SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Decomposition & Mapping

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Reading

- Introduction to Parallel Computing (Second Edition) by Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar
  - Chapter 3: Principles of Parallel Algorithm Design
  - Online Access: https://sfu-primo.hosted.exlibrisgroup.com/permalink/f/usv8m3/01SFUL_ALMA51188913690003611

Constructing Parallel Algorithms

- Typical steps for constructing a parallel algorithm
  - Identify pieces of work that can be performed concurrently
  - Partition and map 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

- Why "typical"? Some of the steps may be omitted.
  - If data is in shared memory, distributing it may be unnecessary
  - If using message passing, there may not be shared data
  - Mapping of work to processors can be done statically by the programmer or dynamically by the runtime

Task Decomposition & Mapping

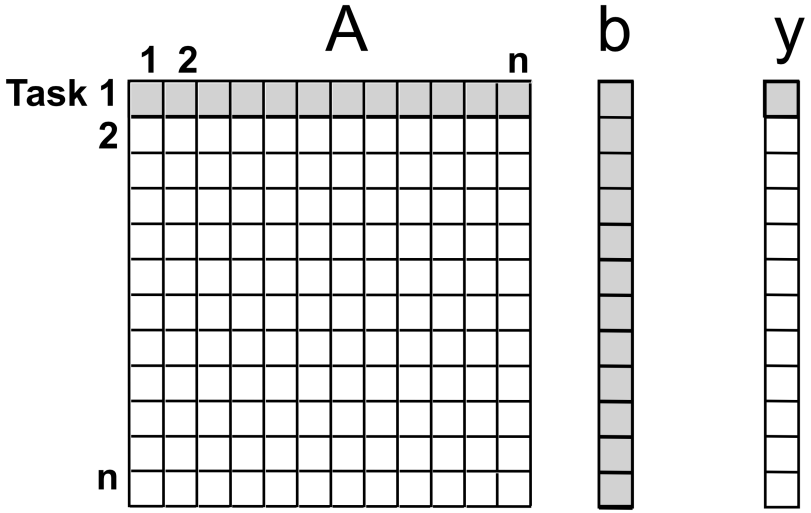
- Standard approaches to think/design parallel solutions
- With a flavor of High Performance Computing (HPC)

- Topics
  - Identifying and analyzing tasks
  - Task dependency graphs, task interaction graphs
  - Task decomposition strategies
  - Task mapping strategies
  - Common design goals and practices

Decomposing Work for Parallel Execution

- Divide work into tasks that can be executed concurrently
- Many different decompositions possible
- Tasks may be same, different, or even indeterminate sizes
- Tasks may be independent or have non-trivial order

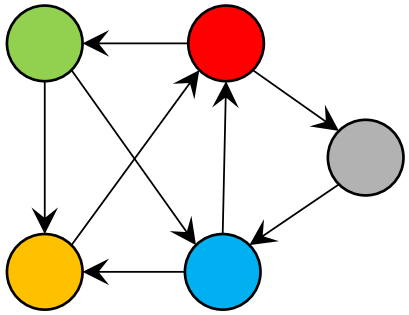
Example: Matrix-Vector Multiplication



- Computing each element of output vector y is independent
- Task decomposition: one task per element in y
- Observations
  - Task size is uniform
  - Tasks share b

Example: PageRank

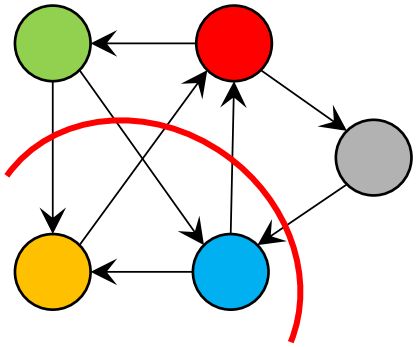
- $\text{pagerank}[v] = 0.15 + 0.85 \times \sum_{\{u \in \text{in}(v)\}} \frac{\text{pagerank}[u]}{\text{degree}(u)}$



- Computing rank of each vertex is independent
- Task decomposition: one task per vertex in graph
- Observations
  - Task size is not uniform
  - Tasks share vertices, edges and rank values

Example: PageRank

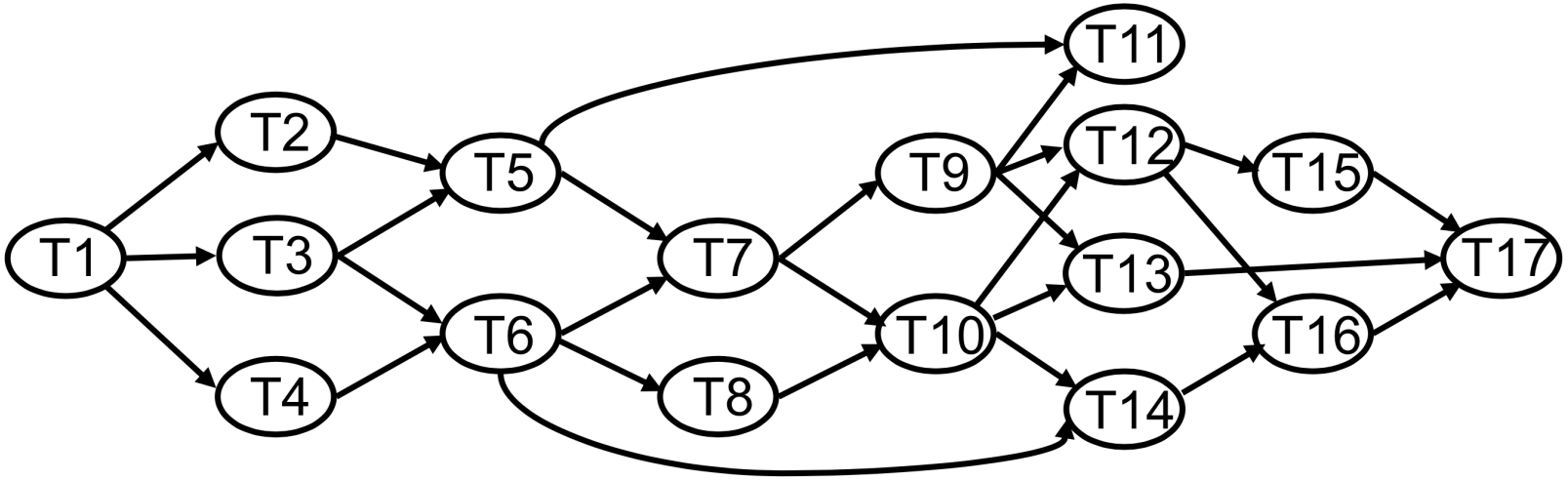
- $\text{pagerank}[v] = 0.15 + 0.85 \times \sum_{\{u \in \text{in}(v)\}} \frac{\text{pagerank}[u]}{\text{degree}(u)}$



- Alternate decomposition: one task per subgraph
- Observations
  - Task size can be made (nearly) uniform
  - Tasks share vertices, edges and rank values
  - Better than previous decomposition?

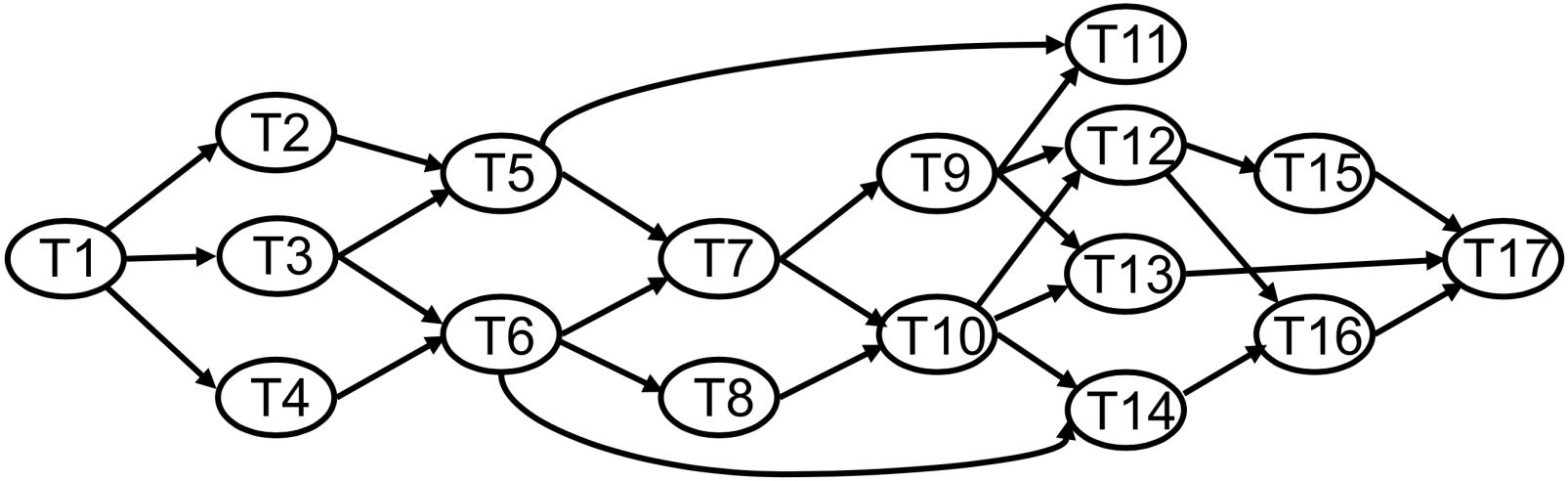
Task Dependence Graph

- Conceptualize tasks and ordering as a Directed Acyclic Graph (DAG)
  - Vertex = task
  - Edge = control dependence



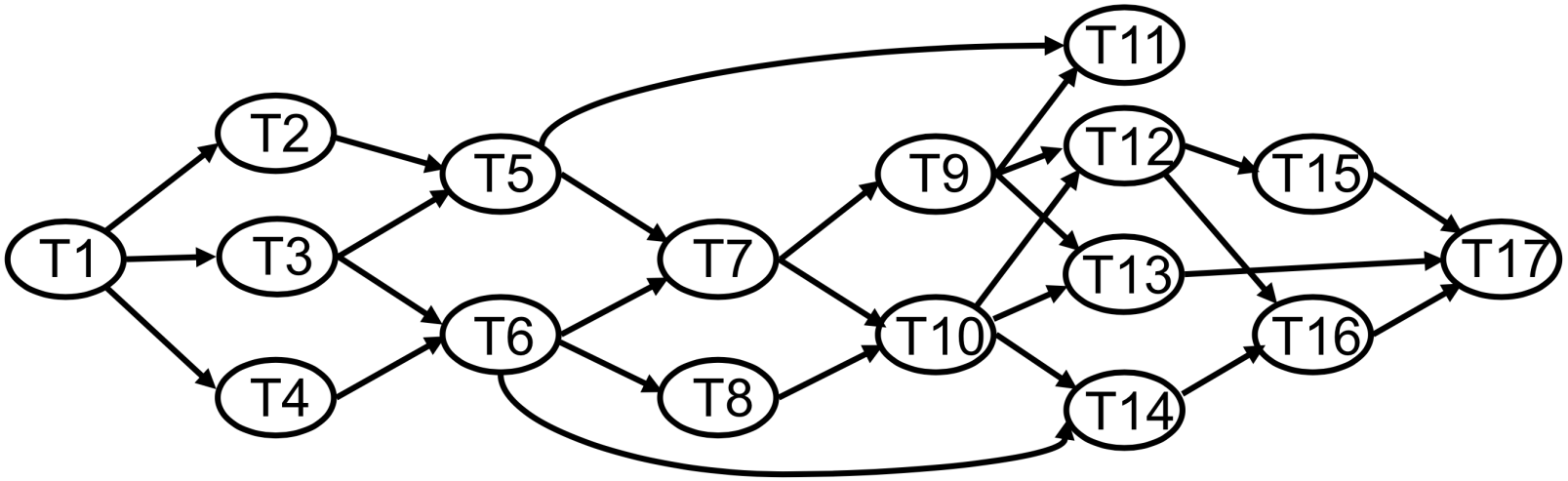
Degree of Concurrency

- Number of tasks that can execute in parallel
- May change during program execution
- Maximum degree of concurrency
  - Maximum number of tasks that can be processed in parallel

