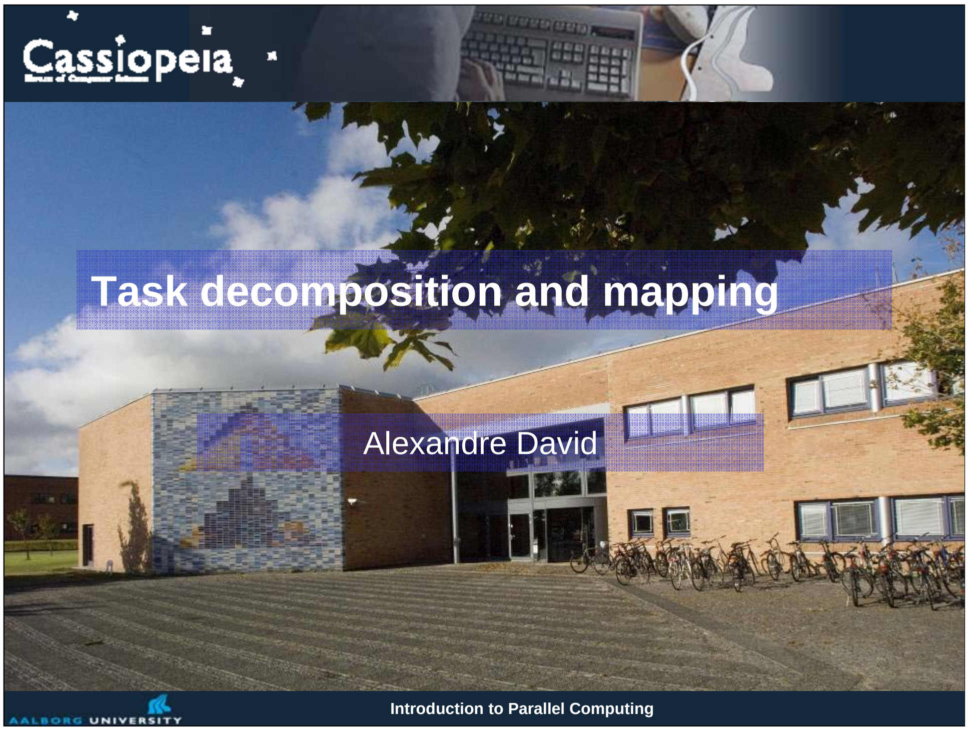

Cassiopeia
Task decomposition and mapping
Alexandre David
AALBORG UNIVERSITY
Introduction to Parallel Computing

Overview

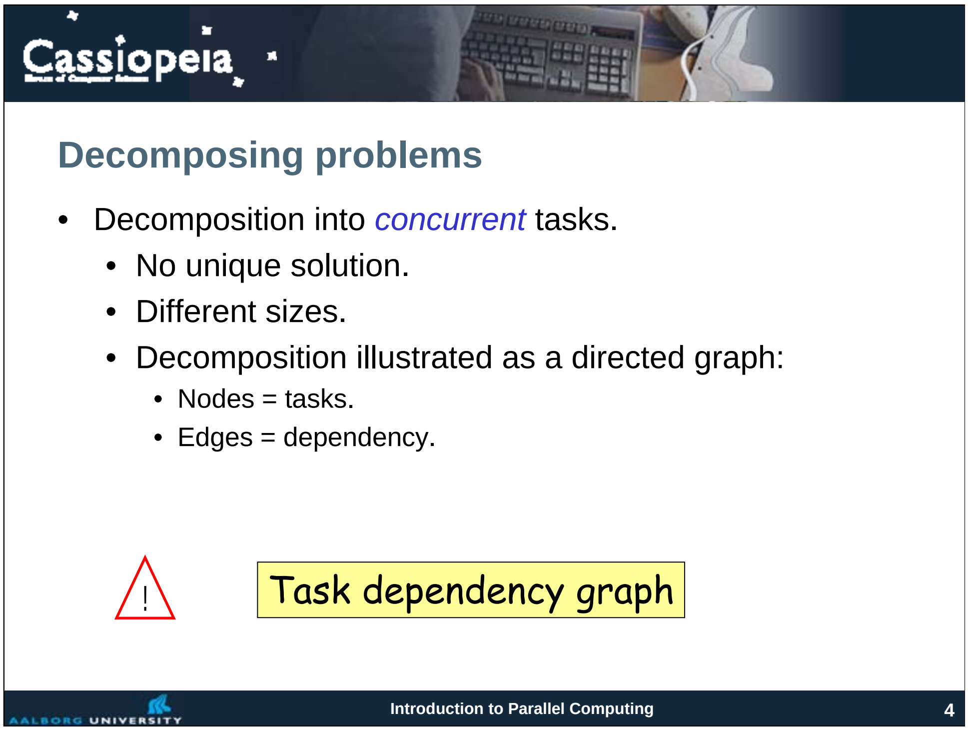
- Introduction to parallel algorithms
- Decomposition techniques
- Task interactions
- Load balancing

Cassiopeia

Introduction

- Parallel algorithms have the added dimension of *concurrency*.
- Typical tasks:
  - Identify concurrent works.
  - Map them to processors.
  - Distribute inputs, outputs, and other data.
  - Manage shared resources.
  - Synchronize the processors.

There are other courses specifically on concurrency. We won’t treat the problems proper to concurrency such as deadlocks, livelocks, theory on semaphores and synchronization. However, we will use them, and when needed, apply techniques to avoid problems like deadlocks.



Many solutions are often possible but few will yield good performance and be scalable. We have to consider the computational and storage resources needed to solve the problems.

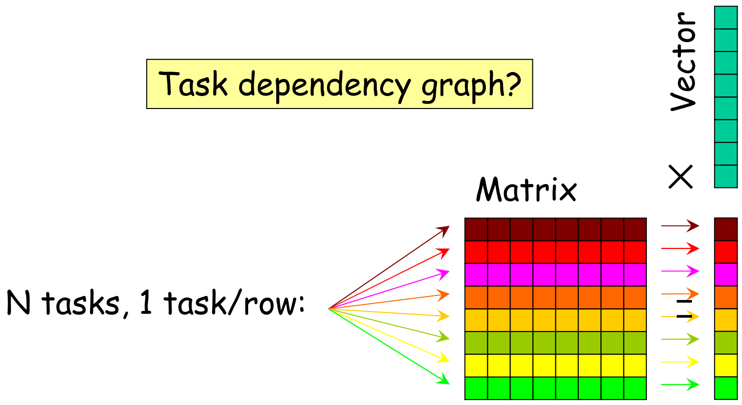
Size of the tasks in the sense of the amount of work to do. Can be more, less, or unknown. Unknown in the case of a search algorithm is common.

Dependency: All the results from incoming edges are required for the tasks at the current node.

We will not consider tools for automatic decomposition. They work fairly well only for highly structured programs or options of programs.

Example: Matrix * Vector

Task dependency graph?



