Parallel Algorithm Design: Decomposition and Concurrency

Wei Wang

Supplement Text book for Decomposition and Concurrency

- Introduction to Parallel Computing (2nd Edition) by Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar. Chapter 3
- Slides have all the information you need. Textbook is not necessary. This textbook here is just in case you really need some extra readings.

Basic Concepts of Decomposition and Concurrency

Typical Steps of Designing Parallel Algorithms

- Identify what pieces of work can be performed concurrently
- Partition concurrent work onto independent processors
- Distribute a program’s input, output, and intermediate data
- Coordinate accesses to shared data: avoid conflicts
- Ensure proper order of work using synchronization

Typical Steps of Designing Parallel Algorithms cont'd

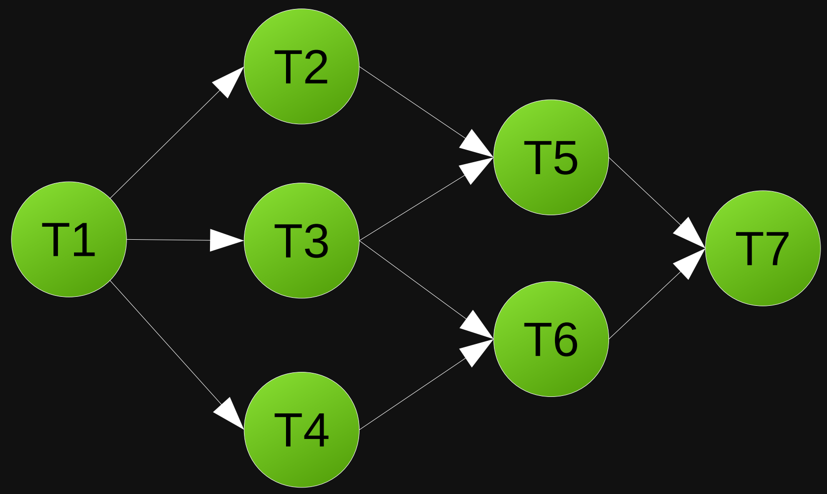
- Why typical? Some steps can be omitted
  - For shared-memory parallel programming model, there is no need to distributed data
  - Fro message passing parallel programming model, there is no need to coordinate shared data
  - Processor partition may be done automatically

Decomposing Work for Parallel Execution

- Identify and divide work into tasks that can be executed concurrently
- There are several ways to decompose a problem
- Tasks may depends on each other, e.g., one task may need the results of other tasks

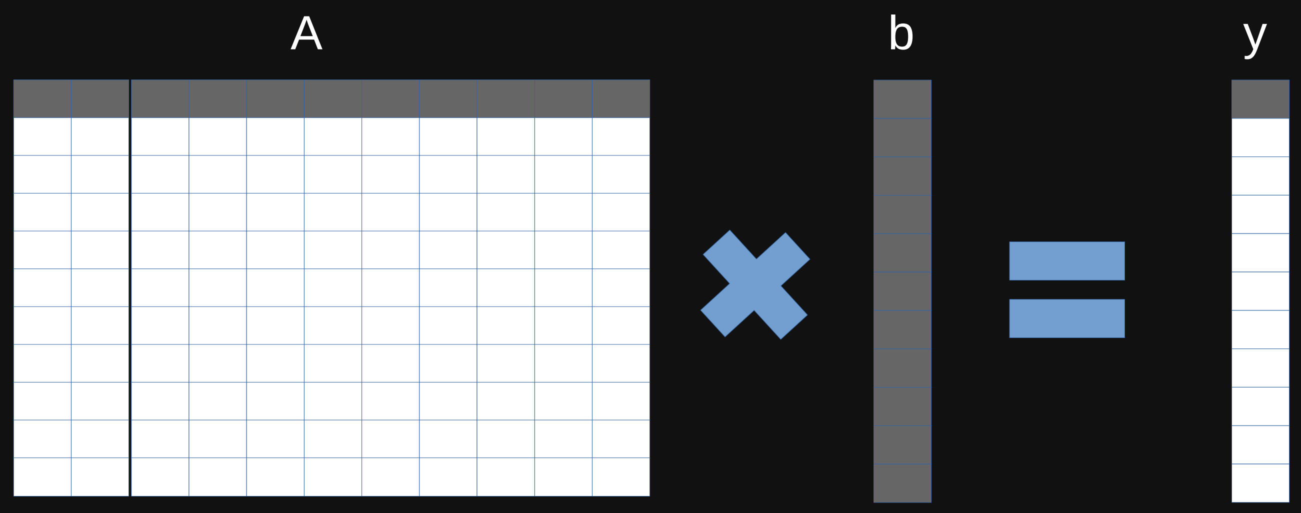
Decomposing Work for Parallel Execution cont’d

- Task dependency graph:
  - Node = task
  - Edge = dependency



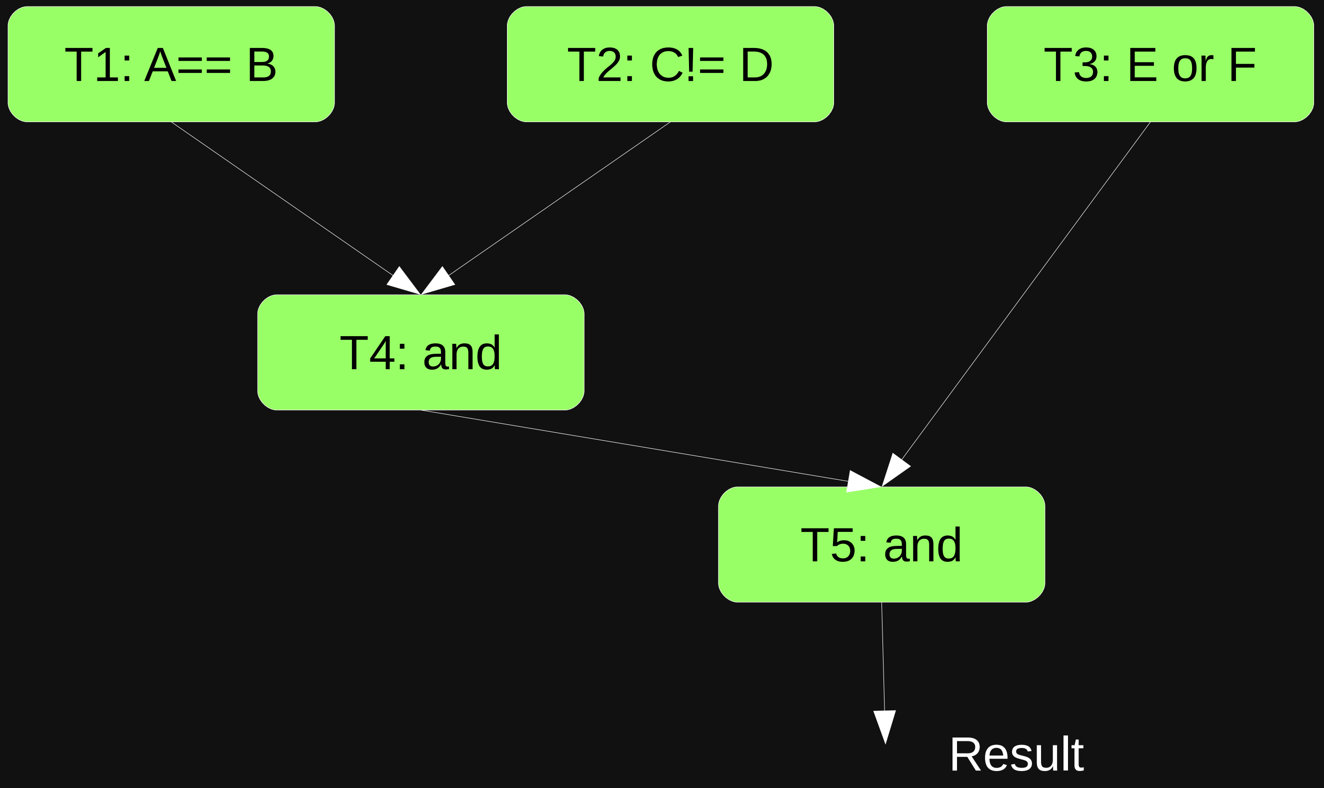
Decomposition Example: Matrix-Vector Multiplication