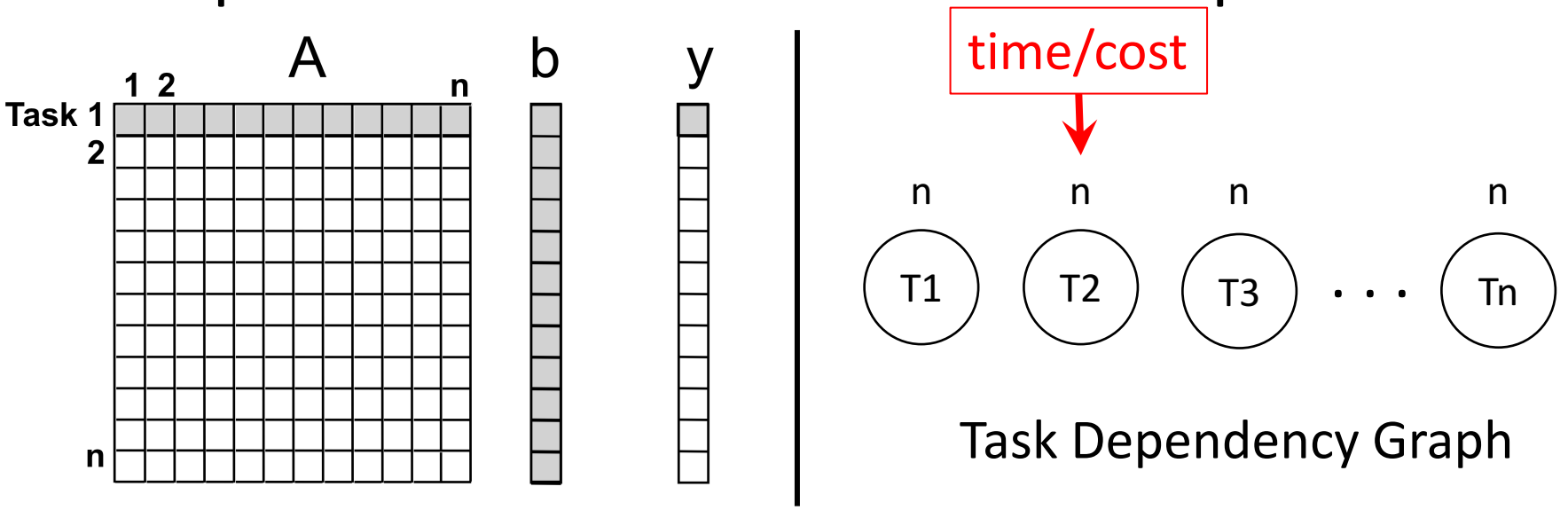


Critical Path

- Edges in task dependency graph represent task serialization
- Critical path = longest weighted path through graph
- Critical path length gives lower bound on parallel execution time
- Average degree of concurrency = $\frac{\text{total work}}{\text{critial path length}}$

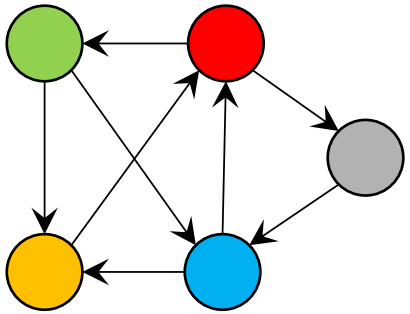


Example: Matrix-Vector Multiplication

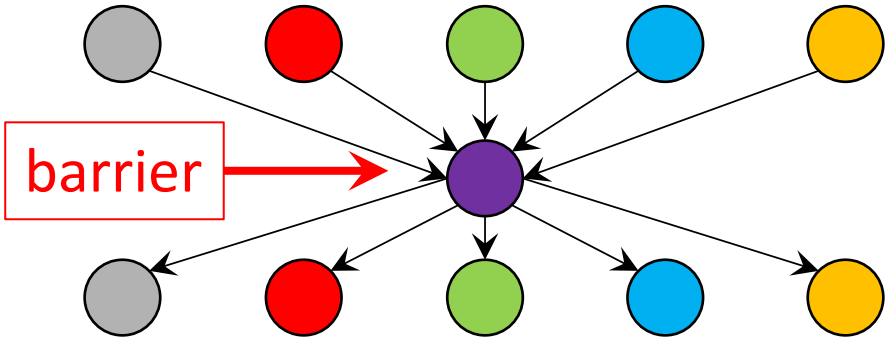


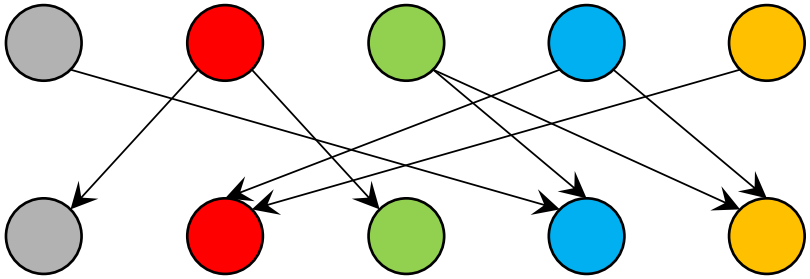
- Critical path length? 1 task
- Shortest parallel execution time? n operations (# of cols)
- Maximum degree of concurrency? n (# of rows)
- Average degree of parallelism? n

Example: PageRank

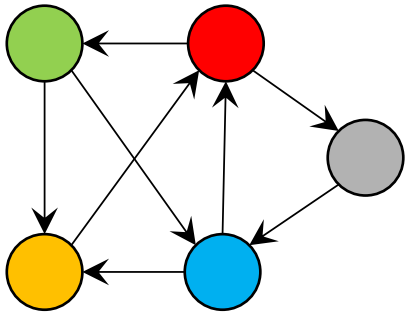


- Iterative execution
- Bulk Synchronous Parallel semantics
- $\text{pagerank}[v] = 0.15 + 0.85 \times \sum_{\{u \in \text{in}(v)\}} \frac{\text{pagerank}[u]}{\text{degree}(u)}$

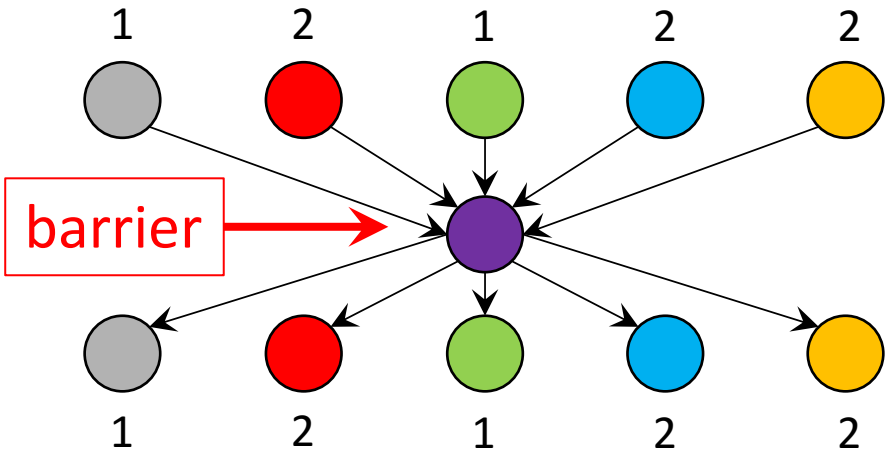


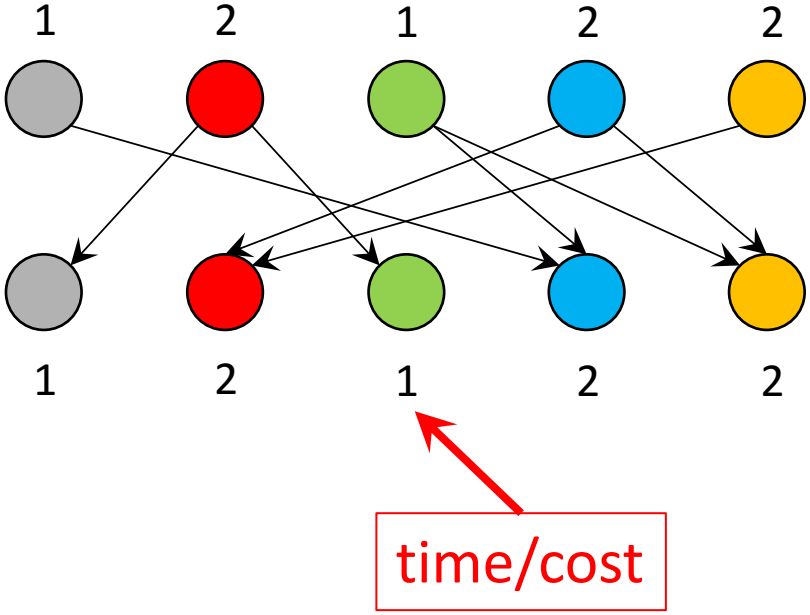


Example: PageRank



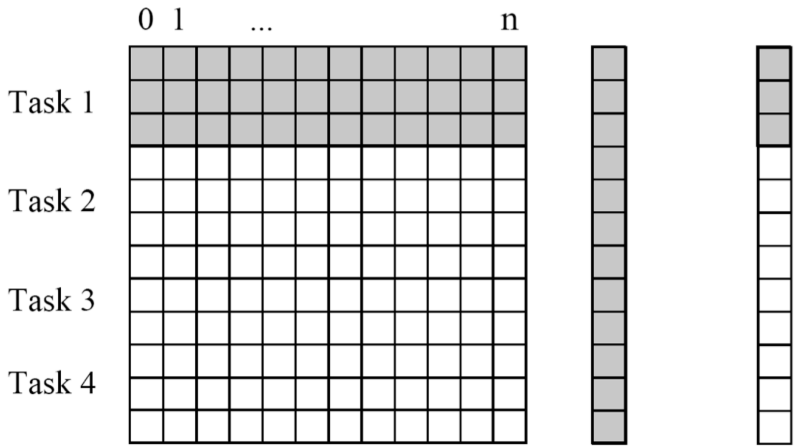
- Iterative execution
- Bulk Synchronous Parallel semantics
- pagerank[v] = $0.15 + 0.85 \times \sum_{\{u \in in(v)\}} \frac{pagerank[u]}{degree(u)}$



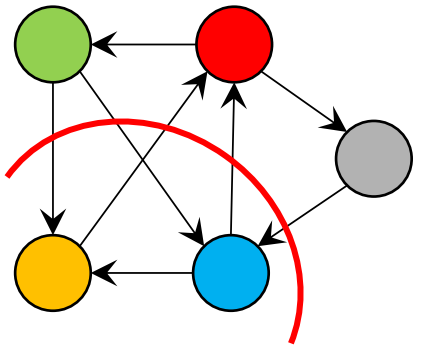


Granularity of Task Decompositions

- Granularity = task size
- Fine-grained = large number of tasks
- Coarse-grained = small number of tasks



