# CSC367 Parallel computing

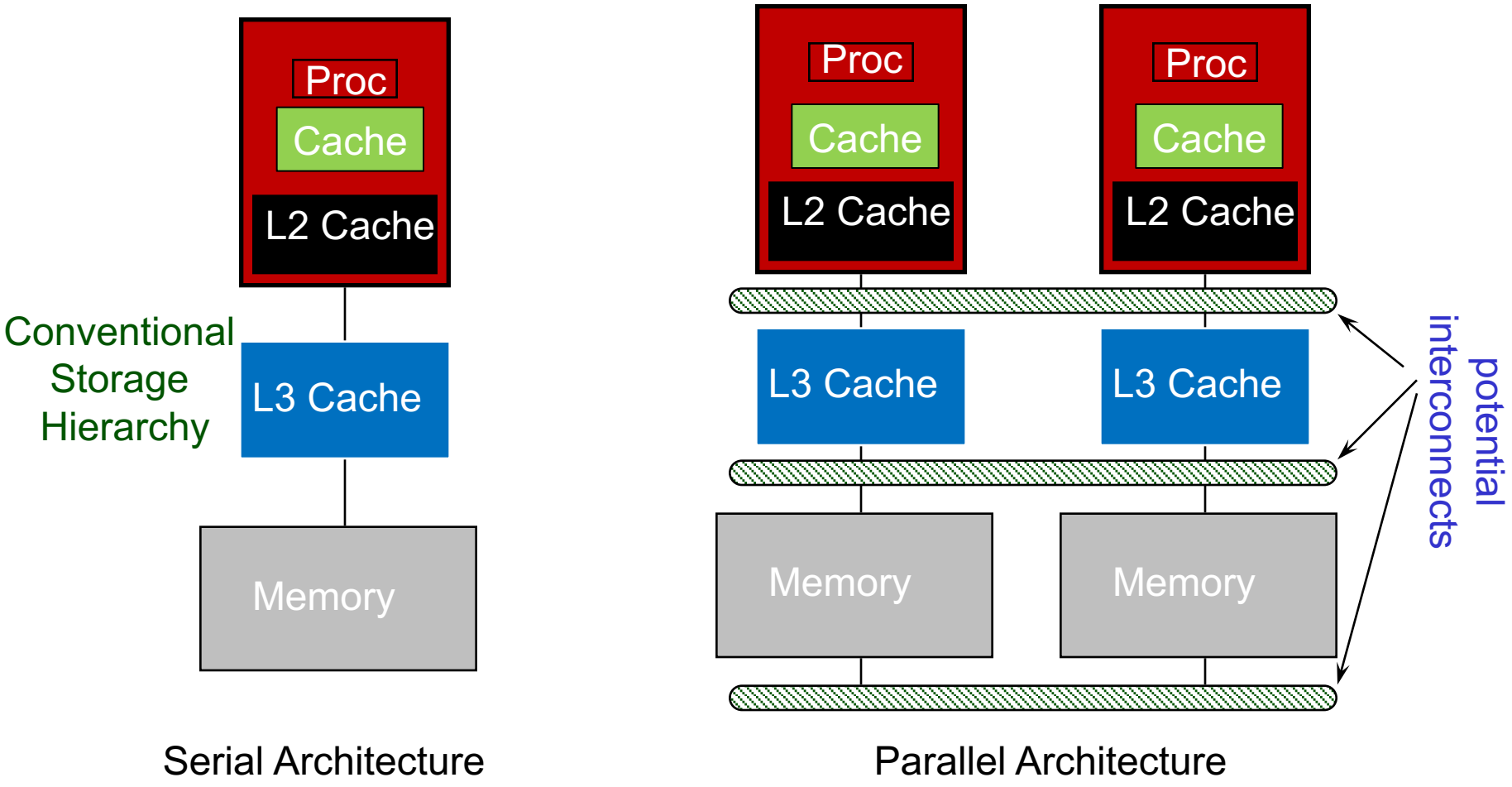
# Lecture 6: Parallel Architectures and Parallel Algorithm Design



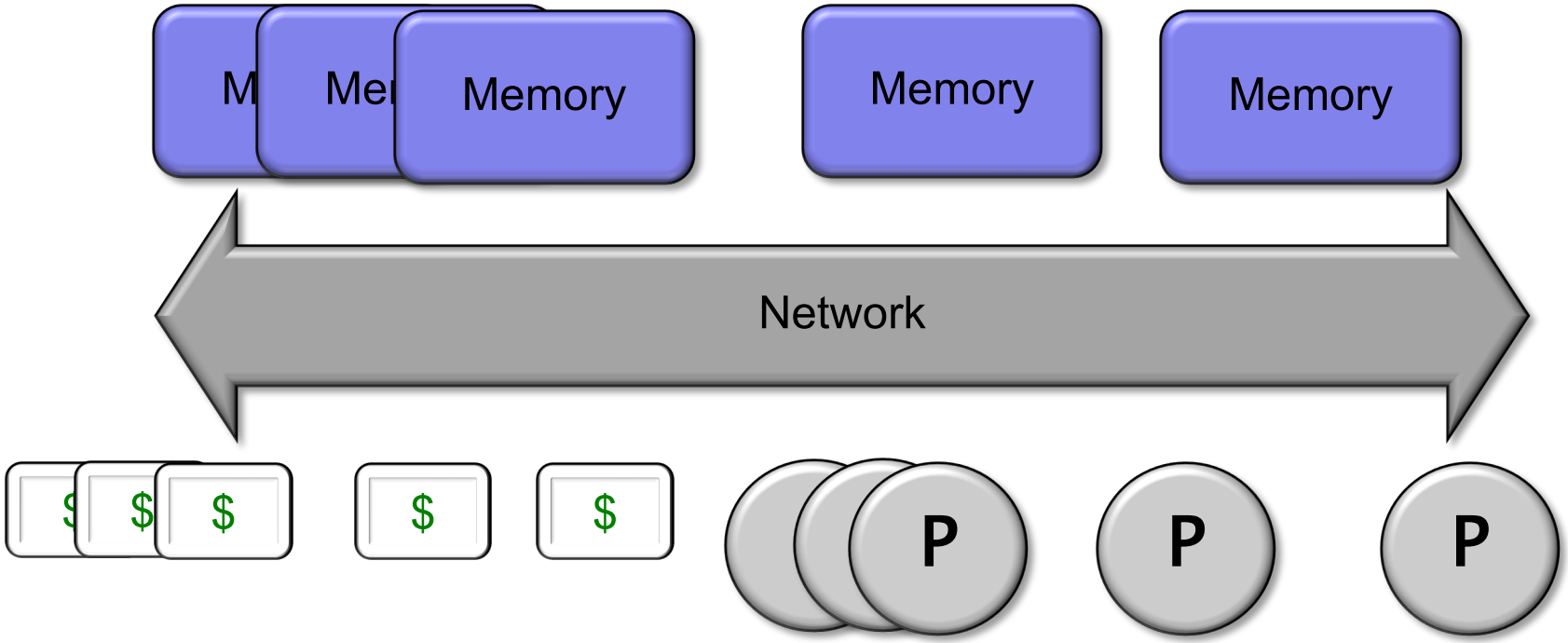


Thanks to Scinet, NSF XSEDE, James Demmel

# Serial and Parallel Architectures



# Essential Components of Parallel Architectures




Where is the memory physically located?

Is it connected directly to processors?

What is the connectivity of the network?

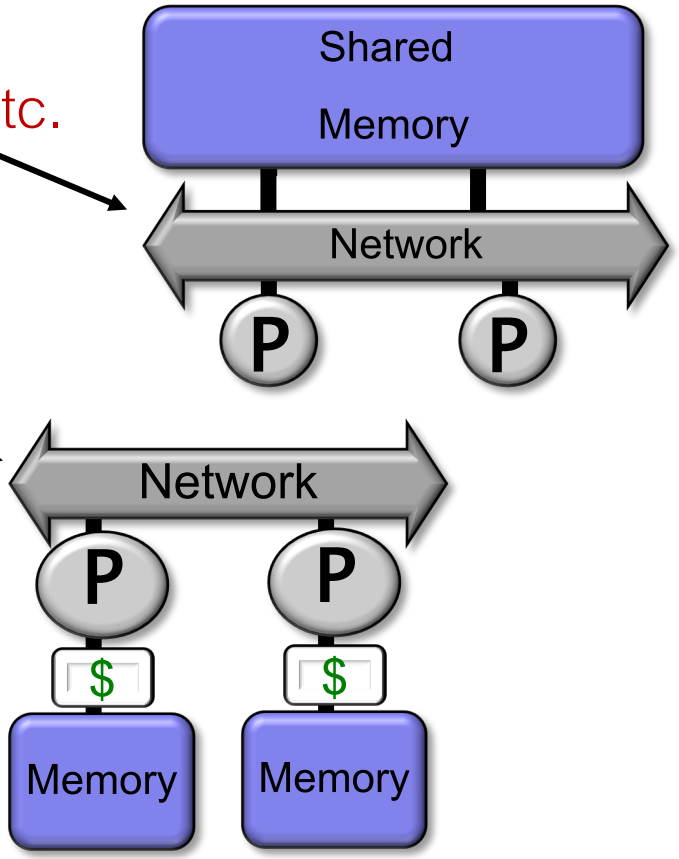
# Parallel Machine Models and Their Programming Models Covered on this class!

