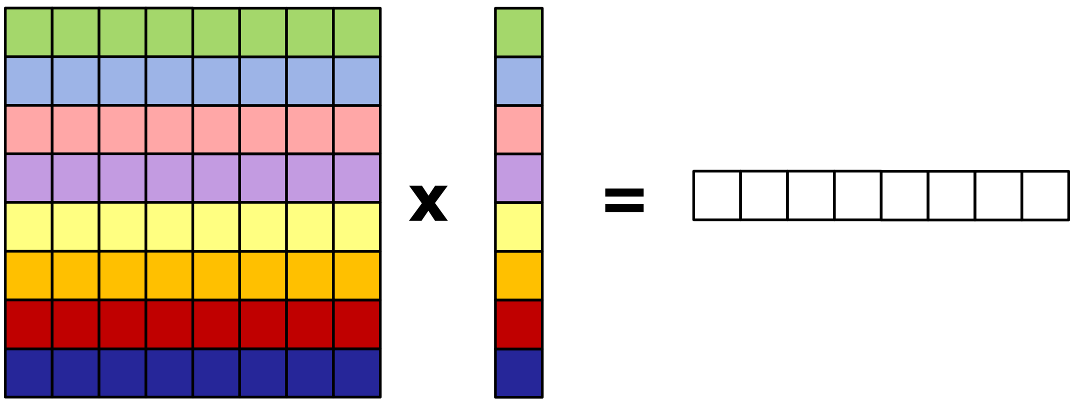
CSC367 Parallel computing

Lecture 7: Parallel Architectures and Parallel Algorithm Design

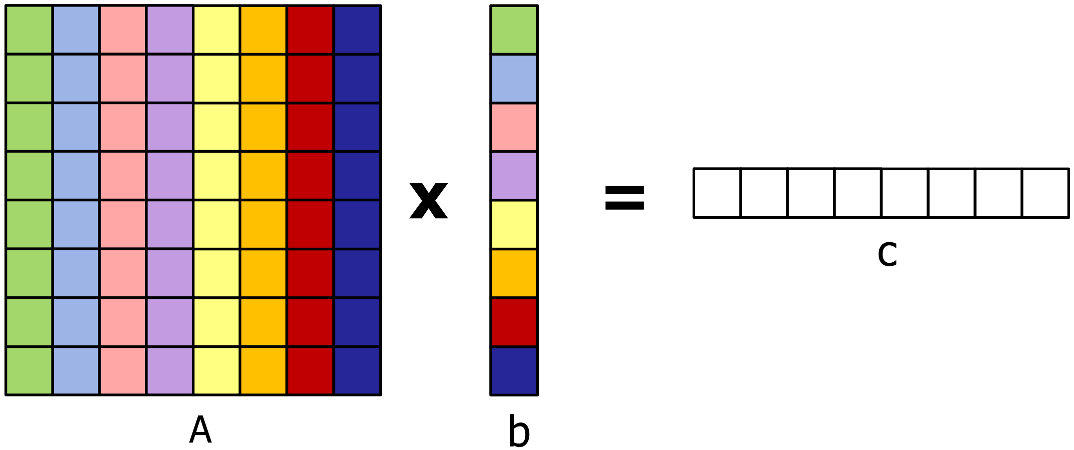
Thanks to Scinet, NSF XSEDE, James Demmel

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?



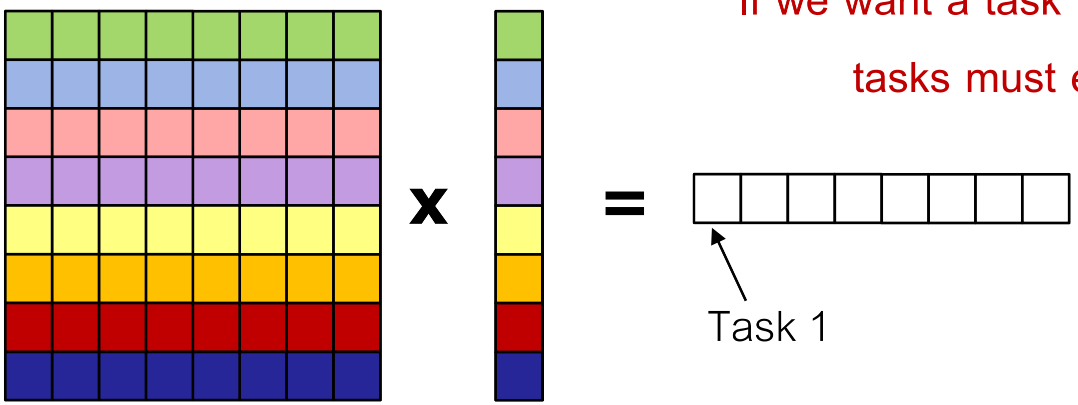
- Now let's choose the partitioning below:



Partition input data – other examples

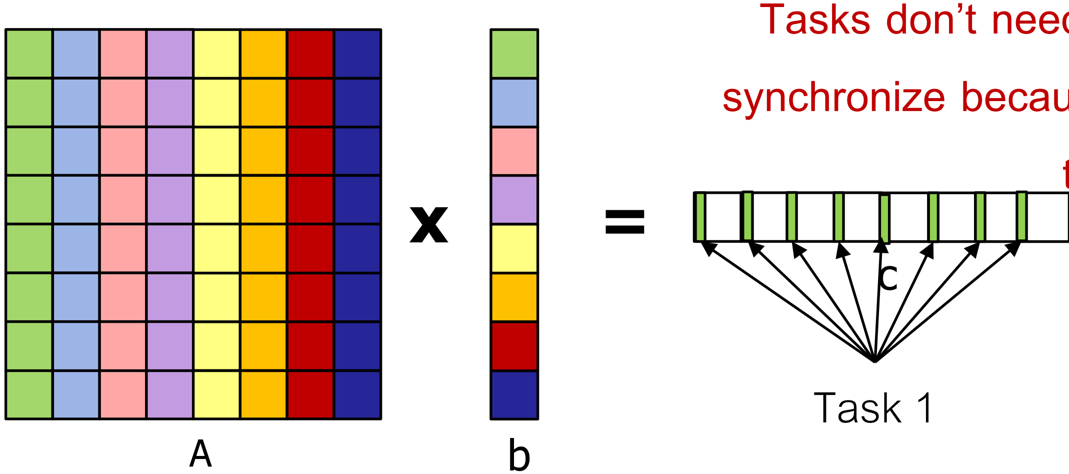
- Matrix-vector example: row-wise partitioning, partition b similarly
  - If each task takes one row of A and one item of b, any task dependencies?
  - Task interactions?

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



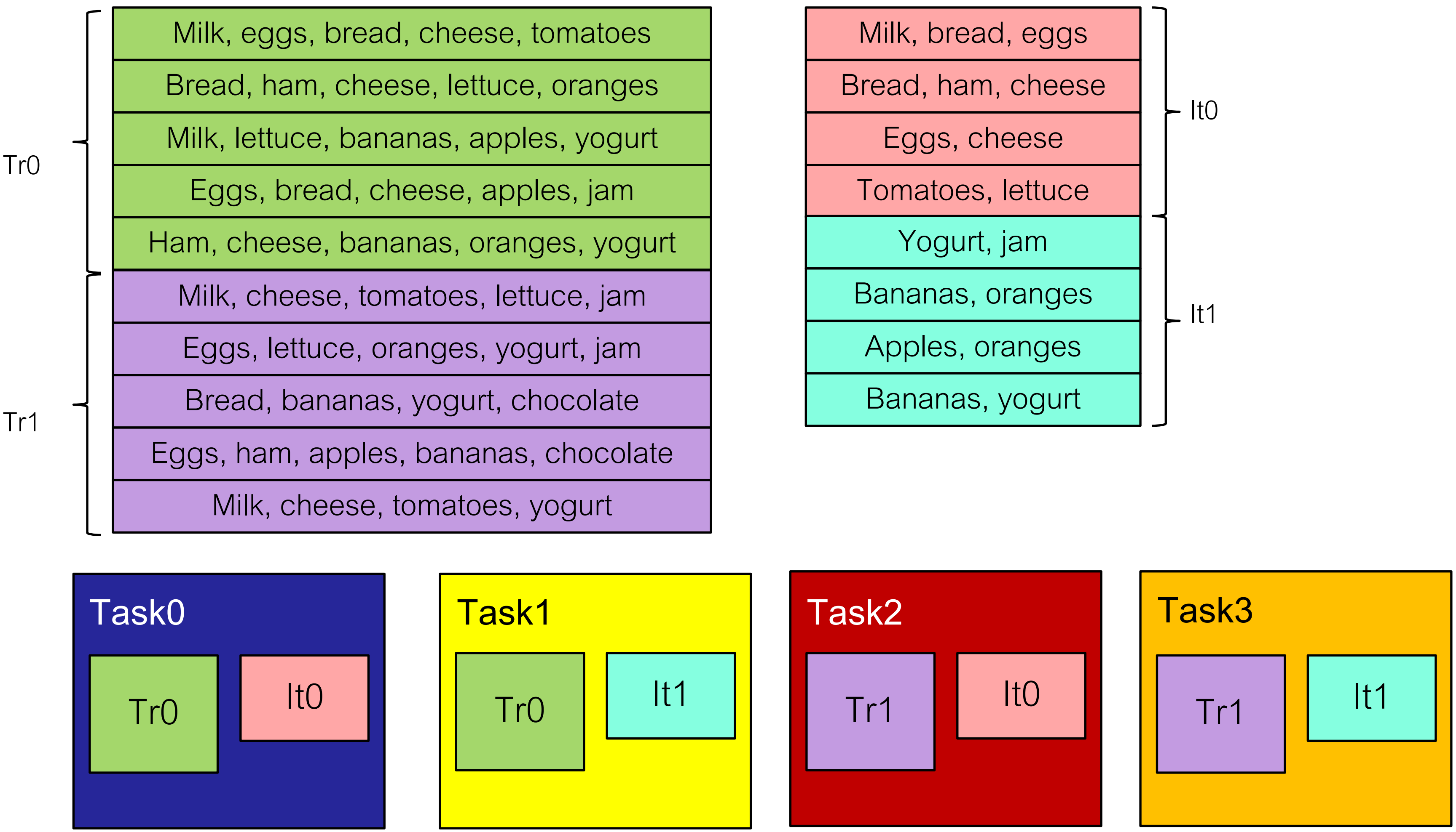
- Now let's choose the partitioning below:

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!



