1	2	3	4	5	6	7	8

Can I do conflict-free output partitioning based data parallelism?

Maybe, for example if we do bucket or radix sort

Parallel Sorting

Input:

8	7	6	5	4	3	2	1

Output:

1	2	3	4	5	6	7	8

Can I do conflict-free output partitioning based data parallelism?
Think about how that could work. (May include as a question in Exam)

Parallel Sorting

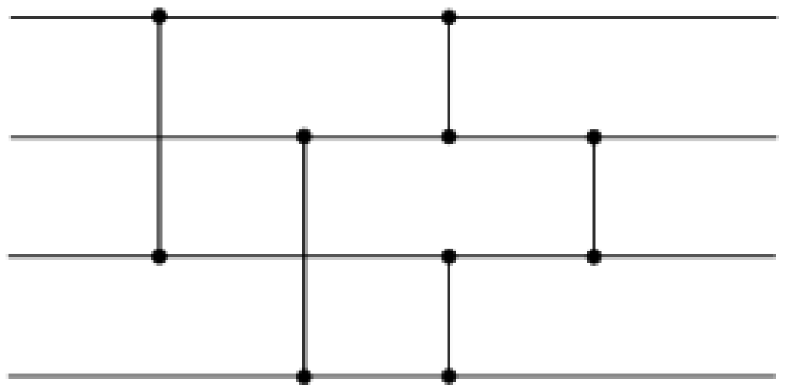
Input:	8	7	6	5	4	3	2	1
Output:	1	2	3	4	5	6	7	8

Can I do conflict-free output partitioning based data parallelism?

We will discuss an input partitioning based approach in this lecture with a focus on comparison based sort

Parallel Sort

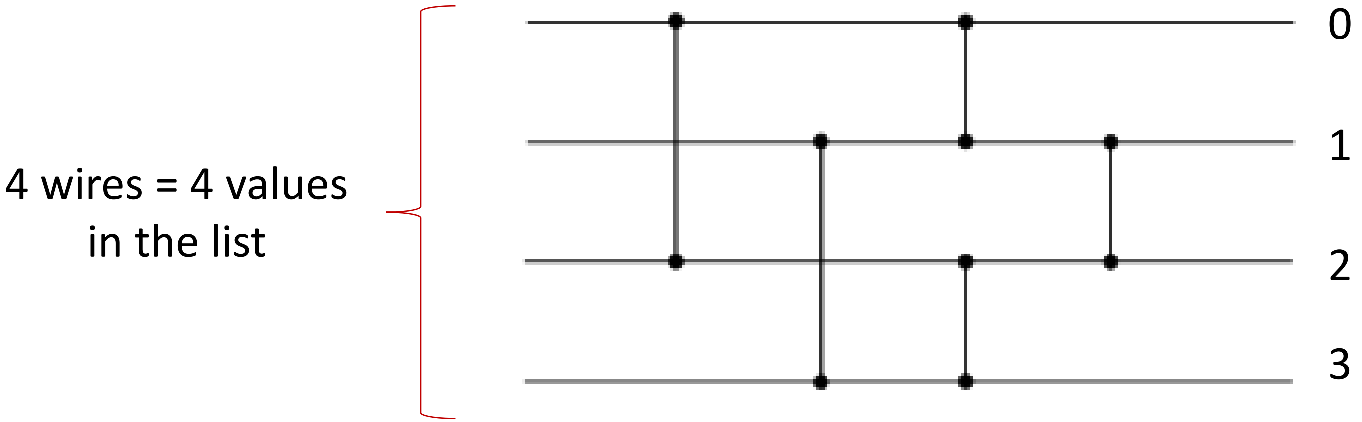
- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order



https://en.wikipedia.org/wiki/Sorting_network

Parallel Sort

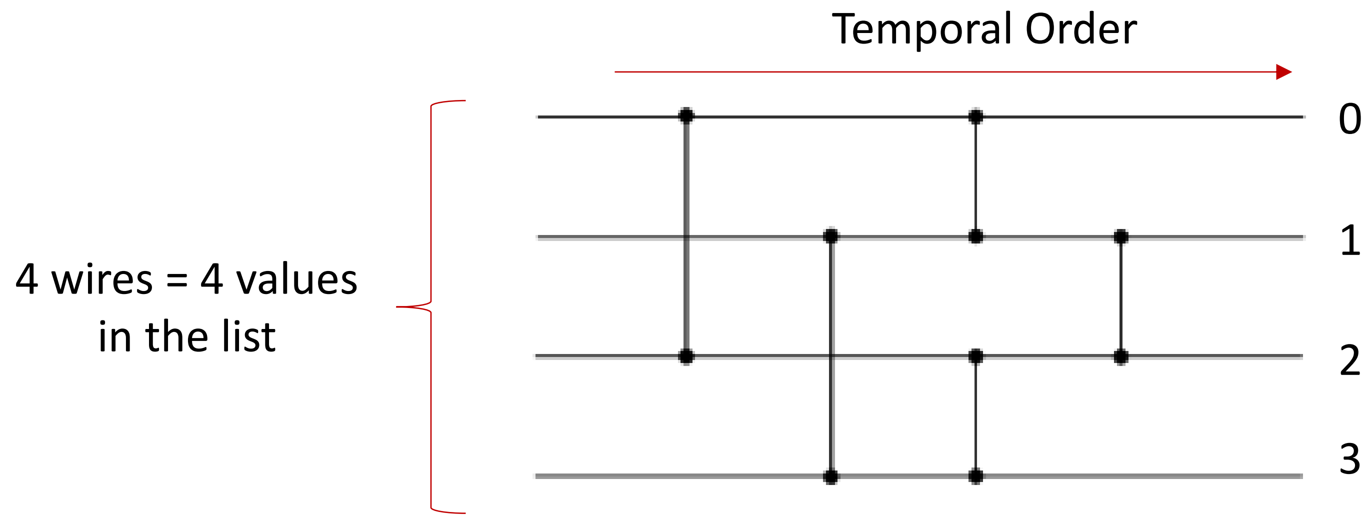
- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order



https://en.wikipedia.org/wiki/Sorting_network

Parallel Sort

- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order



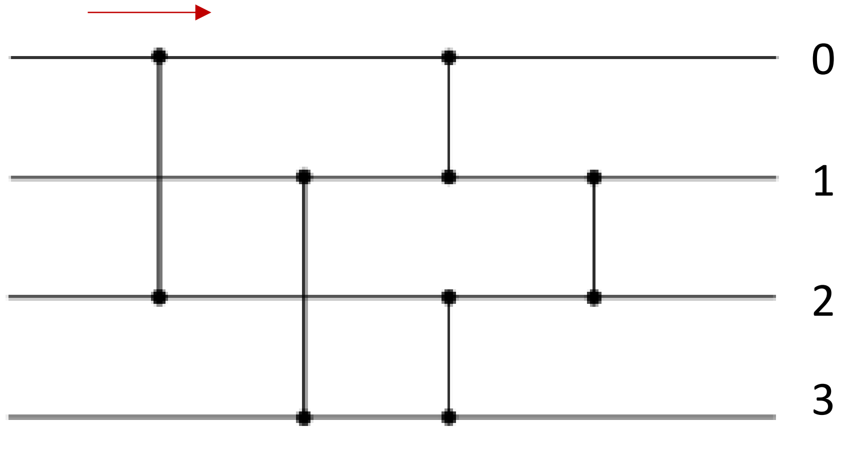
https://en.wikipedia.org/wiki/Sorting_network

