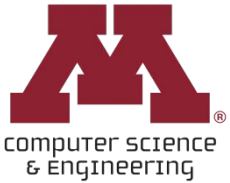
CSCI 5451: Introduction to Parallel Computing

Lecture 4: Parallel Algorithm Design



UNIVERSITY OF MINNESOTA
Driven to Discover®

Announcement (09/15)

- Office Hours are posted to the course site
- Homework Testing
  - We will be using two clusters (plate and cuda) for all homework assignments in this course
  - Examine this pdf and corresponding code to test that you are able to use these machines going forward (these are simple 'hello world' style tests)
  - Doing this in advance gives us the chance to fix any problems you may have with connecting/running programs before we get closer to homework deadlines

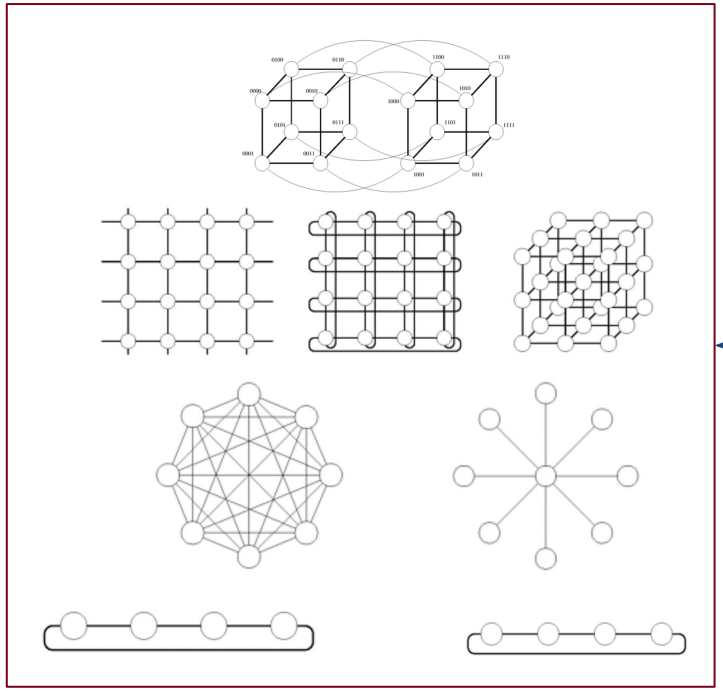
Lecture Overview

- Recap
- Task Decomposition
  - Background
  - MatVec Example Decompositions
  - Metrics & Definitions
- Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)
- Classifying Task Interaction & Generation

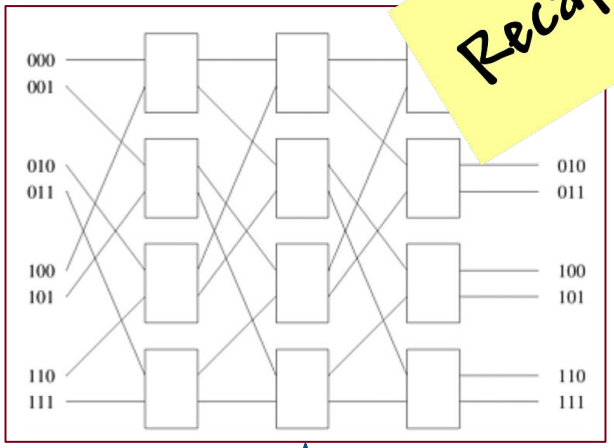
Lecture Overview

- **Recap**
- Task Decomposition
  - Background
  - MatVec Example Decompositions
  - Metrics & Definitions
- Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)
- Classifying Task Interaction & Generation

Recap



Topologies



Omega Networks

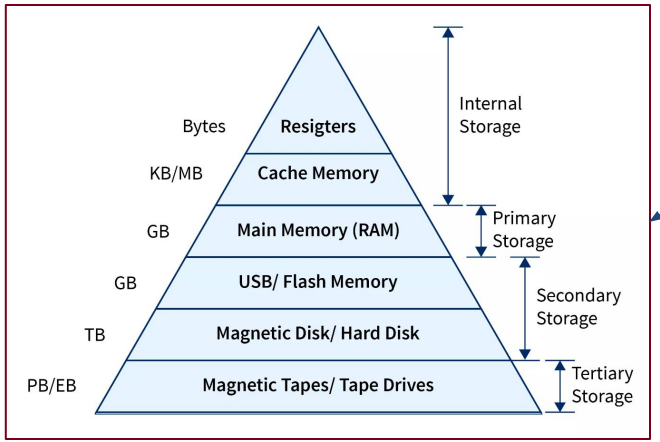
Recap

Recap

Topology Metrics

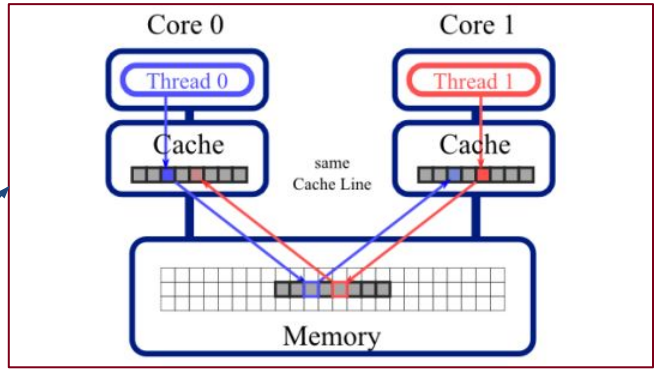
Table 2.1 A summary of the characteristics of various static network to nodes.

Network	Diameter	Bisection Width	Arc Connectivity	(N. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp



Memory Basics

False Sharing



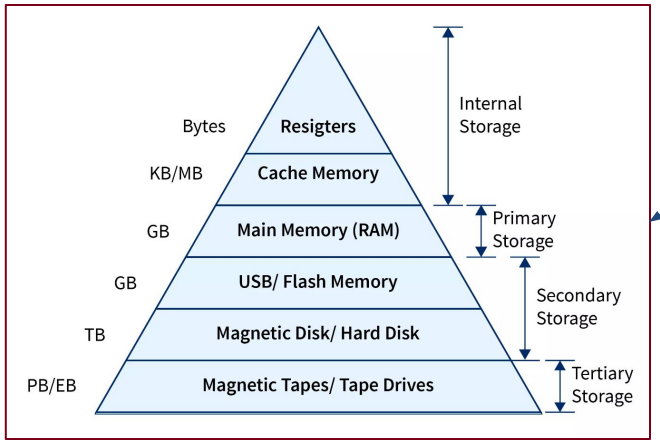
Recap

Recap

Topology Metrics

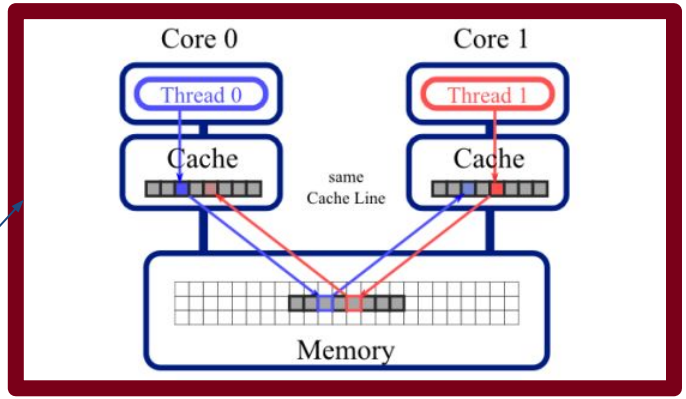
Table 2.1 A summary of the characteristics of various static network to nodes.

Network	Diameter	Bisection Width	Arc Connectivity	(N. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

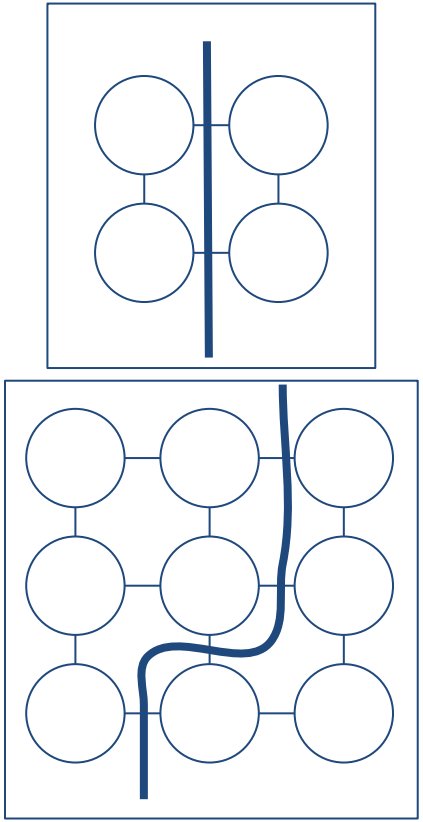


Memory Basics

False Sharing



Recap (Revisiting Mesh Bisection Width)



General Case

- (√p) mod 2 == 0
  - Bisection width = √p
- (√p) mod 2 == 1
  - Bisection width = √p + 1

Lecture Overview

- Recap
- **Task Decomposition**
  - **Background**
  - MatVec Example Decompositions
  - Metrics & Definitions
- Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)
- Classifying Task Interaction & Generation

