High Performance Computing
BE 2019 Pattern

Prof V B More

Unit 2: Parallel Algorithm Design

Syllabus:

- **Principles of Parallel Algorithm Design:** Preliminaries, Decomposition Techniques, Characteristics of Tasks and Interactions, Mapping Techniques for Load Balancing, Methods for Containing Interaction Overheads,
- **Parallel Algorithm Models:** Data, Task, Work Pool and Master Slave Model,
- **Complexities:** Sequential and Parallel Computational Complexity, Anomalies in Parallel Algorithms.

Course Object- Outcome Mapping

Course Objectives:

- To analyze the performance and modeling of parallel programs

Course Outcomes:

- CO2: **Design and Develop** an efficient parallel algorithm to solve given problem

Parallel Algorithm

Recipe to solve a problem using multiple processors

Typical steps for constructing a parallel algorithm

— *identify what pieces of work 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*

— *the mapping of work to processors can be done statically by the programmer or dynamically by the runtime*

Topics for Today

Introduction to parallel algorithms

- ***tasks and decomposition***
- ***threads and mapping***
- ***threads versus cores***

Decomposition techniques

- ***recursive decomposition***
- ***data decomposition***
- ***exploratory decomposition***
- ***hybrid decomposition***

Topics for Today

Characteristics of tasks and interactions

- *—task generation, granularity, and context*
- *—characteristics of task interactions*

Mapping techniques for load balancing

- *—static mappings*
- *—dynamic mappings*

Methods for minimizing interaction overheads

Parallel algorithm design templates

Parallel Algorithm Models

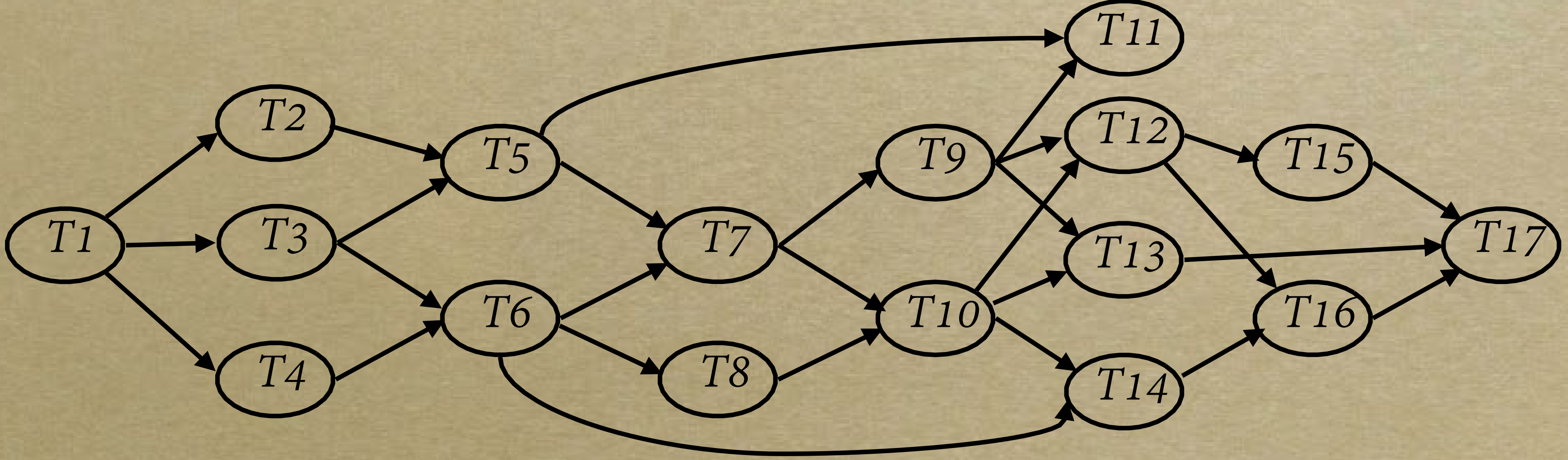
Complexities

Decomposing Work for Parallel Execution

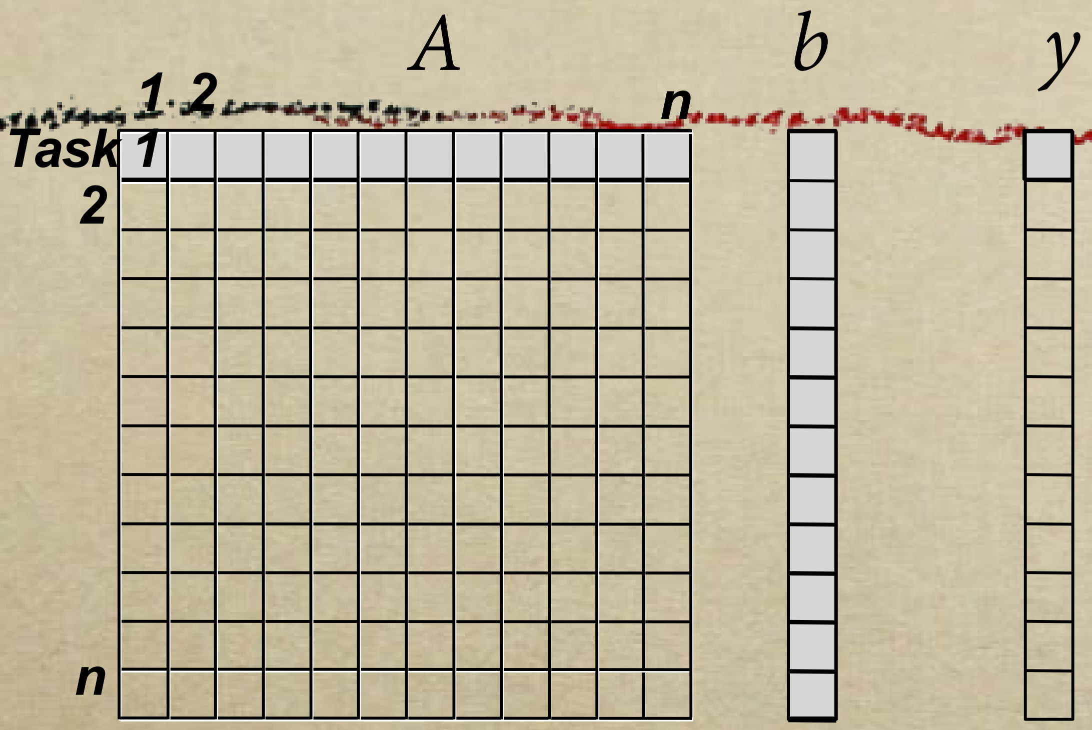
- *Divide work into tasks that can be executed concurrently*
- *Many different decompositions possible for any computation*
- *Tasks may be same, different, or even indeterminate sizes*
- *Tasks may be independent or have non-trivial order*
- *Conceptualize tasks and ordering as task dependency DAG*

node = task

—edge = control dependence



Example: Dense Matrix-Vector Product



- Computing each element of output vector y is independent
- Easy to decompose dense matrix-vector product into tasks

  —one per element in y
- Observations

  —task size is uniform

  —no control dependences between tasks

  —tasks share b

Example: Database Query Processing

Consider the execution of the query:

MODEL = "CIVIC" AND YEAR = 2001 AND

(COLOR = "GREEN" OR COLOR = "WHITE")

on the following database:

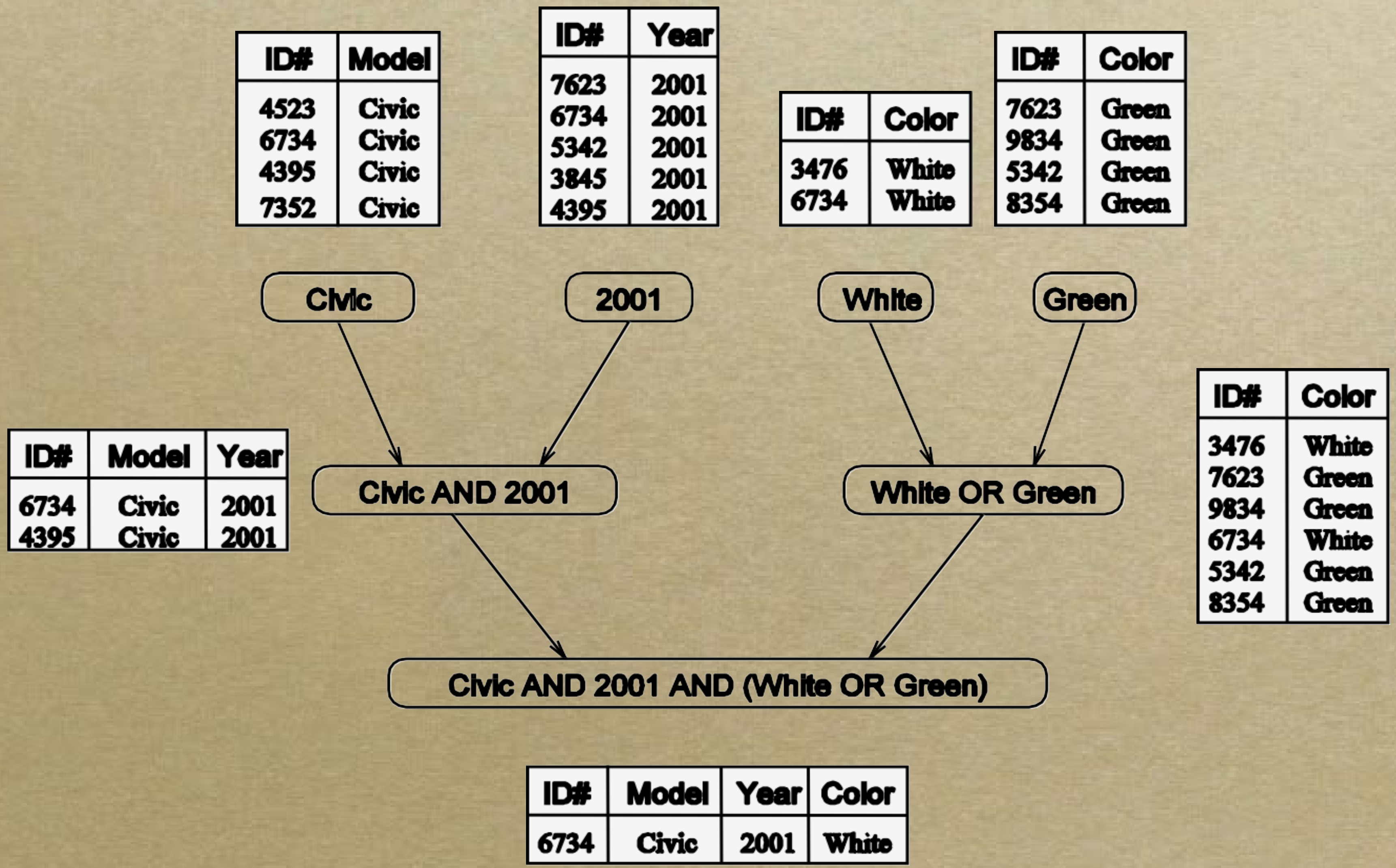
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Example: Database Query Processing

- *Task: compute set of elements that satisfy a predicate*
  - *task result = table of entries that satisfy the predicate*
- *Edge: output of one task serves as input to the next*

MODEL = "CIVIC" AND YEAR = 2001 AND

(COLOR = "GREEN" OR COLOR = "WHITE")



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

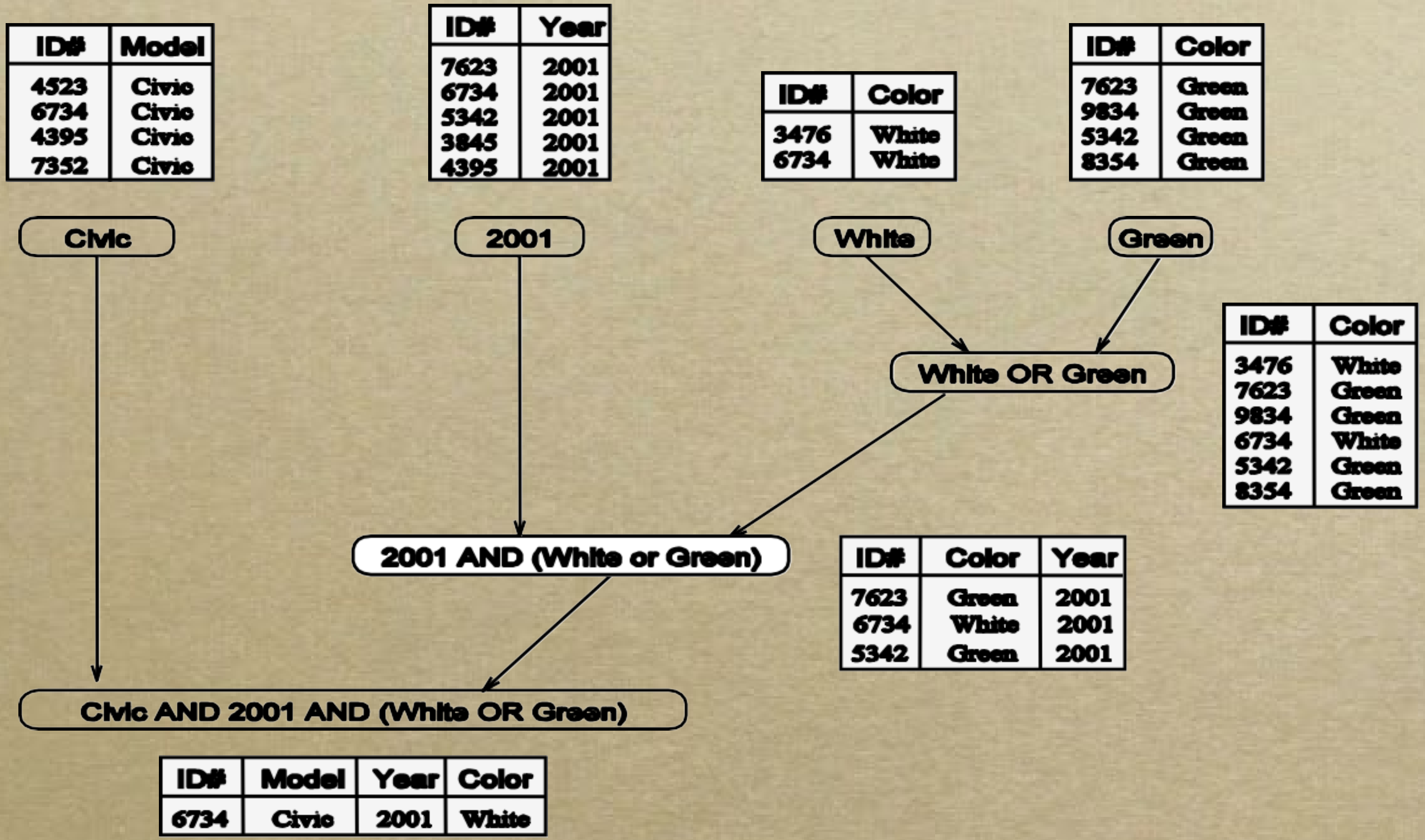
ID#	Model	Year	Color
6734	Civic	2001	White

Example: Database Query Processing

- *Alternate task decomposition for query*

MODEL = "CIVIC" AND YEAR = 2001 AND

(COLOR = "GREEN" OR COLOR = "WHITE")


Civic
2001
White
Green
White OR Green
2001 AND (White or Green)
Civic AND 2001 AND (White OR Green)

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

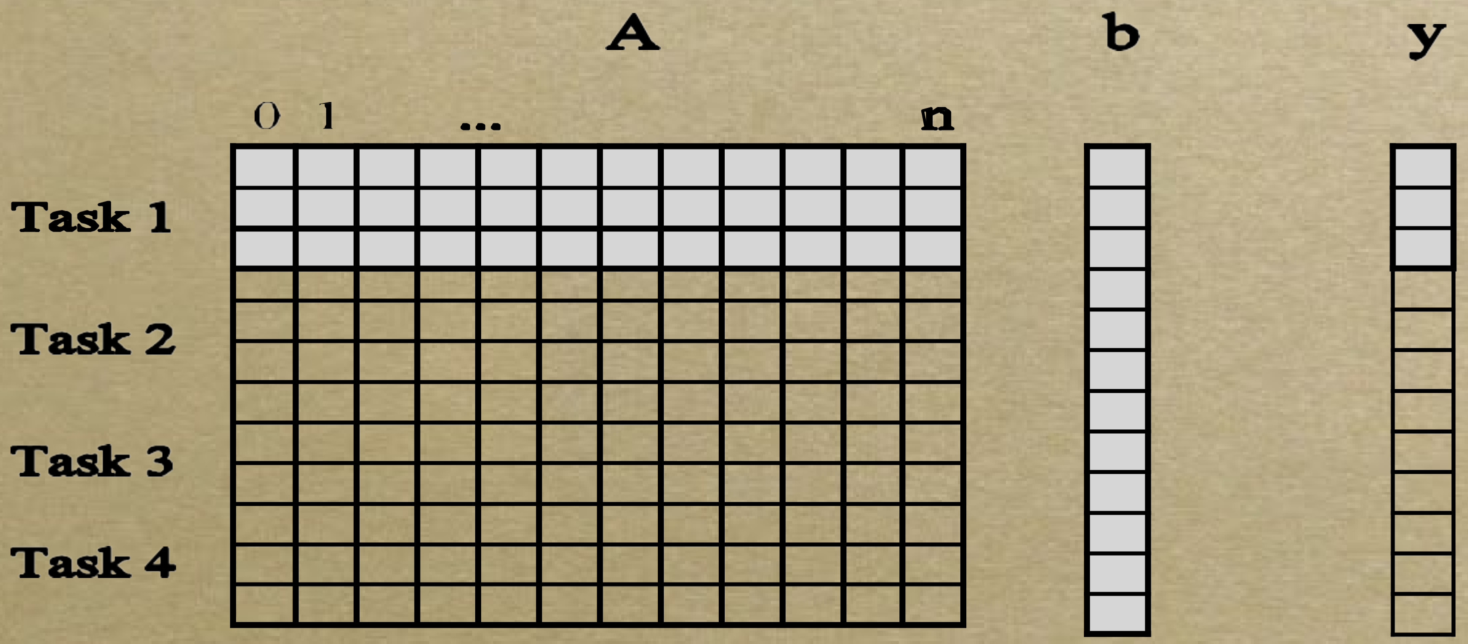
ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

ID#	Model	Year	Color
6734	Civic	2001	White

Lesson: Different decompositions may yield different parallelism and different amounts of work

Granularity of Task Decompositions

- ***Granularity = task size***
  - *—depends on the number of tasks*
- ***Fine-grain = large number of tasks***
- ***Coarse-grain = small number of tasks***
- ***Granularity examples for dense matrix-vector multiply***
  - *—fine-grain: each task represents an individual element in y*
  - *—coarser-grain: each task computes 3 elements in y*



Degree of Concurrency

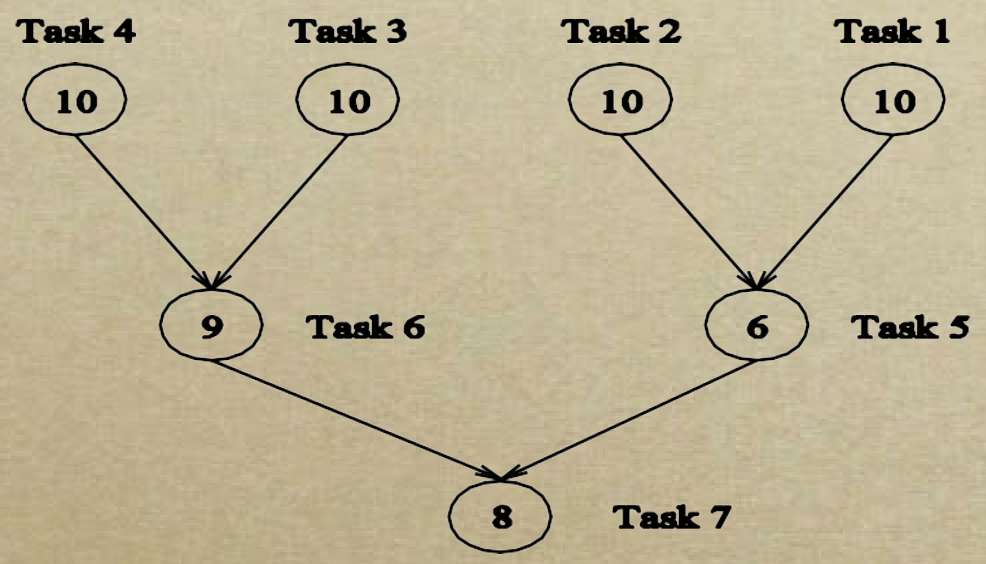
- ***Definition: number of tasks that can execute in parallel***
- ***May change during program execution***
- ***Metrics***
  - ***—maximum degree of concurrency***
    - *largest # concurrent tasks at any point in the execution*
  - ***—average degree of concurrency***
    - *average number of tasks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

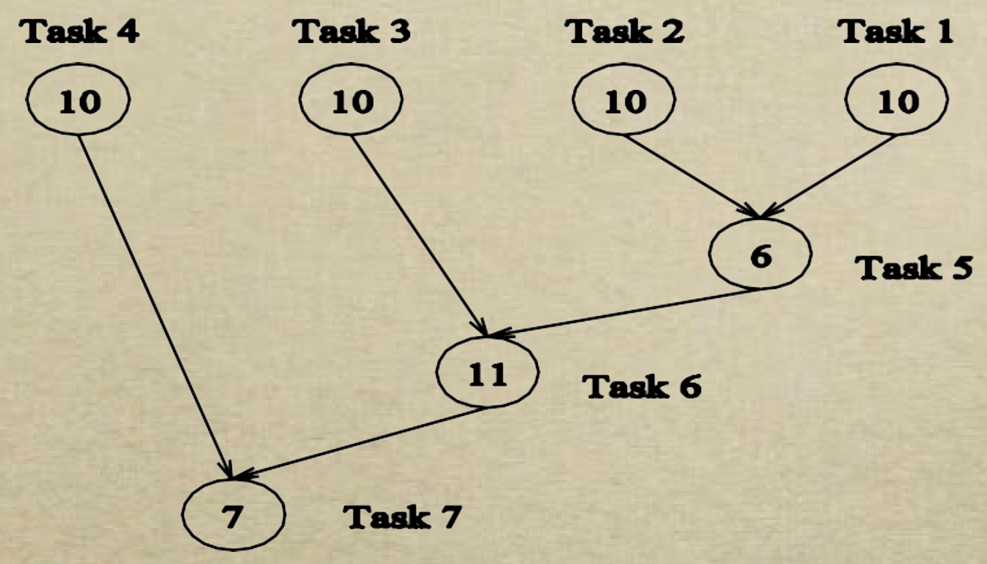
Critical Path

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

Critical Path Length

Examples: database query task dependency graphs





Note: number in vertex represents task cost

What tasks are on the critical path for each dependency graph?