- 10x10 matrix A multiply 10*1 vector b
- Easy to decompose: each row of A times vector b gives an element of y
- Simple decomposition:
  - Task size is uniform
  - No dependencies between task
  - All tasks share b



Decomposition Example: Logical Expression Evaluation

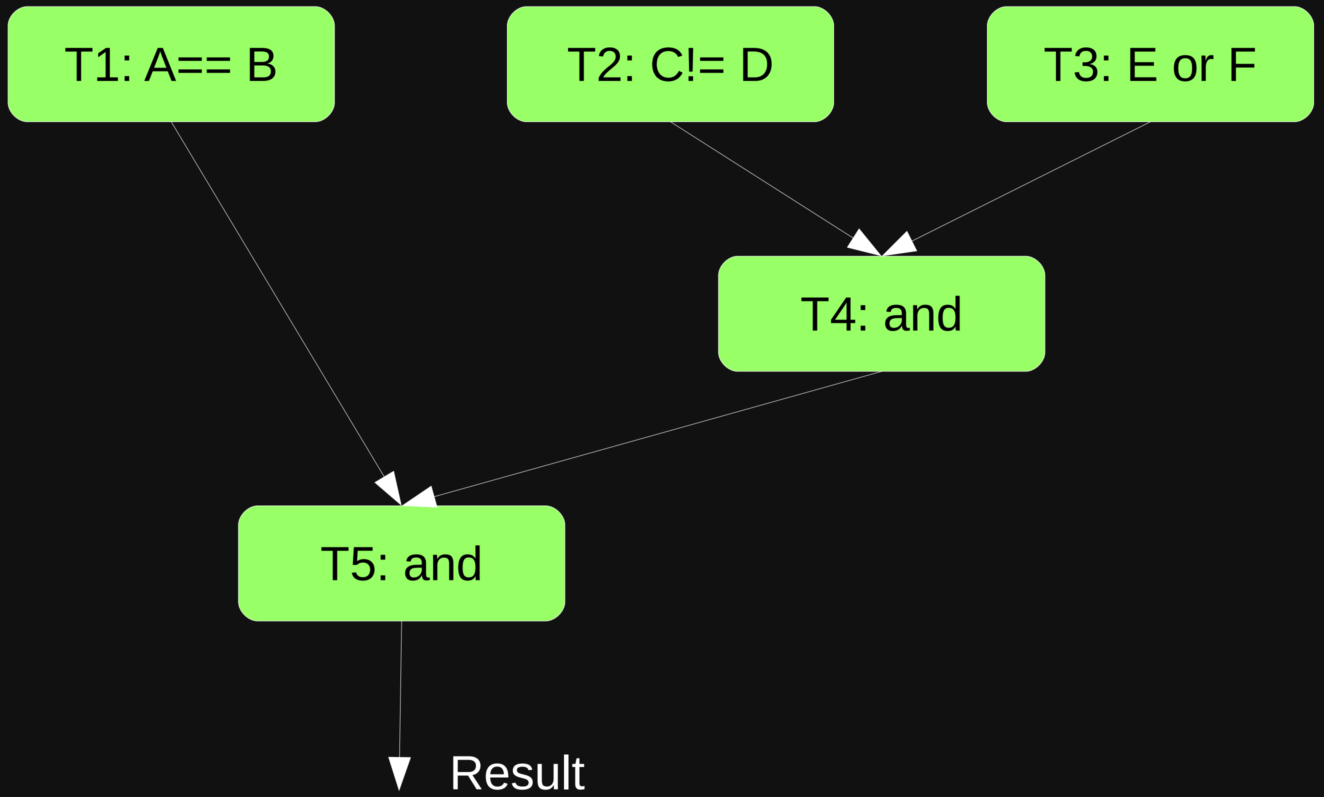
- Evaluate: Y = (A == B) and (C != D) and (E or F)



Decomposition Example: Logical Expression Evaluation cont'd

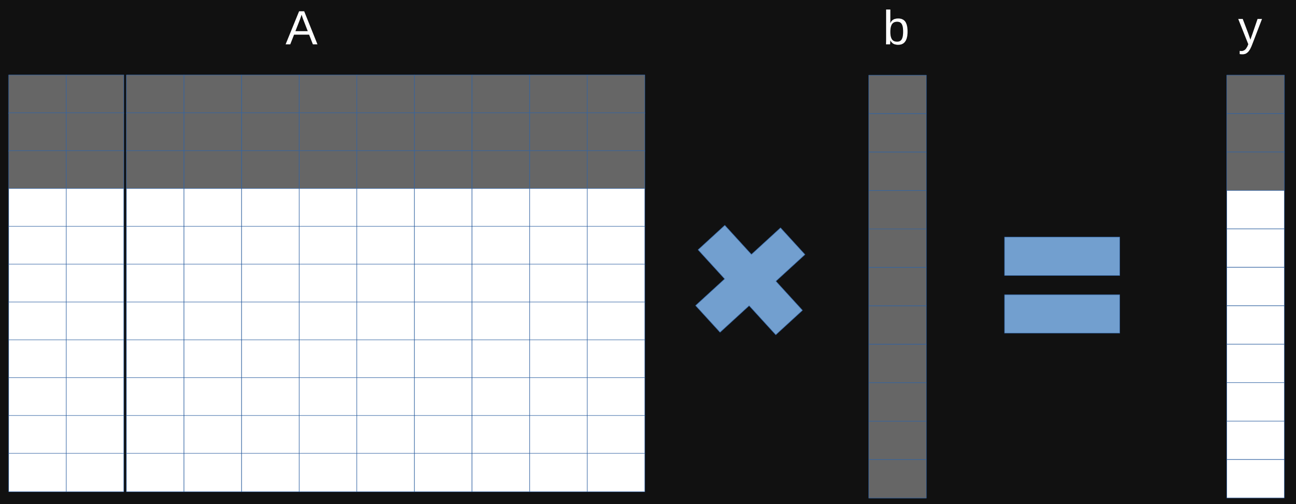
- Evaluate: Y = (A == B) and (C != D) and (E or F)

Alternative dependency graph



Granularity of Task Decomposition

- Granularity = task size
  - Fine-grain = small tasks, large number of tasks
  - Coarse-grain = large tasks, small number of tasks
  - Choose the proper granularity based on the problem and hardware
- Coarse-grained Matrix-Vector Multiplication: each task process three rows of a

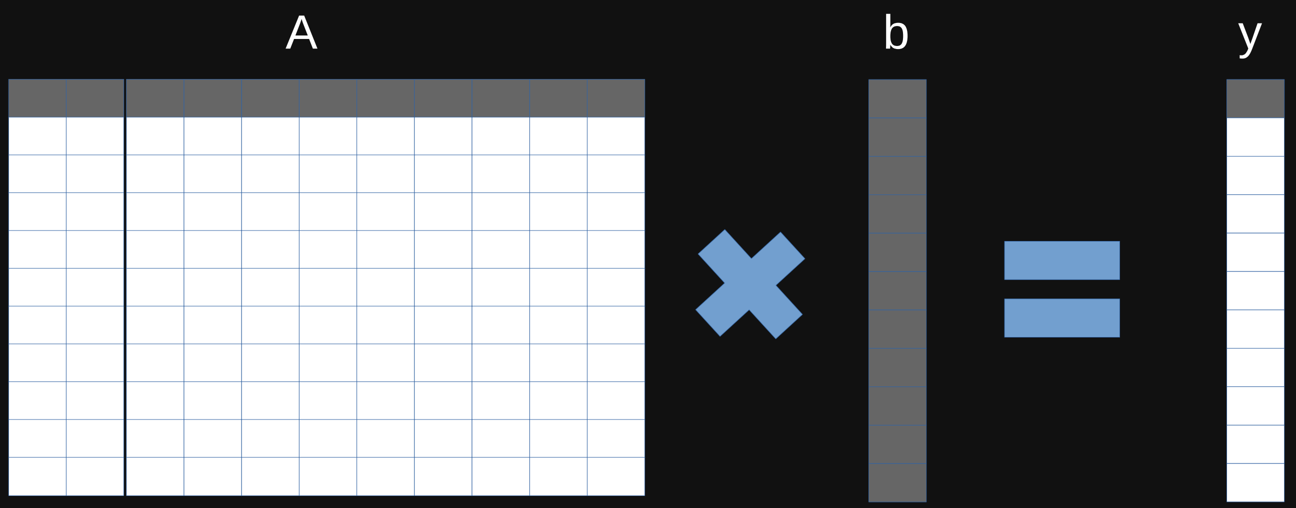


Degree of Concurrency

- Definition: number of tasks that can execute in parallel
  - May change during program execution
- Metrics
  - maximum degree of concurrency: largest # concurrent tasks at any point in the execution
  - average degree of concurrency: average number of tasks that can be processed in parallel
- Degree of concurrency vs. task granularity
  - inverse relationship