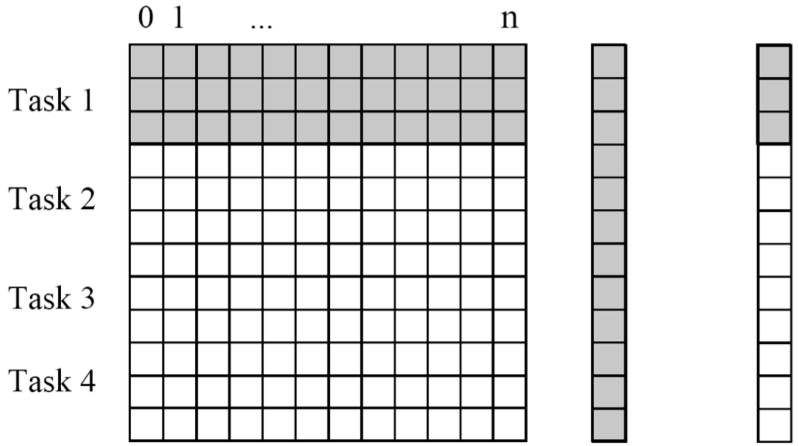
Each task computing 3 elements of vector



Each task computing over multiple vertices

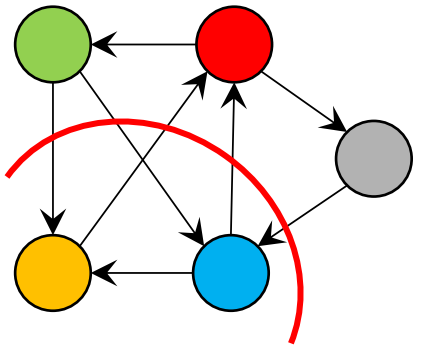
Granularity of Task Decompositions

- Granularity = task size
- Degree of concurrency v/s task granularity?
  - Fine-grained = more concurrency



Each task computing 3 elements of vector

What is the most fine-grained decomposition?



Each task computing over multiple vertices

Limits on Parallel Performance

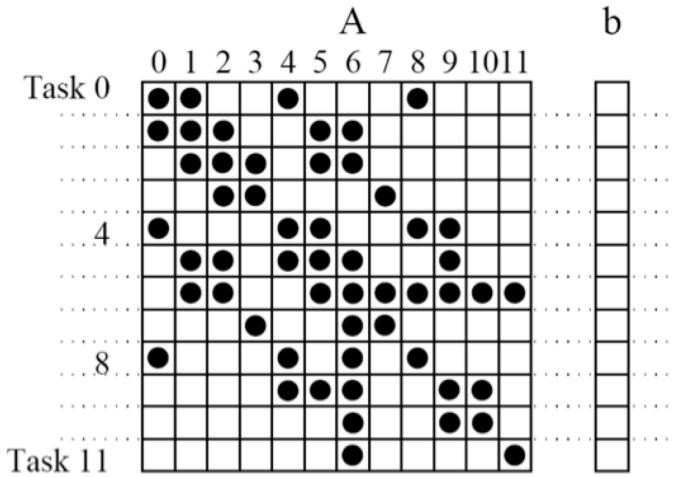
- What bounds parallel execution time?
  - Fraction of application work that can't be parallelized (recall Amdahl's law)
  - Minimum task granularity
    - e.g., matrix-vector multiplication <= n^2 concurrent tasks
  - Synchronization & communication to satisfy dependencies

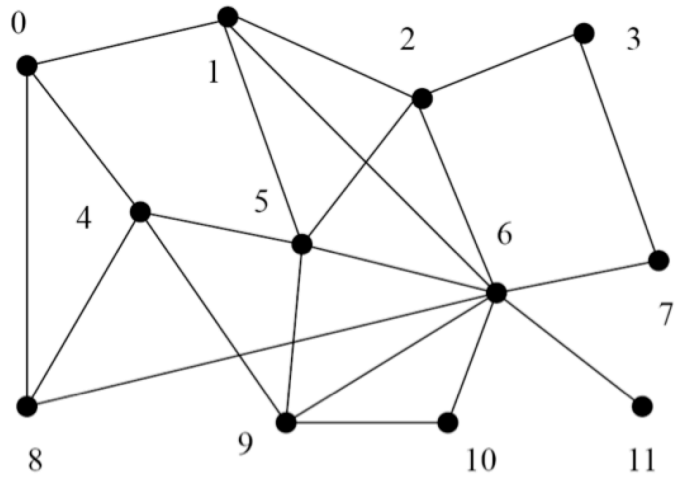
Task Interaction Graphs

- Tasks generally exchange data with others
  - In matrix-vector multiplication, if vector b is not replicated in all tasks, tasks will have to communicate elements of b
- Task interaction graph
  - Vertex = task
  - Edge = interaction of data exchange
- Task interaction graphs v/s task dependency graphs
  - Task interaction graphs represent data dependences
  - Task dependency graphs represent control dependences

Example: Sparse Matrix-Vector Multiplication

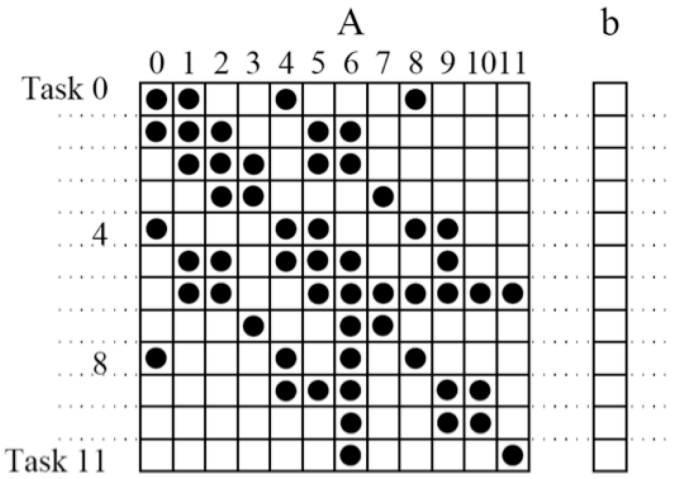
- Computation of each result element = independent task
- Only non-zero elements of sparse matrix A participate
- Vector b is partitioned among tasks

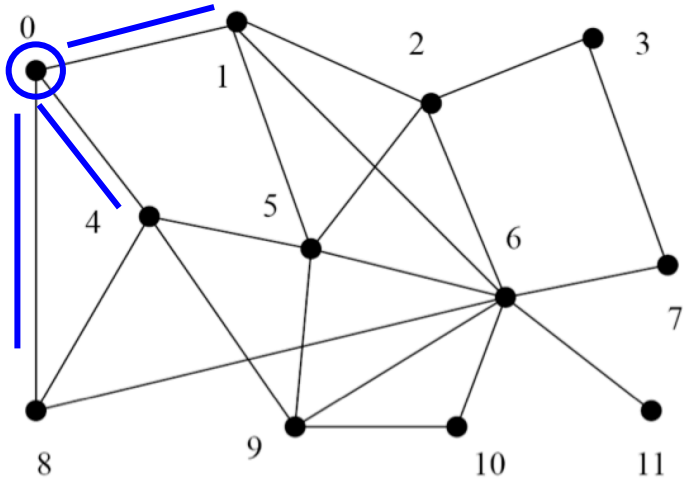




Interaction Graphs

- Finer task granularity increases communication overhead

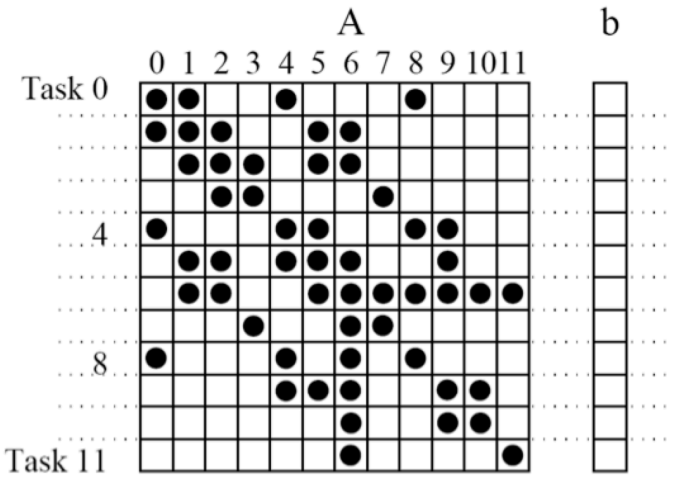


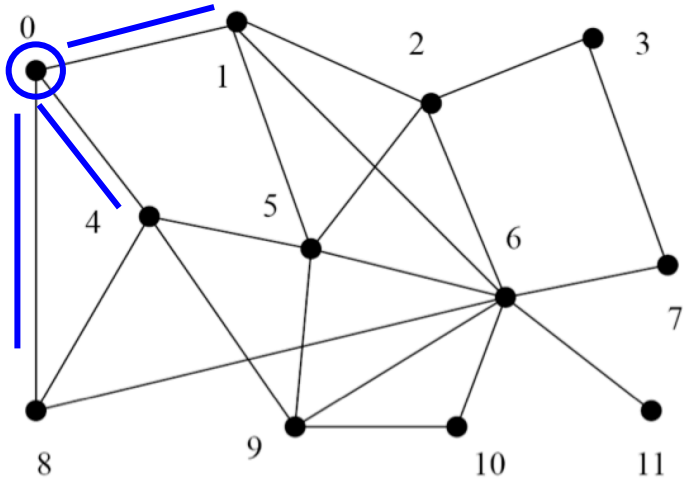


- Assumptions:
  - Each multiplication+addition takes unit time to process
  - Each interaction (edge) causes an overhead of a unit time
- If node 0 is a task: communication = **?**; computation = **?**

Interaction Graphs

- Finer task granularity increases communication overhead

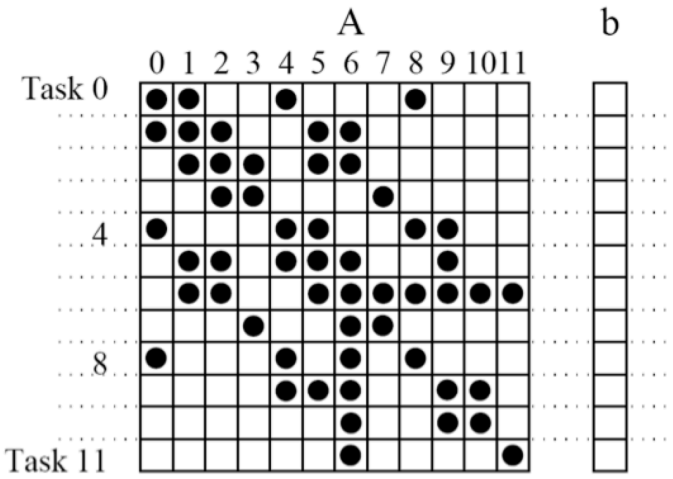


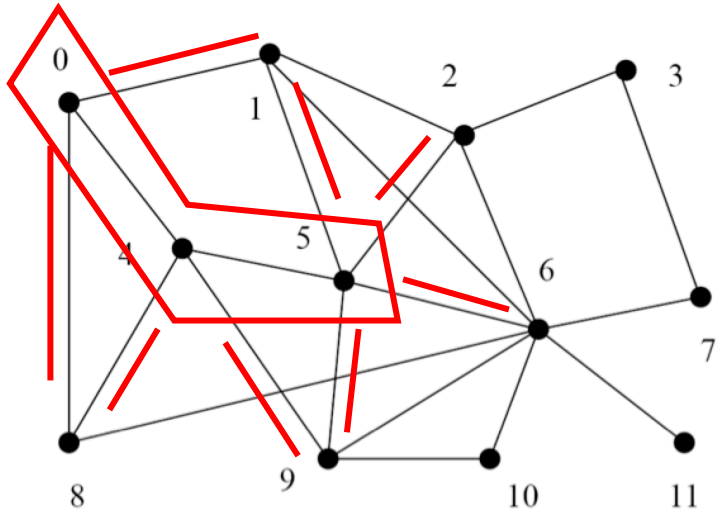


- Assumptions:
  - Each multiplication+addition takes unit time to process
  - Each interaction (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4