Shared Memory: Pthreads, OpenMP, etc.

Distributed Memory: MPI

SIMD and Vector: CUDA

Hybrid: A mix of the above!



# Up Next!

**Parallel Algorithm Design:** Tasks, decomposition, mapping, etc.

Recommended reading for this section (not mandatory but highly recommended, we do cover what is needed in class/slides!): Introduction to Parallel Computing - A. Grama, A. Gupta, G. Karypis, V. Kumar

# Parallel Algorithm Design

General guidelines:

- Identify tasks in your program that can be performed concurrently
- Map concurrent tasks onto multiple threads or processes, to be run in parallel
- Partition the input, output, and/or intermediate data and assign to processes
- Handle concurrent accesses to shared data by multiple processes
- Add synchronization between stages of the parallel execution, where necessary
- Keep in mind the underlying parallel architecture, its advantages and limitations
- Profile performance and determine what the bottlenecks are
- Target optimizations based on profiling information and performance analysis
- Write small benchmarks to test your program in a variety of configurations

# Parallel Algorithm Design: Outline

- Tasks: Decomposition, Task Dependency, Granularity, Interaction, Mapping, Balance
- Decomposition techniques
- Mapping techniques to reduce parallelism overhead
- Parallel algorithm models
- Parallel program performance model

# Decomposition and tasks

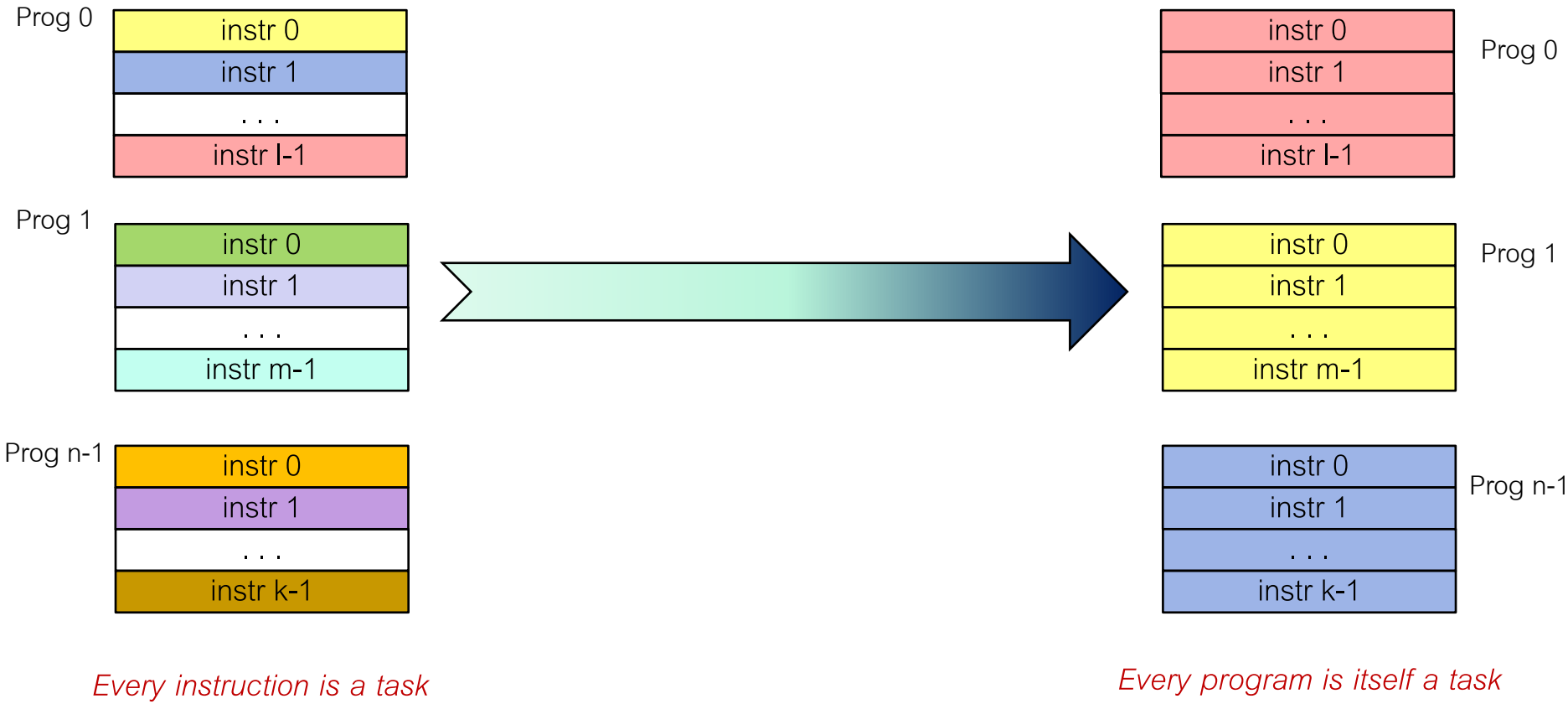
- Decomposition: dividing the computation in a program into **tasks** that could be executed in parallel

- Task: unit of computation that can be extracted from the main program and assigned to a process, and which can be run concurrently with other tasks

- The way to extract tasks and the mapping to processes affects performance!

# Parallel task decomposition

- Tasks can range from individual instructions to entire programs



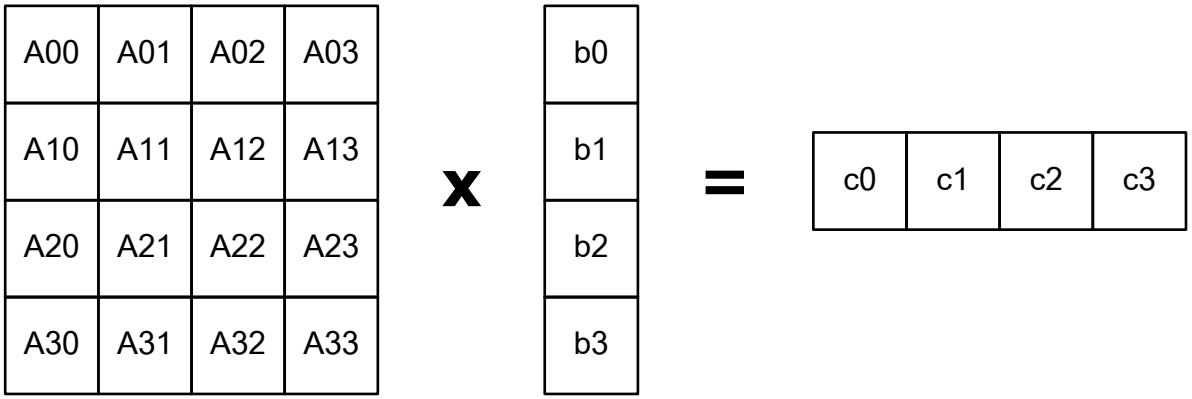

*Every instruction is a task*

*Every program is itself a task*

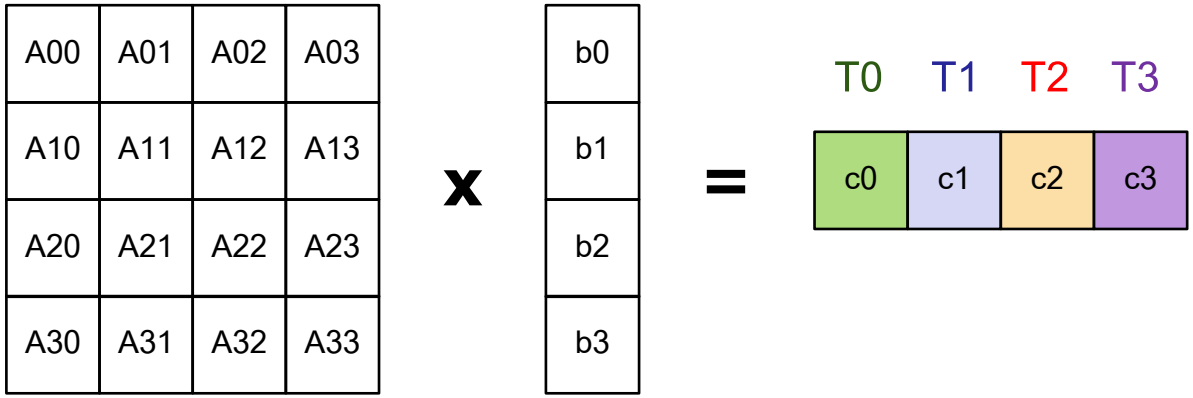
- Which one is best?
  - The answer is always "it depends" .. on the specific application and the parallel platform

# Example: matrix-vector multiplication

- Multiply 4 x 4 dense matrix A with vector b of size 4 => calculate A x b = c




- Say that computing each output item is a task (T0-3)