Decomposition Example: Matrix-Vector Multiplication

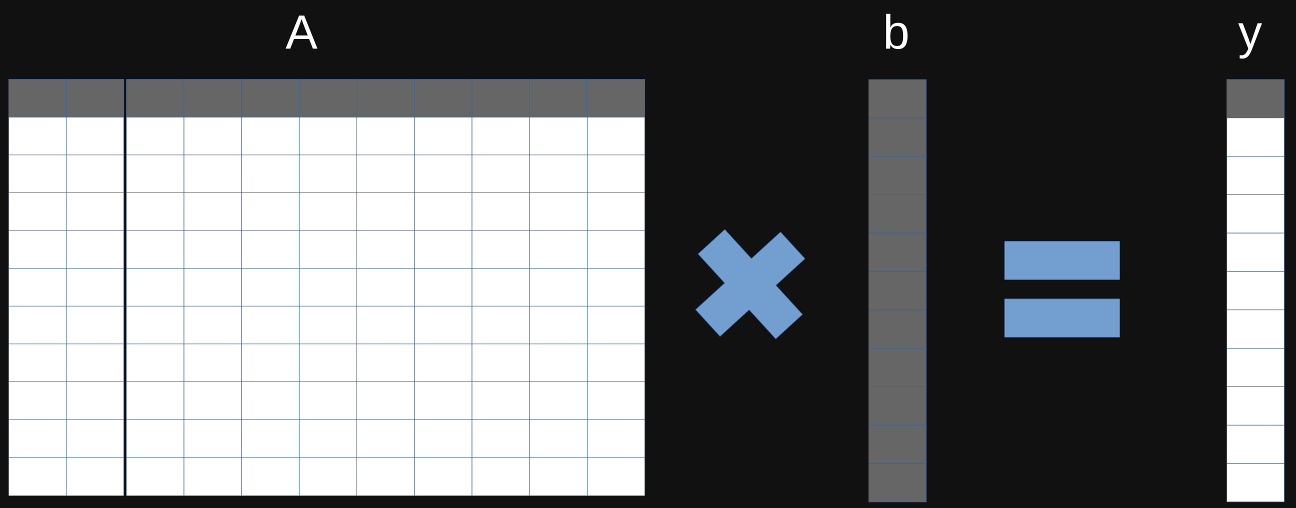
- 10x10 matrix A multiply 10*1 vector b



Question: Is 10 the maximum concurrency possible?

Decomposition Example: Matrix-Vector Multiplication

- 10x10 matrix A multiply 10*1 vector b

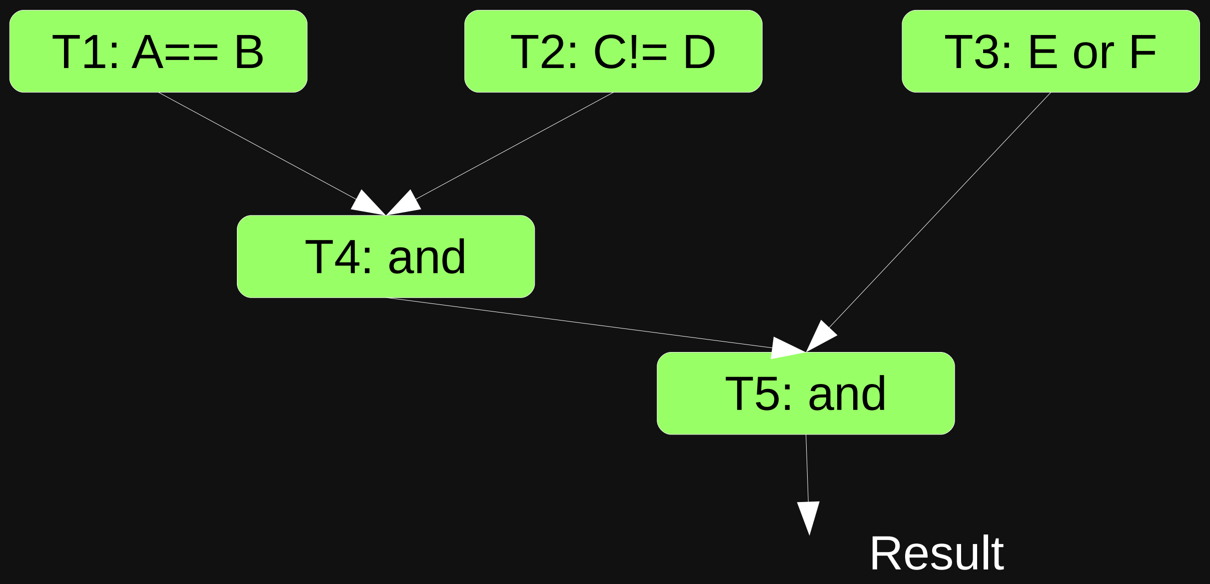


Question: Is 10 the maximum concurrency possible?
A: No, the maximum can be 100, as each task takes one element of A and one element of b and computes their product.

Critical Path

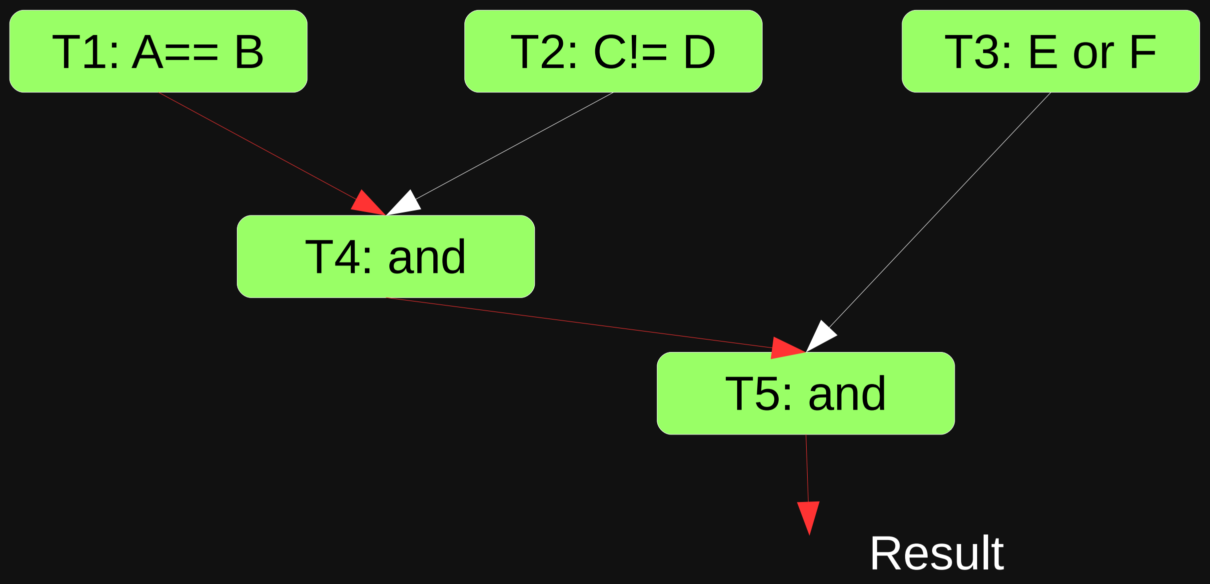
- Edge in task dependency graph represents task serialization
- Critical path = longest path though graph
- Critical path length = lower bound on parallel execution time

Example: Logical Expression Evaluation



- What is the critical path?
- What is the maximum concurrency?
- What is the minimum execution time?

Example: Logical Expression Evaluation



- What is the critical path? ==> Red lines
- What is the maximum concurrency? ==> 3 (T1, T2, T3 are concurrent)
- What is the minimum execution time? ==> 3 (There are three levels)

Example: Matrix-Vector Multiplication

- Assuming each task takes one row of A and all B and computes on element of y
- What is the maximum concurrency?
- What is the minimum execution time?