What is the shortest parallel execution time for each decomposition?

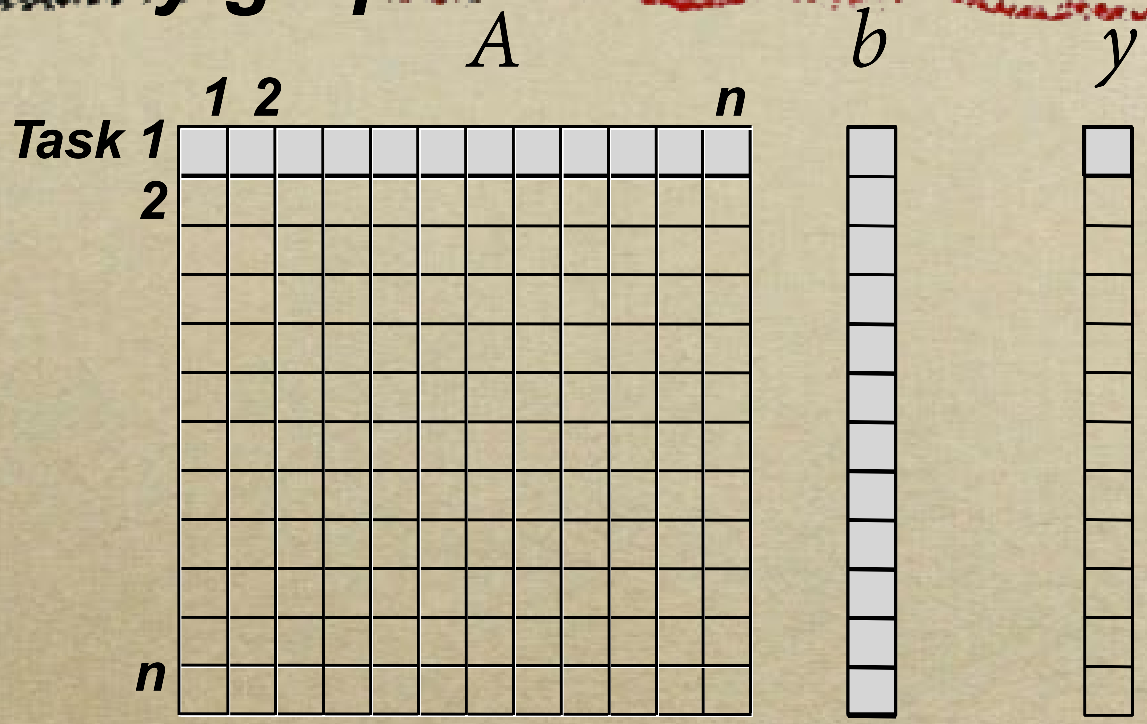
How many processors are needed to achieve the minimum time?

What is the maximum degree of concurrency?

What is the average parallelism?

Critical Path Length

Example: dependency graph for dense-matrix vector product



Questions:

What is the maximum number of tasks possible?

What does a task dependency graph look like for this case?

What is the shortest parallel execution time for the graph?

How many processors are needed to achieve the minimum time?

What is the maximum degree of concurrency?

What is the average parallelism?

Limits on Parallel Performance

- **_What bounds parallel execution time?_**
  - ***minimum task granularity***
    - ***e.g. dense matrix-vector multiplication ≤ n^2 concurrent tasks***
  - ***dependencies between tasks***
  - ***parallelization overheads***
    - ***e.g., cost of communication between tasks***
  - ***fraction of application work that can't be parallelized***
    - ***Amdahl's law***
- **_Measures of parallel performance_**
  - ***speedup = T_1/T_p***
  - ***parallel efficiency = $T_1/(pT_p)$***

Amdahl's Law

A hard limit on the speedup that can be obtained using multiple CPUs

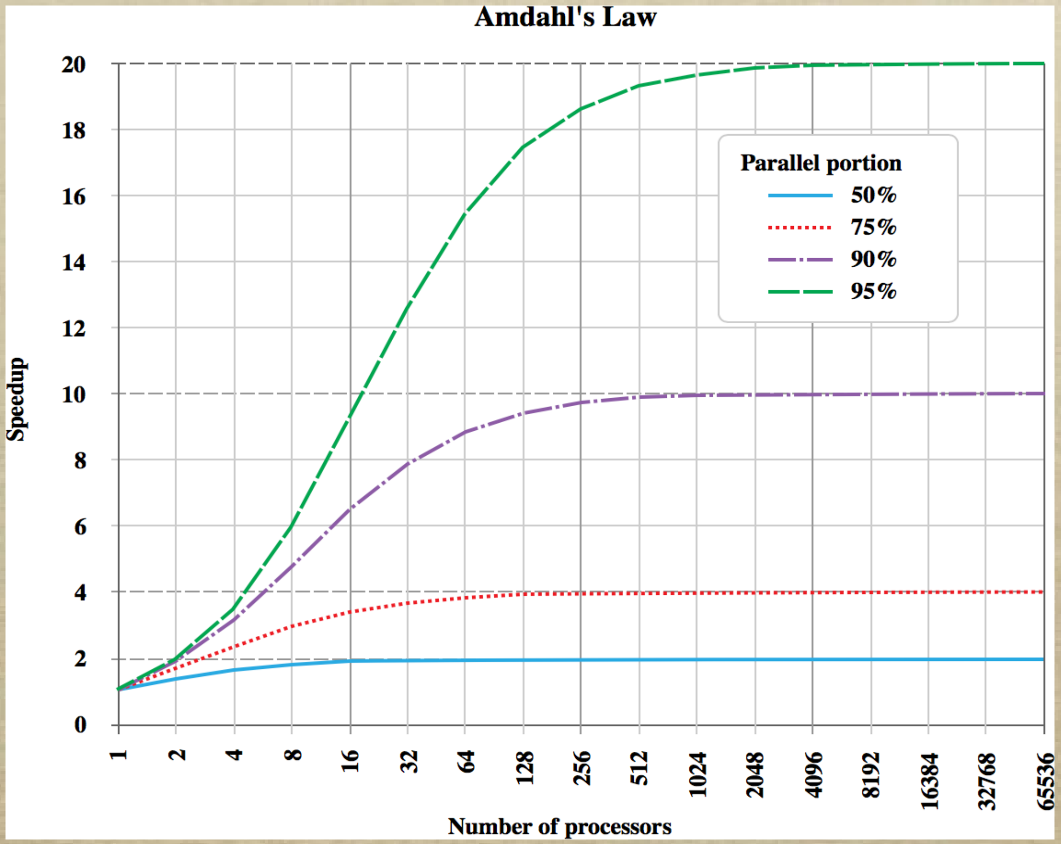
Two expressions of Amdahl's law

—execution time on N CPUs

$$t_N = (f_p/N + f_s)t_1$$

—speedup on N processors

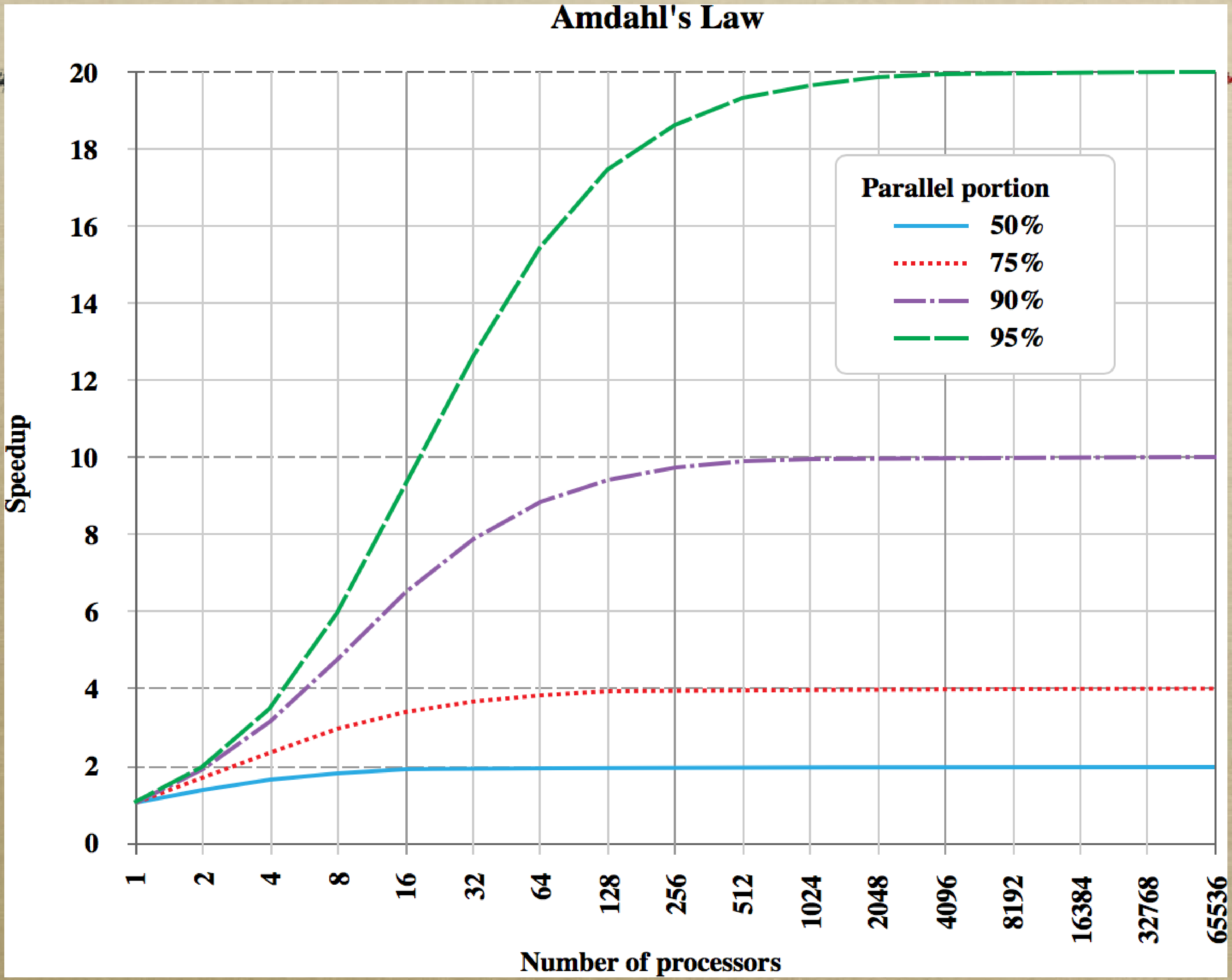
$$S = 1/(f_s + f_p/N)$$



https://en.wikipedia.org/wiki/Amdahl's_law

$$t_1 : time\ on\ 1\ CPU$$
$$t_N : time\ on\ N\ CPUs$$
$$f_p : parallel\ fraction$$
$$f_s : serial\ fraction = 1 - f_p$$

Amdahl's Law



https://en.wikipedia.org/wiki/Amdahl's_law

Task Interaction Graphs

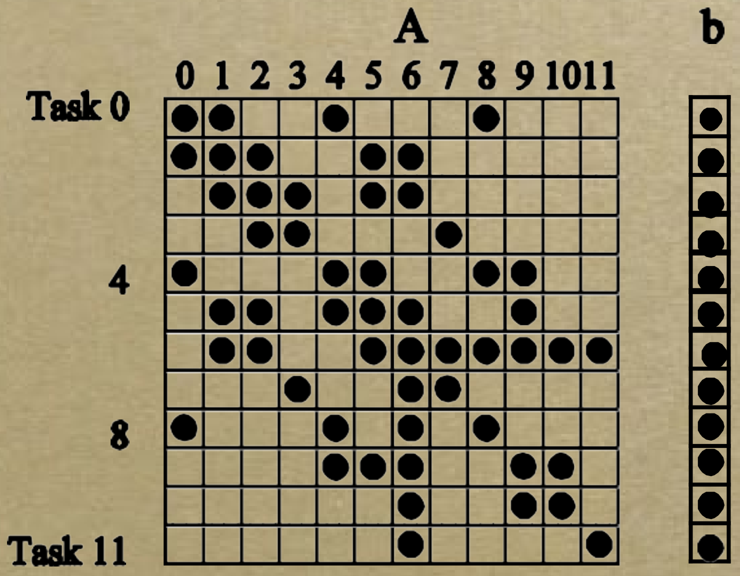
- ***Tasks generally exchange data with others***
  - ***example: dense matrix-vector multiply***
    - ***if vector b is not replicated in all tasks, tasks will have to communicate elements of b***
- ***Task interaction graph***
  - ***node = task***
  - ***edge = interaction or data exchange***
- ***Task interaction graphs vs. task dependency graphs***
  - ***task interaction graphs represent <u>data dependences</u>***
  - ***task dependency graphs represent <u>control dependences</u>***

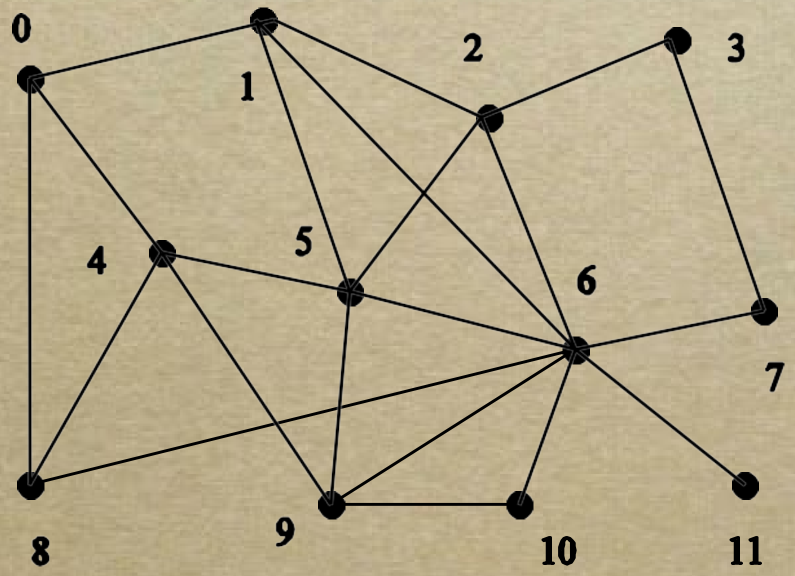
Task Interaction Graph Example

Sparse matrix-vector multiplication

- ***Computation of each result element = independent task***
- ***Only non-zero elements of sparse matrix*** *A* ***participate***
- ***If, b is partitioned among tasks …***
  - — ***structure of the task interaction graph = graph of the matrix*** *A*

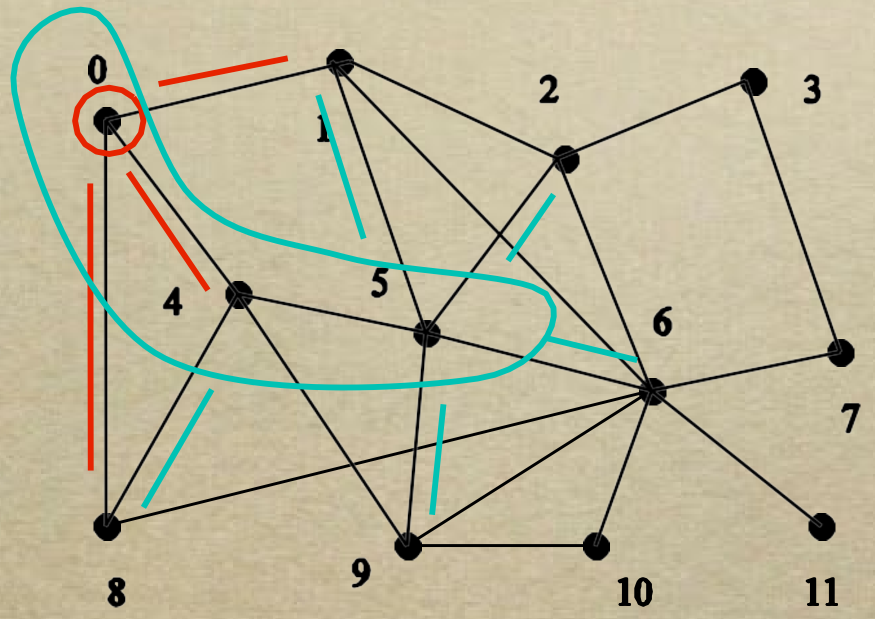
(i.e. the graph for which *A* ***represents the adjacency structure)***





Interaction Graphs, Granularity, & Communication

- ***Finer task granularity increases communication overhead***
- ***Example: sparse matrix-vector product interaction graph***



- ***Assumptions:***
  - — ***each node 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, and 5 are a task: communication = 5; computation = 15***
  - — ***coarser-grain decomposition smaller communication/computation***

Tasks, Threads, and Mapping

Generally

- —*# of tasks > # threads available*
- —*parallel algorithm must map tasks to threads*

Why threads rather than CPU cores?