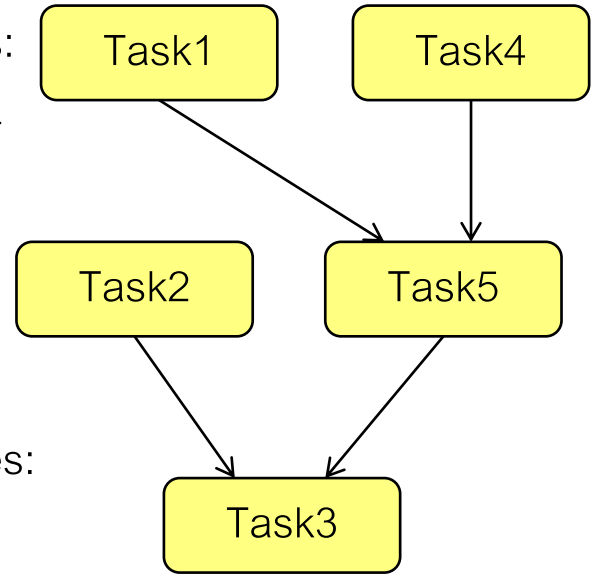

- Consider what each task needs, and if there are data dependencies

# Task dependencies

- Tasks are not independent if they have dependencies on other tasks
- A task might need data produced by other tasks => must wait until input is ready
- Dependencies create an ordering of task execution => task dependency graph
  - Directed acyclic graph (DAG): tasks as nodes, dependencies as edges
  - "Start nodes" = no incoming edges; "Finish nodes" = no outgoing edges
- Examples:

Start nodes:

Tasks 1,2,4

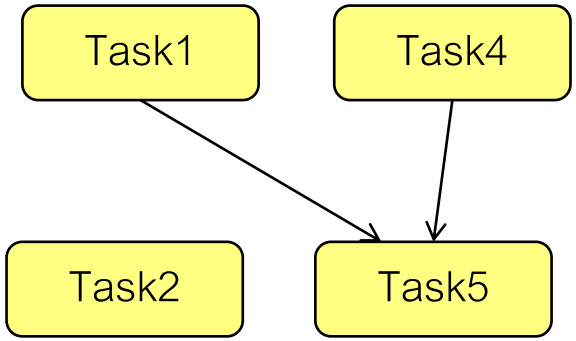



Finish nodes:

Task 3

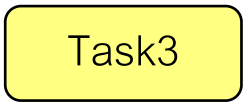
Start nodes:

Tasks 1,2,3,4




Finish nodes:

Tasks 2,3,5

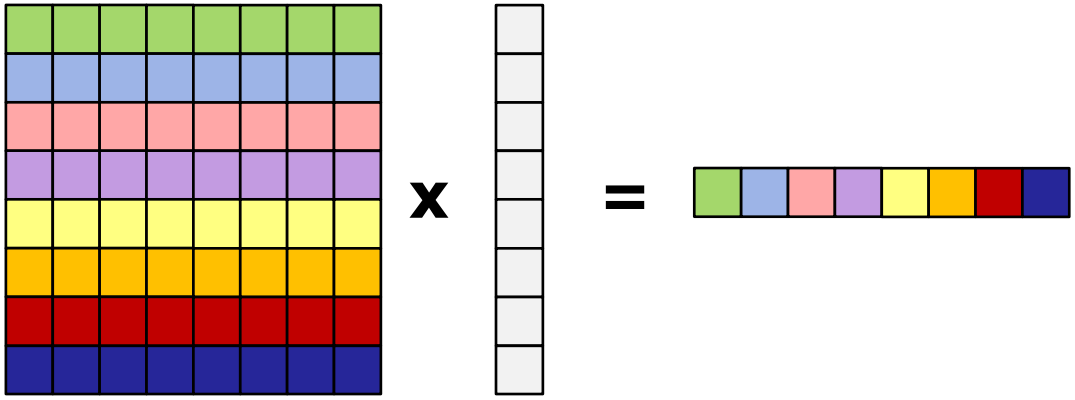


# Granularity

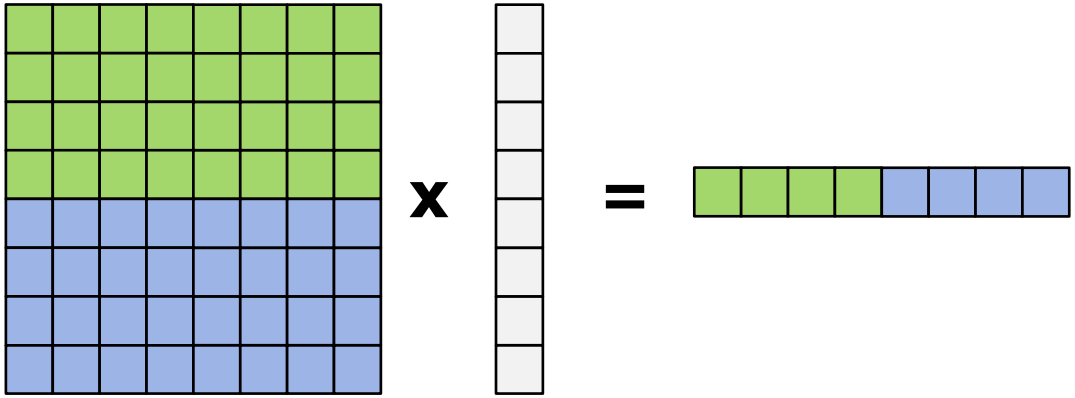
- Granularity: determined by how many tasks and what their sizes are
  - Coarse-grained: a small number of large tasks
  - Fine-grained: a large number of small tasks

# Example: matrix-vector multiplication

- Fine-grained: each task = process a single element of c



- Coarse-grained: each task = process half the elements of c



- Note: we are decomposing into tasks, we will talk later about partitioning data!

# Parallelism and granularity

- Communication between tasks may or may not be necessary
- Ideal parallelism: no communication needed
- Coarse-grained parallelism: Lots of computation performed before communication is necessary for when communication is costly
  - Good match for message-passing environments (MPI, covered later in class)
- Fine-grained parallelism: Frequent communication may be necessary
  - More suitable for shared memory environments (Pthreads, OpenMP)
- Parallelism granularity = how much processing is performed before communication is necessary between processes

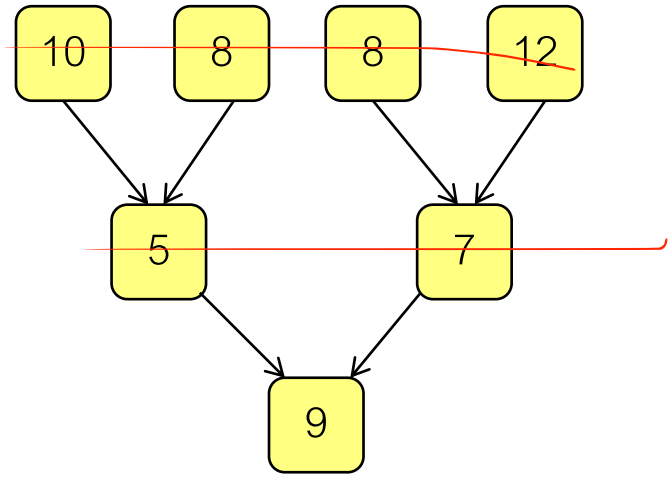
# Degree of concurrency

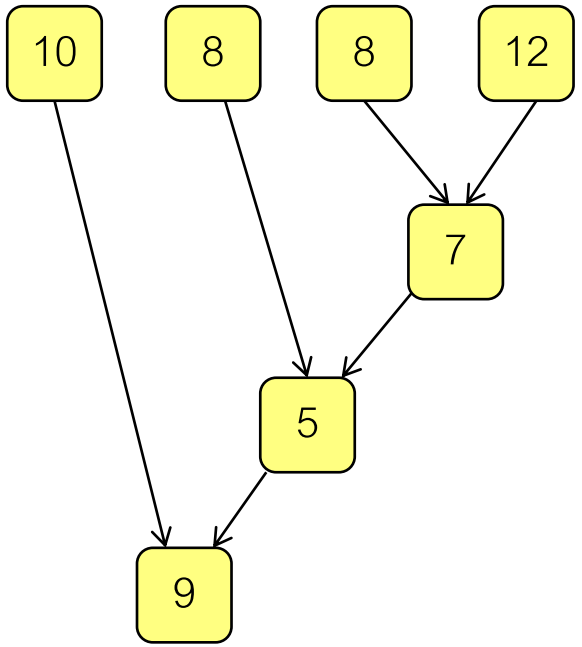
- Maximum degree of concurrency = max number of tasks that can be executed simultaneously at any given time
  - Typically less than total number of tasks, if tasks have dependencies

- Average degree of concurrency = average number of tasks that can be executed concurrently, during the program's execution

# Degree of concurrency

- Nodes can have weights too – tasks may be doing different amounts of work



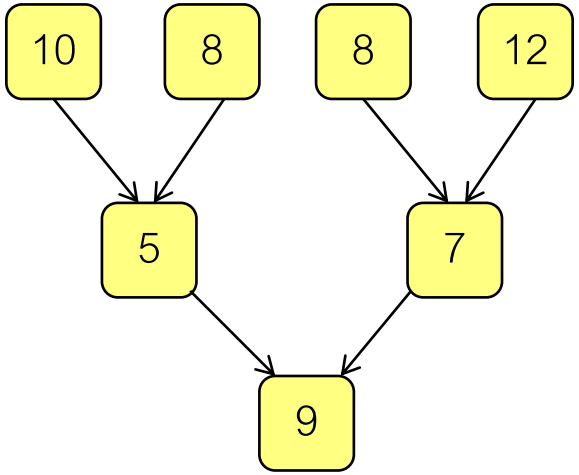

- Max degree of concurrency:
  - a) Max(38, 12, 9) = 38
  - b) Max(38, 7, 5, 9) = 38
- Average degree of concurrency:
  - a) (38+12+9)/(12+7+9) = 59/28 = 2.11
  - b) (38+7+5+9)/(12+7+5+9) = 59/33 = 1.79

critical path (next slide)