Example: Matrix-Vector Multiplication

- Assuming each task takes one row of A and all B and computes on element of y
- What is the maximum concurrency? ==> 10 (10 rows of A)
- What is the minimum execution time? ==> 1

Limits on Parallel Performance

- What bounds parallel execution time?
  - maximum task concurrency, e.g. matrix-vector multiplication example ≤ 100 concurrent tasks
  - dependencies between tasks
  - parallelization overheads, e.g., cost of communication between tasks
  - fraction of application work that can’t be parallelized
- Metrics for parallel performance
  - Speedup = T_1/T_p
  - Parallel efficiency = $T_1/(pT_p)$
  - T_1: sequential execution time; T_p: parallel execution time; p: number of processors used

Tasks, Threads and Mapping

- Generally
  - # of tasks > # threads available
  - parallel algorithm must map tasks to threads
- Why threads rather than cores?
  - One thread may process more than one tasks
    - thread = processing or computing agent that performs work
    - assign collection of tasks and associated data to a thread
  - Operating System maps threads to physical cores/processors
    - More than one threads may execute on one core
- Fundamentally, a task may or may not use all of the computation power of a core, so we group several tasks into a thread, and map several threads to a core
  - Threads/Tasks may be blocked by communication and I/O. Therefore, they cannot use the full power of a core.
  - Communication and I/O costs are usually unknown before execution.
  - Threads are added as an extra layer to communicate with OS

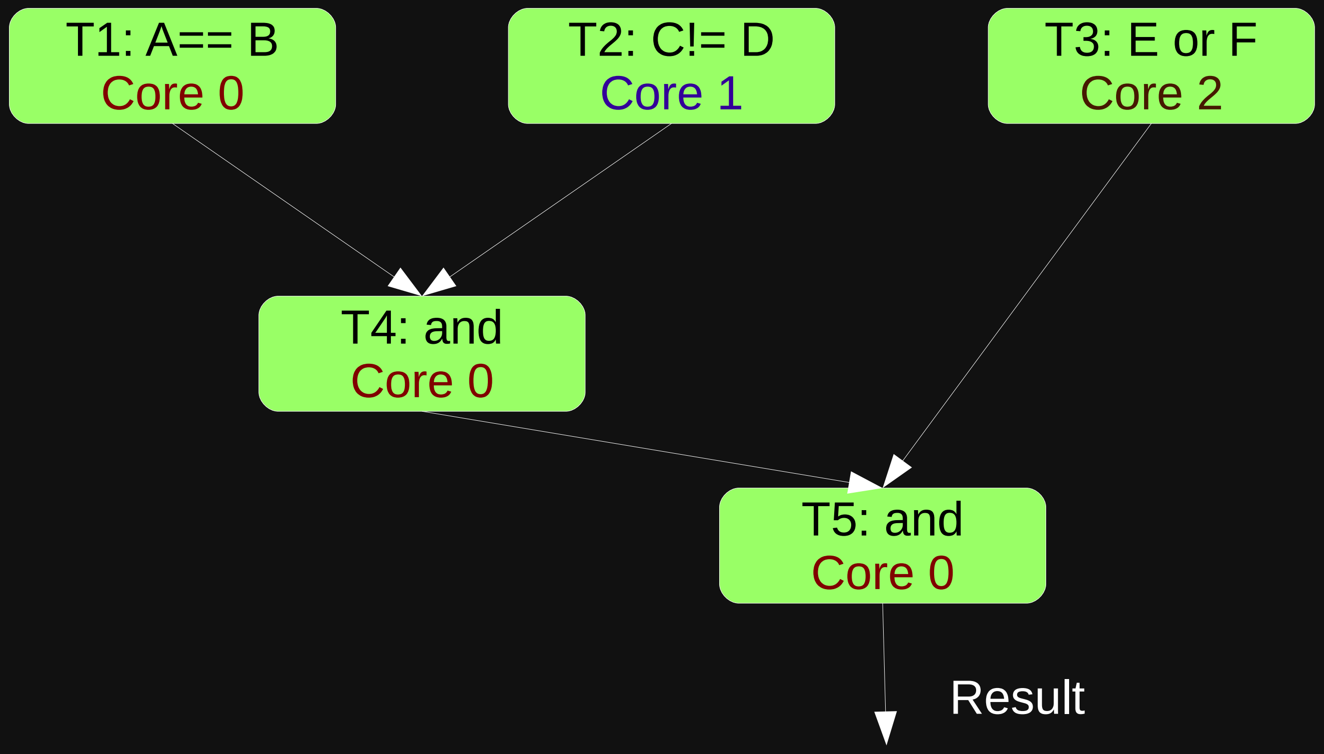
Tasks, Threads and Mapping cont'd

- Mapping tasks to threads is critical for parallel performance
- On what basis should one choose mappings?
  - using task dependency graphs
    - schedule independent tasks on separate threads
      - minimum idling
      - optimal load balance
  - Minimize communication cost
    - Put tasks the communicate to each other in one thread

Tasks, Threads and Mapping cont'd

- A good mapping should:
  - Mapping independent tasks to different threads
  - Assigning tasks on critical path to threads ASAP
  - Minimizing communication cost between threads
- Difficulty: criteria often conflict with one another

Task Mapping Example



- Tasks on the same-level can be executed simultaneously
- Tasks on critical path are sequential and can be executed on the same core

Decomposition Techniques

Decomposition Techniques

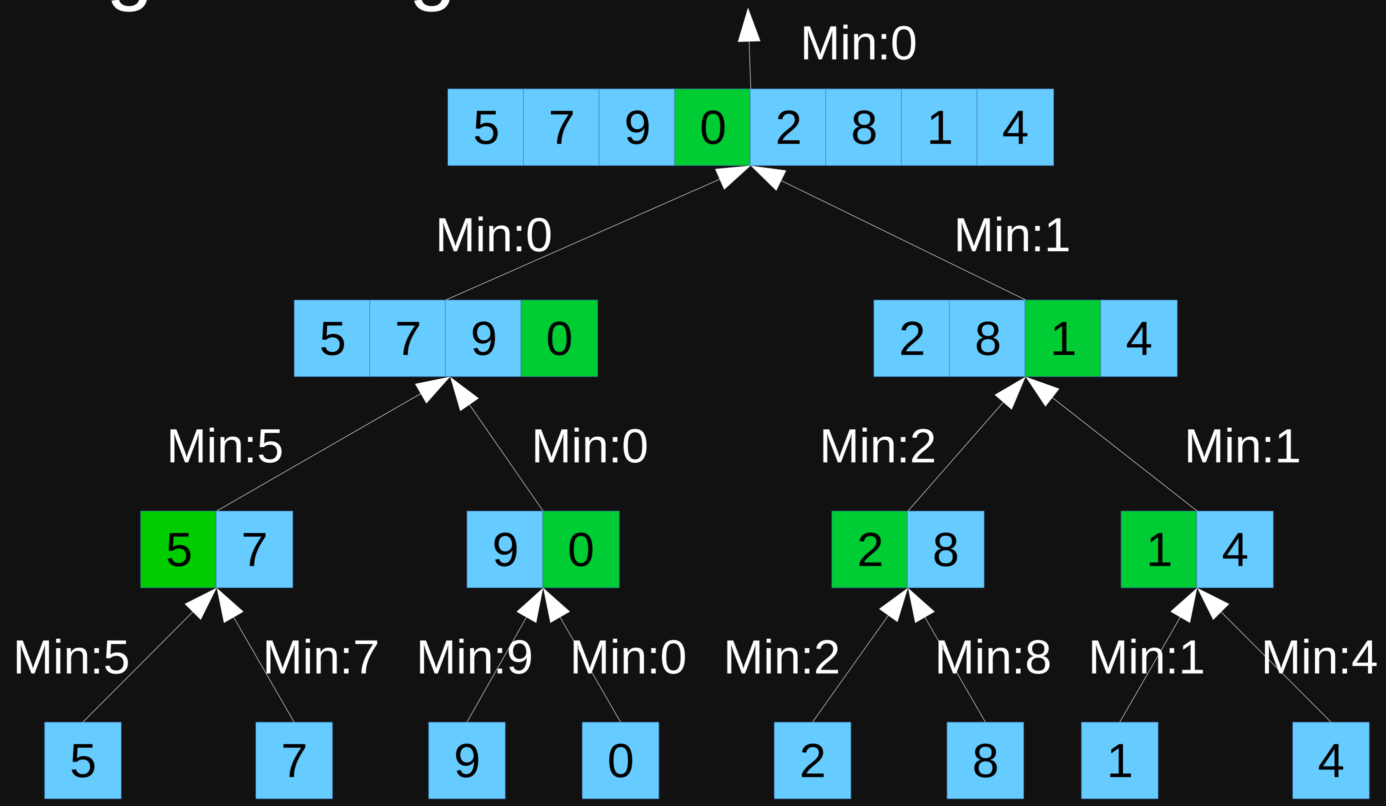
- No single decomposition technique works for all problems
- A variety of techniques are used in practice
  - Recursive decomposition
  - Data decomposition
  - Exploratory decomposition
  - Speculative decomposition

