Scalable Algorithmic Techniques

Decompositions & Mapping

Alexandre David

1.2.05

adavid@cs.aau.dk

Introduction

- Focus on data parallelism, scale with size.
  - Task parallelism limited.
- Notion of scalability is fuzzy in the book.
  - More precision later.
  - Idea: You lose efficiency with # of processors, gain efficiency with the size of the problem. Scalability measures the ratio.
    - You can experiment with assignment 2 to see that.

In Practice...

- Typical tasks:
  - Identify concurrent works.
  - Map them to processors.
  - Distribute inputs, outputs, and other data.
  - Manage shared resources.
  - Synchronize the processors.

Basic Principles

- Large blocks of independent computations.
  - Rare, seti@home.
    - Better when computations >> size of data.
  - Matrix multiplication too.
- Good performance recipe:
  - minimize interaction (= communication)
  - maximize locality (= blocks of computation)

Minimizing Interaction Overheads

- Maximize data locality.
  - Minimize volume of data-exchange.
  - Minimize frequency of interactions.
- Minimize contention and hot spots.
  - Share a link, same memory block, etc...
  - Re-design original algorithm to change the interaction pattern.
  - Use task interaction graph to help.

Minimizing Interaction Overheads

- Overlapping computations with interactions – to reduce idling.
  - Initiate interactions in advance.
  - Non-blocking communications.
  - Multi-threading.
- Replicating data or computation.
- Group communication instead of point to point.
- Overlapping interactions.

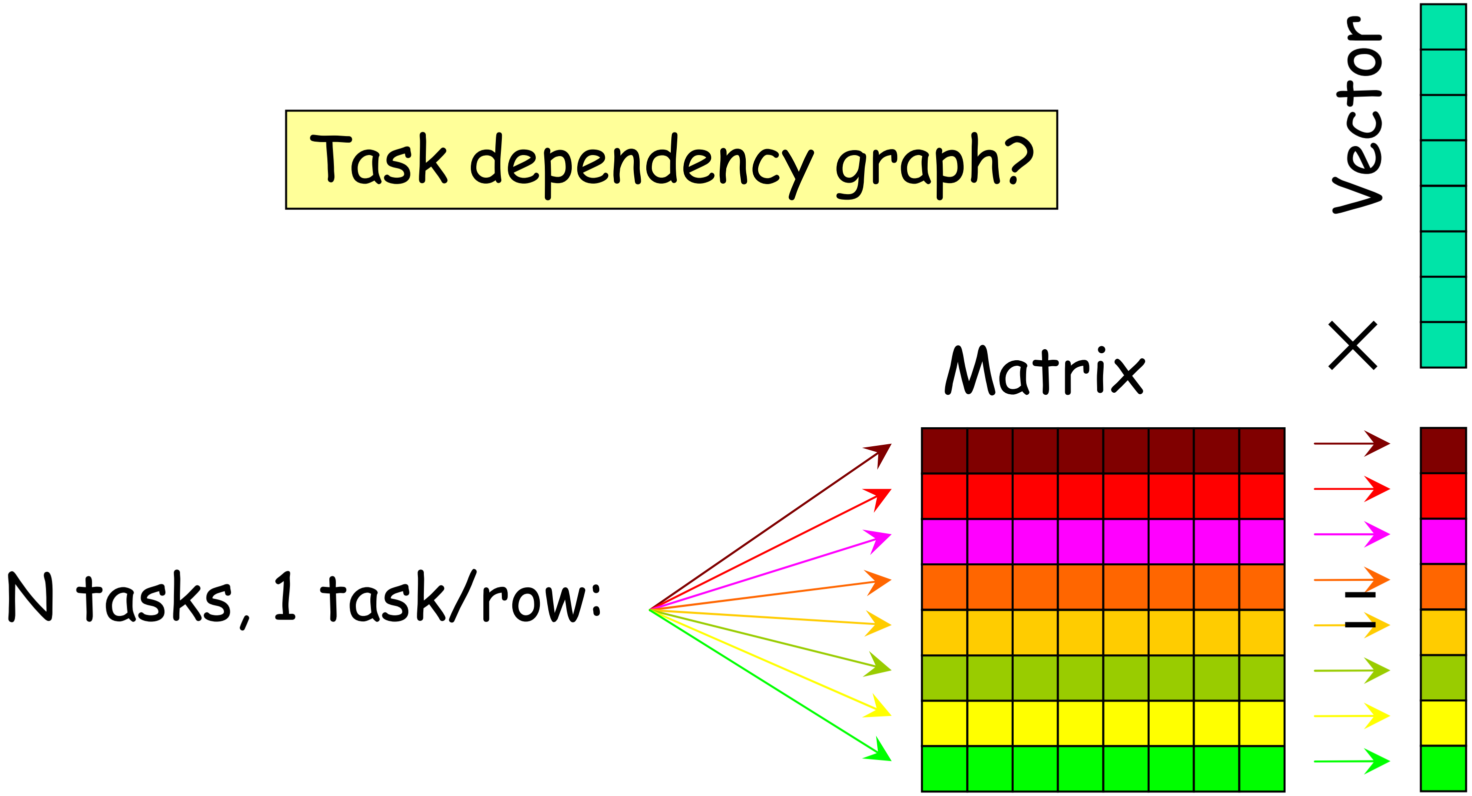
Decomposing Problems

- Decomposition into *concurrent* tasks.
  - No unique solution.
  - Different sizes.
  - Decomposition illustrated as a directed graph:
    - Nodes = tasks.
    - Edges = dependency.

! Task dependency graph

Example: Matrix * Vector

Task dependency graph?



Example: database query processing

MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

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

Table 3.1 A database storing information about used vehicles.

A solution

Measure of concurrency?
Nb. of processors?
Optimal?

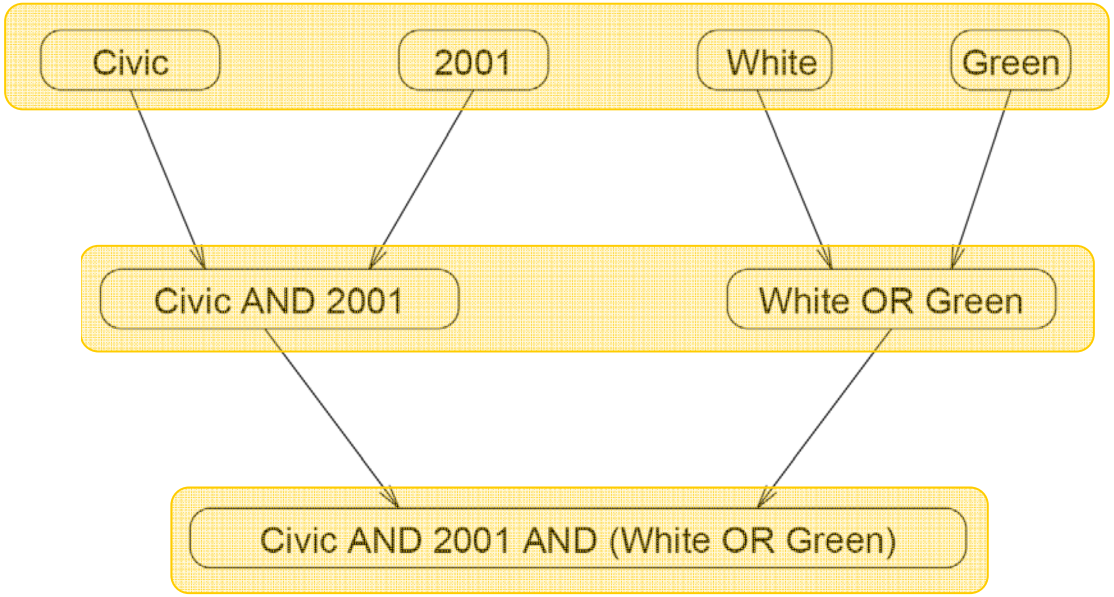


Figure 3.2 The different tables and their dependencies in a query processing operation.

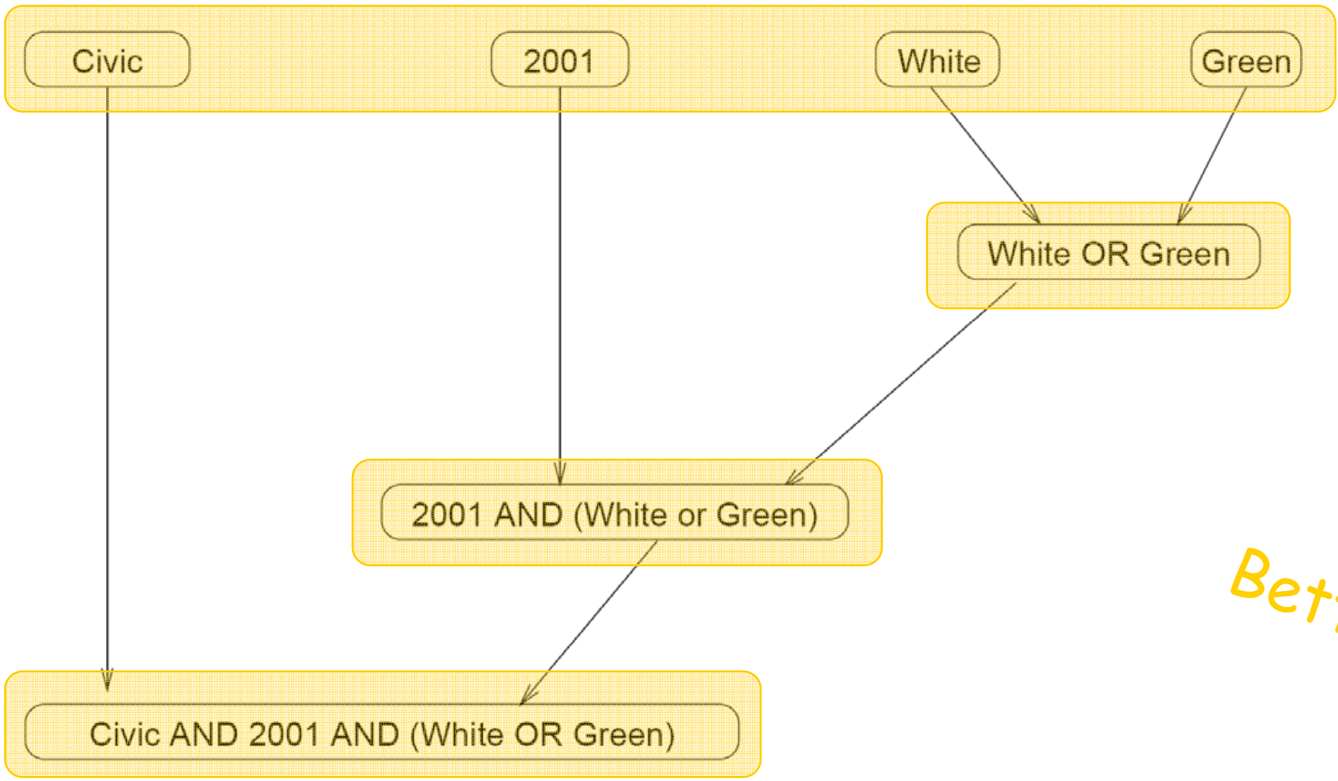
Another Solution

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



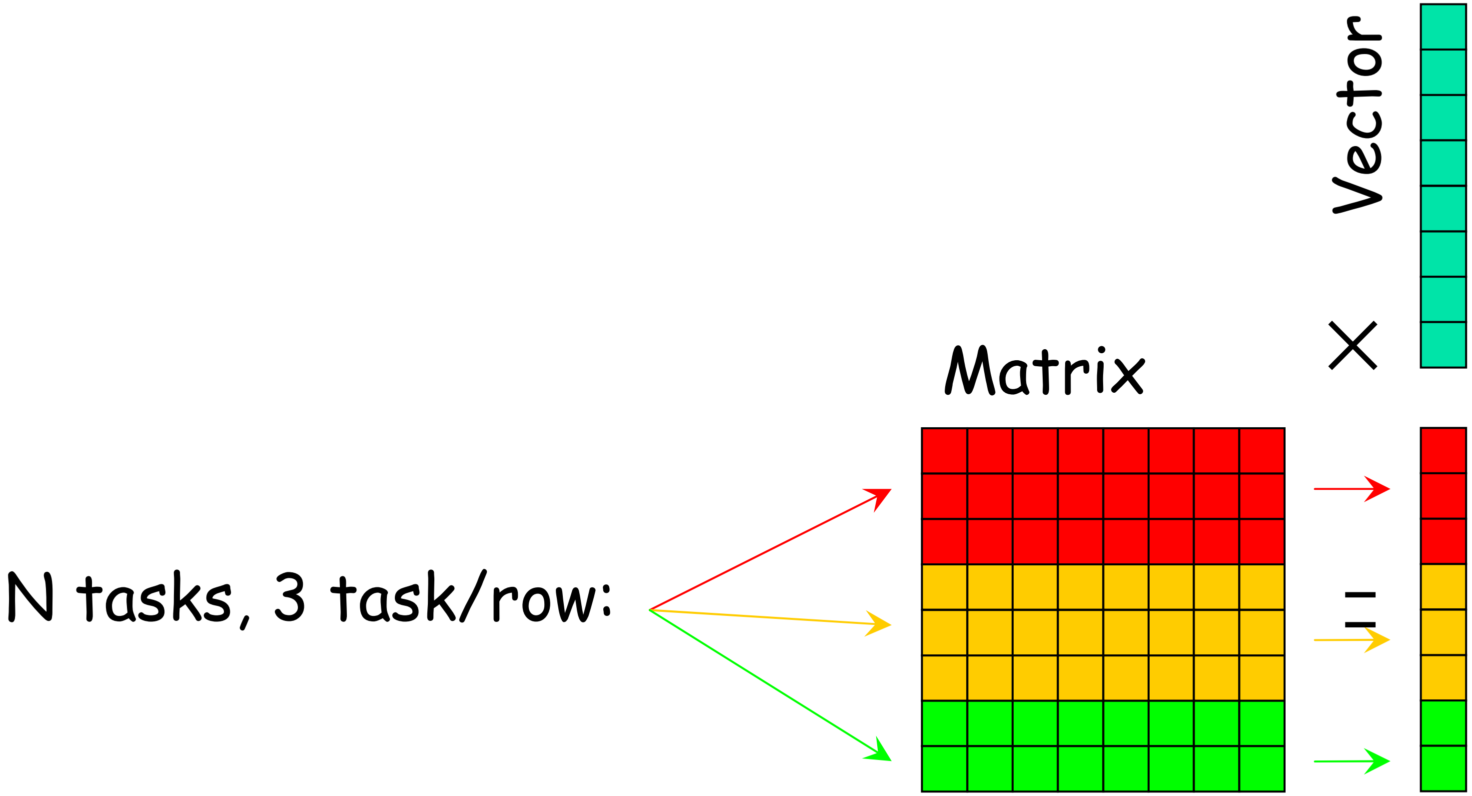
Better/worse?