- —*aggregate tasks into threads*
  - *thread = processing or computing agent that performs work*
  - *assign collection of tasks and associated data to a thread*
- —*operating system maps threads to physical cores*
  - *operating systems often enable one to bind a thread to a core*
  - *for multithreaded cores, the OS can bind multiple software threads to distinct hardware threads associated with a core*

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

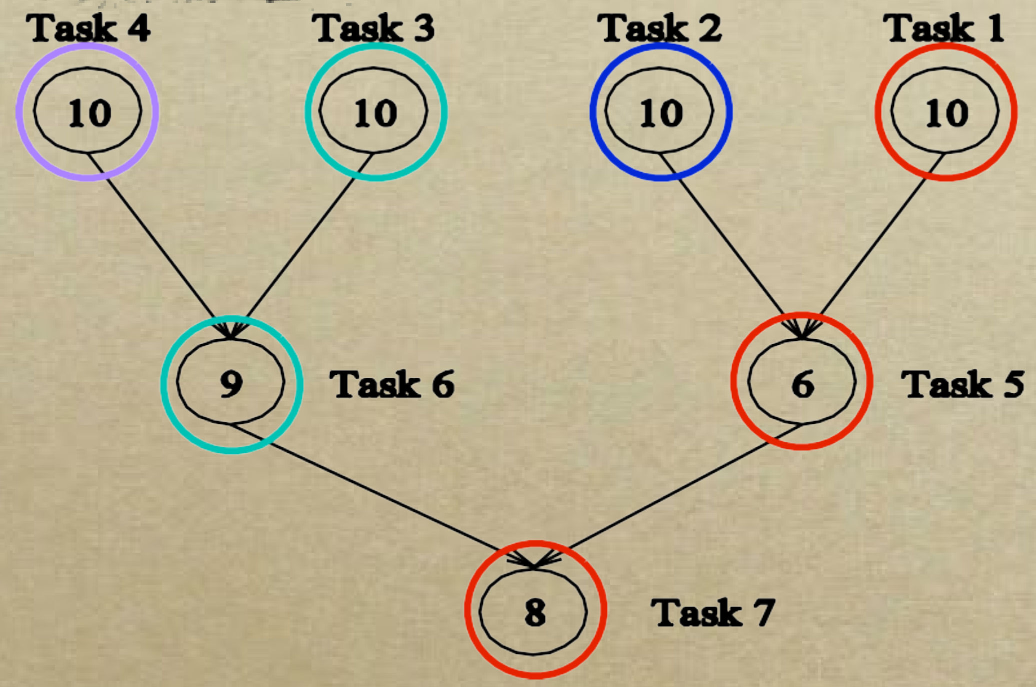
Minimizing interactions between threads

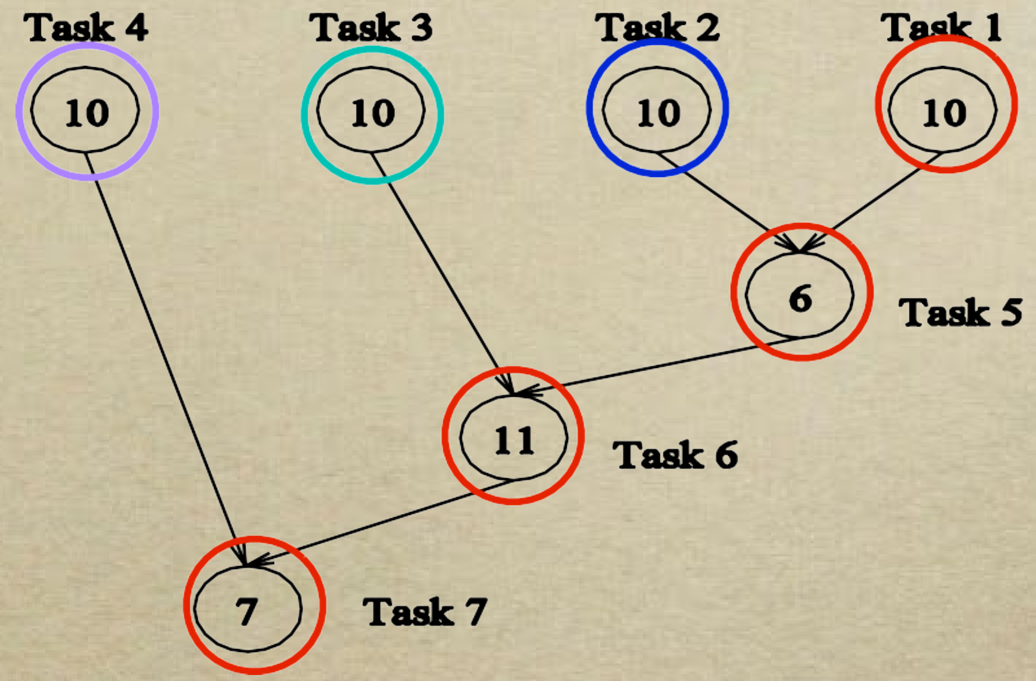
—map tasks with dense interactions to the same thread

Difficulty: criteria often conflict with one another

—e.g. no decomposition minimizes interactions but no speedup!

Tasks, Threads, and Mapping Example





Example: mapping database queries to threads

- ***Consider the dependency graphs in levels***
  - ***no nodes in a level depend upon one another***
  - ***compute levels using topological sort***
- ***Assign all tasks within a level to different threads***

Decomposition Techniques

How should one decompose a task into various subtasks?

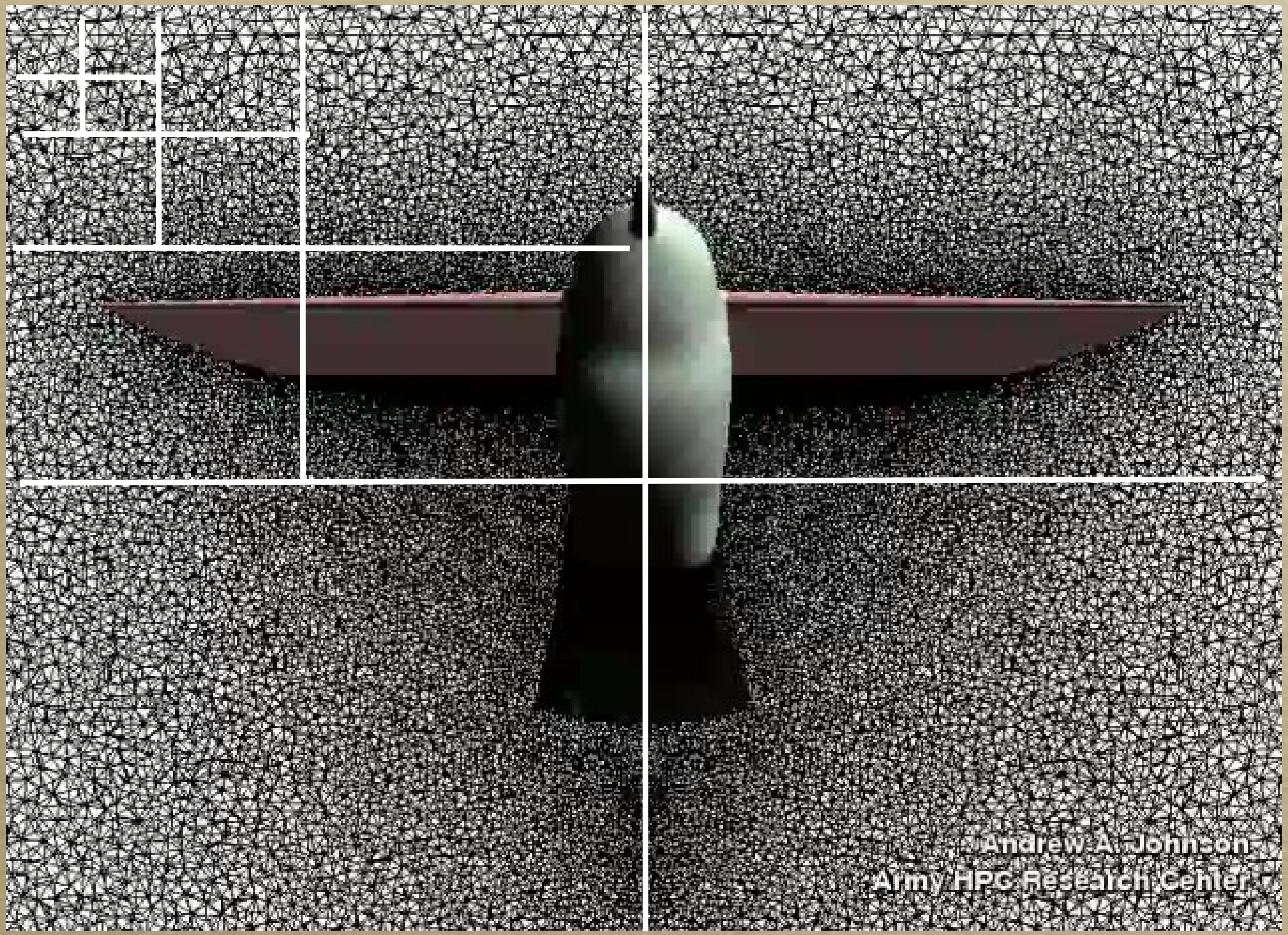
- ***No single universal recipe***
- ***In practice, a variety of techniques are used including***
  - ***recursive decomposition***
  - ***data decomposition***
  - ***exploratory decomposition***
  - ***speculative decomposition***

Recursive Decomposition

Suitable for problems solvable using divide-and-conquer

Steps

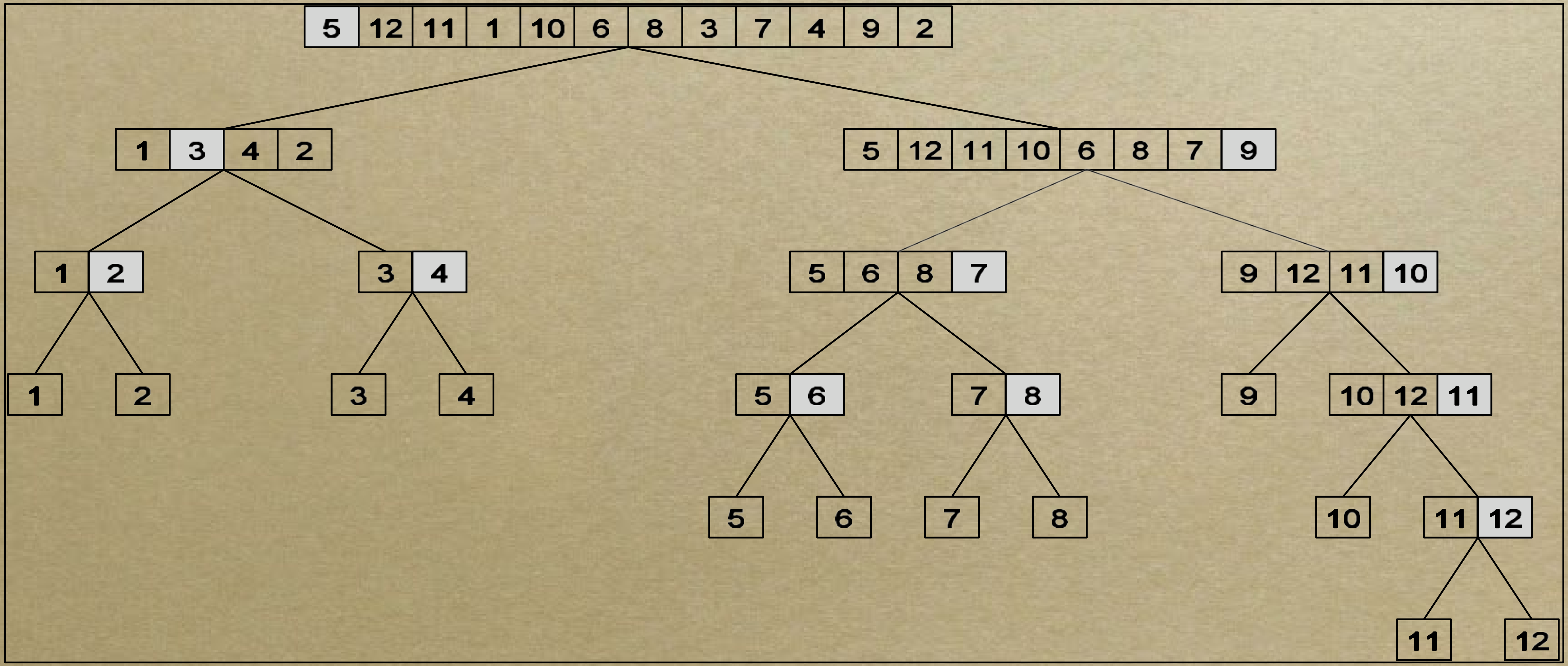
1. *decompose a problem into a set of sub-problems*
2. *recursively decompose each sub-problem*
3. *stop decomposition when minimum desired granularity reached*



Recursive Decomposition for Quicksort

Sort a vector v:

1. ***Select a pivot***
2. ***Partition v around pivot into vleft and vright***
3. ***In parallel, sort vleft and sort vright***

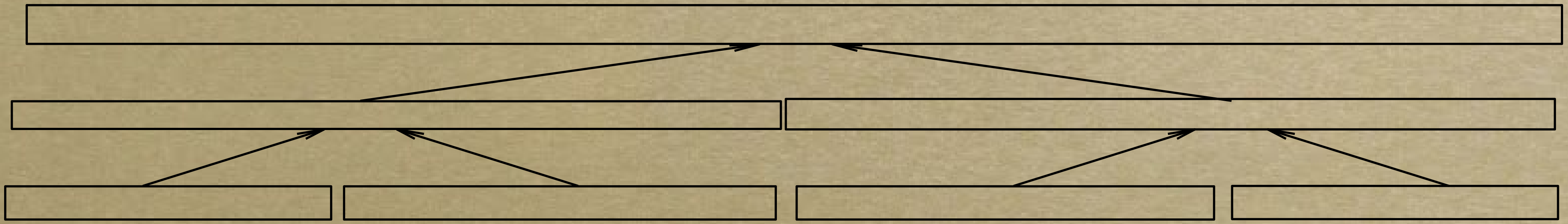


Recursive Decomposition for Min

Finding the minimum in a vector using divide-and-conquer

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

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



Applicable to other associative operations, e.g. sum, AND …

Data Decomposition

Steps

1. *identify the data on which computations are performed*
2. *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 based on

— *input data*

— *output data*

— *input + output data*

— *intermediate data*

Decomposition Based on Input Data

Applicable if each output is computed as a function of the input

May be the only natural decomposition if output is unknown

—*examples*

- *finding the minimum in a set or other reductions*
- *sorting a vector*

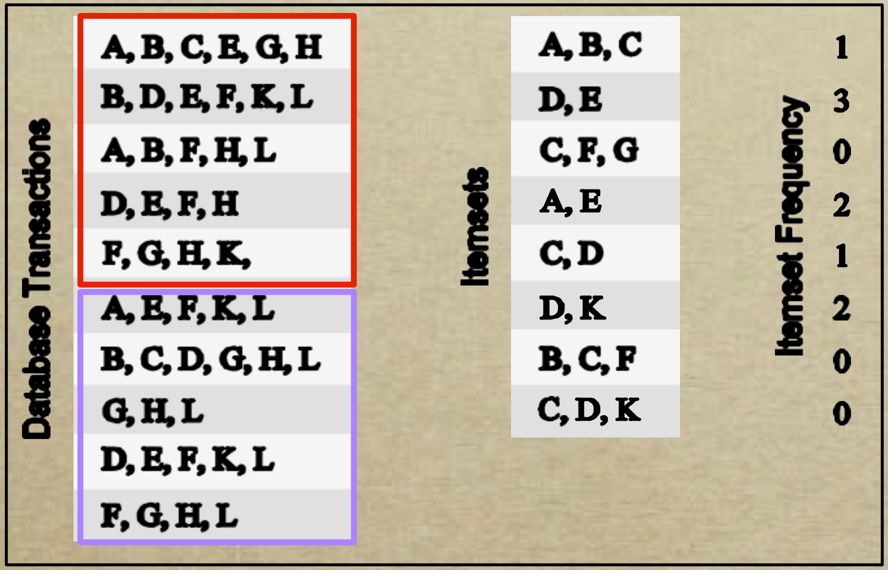
Associate a task with each input data partition

—*task performs computation on its part of the data*

—*subsequent processing combines partial results from earlier tasks*

Example: Decomposition Based on Input Data

Count the frequency of item sets in database transactions

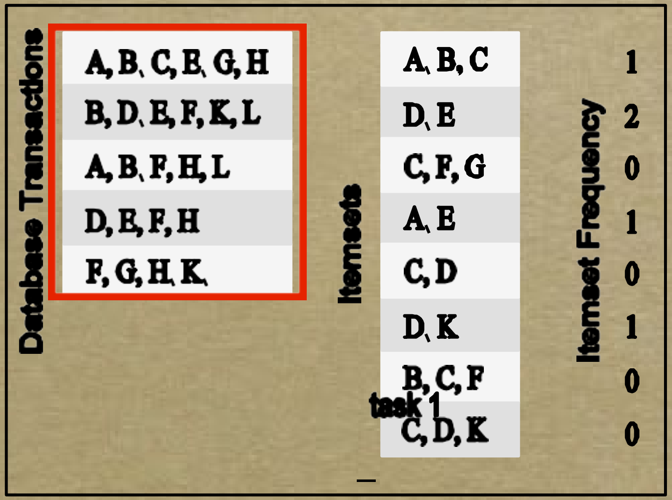


Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Itemsets	Itemset Frequency
A, B, C	1
D, E	3
C, F, G	0
A, E	2
C, D	1
D, K	2
B, C, F	0
C, D, K	0

- ***Partition computation by partitioning the set of transactions***

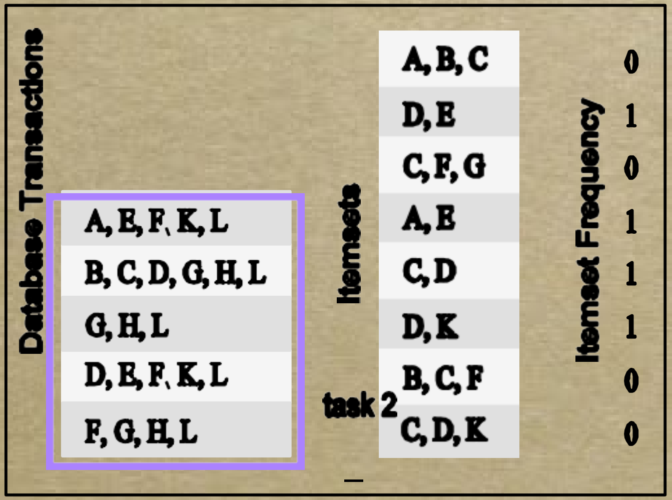
— ***a task computes a local count for each item set for its transactions***



task 1

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,

Itemsets	Itemset Frequency
A, B, C	1
D, E	2
C, F, G	0
A, E	1
C, D	0
D, K	1
B, C, F	0
C, D, K	0



task 2

Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

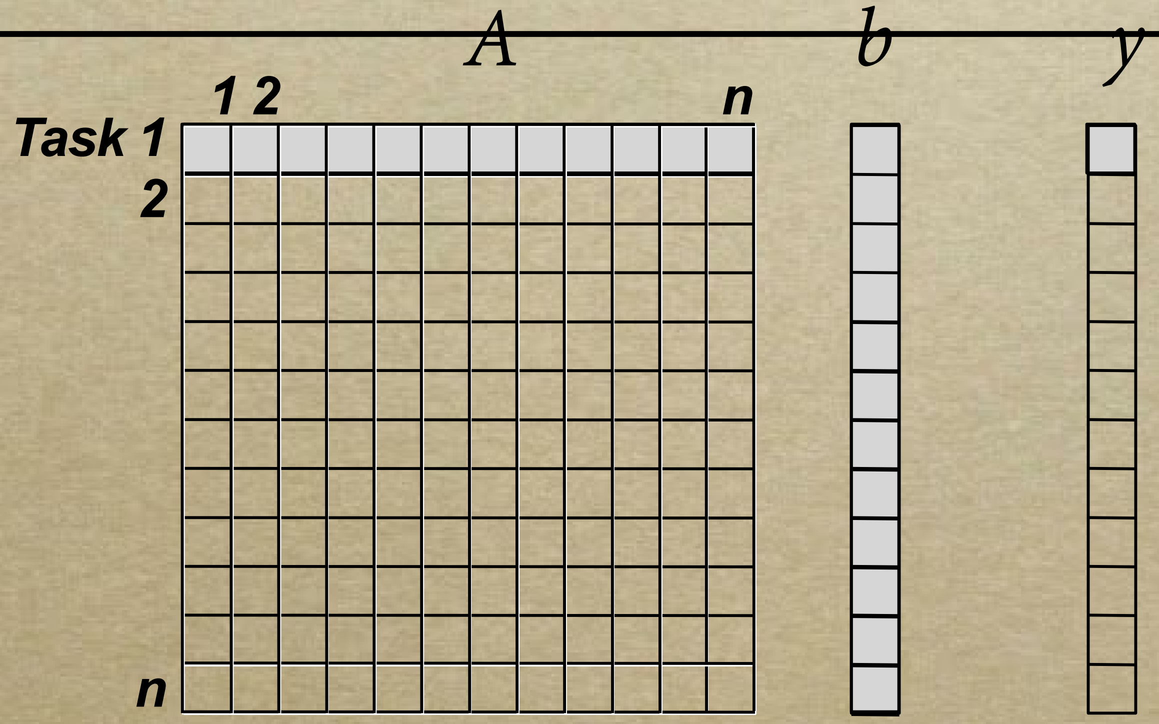
Itemsets	Itemset Frequency
A, B, C	0
D, E	1
C, F, G	0
A, E	1
C, D	1
D, K	1
B, C, F	0
C, D, K	0