Cassiopeia

Example: database query processing

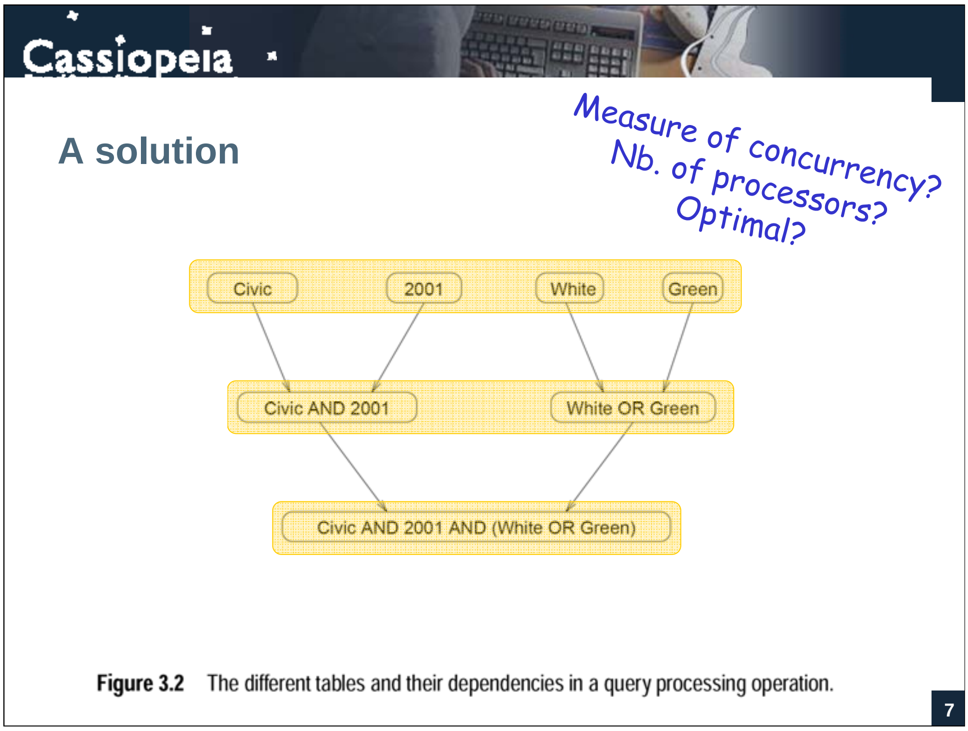
MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

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

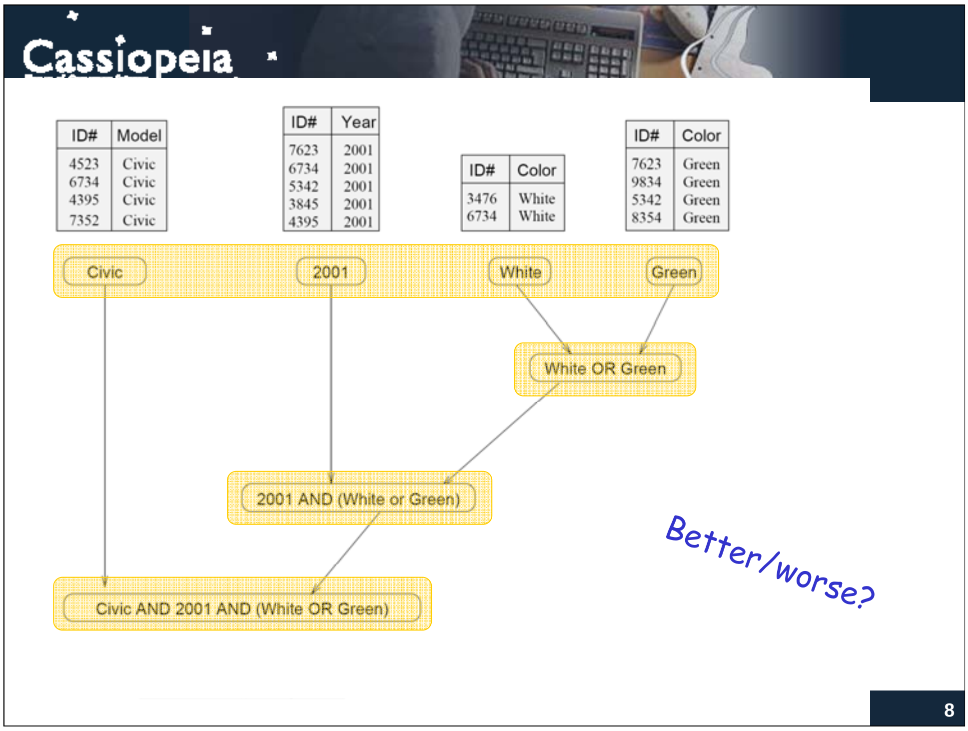
Table 3.1 A database storing information about used vehicles.

6

The question is: How to decompose this into concurrent tasks? Different tasks may generate intermediate results that will be used by other tasks.



How much concurrency do we have here? How many processors to use? Is it optimal?



Is it better or worse? Why?

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.

•Previous matrix*vector fine-grained.

•Database example coarse grained.

Degree of concurrency: Number of tasks that can be executed in parallel.

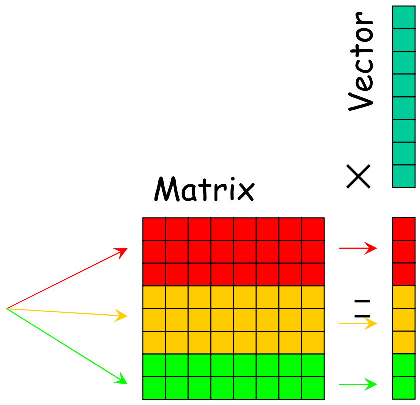
Average degree of concurrency is a more useful measure.

Assume that the tasks in the previous database examples have the same granularity. What's their average degrees of concurrency? 7/3=2.33 and 7/4=1.75.

Common sense: Increasing the granularity of decomposition and utilizing the resulting concurrency to perform more tasks in parallel increases performance. However, there is a limit to granularity due to the nature of the problem itself.

Coarser Matrix * Vector

N tasks, 3 task/row:



Granularity

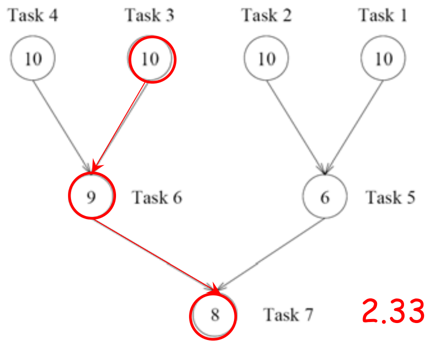
- Average degree of concurrency if we take into account varying *amount of work?*
- Critical path = longest directed path between any start & finish nodes.
- Critical path length = sum of the weights of nodes along this path.
- Average degree of concurrency = total amount of work / critical path length.

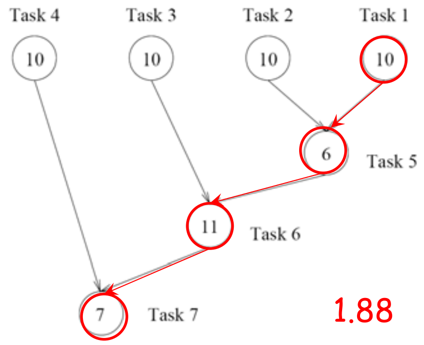
Weights on nodes denote the amount of work to be done on these nodes.
Longest path → shortest time needed to execute in parallel.