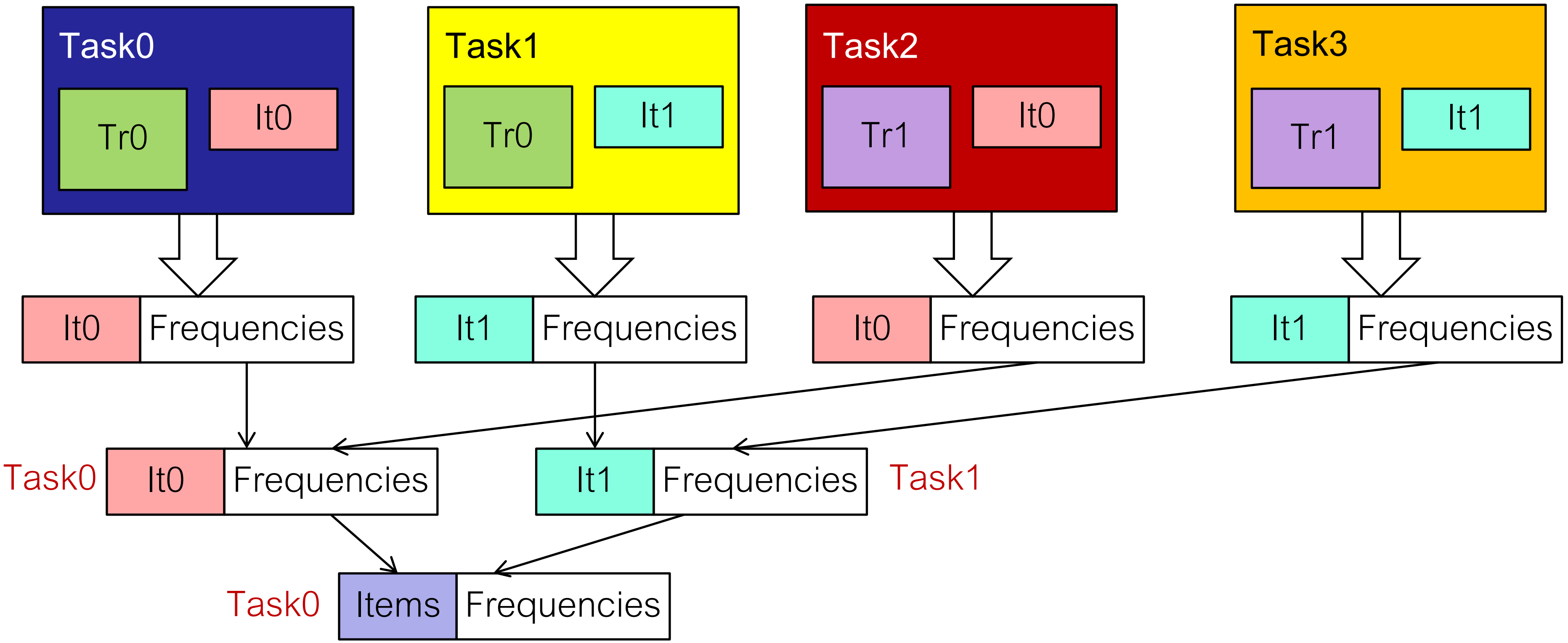
Partition both input and output data

- 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

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

Mapping the Tasks

- Why care about mapping the task, what if we just randomly assign tasks to processors?

Mapping the Tasks

- Why care about mapping the task, what if we just randomly assign tasks to processors?
  - An efficient task mapping is critical to minimize parallel processing overheads: What overheads!