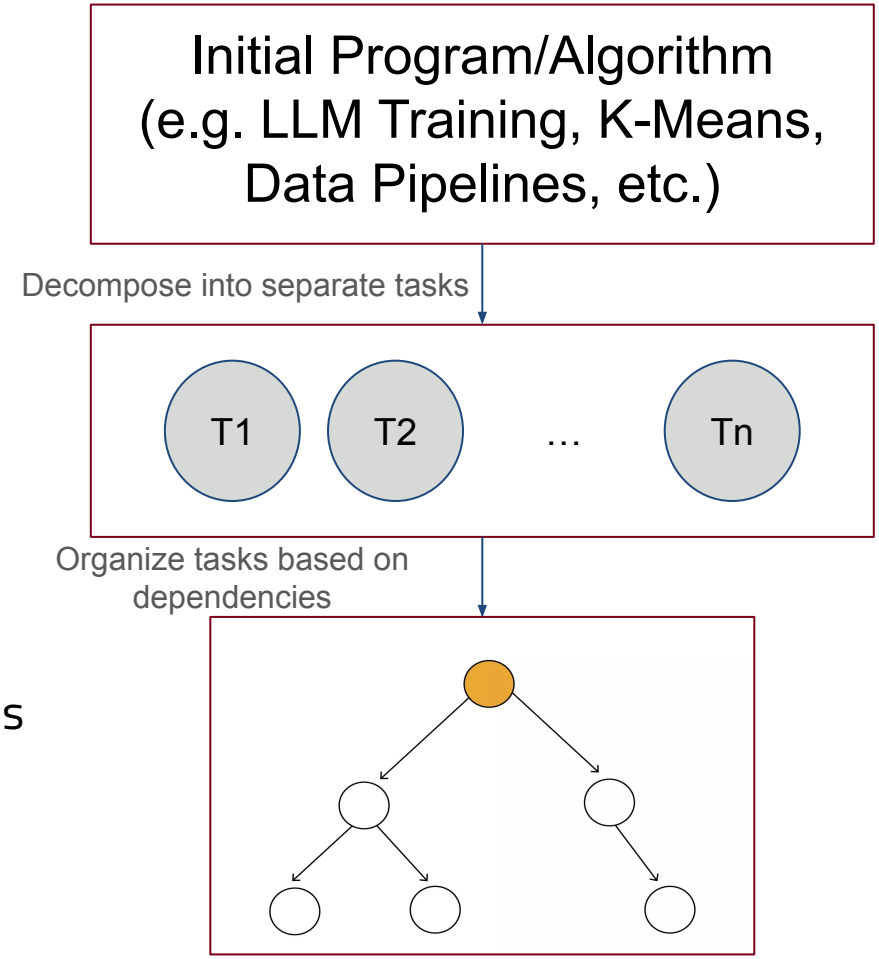
How do we map from a serial program into some parallel program?

How do we map from a serial program into some parallel program?

Decompose the program into smaller tasks, then find out how to organize them.

Task Decomposition

- We have to break down the tasks individually and determine their dependencies
- What are tasks?
  - Atomic units of computation within our program
  - We as programmers decide what these tasks are
  - Once we define a task, we assume that it is both serial and indivisible



Lecture Overview

- Recap
- **Task Decomposition**
  - Background
  - **MatVec Example Decompositions**
  - Metrics & Definitions
- Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)
- Classifying Task Interaction & Generation

Matrix Vector Multiplication

$$A * x = b$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Matrix Vector Multiplication

$$A * x = b$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Simplest decomposition: Don’t decompose any tasks at all. This is the same as serial execution. No parallel speedups = bad task decomposition.

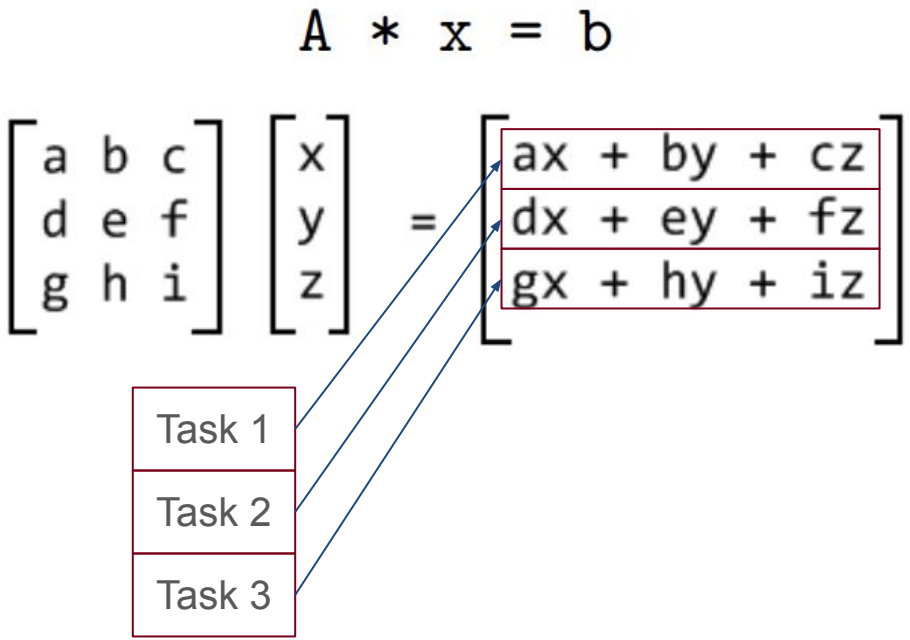
Matrix Vector Multiplication

$$A * x = b$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

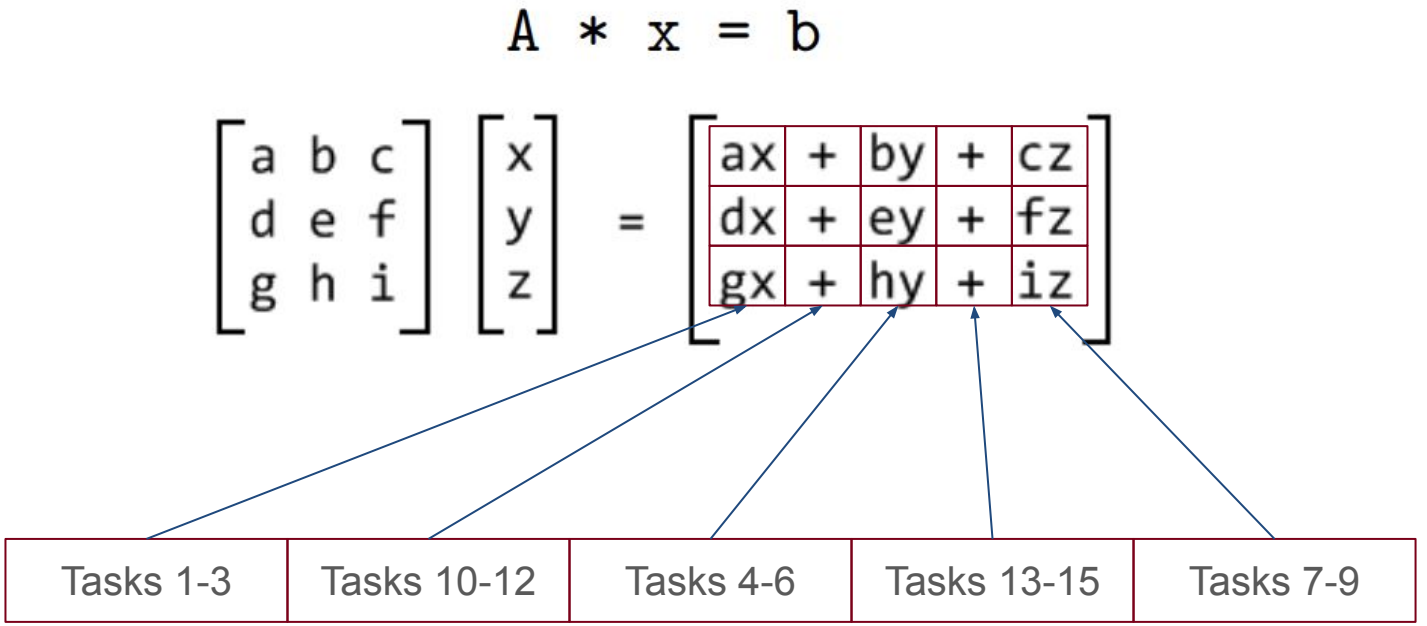
Other Decomposition Ideas?

Matrix Vector Multiplication



Matrix Vector Multiplication

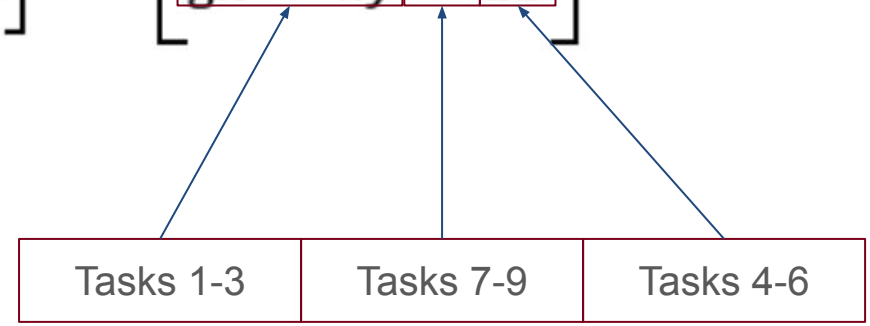
Most Granular*



Matrix Vector Multiplication

$$A * x = b$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$



Task Dependency Graph (TDG)

- Once we break things down into smaller tasks, we need to build out how we should execute them
- There can be *more than one* way to combine the tasks
- Each node is a task, arrows represent dependencies
- Each node will oftentimes have the amount of work required attached to them (typically represented by the number of atomic operations performed within that task - add, subtract, multiply, divide)
- We assume, within each task, that work is serial

Task Dependency Graph (TDG)

- Once we break things down into smaller tasks, we need to build out how we should execute them
- There can be *more than one* way to combine the tasks
- Each node is a task, arrows represent dependencies
- Each node will oftentimes have the amount of work required attached to them (typically represented by the number of atomic operations performed within that task - add, subtract, multiply, divide)
- We assume, within each task, that work is serial

Why might this be a poor way of measuring work?

Task Dependency Graph (TDG)

- Once we break things down into smaller tasks, we need to build out how we should execute them
- There can be *more than one* way to combine the tasks
- Each node is a task, arrows represent dependencies
- Each node will oftentimes have the amount of work required attached to them (typically represented by the number of atomic operations performed within that task - add, subtract, multiply, divide)
- We assume, within each task, that work is serial

For our purposes, we will assume all work is equal until later on in the course.

Matrix Vector Multiplication (TDG)

$$A * x = b$$

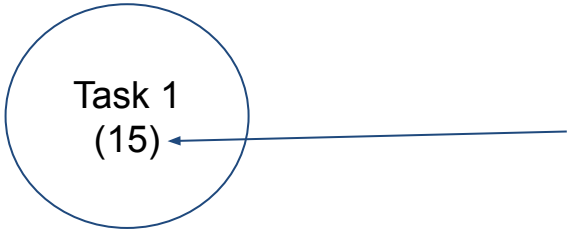
$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

How to represent this decomposition as a TDG?