Parallel Sort

- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order

Swap 0 and 2 if not in order

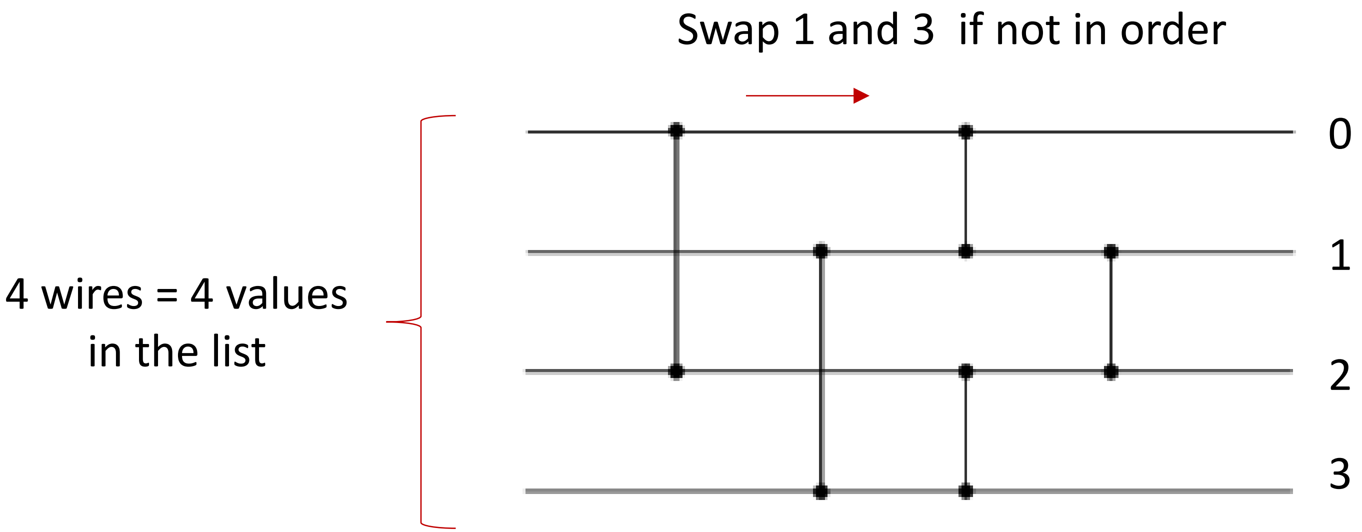
4 wires = 4 values in the list



https://en.wikipedia.org/wiki/Sorting_network

Parallel Sort

- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order

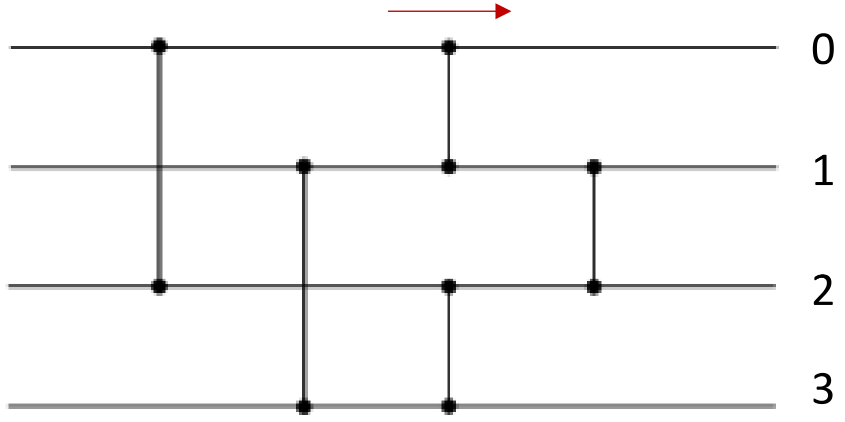


https://en.wikipedia.org/wiki/Sorting_network

Parallel Sort

- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order

4 wires = 4 values in the list



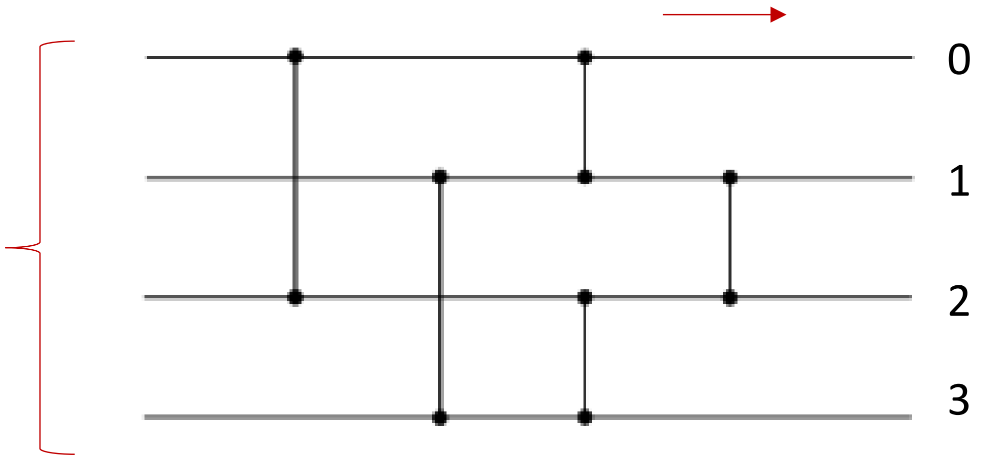
Swap 0 and 1, 2 and3 if not in order (note: updated values used)

https://en.wikipedia.org/wiki/Sorting_network

Parallel Sort

- Definition: Sorting Networks
  - An abstraction built of a fixed number of “wires” that carry value, and comparators between pairs of “wires” that swap values if they are not in the desired order

4 wires = 4 values in the list



Swap 2 and 3 if not in order (note: again updated values used)

https://en.wikipedia.org/wiki/Sorting_network

Parallel Sorting

- Key Idea: Build a sorting network to visualize the computations
- Use the visualization to understand parallelism and design a parallel program
- A number of sorting algorithms exist: bitonic sort, odd-even sort, ….
  - Bitton, Dina, et al. "A taxonomy of parallel sorting." *ACM Computing Surveys (CSUR)* 16.3 (1984): 287-318.
  - Singh, Dhirendra Pratap, Ishan Joshi, and Jaytrilok Choudhary. "Survey of GPU based sorting algorithms." *International Journal of Parallel Programming* 46.6 (2018): 1017-1034.

Parallel Sorting

- We will discuss Odd-Even Sort
- Source: https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-46-improved-gpu-sorting

- (Note the page uses an old GPU programming model)

Parallel Sorting

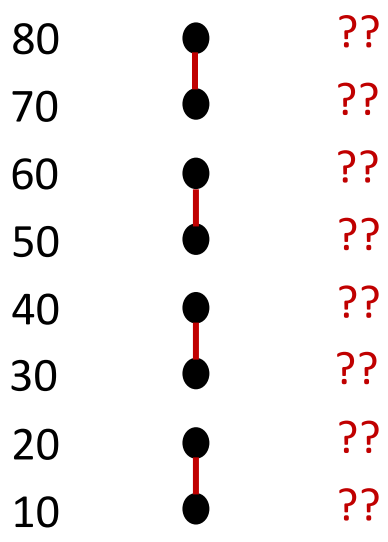
- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

- Two types of iterations
  - Odd: Compare pairs starting at odd index – (1,2); (3,4); (5,6)
  - Even: Compare pairs starting at even index – (0,1); (2,3); (4,5)

Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

Input Array

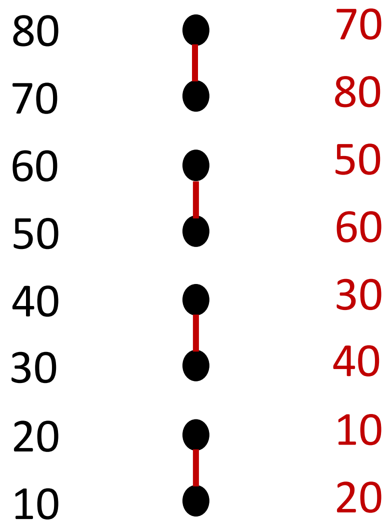


Iteration 1
(even)

Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

Input Array

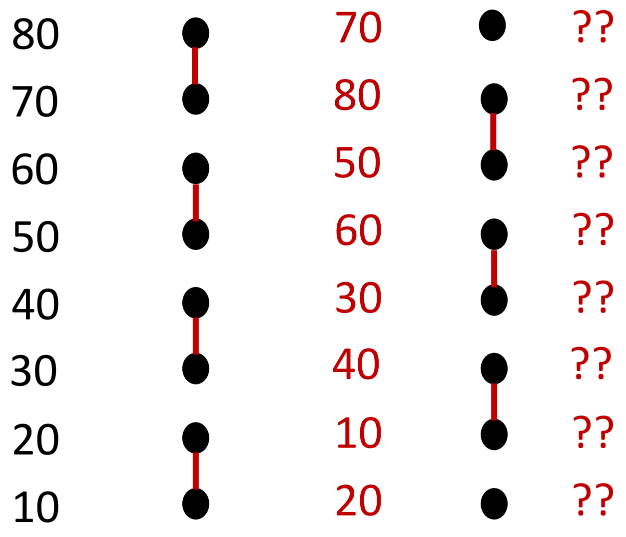


Iteration 1
(even)

Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

Input Array

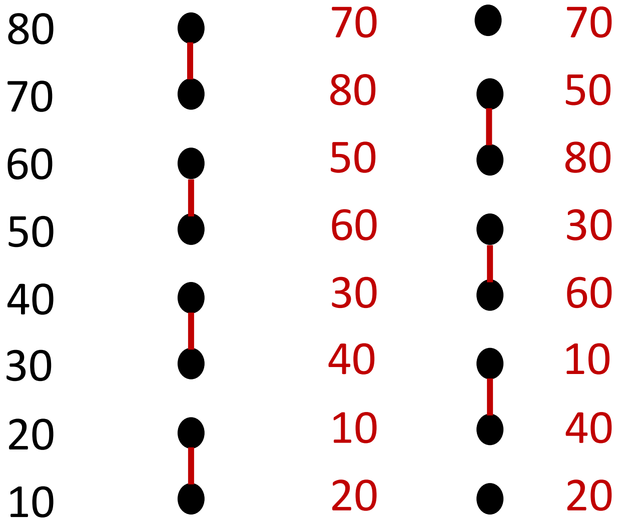


Iteration 2
(odd)

Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

Input Array

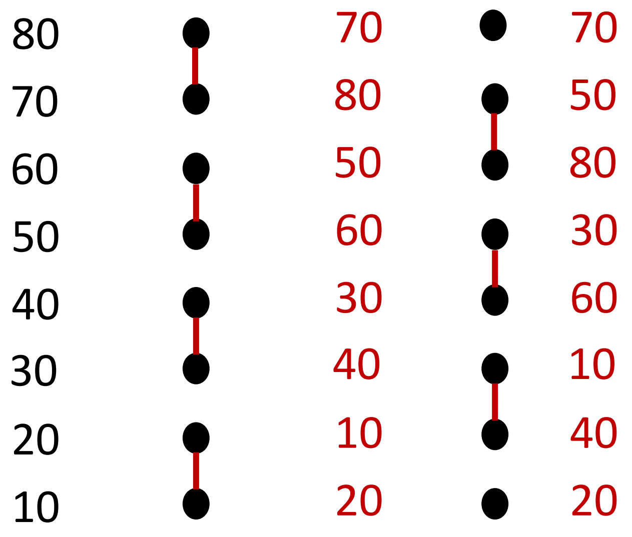


Iteration 2
(odd)

Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

Input Array



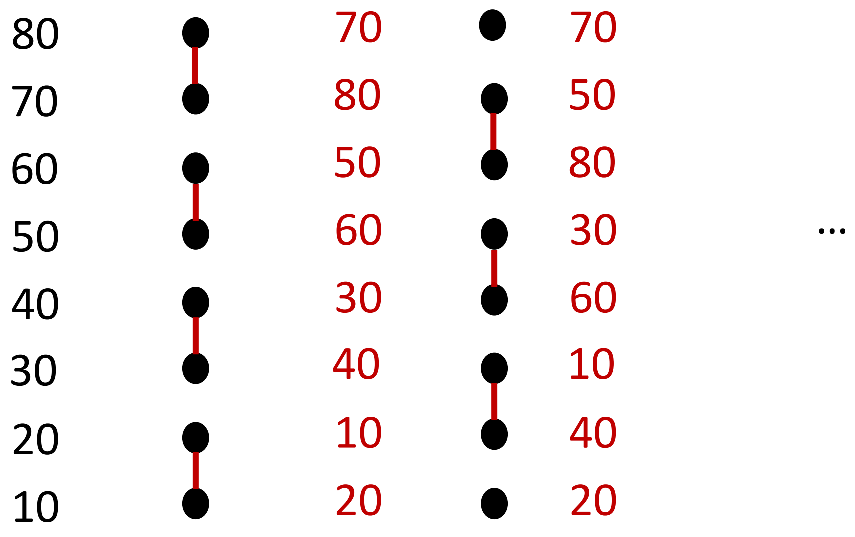
…

Iteration 3
(even)

Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively

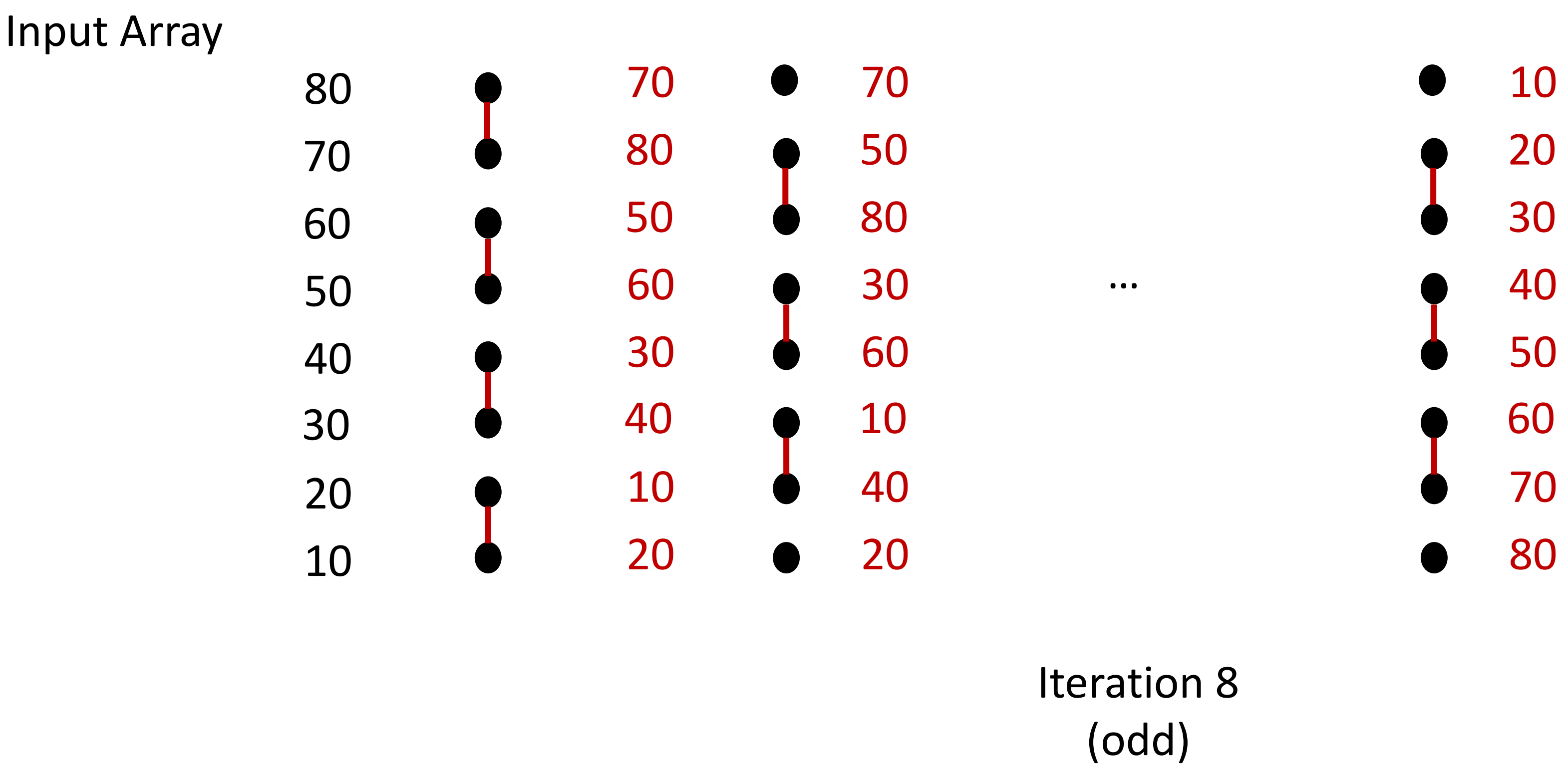
Input Array



Iteration 4
(odd)

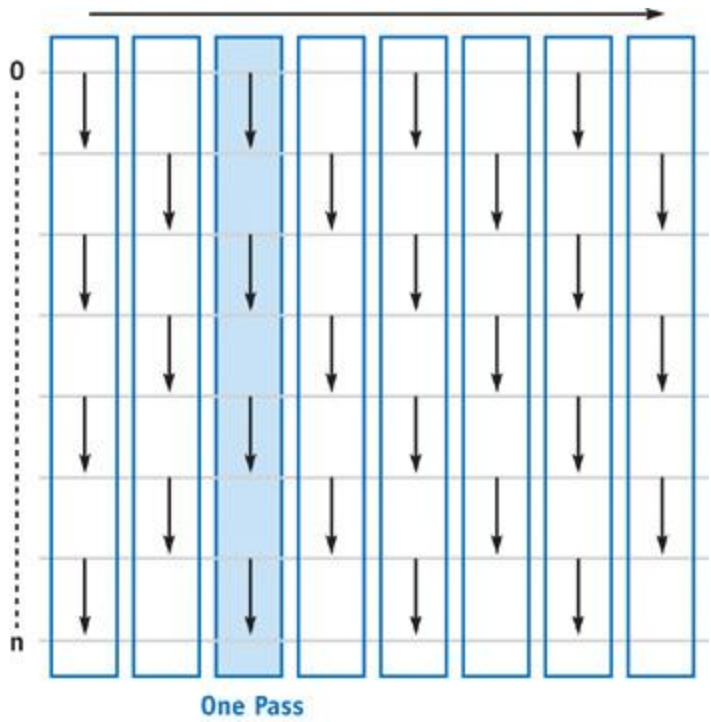
Parallel Sorting

- Odd-Even Sort – Similar to bubble sort,
  - In each iteration, move small keys upwards (and large key downwards) iteratively