Mapping the Tasks

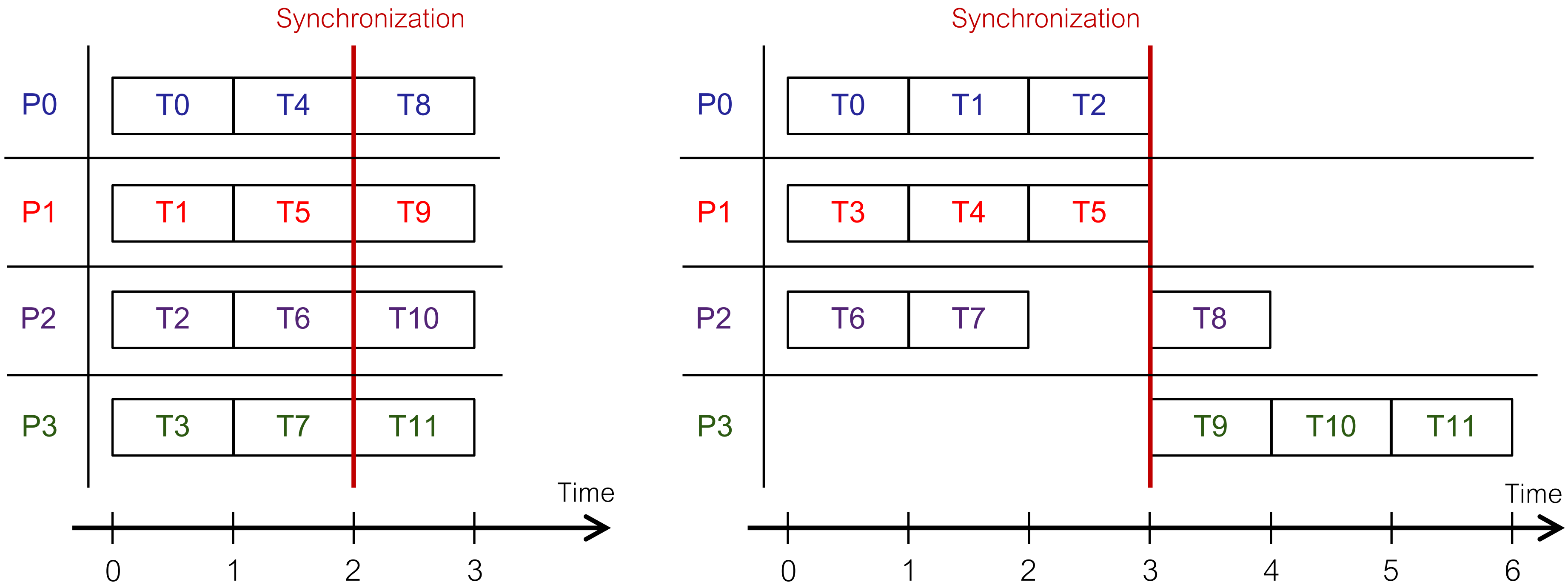
- Why care about mapping the task, what if we just randomly assign tasks to processors?
  - An efficient task mapping is critical to minimize parallel processing overheads: What overheads!
    - Load imbalance
    - Inter-process communication: culprits are synchronization and data sharing

Mapping tasks to processes

- Mapping goal: all tasks must complete in shortest possible time
- To do so, minimize overheads of task execution
  - 1. Load Balancing: Minimize the time spent idle by some processes
  - 2. Minimize the time spent in interactions among processes
- The two goals can be conflicting
  - To optimize 2, put interacting tasks on the same processor => can lead to load imbalance and idling (extreme case: assign all tasks to the same processor)
  - To optimize 1, break down tasks into fine-grained pieces, to ensure good load balance => can lead to a lot more interaction overheads
- Must carefully balance the two goals in the context of the problem!

Mapping tasks to processes to balance load

- Warning: a balanced load may not necessarily mean no idling!
  - If the work is carried out in stages, but assigned workload is not balanced for every stage
- Example: Tasks T0-11, data dependency: T8-11 must all wait for T0-7 to finish
  - Possible decompositions:



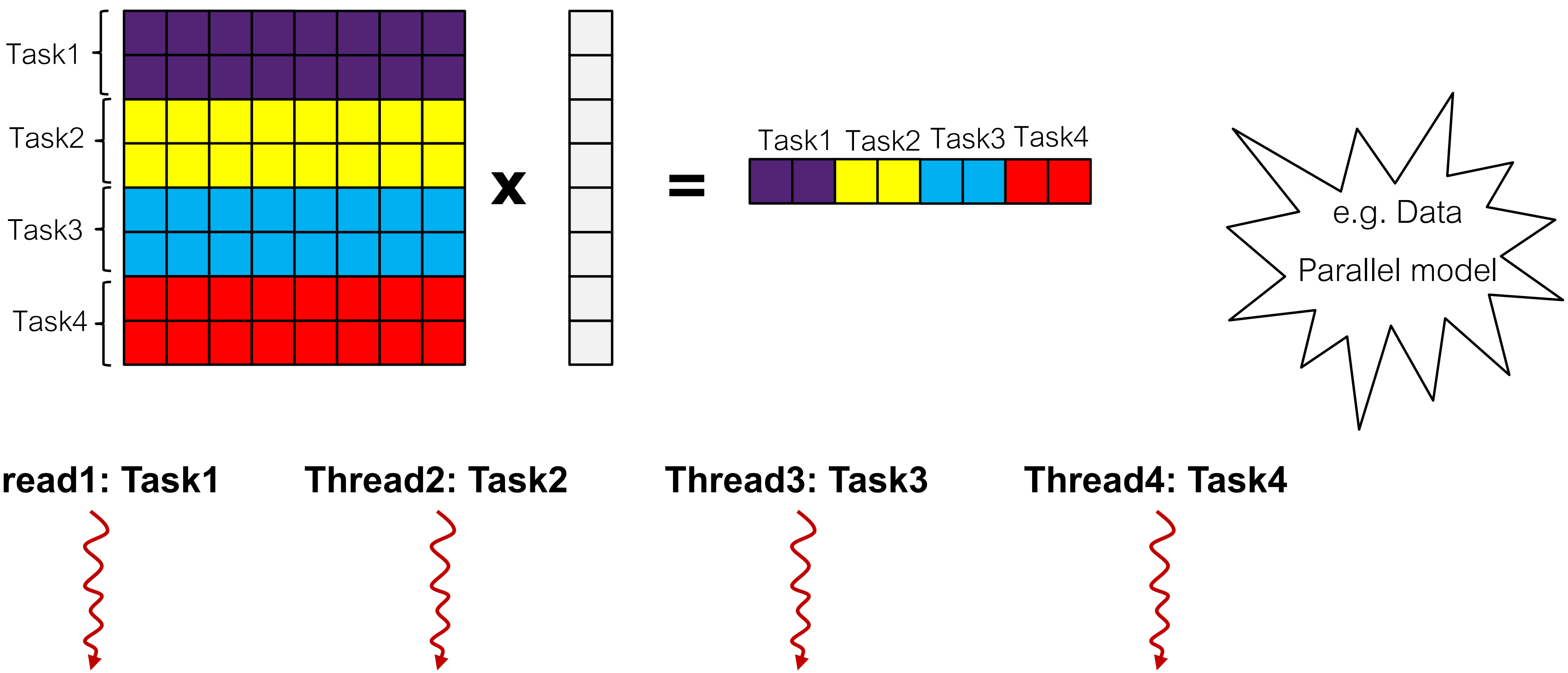
- Must ensure that computations and interactions are well-balanced at each stage