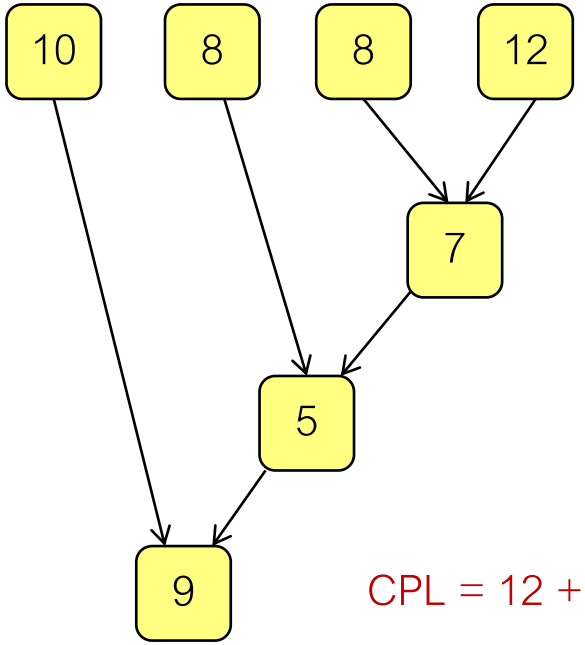
# Critical path

and most weight…

- Critical path = The longest path between any pair of start and finish nodes
- Critical path length = sum of node weights along the critical path




CPL = 12 + 7 + 9

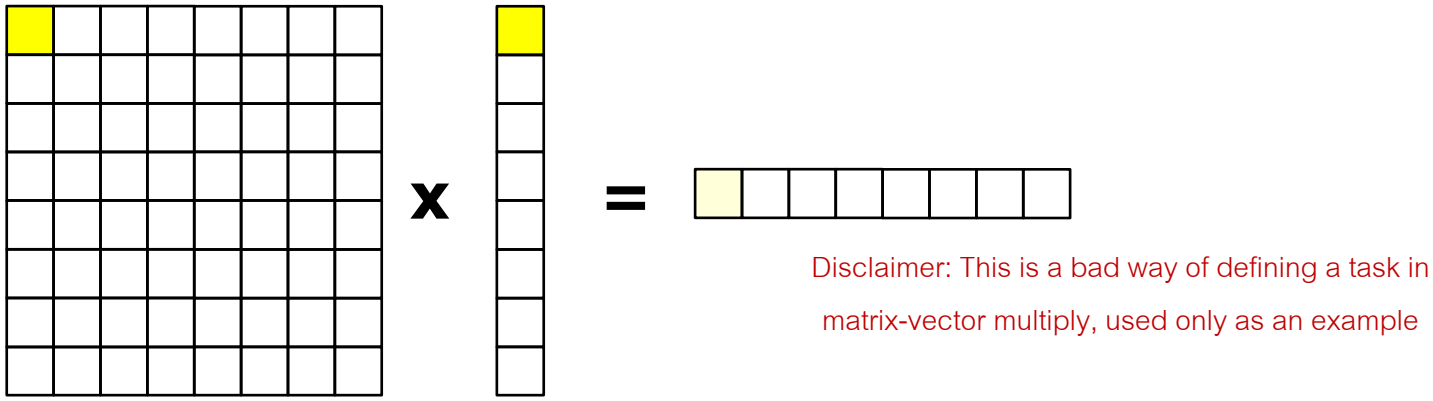



CPL = 12 + 7 + 5 + 9

- Average degree of concurrency = total amount of work / critical path length
- Shorter critical path => higher degree of concurrency

# Granularity and concurrency

- If granularity of decomposition increases (finer-grain), more concurrency available
- More concurrency => more potential tasks to run in parallel
- If so, then reduce program execution time by just increasing granularity of tasks?
- Not quite true!
  - Inherent limits to fine-grained decomposition, e.g., hitting individible tasks, or tasks which cause slowdown if split up
  - For example if a task multiplies one element of A with one element of b to store a partial value of one element of c then all tasks working on the first row of A have to interact!

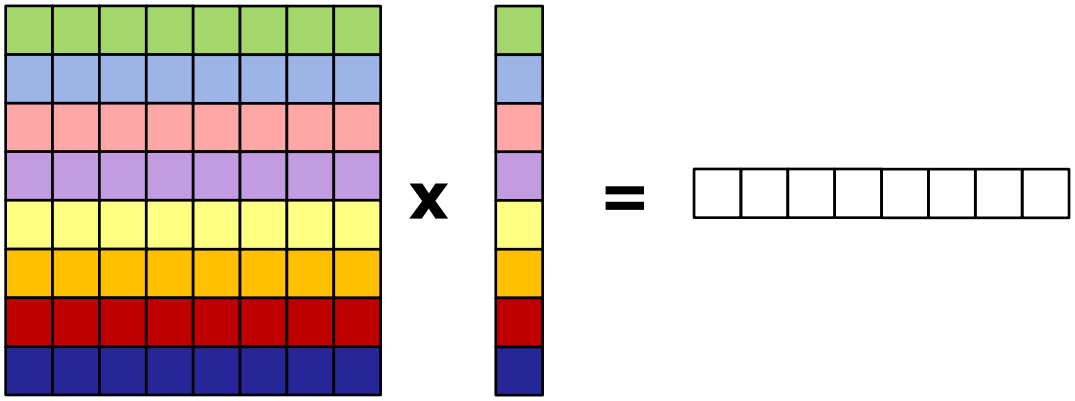



Disclaimer: This is a bad way of defining a task in matrix-vector multiply, used only as an example

  - More tasks => potentially more dependencies => more overhead

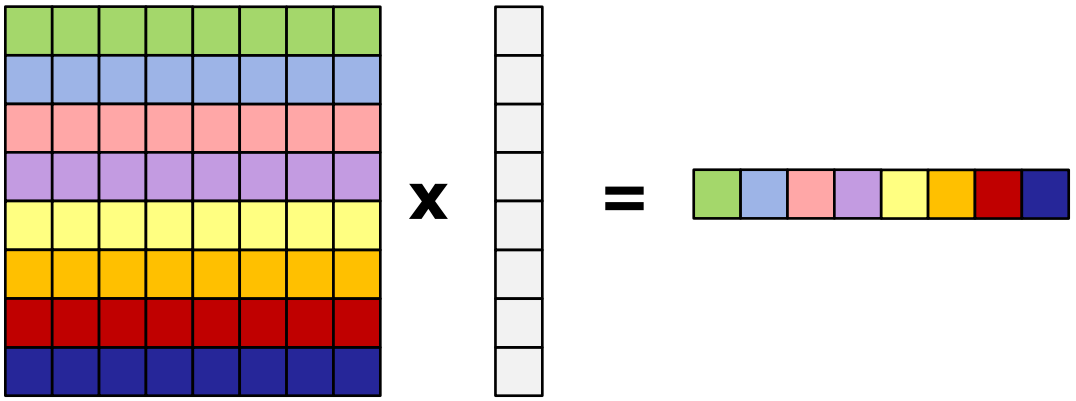
# Task interactions

- A task dependency graph only captures producer-consumer interactions
  - A task's output is used as another task's input
- Interactions might occur among tasks that are independent in the task dependency graph
  - Tasks on different processors might need to exchange data or synchronize
  - e.g., in the below if each task stores one item from b, must exchange their data to get all of b



# Task interactions

- Tasks may share data via task interactions
  - Read-only interactions: tasks only need to read data shared with other tasks

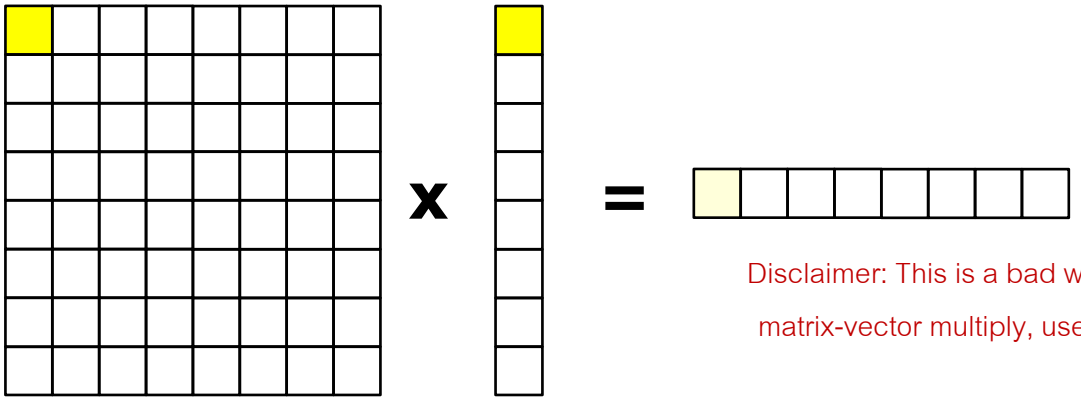



Read-only interactions: all tasks read b

is a type of interaction

# Task interactions

- Tasks may share data via task interactions
  - Read-only interactions: tasks only need to read data shared with other tasks
  - Read-write interactions: tasks can read or write data shared with other tasks




Disclaimer: This is a bad way of defining a task in matrix-vector multiply, used only as an example