Interaction Graphs

- Finer task granularity increases communication overhead

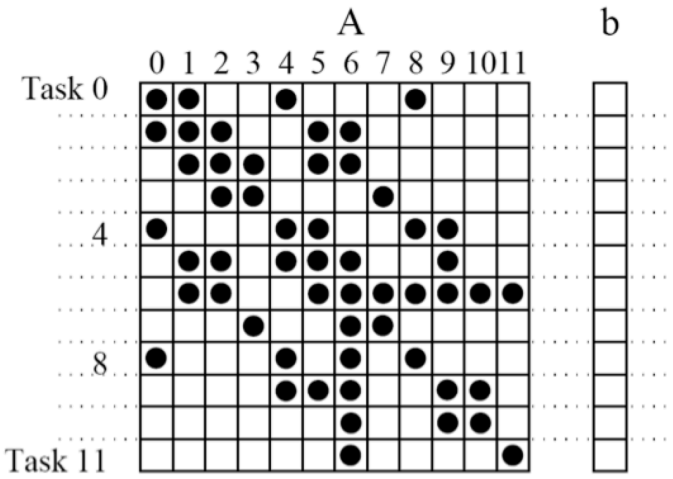


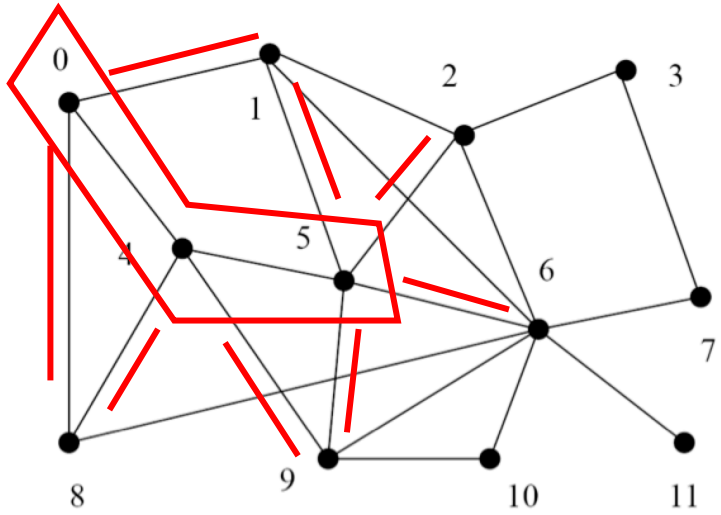


- Assumptions:
  - Each multiplication+addition takes unit time to process
  - Each interaction (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4
- If nodes 0, 4, 5 are a task: communication = **?**; computation = **?**

Interaction Graphs

- Finer task granularity increases communication overhead





- Assumptions:
  - Each multiplication+addition takes unit time to process
  - Each interaction (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4
- If nodes 0, 4, 5 are a task: communication = 5; computation = 15

Decomposition Techniques

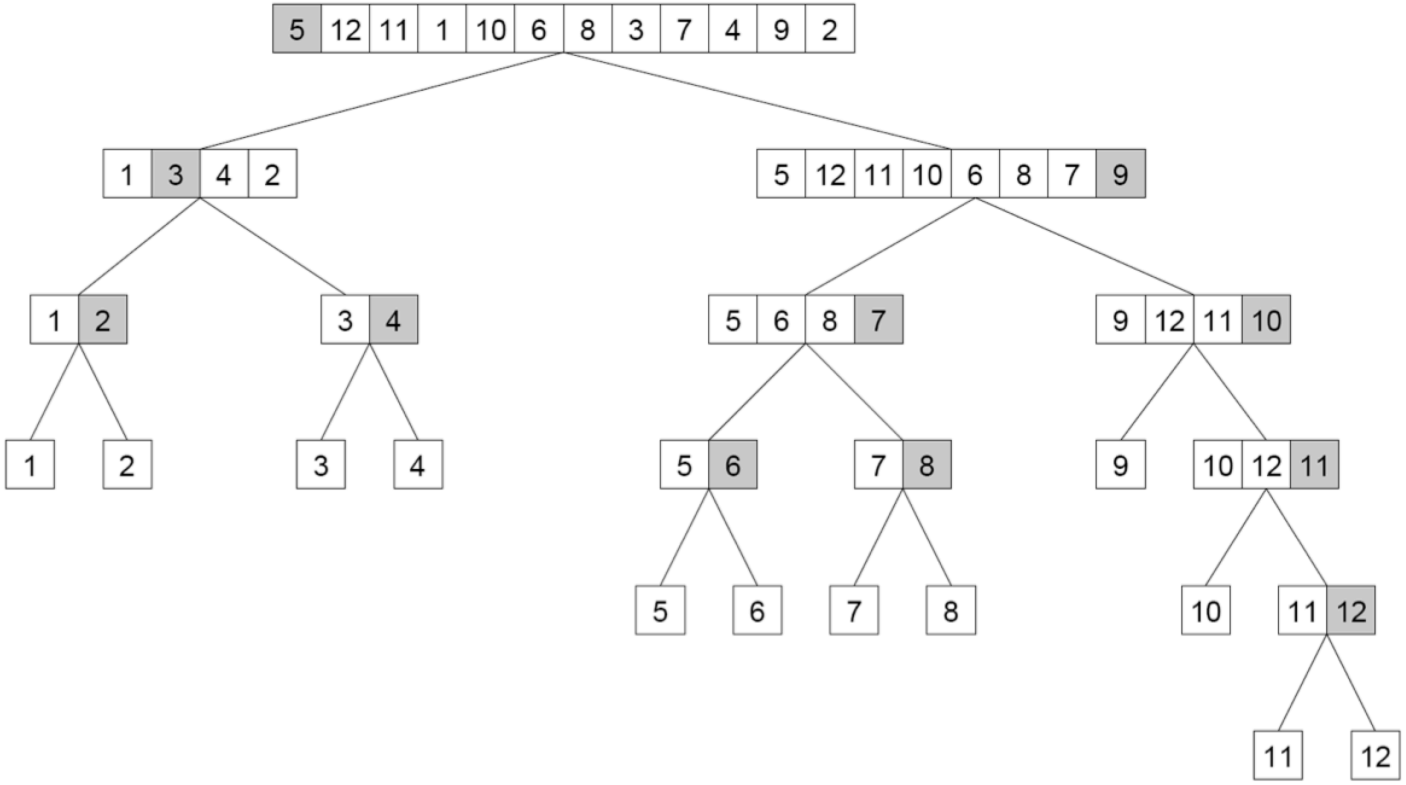
- How should one decompose a task into various subtasks?
- No single universal recipe
- In practice, a variety of techniques are used including
  - Recursive decomposition
  - Data decomposition
  - Exploratory decomposition

Recursive Decomposition

- Suitable for problems solvable using divide-and-conquer
- Steps:
  - Decompose a problem into a set of sub-problems
  - Recursively decompose each sub-problem
  - Stop decomposition when minimum desired granularity reached

Recursive Decomposition for Quicksort

- Select a pivot
- Partition vector v around pivot into v_{left} and v_{right}
- In parallel, sort v_{left} and sort v_{right}



Recursive Decomposition for Min

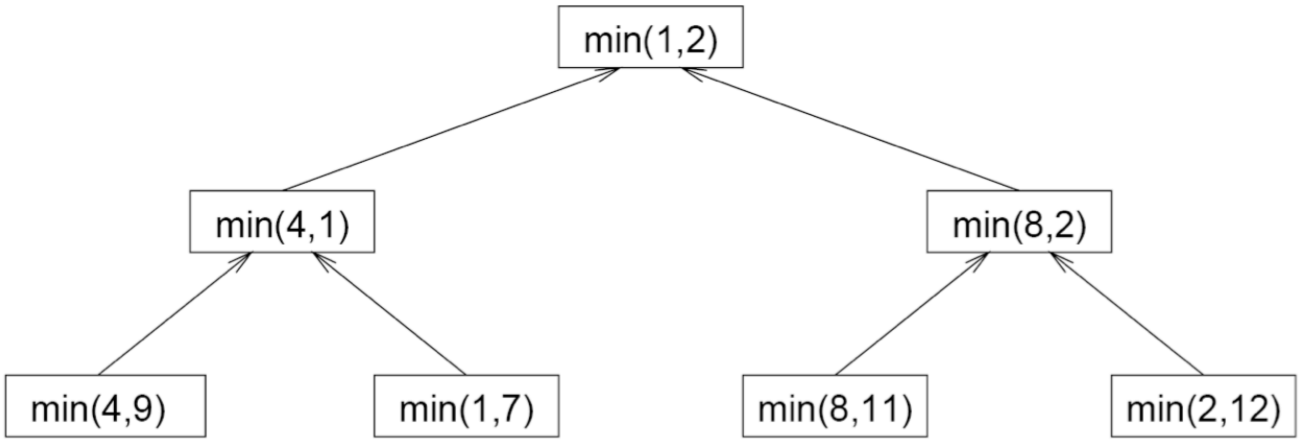
- Finding minimum element in a vector

```
procedure SERIAL_MIN (A, n)
begin
min = A[0];
for i := 1 to n − 1 do
    if (A[i] < min) min := A[i];
endfor;
return min;
end SERIAL_MIN
```

```
procedure RECURSIVE_MIN (A, n)
begin
if (n = 1) then
    min := A[0];
else
    lmin := RECURSIVE_MIN (A, n/2);
    rmin := RECURSIVE_MIN (&(A[n/2]), n − n/2);
    if (lmin < rmin) then
        min := lmin;
    else
        min := rmin;
    endelse;
endelse;
return min;
end RECURSIVE_MIN
```

Recursive Decomposition for Min

- v = [4, 9, 1, 7, 8, 11, 2, 12]



Data Decomposition

- Steps:
  - Identify the data on which computations are performed
  - Partition the data across various tasks
    - Partitioning induces a decomposition of the problem
- Data can be partitioned in various ways
  - Appropriate partitioning is critical to parallel performance
- Decomposition can be based on:
  - Input data
  - Output data
  - Input + Output data
  - Intermediate data

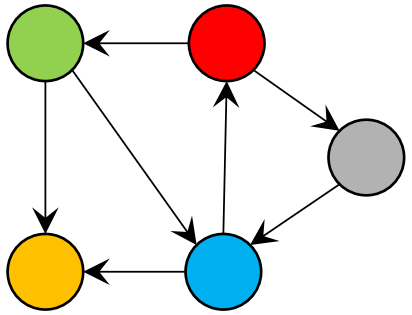
Often similar, just different perspectives

Decomposition Based on Input Data

- Applicable when 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
- Example: quicksort, finding minimum

Example: Degree Distribution

- Find the distribution of in/out degrees in the graph
- Useful to analyze structural properties
- Decomposition based on graph vertices and not based on the degrees
  - Task = degree computation for each vertex