— ***sum local count vectors for item sets to produce total count vector***

Decomposition Based on Output Data

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

Example:
dense matrix-vector multiply



Output Data Decomposition: Example

- ***Matrix multiplication: C = A x B***
- ***Computation of C can be partitioned into four tasks***

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

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

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

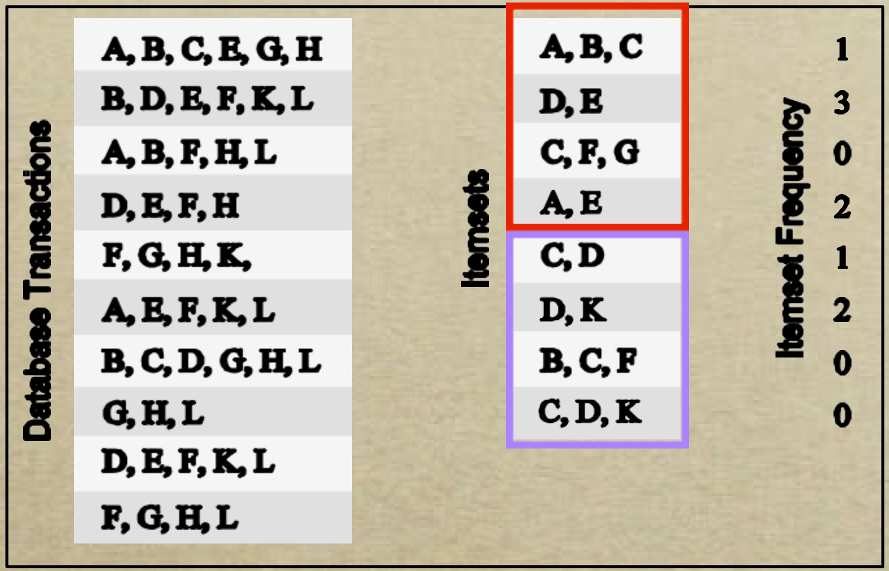
Task 3:
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

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

Other task decompositions possible

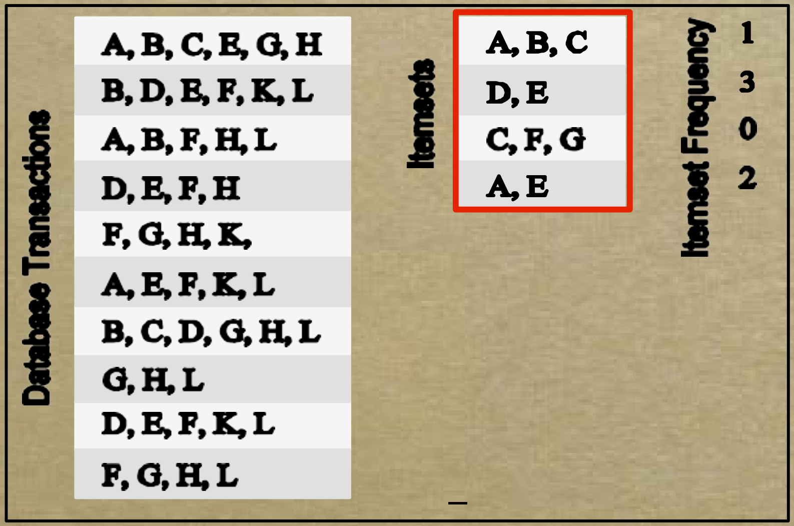
Example: Decomposition Based on Output Data

Count the frequency of item sets in database transactions



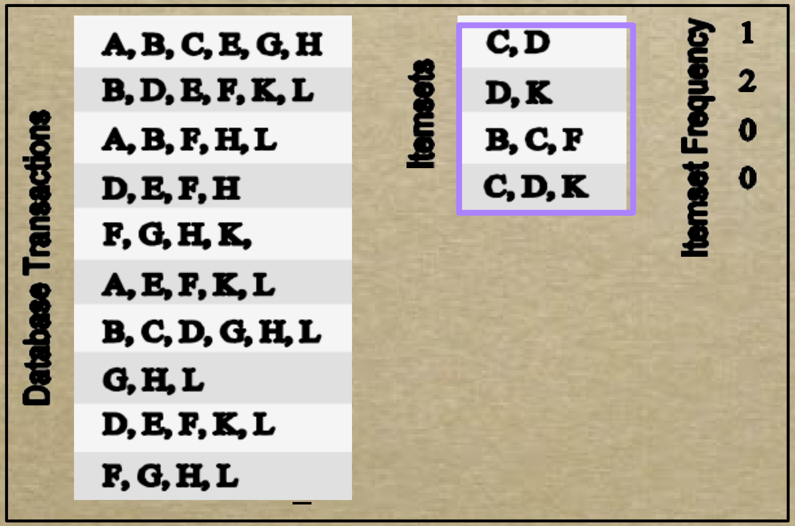
Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

- ***Partition computation by partitioning the item sets to count***
  - — ***each task computes total count for each of its item sets***



Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 1



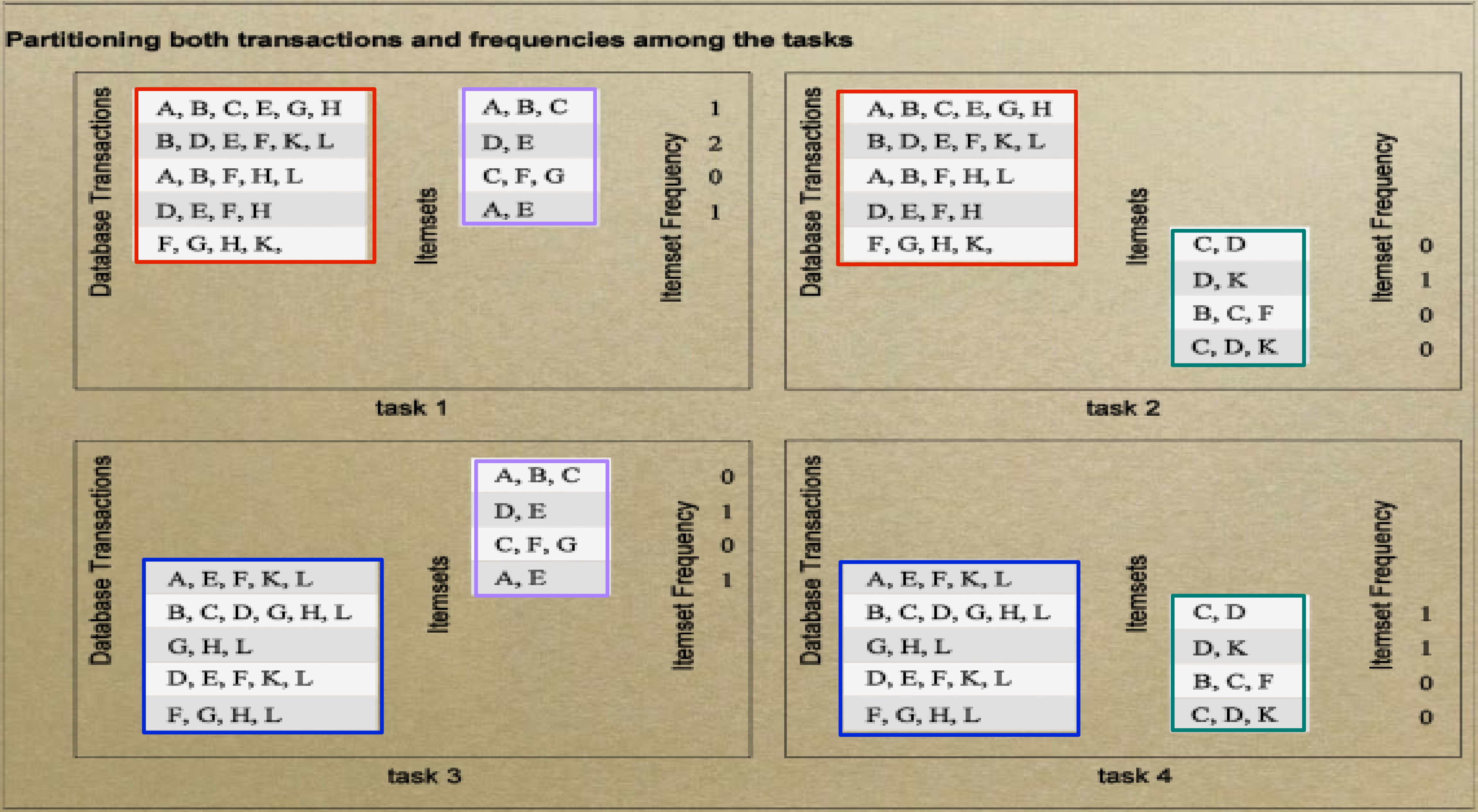
Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	C, D	1
B, D, E, F, K, L	D, K	2
A, B, F, H, L	B, C, F	0
D, E, F, H	C, D, K	0
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 2

  - — ***append total counts for item subsets to produce result***

Partitioning Input *and* Output Data

- ***Partition on both input and output for more concurrency***
  - ***Example: item set counting***



Intermediate Data Partitioning

- ***If computation is a sequence of transforms***
  - ***(from input data to output data)***
- ***Can decompose based on data for intermediate stages***

Example: Intermediate Data Partitioning

***Dense Matrix Multiply* Decomposition of intermediate data: yields 8 + 4 tasks**

Stage I

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

Stage II

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

Task 01: $D_{1,1,1} = A_{1,1} B_{1,1}$ ***Task 02:*** $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1} B_{1,2}$ ***Task 04:*** $D_{2,1,2} = A_{1,2} B_{2,2}$

Task 05: $D_{1,2,1} = A_{2,1} B_{1,1}$ ***Task 06:*** $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1} B_{1,2}$ ***Task 08:*** $D_{2,2,2} = A_{2,2} B_{2,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$ ***Task 10:*** $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

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

Intermediate Data Partitioning: Example

Tasks: dense matrix multiply decomposition of intermediate data

Task 01: $D_{1,1,1} = A_{1,1} B_{1,1}$ **Task 02:** $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1} B_{1,2}$ **Task 04:** $D_{2,1,2} = A_{1,2} B_{2,2}$

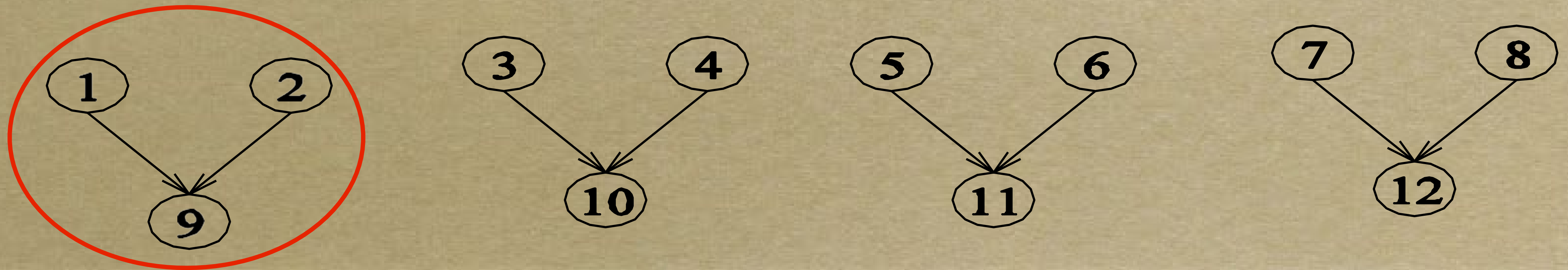
Task 05: $D_{1,2,1} = A_{2,1} B_{1,1}$ **Task 06:** $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1} B_{1,2}$ **Task 08:** $D_{2,2,2} = A_{2,2} B_{2,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$ **Task 10:** $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

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

Task dependency graph



Owner Computes Rule

- Each datum is assigned to a thread
- Each thread computes values associated with its data

Implications

- —input data decomposition
  - all computations using an input datum are performed by its thread
- —output data decomposition
  - an output is computed by the thread assigned to the output data

Exploratory Decomposition

Exploration (search) of a state space of solutions

- ***problem decomposition reflects shape of execution***

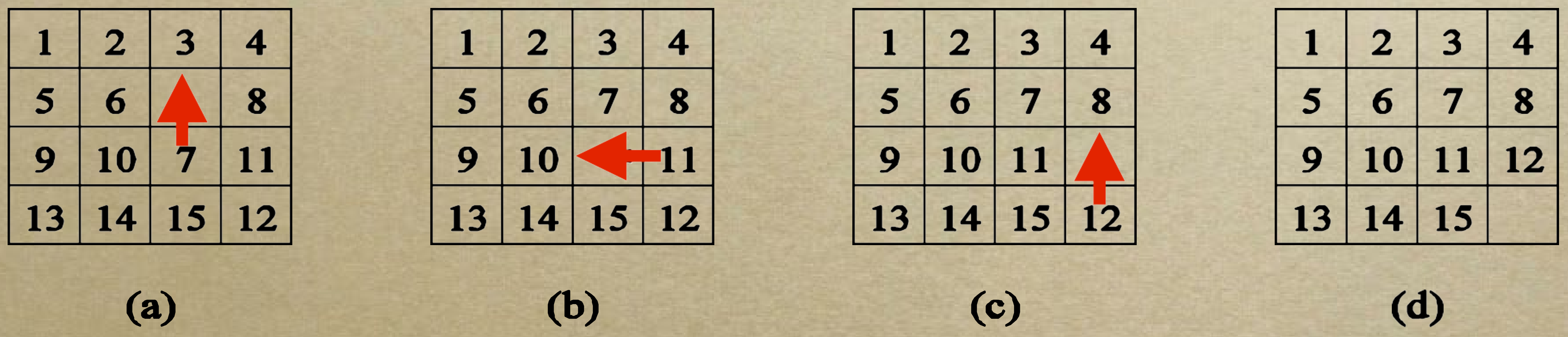
Examples

- ***discrete optimization***
  - ***0/1 integer programming***
- ***theorem proving***
- ***game playing***

Exploratory Decomposition Example

Solving a 15 puzzle

- ***Sequence of three moves from state (a) to final state (d)***



- ***From an arbitrary state, must search for a solution***

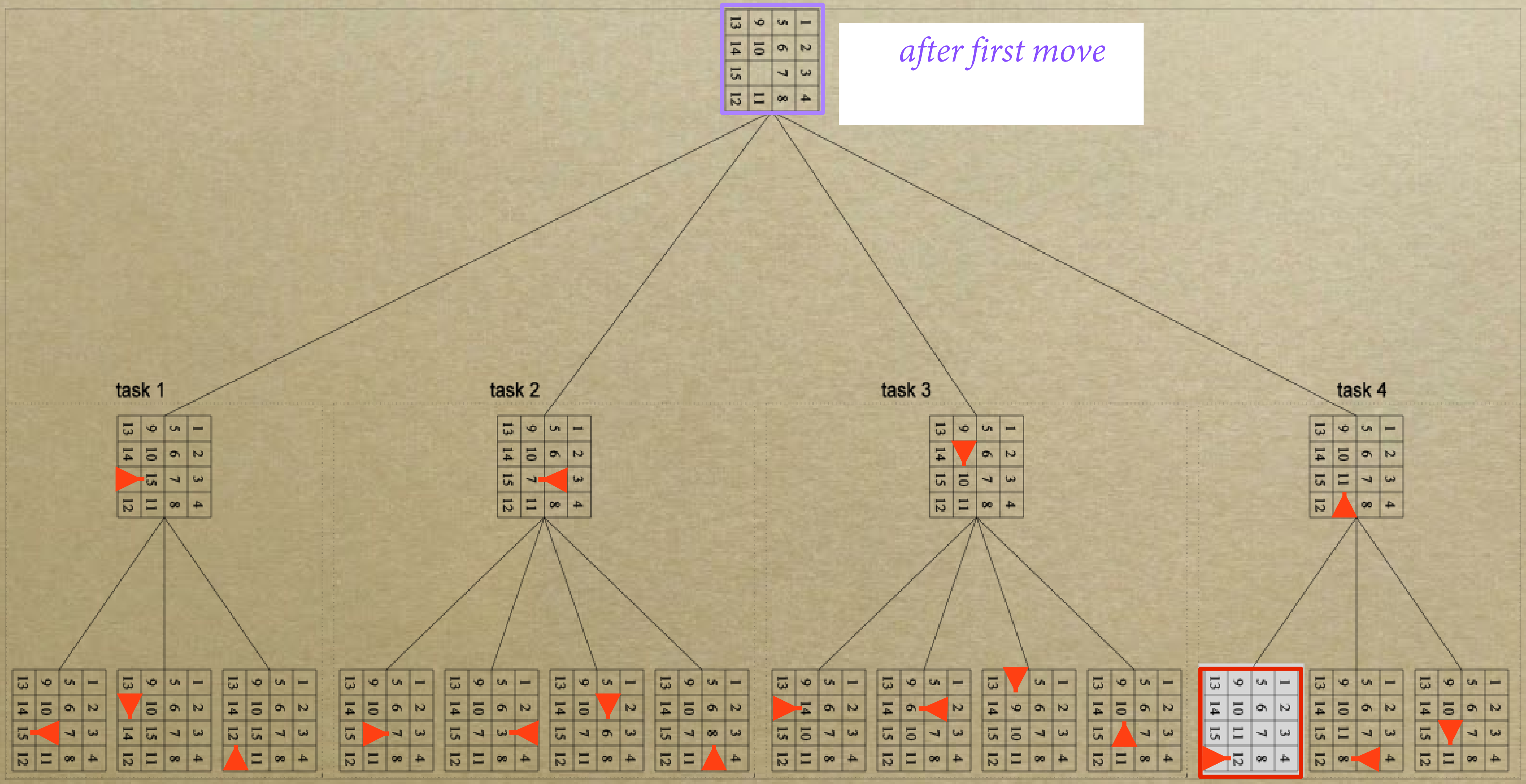
Exploratory Decomposition: Example

Solving a 15 puzzle

Search

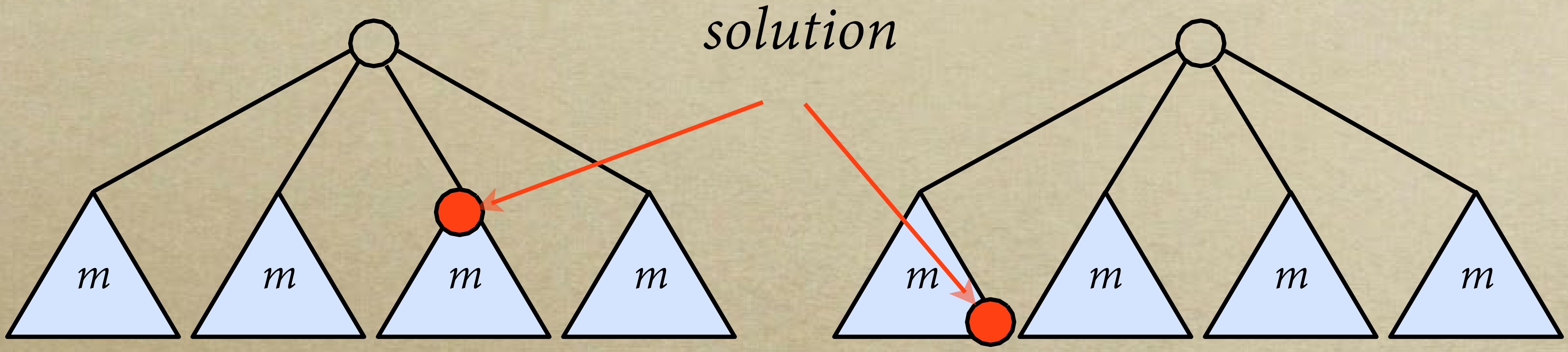
— *generate successor states of the current state*

— *explore each as an independent task*



Exploratory Decomposition Speedup

- ***Parallel formulation may perform a different amount of work***



total serial work = 2m + 1
total parallel work = 4

total serial work = m total parallel work = 4m

- ***Can cause super- or sub-linear speedup***

Speculative Decomposition

Dependencies between tasks are not always known a-priori

- ***makes it impossible to identify independent tasks***

Conservative approach

- ***identify independent tasks only when no dependencies left***

Optimistic (speculative) approach

- ***schedule tasks even when they may potentially be erroneous***

Drawbacks for each

- ***conservative approaches***
  - ***may yield little concurrency***
- ***optimistic approaches***
  - ***may require a roll-back mechanism if a dependence is encountered***