Read-write interactions: task write partial sums to b

second row need to wait for all previous row computation to finish

# Task interactions

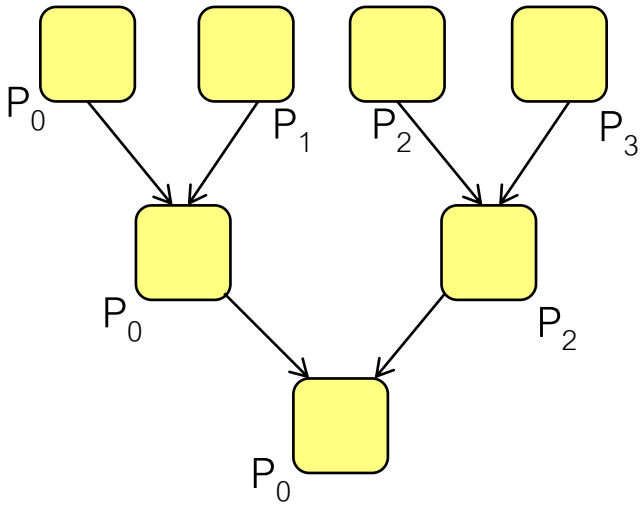
- Tasks may share data via task interactions
  - Read-only interactions: tasks only need to read data shared with other tasks
  - Read-write interactions: tasks can read or write data shared with other tasks
  - Think of the kind of interactions found in the following problem, when solved in parallel:
    - matrix-vector multiplication
- Type of sharing can affect which tasks should get mapped to which processes
  - Read-write interactions should be kept on the same process as much as possible

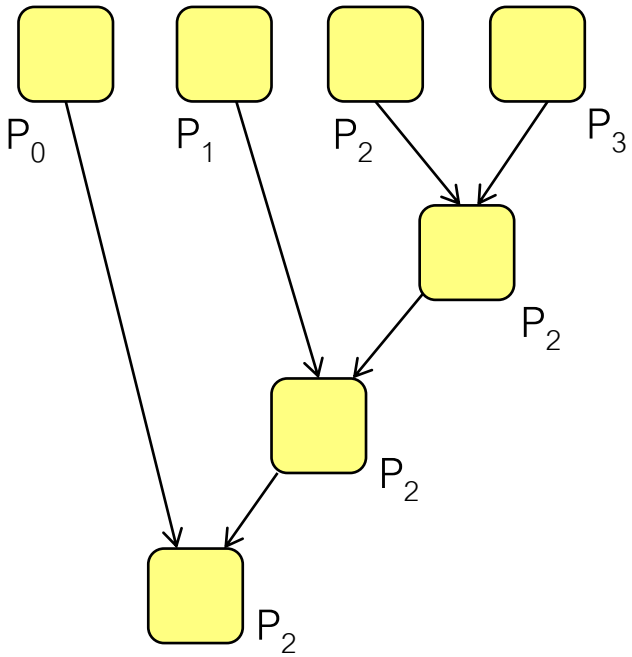
# Mapping tasks to processes

- Mapping = Assigning tasks to processes (more on this later!)
- The choice of decomposition affects the ability to select a good mapping
- Goals of a good mapping:
  - Maximize the use of concurrency
  - Minimize the total completion time
  - Minimize interaction among processes
- Often, the task decomposition and mapping can result in conflicting goals
  - Must find a good balance to optimize for all goals
- Degree of concurrency is affected by decomposition choice, but the mapping affects how much of the concurrency can be efficiently utilized

# Example: mapping tasks to processes

- Map the tasks to processes, in each of the two situations
- Key questions: How many processes can be used? How effectively are you using them and why?




- Max degree of concurrency is 4 => max 4 useful processes
- Map first 4 tasks, each on a separate process, then consider the other 3

# Task Size and Balance

- Task size = proportional to time needed to complete the task
  - Uniform tasks: require roughly the same amount of time
  - Non-uniform tasks: execution times vary widely

- Size of data associated with tasks = how much data does each task process
  - Impacts whether the tasks are well-balanced
  - Affects performance if a task's data must be moved from a remote processor
  - Input data might be small, but output data is large, or vice-versa, etc.

# Parallel Algorithm Design: Outline

- Tasks: Decomposition, Task Dependency, Granularity, Interaction, Mapping, Balance
- Decomposition techniques
- Mapping techniques to reduce parallelism overhead
- Parallel algorithm models
- Parallel program performance model

# Two Commonly Used Decomposition techniques

- Recursive decomposition: Primarily decomposes tasks
- Data decomposition: Partitions the data to induce task decomposition

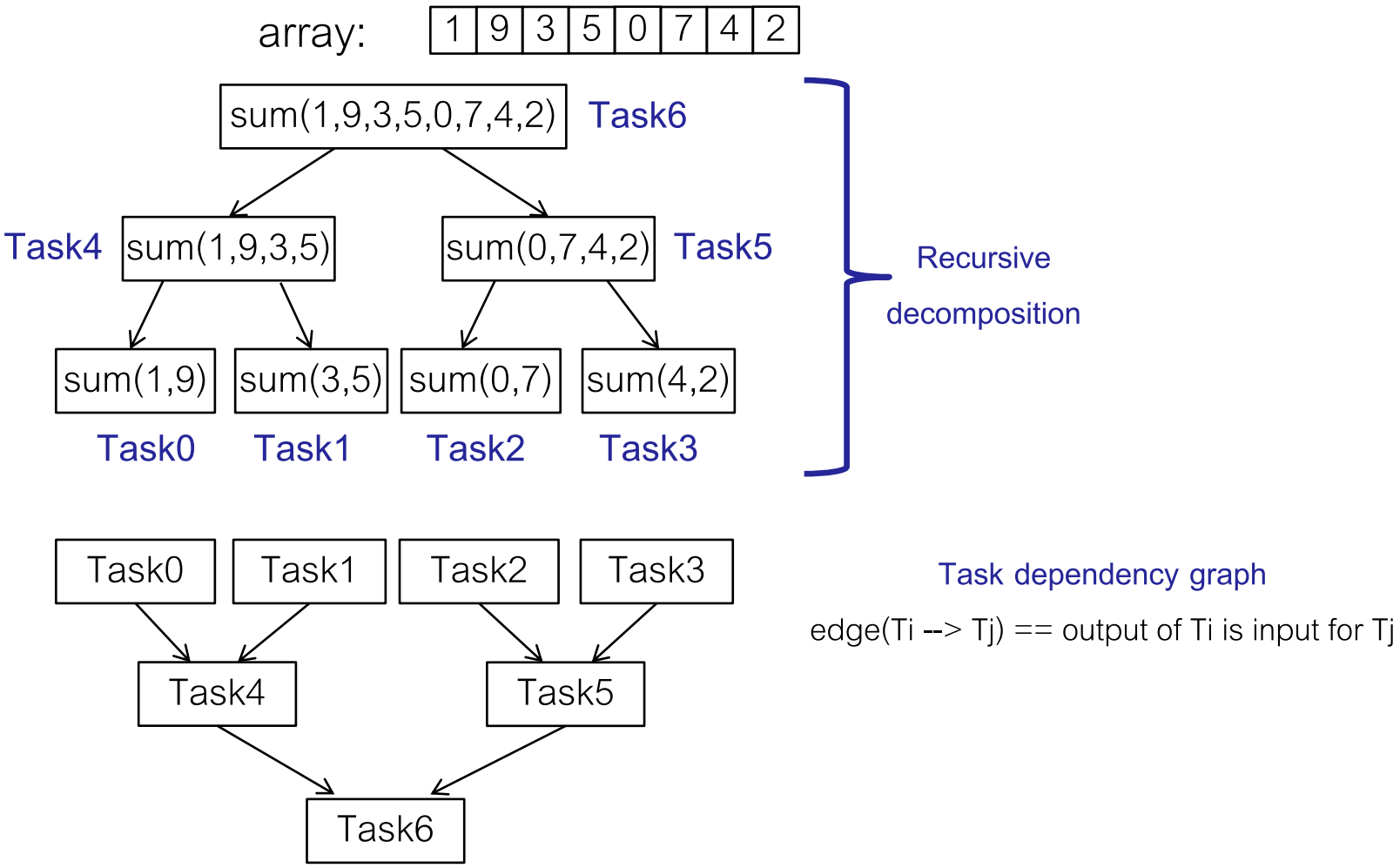
# Recursive decomposition

- Recursive decomposition is primarily based on task decomposition
- Useful for problems that can be approached using a divide-and-conquer strategy
- Divide problem into subproblems, solve subproblems by subdividing recursively the same way and combining results
- Subproblems can be solved concurrently
- Example: Mergesort

```
mergesort(A, lo, hi)
    if lo+1 < hi then // At least 2 elements
        mid = ⌊(lo + hi) / 2⌋
        mergesort(A, lo, mid)
        mergesort(A, mid, hi)
        merge(A, lo, mid, hi) // merge the 2 halves
```

# Recursive decomposition

- Not just for naturally recursive problems like mergesort, quicksort, etc.
- Consider the problem of calculating the sum of an array – decompose it into tasks.