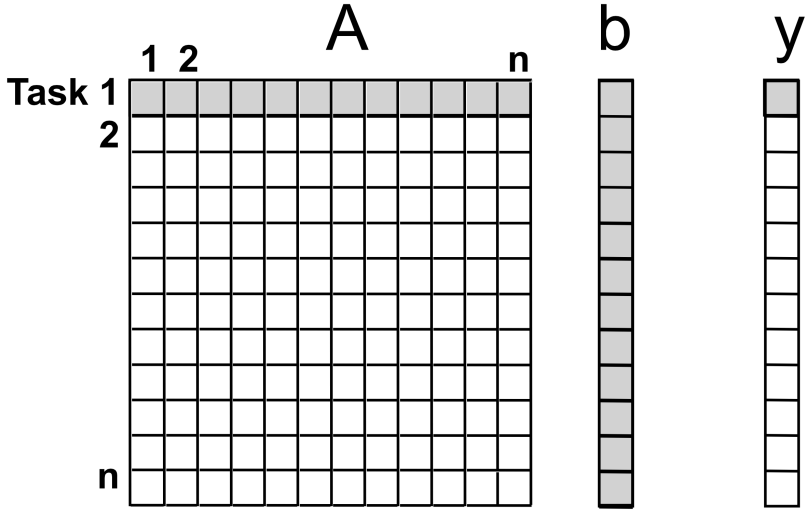
- Input decomposition useful when output space is not known

In-Degree	Count
1	3
2	2

Out-Degree	Count
0	1
1	1
2	3

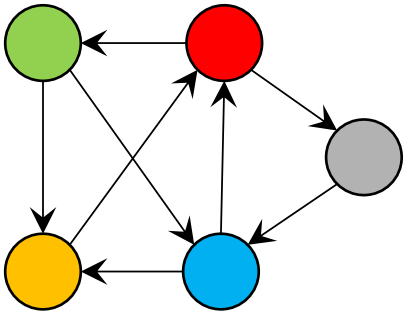
Decomposition Based on Output Data

- Applicable when each element of the output can be computed independently
- Partition the output data across tasks
- Have each task perform the computation for its outputs
- Example: matrix-vector multiplication



Example: PageRank

- $\text{pagerank}[v] = 0.15 + 0.85 \times \sum_{\{u \in \text{in}(v)\}} \frac{\text{pagerank}[u]}{\text{degree}(u)}$
- Decomposition based on ranks
  - Task = rank computation for each vertex
  - Ultimately results in decomposing based on vertices



- Decomposition based on input data and output data often lead to similar tasks

Intermediate Data Partitioning

- Applicable when computation is a sequence of transformations from input data to output data
- Can decompose based on data for intermediate stages

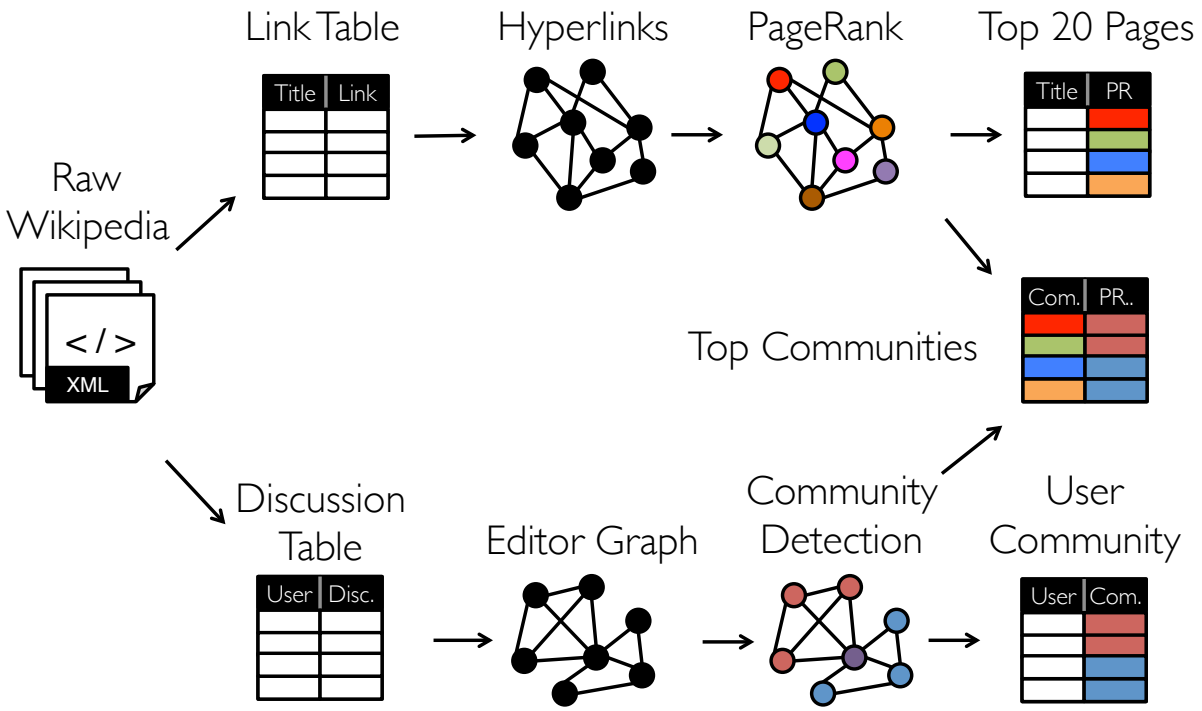


Figure: https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_gonzalez.pdf

Intermediate Data Partitioning

- Each pair of consecutive stages analyzed w.r.t. input or output decomposition
  - Balance the cost v/s benefit of one over other

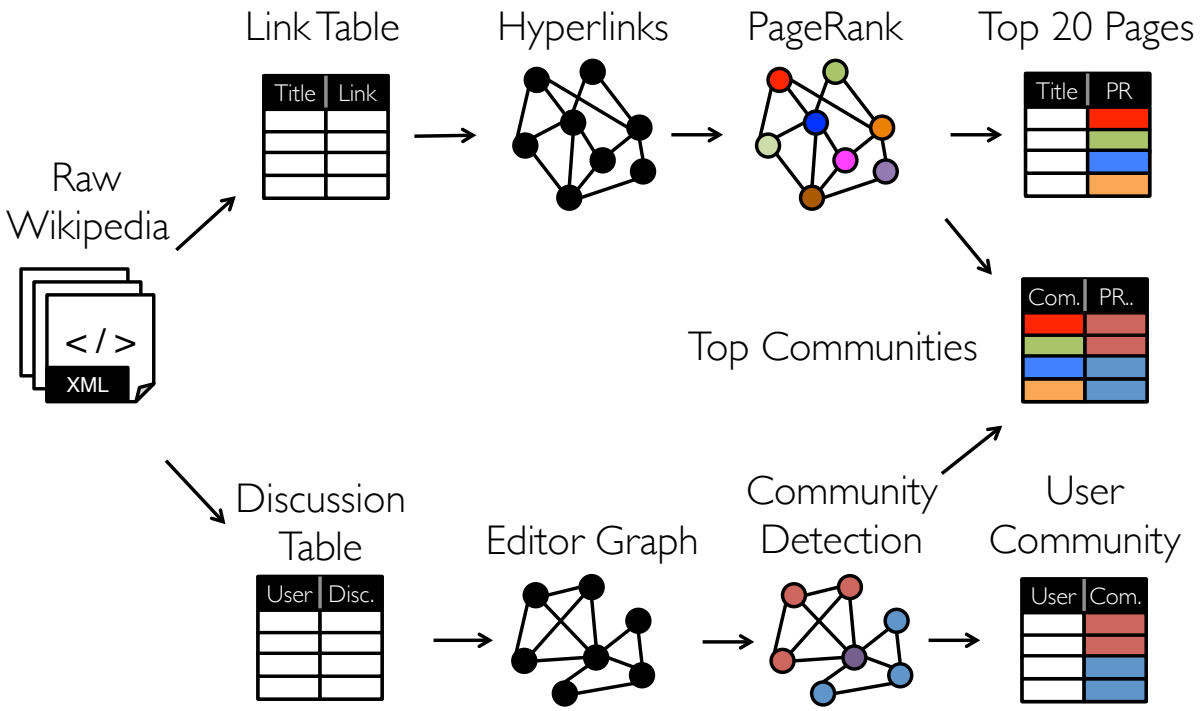


Figure: https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_gonzalez.pdf

Owner Computes Rule

- General rule to reduce communication/synchronization
- Each data element is assigned to a task
- Each task computes values associated with its data
- Input data decomposition
  - Task performs all computations using its input data
- Output data decomposition
  - Task computes all output data assigned to it

Exploratory Decomposition

- Search in a state space of solutions
- Examples: game playing, theorem proving, verification
- Partition search space into smaller search spaces
  - Parts to be searched concurrently

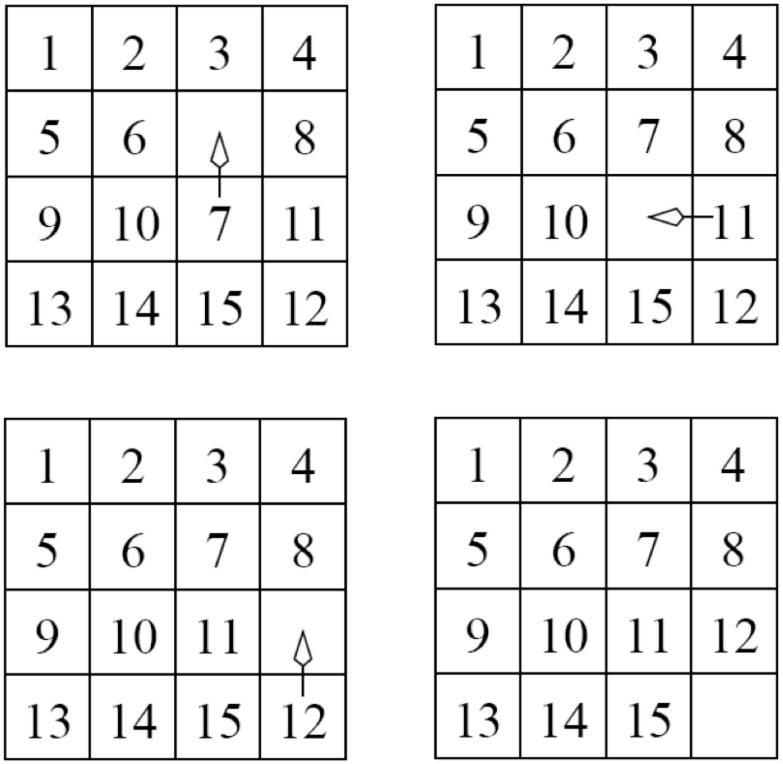
Searching in a Graph

```
visited[source] = true;
q.enqueue(source);
while(!q.empty()) {
   u = q.dequeue();
   for v in outNeighbors(u) {
      if(color[v] == goal_color)
         return v;
      if(visited[v] == false) {
         visited[v] = true;
         q.enqueue(v);
      }
   }
}
```