Database example

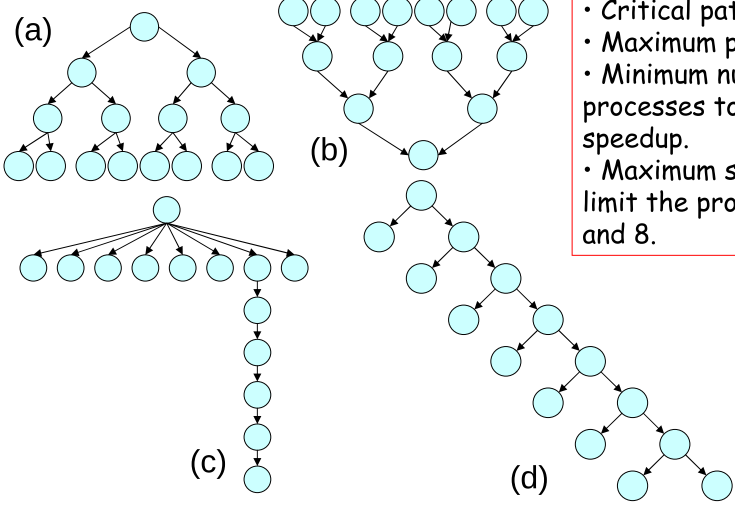
Critical path (3).
Critical path length = 27.
Av. deg. of concurrency = 63/27.

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





Exercise



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

Cassiopeia

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.

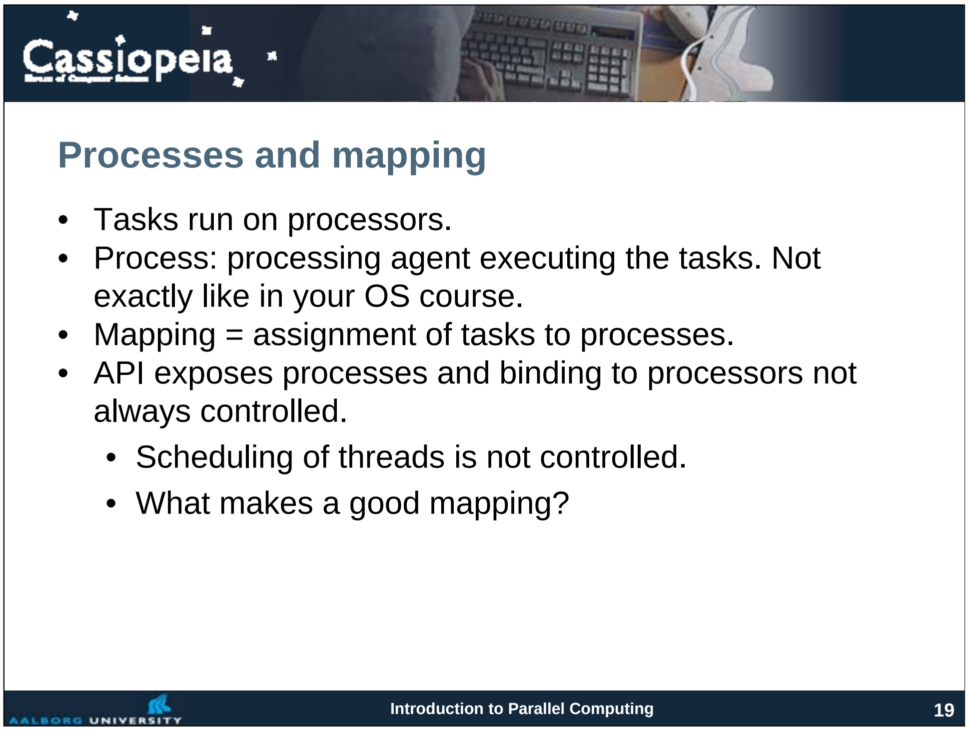
AALBORG UNIVERSITY Introduction to Parallel Computing 18

Another important factor is interaction between tasks *on different processors*.

Share data implies synchronization protocols (mutual exclusion, etc) to ensure **consistency**.

Edges generally undirected. When directed edges are used, they show the direction of the flow of data (and the flow is unidirectional).

Dependency between tasks implies interaction between them.



Here we are not talking directly on the mapping to processors. A processor can execute two processes.

Good mapping:

•Maximize concurrency by mapping independent tasks to different processes.

•Minimize interaction by mapping interacting tasks on the same process.

Can be conflicting, good trade-off is the key to performance.

Decomposition determines degree of concurrency.

Mapping determines how much concurrency is utilized and how efficiently.

Mapping example

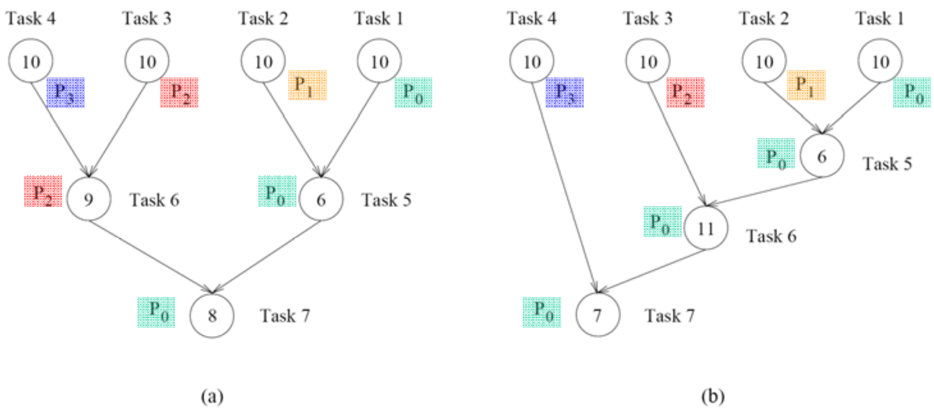
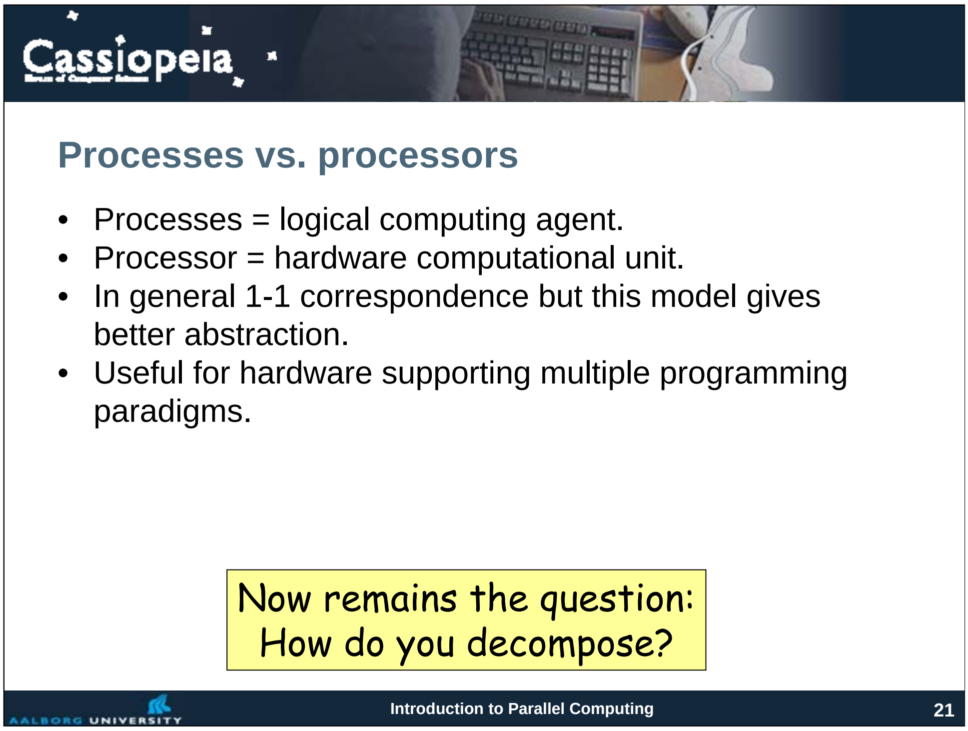


Figure 3.7 Mappings of the task graphs of Figure 3.5 onto four processes.