Static mapping

- Static mapping: assign tasks to processes before execution starts
- Static mapping allows for static load balancing
- Mapping quality depends on knowledge of task sizes, size of data associated with tasks, characteristics of task interactions, and parallel programming paradigm
- If task sizes not known => can potentially lead to severe load imbalances
- Usually done with static and uniform partitioning of data: data parallel problems!
- Tasks are tied to chunks of data generated by the partitioning approach
- Mapping tasks to processes essentially closely tied to mapping data to processes

Static mapping

- We create 4 tasks, each computing on 2 elements of c, and statically assign a process/thread to a task before execution. As you see our task assignment is tied to uniform partitioning of data!

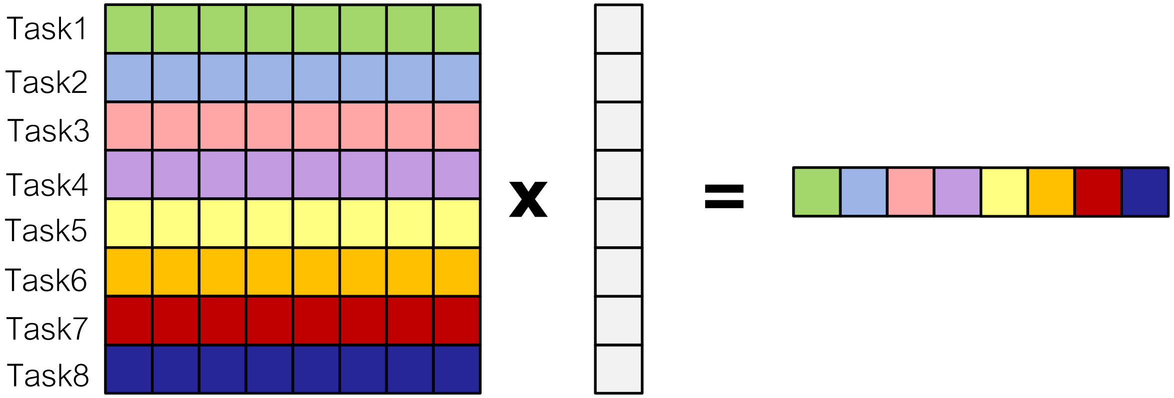


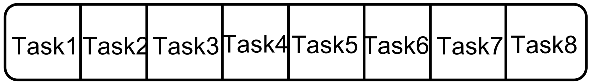
Dynamic mapping

- Dynamic mapping: assign tasks to processes during execution
- Dynamic mapping allows for dynamic load balancing
    - If task sizes are unknown => dynamic mappings are more effective than static ones
    - If much more data than computation => large overheads for data movement => static may be preferable
    - Depends on the parallel paradigm and interaction type though (shared address space vs distributed memory, read-only vs read-write interaction, etc.)