Granularity

- Number and size of tasks.
  - Fine-grained: many small tasks.
  - Coarse-grained: few large tasks.

- Related: *degree of concurrency*.
  (Nb. of tasks executable in parallel).
  - Maximal degree of concurrency.
  - Average degree of concurrency.

Coarser Matrix * Vector

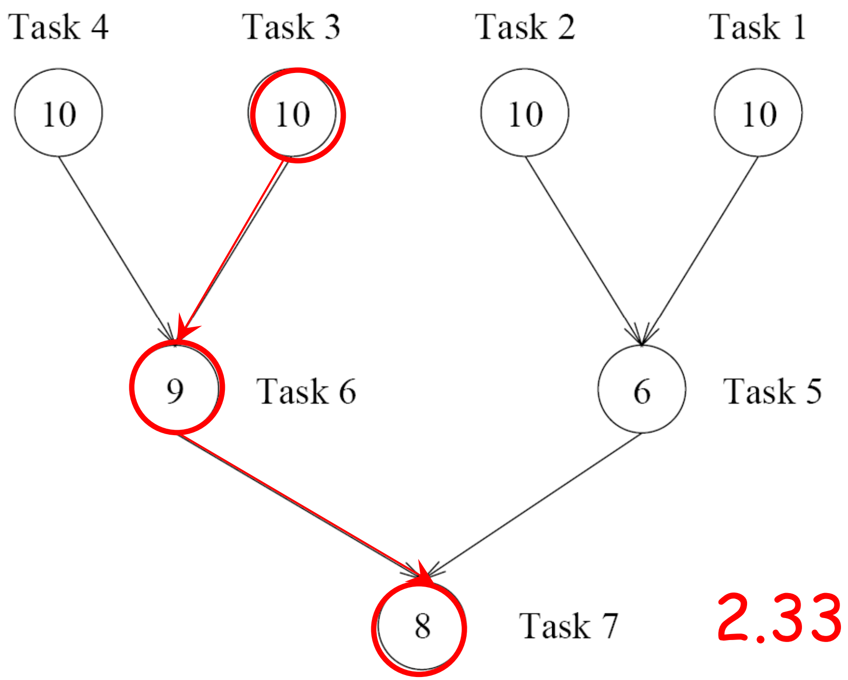


Measures

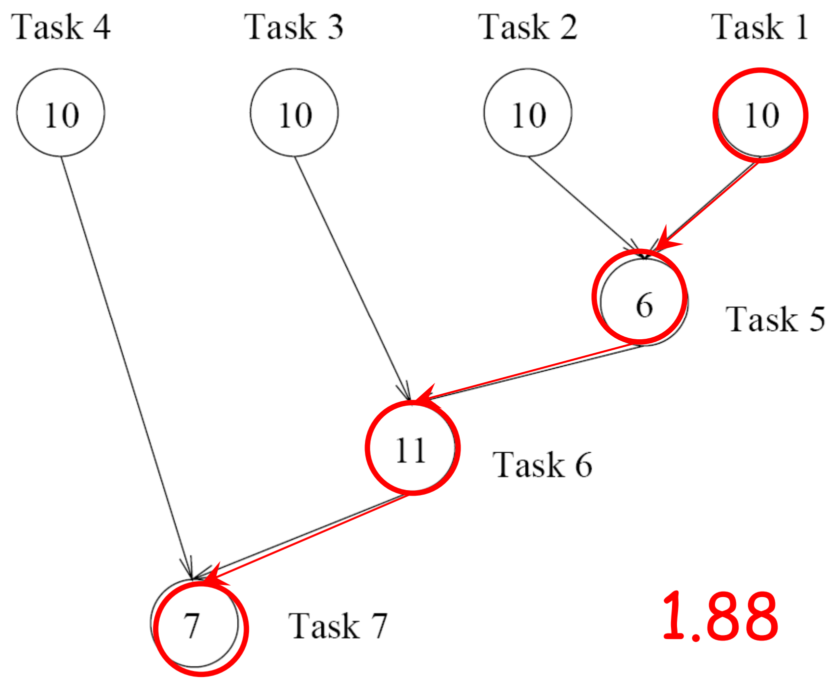
- Average degree of concurrency if we take into account varying *amount of work?*
- Critical path = longest directed path between any start & finish nodes.
- Critical path length = sum of the weights of nodes along this path.
- Average degree of concurrency = total amount of work / critical path length.

Database example

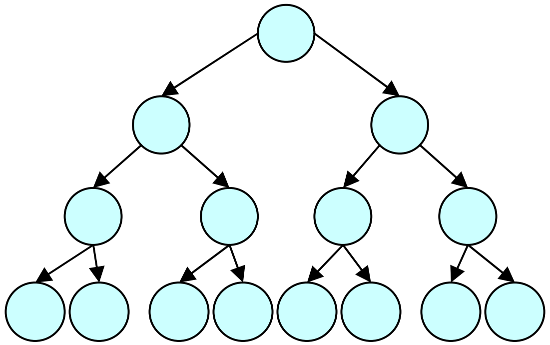
Critical path (3).
Critical path length = 27.
Av. deg. of concurrency = 63/27.



Critical path (4).
Critical path length = 34.
Av. deg. of conc. = 64/34.



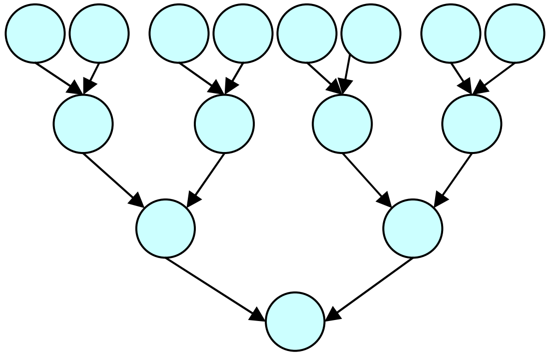
Example



Number of tasks: 15.

- Maximum degree of concurrency: 8.
- Critical path length: 4.
- Maximum possible speedup: 15/4.
- Minimum number of processes to reach this speedup: 8.
- Maximum speedup if we limit the processes to 2,4, and 8: 15/8, 3, and 15/4.

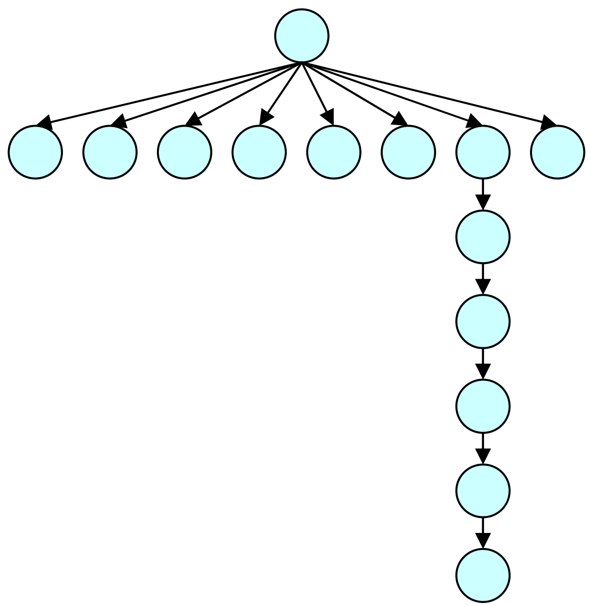
Example



Number of tasks: 15.

- Maximum degree of concurrency: 8.
- Critical path length: 4.
- Maximum possible speedup: 15/4.
- Minimum number of processes to reach this speedup: 8.
- Maximum speedup if we limit the processes to 2,4, and 8: 15/8, 3, and 15/4.

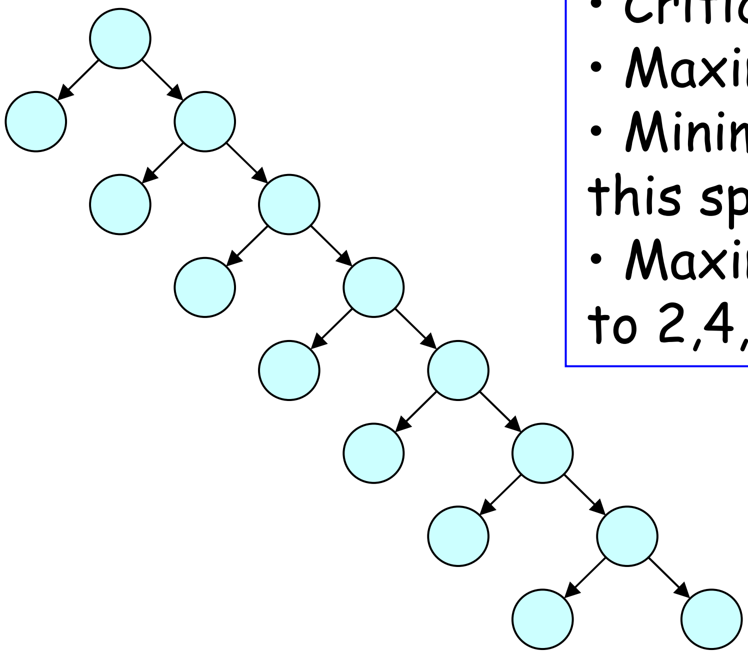
Example



- Maximum degree of concurrency: 8.
- Critical path length: 7.
- Maximum possible speedup: 14/7.
- Minimum number of processes to reach this speedup: 3.
- Maximum speedup if we limit the processes to 2,4, and 8: 14/8, 14/7, and 14/7.

Number of tasks: 14.

Example



- Maximum degree of concurrency: 2.
- Critical path length: 8.
- Maximum possible speedup: 15/8.
- Minimum number of processes to reach this speedup: 2.
- Maximum speedup if we limit the processes to 2,4, and 8: 15/8.

Number of tasks: 15.

Interaction Between Tasks

- Tasks often share data.
- Task interaction graph:
  - Nodes = tasks.
  - Edges = interaction.
  - Optional weights.
- Task dependency graph is a sub-graph of the task interaction graph.

Characteristics of Task Interactions

- One-way interactions.
  - Only one task initiates and completes the communication ***without*** interrupting the other one.
- Two-way interactions.
  - Producer – consumer model.

Processes and Mapping

- Tasks run on processors.
- Process: processing agent executing the tasks. *Not exactly like in your OS course.* Processes ~ threads here.
- Mapping = assignment of tasks to processes.
- API exposes processes and binding to processors not always controlled.
  - Scheduling of threads is not controlled.
  - What makes a good mapping?