Ungraded HW Assignment: Can you simulate the steps of all 8 iterations

Parallel Sorting



Entire Sorting Network – Arrow can be used to represent the direction of comparison

https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-46-improved-gpu-sorting

Parallel Sorting

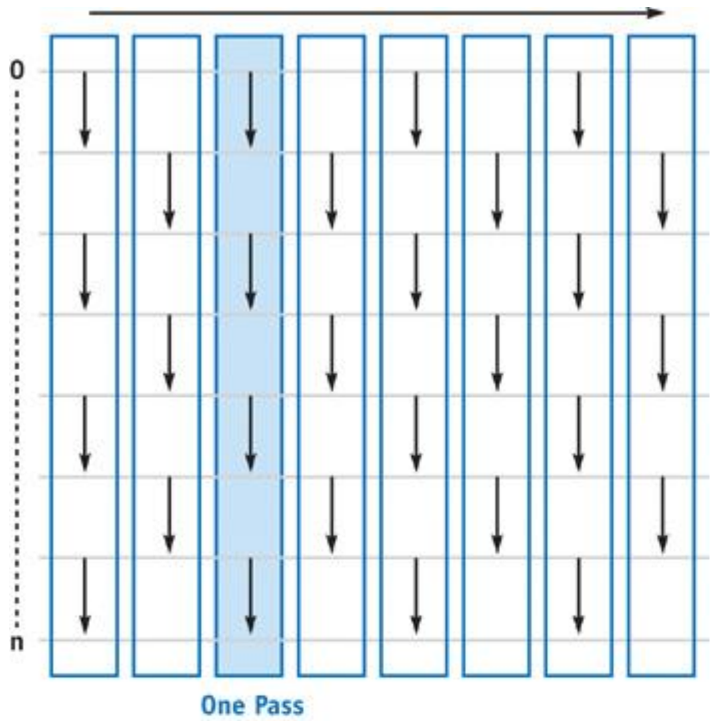
- Odd-Even Sort
- For an array of n elements, terminates in at most $2n$ iterations
- Serial Complexity: ??

Parallel Sorting

- Odd-Even Sort
- For an array of n elements, terminates in at most $2n$ iterations
- Serial Complexity: $O\left(n^2\right)$

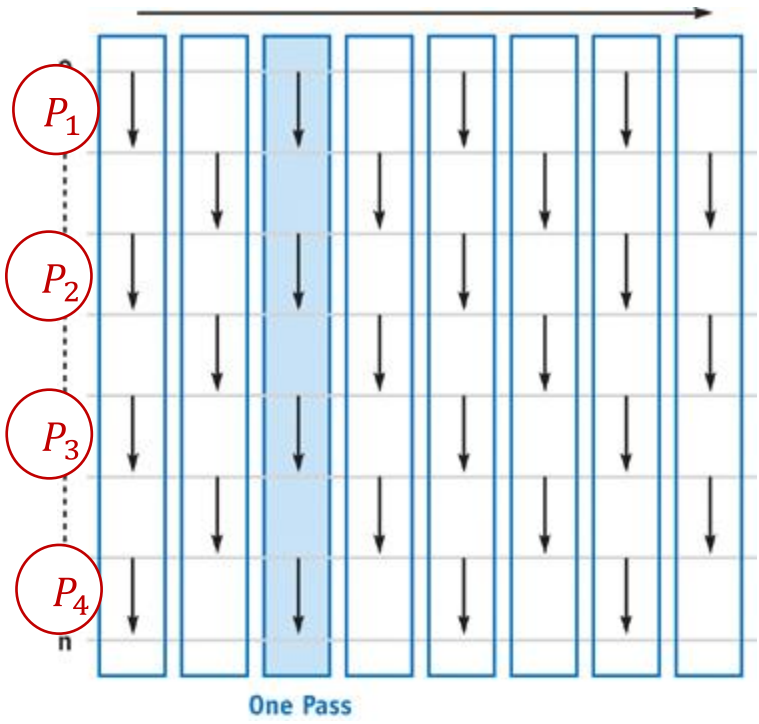
Parallel Sorting

- How do we parallelize?



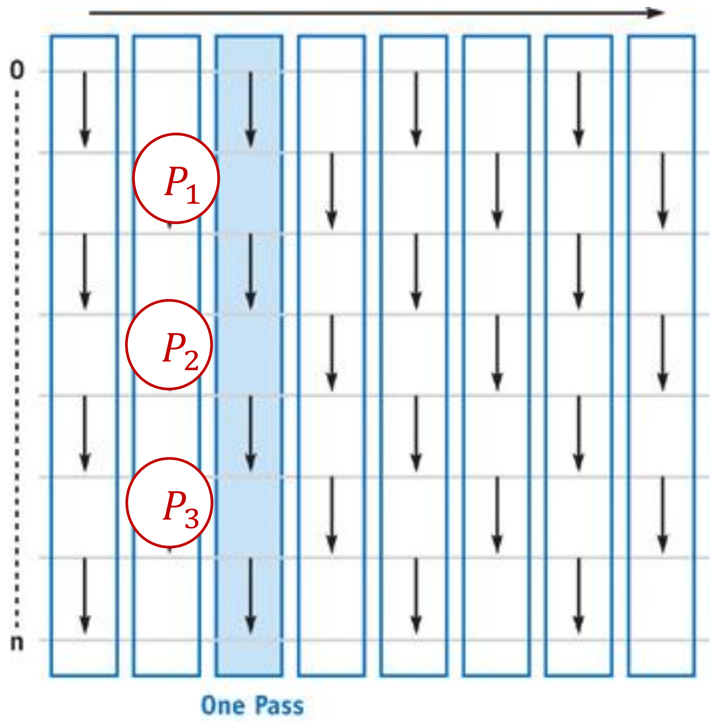
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
  - Iterations still need to occur sequentially



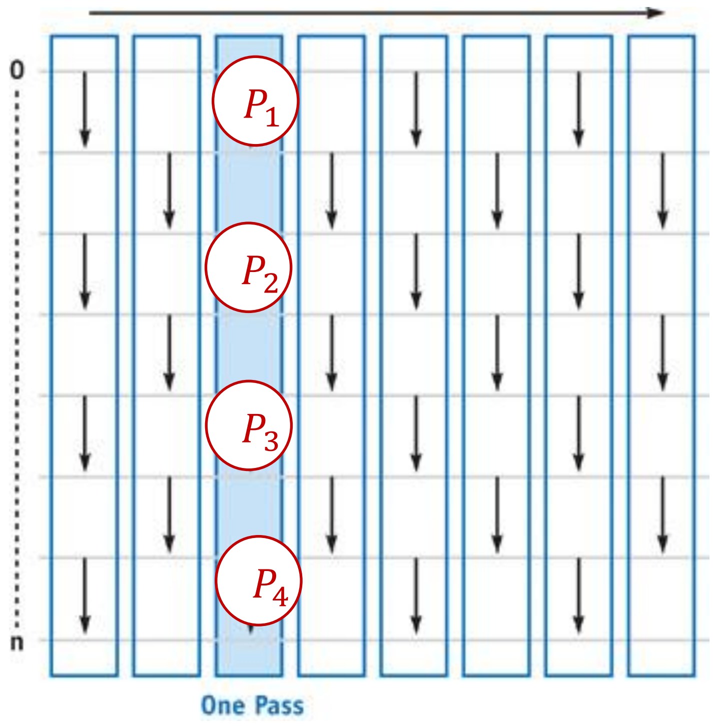
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
  - Iterations still need to occur sequentially