Notice that the mapping keeps one process from the previous stage because of dependency: We can avoid interaction by keeping the same process.



Example of hybrid hardware: cluster of MP machines. Each node has shared memory and communicates with other nodes via MPI.

1. Decompose and map to processes for MPI.
2. Decompose again but suitable for shared memory.

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.

Small problem is to start and finish: with one process only.

Quicksort example

5	12	11	1	10	6	8	3	7	4	9	2

Figure 3.8 The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.

24

Recall on the quicksort algorithm:

•Choose a pivot.

•Partition the array.

•Recursive call.

•Combine result: nothing to do.

Minimal number

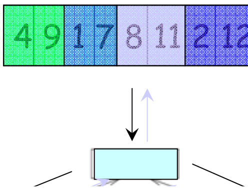


Figure 3.9 The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.

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.

Partitioning of input data is a bit similar to divide-and-conquer.

Cassiopeia

Matrix multiplication by block

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

(a)

$$\begin{aligned}
\text{Task 1: } C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
\text{Task 2: } C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
\text{Task 3: } C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
\text{Task 4: } C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}$$

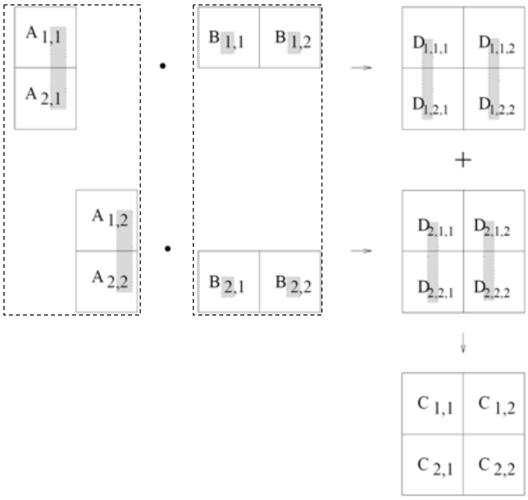
(b)

Figure 3.10 (a) Partitioning of input and output matrices into 2×2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

We can partition further for the tasks. Notice the dependency between tasks. What is the task dependency graph?

Cassiopeia

Intermediate data partitioning

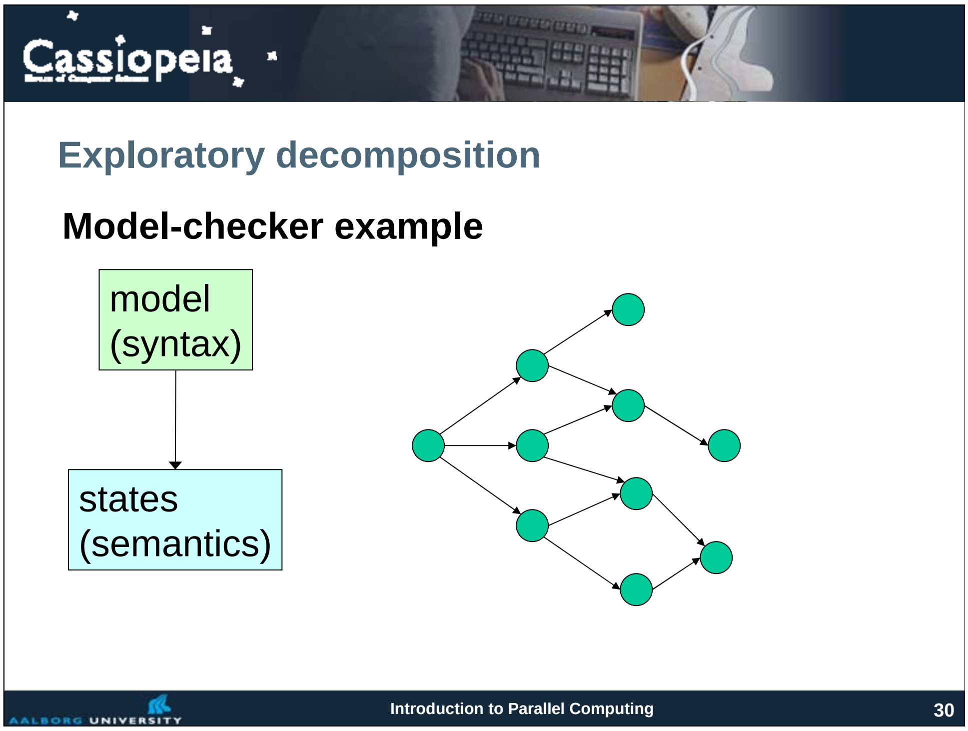


Linear combination of the intermediate results.

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.

Important rule, very useful, in particular stresses locality.



Suitable for search algorithms. Partition the search space into smaller parts and search in parallel. We search the solution by a tree search technique.

Performance anomalies

Work depends on the order of the search!

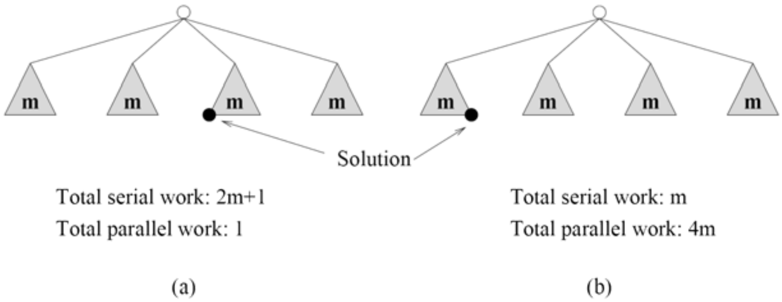


Figure 3.19 An illustration of anomalous speedups resulting from exploratory decomposition.

Cassiopeia

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.

AALBORG UNIVERSITY Introduction to Parallel Computing 32

Not possible to identify independent tasks in advance. Conservative approaches may yield limited concurrency. Optimistic approach = speculative. Optimistic approach is similar to branch prediction algorithms in processors.

So far…

- Decomposition techniques.
  - Identify tasks.
  - Analyze with task dependency & interaction graphs.
  - Map tasks to processes.
- Now properties of tasks that affect a good mapping.
  - Task generation, size of tasks, and size of data.

Cassiopeia

Task generation

- Static task generation.
  - Tasks are known beforehand.
  - Apply to well-structured problems.
- Dynamic task generation.
  - Tasks generated on-the-fly.
  - Tasks & task dependency graph not available beforehand.

The well-structured problem can typically be decomposed using data or recursive decomposition techniques.

Dynamic tasks generation: Exploratory or speculative decomposition techniques are generally used, but not always. Example: quicksort.

Cassiopeia

Task sizes

- Relative amount of time for completion.
  - Uniform – same size for all tasks.
    - Matrix multiplication.
  - Non-uniform.
    - Optimization & search problems.