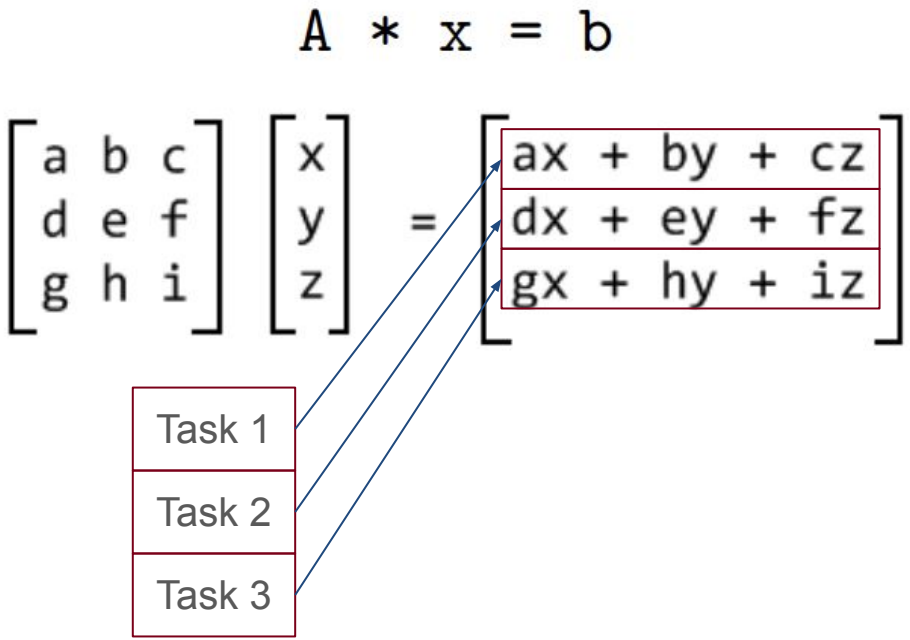
Matrix Vector Multiplication (TDG)

$$A * x = b$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$



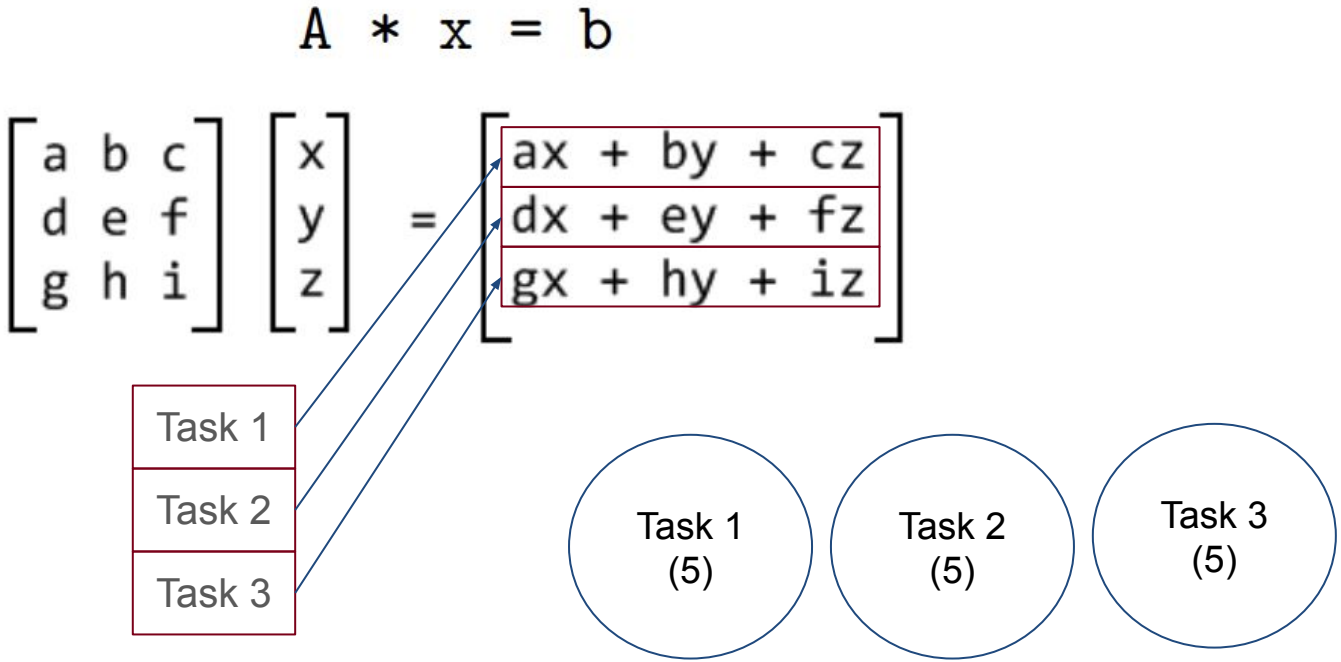
Matrix Vector Multiplication (TDG)



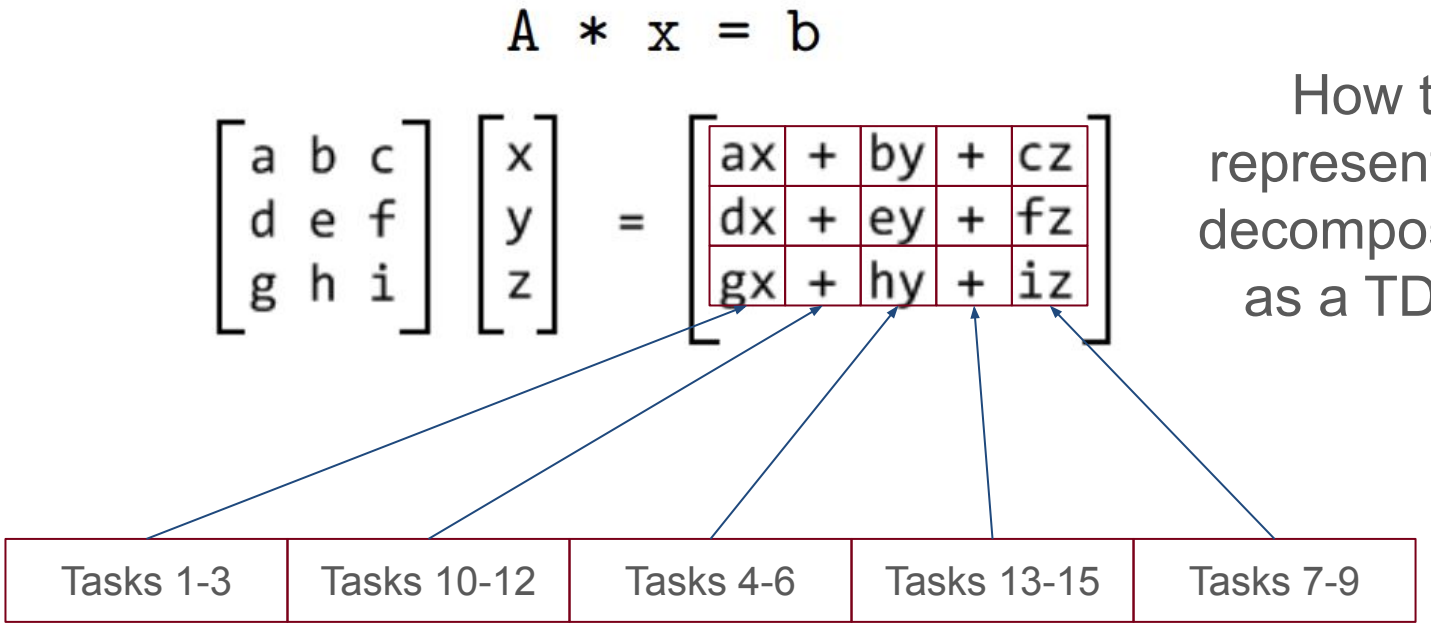
How to represent this decomposition as a TDG?

Matrix Vector Multiplication (TDG)