Recursive Decomposition

- Similarly to recursive algorithms
- Example: Finding the minimum integer from an array A[n].
  - Recursive algorithm for finding the minimum:
    (1) If n == 1 (i.e., only one element), return the only element as the minimum. Otherwise, go to step (2).
    (2) Partition A into two [n/2] sub-arrays
    (3) Find the minimums of the two sub-arrays
      - Recursively solved using this algorithm
    (4) Pick the smaller of the two sub-array minimums as the minimum of the original array A[n]

Recursive Decomposition cont'd

- Task dependency graph of this recursive finding-min algorithm



Summary of Recursive Decomposition

- Steps for recursive decomposition
    1. Decompose a problem into a set of sub-problems
    2. Recursively decompose each sub-problem
    3. Stop decomposition when minimum desired subproblem size reached

Data Decomposition

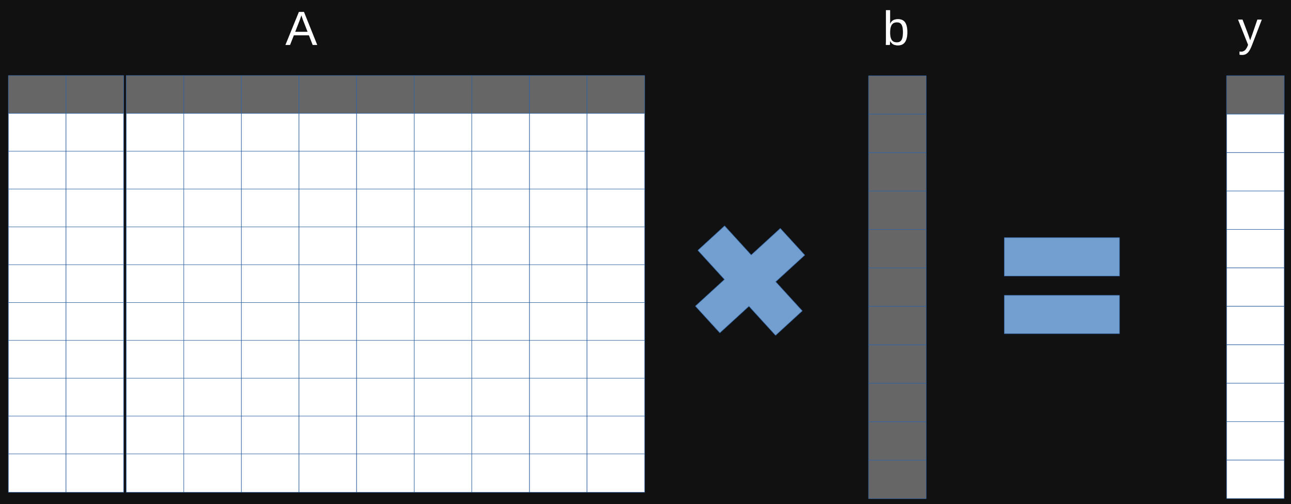
- Essentially partition the data into multiple parts and have each task process on part of the data
- Data can be
  - input data
  - output data
  - Intermediate data
- Data can be partitioned in different ways
  - appropriate partitioning is critical to parallel performance

Decomposition Based on Input Data

- Applicable if each output is computed as a function of the input
- Associate a task with each input data partition
  - task performs computation on its part of the data
  - subsequent processing combines partial results from earlier tasks

Decomposition Based on Input Data cont'd

- Example: Matrix-vector multiplication



Decomposition Based on Output Data

- If each element of the output can be computed independently
- Partition the output data across tasks
- Have each task perform the computation for its outputs

Decomposition Based on Output Data cont'd

- Example: matrix muplication: C = A x B
- Computation of C can be partitioned into four tasks:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & c_{2,2} \end{pmatrix}$$

  – Task1: $C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$
  – Task2: $C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$
  – Task3: $C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$
  – Task4: $C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$

Intermediate Data Partitioning

- If computation is a sequence of transforms
  - Input data computed to intermediate data, then intermediate data computed to output data
- Can be decomposed based on data for intermediate stages
  - Usually employed to reduce communication cost (e.g., reduce the cost to communicate intermediate results)

Intermediate Data Partitioning cont'd

- Example: matrix muplication: C = A x B
- Two staged algorithm:
  - Stage1: $\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$
  - Stage2: $\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & c_{2,2} \end{pmatrix}$

Intermediate Data Partitioning cont'd

- Example: matrix muplication: C = A x B
- Tasks:

  Task 1: $D_{1,1,1} = A_{1,1} \cdot B_{1,1}$ Task 2: $D_{2,1,1} = A_{1,2} \cdot B_{2,1}$

  Task 3: $D_{1,1,2} = A_{1,1} \cdot B_{1,2}$ Task 4: $D_{2,1,2} = A_{1,2} \cdot B_{2,2}$

  Task 5: $D_{1,2,1} = A_{2,1} \cdot B_{1,1}$ Task 6: $D_{2,2,1} = A_{2,2} \cdot B_{2,1}$

  Task 7: $D_{1,2,2} = A_{2,1} \cdot B_{1,2}$ Task 8: $D_{2,2,2} = A_{2,2} \cdot B_{2,2}$

  Task 9: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$ Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

  Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$ Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Summary of Data Decomposition

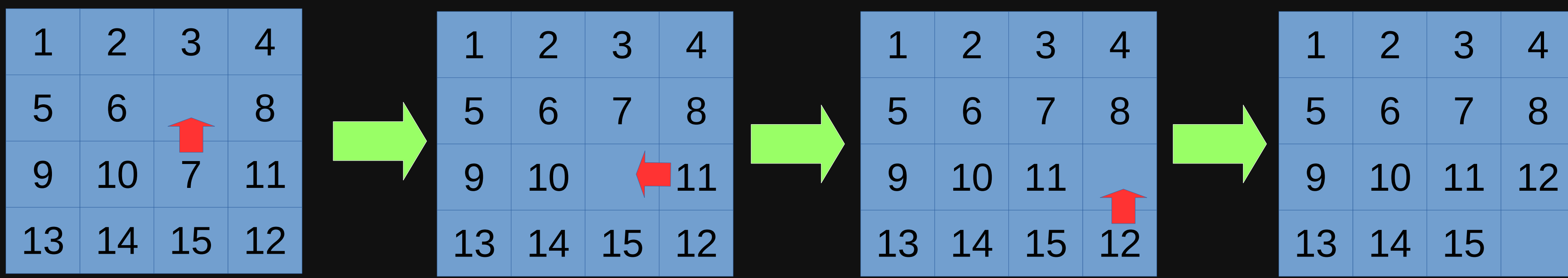
- Partition the data, and let each task work on one part of the data
- Can be decomposed based on
  - Input data
  - Output data
  - Intermediate data
  - Which partition is better depends on the problem, data structure and hardware structure

Exploratory Decomposition

- Usually used in exploring a search space for the solution
  - Problem decomposition reflects the shape of execution, i.e., the shape of the search space
- Examples
  - Discrete optimization (Integer Programming)
  - Theorem proving
  - Game plays