Mapping example

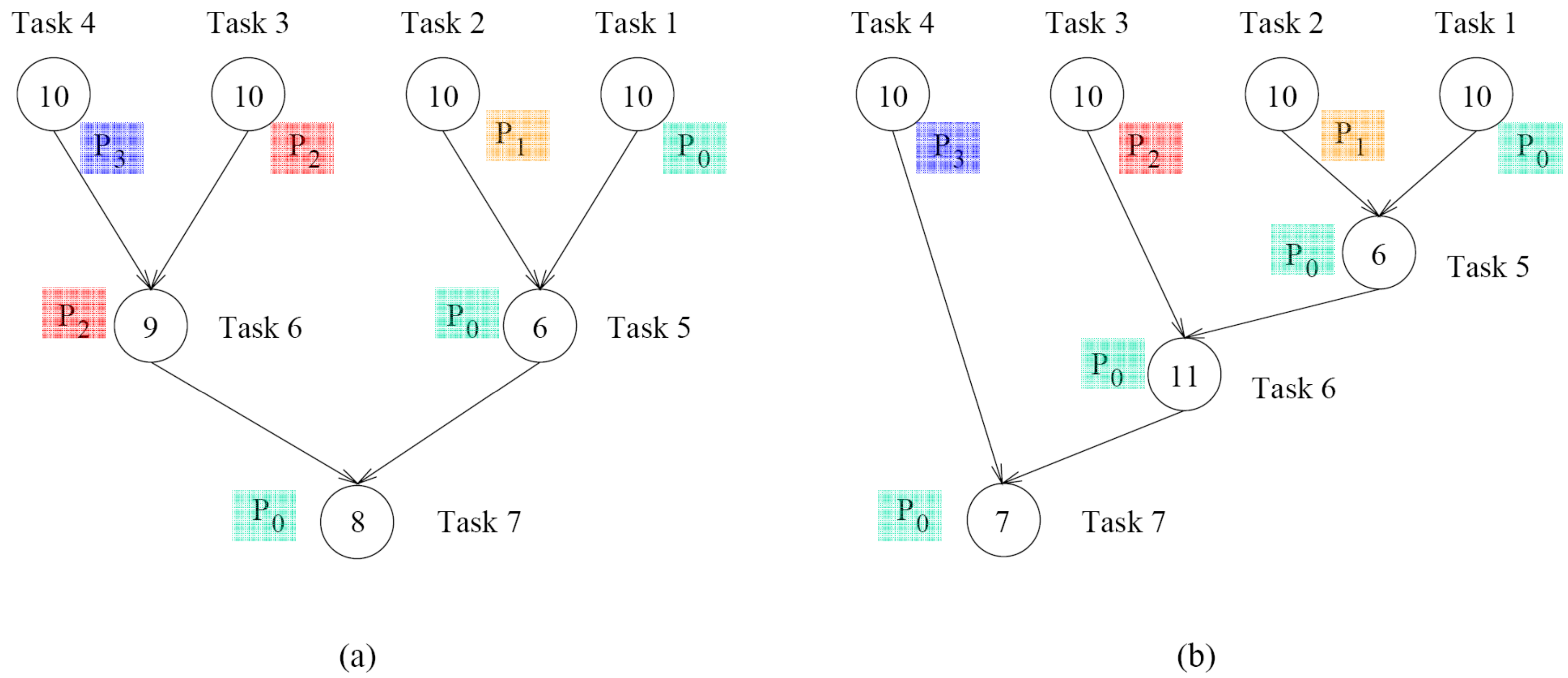


Figure 3.7 Mappings of the task graphs of Figure 3.5 onto four processes.

Processes vs. processors

- Processes = logical computing agent.
- Processor = hardware computational unit.
- In general 1-1 correspondence but this model gives better abstraction.
- Useful for hardware supporting multiple programming paradigms.

How do you decompose?

Decomposition Techniques

- Recursive decomposition.
  - Divide-and-conquer.
- Data decomposition.
  - Large data structure.
- Exploratory decomposition.
  - Search algorithms.
- Speculative decomposition.
  - Dependent choices in computations.

Recursive decomposition

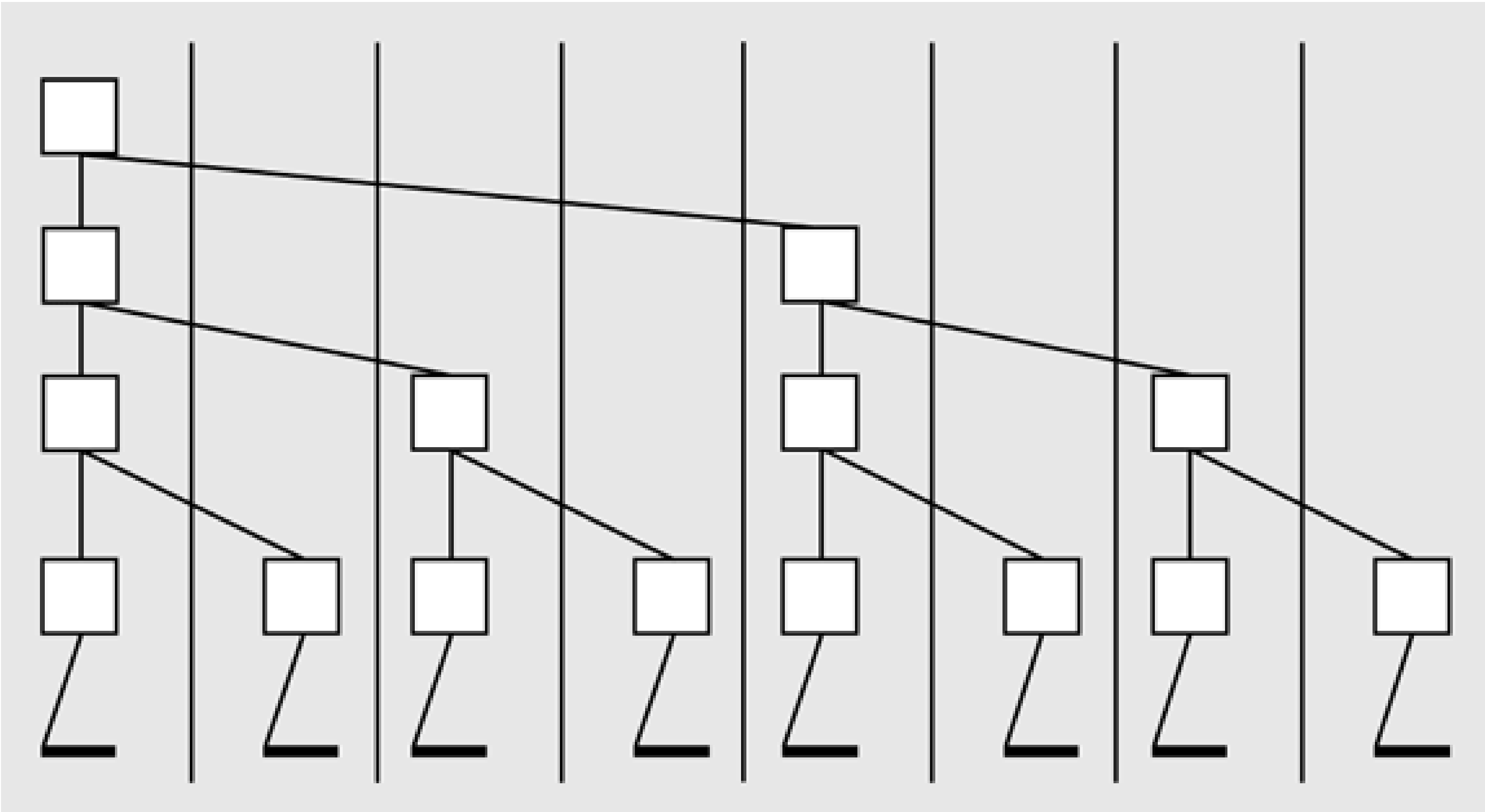
- Problem solvable by divide-and-conquer:
  - Decompose into sub-problems.
    - Do it recursively.
  - Combine the sub-solutions.
    - Do it recursively.
- Concurrency: The sub-problems are solved in parallel.

Schwartz's Algorithm
+ reduce by block

- Reduce with maximal concurrency:
  - one thread per pair (n/2)
  - combine results in a tree structure
- Schwartz:
  - one thread per n/p block of numbers
  - local sums
  - combine results in a tree structure
  - *follows recipe*

Recursive Decomposition

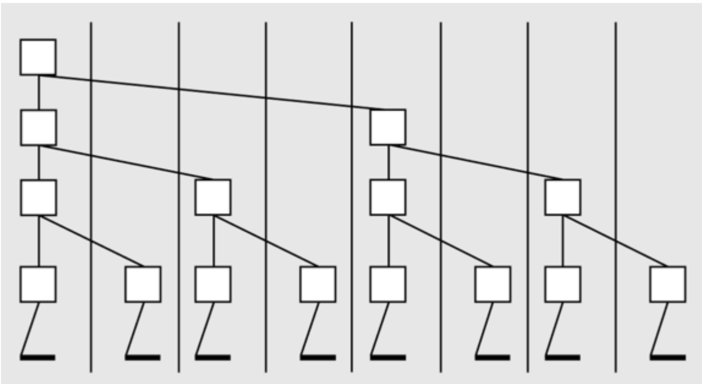
Combining Results



Peril-L

full/empty variable

intermediate value – constant amount of extra space



```
int nodeval'[P];
…
forall(index in (0..P-1))
{
  int tally;
  stride=1;
  …
  while(stride<P)
  {
    if (index%(2*stride)==0)
    {
      tally += nodeval'[index+stride];
      stride *= 2;
    }
    else
    {
      nodeval'[index]=tally;
      break;
    }
  }
}
```

Reduce & Scan Abstractions

- Reduce: combine values to a single one.
  - Almost always needed.
- Scan: prefix computation.
  - Logic that performs sequential operations and carries along intermediate results.
- Lesson: Try to use them as much as possible.
  - Abstract them as functions.
    - high-level, contain information
    - may customize implementation (e.g. BlueGene).

Small typo p130

$A=\{0,2,4\} \Rightarrow A=\{0,2,6\}$

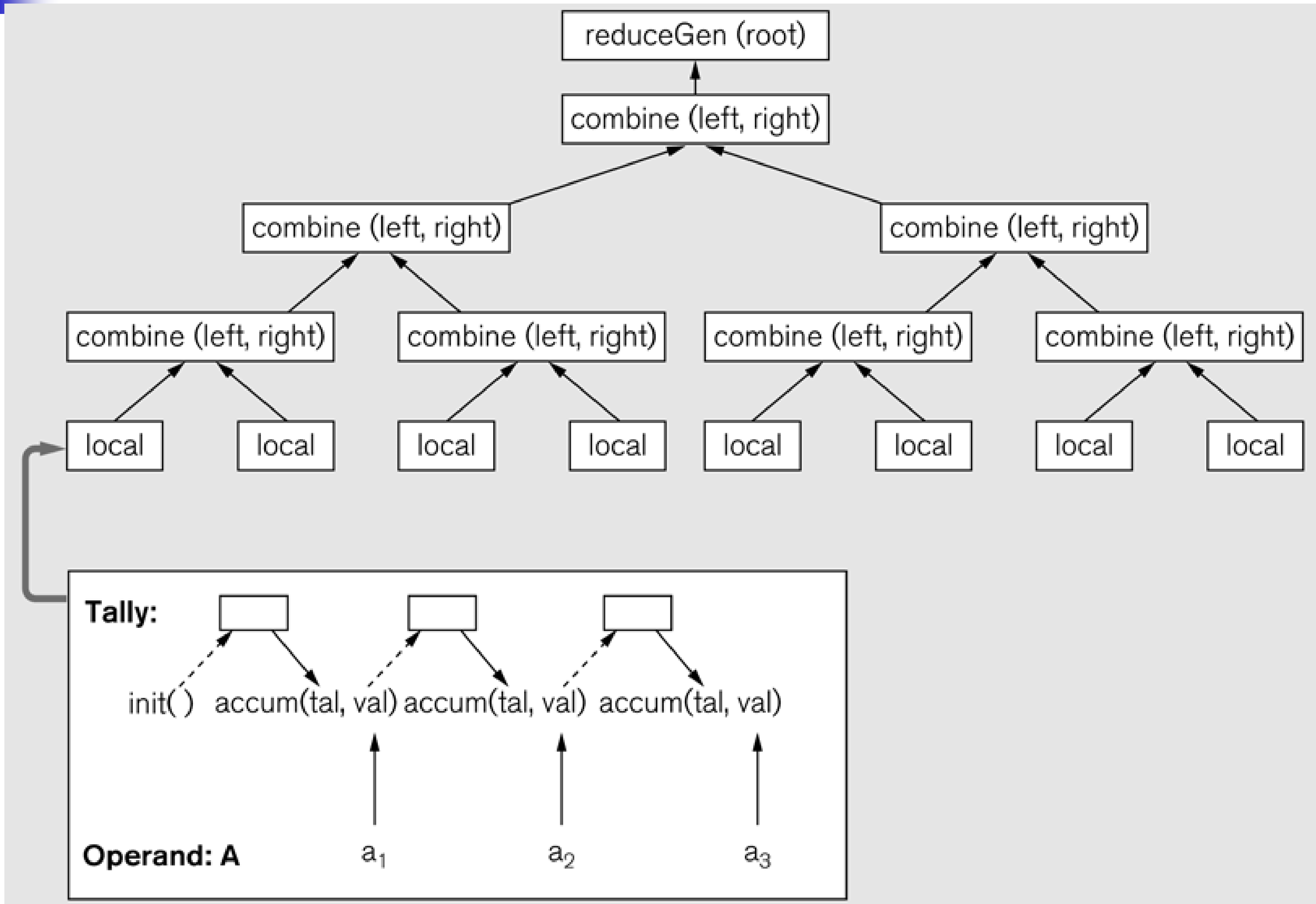
Basic Structure

- Idea:
  - Assume block allocation,
  - use Schwartz's like algorithm,
  - local variable – tally – stores intermediate results.
- Primitives:
  - `init()` – init tally
  - `accum()` – local accumulation
  - `combine()` – combines tally results
  - `x-gen()` – final answer

Example: + reduce

- `init()`: tally=0
- `accum(tally,val)`: tally+=value
- `combine(left,right)`: left+right sent to parent
- `reduce-gen(root)`: return