Speculative Decomposition in Practice

Discrete event simulation

- Data structure: centralized time-ordered event list
- Simulation
  - — extract next event in time order
  - — process the event
  - — if required, insert new events into the event list
- Optimistic event scheduling
  - — assume outcomes of all prior events
  - — speculatively process next event
  - — if assumption is incorrect, roll back its effects and continue

Time Warp

David Jefferson. "Virtual Time,"
ACM TOPLAS, *7(3):404-425, July 1985*

Speculative Decomposition in Practice

Time Warp OS http://bit.ly/twos-94

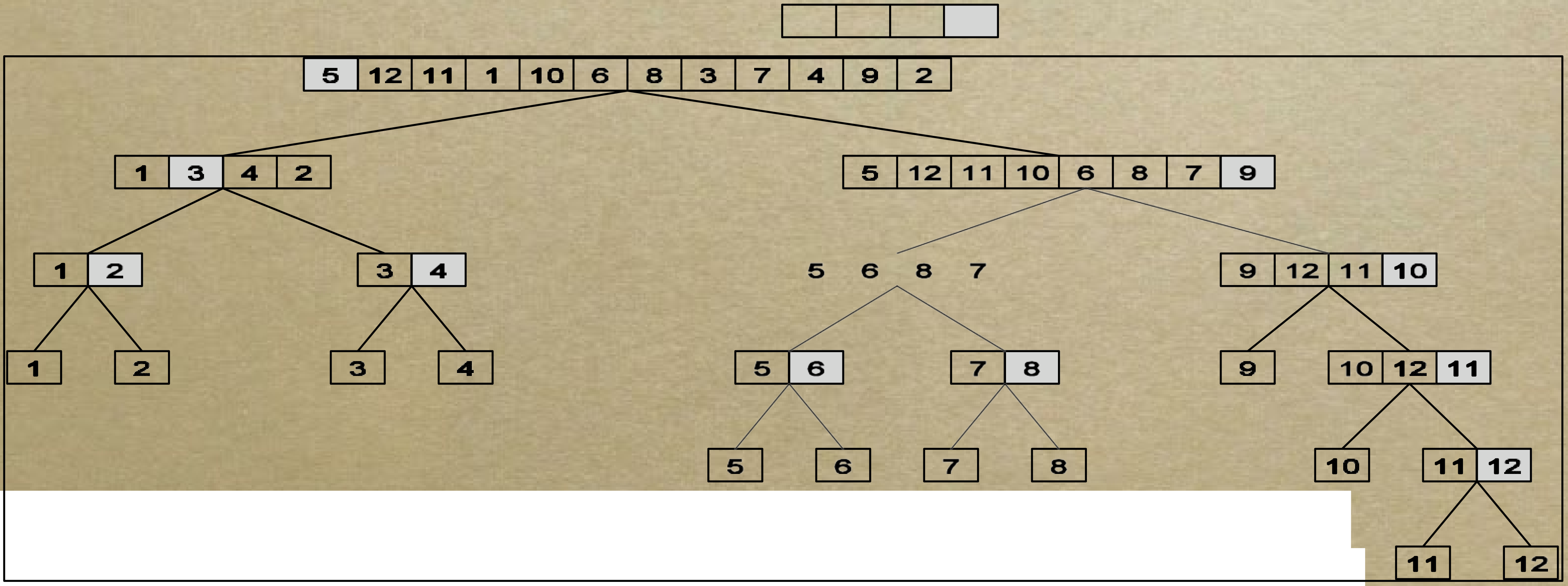
- A new operating system for military simulations
  —expensive computational tasks
  —composed of many interacting subsystems
  —highly irregular temporal behavior
- Optimistic execution and process rollback
  —don't treat rollback as a special case for handling exceptions, breaking deadlock, aborting transactions, …
  —use rollback as frequently as other systems use blocking
- Why a new OS?
  —rollback forces a rethinking of all OS issues
    – scheduling, synchronization, message queueing, flow control, memory management, error handling, I/O, and commitment
  —building Time Warp on top of an OS would require two levels of synchronization, two levels of message queues, …

Hybrid Decomposition

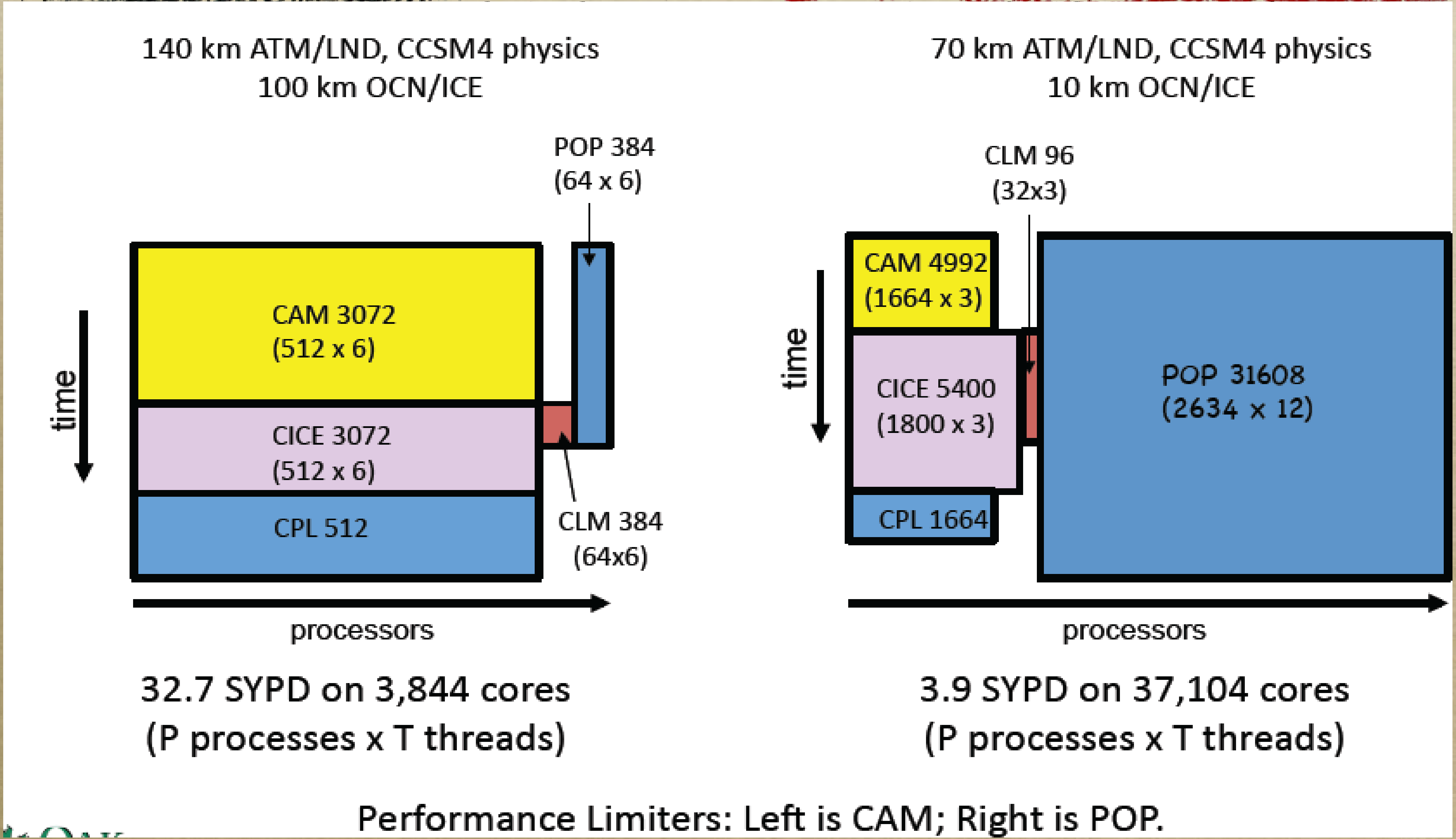
Use multiple decomposition strategies together

Often necessary for adequate concurrency

- *Quicksort*
  - *recursive decomposition alone limits concurrency*
  - *augmenting recursive with data decomposition is better*
    - *can use data decomposition on input data to compute a split*



Data decomposition within atmosphere, ocean, land, and sea-ice tasks



32.7 SYPD on 3,844 cores
(P processes x T threads)

3.9 SYPD on 37,104 cores
(P processes x T threads)

Performance Limiters: Left is CAM; Right is POP.

Figure courtesy of Pat Worley (ORNL)

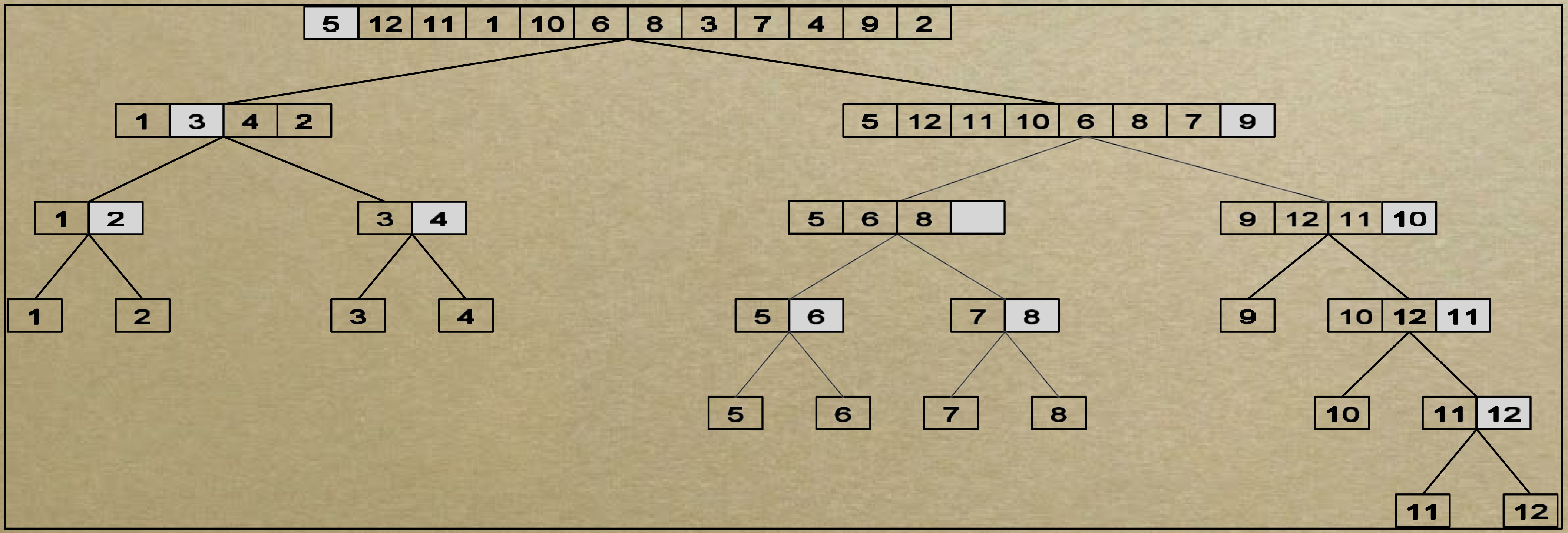
Task Generation

- Static task generation
  - —identify concurrent tasks a-priori
  - —typically decompose using data or recursive decomposition
  - —examples
    - matrix operations
    - graph algorithms on static graphs
    - image processing applications
    - other regularly structured problems
- Dynamic task generation
  - —identify concurrent tasks as a computation unfolds
  - —typically decompose using exploratory or speculative decompositions
  - —examples
    - puzzle solving
    - game playing

Task Size

- Uniform: all the same size
- Non-uniform
  - sometimes sizes known or can be estimated a-priori
  - sometimes not
    - example: tasks in quicksort

size of each partition depends upon pivot selected



Size of Data Associated with Tasks

- ***Data may be small or large compared to the computation***
  - ***size(input) < size(computation), e.g., 15 puzzle***
  - ***size(input) = size(computation) > size(output), e.g., min***
  - ***size(input) = size(output) < size(computation), e.g., sort***

- ***Implications***
  - ***small data: task can easily migrate to another thread***
  - ***large data: ties the task to a thread***
    - ***possibly can avoid communicating the task context***

      reconstruct/recompute the context elsewhere

Characteristics of Task Interactions

Orthogonal classification criteria

- ***Static vs. dynamic***
- ***Regular vs. irregular***
- ***Read-only vs. read-write***
- ***One-sided vs. two-sided***

Characteristics of Task Interactions

-

- ***Static interactions***
  - ***—tasks and interactions are known a-priori***
  - ***—simpler to code***
- ***Dynamic interactions***
  - ***—timing or interacting tasks cannot be determined a-priori***
  - ***—harder to code***
    - – ***especially using two-sided message passing APIs***

Characteristics of Task Interactions

Regular interactions

—*interactions have a pattern that can be described with a function*

- *e.g. mesh, ring*

—*regular patterns can be exploited for efficient implementation*

- *e.g. schedule communication to avoid conflicts on network links*

Irregular interactions

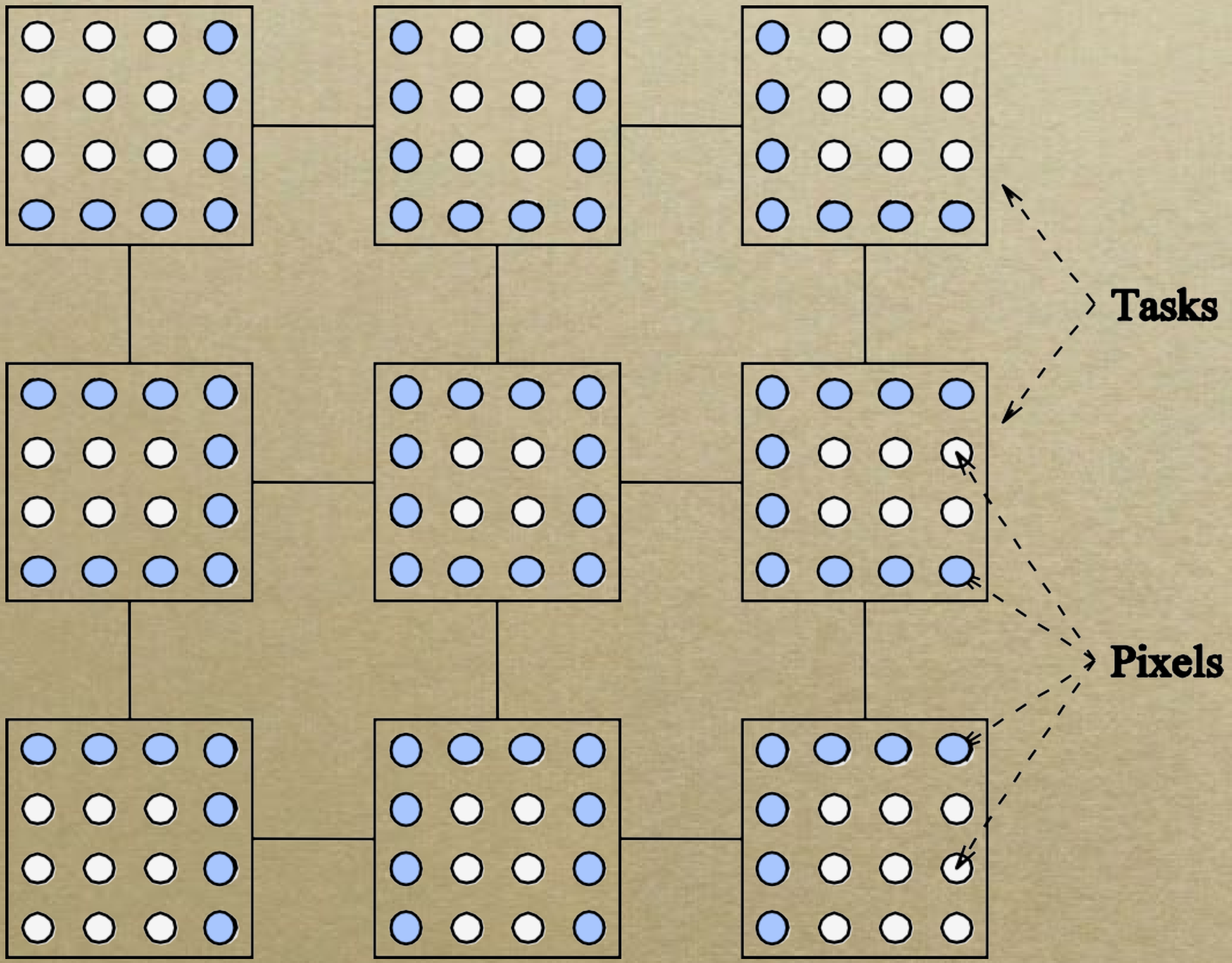
—*lack a well-defined topology*

—*modeled by a graph*

Static Regular Task Interaction Pattern

Image operations, e.g., edge detection

Nearest neighbor interactions on a 2D mesh



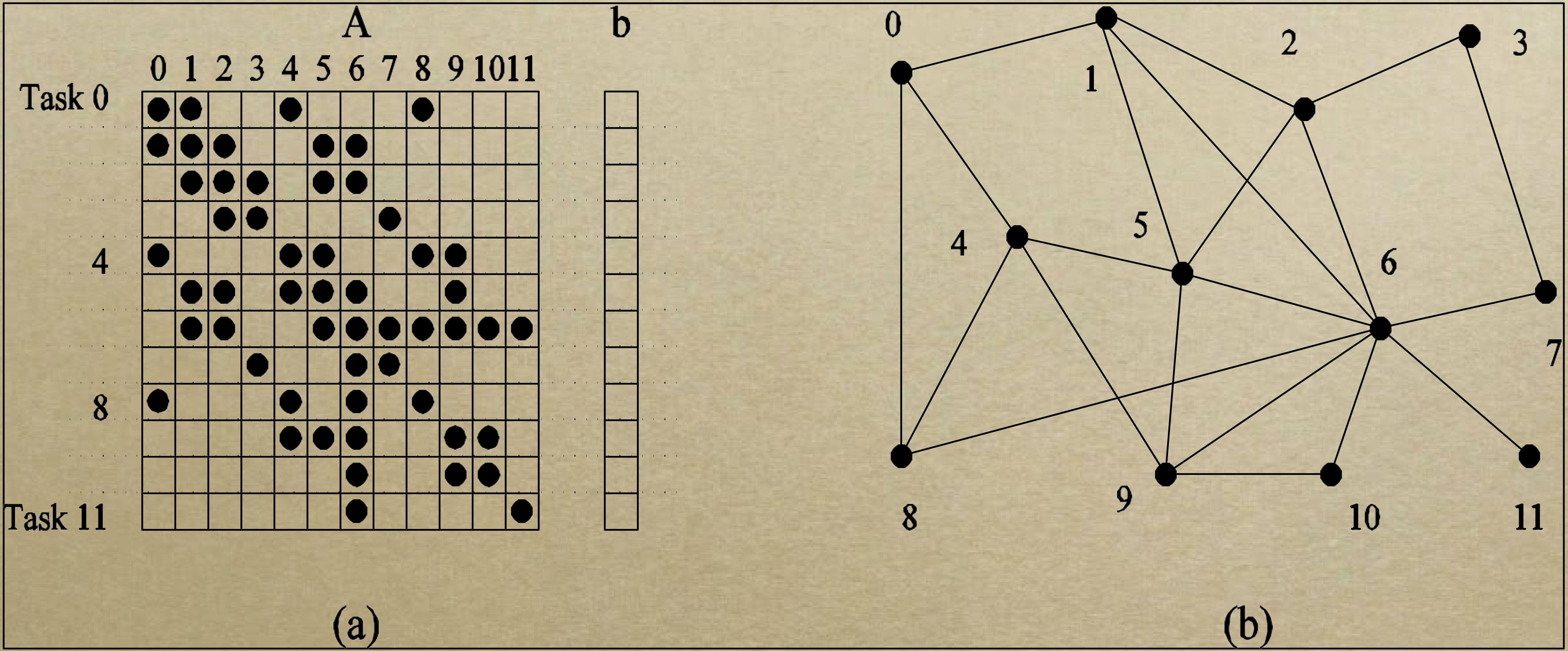
Sobel Edge Detection Stencils

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Static Irregular Task Interaction Pattern

Sparse matrix-vector multiply



Characteristics of Task Interactions

Read-only interactions

- *tasks only read data associated with other tasks*

Read-write interactions

- *read and modify data associated with other tasks*
- *harder to code: requires synchronization*
  - *need to avoid read-write and write-write ordering races*

Characteristics of Task Interactions

One-sided

—initiated & completed independently by 1 of 2 interacting tasks

- ***READ or WRITE***
- ***GET or PUT***

Two-sided

—both tasks coordinate in an interaction

- ***SEND and RECV***

☛ • *Mapping techniques for load balancing*
  - *static mappings*
  - *dynamic mappings*
• *Methods for minimizing interaction overheads*
• *Parallel algorithm design templates*