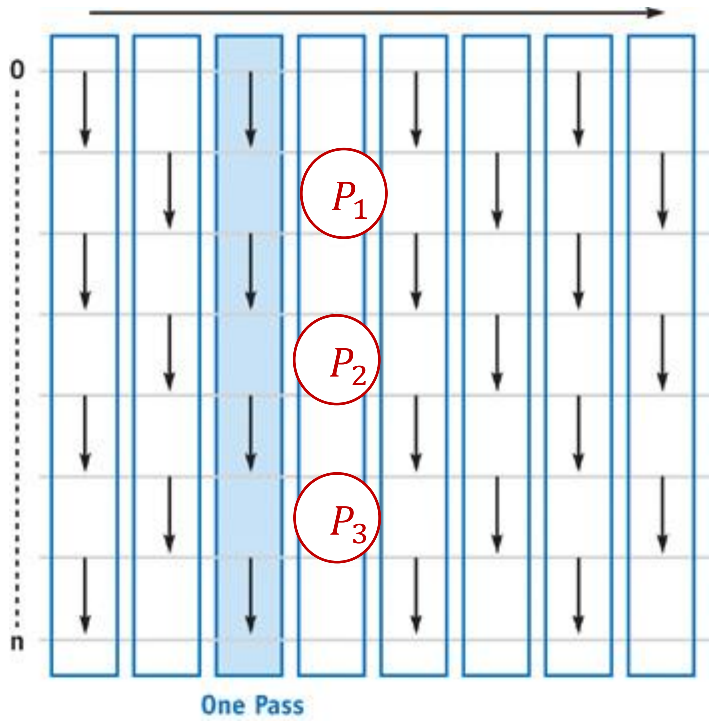
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
  - Iterations still need to occur sequentially



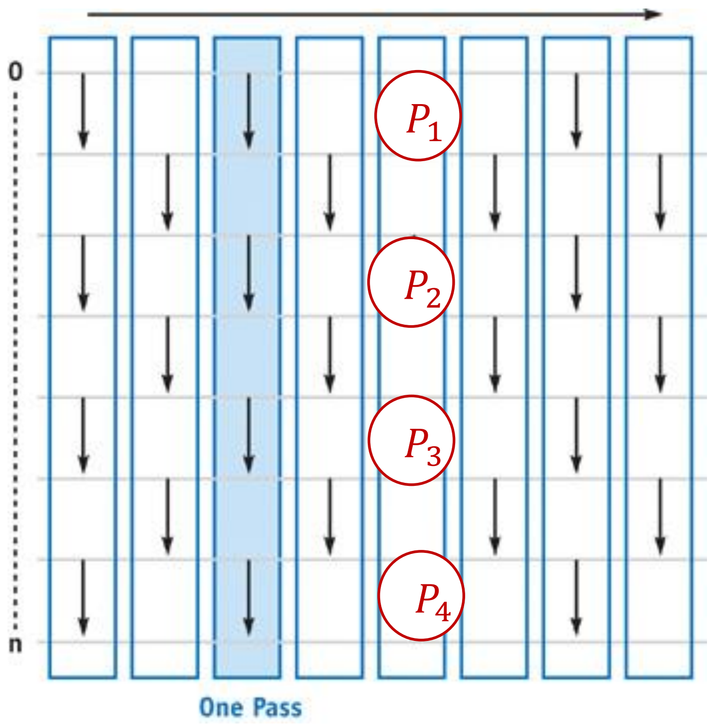
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
    - Iterations still need to occur sequentially



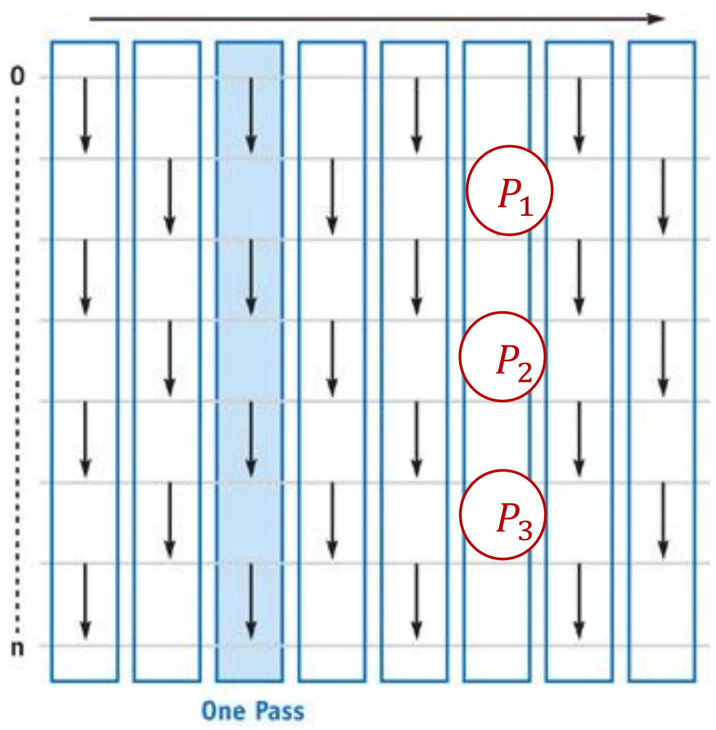
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
  - Iterations still need to occur sequentially



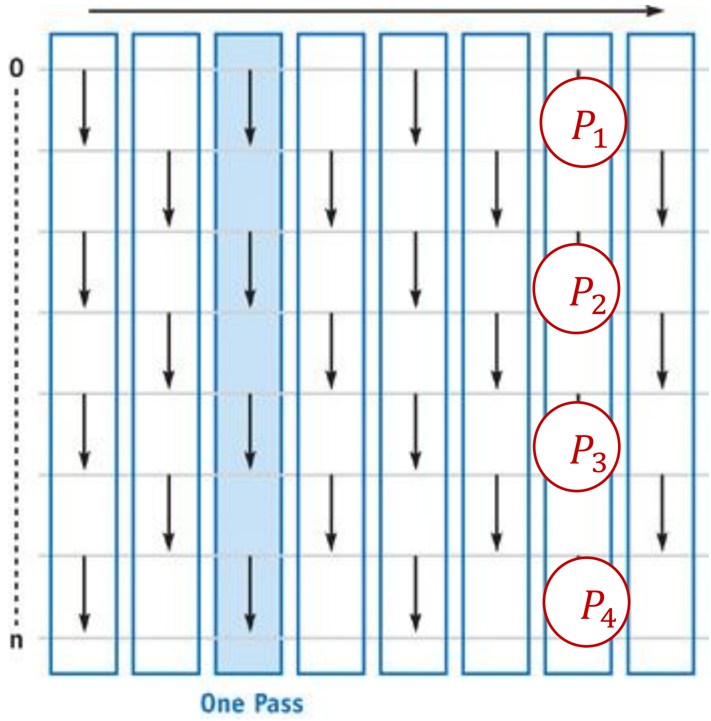
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
    - Iterations still need to occur sequentially



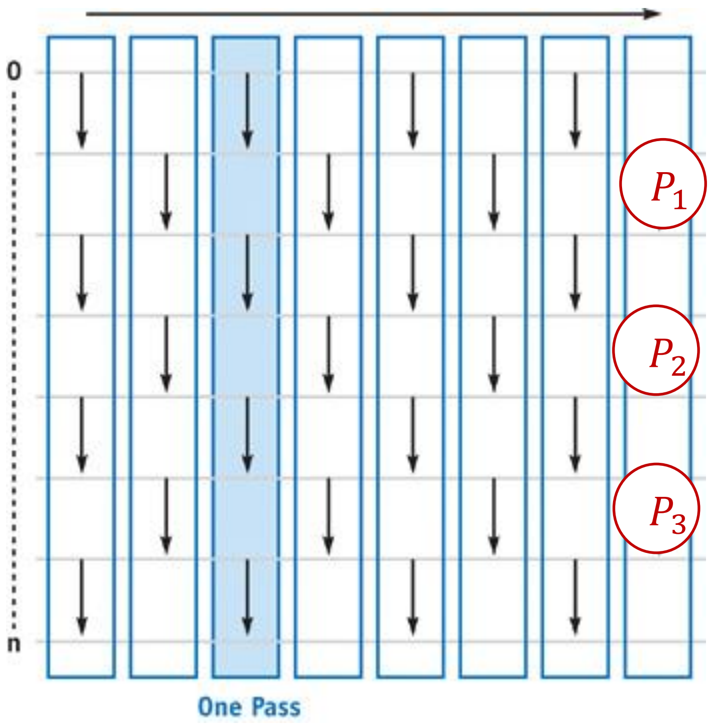
Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
  - Iterations still need to occur sequentially



Parallel Sorting

- How do we parallelize? In each iteration, use a maximum of $\frac{n}{2}$ processors in parallel to do the compare and swap operation
  - Iterations still need to occur sequentially



Parallel Sorting

- Can you calculate speedup. Efficiency, cost, and whether this parallel algorithm is scalable or cost optimal or not? A question on WA 1

- Can you create a GPU code of the algorithm?