No edges because there are no dependencies between tasks

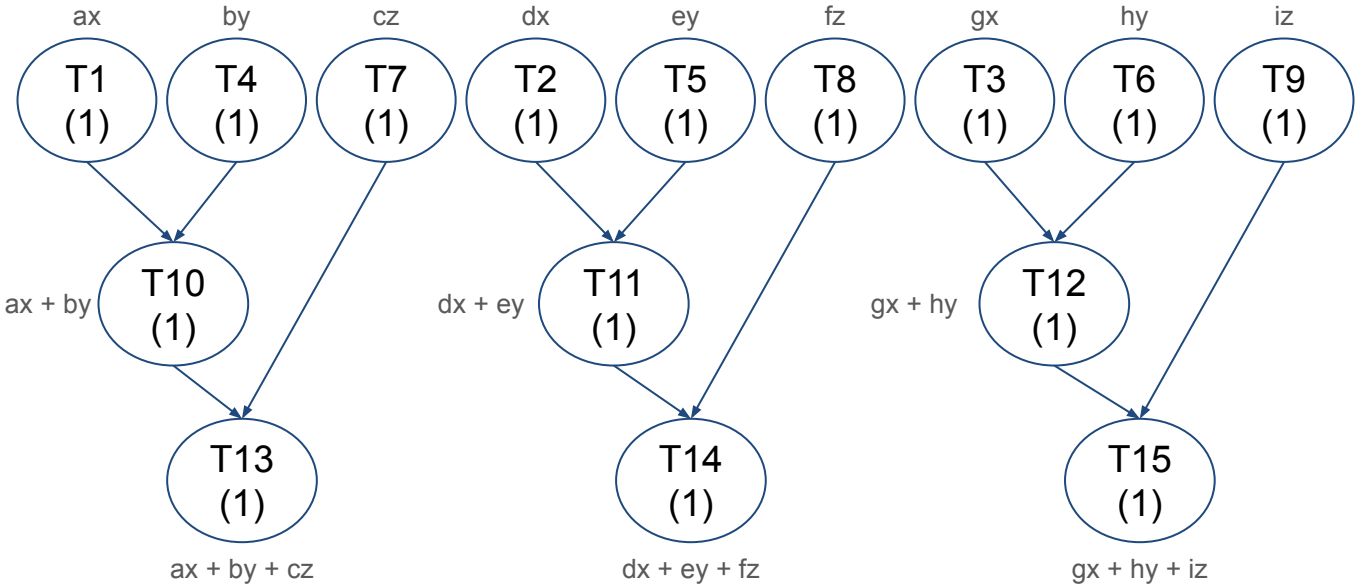


Matrix Vector Multiplication

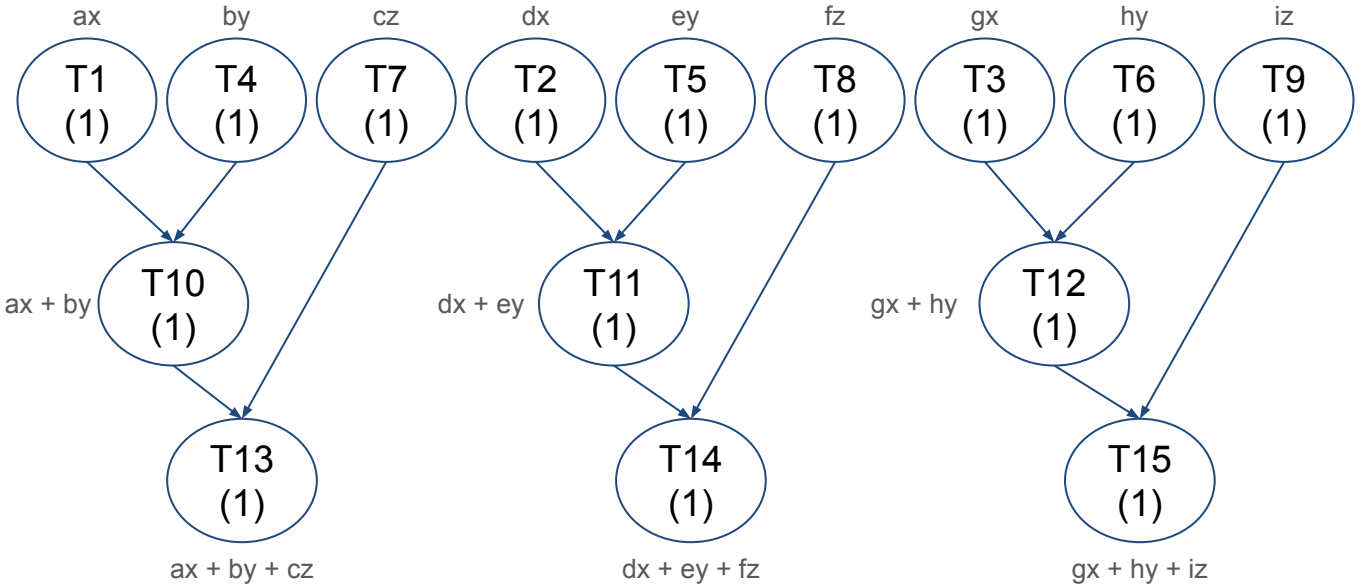


How to represent this decomposition as a TDG?

Matrix Vector Multiplication



Matrix Vector Multiplication



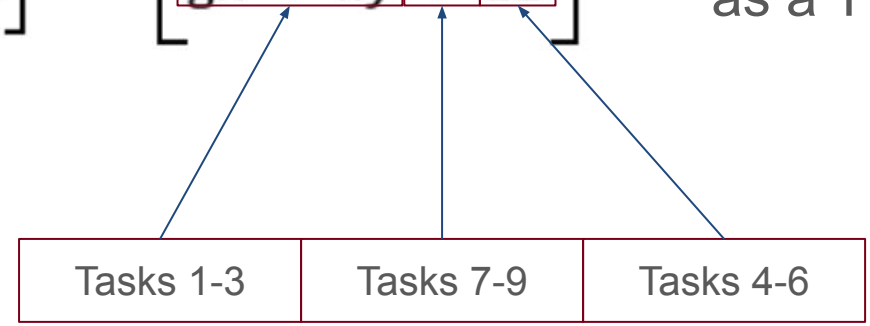
Alternative Decompositions?

Matrix Vector Multiplication

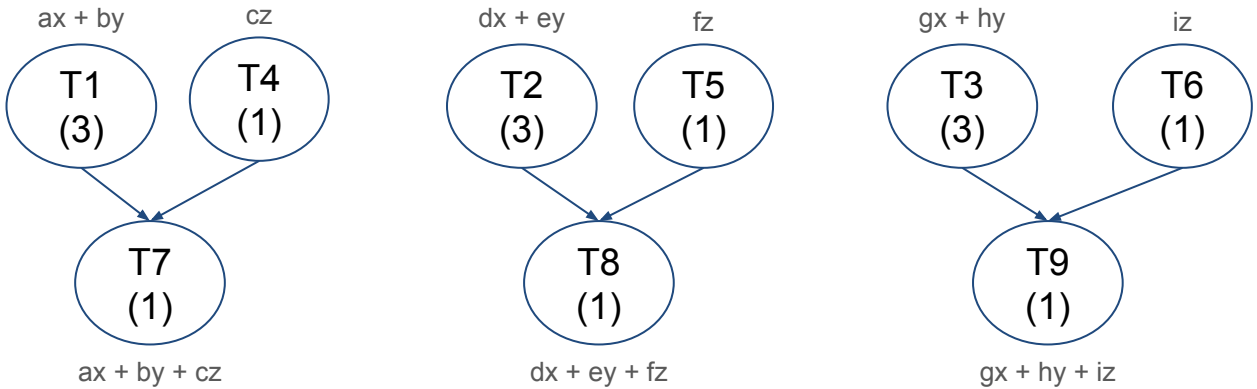
$$A * x = b$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

How to represent this decomposition as a TDG?



Matrix Vector Multiplication

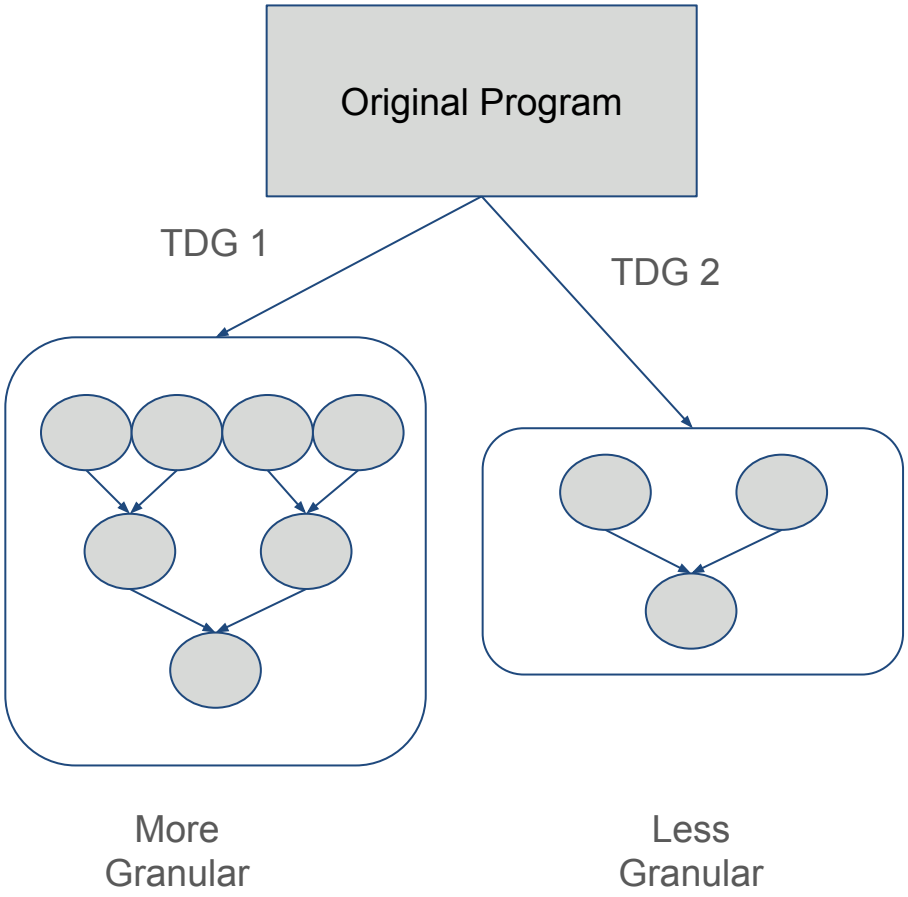


Lecture Overview

- Recap
- **Task Decomposition**
  - Background
  - MatVec Example Decompositions
  - **Metrics & Definitions**
- Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)
- Classifying Task Interaction & Generation

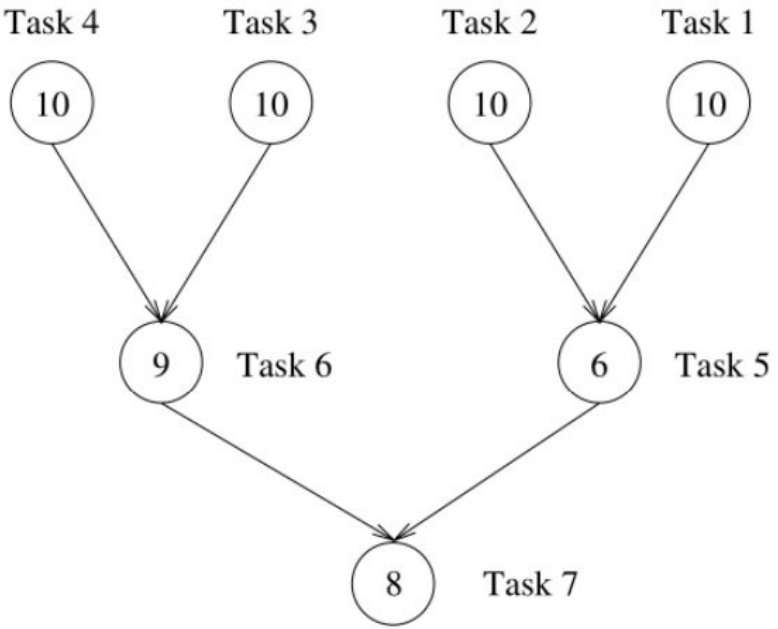
Granularity

- Defines how fine-grain the parallel program has been decomposed
- More granular = smaller tasks with less work per task
- Less granular = larger tasks with more work per task