Exploratory Decomposition cont'd

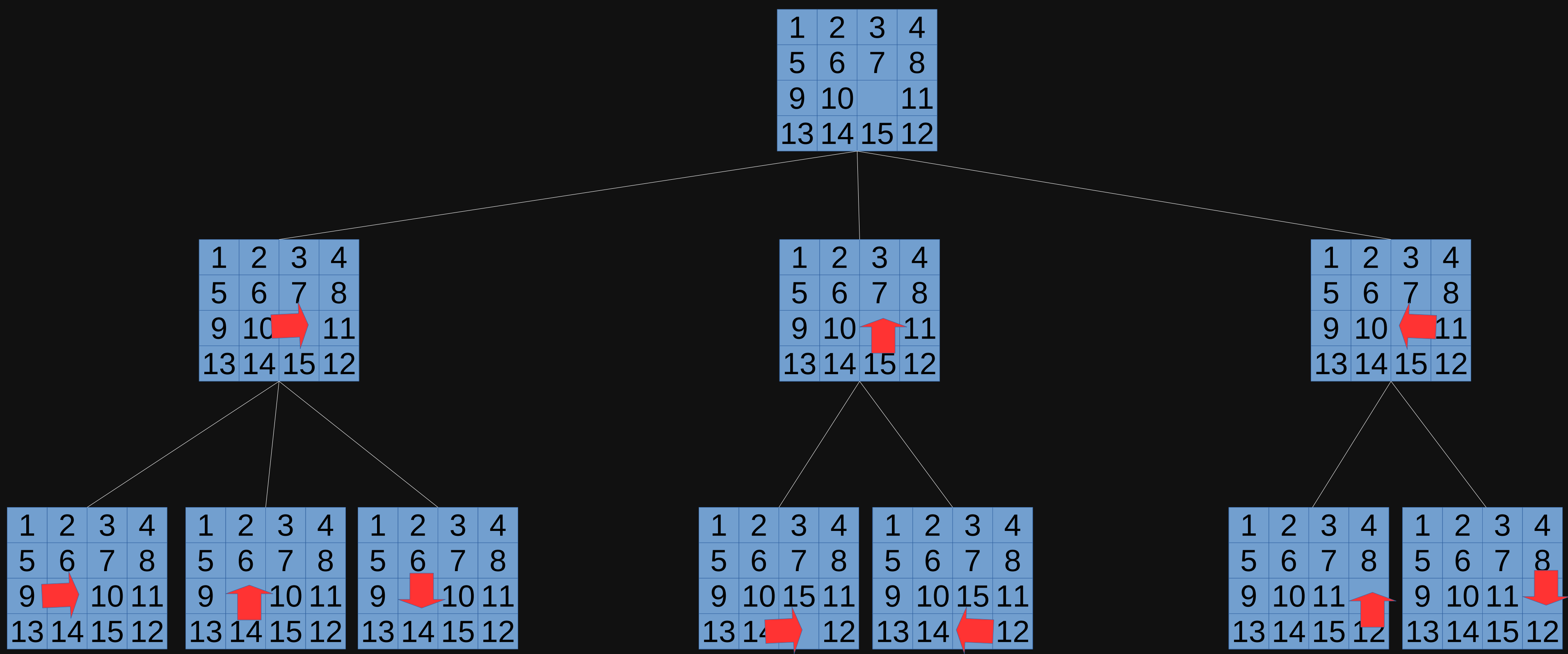
- Example: 15 puzzle



- From computer to solve a 15 puzzle, the computer has to search for a solution

Exploratory Decomposition cont'd

- Computer search for 15 puzzle solution. Search Tree after the first move



Final Solution

Exploratory Decomposition cont'd

- To find a solution, a computer has to search the solution tree one level at a time, until a solution is found
- Each level can be processed in parallel
  - Each tree node is a task
- The number of tasks at each level depends on the previous states, i.e., the task count and decomposition is not known before each level is processed
- The total number of tasks to be processed is also unknown at the beginning
  - Therefore the performance is hard to anticipate
  - Parallel algorithms with exploratory decomposition may experience no speedup (over sequential algorithms), super-linear speedup or anything in between
  - In some extreme cases, parallel algorithms with exploratory decomposition may even experience slow down comparing to their corresponding sequential algorithms.

Speculative Decomposition

- Dependencies between tasks are not always known a-priori
  - makes it impossible to identify independent tasks
- Conservative approach
  - identify independent tasks only when no dependencies left
- Optimistic (speculative) approach
  - schedule tasks even when they may potentially be erroneous
- Drawbacks for each
  - conservative approaches
    - may yield little concurrency
  - optimistic approaches
    - may require a roll-back mechanism if a speculation is wrong

Hybrid Decomposition

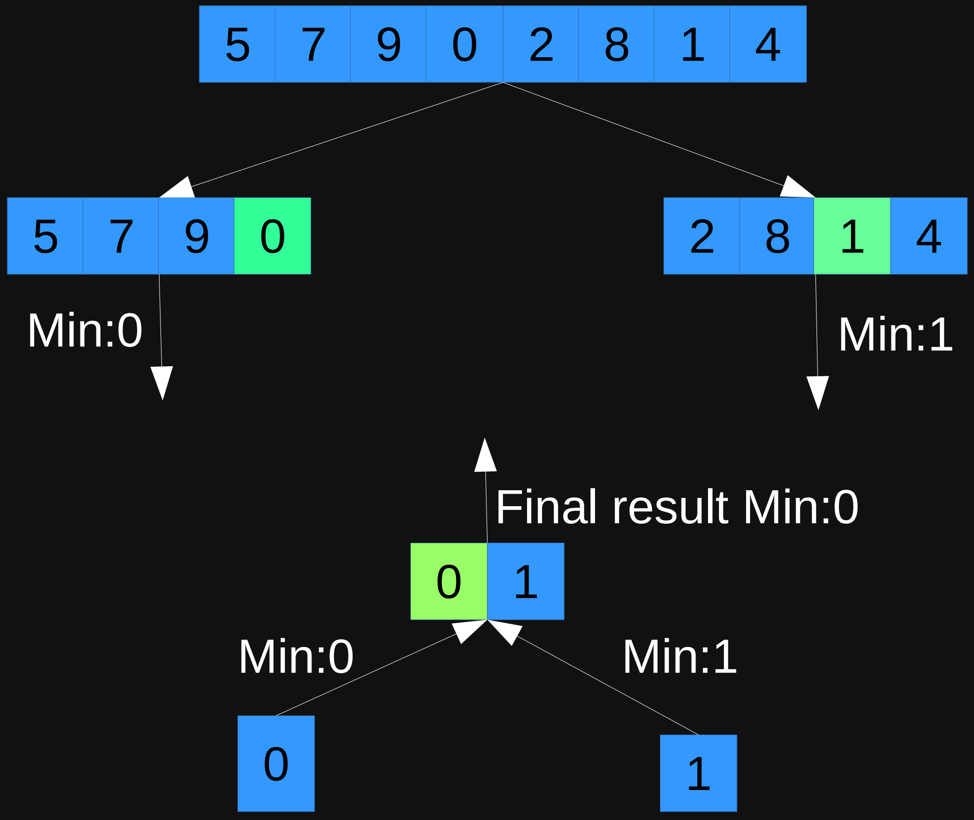
- Use multiple decomposition strategies together
- Often used to improve concurrency or reduce parallel overhead
- Example: Find the minimum of an array
  - Recursive decomposition may generate too many tasks, more tasks than the processors; too many tasks incurs high scheduling/communication cost
  - A hybrid decomposition for this problem can be – first use input data decomposition to get several minimums, then use recursively decomposition to find the minimum of the minimums

Hybrid Decomposition

- Example: determine the minimum of an array

Step 1: use input data decomposition to partition input data into two arrays and find the min of each array

Step 2: put the two minimums into one array and find their minimum using recursive decomposition



Characteristics of Tasks and Interactions

Characteristics of Tasks

- Key characteristics
  - Generation strategy
  - Associated work
  - Associated data size
- Affect performance of parallel algorithm

Characteristic 1: Task Generation

- Static task generation
  - Tasks are identified before execution
  - Typically decomposed using data or recursive decomposition
  - Examples
    - Matrix operations
    - Graph algorithms on static graph
    - Image processing
- Dynamic task generation
  - Task are identified during execution
  - Typically decomposed using exploratory or speculative decompositions
  - Examples
    - Games, puzzles
    - Simulations

Characteristic 2: Task Work Size

- Uniform: all tasks have the same size
- Non-uniform
  - Sometimes sizes known or can be estimate before execution
  - Sometime not
    - Examples:
      - Quick sort
      - Games, puzzles