- Assume $N = 2 \times S \times p$

- Where do you see a challenge? (We will explore this more in WA 2)

Parallel Sorting

- Even for comparison based sort, algorithms with better cost/work exist
- Check out these papers:
  - Bitton, Dina, et al. "A taxonomy of parallel sorting." *ACM Computing Surveys (CSUR)* 16.3 (1984): 287-318.
  - Singh, Dhirendra Pratap, Ishan Joshi, and Jaytrilok Choudhary. "Survey of GPU based sorting algorithms." *International Journal of Parallel Programming* 46.6 (2018): 1017-1034.
- You are not expected to know these algorithms for exams
  - But I may give you an algorithm and ask you to do analysis or convert into a GPU code.

Outline

- Parallel Program Analysis (Quick Review)
- Data Parallel Algorithms
  - Sorting
- Task Parallelism

Writing Parallel Programs

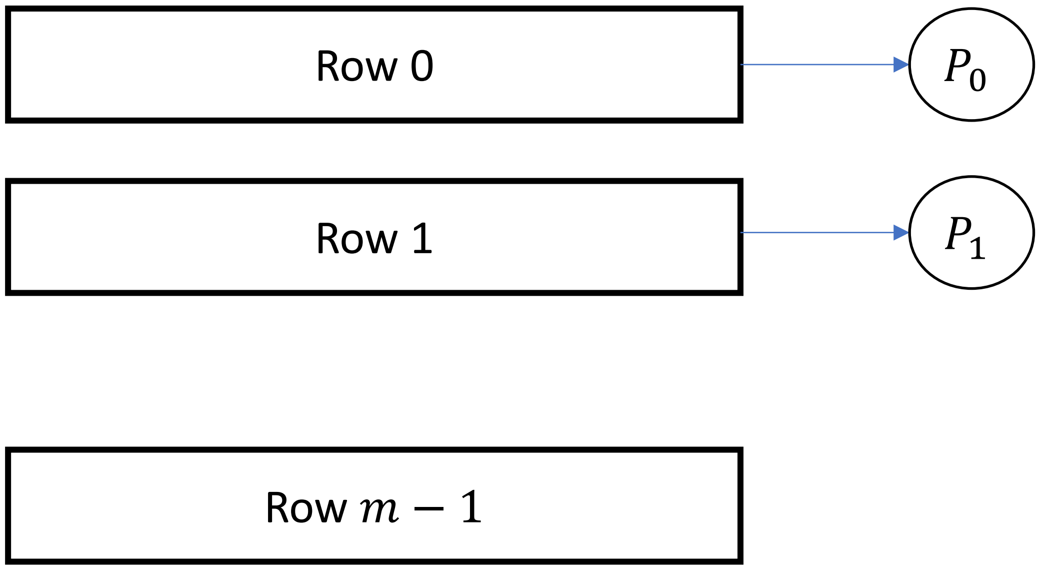
- Key Idea: Need to decompose an algorithm into chunks of independent tasks.

- Such that it can get best performance

- Two approaches
  - Data Parallelism (SPMD): decompose algorithm into tasks that perform “same” work on different data
  - Task Parallelism: decompose algorithm into tasks that perform “different” work

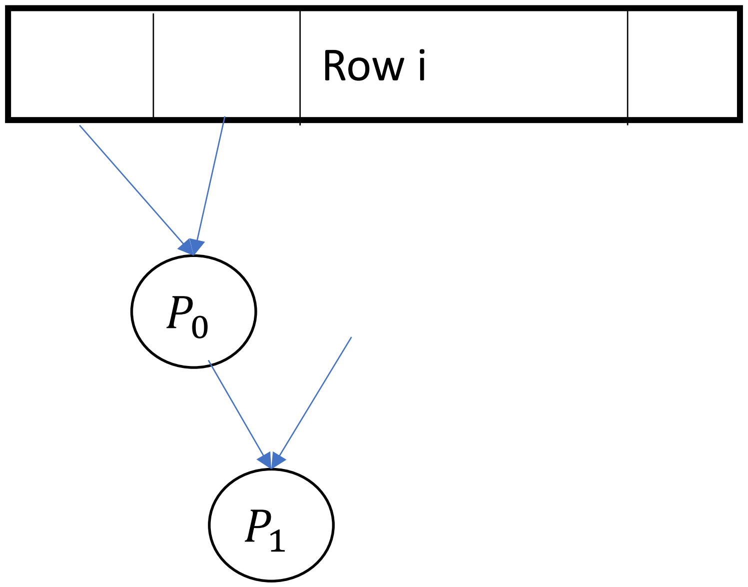
Task vs Data Parallelism

- Consider Matrix Vector Multiplication



Data Parallelism

Note: Data Parallelism is a special case of task parallelism



Task Parallelism

Task Parallelism Techniques

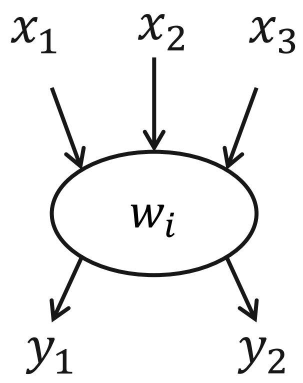
- How to write parallel program using task parallelism paradigm?
- Can be tricky, needs careful analysis of tasks (operations to be performed) and their dependencies
- Careful selection of the granularity of tasks
  - Lowest granularity – operation level (e.g., Task parallel MV)
  - Coarser granularity – example, matrix row level in MV
  - Coarser granularity leads to reduced overheads, but may lose on opportunities of parallelism

Tasks and Dependencies

Computation = decompose into tasks

Task = Set of instructions (program segment)

Inputs & outputs



Begin execution once **all** inputs are available

Task size?

= weight of the node
(e.g., # of instructions executed)

Note: tasks need not be of the same size

Fine grain

Coarse grain

Task Dependency Graph

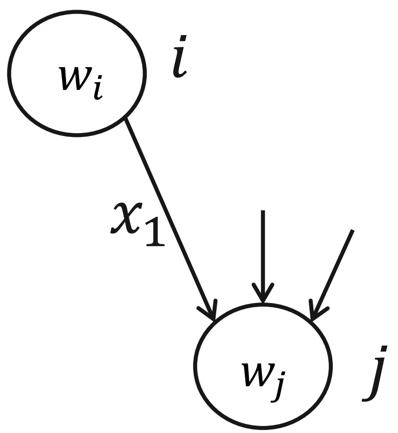
Directed Acyclic Graph

Task_j cannot start until Task_i completes

Data x_1: output of Task_i

Input to Task_j

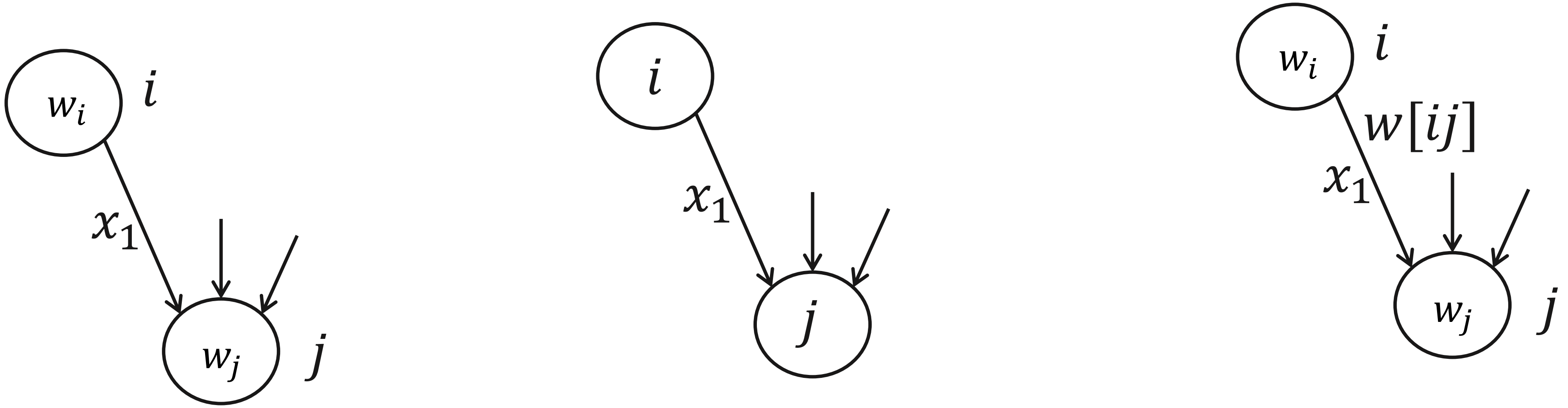
Weight of a node: task size



Task dependency graph need not be connected

Graph is acyclic

Task Dependency Graph



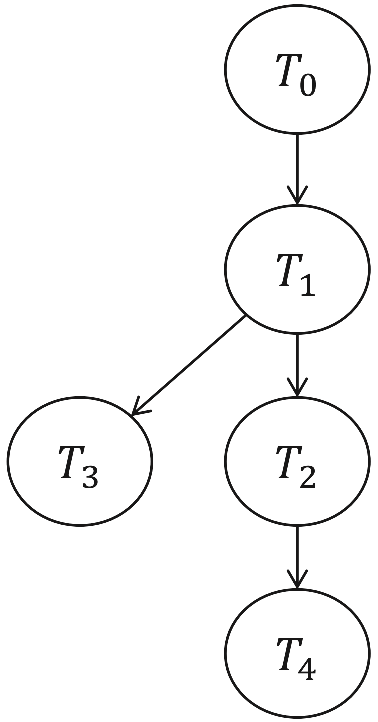
Example (1)