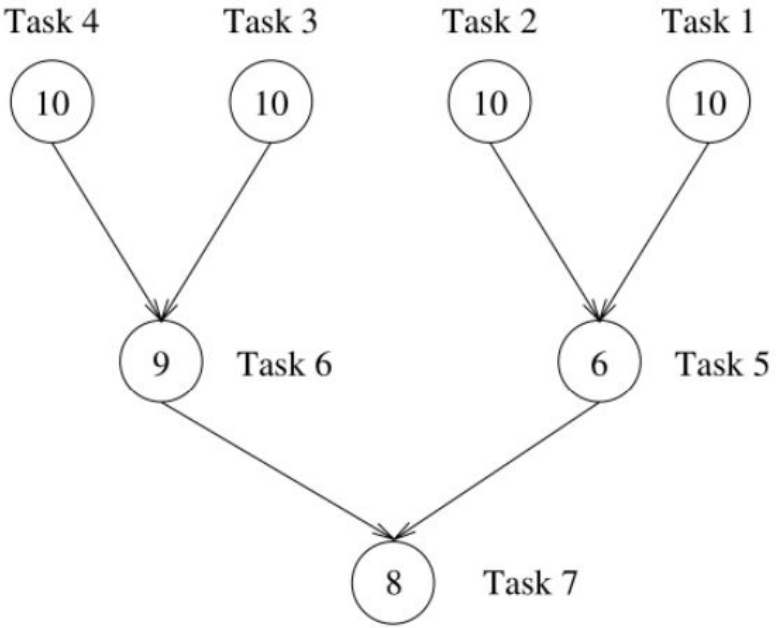
Total Work

Sum of all work completed over all tasks



Total Work

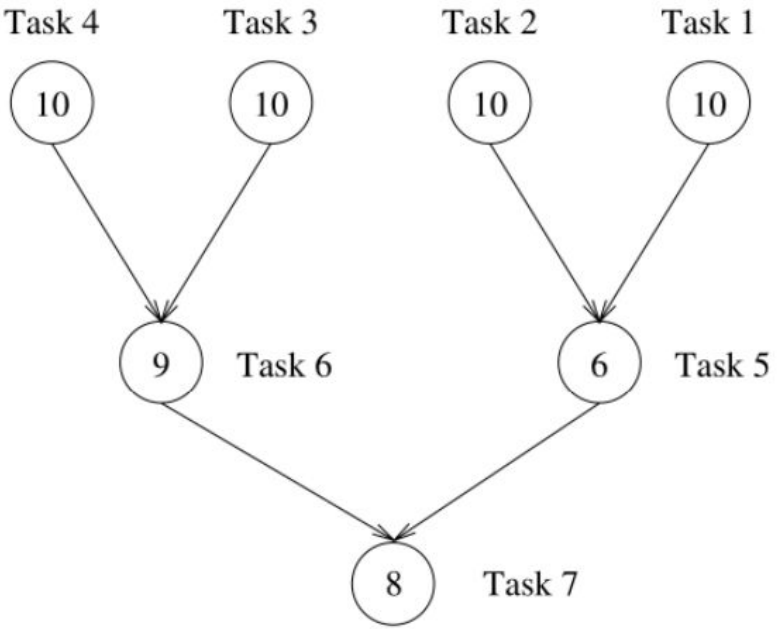
Sum of all work completed over all tasks



Total Work = 63

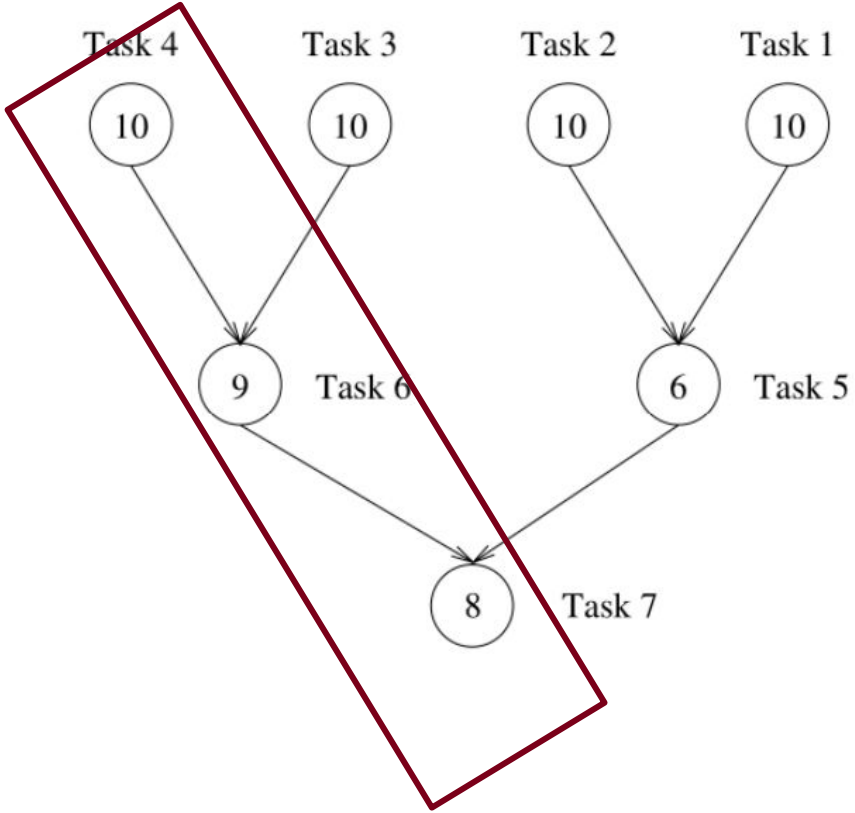
Critical Path Length

Largest amount of *sequential* work which must be performed in order to complete program execution



Critical Path Length

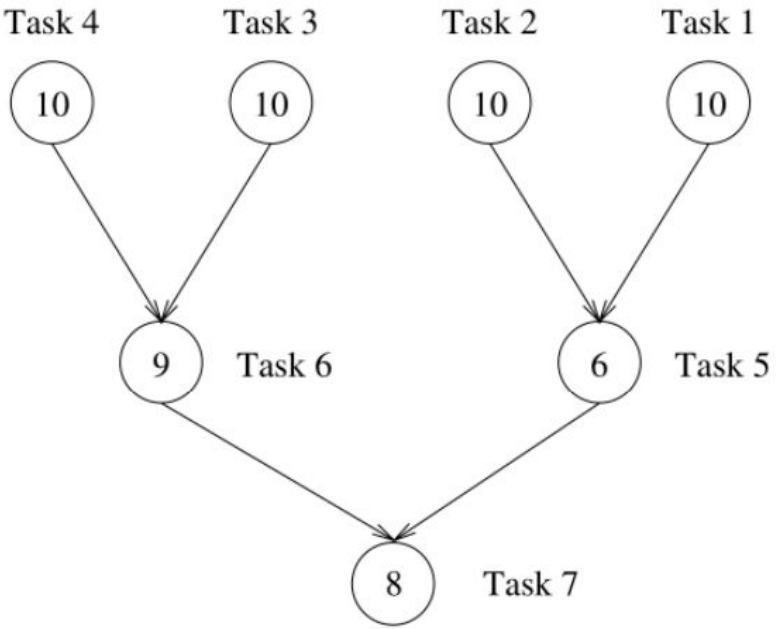
Largest amount of *sequential* work which must be performed in order to complete program execution



Critical Path Length = 27

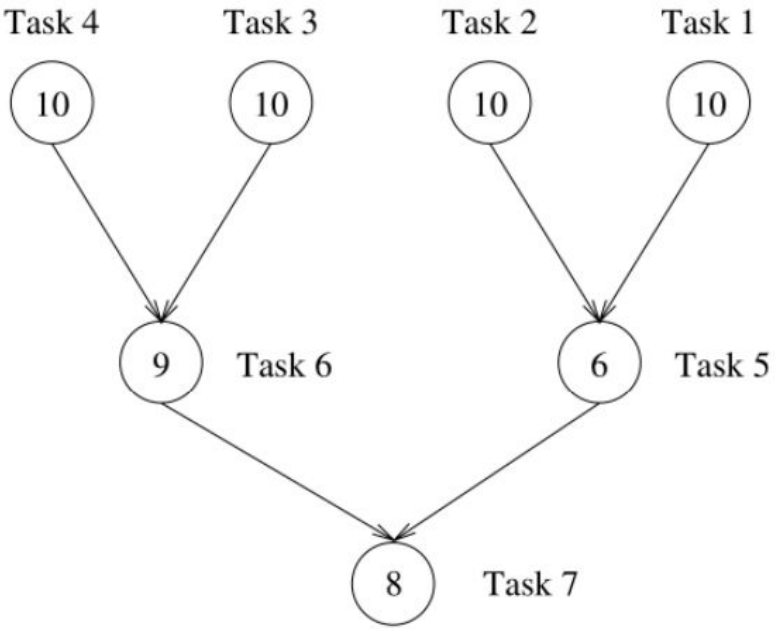
Concurrency

- Maximum Concurrency
  - Largest number of tasks which can be completed concurrently
  - The largest *width* of the TDG
- Average Concurrency
  - Total Work/Critical Path Length
  - This term usually sets an upper bound on parallel speedups



Concurrency

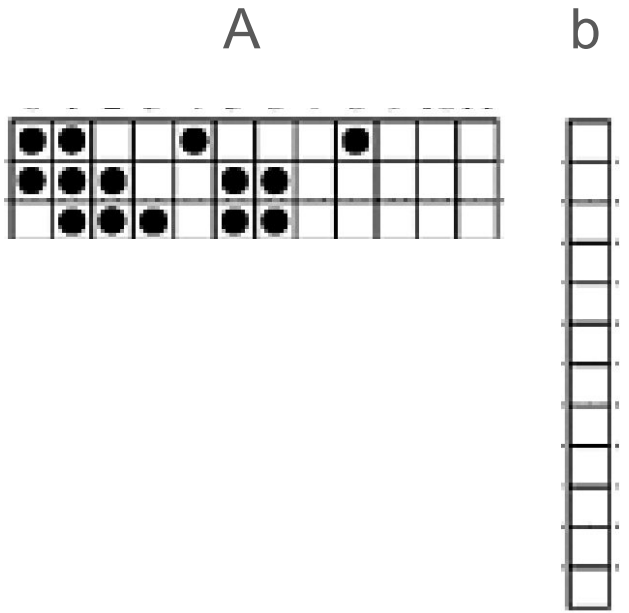
- Maximum Concurrency
  - Largest number of tasks which can be completed concurrently
  - The largest *width* of the TDG
- Average Concurrency
  - Total Work/Critical Path Length
  - This term usually sets an upper bound on parallel speedups



Maximum Concurrency = 4
Average Concurrency = 2.33

Sparse MatVec Example TDG + Metrics

- Assume
  - *A* is sparse
  - *b* is dense
- In Class Example
  - Decompose this into tasks
  - Create a Task Decomposition Graph (TDG) for these tasks
  - Define the following metrics for this TDG
    - ✔ Total Work
    - ✔ Critical Path Length
    - ✔ Average + Maximum Currency



Lecture Overview

- Recap
- Task Decomposition
  - Background
  - MatVec Example Decompositions
  - Metrics & Definitions
- **Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)**
- Classifying Task Interaction & Generation

What are some general strategies we can use for decomposing algorithms?