Typically the size of non-uniform tasks is difficult to evaluate beforehand.

Size of data associated with tasks

- Important because of locality reasons.
- Different types of data with different sizes
  - Input/output/intermediate data.
- Size of context – cheap or expensive communication with other tasks.

Cassiopeia

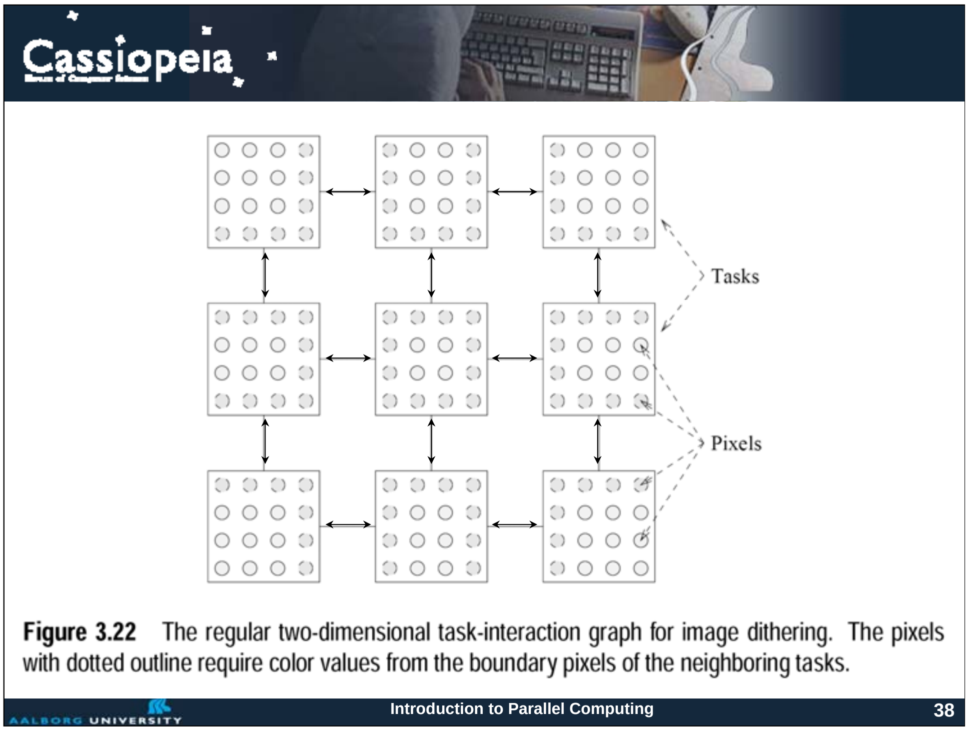
Characteristics of task interactions

- Static interactions.
  - Tasks and interactions known beforehand.
  - And interaction at pre-determined times.
- Dynamic interactions.
  - Timing of interaction unknown.
  - Or set of tasks not known in advance.
- Regular interactions.
  - The interaction graph follows a pattern.
- Irregular interactions.
  - No pattern.

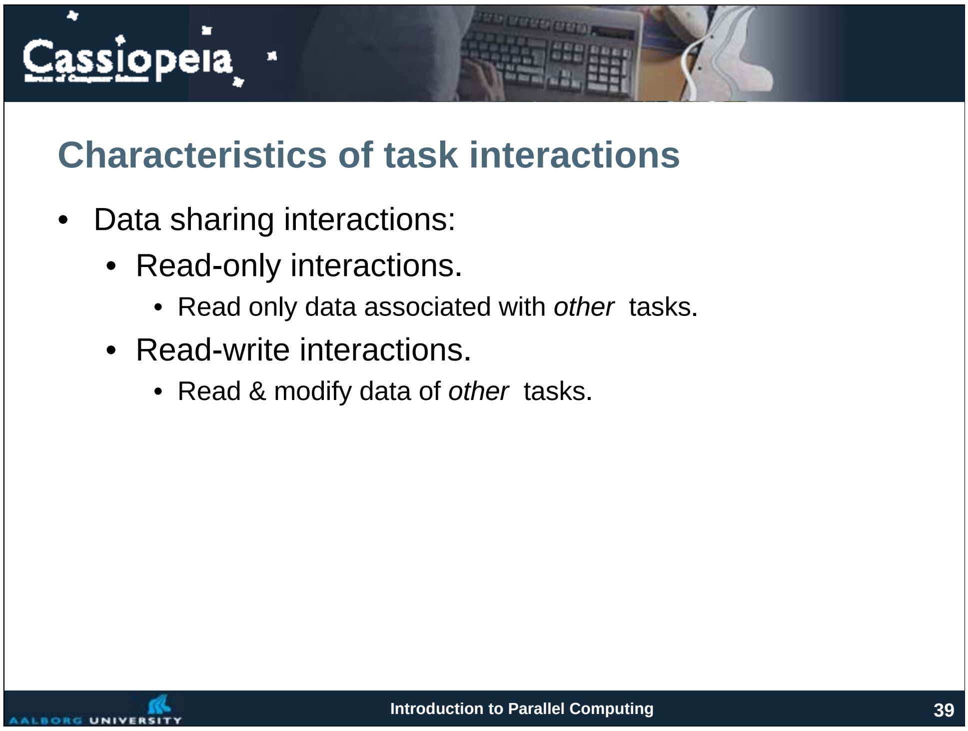
Static vs. dynamic.

Static or dynamic interaction pattern.

Dynamic harder to code, more difficult for MPI.



The color of each pixel is determined as the weighted average of its original value and the values of the neighboring pixels. Decompose into regions, 1 task/region. Pattern is a 2-D mesh. Regular pattern.



Read-only vs. read-write.

Read-only example: matrix multiplication (share input). Read-write example: 15-puzzle with shared priority list of states to be explored; Priority given by some heuristic to evaluate the distance to the goal.

Cassiopeia

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.

AALBORG UNIVERSITY   Introduction to Parallel Computing   40

One-way vs. two-way.

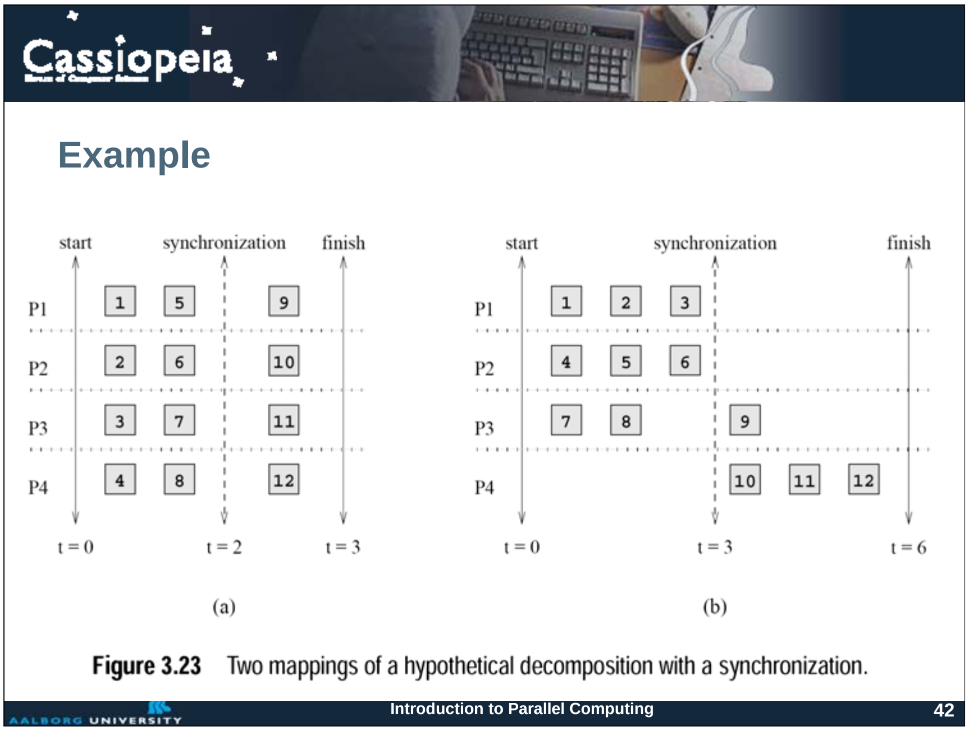
One-way more difficult with MPI since MPI has an explicit send & receive set of calls. Conversion one-way to two-way with polling or another thread waiting for communication.