Common scheme for dynamic mapping

- Keep tasks in a centralized pool of tasks, assign them as processes become idle
  - The process managing the pool of ready tasks = master process
  - Other processes performing the tasks = worker processes, or slaves
- Tasks may get added to the pool, concurrently with the workers taking tasks out
- e.g., matrix-vector multiplication: task pool has tasks that each computes an item in c:

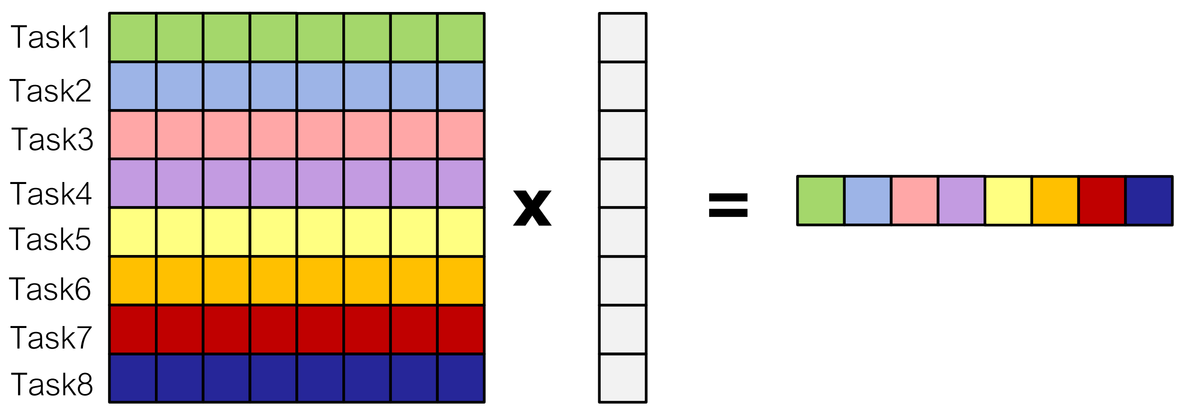


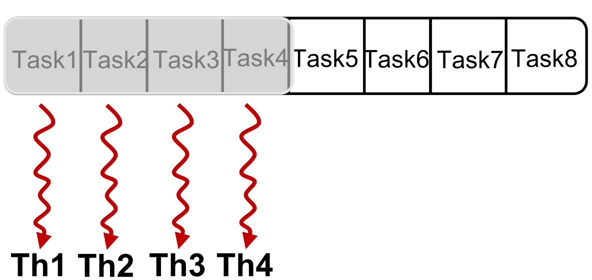


You can create a work pool where the tasks are put inside a queue and the next free thread will grab the next available task.

Common scheme for dynamic mapping

- Keep tasks in a centralized pool of tasks, assign them as processes become idle
  - The process managing the pool of ready tasks = master process
  - Other processes performing the tasks = worker processes, or slaves
- Tasks may get added to the pool, concurrently with the workers taking tasks out
- e.g., matrix-vector multiplication: task pool has tasks that each computes an item in c:

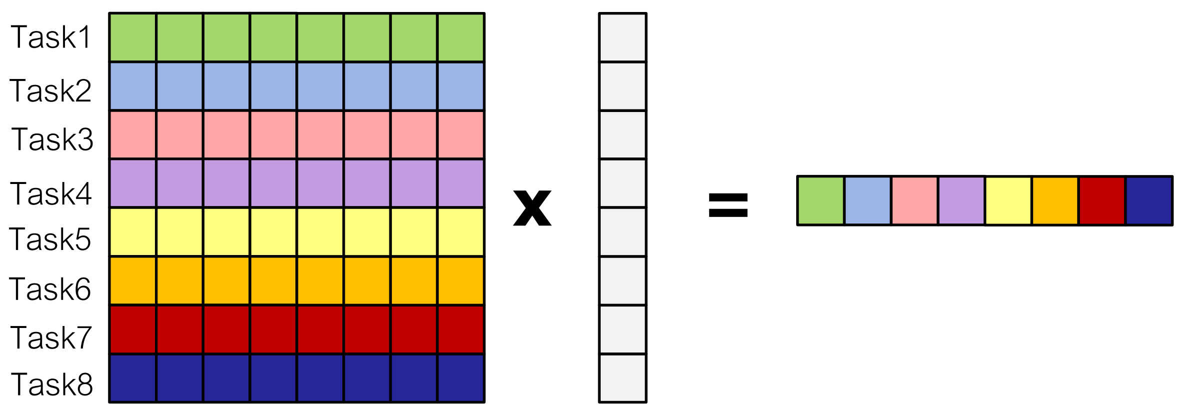


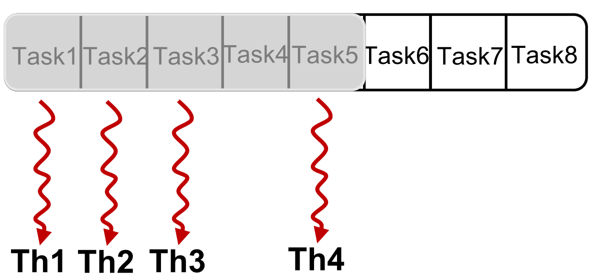


Each thread grabs a task from the work pool.