Reduce Basic Structure



General Reduce in Peril-L

```
int nodeval'[P];                                      Global full/empty variables
int result;
forall(index in(0..P-1))
{
  int myData[size]=localize(dataarray[]);             Local portion of global data values
  int tally;
  int stride=1;
  tally=init ()                                       Initialize tally
  for(i=0; i<size; i++)
  {
    tally=accum (tally, myData[i]);                   Local accumulation
  }
  nodeval'[index]=tally;                              Send initially to parent
  while(stride < P)                                   Begin logic for tree
  {
    if(index%(2*stride)==0)
    {                                                 Combine values globally
      nodeval'[index]=combine(nodeval'[index],
                              nodeval'[index+stride]);
      stride=2*stride;
    }
    else
    {
      break;
    }
  }
  if(index==0)
  {
    result=reduceGen (nodeval'[0]);                   Generate reduced value
  }
}
```

2^{nd} Min in Peril-L

```
struct tally
{
  float smallest1;                          Smallest element
  float smallest2;                          Second smallest
};

tally init()                                Initialize tally
{
  tally t;
  t.smallest1=MAX_FLOAT;
  t.smallest2=MAX_FLOAT;
  return t;
}

tally accum(tally t, float elem)            Local accumulation
{
  if(t.small1>elem)                         Is this a new smallest?
  {
    t.smallest2=t.smallest1;
    t.smallest1=elem;
  }
  else
  {
    if(t.smallest2>elem)                    Is it a new second smallest?
    {
      t.smallest2=elem;
    }
    return t;
  }
```

2nd Min in Peril-L

```
    if(t.smallest2>elem)
    {
      t.smallest2=elem;
    }
    return t;
  }
}

tally combine(tally left, tally right)
{
  tally t;
  t=accum(left, right.smallest1);
  t=accum(t, right.smallest2);
  return t;
}

float reduceGen(tally t)
{
  return t.smallest2;
}
```

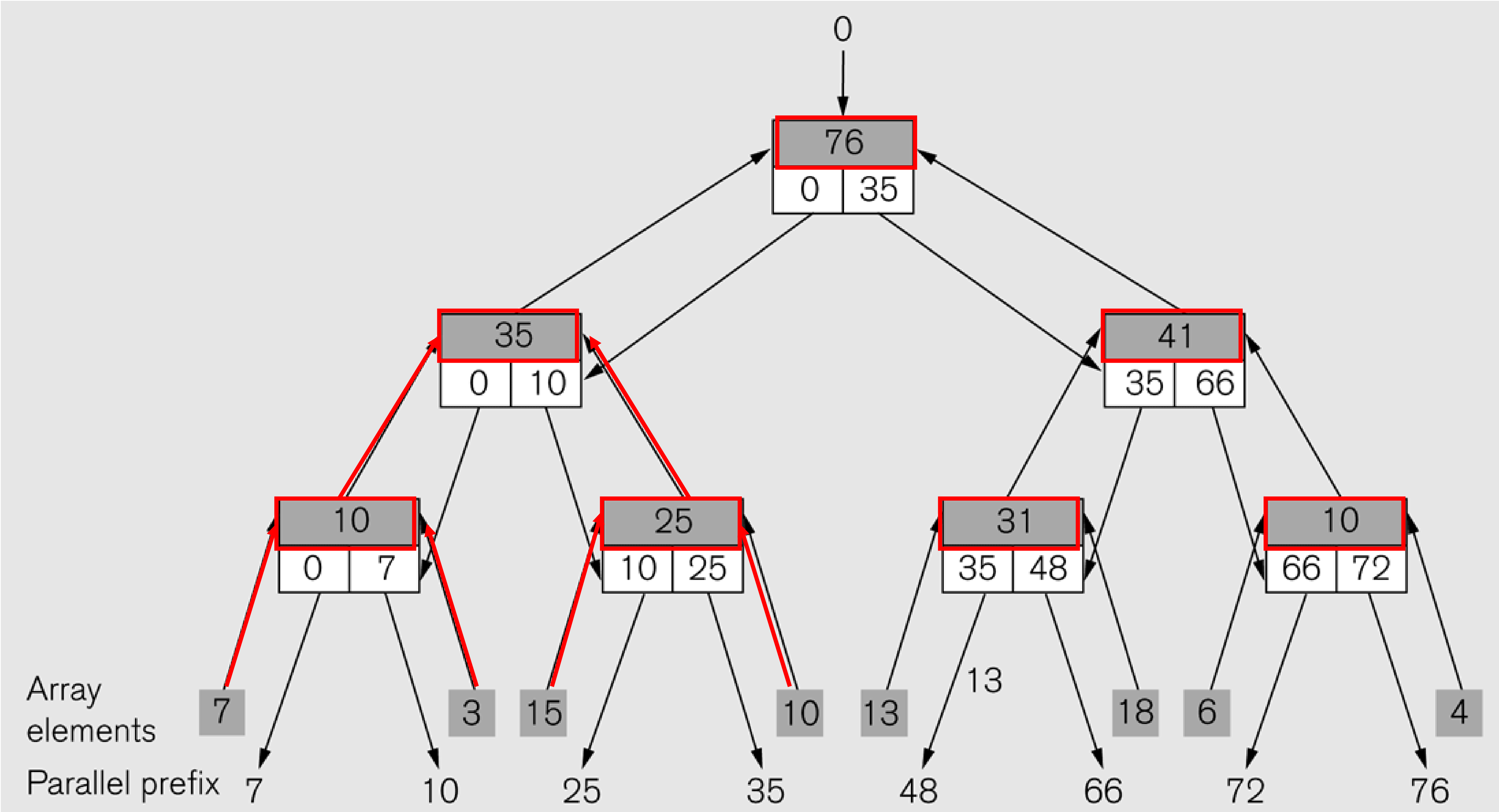
Is it a new second smallest?

Combine into "left" by accumulating right values

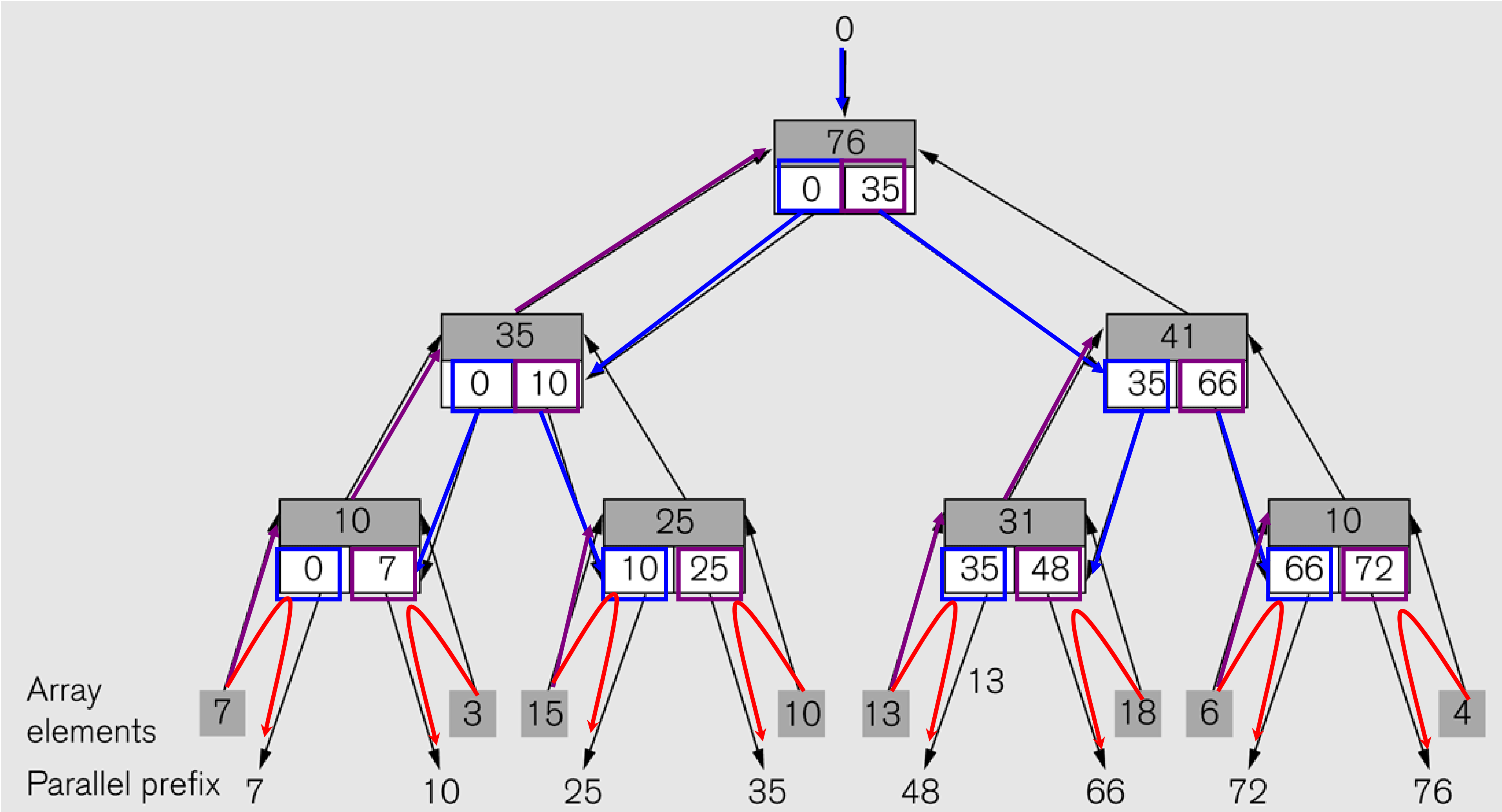
General Scan

- Difference with reduce:
  - need to pass intermediate results too.
  - Propagate tally down the tree:
    value from a parent = tally from the left sub-tree of the parent.
  - root has no parent – fix that
- Idea:
  - up-sweep with reduce
  - down-sweep to propagate tallys

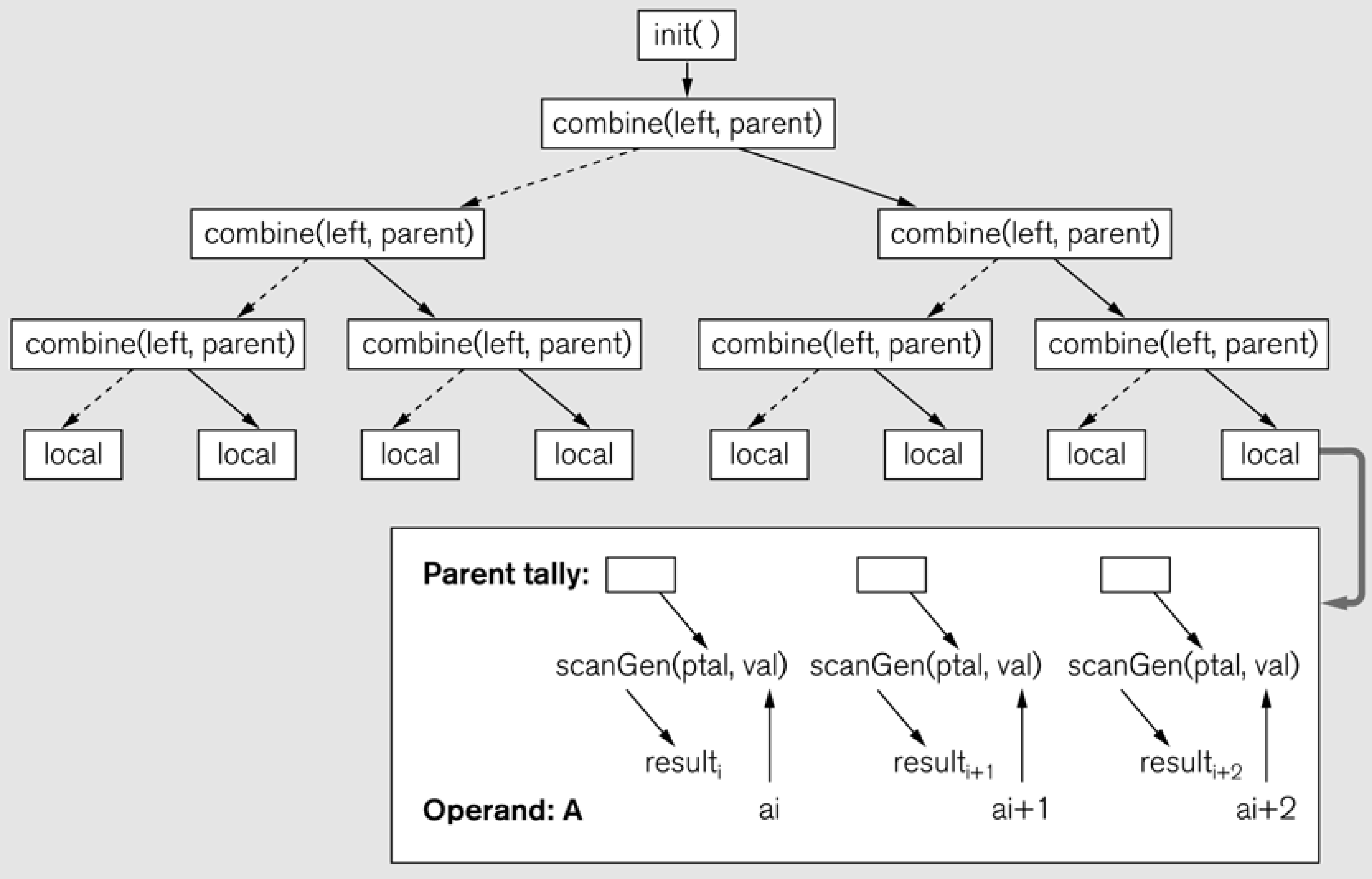
Prefix Sum - sum



Prefix Sum - prefix



Down-sweep



init( )
combine(left, parent)
combine(left, parent)
combine(left, parent)
combine(left, parent)
combine(left, parent)
combine(left, parent)
combine(left, parent)
local
local
local
local
local
local
local
local
Parent tally:
scanGen(ptal, val)
scanGen(ptal, val)
scanGen(ptal, val)
result_i
result_i+1
result_i+2
Operand: A
ai
ai+1
ai+2