Mapping techniques for load balancing

- Map tasks onto processes.
- Goal: minimize overheads.
  - Communication.
  - Idling.
- Uneven load distribution may cause idling.
  - Constraints from task dependency $\rightarrow$ wait for other tasks.

Minimizing communication may contradict minimizing idling. Put tasks that communicate with each other on the same process but may unbalance the load -> distribute them but increase communication.

Load balancing is not enough to minimize idling.



Global balancing OK but due to task dependency P4 is idling.

Cassiopeia

Mapping techniques

- Static mapping.
  - NP-complete problem for non-uniform tasks.
  - Large data compared to computation.
- Dynamic mapping.
  - Dynamically generated tasks.
  - Task size unknown.

Even static mapping may be difficult: The problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks. In practice simple heuristics provide good mappings.

Cost of moving data may out-weight the advantages of dynamic mapping.

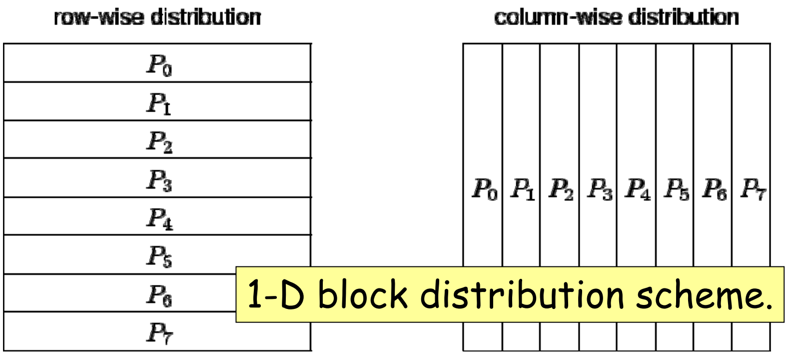
In shared address space dynamic mapping may work well even with large data, but be careful with the underlying architecture (NUMA/UMA) because data may be moved physically.

Schemes for static mapping

- Mappings based on data partitioning.
- Mappings based on task graph partitioning.
- Hybrid mappings.

Array distribution scheme

- Combine with "owner computes" rule to partition into sub-tasks.



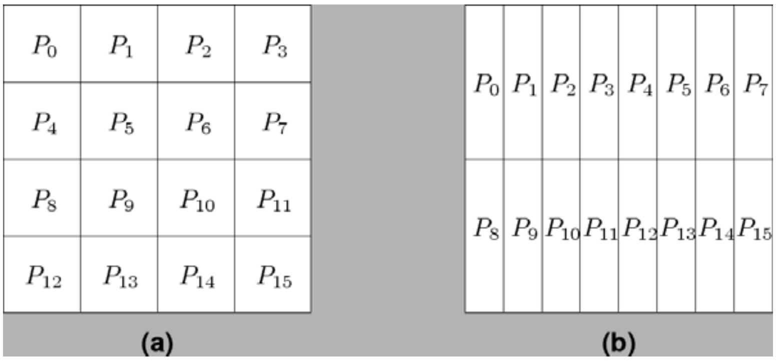
Data partitioning mapping.

Mapping data = mapping tasks.

Simple block-distribution.

Cassiopeia

Block distribution cont.



Generalize to higher dimensions: 4x4, 2x8.

Example: Matrix*Matrix

- Partition output of C=A*B.
- Each entry needs the same amount of computation.
- Blocks on 1 or 2 dimensions.
- Different data sharing patterns.
- *Higher dimensional* distributions
  - means we can use *more processes*.
  - sometimes *reduces* interaction.

In the case of matrix n*n multiplication, 1-D -> n processes at most, 2-D n^2 processes at most.

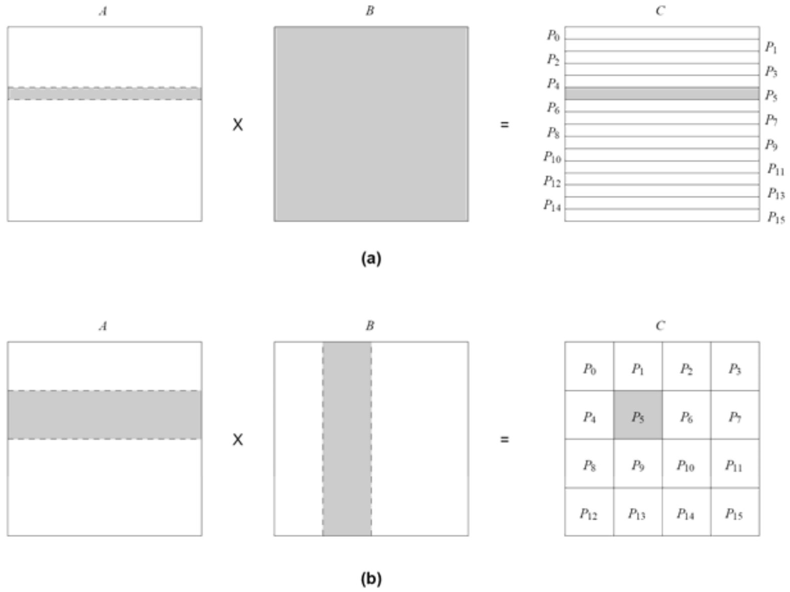


Figure 3.26 Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

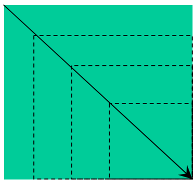
48

O(n^2/sqrt(p)) vs. O(n^2) shared data.

Cassiopeia

Imbalance problem

- If the amount of *computation* associated with data *varies* a lot then *block decomposition* leads to *imbalances*.
- Example: LU factorization (or Gaussian elimination).

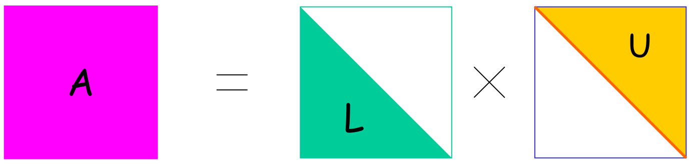


Computations

Exercise on LU-decomposition.