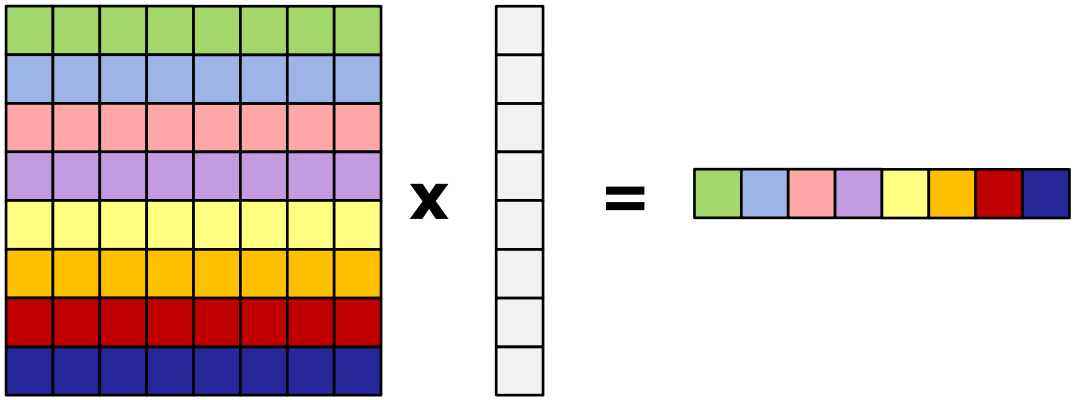


# Data decomposition

- Partition the data on which computations are performed
- Use the data partitioning to perform the decomposition of computation into tasks
- Used to exploit concurrency on problems that operation on large data
- Data decomposition is typically performed in two stages:
  - Step 1: Partition the data
  - Step 2: Induce task decomposition from the partitioned data (might have to re-iterate between steps 1 and 2)
- Data partitioning comes in different flavors:
  - Partition output data
  - Partition input data
  - Partition both input and output data

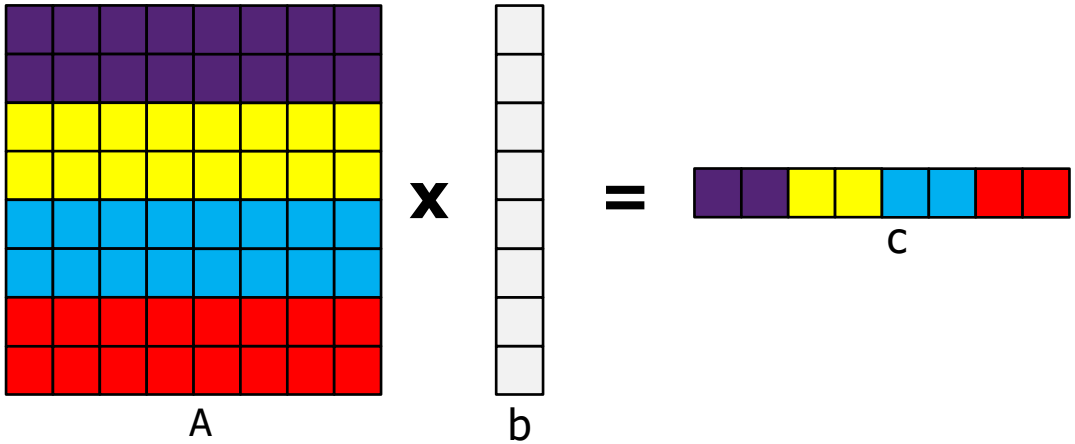
# Partition output data

- Matrix-vector example: (1) each element of the output can be computed independently: In this case, this induces a partitioning of the input matrix as well




(2) Decompose the computation into tasks

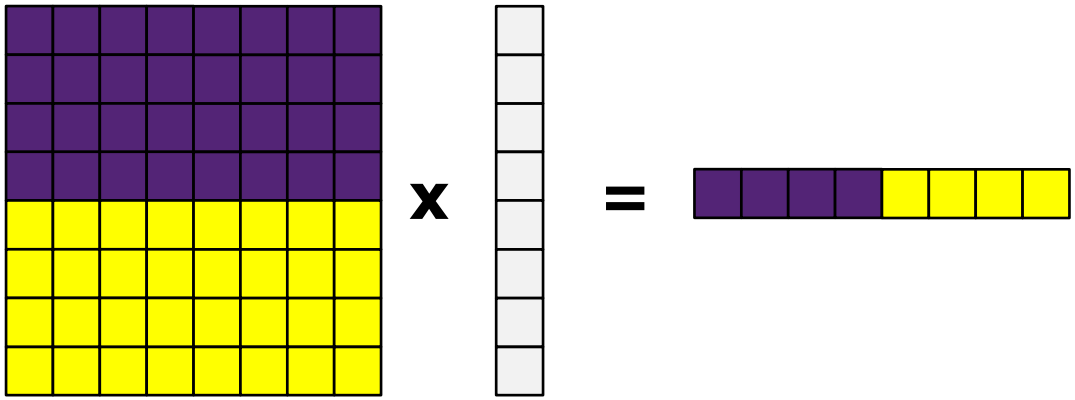
- Option 1: 4 tasks, each computes 2 consecutive elements of the result




Partitioning data != Decomposing computation into tasks

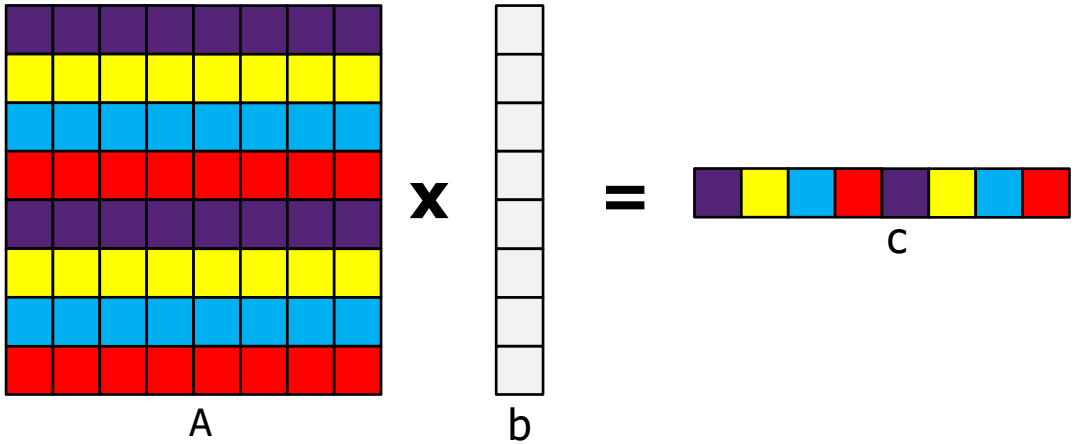
# Other task decompositions

- Option 2: 2 tasks, each computes 4 consecutive elements of c=> coarser-grained!
  - Is this better than the previous decomposition?



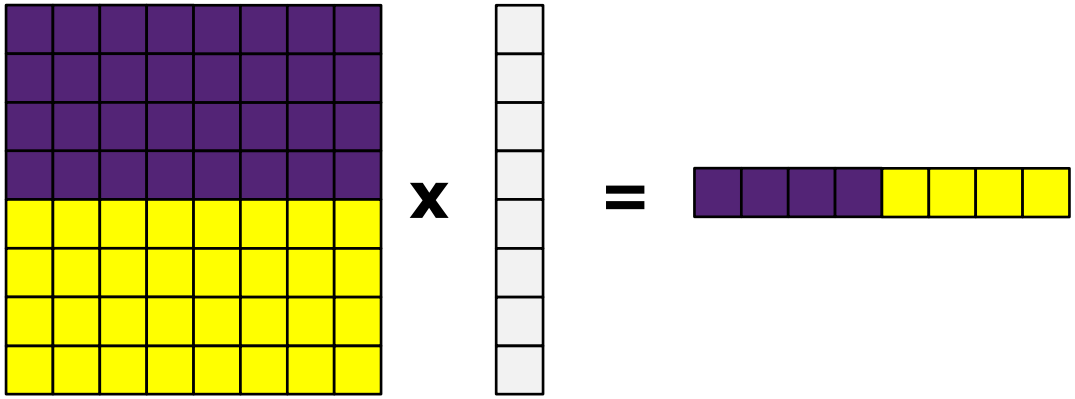
Cant say which one is better without knowing the mapping strategy and the parallel architecture/programming model!

- Option 3: 4 tasks, each computes 2 non-consecutive (strided) elements c
  - How does this compare to previous decompositions?

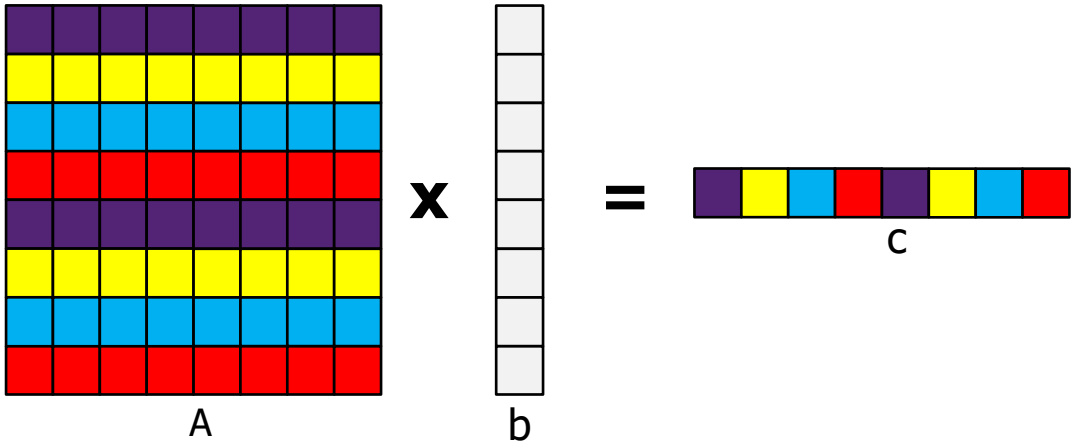


# Other task decompositions

- Option 2: 2 tasks, each computes 4 consecutive elements of c=> coarser-grained!
  - Is this better than the previous decomposition?