potential solutions dynamically result in tasks

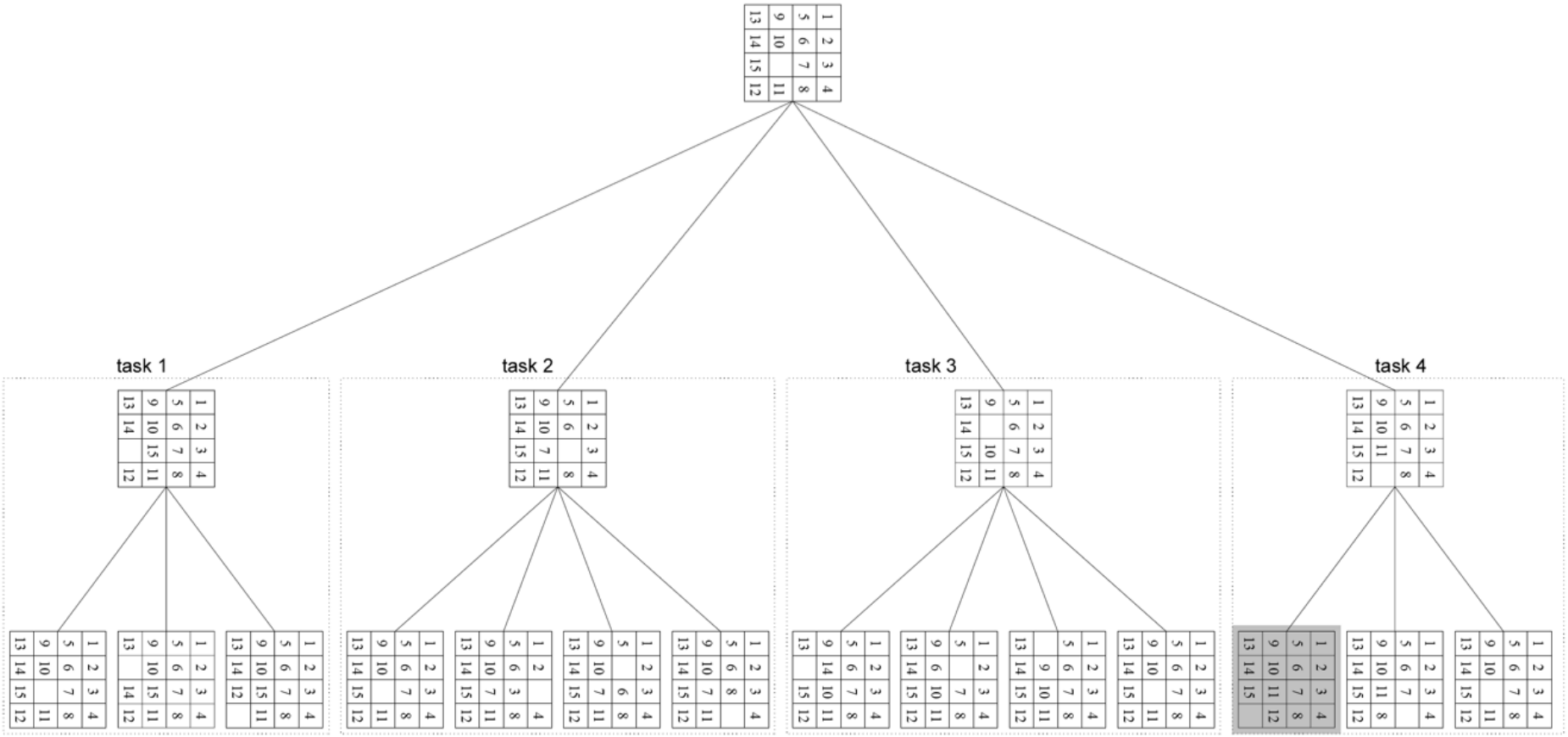
Example: Exploratory Decomposition

- Solving a 15-puzzle
- Sequence of three moves from initial state to final state



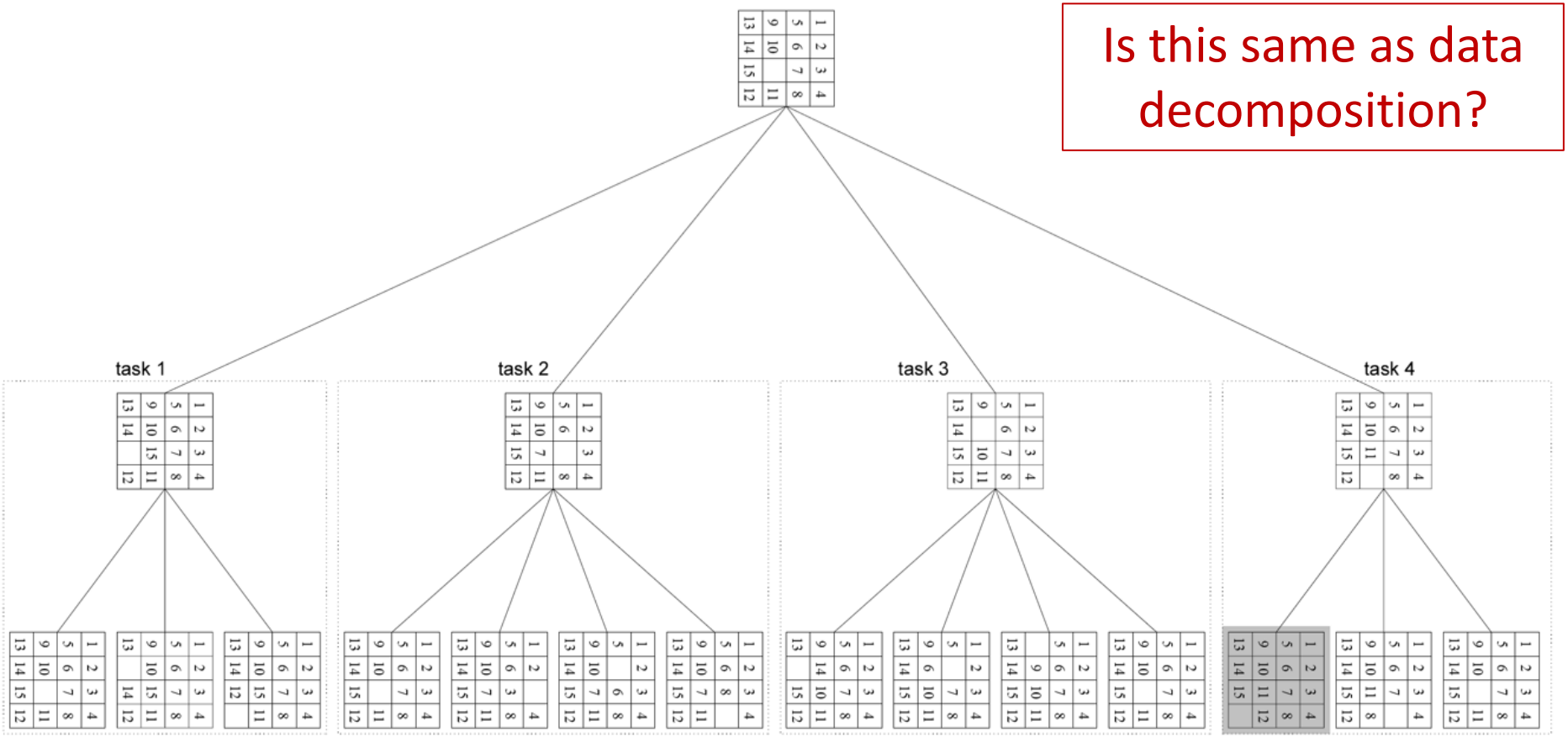
Example: Exploratory Decomposition

- Generate successor states of the current state
- Explore each as an independent task



Example: Exploratory Decomposition

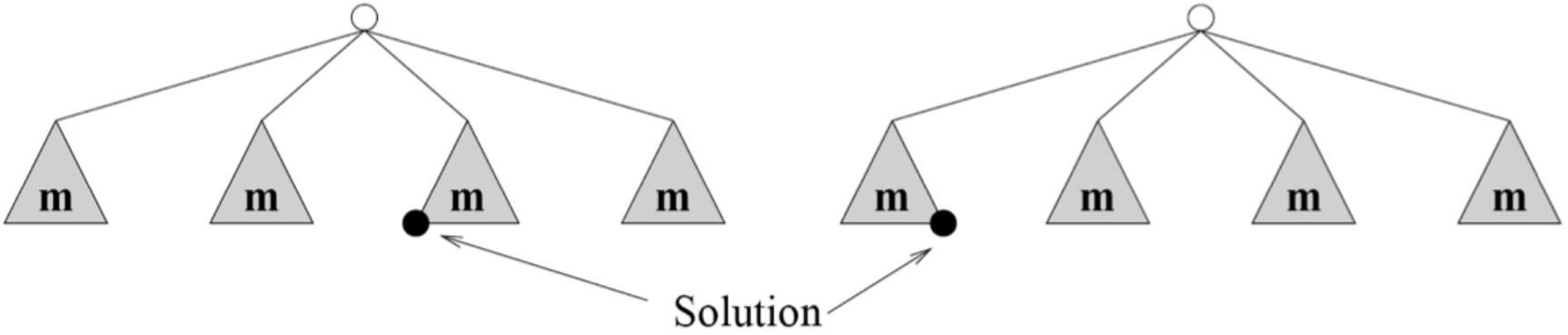
- Generate successor states of the current state
- Explore each as an independent task



Is this same as data decomposition?

Exploratory Decomposition Speedup

- Parallel formulation may perform different amount of work
- Can cause super-linear or sub-linear speedup



Total serial work: 2m+1
Total parallel work: 1

Total serial work: m
Total parallel work: 4m

Dynamic Task Generation

- Tasks might be known a-priori, but their execution might not be known
- Example: Shortest Paths
  - Task = each vertex (which is known)
  - Execution of task depends on how exploration proceeds

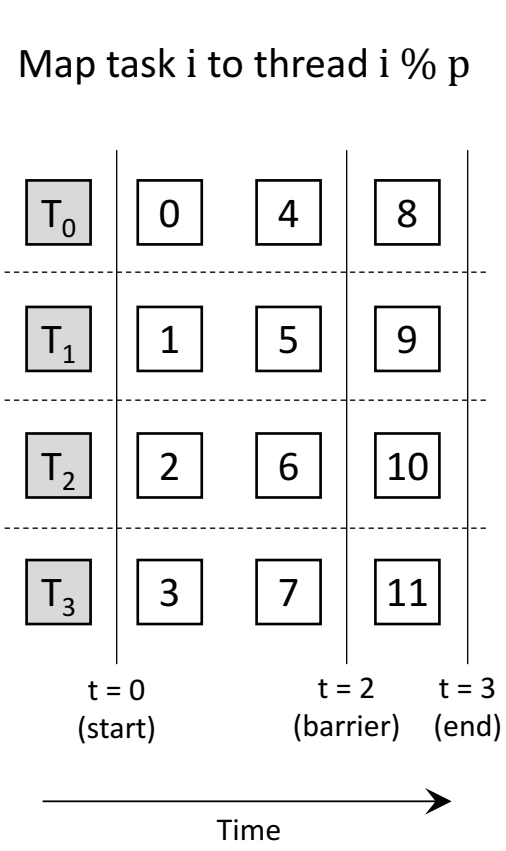
```
pq.enqueue(<0, source>);   // Priority Queue
while(!pq.empty()) {
    u = pq.dequeue().second;
    for v in outNeighbors(u) {
        if(d[u] + w(u, v) < d(v)) {
            d[v] = d[u] + w(u, v);
            pq.enqueue(<d[v], v>);
        }
    }
}
```