Code

A[2] = A[0] + 1
B[0] = A[2] + 1

Instructions		**Tasks**
Load	R0 ← A[0]	T_0
Add	R1 ← R0 + 1	T_1
Add	R2 ← R1 + 1	T_2
Store	A[2] ← R1	T_3
Store	B[0] ← R2	T_4

Task Dependency Graph



Example (2)

Matrix Vector Multiplication

C[i] = ∑A[i][k] * B[k]

Task_i − Compute C[i]

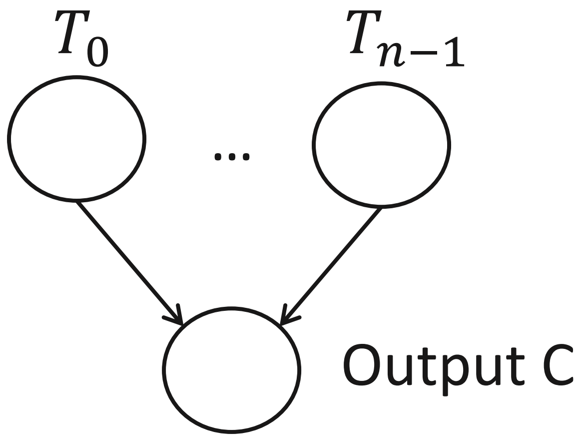
$n \times n$

Parallel for i = 1 to n

Compute C[i]

End

Output C



Maximum Degree of Concurrency (1)

Given a task dependency graph,
maximum number of tasks that can be executed concurrently

Maximum Parallelism Achievable at any Point in the code -> Decide number of processors

Maximum Degree of Concurrency (2)

Example: **level by level ordering (topological sort)**

Task dependency graph

Order tasks level by level

- Any level i task has dependency with some task in level $i-1$ (and possibly with other lower levels) but no dependency with level i

- All tasks in any level i are independent

Execute level i tasks (in parallel), and then level $i+1$ tasks

Maximum Degree of Concurrency (3)

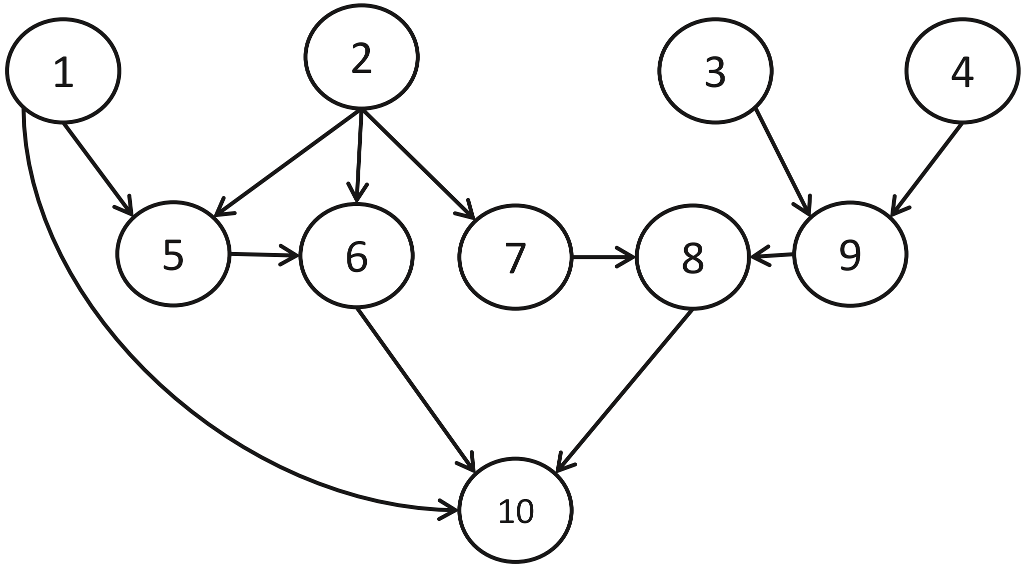
Example: **level by level ordering (cont.)**

Find the number of tasks in each level

Take maximum over all levels = maximum degree of concurrency
(if scheduled using level by level ordering)

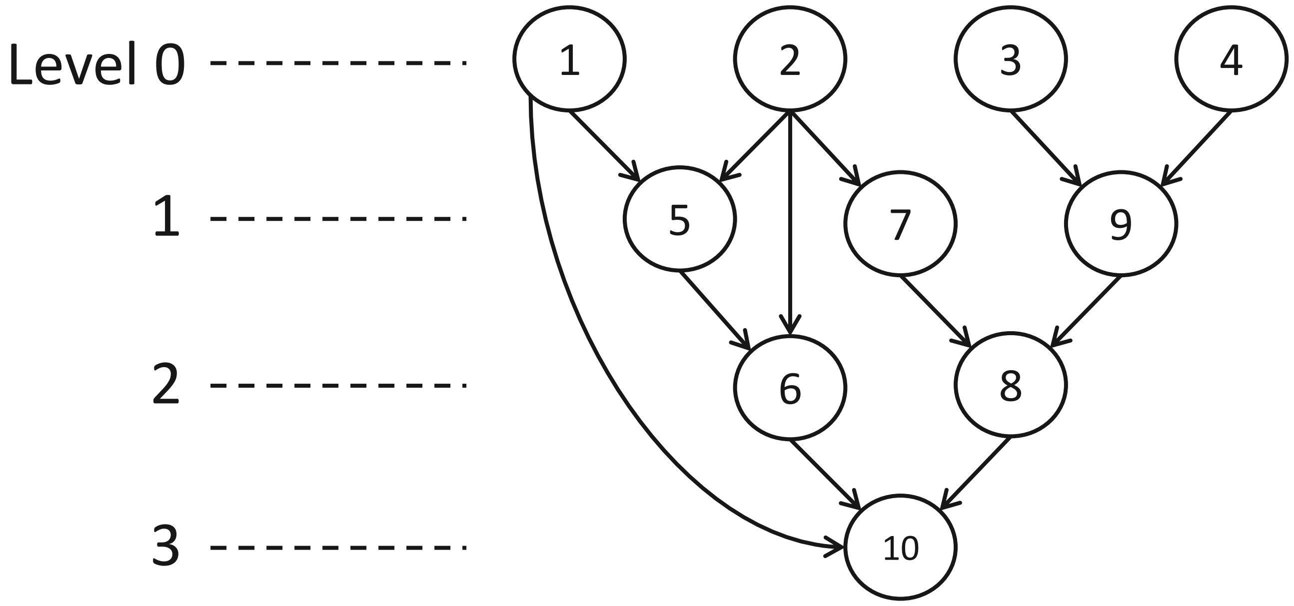
Maximum Degree of Concurrency (4)

Example



Maximum Degree of Concurrency (5)

Example (cont.)