Mapping Techniques

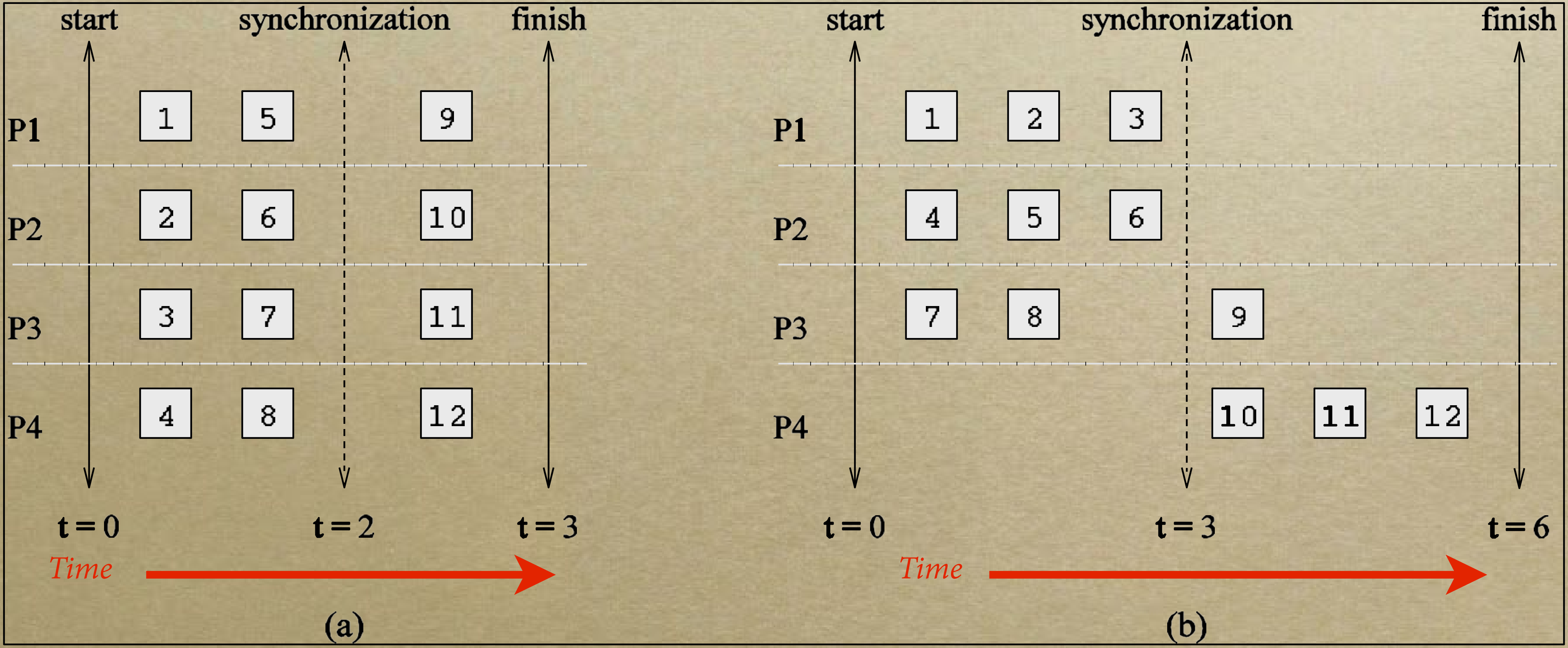
Map concurrent tasks to threads for execution

- ***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 for Minimum Idling

Must simultaneously minimize idling and load balance

Balancing load alone does not minimize idling



Mapping Techniques for Minimum Idling

Static vs. dynamic mappings

- **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., even decomposition 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*

Factors that influence choice of mapping

- *size of data associated with a task*
- *nature of underlying domain*

Schemes for Static Mapping

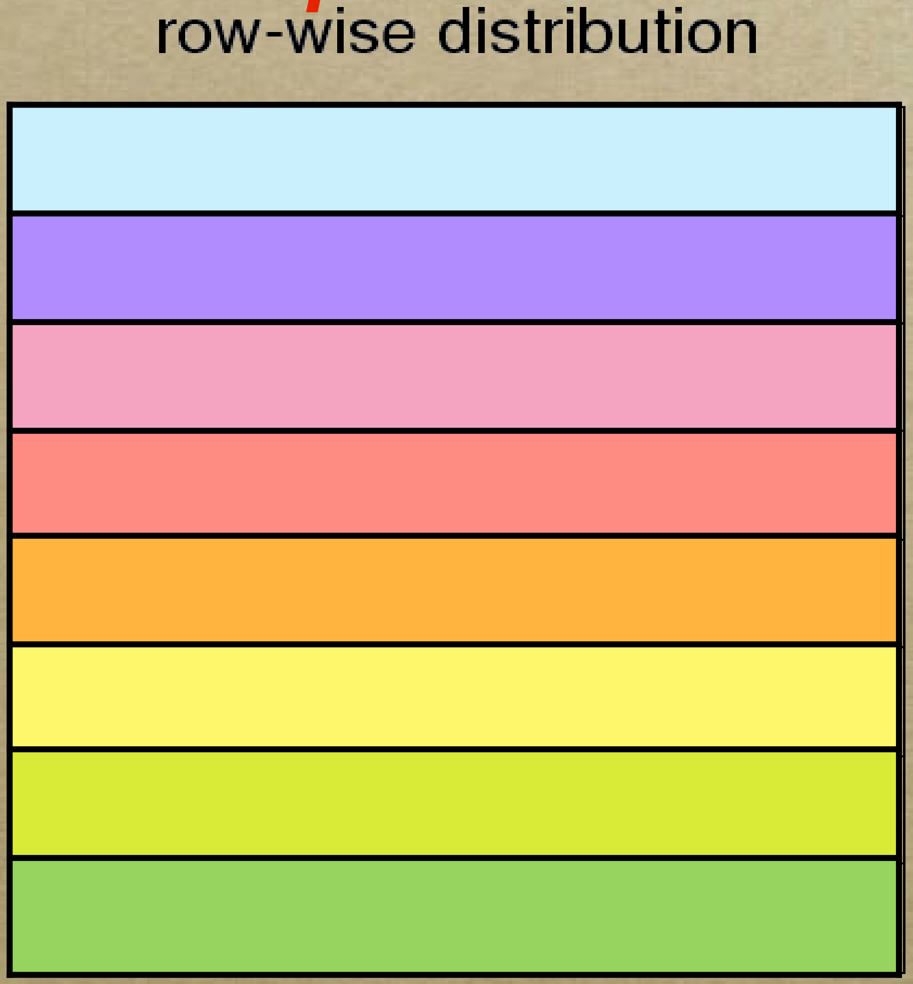
- ***Data partitionings***
- ***Task graph partitionings***
- ***Hybrid strategies***

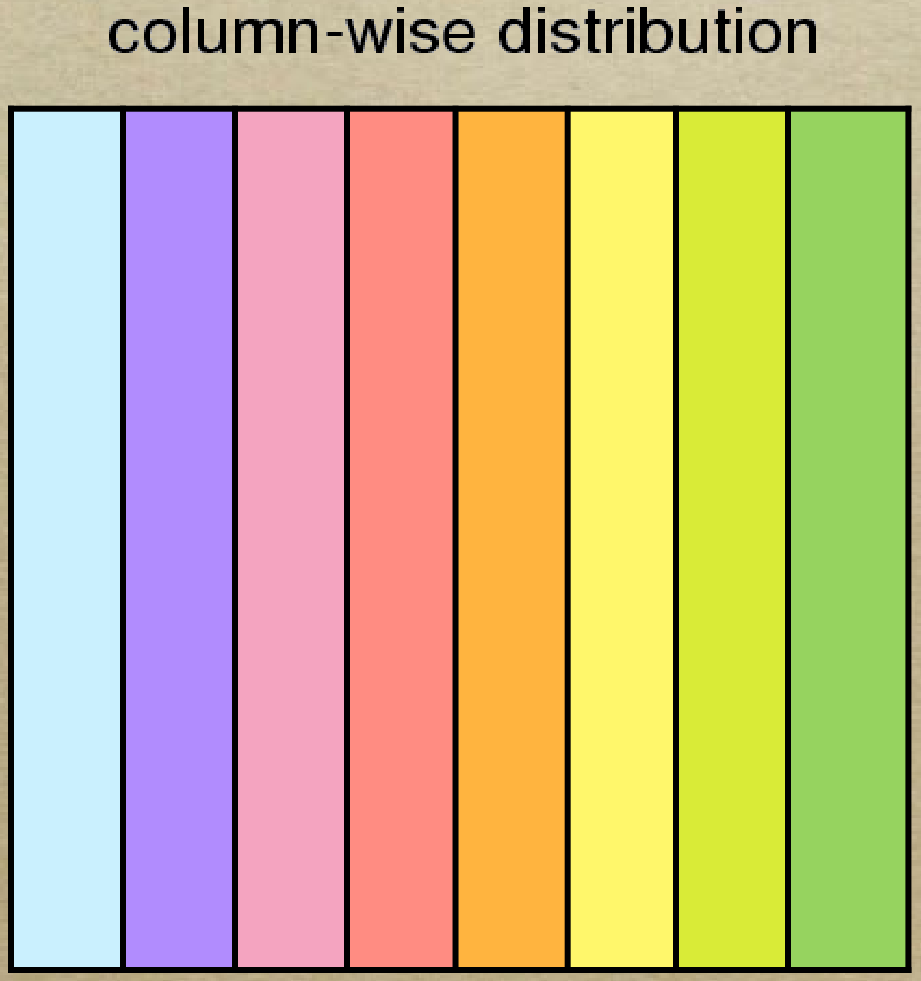
Mappings Based on Data Partitioning

Partition computation using a combination of

—***data partitioning***
—***owner-computes rule***

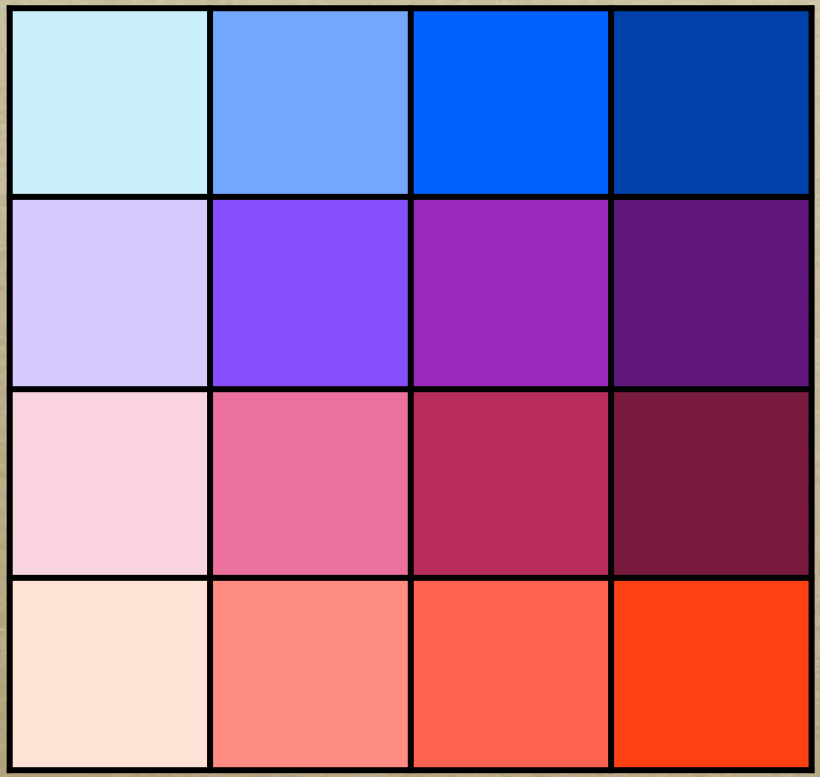
Example: 1-D block distribution for dense matrices

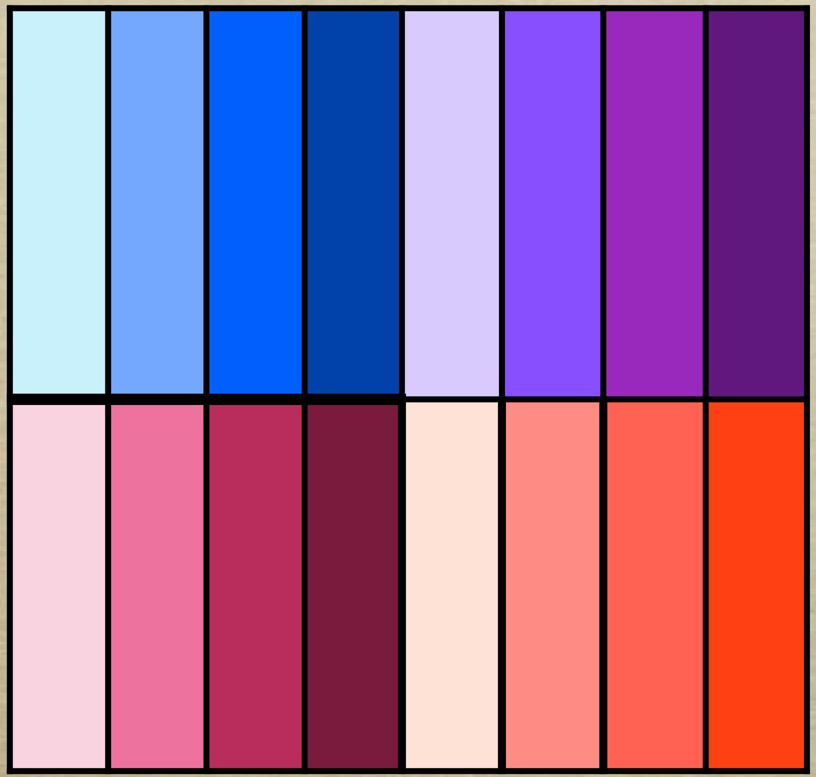




Block Array Distribution Schemes

Multi-dimensional block distributions





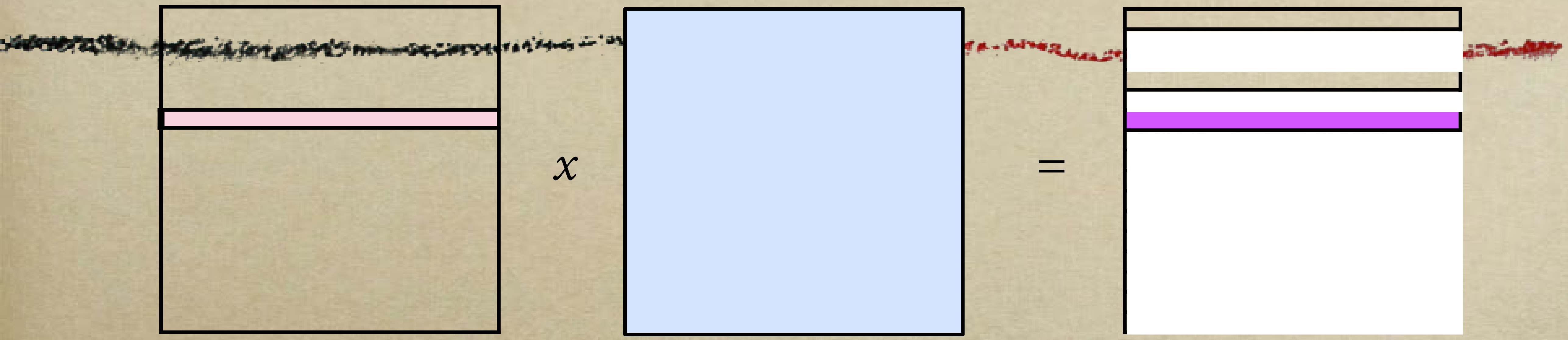
Multi-dimensional partitioning enables larger # of threads

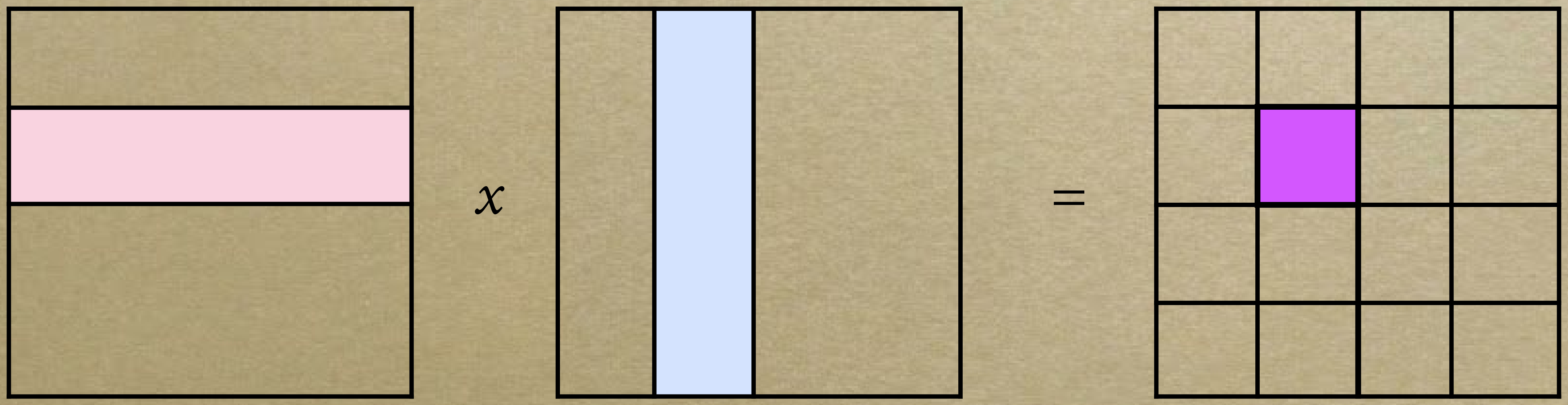
Block Array Distribution Example

Multiplying two dense matrices C = A x B

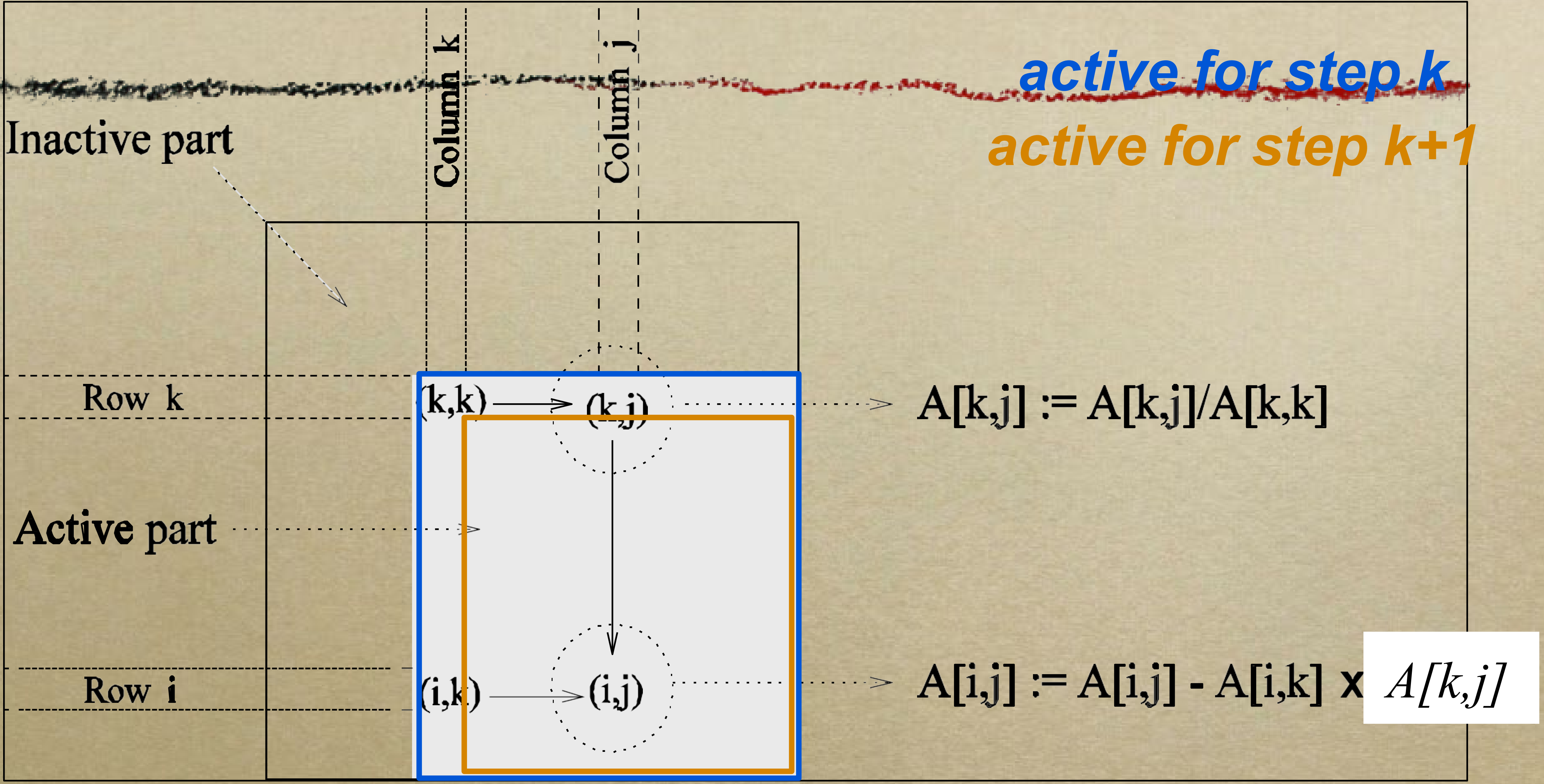
- ***Partition the output matrix C using a block decomposition***
- ***Give each task the same number of elements of C***
  - ***each element of C corresponds to a dot product***
  - ***even load balance***
- ***Obvious choices: 1D or 2D decomposition***
- ***Select to minimize associated communication overhead***