LU factorization

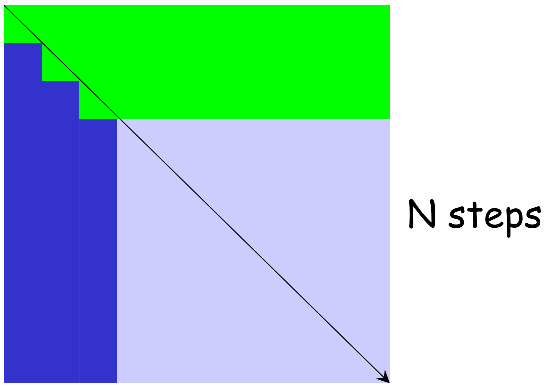
- Non singular square matrix A (invertible).
- A = L*U.
- Useful for solving linear equations.



Cassiopeia

LU factorization

In practice we work on A.



LU algorithm

U[k,k]

```
Proc LU(A)
begin
  for k := 1 to n-1 do
    for j := k+1 to n do
      A[j,k] := A[j,k]/A[k,k]
    endfor
    for j := k+1 to n do
      for i := k+1 to n do
        A[i,j] := A[i,j] - A[i,k]*A[k,j]
      endfor
    endfor
  endfor
end
```

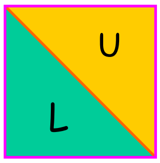
L[j,k]

Normalize L

U[k,j] := A[k,j]/L[k,k]

L[i,k] U[k,j]

A



Cassiopeia

Decomposition

Exercise:
- Task dependency graph?
- Mapping to 3 & 4 processes?

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$	6: $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$	11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$
2: $L_{2,1} = A_{2,1}U_{1,1}^{-1}$	7: $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$	12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$
3: $L_{3,1} = A_{3,1}U_{1,1}^{-1}$	8: $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$	13: $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$
4: $U_{1,2} = L_{1,1}^{-1}A_{1,2}$	9: $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$	14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$
5: $U_{1,3} = L_{1,1}^{-1}A_{1,3}$	10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$	

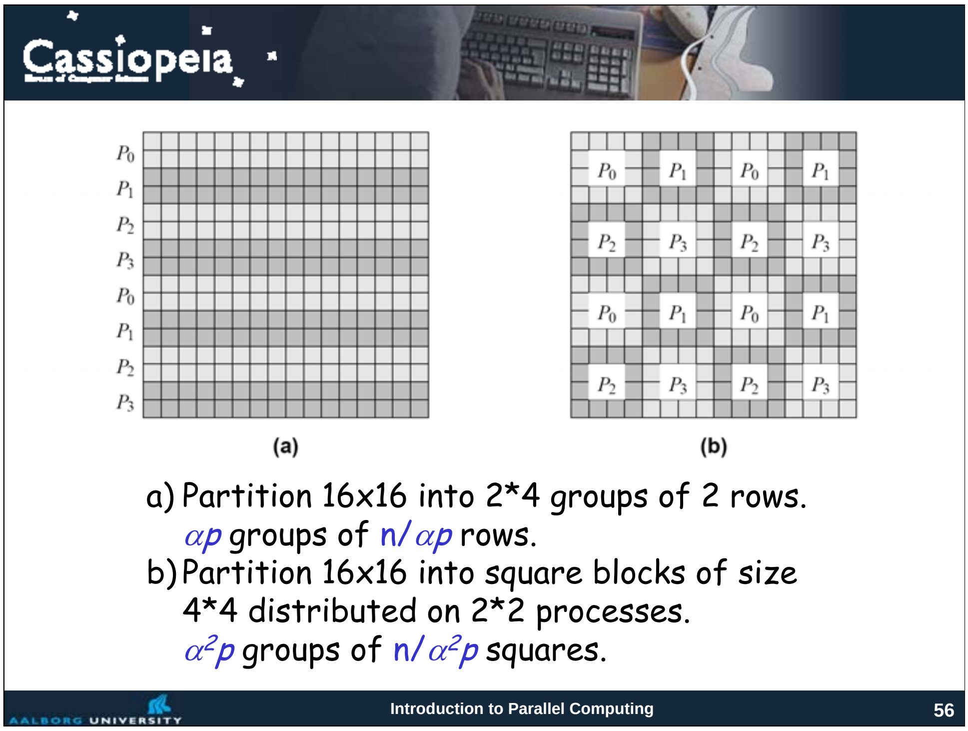
Figure 3.27 A decomposition of LU factorization into 14 tasks.

AALBORG UNIVERSITY Introduction to Parallel Computing 53

Load imbalance for individual tasks. Load imbalance from dependencies.

Cyclic and block-cyclic distributions

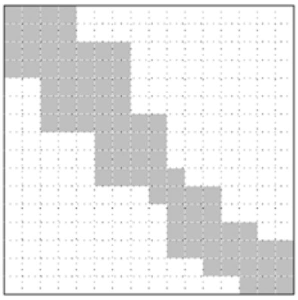
- Idea:
  - Partition an array into many *more blocks than available processes*.
  - Assign partitions (tasks) to processes in a round-robin manner.
- $\rightarrow$ each process gets several non adjacent blocks.



Reduce the amount of idling because all processes have a sampling of tasks from *all parts* of the matrix.

But lack of locality may result in performance penalties + leads to high degree of interaction. Good value for α to find a compromise.

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

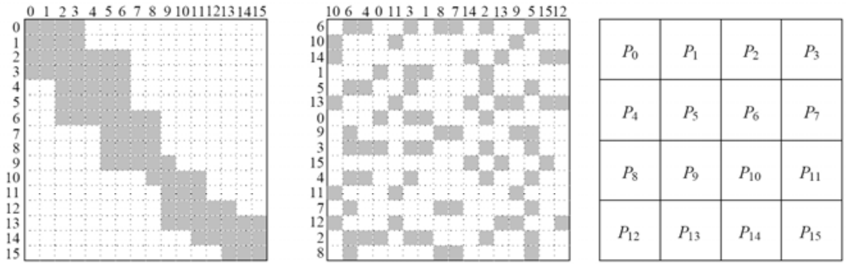
$\text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$

mapping = 8 2 6 0 3 7 11 1 9 5 4 10

P_0 P_1 P_2 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.