- Option 3: 4 tasks, each computes 2 non-consecutive (strided) elements c
  - How does this compare to previous decompositions?




Output data partitioning:

typically good if parts of the output can be naturally computed as a function of the input data!

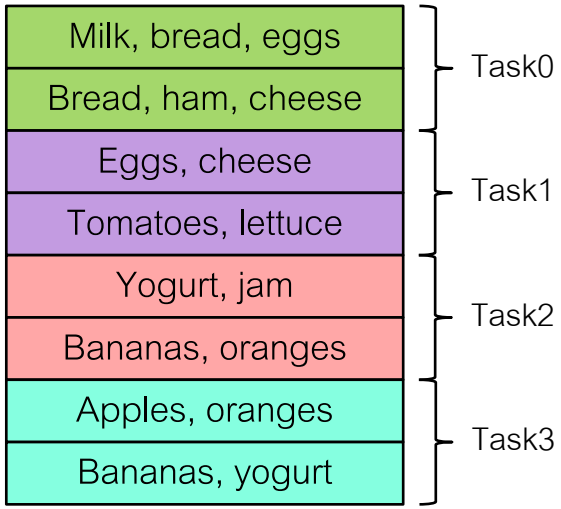
# Another example – partition output data

- Analysis of items bought together frequently – how frequently is each item set found in the store's recent transactions record:

Transactions

| |
|---|
| Milk, eggs, bread, cheese, tomatoes |
| Bread, ham, cheese, lettuce, oranges |
| Milk, lettuce, bananas, apples, yogurt |
| Eggs, bread, cheese, apples, jam |
| Ham, cheese, bananas, oranges, yogurt |
| Milk, cheese, tomatoes, lettuce, jam |
| Eggs, lettuce, oranges, yogurt, jam |
| Bread, bananas, yogurt, chocolate |
| Eggs, ham, apples, bananas, chocolate |
| Milk, cheese, tomatoes, yogurt |

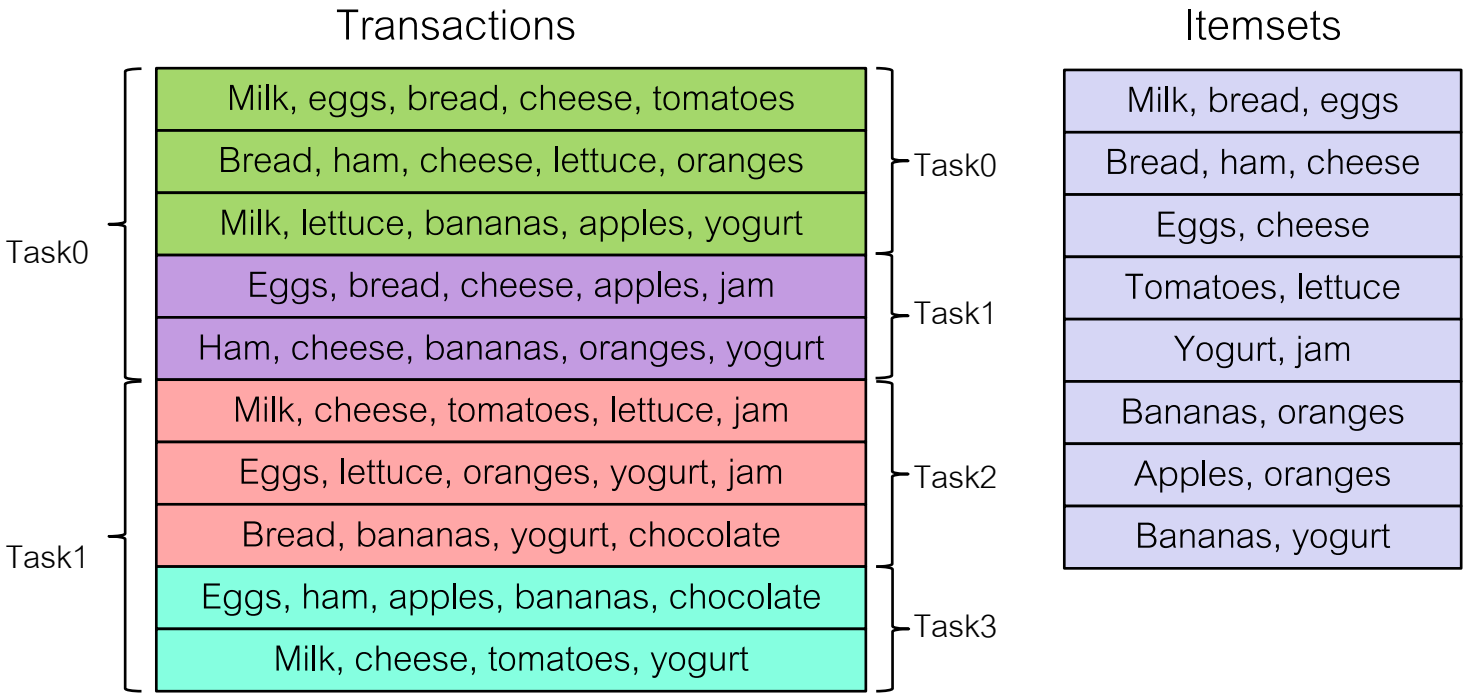
Itemsets




- Partition based on the output data and decompose into tasks - one example:
  - Partition output data into 4 chunks, decompose into 1 task per chunk
  - Each task computes frequencies of its itemsets against all the store transactions

# Partition input data

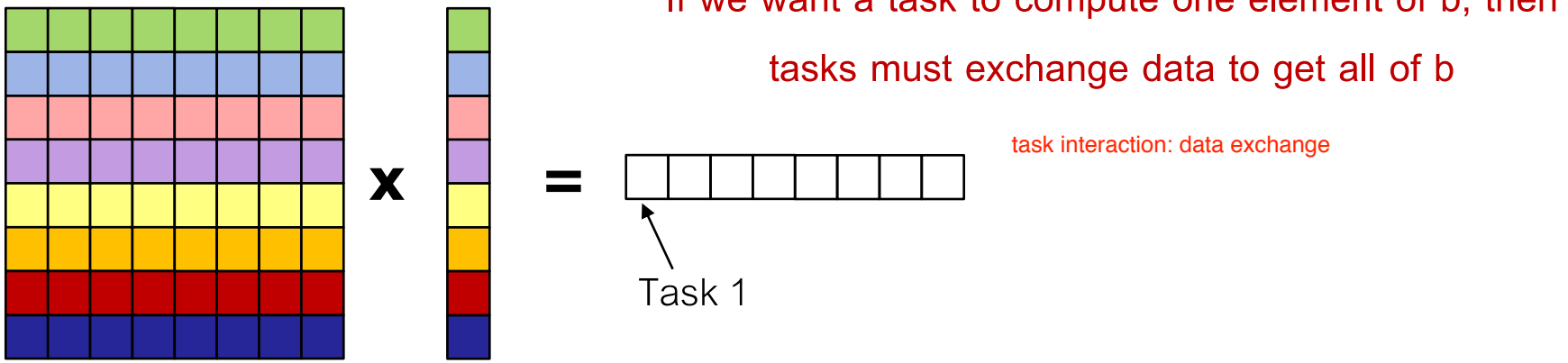
- Analysis of items bought together frequently – how frequently is each item set found in the store's recent transactions record:




- Partition based on the input data and decompose into tasks - one example:
  - Partition input data into 4 roughly-equal chunks, decompose into 2 chunks per task
  - Each task computes frequencies of all itemsets against its chunk of store transactions

# Partition input data – other examples

- Matrix-vector example: row-wise partitioning, partition b similarly
  - If each task takes one row of A and one item of b, any task dependencies?
  - Task interactions?