Scan in Peril-L

```
int nodeval'[P];                                Global full/empty memory
int ltally[P];                                  Store left operand of combine
forall(index in(0..P-1))
{
  int myData[size]=localize(operandArray[]);    Local data values
  int tally;                                    Tally
  int ptally;                                   Tally from parent
  int stride=1;
  tally=init ();                                Initialize
  for(i=0; i<size; i++)
  {
    tally=accum (tally, myData[i]);             Accumulate
  }
  nodeval'[index]=tally;                        Send initially to parent
  while(stride<P)                               Begin logic for tree
```

```
    {
      if(index%(2*stride)==0)
      {                                                         Combine
        ltally[index+stride]=nodeval'[index];
        nodeval'[index]=combine (ltally[index+stride],
                                 nodeval'[index+stride]);
        stride=2*stride;
      }
      else
      {
        break;
      }
    }
    stride=P/2;
    if(index==0)
    {
      ptally=nodeval'[0];                                       Clear existing up sweep value
      nodeval'[0]=init ();                                      Set init() as parent input
    }
    while(stride>1)                                             Begin logic for tree descent
    {
      ptally=nodeval'[index];                                   Grab parent value
      nodeval'[index]=ptally;                                   Send it down to left
      nodeval'[index+stride]=                                   Send parent + left child right
        combine (ptally, ltally[index+stride]);
      stride=stride/2;                                          Go down to next level
    }
    for(i=0; i<size; i++)
    {
      myResult[i]=scanGen (ptally, myData[i]);                  Generate Scan
    }
}
```

Lesson

- Structure the algorithm with reduce & scan.
- Use efficient implementations of reduce & scan.

Data Decomposition

- 2 steps:
  - Partition the data.
  - Induce partition into tasks.
- How to partition data?
- Partition output data:
  - Independent “sub-outputs”.
- Partition input data:
  - Local computations, followed by combination.
- 1-D, 2-D, 3-D block decomposition.

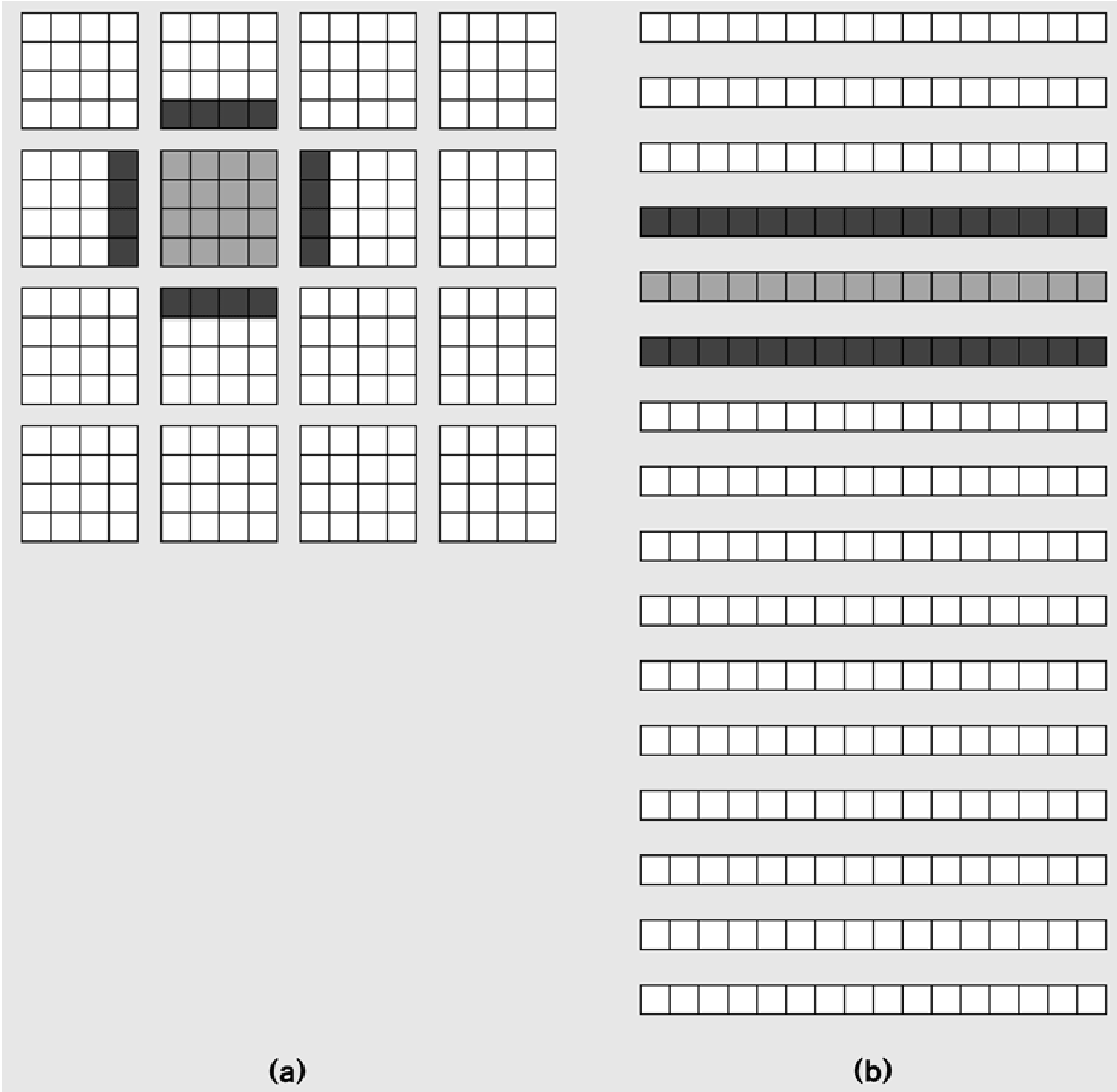
Static Allocation of Work to Processes

- # of threads fixed but unknown.
  - Allocate data to threads.
  - Owner compute rule.
- Block allocation – maximize locality
  - 1-D or 2-D depending on the communication pattern – minimize communication
    surface area to volume in favour of 2-D

Owner-Compute Rule

- Process assigned to some data
  - is responsible for all computations associated with it.
- Input data decomposition:
  - All computations done on the (partitioned) input data are done by the process.
- Output data decomposition:
  - All computations for the (partitioned) output data are done by the process.

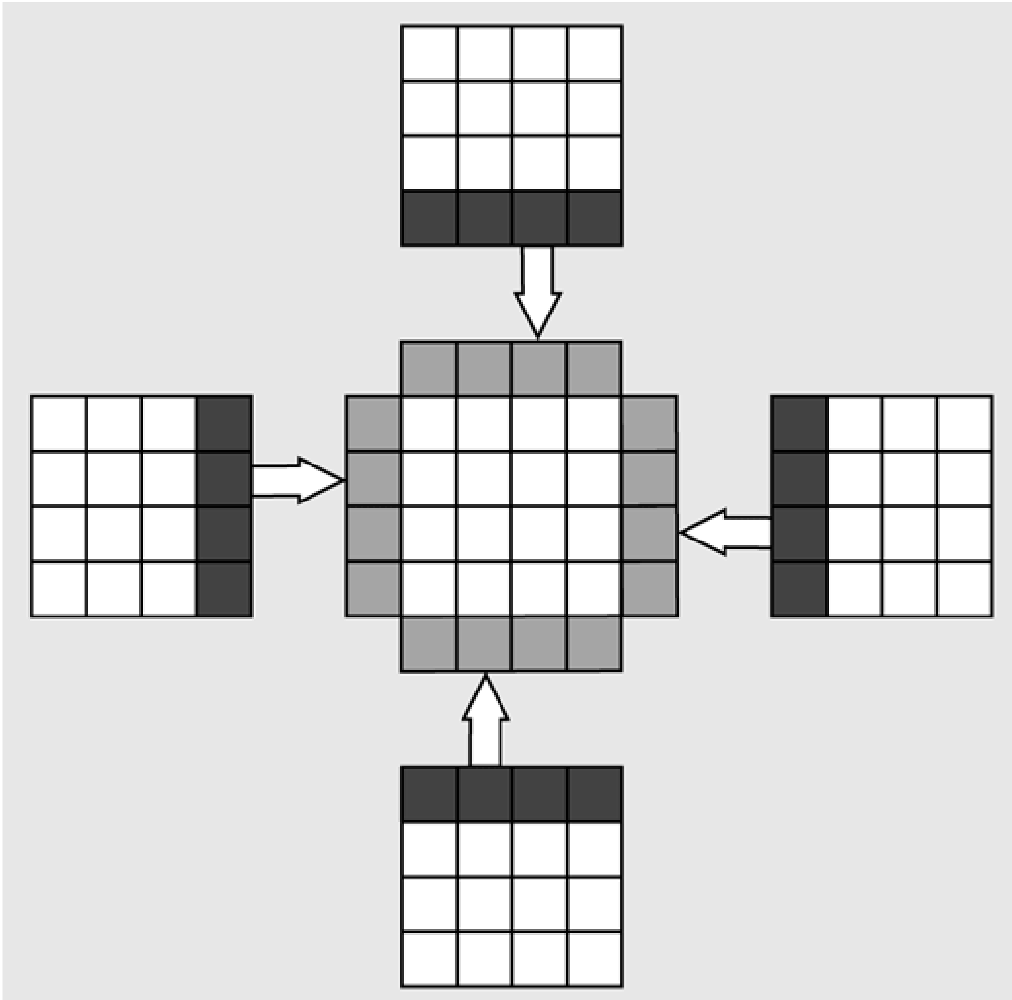
1-D & 2-D Block Allocations



Overlap Regions

- Obtain data from neighbors.
- Compute locally.
- Avoid false sharing.
- Use local matrix
  - no special edge cases
  - uniform indices
  - batch communication cheaper

Overlap Regions

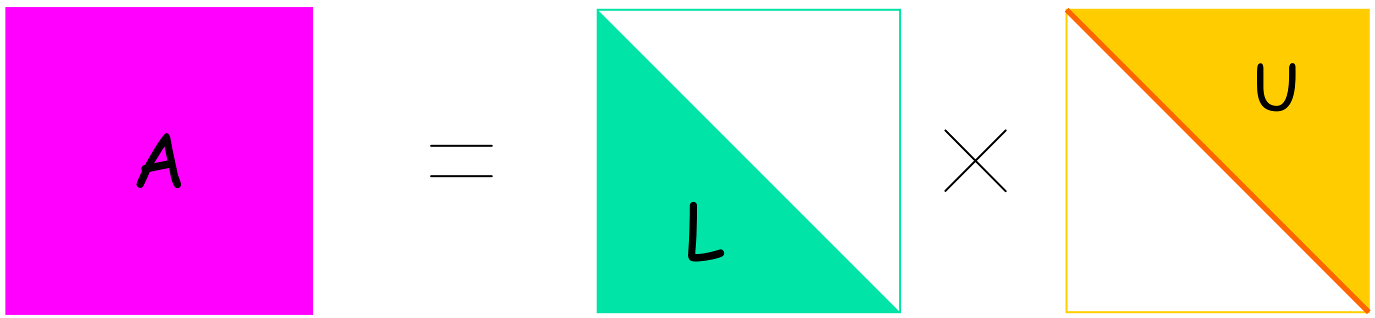


Cyclic & Block Cyclic

- Cyclic = round-Robin. Idea:
  - Partition an array into many *more blocks than available processes*.
  - Assign partitions (tasks) to processes in a round-robin manner.
  - $\rightarrow$ each process gets several non adjacent blocks.
- Useful when computations are not proportional to the data.
  - ex: assignment 2
  - otherwise poor load balance
- Good: load balance.
- Bad: more communication, break large blocks.

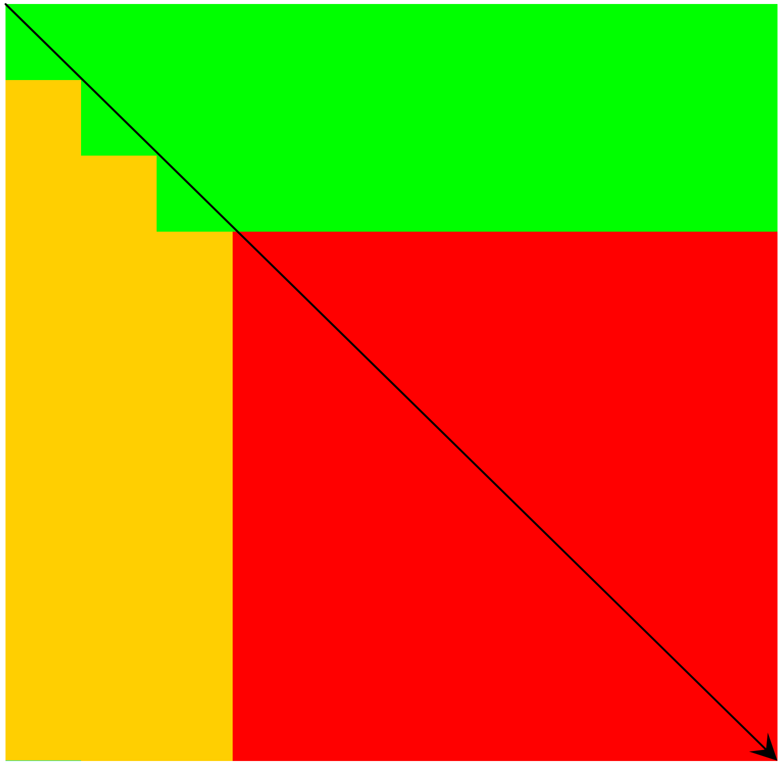
Example: LU factorization

- Non singular square matrix A (invertible).
- A = L*U.
- Useful for solving linear equations.