Common scheme for dynamic mapping

- Keep tasks in a centralized pool of tasks, assign them as processes become idle
  - The process managing the pool of ready tasks = master process
  - Other processes performing the tasks = worker processes, or slaves
- Tasks may get added to the pool, concurrently with the workers taking tasks out
- e.g., matrix-vector multiplication: task pool has tasks that each computes an item in c:

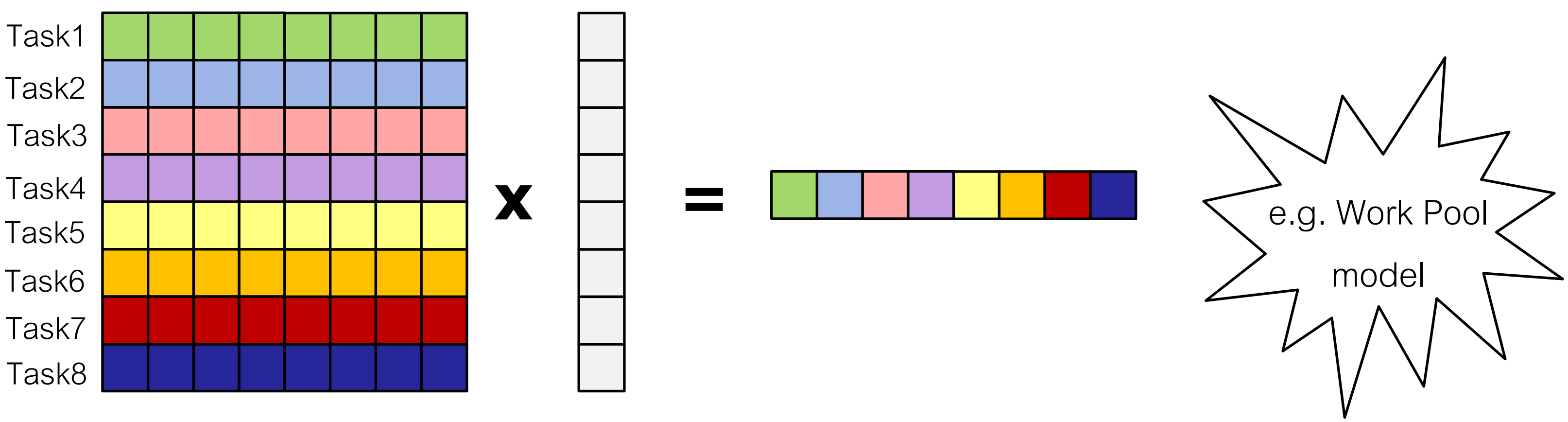


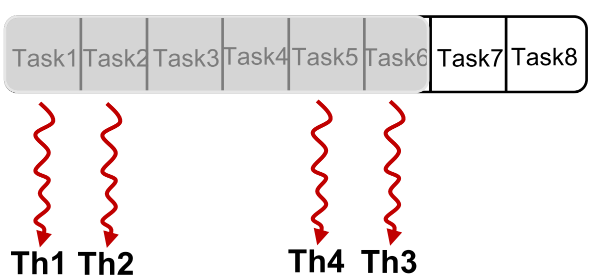


Thread 4 finished its work on task 4 and is now ready to start working on the next available task (task 5), the other threads are still working on their initially assigned tasks!

Common scheme for dynamic mapping

- Keep tasks in a centralized pool of tasks, assign them as processes become idle
  - The process managing the pool of ready tasks = master process
  - Other processes performing the tasks = worker processes, or slaves
- Tasks may get added to the pool, concurrently with the workers taking tasks out
- e.g., matrix-vector multiplication: task pool has tasks that each computes an item in c:





Thread 3 finished its work on task 3 and is now ready to start working on the next available task (task 6). This will go on until the work pool is empty!

Methods for containing interaction overheads

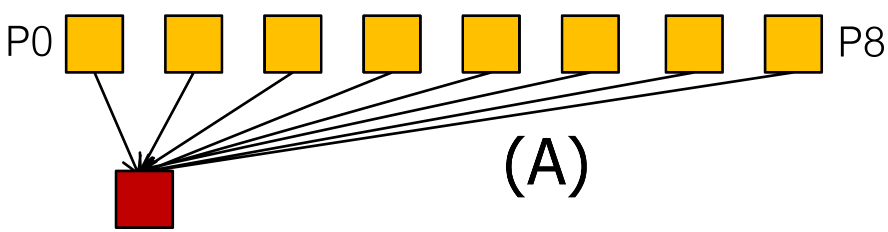
- Maximizing data locality
- Minimizing contention and hot spots
- Overlapping computations with interactions
- Replicating data or computations