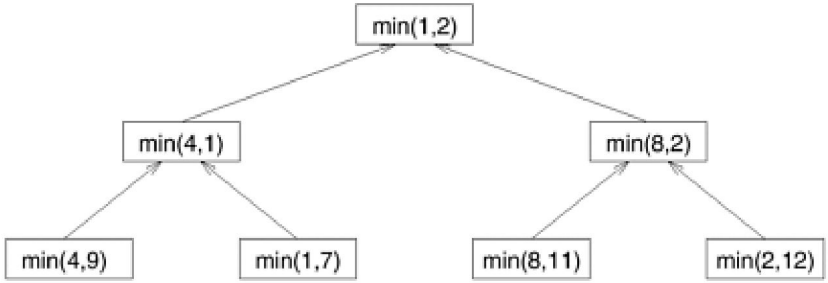
Recursive Decomposition

- Problems which can be solved via some divide + conquer strategies
- Subproblems can be recursively parallelized
- Partition tasks to each recursive call
- Examples
  - Finding the minimum element of an array
  - Quicksort Example

Recursive Decomposition

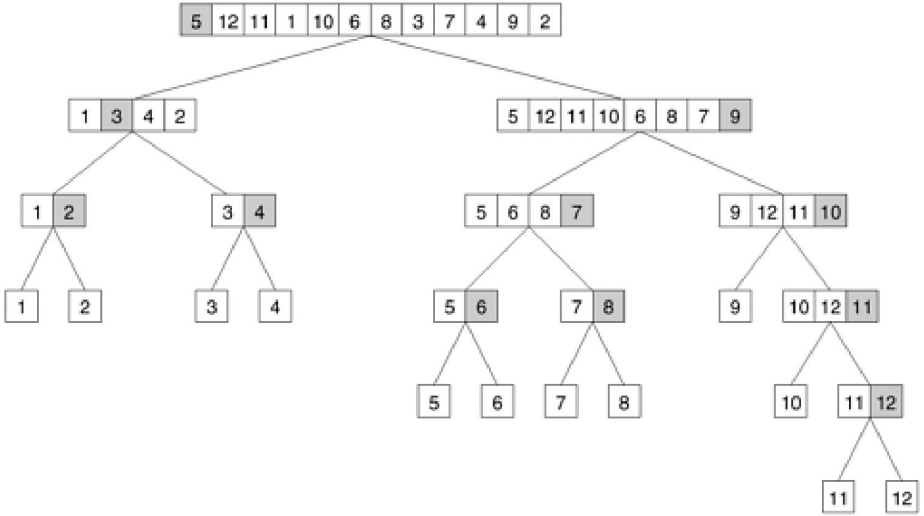
- Problems which can be solved via some divide + conquer strategies
- Subproblems can be recursively parallelized
- Partition tasks to each recursive call
- Examples
  - Finding the minimum element of an array
  - Quicksort Example

{4, 9, 1, 7, 8, 11, 2, 12}



Recursive Decomposition

- Problems which can be solved via some divide + conquer strategies
- Subproblems can be recursively parallelized
- Partition tasks to each recursive call
- Examples
  - Finding the minimum element of an array
  - Quicksort Example

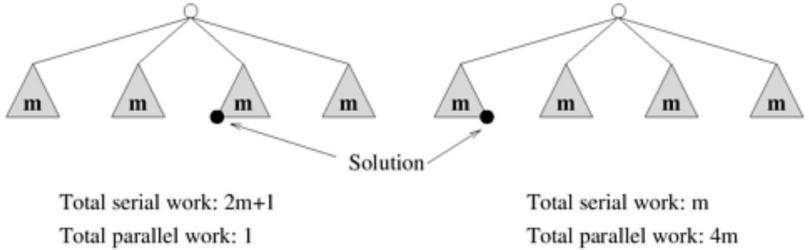


Exploratory Decomposition

- Use this for problems involving some kind of search
- Partition the search space among different processes
- Can lead to much anomalous speedups (slower or faster)
- Example
  - 15-puzzle search

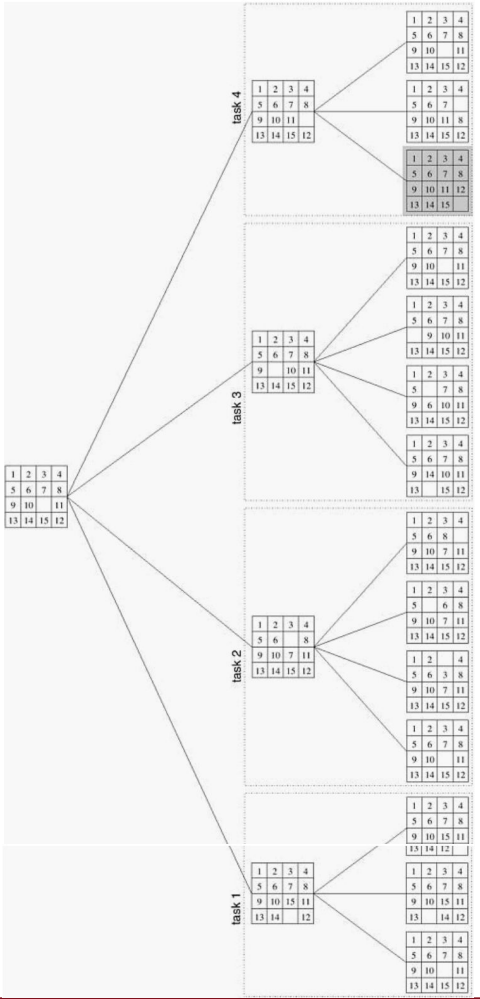
Exploratory Decomposition

- Use this for problems involving some kind of search
- Partition the search space among different processes
- Can lead to much anomalous speedups (slower or faster)
- Example
  - 15-puzzle search



Exploratory Decomposition

- Use this for problems involving some kind of search
- Partition the search space among different processes
- Can lead to much anomalous speedups (slower or faster)
- Example
  - 15-puzzle search



Data Decomposition (Output)

- Take the expected outputs of your program, then split them among your processes
- Assign tasks to each of these output elements

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{1,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{1,1} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \boxed{C_{1,1}} & \boxed{C_{1,2}} \\ \boxed{C_{2,1}} & \boxed{C_{2,2}} \end{pmatrix}$$

Data Decomposition (Intermediate)

- Many algorithms have multiple, sequential stages of computation
- First partition the problem into these intermediate computations
- Then assign tasks based on which processes *own* which computation

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \left(\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \right)$$

Stage II

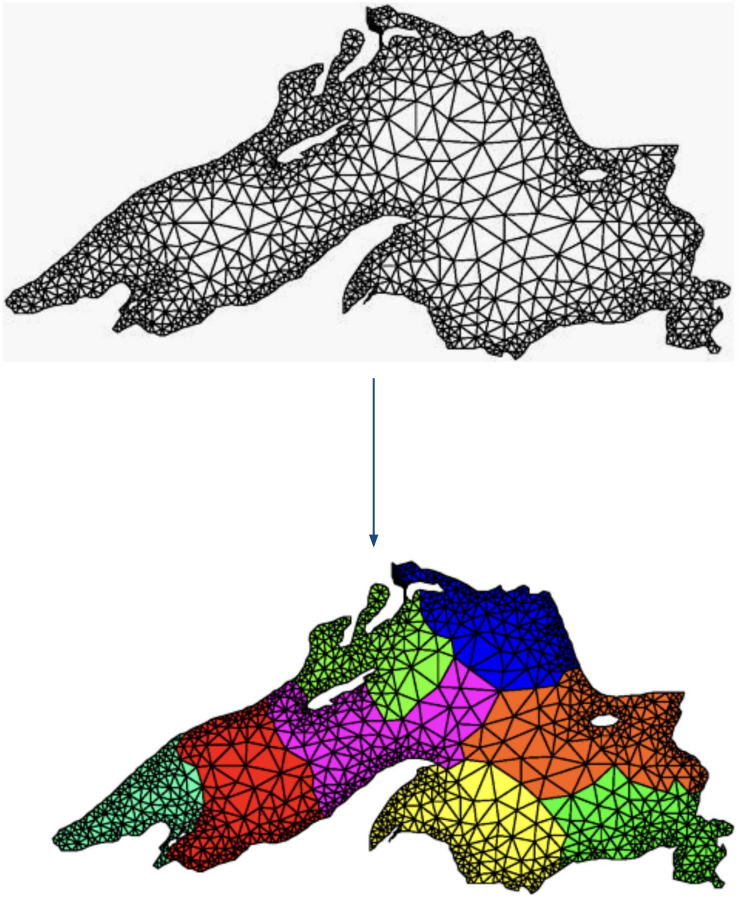
$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

$$\begin{aligned}
\text{Task 01:} \quad & D_{1,1,1} = A_{1,1}B_{1,1} \\
\text{Task 02:} \quad & D_{2,1,1} = A_{1,2}B_{2,1} \\
\text{Task 03:} \quad & D_{1,1,2} = A_{1,1}B_{1,2} \\
\text{Task 04:} \quad & D_{2,1,2} = A_{1,2}B_{2,2} \\
\text{Task 05:} \quad & D_{1,2,1} = A_{2,1}B_{1,1} \\
\text{Task 06:} \quad & D_{2,2,1} = A_{2,2}B_{2,1} \\
\text{Task 07:} \quad & D_{1,2,2} = A_{2,1}B_{1,2} \\
\text{Task 08:} \quad & D_{2,2,2} = A_{2,2}B_{2,2} \\
\text{Task 09:} \quad & C_{1,1} = D_{1,1,1} + D_{2,1,1} \\
\text{Task 10:} \quad & C_{1,2} = D_{1,1,2} + D_{2,1,2} \\
\text{Task 11:} \quad & C_{2,1} = D_{1,2,1} + D_{2,2,1} \\
\text{Task 12:} \quad & C_{2,2} = D_{1,2,2} + D_{2,2,2}
\end{aligned}$$