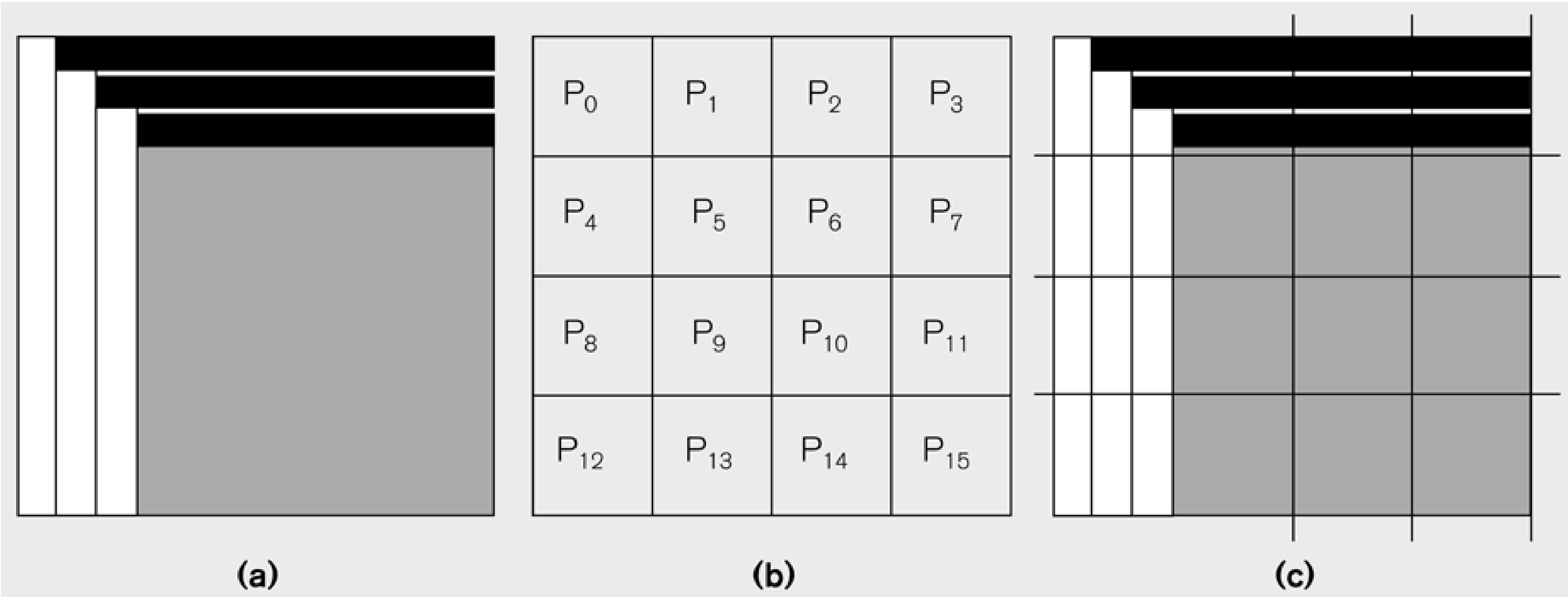
LU factorization

In practice we work on *A*.



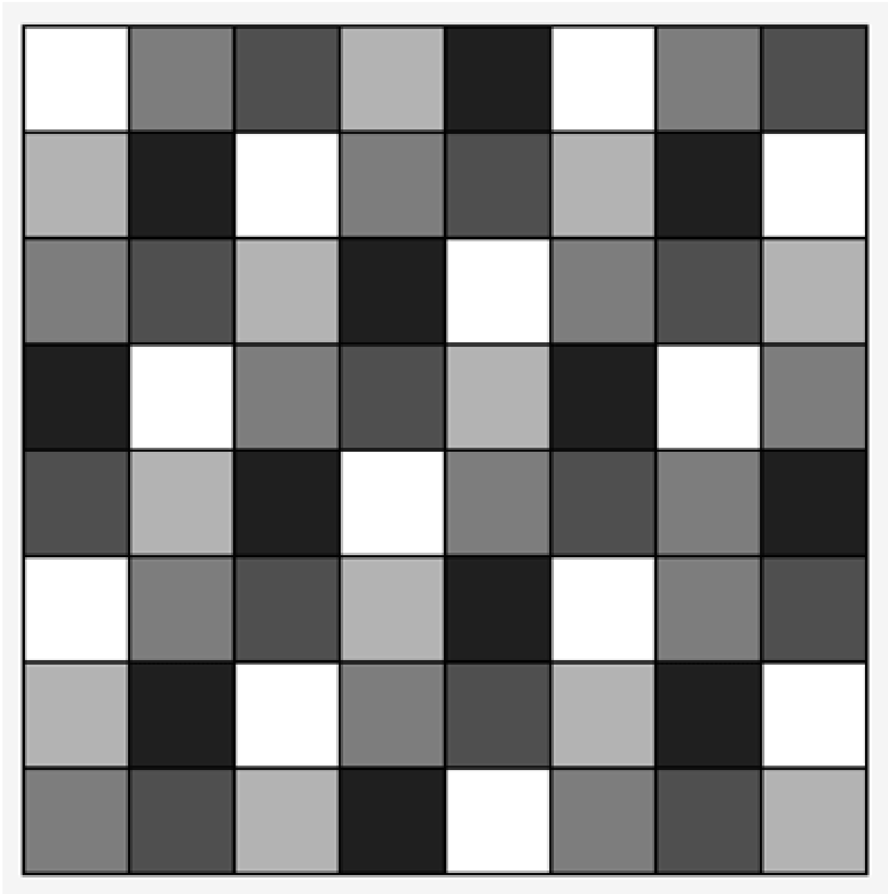
N steps

Load Imbalance: LU-Decomposition

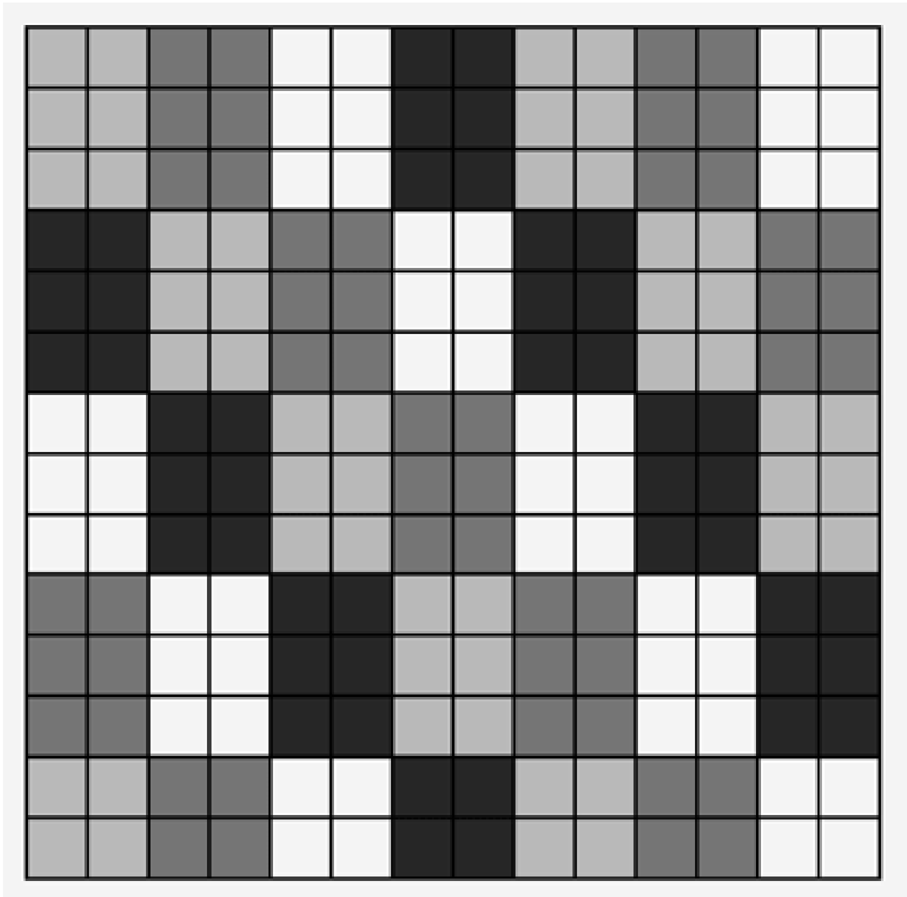


Matrix inversion – similar.

8x8 Array on 5 Processes



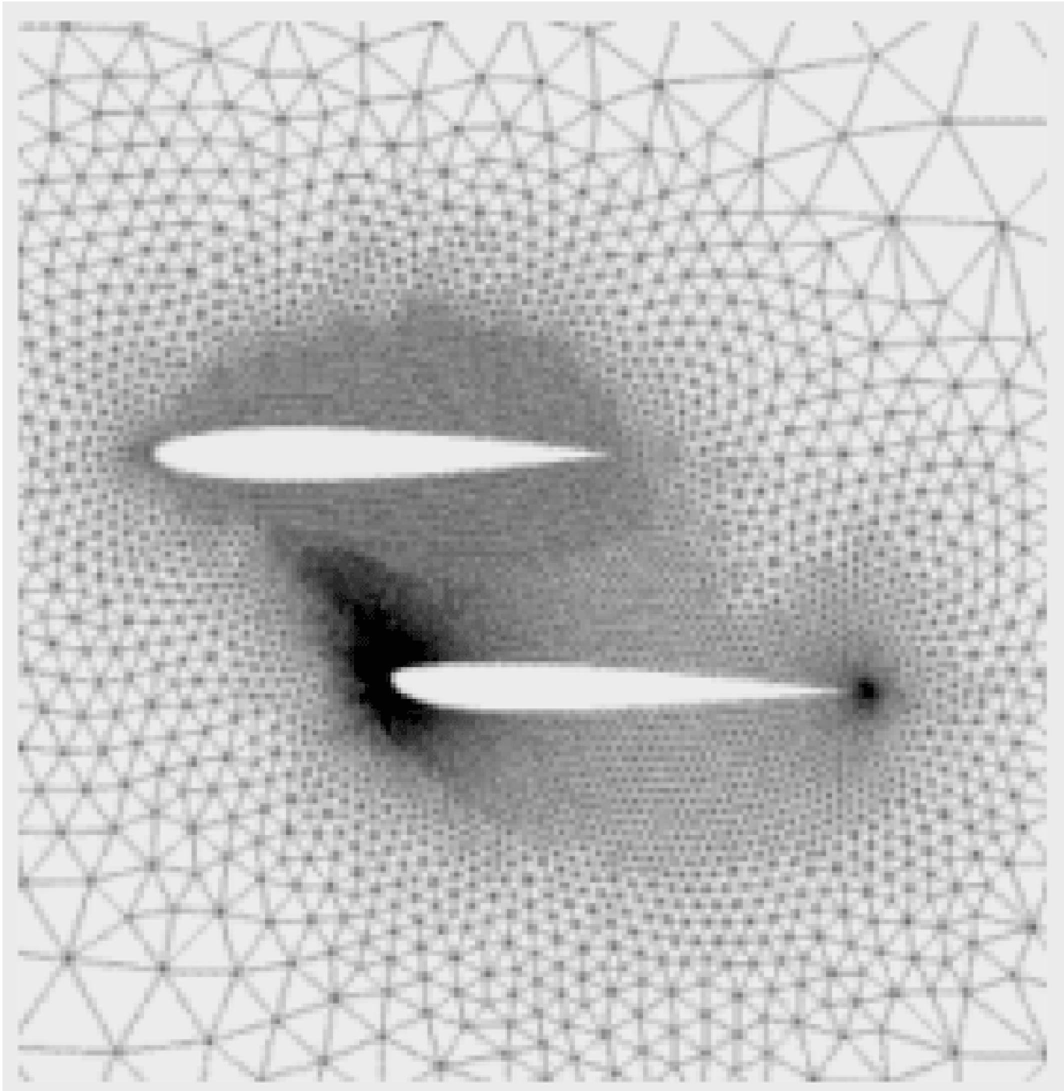
Block Cyclic Distribution



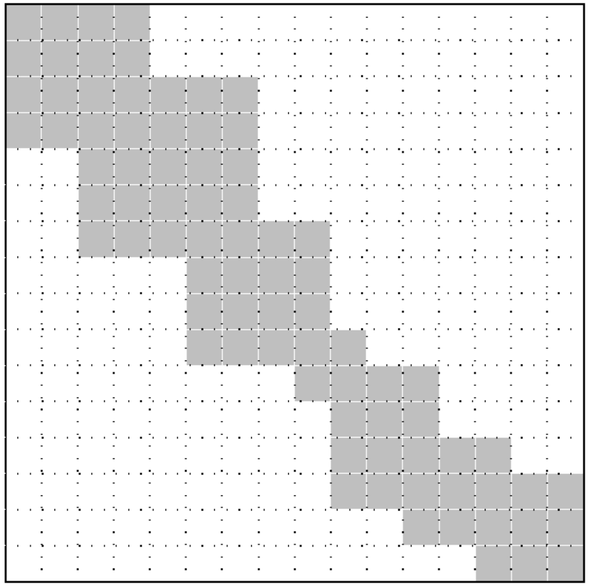
Julia Sets

Assignment: Mandelbrot

Irregular Allocations



Randomized Distributions



(a)

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

Irregular distribution with regular mapping!
Not good.