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

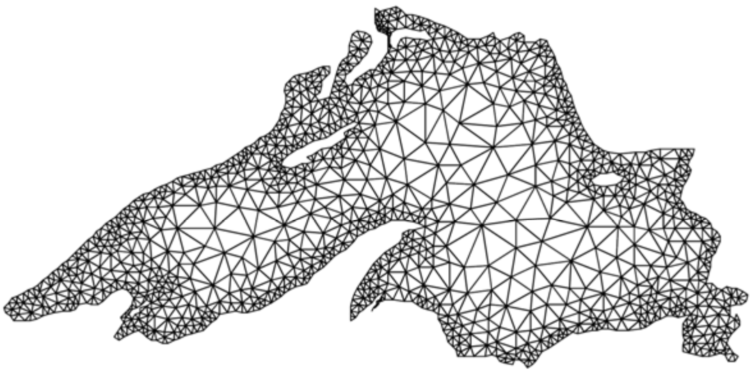
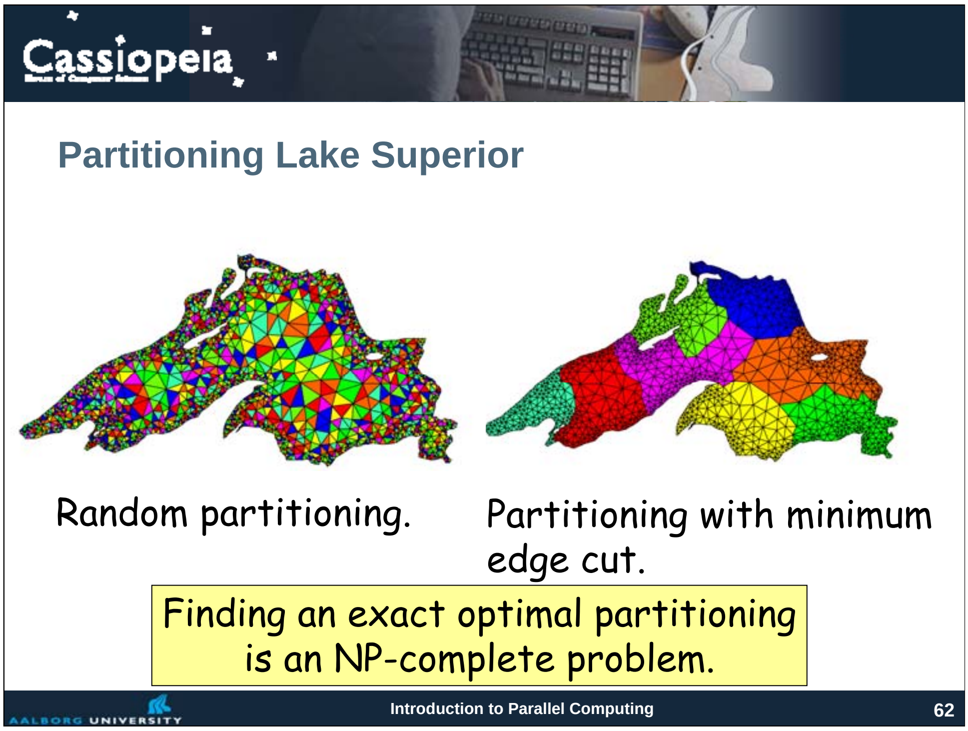


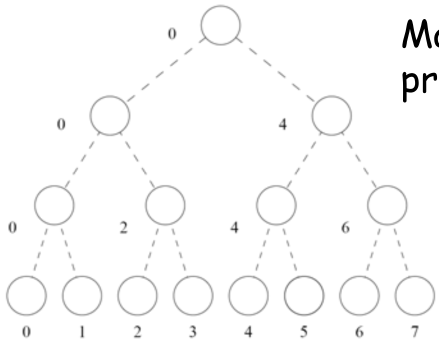
Figure 3.34 A mesh used to model Lake Superior.



Minimum edge cut from a graph point of view. Keep locality of data with processes to minimize interaction.

Mappings based on task partitioning

- Partition the task dependency graph.
  - Good when static task dependency graph with known task sizes.



Mapping on 8 processes.

Determining an optimal mapping is NP-complete. Good heuristics for structured graphs.

Binary tree task dependency graph: occurs in recursive decompositions as seen before. The mapping minimizes interaction. There is idling but it is inherent to the task dependency graph, we do not add more.

This example good on a hypercube. See why?

Hierarchical mappings

- Combine several mapping techniques in a structured (hierarchical) way.
- Task mapping of a binary tree (quicksort) does not use all processors.
  - Mapping based on task dependency graph (hierarchy) & block.

Schemes for dynamic mapping

- Centralized Schemes.
  - Master manages pool of tasks.
  - Slaves obtain work.
  - Limited scalability.
- Distributed Schemes.
  - Processes *exchange tasks* to balance work.
  - Not simple, many issues.

Centralized schemes are easy to implement but present an obvious bottleneck (the master).

Self-scheduling: slaves pick up work to do whenever they are idle.

Bottleneck: tasks of size M, it takes t to assign work to a slave → at most M/t processes can be kept busy.

Chunk-scheduling: a way to reduce bottlenecks by getting a group of tasks. Problem for load imbalances.

Distributed schemes more difficult to implement.

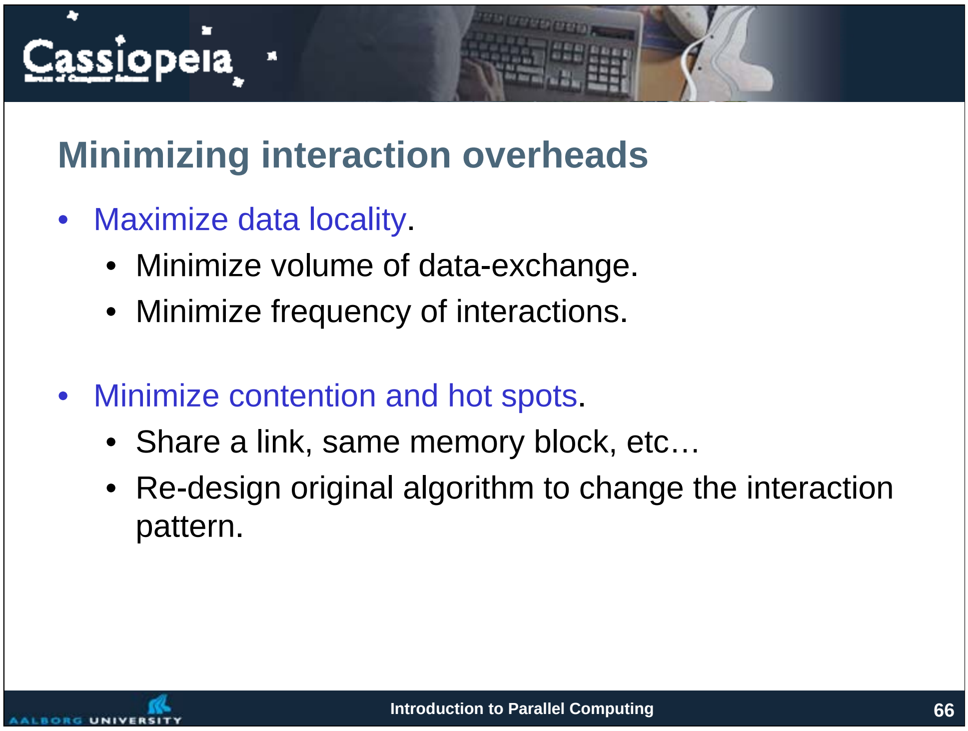
How do you choose sender & receiver? i.e. if A is overloaded, which process gets something?

Initiate transfer by sender or receiver? i.e. A overloaded sends work or B idle requests work?

How much work to transfer?

When to transfer?

Answers are application specific.



Minimize volume of exchange $\rightarrow$ maximize temporal locality. Use higher dimensional distributions, like in the matrix multiplication example. We can store intermediate results and update global results less often.

Minimize frequency of interactions $\rightarrow$ maximize spatial locality.

Related to the previously seen cost model for communications.

Changing the interaction pattern: For the matrix multiplication example, the sum is commutative so we can re-order the operations modulo sqrt(p) to remove contention.

Cassiopeia

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.

AALBORG UNIVERSITY Introduction to Parallel Computing 67

Replication is useful when the cost of interaction is greater than replicating the computation. Replicating data is like caching, good for read-only accesses. Processing power is cheap, memory access is expensive – also apply at larger scale with communicating processes.

Collective communication such as broadcast. However, depending on the communication pattern, a custom collective communication may be better.