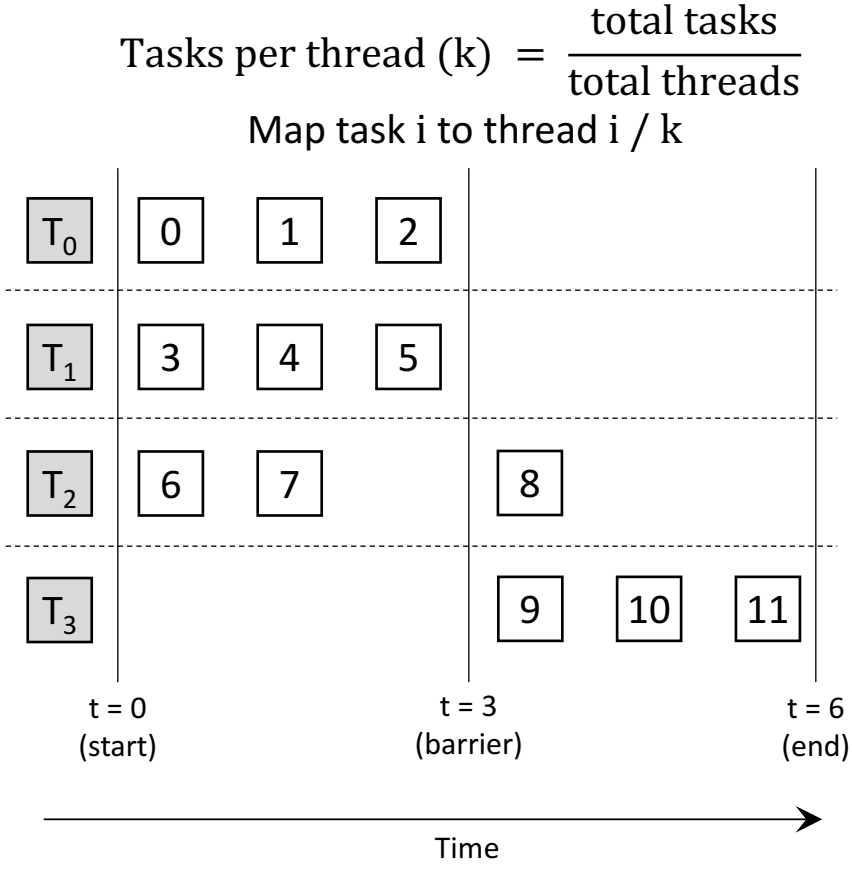
Tasks, Threads, and Mapping

- 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
- Using task interaction graphs
  - Want threads to have minimum interaction with one another
    - Minimum communication

Tasks, Threads, and Mapping

- A good mapping must minimize parallel execution time by
  - Mapping independent tasks to different threads
  - Assigning tasks on critical path to threads ASAP





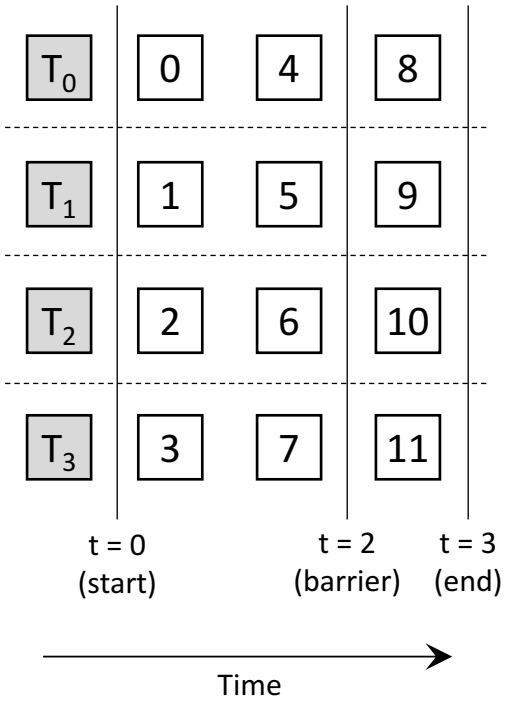
Mapping Techniques

- Map concurrent tasks to threads for execution
- Overheads of mappings
  - Serialization (idling)
  - Communication
- Select mapping to minimize overheads
- Conflicting objectives: minimizing one increases the other
  - Assigning all work to one thread
    - Minimizes communication
    - Significant idling
  - Minimizing serialization introduces communication

Mapping Techniques

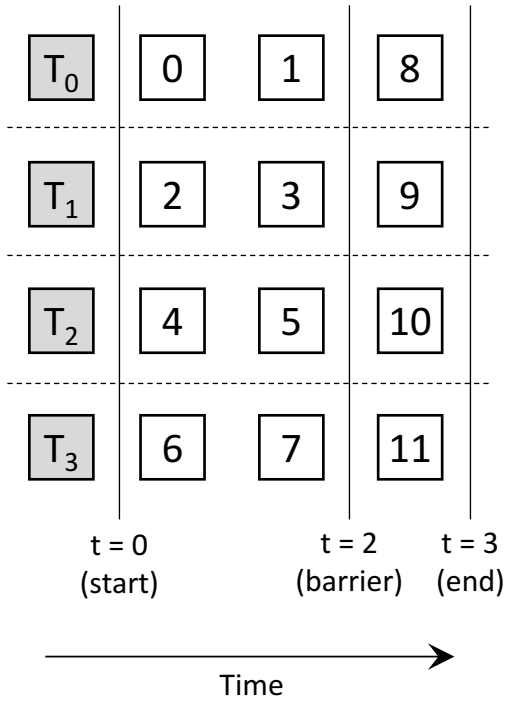
- Task interactions can differ for mapping resulting into same amount of serialization/idling

Map task i to thread i % p



$$\text{Tasks per thread in phase p (kp)} = \frac{\text{total tasks in phase p}}{\text{total threads}}$$

Map task i to thread $\frac{(i - m)}{k_p}$, where m is first task id in phase



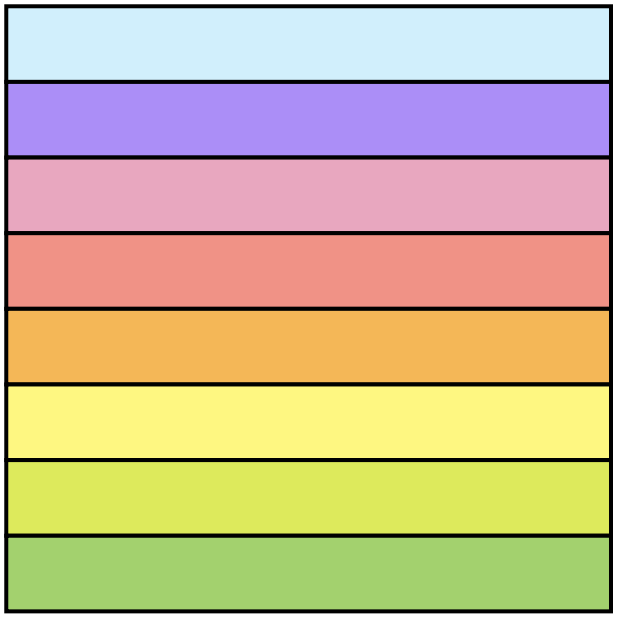
Mapping Techniques for Minimum Idling

- Static mapping
  - A-priori mapping of tasks to threads or processes
  - Requirements
    - A good estimate of task size
    - Even so, computing an optimal mapping may be NP hard
      - E.g., Decomposing evenly is analogous to bin packing
- Dynamic mapping
  - Map tasks to threads or processes at runtime
  - Why?
    - Tasks are generated at runtime, or
    - Their sizes are unknown

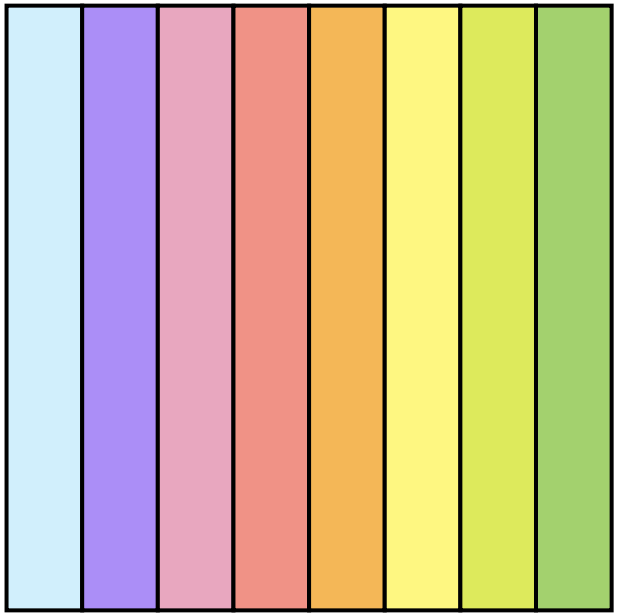
Mappings Based on Data Partitioning

- Partition computation using a combination of
  - data decomposition
  - owner-computes rule
- Example: 1-D block distribution for dense matrices