1-D Randomized Distribution

$$V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$$

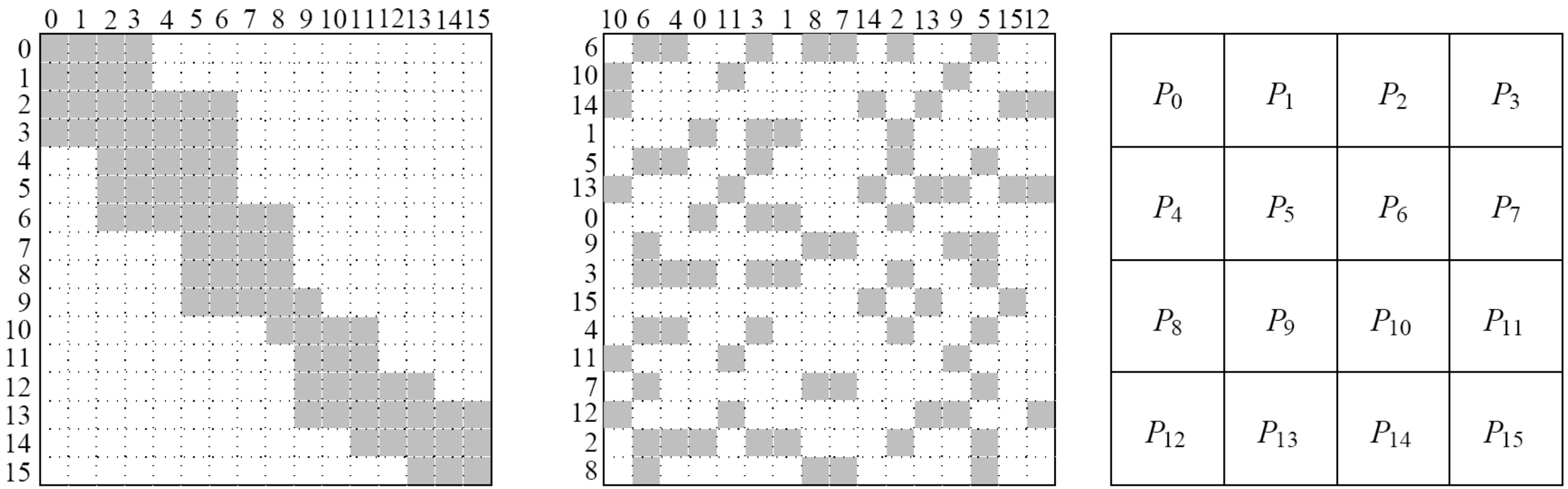
Permutation

$$\text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$$

$$\text{mapping} = \underbrace{8\ \ 2\ \ 6}_{P_0}\ \ \underbrace{0\ \ 3\ \ 7}_{P_1}\ \ \underbrace{11\ \ 1\ \ 9}_{P_2}\ \ \underbrace{5\ \ 4\ \ 10}_{P_3}$$

Figure 3.32 A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., $\alpha = 3$).

2-D Randomized Distribution



(a) (b) (c)

2-D block random distribution.

Block mapping.

Irregular Allocations

- Same idea as overlap regions:
  - get data local – *inspector*
  - local computations – *executor*

Dynamic Allocations

- Keys
  - dynamic load balancing
  - dynamic interactions
  - choose right granularity of tasks
- Work queues
  - e.g. producer/consumer
  - centralized schemes with master/slave
  - different queue orderings
  - multiple queues – issues with load balancing

Graph Partitioning

- For sparse data structures and data dependent interaction patterns.
  - Numerical simulations. Discretize the problem and represent it as a mesh.
- Sparse matrix: assign equal number of nodes to processes & minimize interaction.
- Example: simulation of dispersion of a water contaminant in Lake Superior.

Discretization

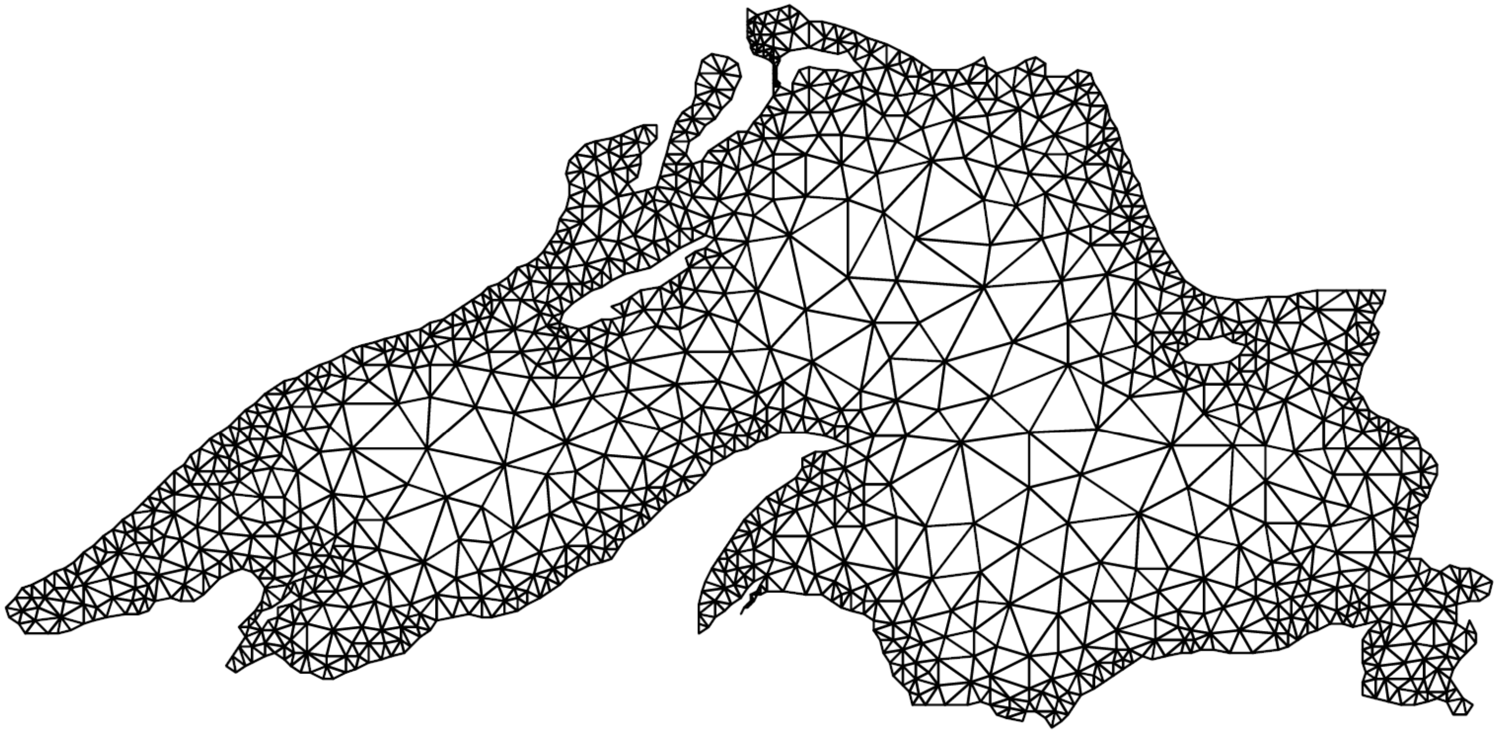
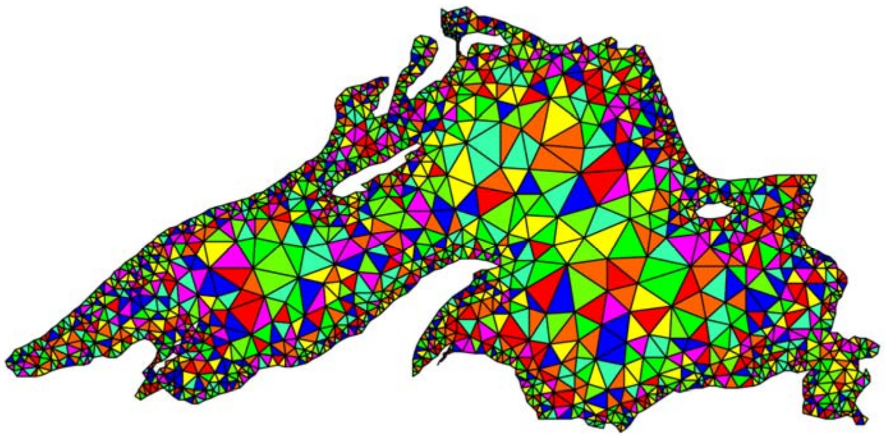
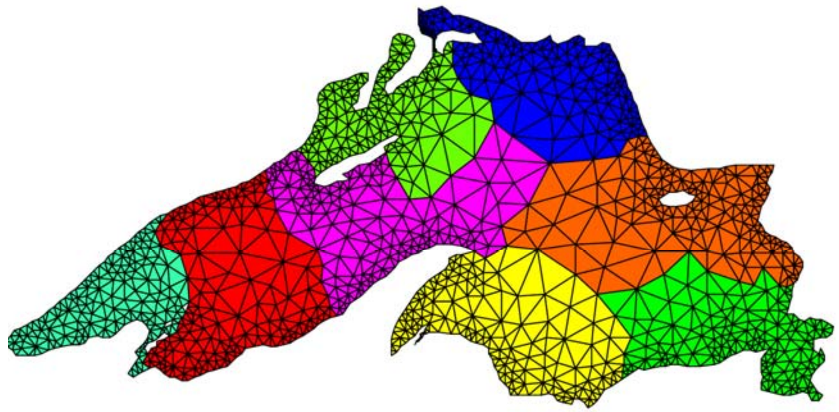


Figure 3.34 A mesh used to model Lake Superior.

Partitioning Lake Superior



Random partitioning.

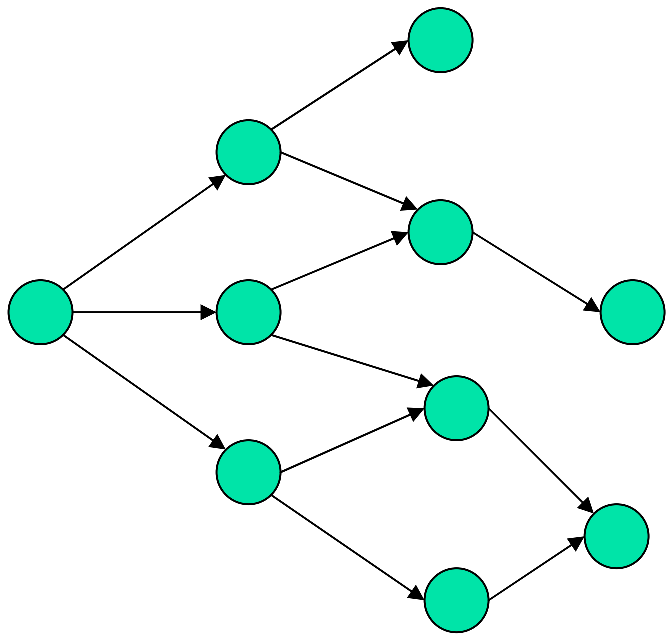


Partitioning with minimum edge cut.

Finding an exact optimal partitioning is an NP-complete problem.