Characteristic 3: Task Data Size

- Large data and small data
  - Usually data is compared with computation
    - Cost (time) of data communication V.S. cost (time) of computation
  - Large Data: Data > Computation
    - Ties task to a thread/core to avoid communication
  - Small Data: Data < Computation
    - Tasks can be easily migrated (e.g., migrate to a faster processor when the processor becomes available)
- Data size != input size != output size
  - Intermediate data can be larger than both input and output
    - Example: 15 puzzle,

Task Interactions

- Four orthogonal types of task interactions
  - Static vs Dynamic
  - Regular vs Irregular
  - Read-only vs Read-write
  - One-sided vs Two-sided

Static vs Dynamic Interactions

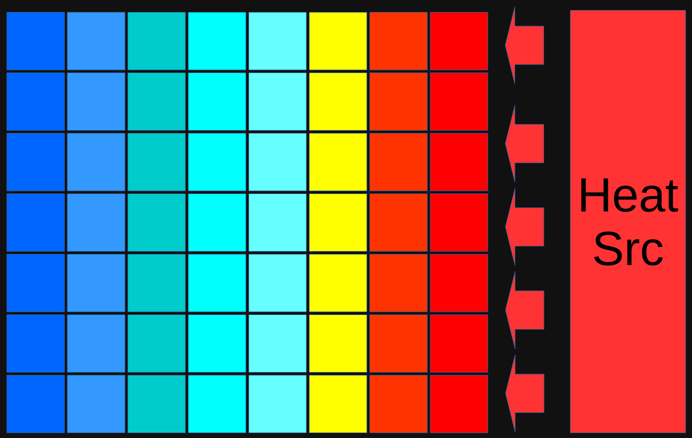
- Static interactions
  - Tasks and their interactions are known before execution
  - Algorithms are easy to design
- Dynamic Interactions
  - Timing and interacting tasks are unknown before execution
  - Algorithms difficult to design

Regular vs Irregular Interactions

- Regular Interactions:
  - Interactions have a pattern that can be described with a function
    - Examples: mesh, ring
  - Regular patterns can be exploited for efficient implementation
    - Schedule communication to avoid conflicts on network links
- Irregular Interactions:
  - Lack a well defined topology
  - Modeled by a graph

Regular vs Irregular Interactions cont'd

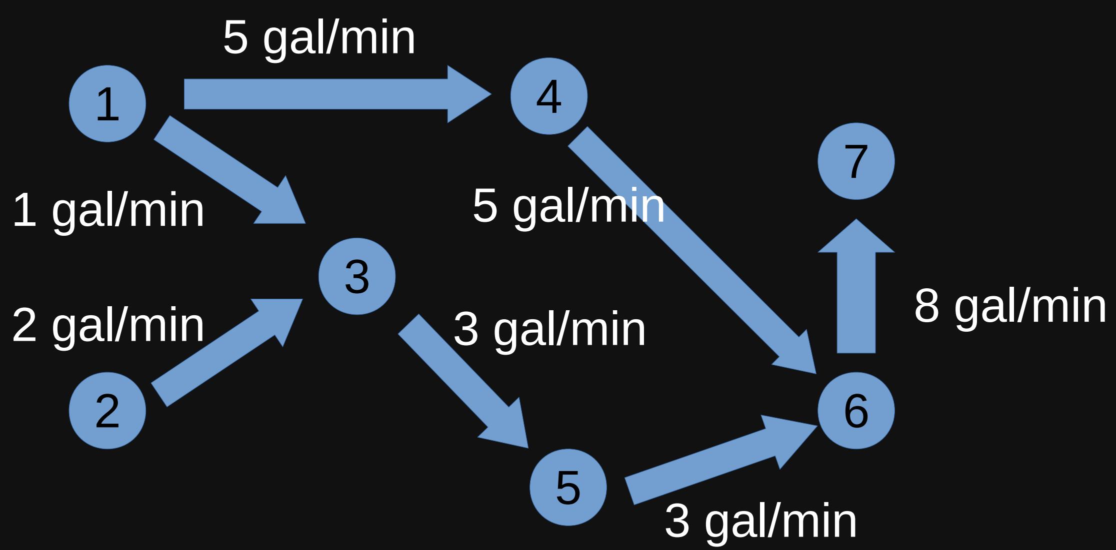
- Example of regular interaction: Heat propagation of a metal plate



The temperature of a cell t[i,j] = ½ t[i, j+1], i.e., a cell's temperature is half of the temperature of its adjacent cell

Regular vs Irregular Interactions cont’d

- Example of irregular communication: water flow in a sewer systems (an arrow represents a pipe and a circle represents an intersection)



Read-only vs Read-write Interactions

- Read-only interactions
  - Tasks only read data from other tasks
- Read-write interactions
  - Read and write data of other tasks
  - More difficult to code, requires synchronization to void multiple tasks writing to one data at the same time

One-sided vs Two-sided Interactions

- One-sided
  - Initiated and completed by only one task. Usually requires one of the following functions to implement,
    - READ or GET
    - WRITE or PUT
- Two-sided
  - Both tasks coordinate in an interaction. Usually requires two functions to implement
    - SEND and RECEIVE

Mapping Techniques for Load Balancing

Mapping Techniques

- Mapping:
  - Assign concurrent tasks to threads for execution
  - Assign concurrent threads to processors/cores for execution
  - Essentially, assigning tasks to processors/cores
- Overheads from (bad) mappings
  - Serialization (idling)
  - Communication
- Goal of Mapping
  - Optimize performance and minimize overheads
- Conflicting objectives:
  - Reduce communication ==> increase idling
  - Reduce idling ==> increase communication
  - Good mapping find a sweet point between idling and communication

Mapping to Minimize Idling

- Should try to minimize idling and balance load simultaneously
- Balancing load does not automatically minimize idling
  - Tasks sizes are hard to know
  - Other overheads: task scheduling, communication etc.

Mapping to Minimize Idling

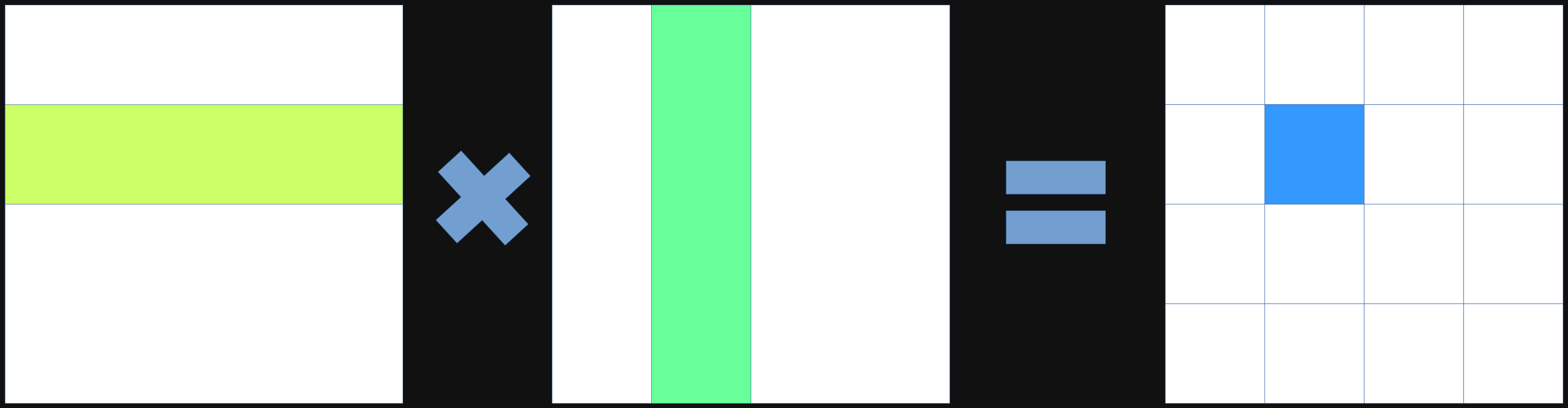
- Static mapping:
  - Mapping tasks to threads/processors before execution
  - Requirements: a good estimation of task sizes
  - Finding the optimal mapping is NP hard (similar to bin packing problem)
- Dynamic mapping:
  - Map tasks to threads/processors during execution
  - Why
    - Tasks are generated at run-time
    - Tasks sizes are unknown (usually true)