Data Decomposition (Input)

- First split the inputs of the problem into separate pieces so that each process will only have some subset of data
- Then assign tasks based on what data each of your processes stores
- Example:
  - Measuring temperatures at various positions in Lake Superior
  - Want to predict temperatures in the future



Speculative Decomposition

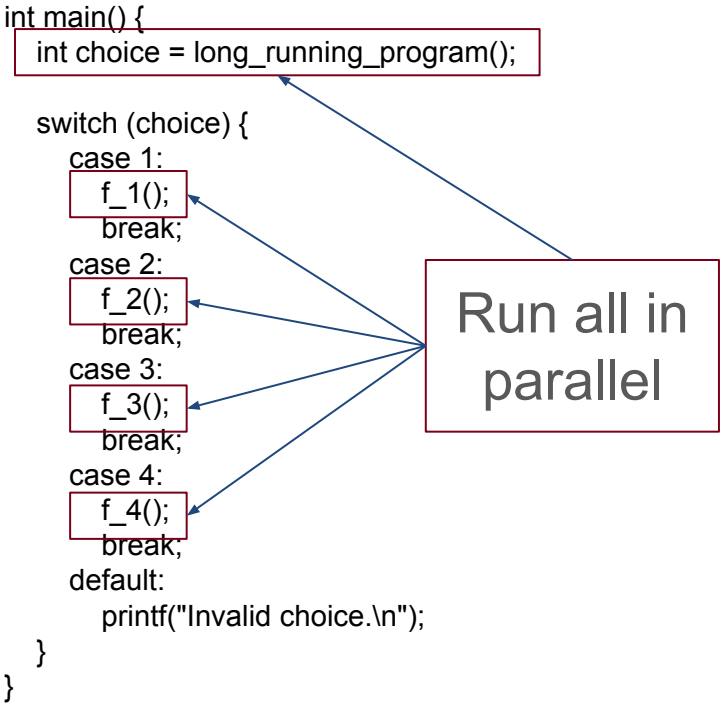
- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation

```
int main() {
    int choice = long_running_program();

    switch (choice) {
        case 1:
            f_1();
            break;
        case 2:
            f_2();
            break;
        case 3:
            f_3();
            break;
        case 4:
            f_4();
            break;
        default:
            printf("Invalid choice.\n");
    }
}
```

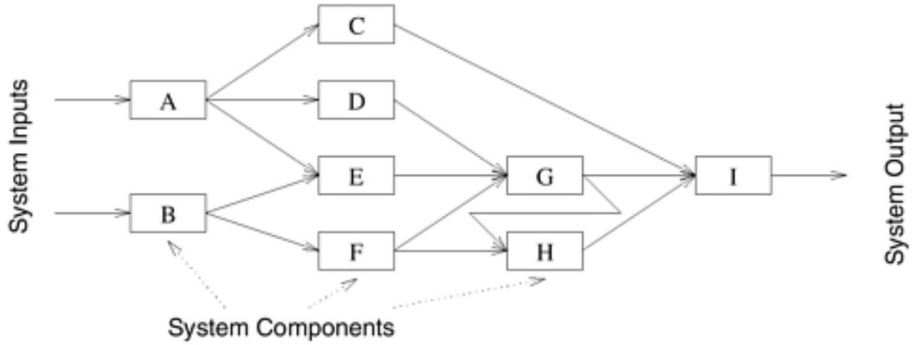
Speculative Decomposition

- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation



Speculative Decomposition

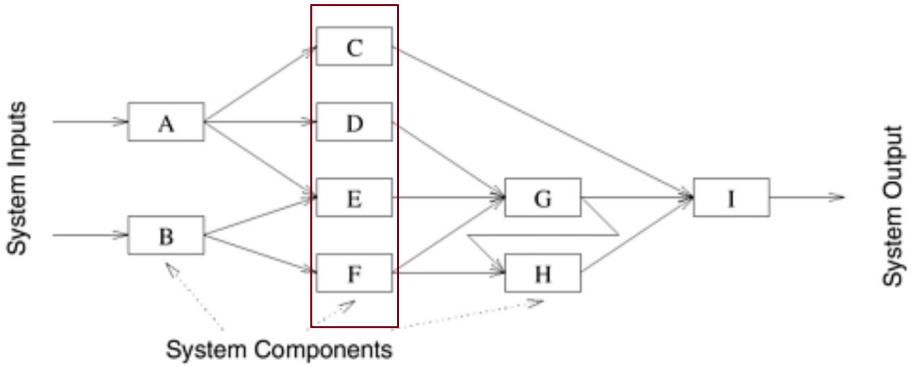
- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation



Speculative Decomposition

- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation

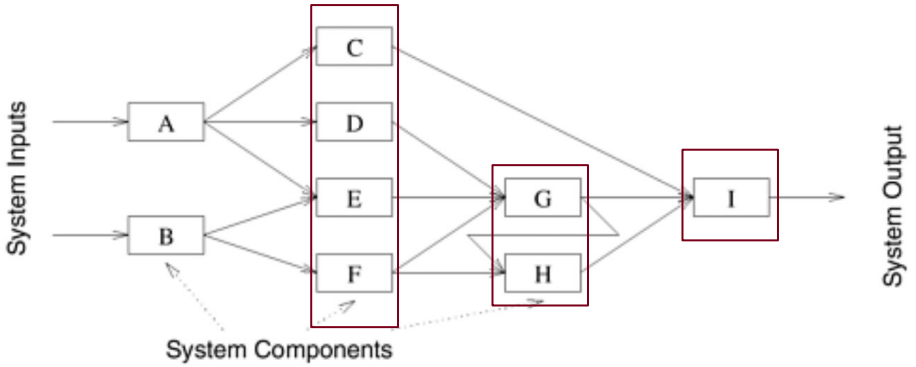
Run multiple conditional branches at a time



Speculative Decomposition

- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation

Run multiple conditional branches at a time

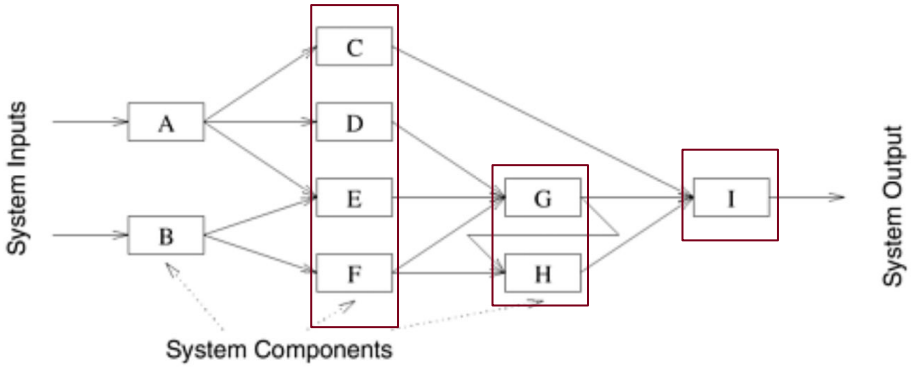


…we can also run computations at greater depth in parallel

Speculative Decomposition

- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation

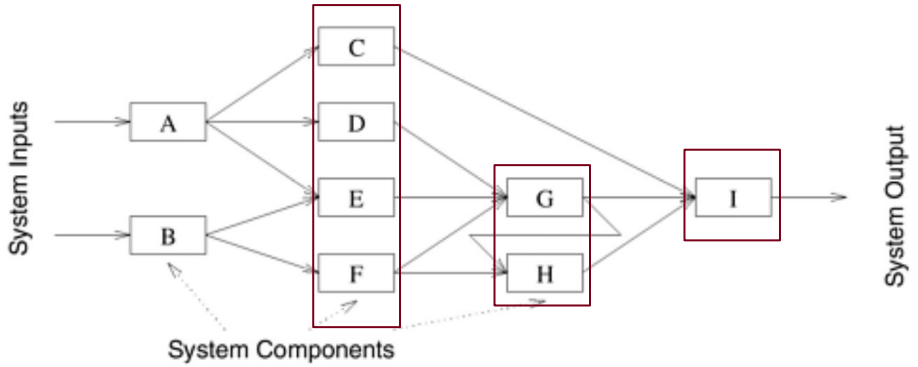
Under what conditions, can we not run each of the following tasks at the same time?



Speculative Decomposition

- Useful when a program has many conditional branches & computationally intensive statements to execute to determine the appropriate branch
- Examples
  - Simple Switch Statement
  - Discrete Event Simulation

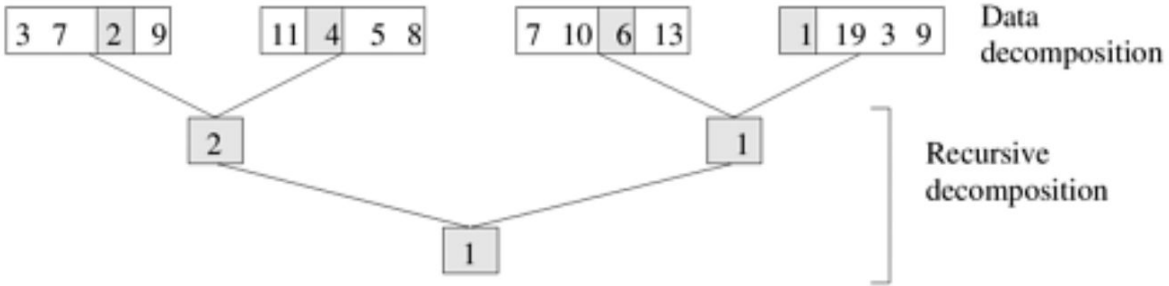
Under what conditions, can we not run each of the following tasks at the same time?



…when the data produced from earlier tasks is required for later ones (i.e. computing ***G*** is dependent on ***A***)

Hybrid Decomposition

- Combine multiple decomposition techniques at the same time
- More typical in practical settings
- Example (Min-Array)



Lecture Overview

- Recap
- Task Decomposition
  - Background
  - MatVec Example Decompositions
  - Metrics & Definitions
- Decompositions (Recursive, Exploratory, Data, Speculative, Hybrid)
- **Classifying Task Interaction & Generation**

How can we classify task generation & interaction?