Data Usage in Dense Matrix Multiplication





Consider: Gaussian Elimination



Active submatrix shrinks as elimination progresses

Imbalance and Block Array Distributions

- ***Consider a block distribution for Gaussian Elimination***
  - *amount of computation per data item varies*
  - *a block decomposition would lead to significant load imbalance*

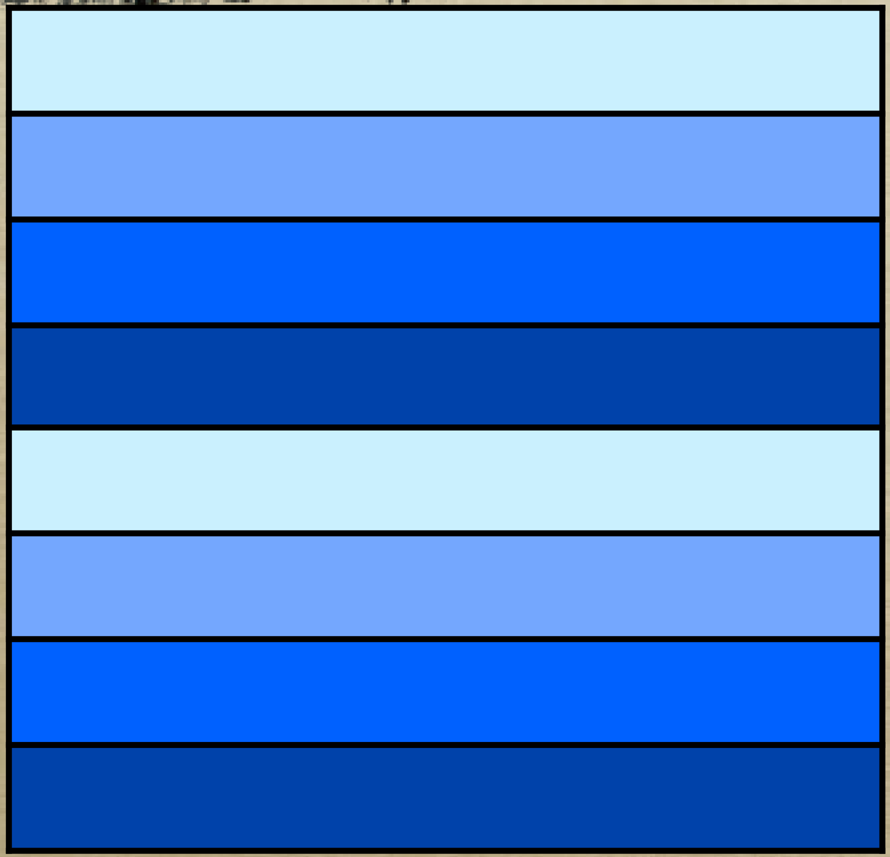
Block Cyclic Distribution

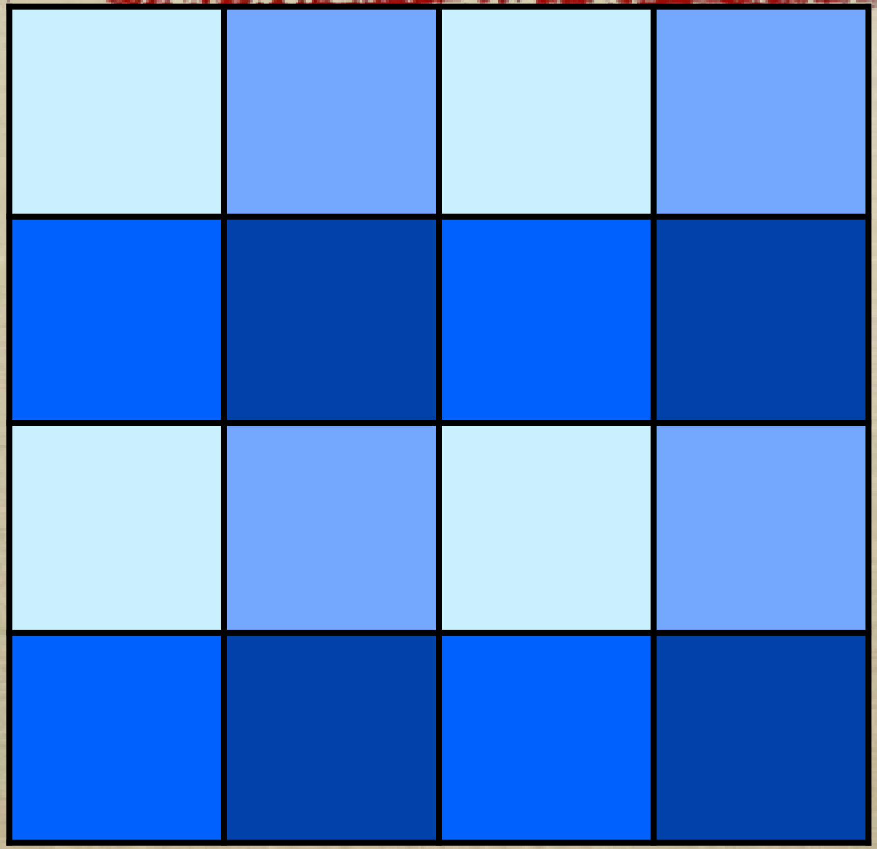
Variant of the block distribution scheme that can be used to alleviate the load-imbalance and idling

Steps

1. ***partition an array into many more blocks than the number of available threads or processes***
2. ***round-robin assignment of blocks to threads or processes***
   - ***each thread or process gets several non-adjacent blocks***

Block-Cyclic Distribution


1D block-cyclic


2D block-cyclic

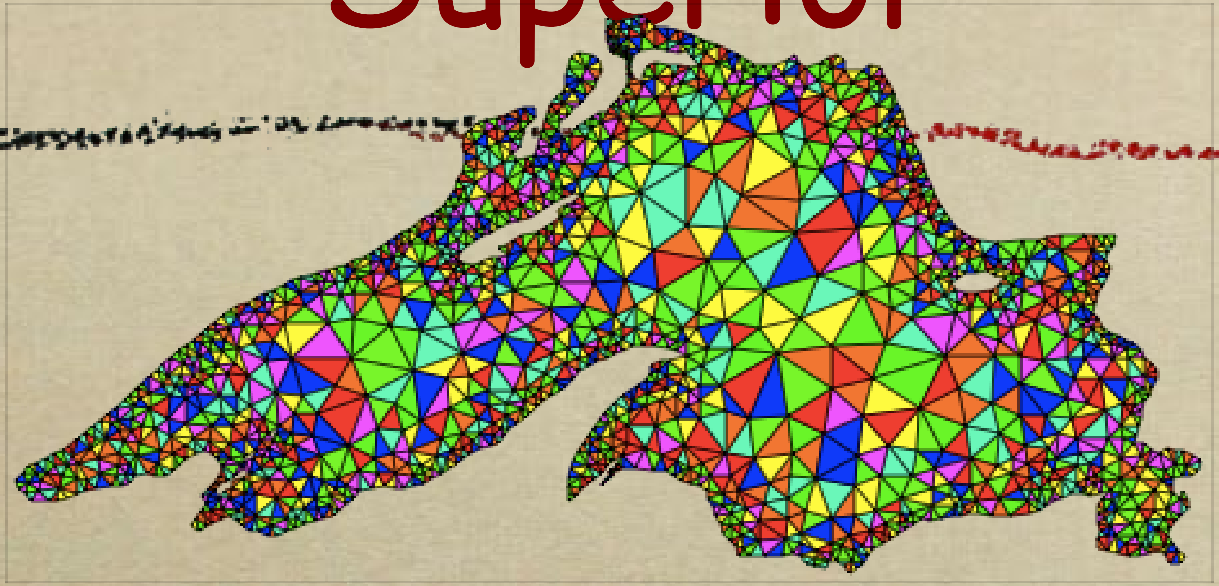
- Cyclic distribution: special case with block size = 1
- Block distribution: special case with block size is n/p
  —n is the dimension of the matrix; p is the # of threads

Decomposition by Graph Partitioning

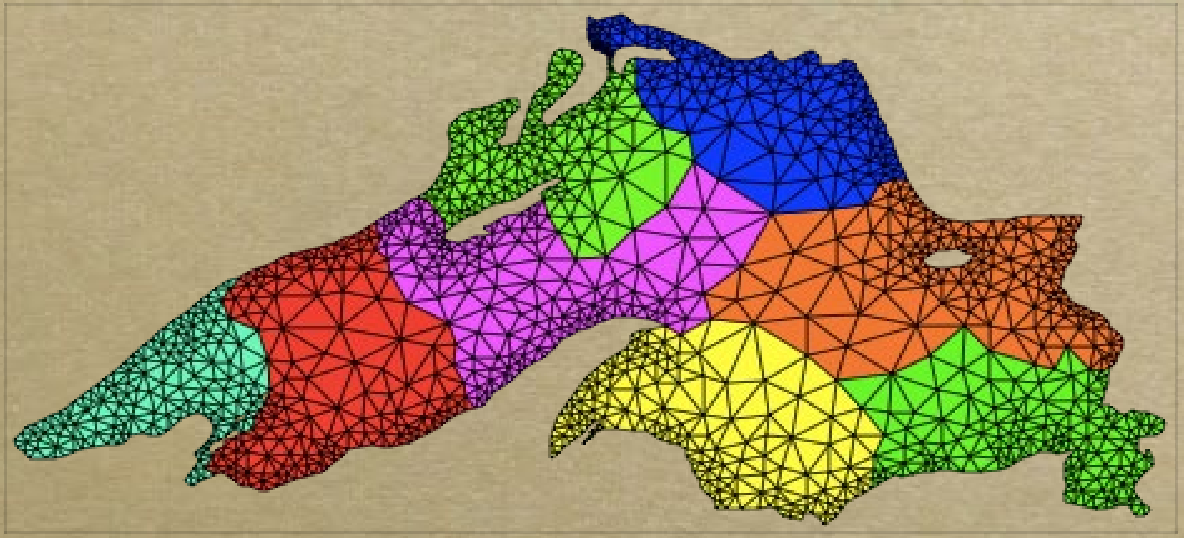
Sparse-matrix vector multiply

- ***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 equal number of nodes to each thread*
  - *minimize edge count of the graph partition*

Partitioning a Graph of Lake Superior



Random Partitioning



Partitioning for minimum edge-cut

Mappings Based on Task Partitioning

Partitioning a task-dependency graph

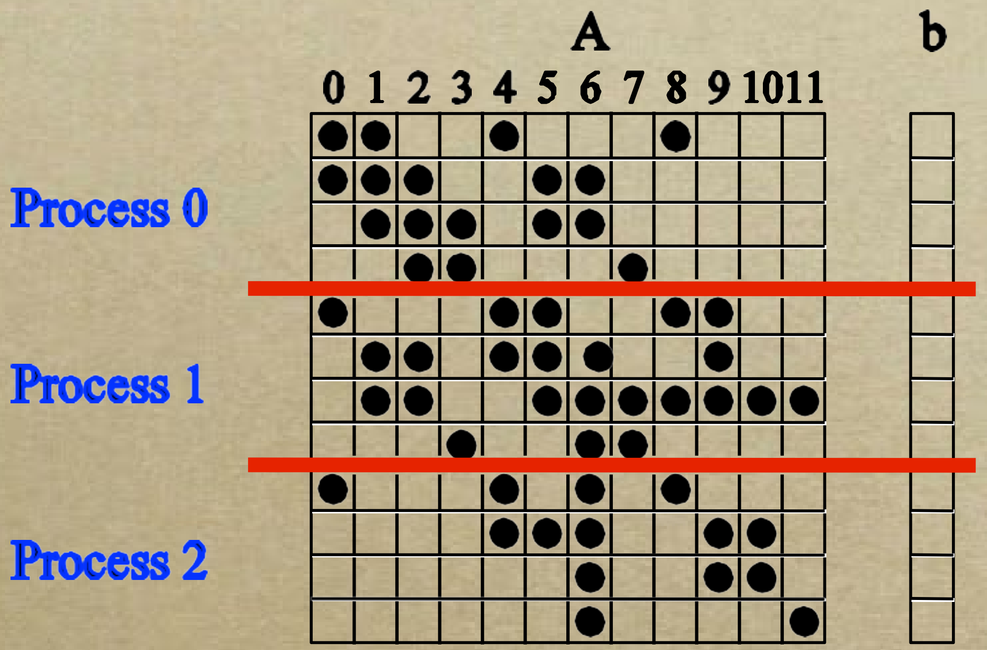
- ***Optimal partitioning for general task-dependency graph***

  — NP-hard problem

- ***Excellent heuristics exist for structured graphs***

Mapping a Sparse Matrix

Sparse matrix-vector product



sparse matrix structure

17 items to communicate

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

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

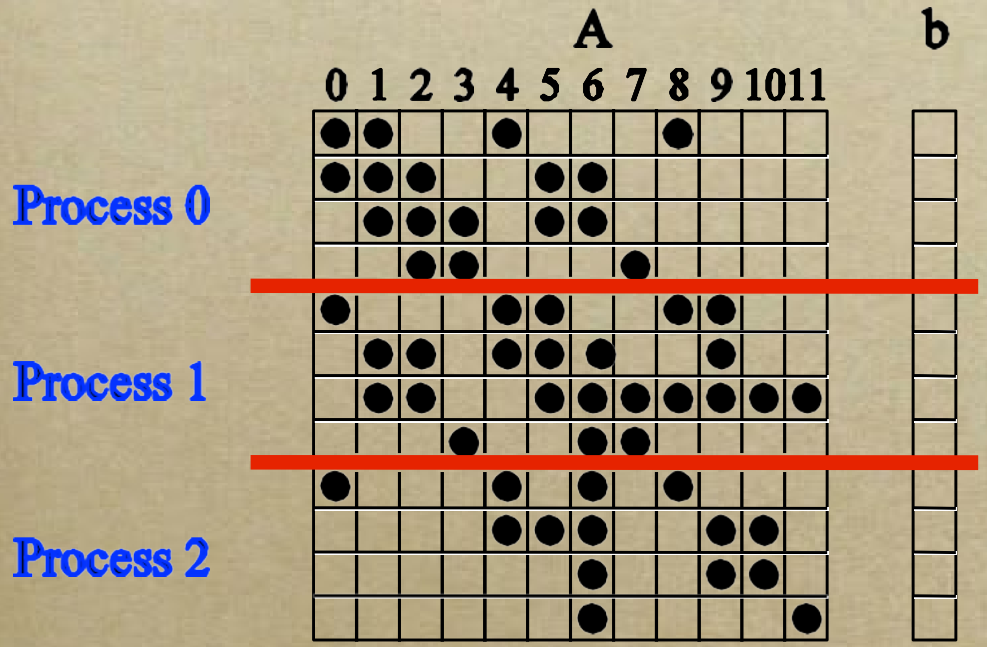
C2 = (0,4,5,6)

mapping

partitioning

Mapping a Sparse Matrix

Sparse matrix-vector product



17 items to communicate

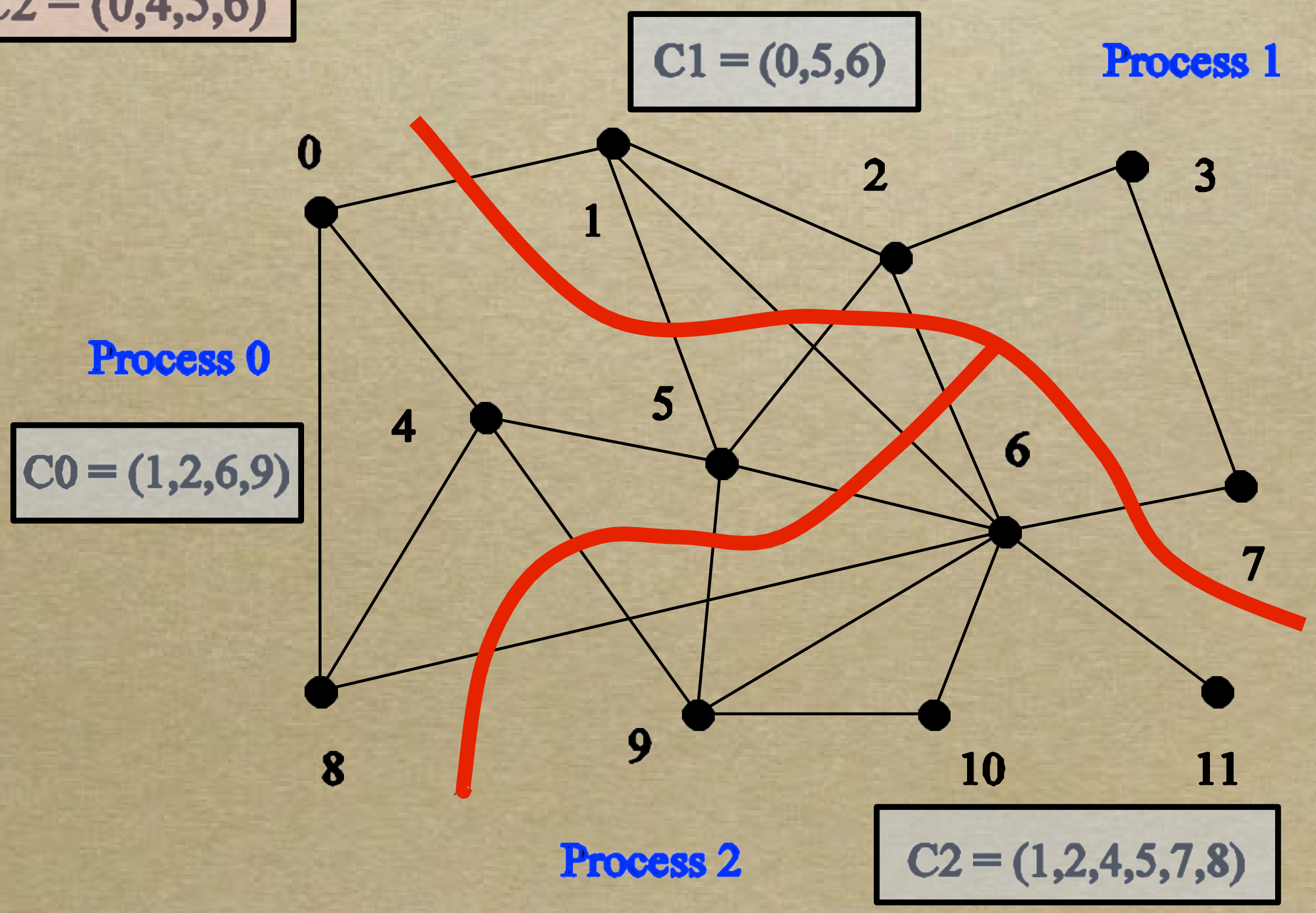
C0 = (4,5,6,7,8)

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

C2 = (0,4,5,6)