row-wise distribution

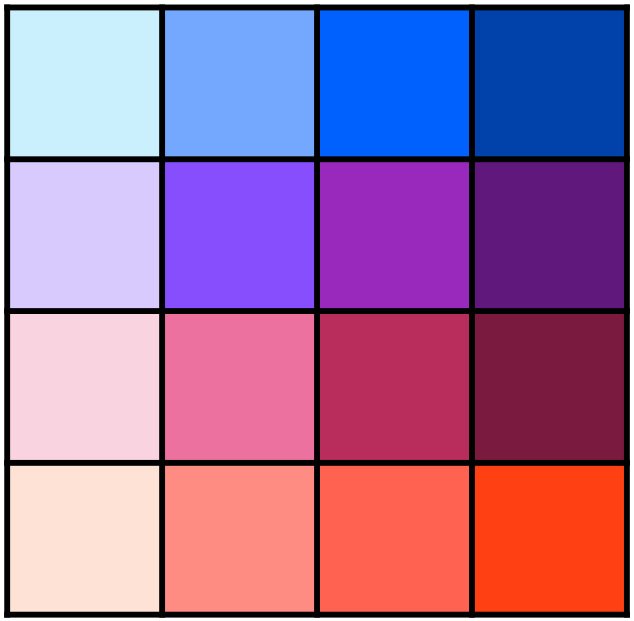


column-wise distribution

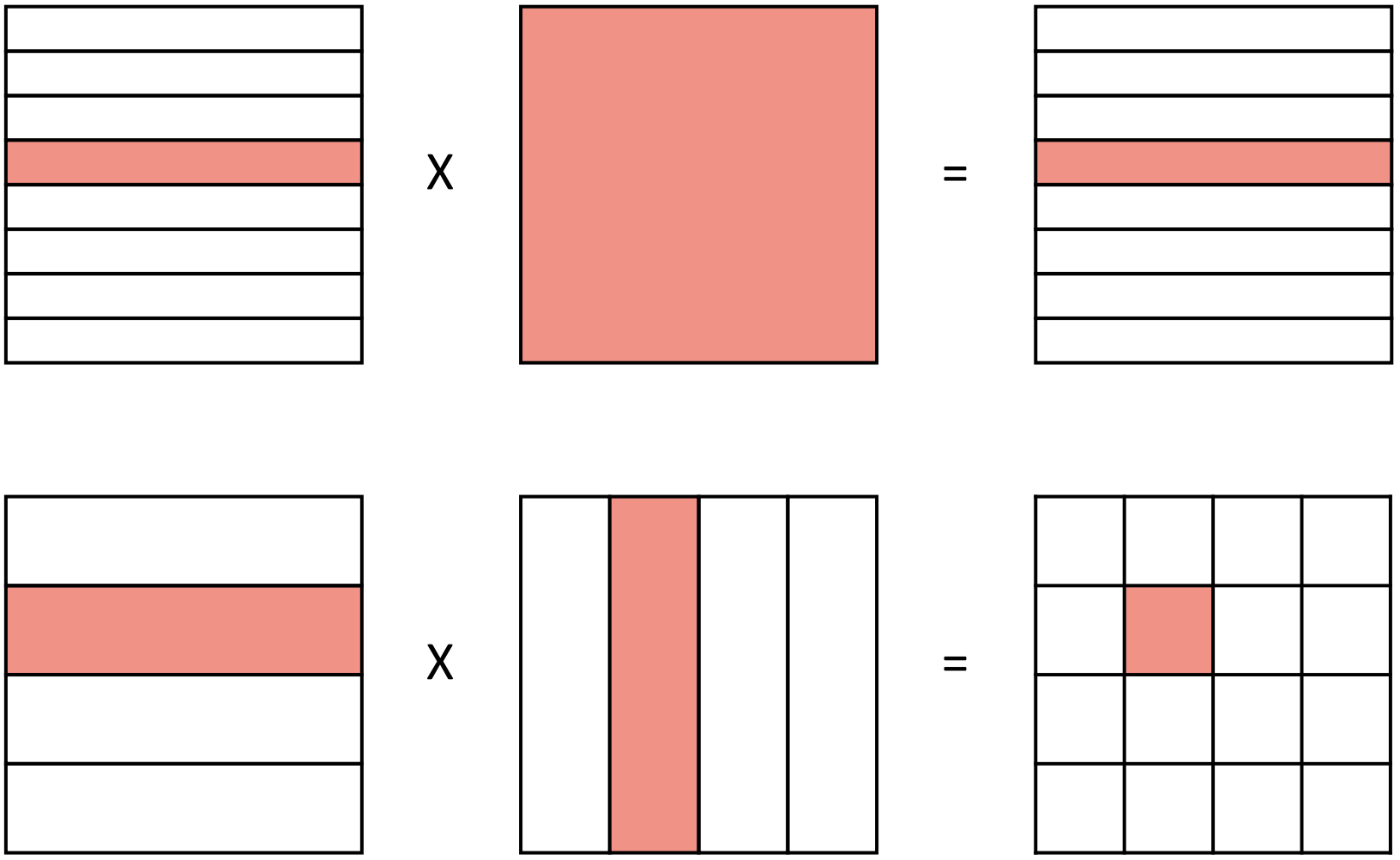


Block Array Distribution Schemes

- Multi-dimensional block distributions
- Enables larger # of threads



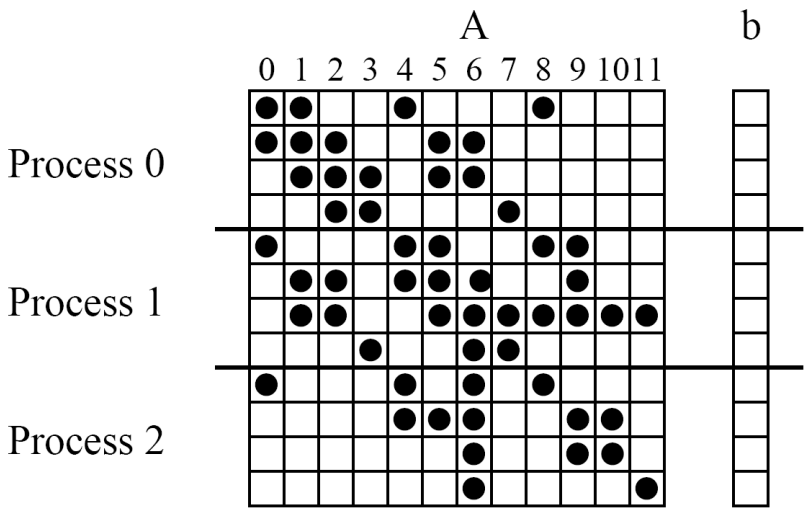
Data Usage in Dense Matrix Multiplication



Decomposition by Graph Partitioning

- Sparse-matrix vector multiplication
- Graph of the matrix is useful for decomposition
  - Work = number of edges
  - Communication for a node = node degree
- Goal: balance work & minimize communication
- Partition the graph
  - Assign (approximately) equal number of edges to each thread
  - Minimize edge cut of the graph partition

Sparse-matrix vector multiplication

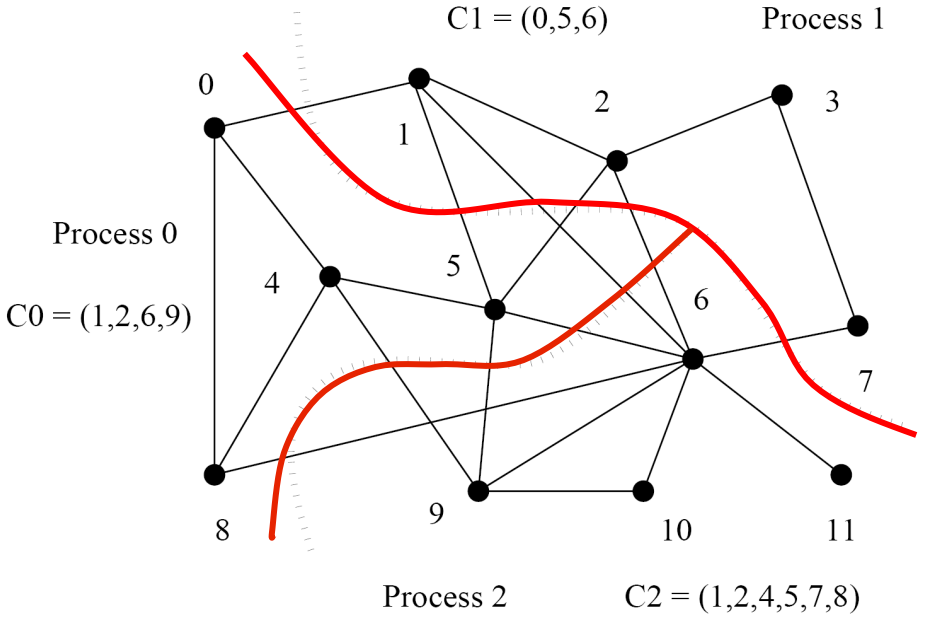


C0 = (4,5,6,7,8)

C1 = (0,1,2,3,8,9,10,11)

C2 = (0,4,5,6)

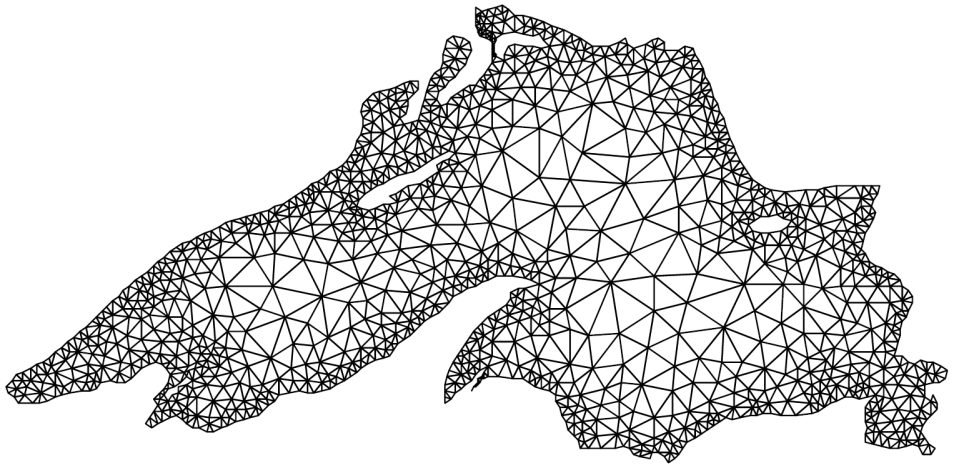
Communication = 17



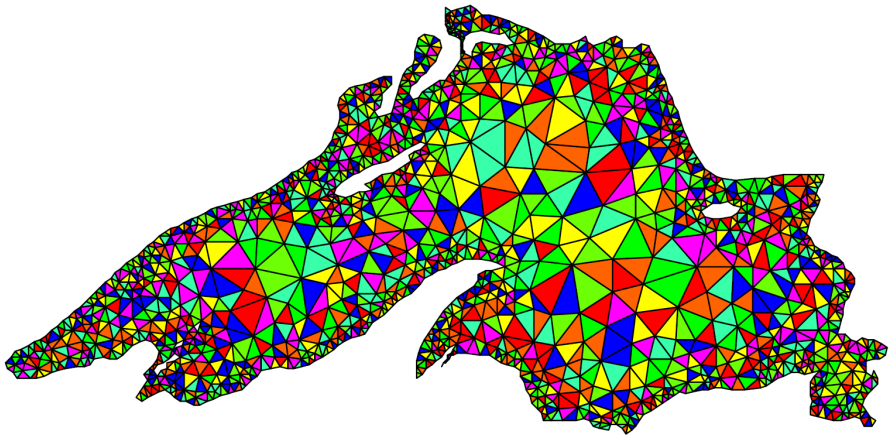
Communication = 13

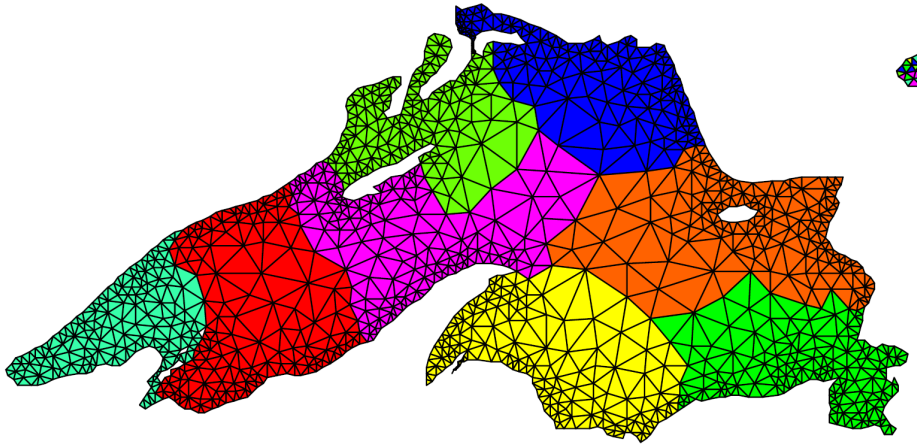
- Optimal graph partitioning is NP-hard
- Rely on partitioning heuristics

Partitioning a Graph of Lake Superior



Random Partitioning





Partitioning for minimum edge-cut

Dynamic Mapping

- Dynamic load balancing can be achieved by dynamic task mapping
- Two ways to achieve dynamic mapping
  - Centralized
  - Distributed

Centralized Dynamic Mapping

- Threads types: Masters or Slaves
- General strategy
  - When a slave runs out of work → request more from master
- Issue
  - Master may become bottleneck for large # of threads
- Solution
  - Chunk scheduling: thread picks up several of tasks at once
  - Chunk-size matters:
    - Large chunk sizes may cause significant load imbalances
    - Potential solution: Gradually decrease chunk size as computation progresses

Distributed Dynamic Mapping

- All threads act as peers
- Each thread can send or receive work from other threads
  - Eliminates centralized bottleneck
- Critical design questions (application-specific):
  - How are sending and receiving threads paired together?
  - Who initiates work transfer?
  - How much work is transferred?
  - When is a transfer triggered?
- Efficient distributed mapping solutions are difficult to design compared to centralized mapping solutions

Minimizing Interaction Overheads

- Maximize data locality
  - Don’t fetch data you already have
  - Restructure computation to reuse data promptly
- Minimize volume of data exchange
  - Partition interaction graph to minimize edge crossings
- Minimize frequency of communication
  - Try to aggregate messages where possible
- Minimize contention and hot-spots
  - Use decentralized techniques

Minimizing Interaction Overheads

- Overlap communication with computation
  - Use non-blocking communication primitives
    - Overlap communication with your own computation
    - One-sided: prefetch remote data to hide latency
  - Multithread code
    - Overlap communication with another thread's computation
- Replicate data (or computation) to reduce communication
- Use group communication primitives instead of point-to-point primitives
- Issue multiple communications and overlap their latency

Reading (for Next Class)

- [AMP] Chapter 9