Static vs. Dynamic Task Generation

- Static Task Generation: All tasks are known *exactly* before program execution
- Dynamic Task Generation: Tasks and the Task Dependency Graph are not known exactly before program execution

Static vs. Dynamic Task Generation

- Static Task Generation: All tasks are known *exactly* before program execution
- Dynamic Task Generation: Tasks and the Task Dependency Graph are not known exactly before program execution

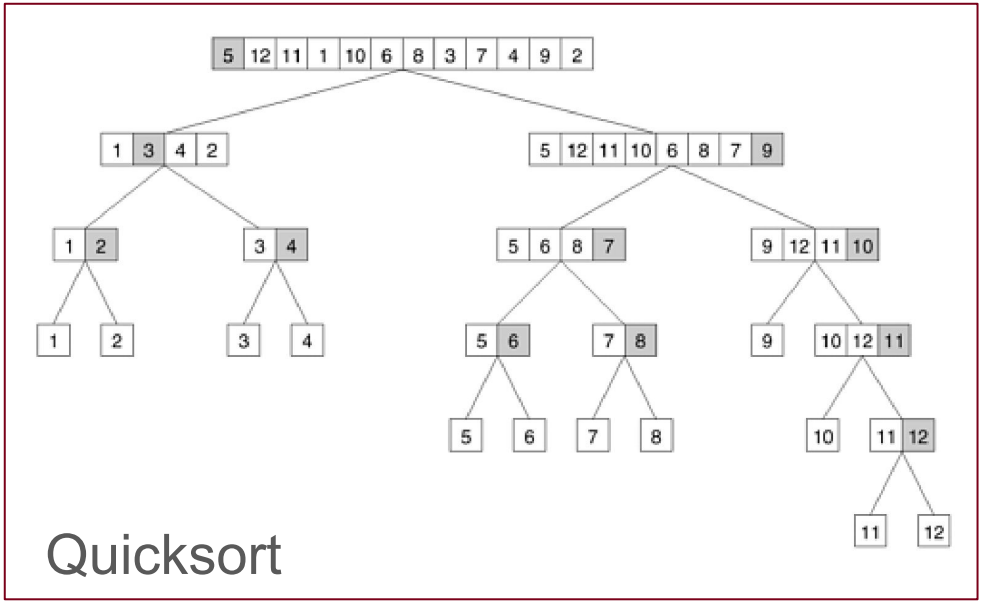
$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Static vs. Dynamic Task Generation

- Static Task Generation: All tasks are known *exactly* before program execution
- Dynamic Task Generation: Tasks and the Task Dependency Graph are not known exactly before program execution



Uniform vs. Non-Uniform Task Sizes

- Uniform: All tasks require the same amount of time/work to complete
- Non-Uniform: Tasks require significantly different amounts of time to complete

Uniform vs. Non-Uniform Task Sizes

- Uniform: All tasks require the same amount of time/work to complete
- Non-Uniform: Tasks require significantly different amounts of time to complete

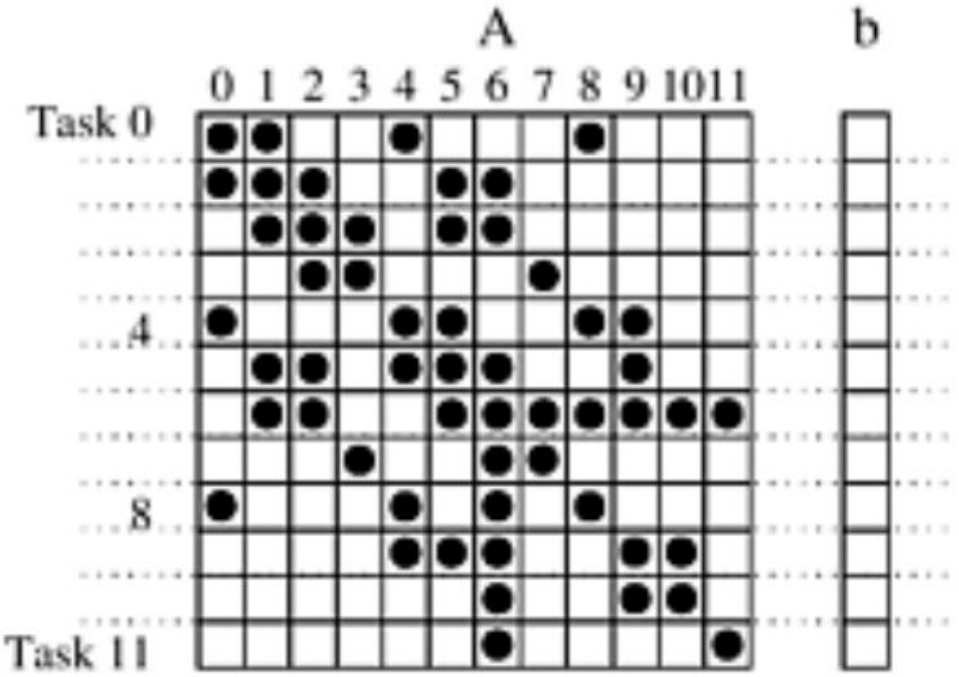
$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Uniform vs. Non-Uniform Task Sizes

- Uniform: All tasks require the same amount of time/work to complete
- Non-Uniform: Tasks require significantly different amounts of time to complete

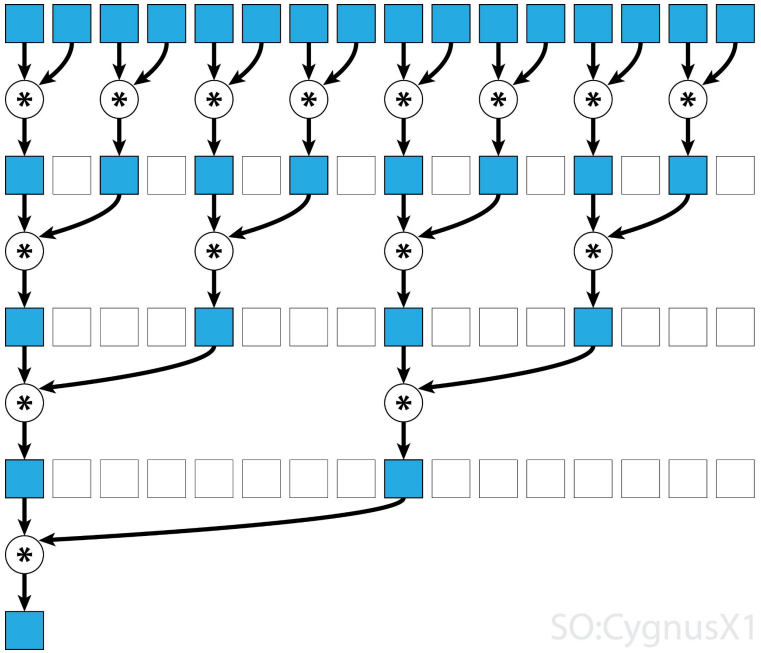


Static vs. Dynamic Task Interactions

- Static Interactions: Task *edges* & process communications take place at predetermined times (requires static task generation & the TDG to be known a priori)
- Dynamic Interactions: Task *edges* & process communications is not known beforehand

Static vs. Dynamic Task Interactions

- Static Interactions: Task *edges* & process communications take place at predetermined times (requires static task generation & the TDG to be known a priori)
- Dynamic Interactions: Task *edges* & process communications is not known beforehand



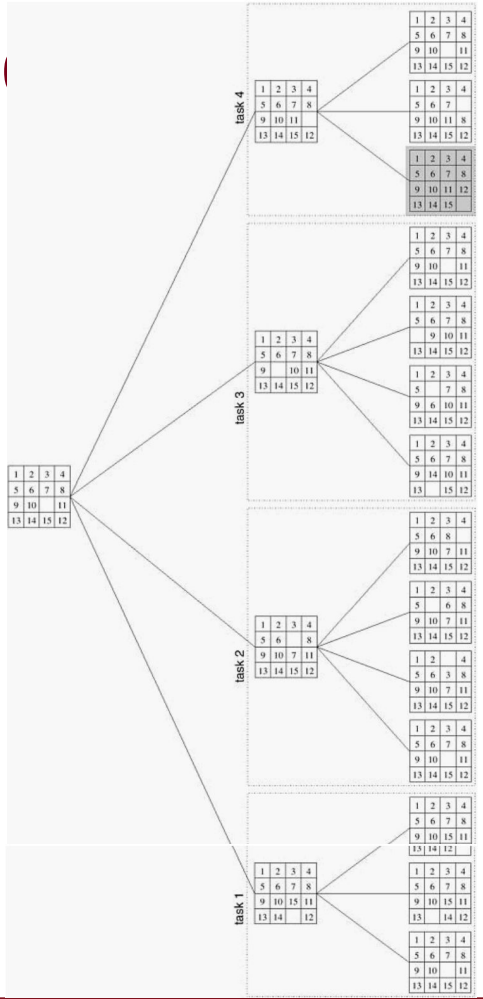
SO:CygnusX1

Parallel Array Sum
Interactions are fixed

Static vs. Dynamic Task Interacti

- Static Interactions: Task *edges* & process communications take place at predetermined times (requires static task generation & the TDG to be known a priori)
- Dynamic Interactions: Task *edges* & process communications is not known beforehand

Search Problems require communication to prevent redundant rollouts



Regular vs. Irregular Task Interactions

- Regular Interactions: Interactions have some general structure which can be exploited for better efficiency. In other words, tasks interactions *look* similar to one another.
- Irregular Interactions: No such pattern of interactions exists. Each task interacts with others in largely different ways

Regular vs. Irregular Task Interactions

- Regular Interactions: Interactions have some general structure which can be exploited for better efficiency. In other words, tasks interactions *look* similar to one another.
- Irregular Interactions: No such pattern of interactions exists. Each task interacts with others in largely different ways

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\begin{aligned} &\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ &\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ &\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ &\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Regular vs. Irregular Task Interactions

- Regular Interactions: Interactions have some general structure which can be exploited for better efficiency. In other words, tasks interactions *look* similar to one another.
- Irregular Interactions: No such pattern of interactions exists. Each task interacts with others in largely different ways