13 items to communicate

sparse matrix structure

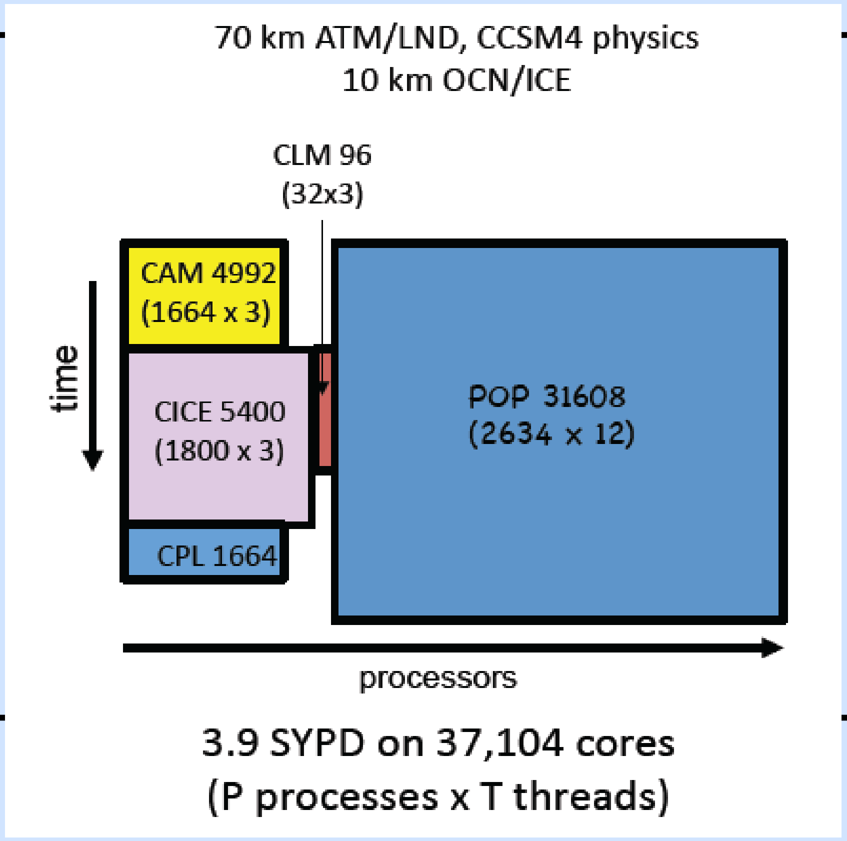


mapping
partitioning

Hierarchical Mappings

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

Example:
Hybrid Decomposition
+ Data Partitioning for
Community Earth System Model



☛ — *dynamic mappings*

- ***Methods for minimizing interaction overheads***
  - ***Parallel algorithm design templates***

Schemes for Dynamic Mapping

Dynamic mapping AKA dynamic load balancing

- *—load balancing is the primary motivation for dynamic mapping*

Styles

- *—centralized*
- *—distributed*

Centralized Dynamic Mapping

- Threads types: masters or slaves

General strategy

- —when a slave runs out of work → request more from master
- Challenge
- —master may become bottleneck for large # of threads
- Approach
- —chunk scheduling: thread picks up several of tasks at once
- —however
    - ***large chunk sizes may cause significant load imbalances***
    - ***gradually decrease chunk size as the computation progresses***

Distributed Dynamic Mapping

All threads as peers

Each thread can send or receive work from other threads

- *—avoids centralized bottleneck*

Four critical design questions

- *—how are sending and receiving threads paired together?*
- *—who initiates work transfer?*
- *—how much work is transferred?*
- *—when is a transfer triggered?*

Ideal answers can be application specific

Cilk uses a distributed dynamic mapping: “work stealing”

☛ • ***Methods for minimizing interaction overheads***
    • ***Parallel algorithm design templates***

Minimizing Interaction Overheads

"Rules of thumb"

- ***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 (avoidance)*

Minimizing Interaction Overheads

Techniques

- ***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 instead of point-to-point primitives***
- ***Issue multiple communications and overlap their latency***

(reduces exposed latency)

Parallel Algorithm Model

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

Common Parallel Algorithm Templates

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

Approaches for parallel algorithms design

1. Parallelize the existing sequential algorithms or modify an existing sequential algorithm exploiting those parts of the algorithm that are naturally parallelizable.
2. Design a completely new parallel algorithm that might be adapted to parallel architectures.
3. Design a new parallel algorithm from the existing parallel algorithm.

Relation between parallel architectures and parallel algorithms

Parallel Algorithm	**Parallel Architecture**
• Granularity Complexity	Concurrency Operation Mode
• Data Structure	Memory Structure
• Communication Environment	Interconnection Network
• Algorithm Size	Number of Processing Elements
• Programming Approach	Architecture Category

Complexities

- The complexity of a computation measures the space and time requirements of that computation.
- The time complexity is measured by the number of required execution time units
- The space complexity is measured as one unit of space for every register required for an individual computation.

Sequential Computational Complexity

- sequential computation is the function T(n) which is the
- maximum time taken by the program to execute over all inputs of size n.

Parallel Computation Complexity

- We can assume that the computation model consists of p processors only, where p > 1.
- This is referred to as ***bounded parallelism***.
- ***Unbounded parallelism*** *refers to the situation in which it is assumed* that we have an unlimited number of processors.

Parallel Computation Complexity

- The time complexity of a parallel algorithm to solve a problem of size n is a function T(n,p) which is the maximum time that elapses between the start of the algorithm's execution by one processor and its termination by one or more processors with regard to any arbitrary input data.

Parallel Computation Complexity

Operations associated with parallel algorithms

- Elementary operations is an arithmetic or logical operation performed locally by a processor. Which can be executed simultaneously by a set of processors,
- The time complexity of an elementary step is referred to as constant or O(1).

Parallel Computation Complexity

Operations associated with parallel algorithms

- Data routing operations is a *data routing step refers to the set* of data routing operations which can be executed among a set of processors.
- Which depends on the interconnection pattern used in the parallel system.

Parallel Computation Complexity

- *Processor complexity*
- The maximum number of processors used by the algorithm as a function of the problem size n.
- A serial algorithm to find the maximum number of a set of n numbers has the time complexity O(n), since it requires (n - 1) comparisons
- A trivial parallel algorithm for the same problem has the time complexity *O(log n) and processor complexity O(n).*

Anomalies

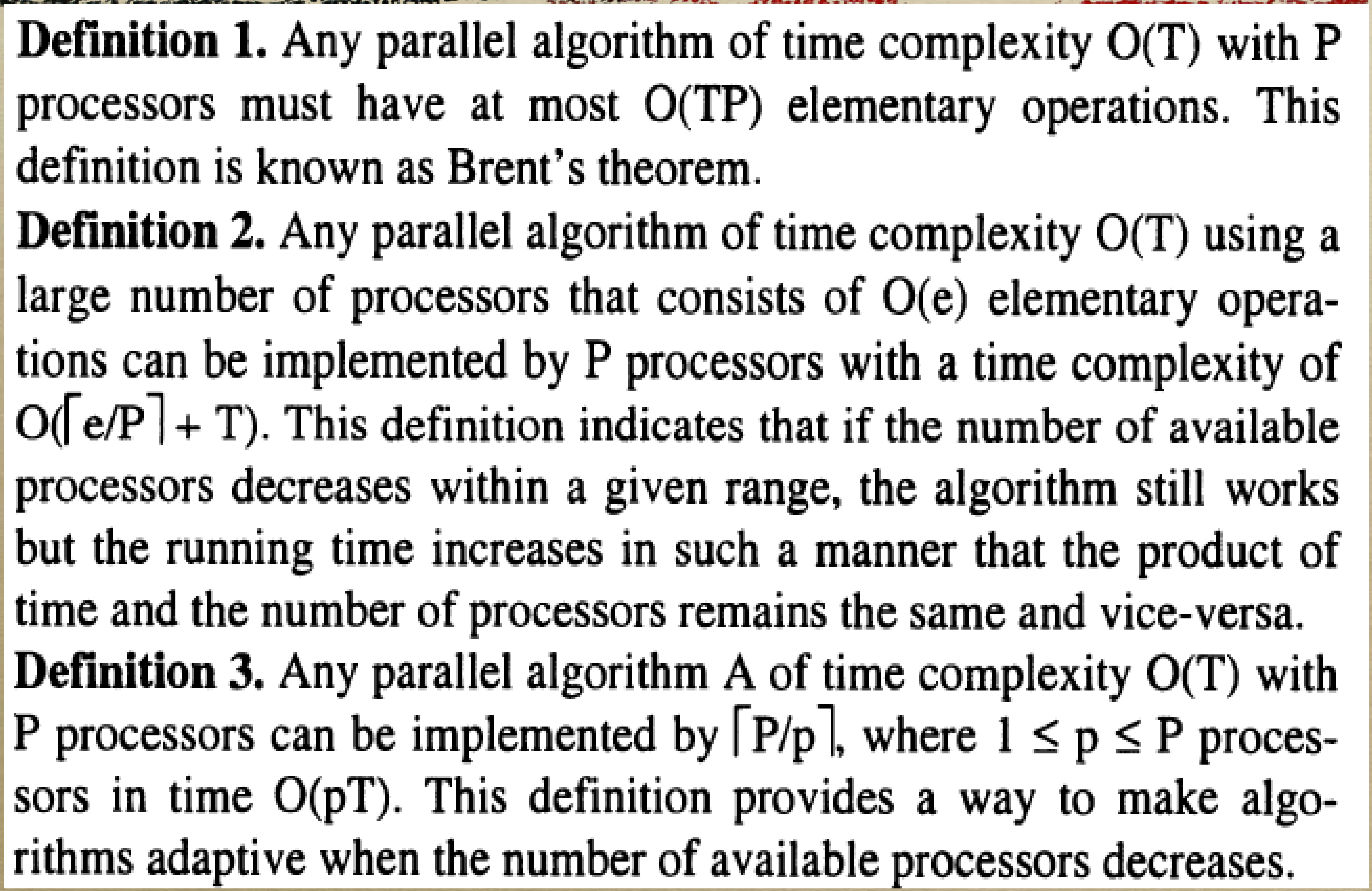
Sometimes the unbounded parallelism reflects an undesirable situation. For example, if we assume the algorithm can use as many processors as it wants and there are no communication and memory access restrictions, the computational time cannot be reduced below a certain limit. This is due to the fact that some intermediate results must be known before other parts of the computation proceed.

The unbounded parallel time complexity of a problem reflects this characteristic of a problem and is referred to as *anomalies.*

- A balanced distribution each processor contains either

  $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ data items.

Time Complexities



Definition 1. Any parallel algorithm of time complexity O(T) with P processors must have at most O(TP) elementary operations. This definition is known as Brent's theorem.

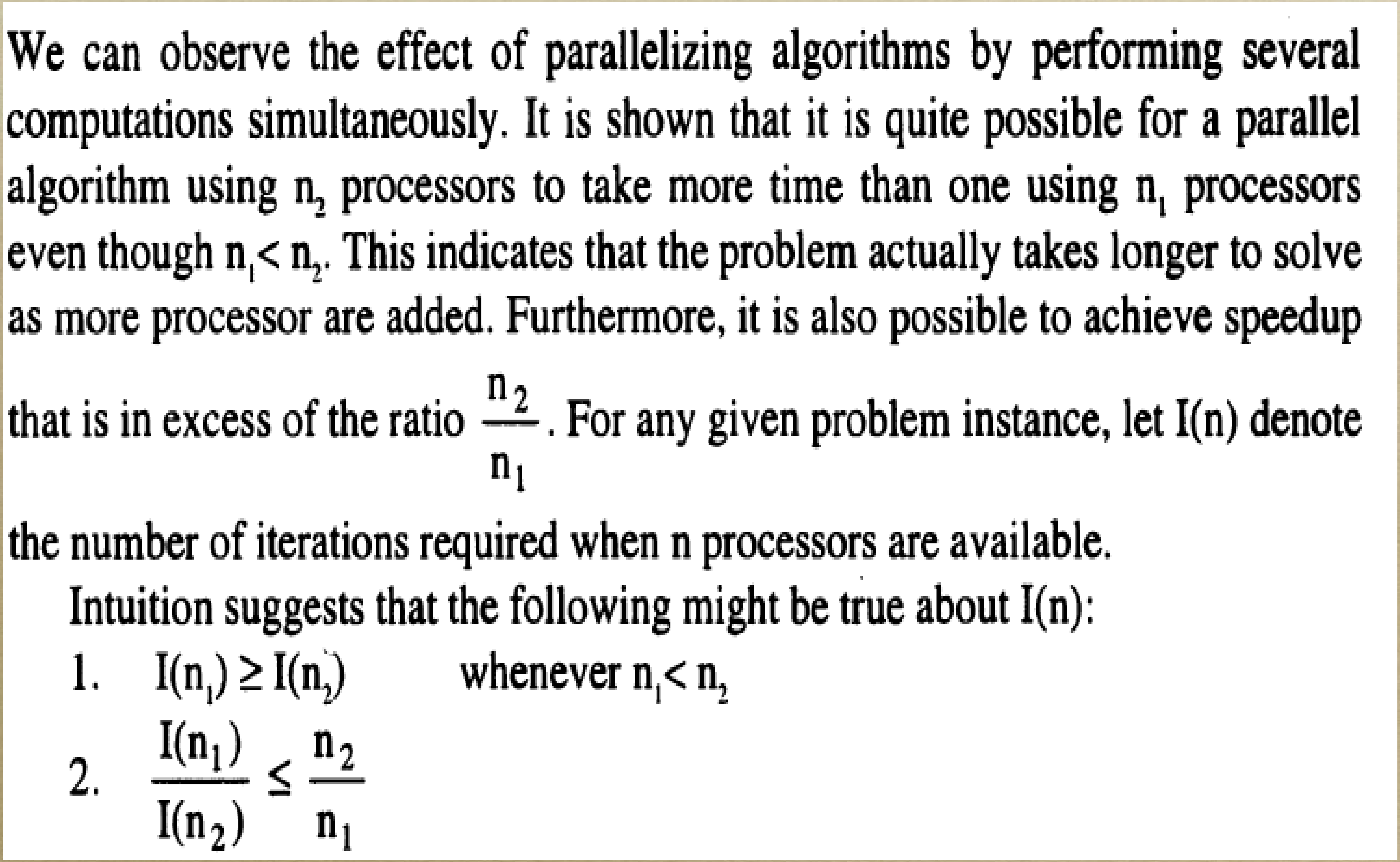
Definition 2. Any parallel algorithm of time complexity O(T) using a large number of processors that consists of O(e) elementary operations can be implemented by P processors with a time complexity of $O(\lceil e/P \rceil + T)$. This definition indicates that if the number of available processors decreases within a given range, the algorithm still works but the running time increases in such a manner that the product of time and the number of processors remains the same and vice-versa.

Definition 3. Any parallel algorithm A of time complexity O(T) with P processors can be implemented by $\lceil P/p \rceil$, where $1 \le p \le P$ processors in time O(pT). This definition provides a way to make algorithms adaptive when the number of available processors decreases.

Time Complexities

- The time complexity for an algorithm expresses its time requirements by giving, for each possible **input length, the largest amount of time needed by the algorithm to solve a problem of that size**

Anomalies in Parallel Algorithms

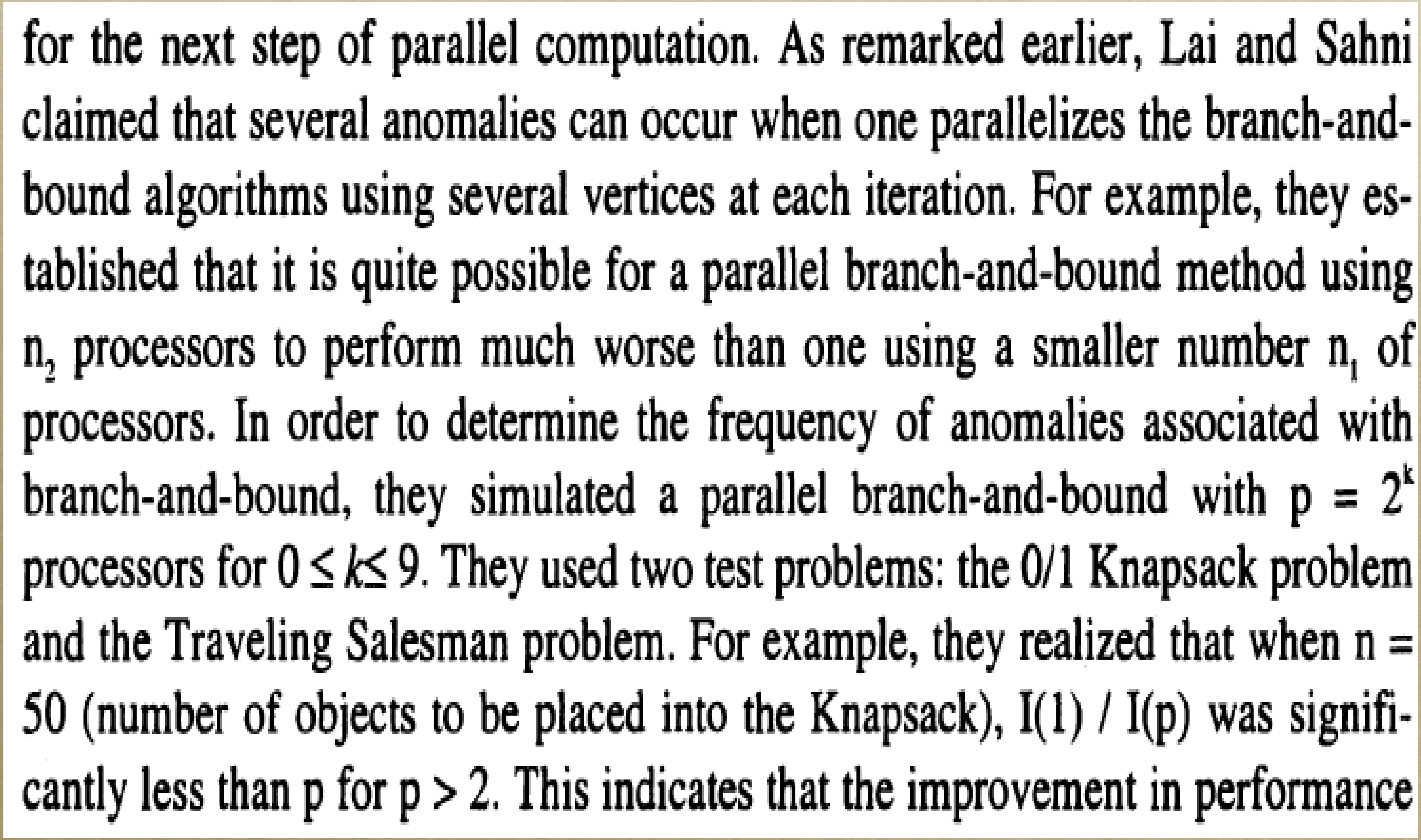


We can observe the effect of parallelizing algorithms by performing several computations simultaneously. It is shown that it is quite possible for a parallel algorithm using n_2 processors to take more time than one using n_1 processors even though $n_1 < n_2$. This indicates that the problem actually takes longer to solve as more processor are added. Furthermore, it is also possible to achieve speedup that is in excess of the ratio $\frac{n_2}{n_1}$. For any given problem instance, let I(n) denote the number of iterations required when n processors are available.

Intuition suggests that the following might be true about I(n):

1. $I(n_1) \geq I(n_2)$ whenever $n_1 < n_2$
2. $\frac{I(n_1)}{I(n_2)} \leq \frac{n_2}{n_1}$

Lai and Sahni {Branch and Bound Anomalies}


for the next step of parallel computation. As remarked earlier, Lai and Sahni claimed that several anomalies can occur when one parallelizes the branch-and-bound algorithms using several vertices at each iteration. For example, they established that it is quite possible for a parallel branch-and-bound method using n_2 processors to perform much worse than one using a smaller number n_1 of processors. In order to determine the frequency of anomalies associated with branch-and-bound, they simulated a parallel branch-and-bound with $p = 2^k$ processors for $0 \leq k \leq 9$. They used two test problems: the 0/1 Knapsack problem and the Traveling Salesman problem. For example, they realized that when n = 50 (number of objects to be placed into the Knapsack), I(1) / I(p) was significantly less than p for p > 2. This indicates that the improvement in performance

Types of anomalies

- Increasing p will lower the time for the first phase and raise
- the time for the second phase, which can increase the total execution time. In summary, for some executions the parallel version provides a solution after examining more alternatives, resulting in sublinear speedups. This is referred to as ***deceleration anomalies***. *Execution yielding speedups greater than p by using p* processors (superlinear speedup) is referred to as ***acceleration anomalies***.
- As a consequence, there is little advantage to expand more than *k vertices*