Maximize data locality

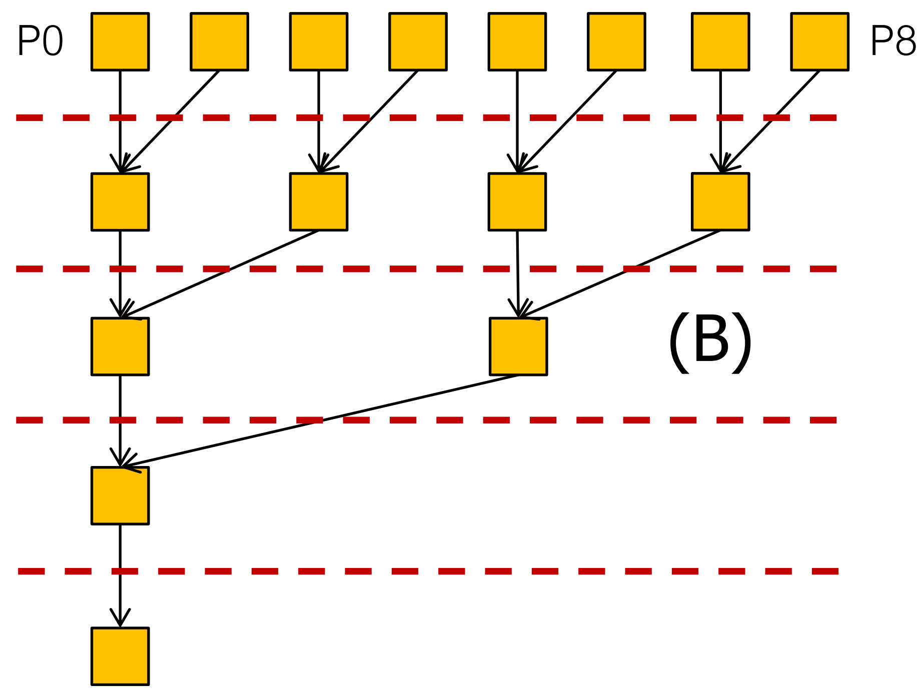
- Processes share data and/or may require data generated by other processes
- Goals:
  - 1. Minimize the volume of interaction overheads (minimize non-local data accesses and maximize local data utilization)
    - Minimize the volume of shared data and maximize cache reuse
    - Use a suitable decomposition and mapping
    - Use local data to store intermediate results (decreases the amount of data exchange)
  - 2. Minimize the frequency of interactions
    - Restructure the algorithm to access and use large chunks of shared data (amortize interaction cost by reducing frequency of interactions)
    - Shared address space: spatial locality in a cache line, etc.

Minimizing contention and hotspots

- Accessing shared data concurrently can generate contention
  - e.g., concurrent accesses to the same memory block, flooding a specific process with messages, etc.
- Solutions:
  - Restructure the program to reorder accesses in a way that does not lead to contention
  - Decentralize the shared data, to eliminate the single point of contention



Reducing the elements of an array: the threads all need to access the red block in (A) while (B) decentralizes this single point of contention.



Overlap computations with interactions

- Process may idle waiting for shared data => do useful computations while waiting
- Strategies:
  - Initiate an interaction earlier than necessary, so it's ready when needed
  - Grab more tasks in advance, before current task is completed
- May be supported in software (compiler, OS), or hardware (e.g., prefetching hardware, etc.).
- Harder to implement with shared memory models (Pthreads, OpenMP), applies more to distributed and GPU architectures (more on it later in class!)

Replicate data or computations

- To reduce contention for shared data, may be useful to replicate it on each process => no interaction overheads
- Beneficial if the shared data is accessed in read-only fashion
    - Shared-address space paradigm: cache local copies of the data
    - Message-passing paradigm (MPI), more on this later: replicate data to eliminate data transfer overheads
- Disadvantages:
    - Increases overall memory usage by keeping replicas => use sparingly
    - If shared data is read-write, must keep the copies coherent => overheads might dwarf the benefits of local accesses via replication

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

Parallel algorithm models

Model = 1 decomposition type + 1 mapping type + strategies to minimize interactions

Commonly used parallel algorithm models:

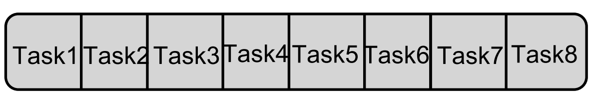
- Data parallel model
- Work pool model
- Master slave model

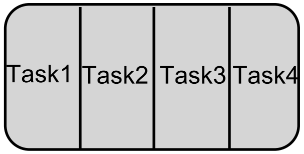
Data parallel model

- Decomposition: typically static and uniform data partitioning
- Mapping: static (mostly)
- Same operations on different data items, aka data parallelism
- Possible optimization strategies (depending on the problem and paradigm):
  - Choose a locality-preserving decomposition
  - Overlap computation with interaction
- This model scales really well with problem size (by adding more processes)

Work pool model

- Tasks are taken by processes from a common pool of work
- Decomposition: highly depends on the problem (data, recursive, etc.)
    - Can be statically available at start, or dynamically create more tasks during execution
- Mapping: dynamic
    - Any task can be performed by any process
- Possible strategies for reducing interactions:
    - Adjust granularity: tradeoff between load imbalance and overhead of accessing work pool





Master slave model

- Commonly used in distributed parallel architectures (more on this later)
- A master process generates work and allocates to worker (slave) processes
  - Could involve several masters, or a hierarchy of master processes
- Decomposition: highly depends on the problem (data, recursive)
  - Might be static if tasks are easy to break down a priori, or dynamic
- Mapping: Often dynamic
  - Any worker can be assigned any of the tasks
- Possible strategies for reducing interactions:
  - Choose granularity carefully so that master does not become a bottleneck
  - Overlap computation on workers with master generating further tasks