Exploratory Decomposition - Trees

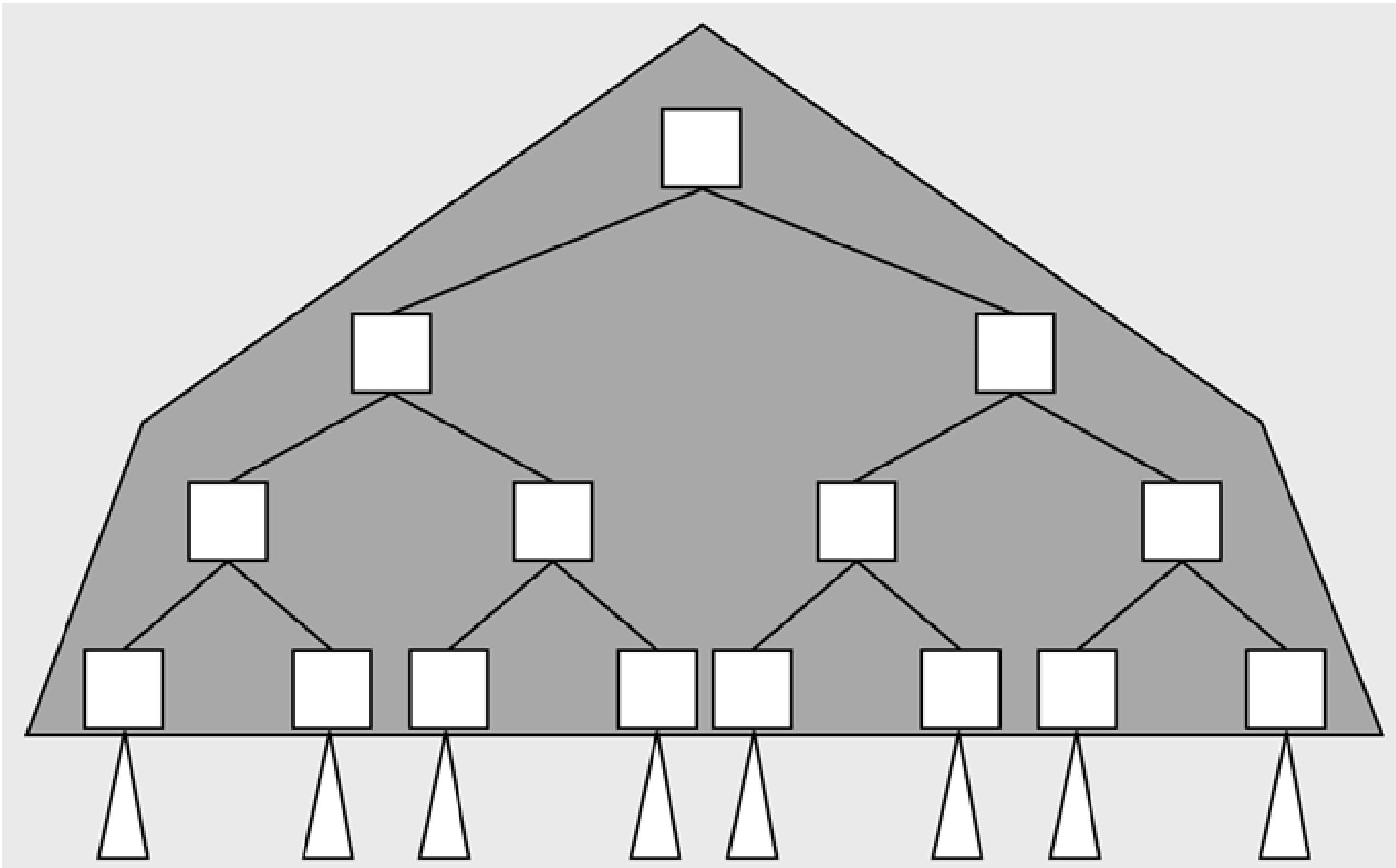
Exploration of states.



Trees

- Useful data-structures.
- Usually constructed with pointers.
- Challenges for
  - communication
  - load balance on irregular trees
- If little communication among sub-trees:
  - Allocate sub-trees to processes, copy the “cap”.
  - All processes know the structure.

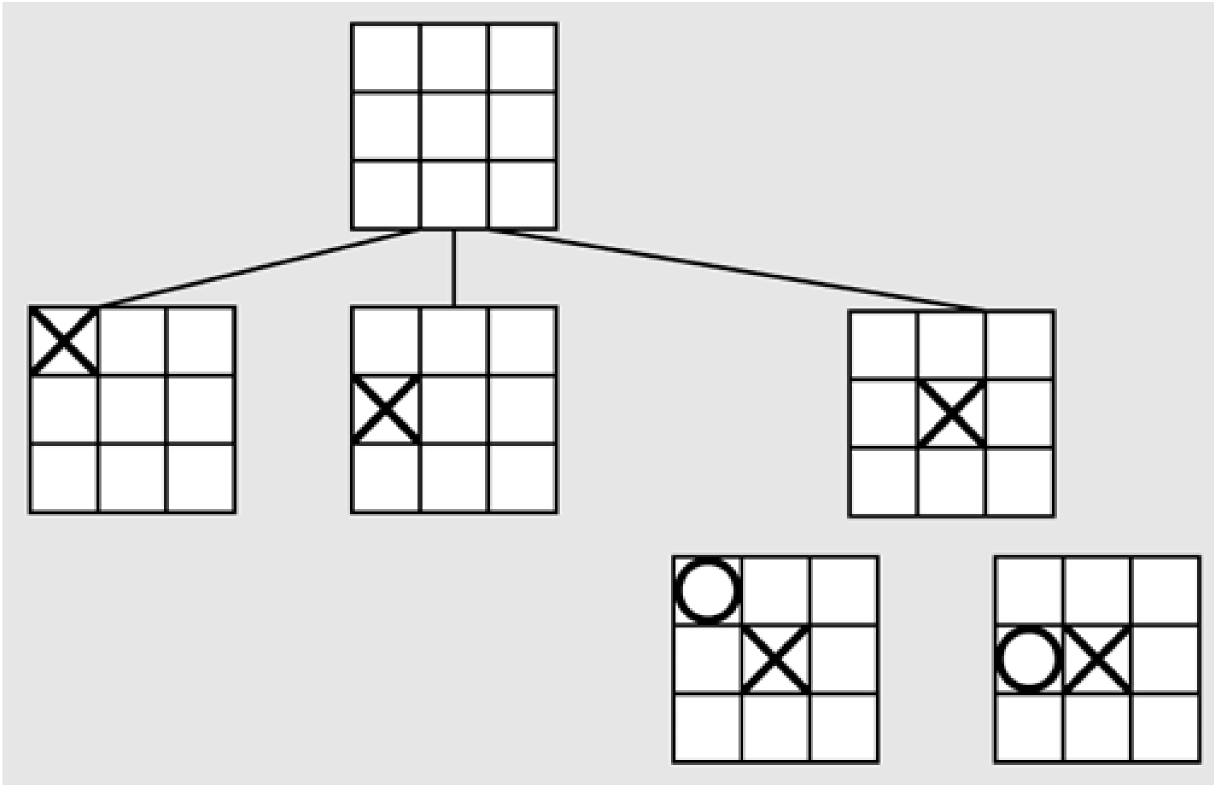
Cap Copy & Subtree Allocation



Dynamic Allocations

- Dynamic & unpredictable trees.
  - Search with different algorithms.
  - Work queues useful.
  - Pruning involves communication.
  - Termination may be an issue!

Application to Game Search



Performance Anomalies

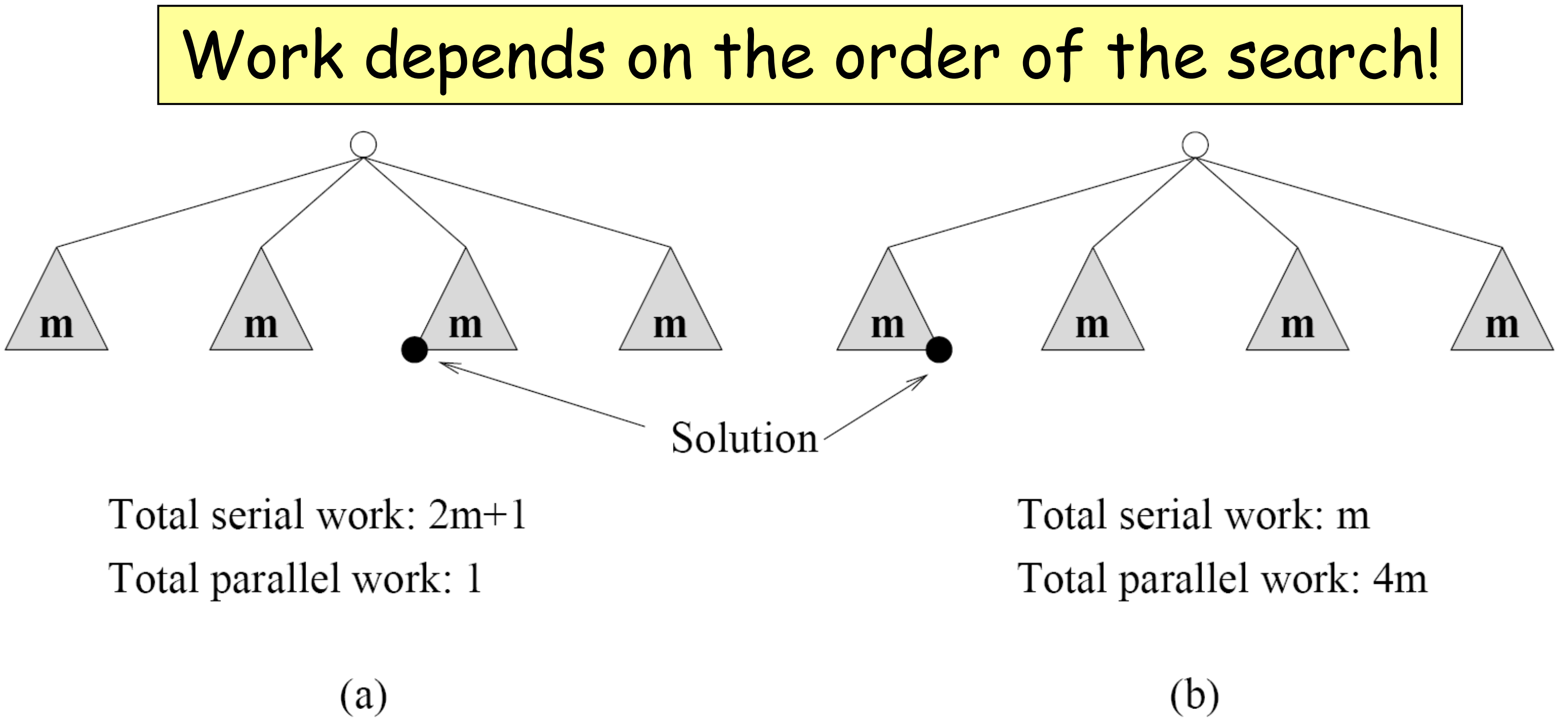


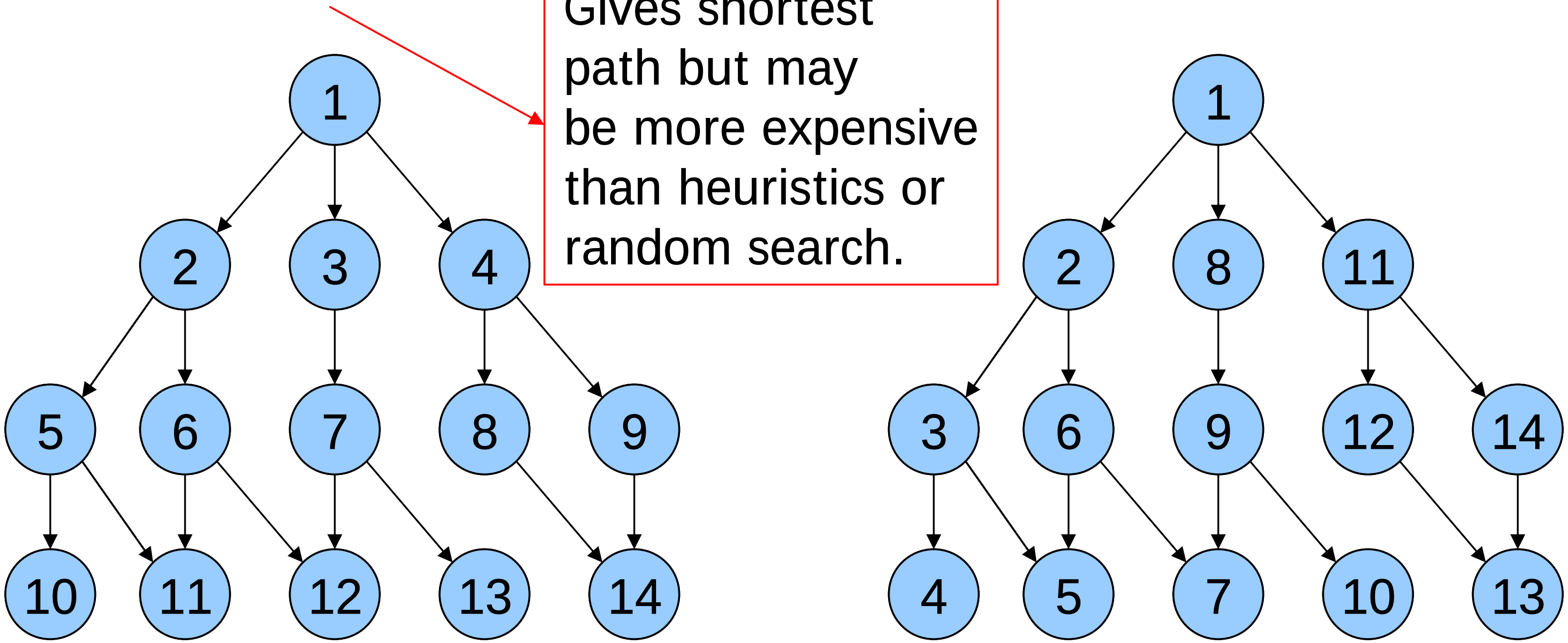
Figure 3.19 An illustration of anomalous speedups resulting from exploratory decomposition.

Search Orderings Issues

Breadth-first-search (BFS)

Depth-first-search (DFS)

Gives shortest path but may be more expensive than heuristics or random search.



Speculative Decomposition

- Dependencies between tasks are not known a-priori.
  - How to identify independent tasks?
  - Conservative approach: identify tasks that are *guaranteed* to be independent.
  - Optimistic approach: schedule tasks even if we are not sure – may roll-back later.