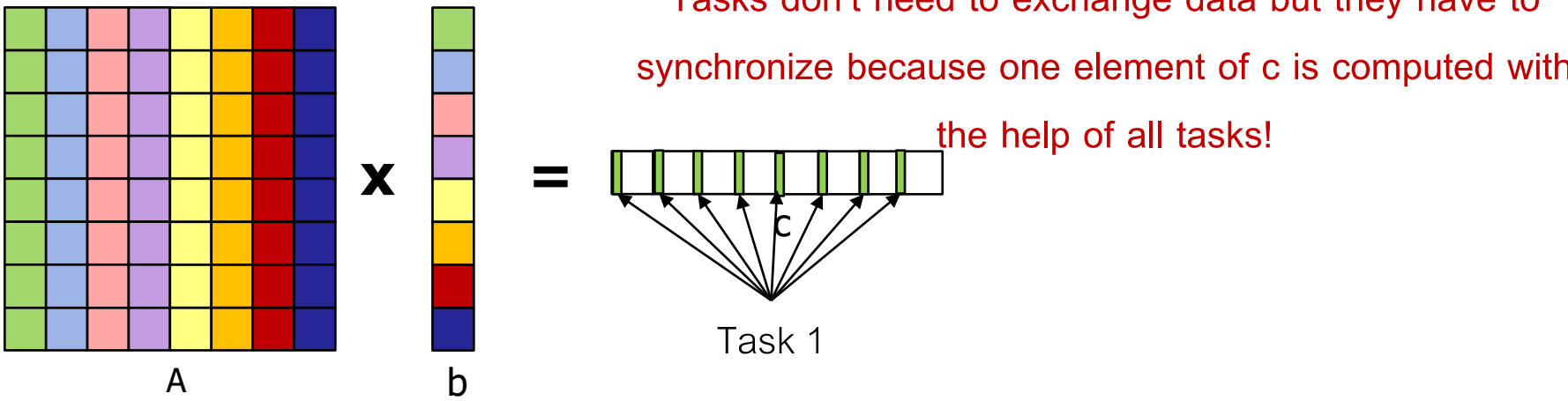



If we want a task to compute one element of b, then tasks must exchange data to get all of b

task interaction: data exchange

- Now let's choose the partitioning below:

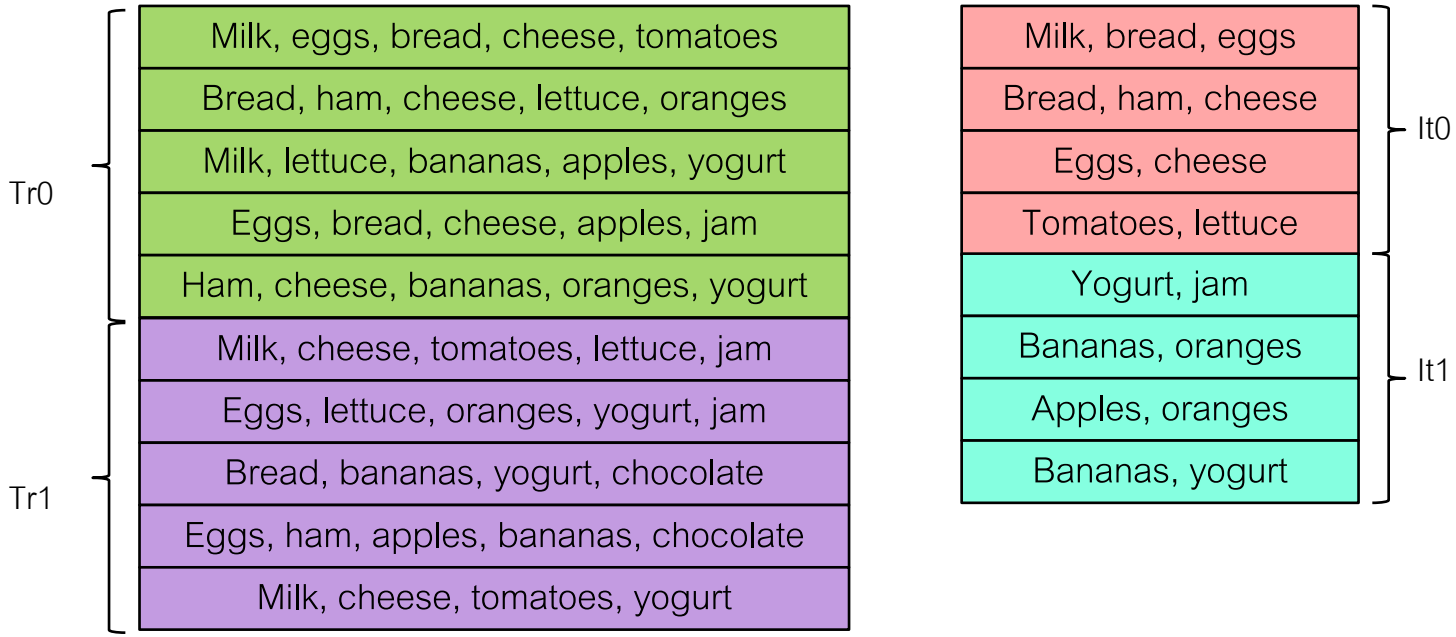
task interaction: synchronization

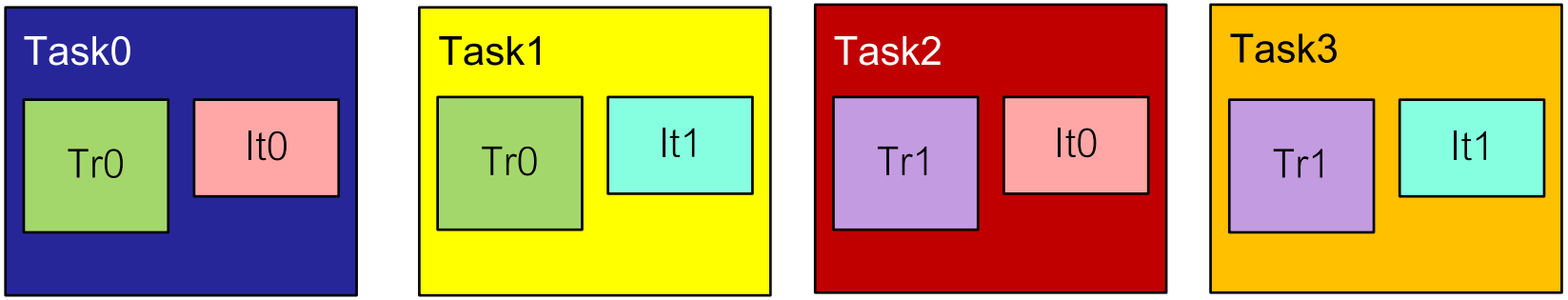



Tasks don't need to exchange data but they have to synchronize because one element of c is computed with the help of all tasks!

# Partition both input and output data

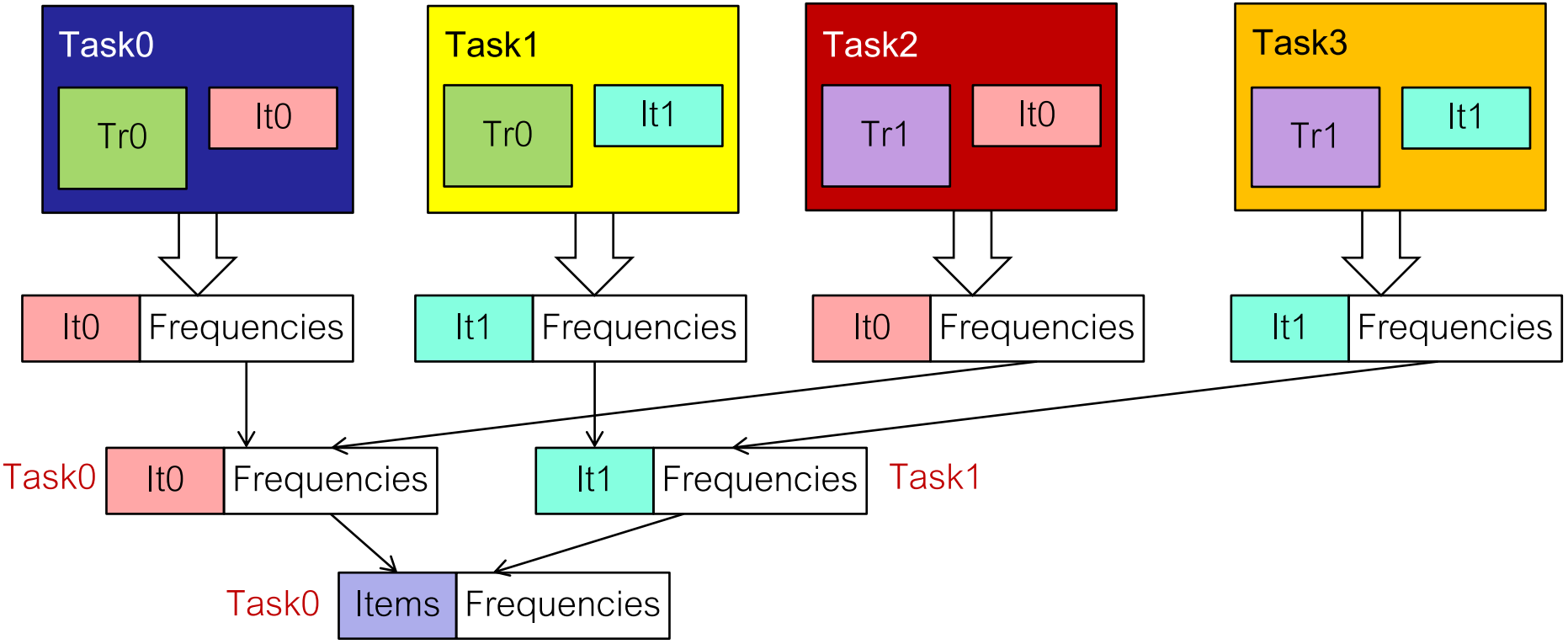
- Partition based on both the input data and output data and create tasks
  - Each task handles the frequency of 1 chunk of itemsets into 1 chunk of transactions



# Partition both input and output data

- Each Task produces a number of matches for each itemset in its chunk of itemsets => must combine the intermediate data




- One possibility: One of the tasks for It0 and It1 will fetch the results to combine them, then one of them combines the final result