Schemes for Static Mapping

- Data partitioning
- Task graph partitioning
- Hierarchical strategies

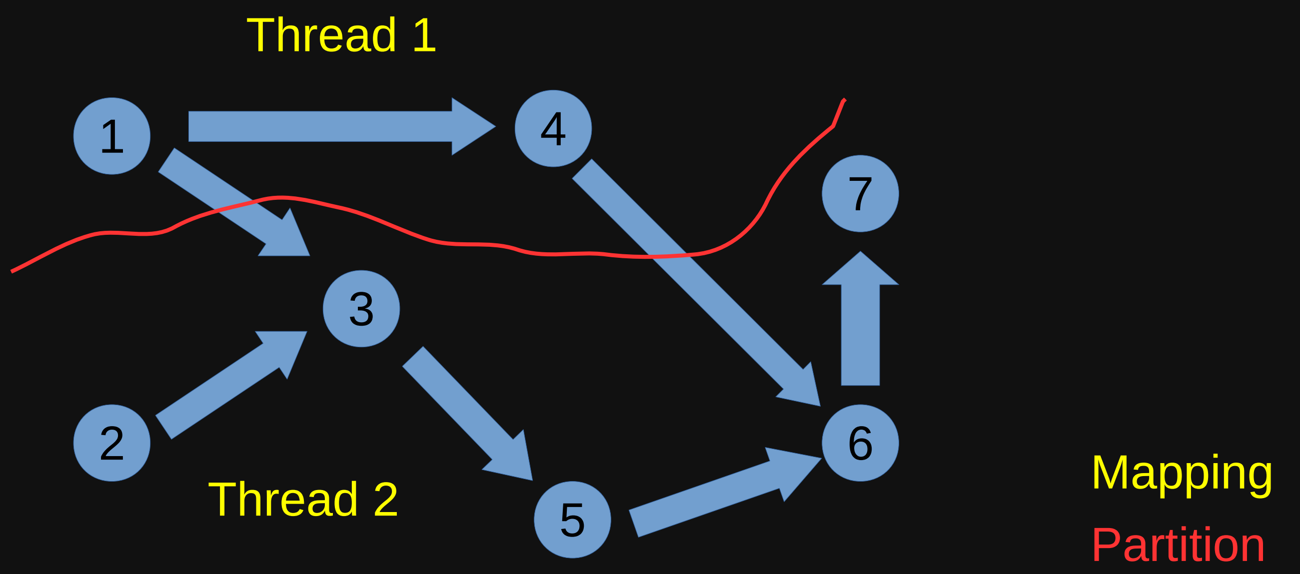
Mapping Based on Data Partitioning

- Similar to data based decomposition – assign a chunk of data and its computation to one thread/processor
- Example: Matrix multiplication



Mappings Based on Task Graph

- Partition tasks in task dependency graph, each partition is mapped to one thread/processor
- Example: Water flow in sewer pipes



Mappings Based on Task Graph

- Optimal partitioning for general task-dependency graph
  - NP-hard problem
  - Excellent heuristics exist for structured graphs

Hierarchical Mapping

- Sometimes a single-level mapping is inadequate
- Hierarchical approach
  - use a task mapping at the top level
  - data partitioning within each task

Schemes for Dynamic Mapping

- Dynamic mapping, a.k.a., dynamic load balancing
  - Load balancing is the primary motivation for dynamic mapping
- Styles
  - Centralized
  - Distributed

Centralized Dynamic Mapping

- Threads types: masters or slaves
- General strategy
  - when a slave runs out of work → request more from master
- Advantage
  - Easy to implement
- Disadvantage
  - master may become bottleneck for large # of threads
- Approach
  - chunk scheduling: thread picks up several of tasks at once
    - large chunk sizes may cause significant load imbalances
    - gradually decrease chunk size as the computation progresses

Distributed Dynamic Mapping

- All threads as peers
- Each thread can send or receive work from other threads
  - avoids centralized bottleneck
  - Hard to implement
- Four critical design questions
  - how are sending and receiving threads paired together?
  - who initiates work transfer?
  - how much work is transferred?
  - when is a transfer triggered?
- Ideal answers can be application specific
- The most popular distributed dynamic mapping: “work stealing”
  - An idle thread steal work/task from another busy thread

Methods for Minimizing Interaction Overheads

Minimizing Interaction Overheads: Principles

- Maximize data locality
  - don’t fetch data you already have
  - restructure computation to reuse data promptly
- Minimize volume of data exchange
  - partition dependency graph to minimize edge crossings
- Minimize frequency of communication
  - try to aggregate messages where possible
- Minimize contention and hot-spots
  - use decentralized techniques (avoidance)

Minimizing Interaction Overheads: Techniques

- Overlap communication with computation
  - For one thread on each processor, non-blocking communication primitives/functions
    - overlap communication with your own computation
    - prefetch remote data to hide latency
  - For multiple threads share one processor
    - Schedule threads waiting for communication out-of processor, and schedule other threads to run on the processor
- Replicate data or computation to reduce communication
- Use group communication instead of point-to-point primitives
- Issue multiple communications and overlap their latency (reduces exposed latency)

Hardware Consideration for Mapping and Communication

Hardware Considerations for Mapping and Communication

- In practice, hardware adds additional constraints for mapping and communication
  - It is common that hardware is the primary reason that a mapping/communication strategy is chosen
- Examples of hardware constraints
  - Differences in the computation power of processors
    - More powerful processors handle more tasks
  - Differences of inter-processors/cores connections
    - Fast: shared-cache/DRAM
    - Median: On-board (motherboard) inter-processor connections
    - Slow: LAN/network
  - Processors connection topology, e.g., mesh or ring
    - Minimize communication distance
    - Avoid congestion
  - Resource contention
    - Contention for shared cache space
    - Contention for shared memory bandwidth

Parallel Algorithm Model

Parallel Algorithm Model

- Definition: ways of structuring a parallel algorithm
- Aspects of a model
  - decomposition
  - mapping technique
  - strategy to minimize interactions

Common Parallel Algorithm Templates

- Data parallel
  - each task performs similar operations on different data
  - typically statically map tasks to threads or processes
- Task graph
  - use task dependency graph relationships to promote locality, or reduce interaction costs
- Master-slave
  - one or more master threads generate work
  - allocate it to worker threads
  - allocation may be static or dynamic
- Pipeline / producer-consumer
  - pass a stream of data through a sequence of workers
  - each performs some operation on it
- Hybrid
  - apply multiple models hierarchically, or
  - apply multiple models in sequence to different phases

Summary of Parallel Algorithm Design

- Basic Concepts
  - Task dependency graph
  - Degree of concurrency, granularity, critical path, limits on parallel performance
  - Tasks, threads, processors and mapping
  - Metrics: speedup and parallel efficiency
- Characteristics of tasks and interactions
- Decomposition Techniques
- Mapping Techniques
- Minimizing Communication/Interaction Techniques
- Hardware considerations for mapping and communication
- Parallel Models

Summary of Parallel Algorithm Design cont'd

- Basic Concepts
- Characteristics of Tasks and Interactions
  - Characteristics: statically/dynamically generated, data size, computation size
  - Interactions: static vs dynamic, regular vs irregular, read-only vs read-write, one-sided vs two-sided
- Decomposition Techniques
- Mapping Techniques
- Minimizing Communication/Interaction Techniques
- Hardware considerations for mapping and communication
- Parallel Models

Summary of Parallel Algorithm Design cont'd

- Basic Concepts
- Characteristics of Tasks and Interactions
- Decomposition Techniques
  - Recursive
  - Data
  - Exploratory
  - Speculative
  - Hybrid
- Mapping Techniques
- Minimizing Communication/Interaction Techniques
- Hardware considerations for mapping and communication
- Parallel Models

Summary of Parallel Algorithm Design cont'd

- Basic Concepts
- Characteristics of Tasks and Interactions
- Decomposition Techniques
- Mapping Techniques
  - Static
  - Dynamic
  - Hierarchical
- Minimizing Communication/Interaction Techniques
- Hardware considerations for mapping and communication
- Parallel Models

Acknowledgement

- Slides based on John Mellor-Crummey's Parallel Computing class at Rice University