Maximum degree of concurrency = 4

Critical Path (1)

Dependency graph

Start nodes (indeg = 0)

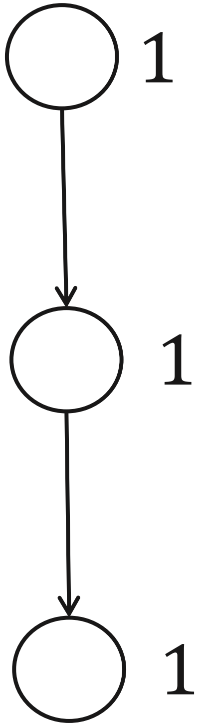
Finish nodes (outdeg = 0)

Critical path = A longest path from a start node to a finish node (# of edges)

Critical path length = Sum of the task weights of the nodes along the critical path

Critical Path (2)

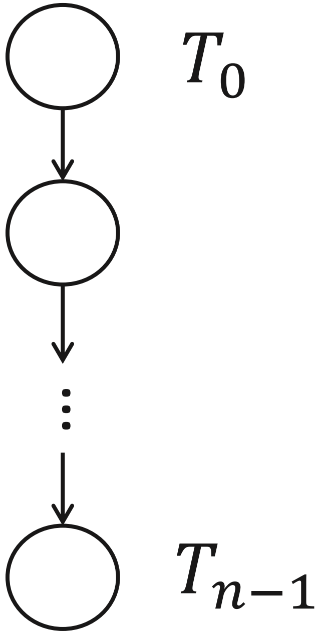
Note: Critical path length considers critical path only (long paths)

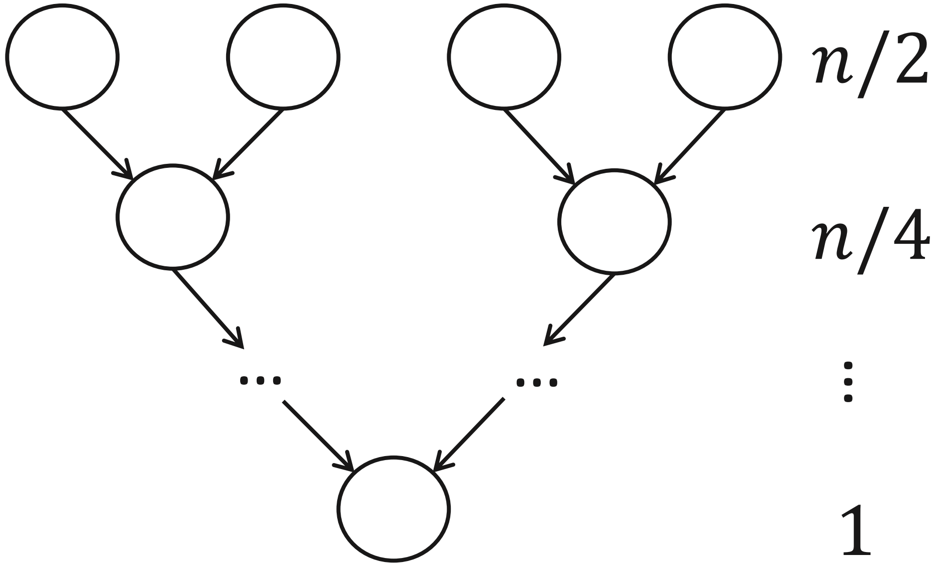


For a given number of tasks,

Longer critical path ⇒ Longer execution time
(may also mean less concurrency)

Critical Path (3)





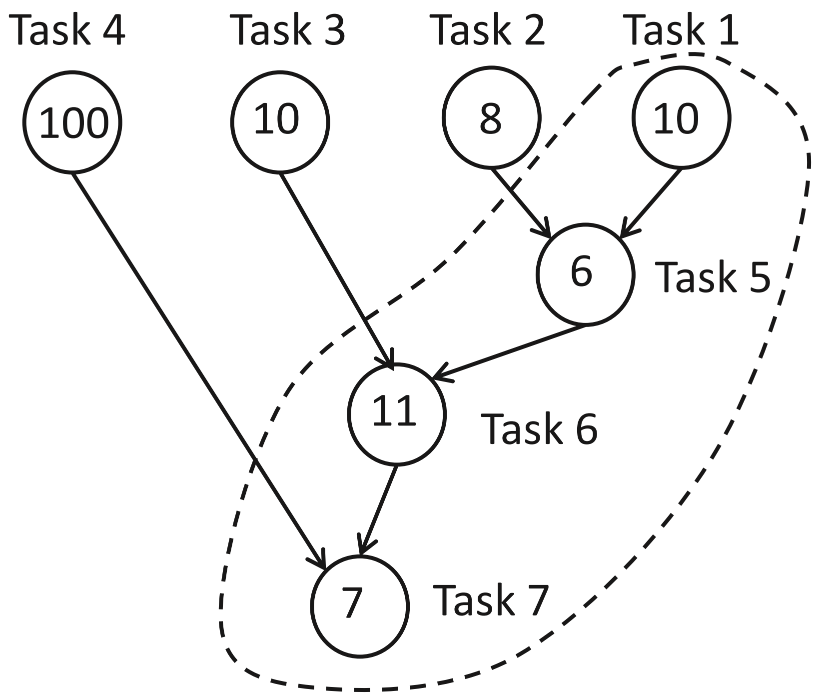
Total no. of tasks $= n$

Critical path length $= n$

Total no. of tasks $= n - 1$

Critical path length $= \log_2 n$

Critical Path (4)



Critical path length = 10+6+11+7 = 34

Task dependency graph to Parallel Program (1)

Given a task dependency graph

Assume weight of each node = 1

Maximum degree of concurrency = c

Critical path length = l

Task dependency graph to Parallel Program (2)

Idea

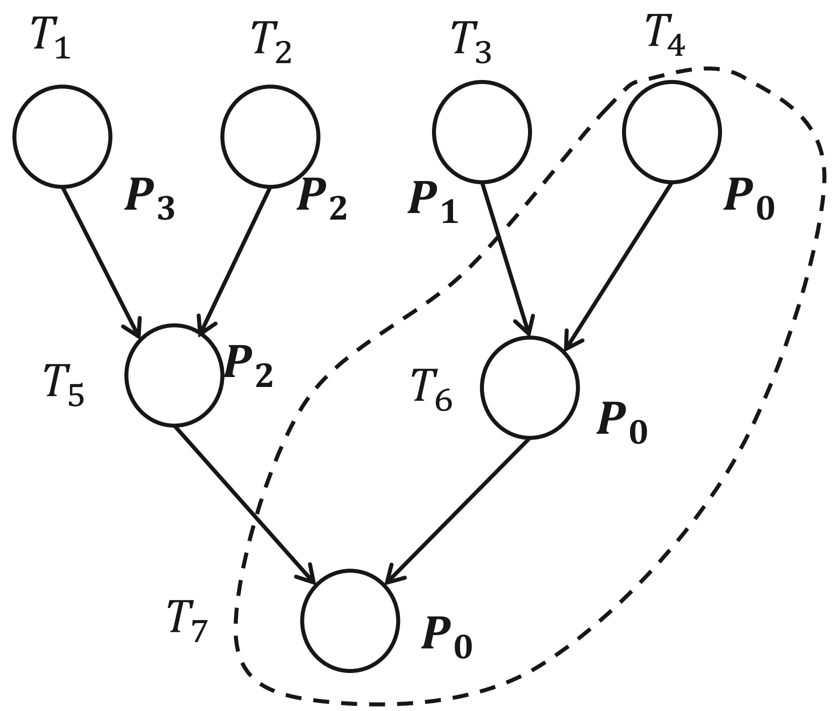
DAG $\longrightarrow$ Organize into levels 0, 1, …, l (level by level ordering)

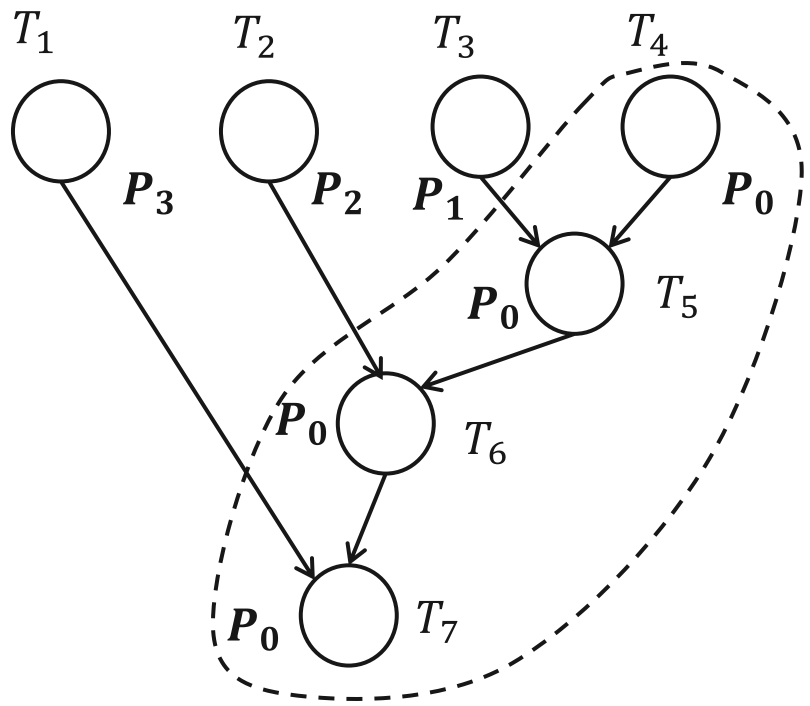
$(l + 1)$ levels

Execute level by level, 0 to l

Total number of processors needed $\leq c$

Mapping Tasks to Processes





Next Class

- 9/18 Lecture 8 – Systolic Array Architecture – Modeling, Matrix Multiplication;

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu