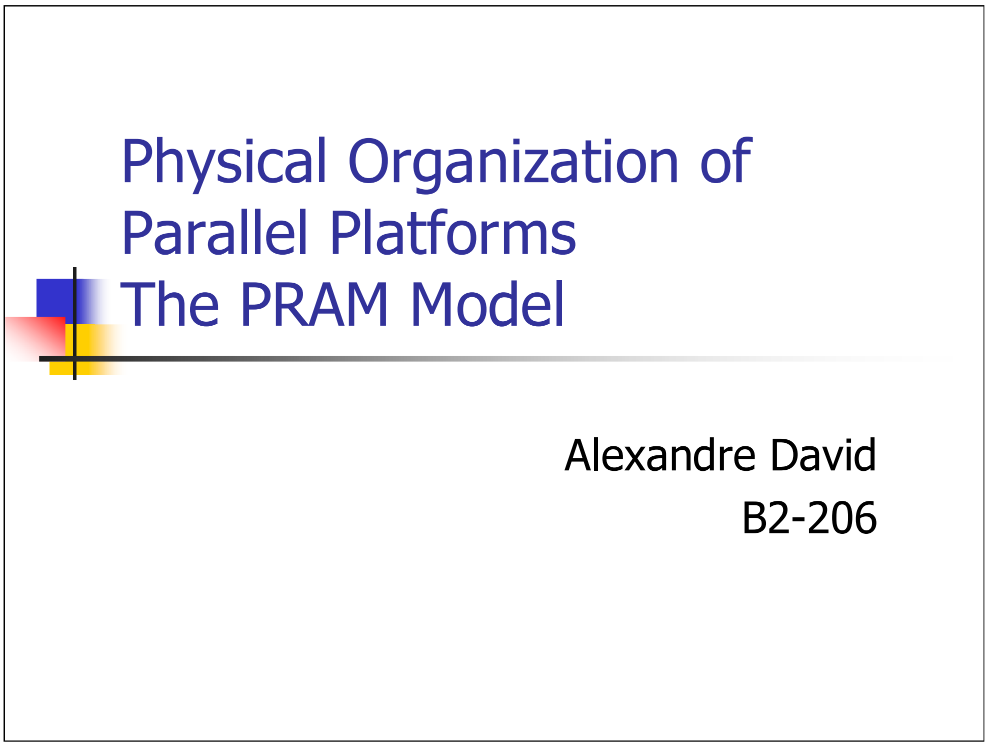


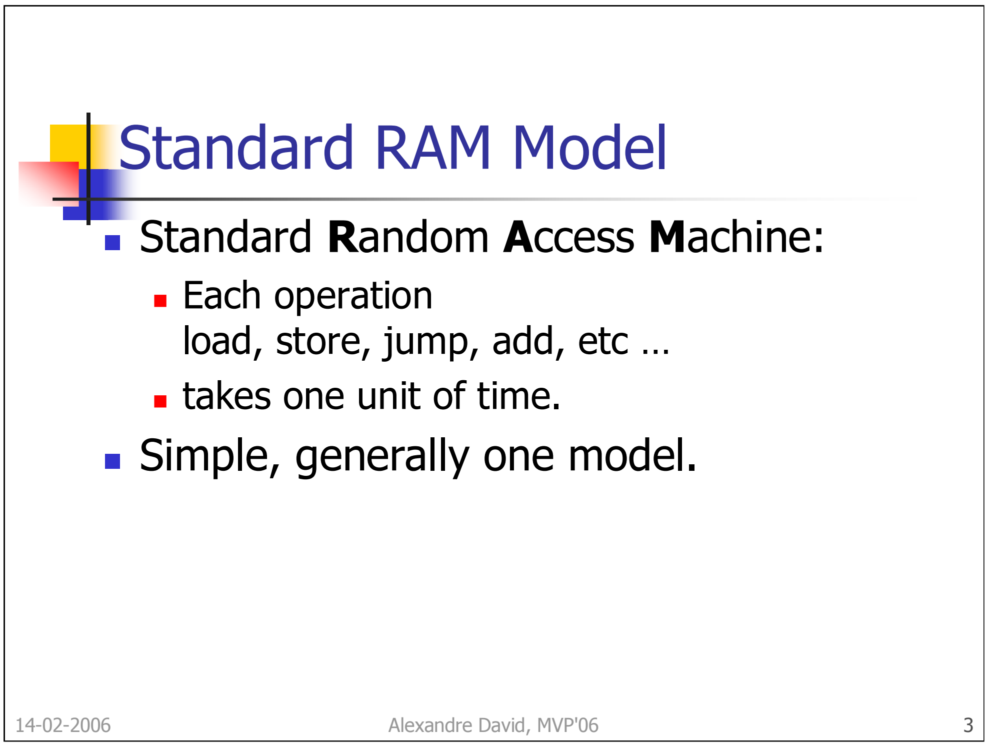
Physical Organization of Parallel Platforms The PRAM Model

Alexandre David

B2-206

Today

- Introduction to Parallel Algorithms (Sven Skyum)
    - PRAM model
    - Optimality
    - Examples
- Physical Organization of Parallel Platforms (2.4)

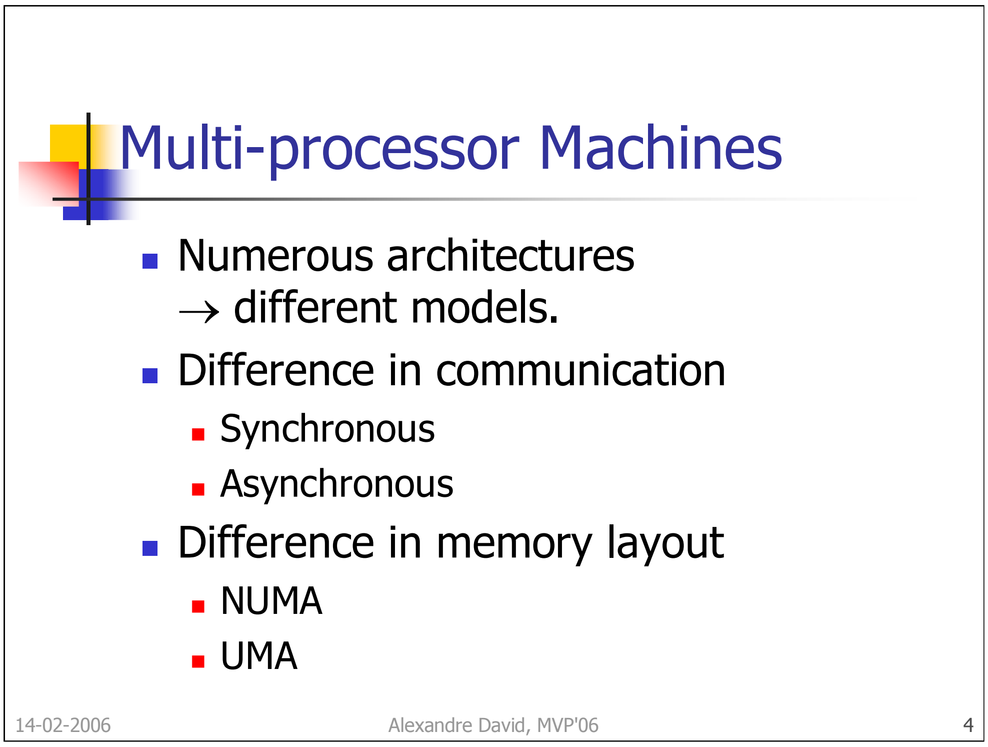


Standard RAM Model

- Standard **R**andom **A**ccess **M**achine:
  - Each operation
    load, store, jump, add, etc …
  - takes one unit of time.
- Simple, generally one model.

14-02-2006 Alexandre David, MVP'06 3

The RAM is the basic machine model behind *sequential* algorithms.



Multi-processor Machines

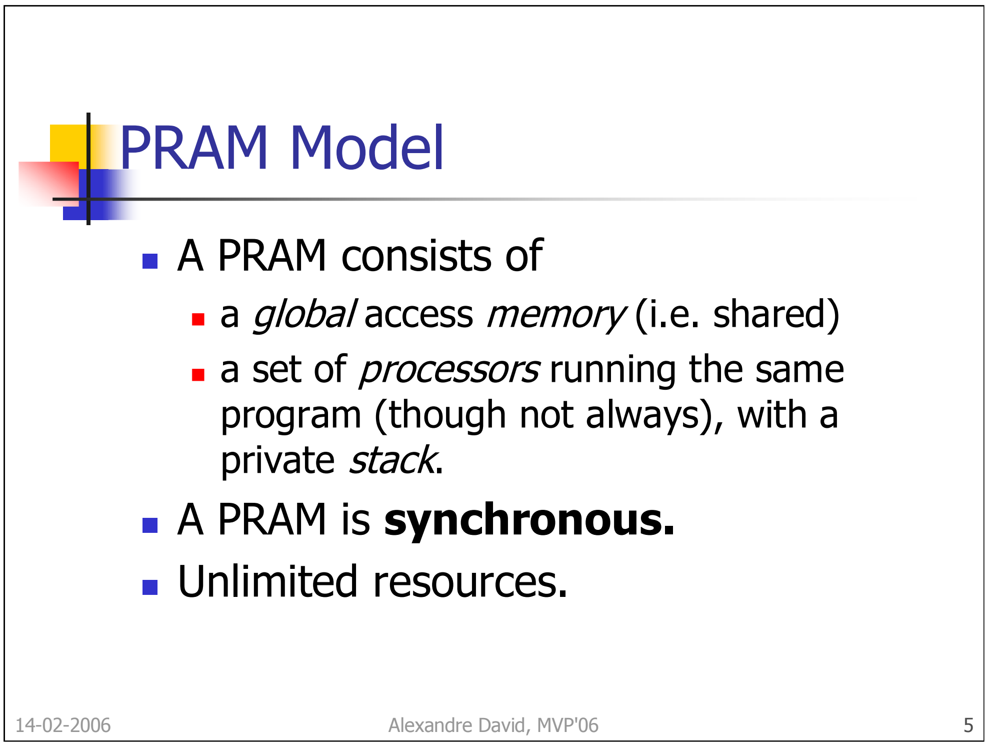
- Numerous architectures
  $\rightarrow$ different models.
- Difference in communication
  - Synchronous
  - Asynchronous
- Difference in memory layout
  - NUMA
  - UMA

14-02-2006 Alexandre David, MVP'06 4

Even if there are different architectures and models, the goal is to *abstract* from the hardware and have a model on which to reason and analyze algorithms. Synchronous vs. asynchronous communication is like *blocking* vs. *non-blocking* communication. NUMA is assumed most often when the model talks about local memory to a given processor.

Clusters of computers correspond to NUMA in practice. They are best suited for message passing type of communication.

Shared memory systems are easier from a programming model point of view but are more expensive.



PRAM Model

- A PRAM consists of
  - a *global* access *memory* (i.e. shared)
  - a set of *processors* running the same program (though not always), with a private *stack*.
- A PRAM is **synchronous.**
- Unlimited resources.

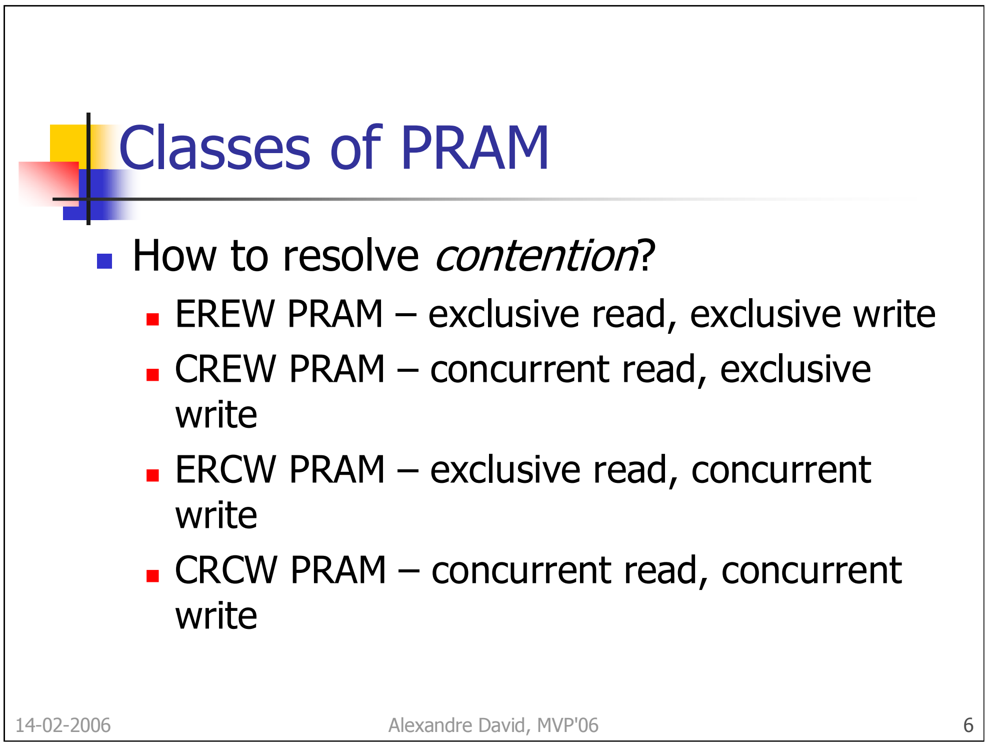
14-02-2006 Alexandre David, MVP'06 5

PRAM **model** – Parallel Random Access Machine.

In the report the stack is called accumulator.

Synchronous PRAM means that all processors follow a global clock (ideal model!). There is no direct or explicit communication between processors (such as message passing). Two processors communicate if one *reads* what another *writes*.

Unlimited resources means we are not limited by the size of the memory and the *number of processors* varies in function of the size of the problem, i.e., we have access to as many processors as we want. Designing algorithms for many processors is very fruitful in practice even with very few processors in practice whereas the opposite is limiting.



Classes of PRAM

- How to resolve *contention*?
  - EREW PRAM – exclusive read, exclusive write
  - CREW PRAM – concurrent read, exclusive write
  - ERCW PRAM – exclusive read, concurrent write
  - CRCW PRAM – concurrent read, concurrent write

14-02-2006 Alexandre David, MVP'06 6

The most powerful model is of course CRCW where everything is allowed but that's the most unrealistic in practice too. The weakest model is EREW where concurrency is limited, closer to real architectures although still infeasible in practice (need *m*p* switches to connect *p* processors to *m* memory cells and provide exclusive access).

Exclusive read/write means access is serialized.

Main protocol to **resolve contention** (writing is the problem):

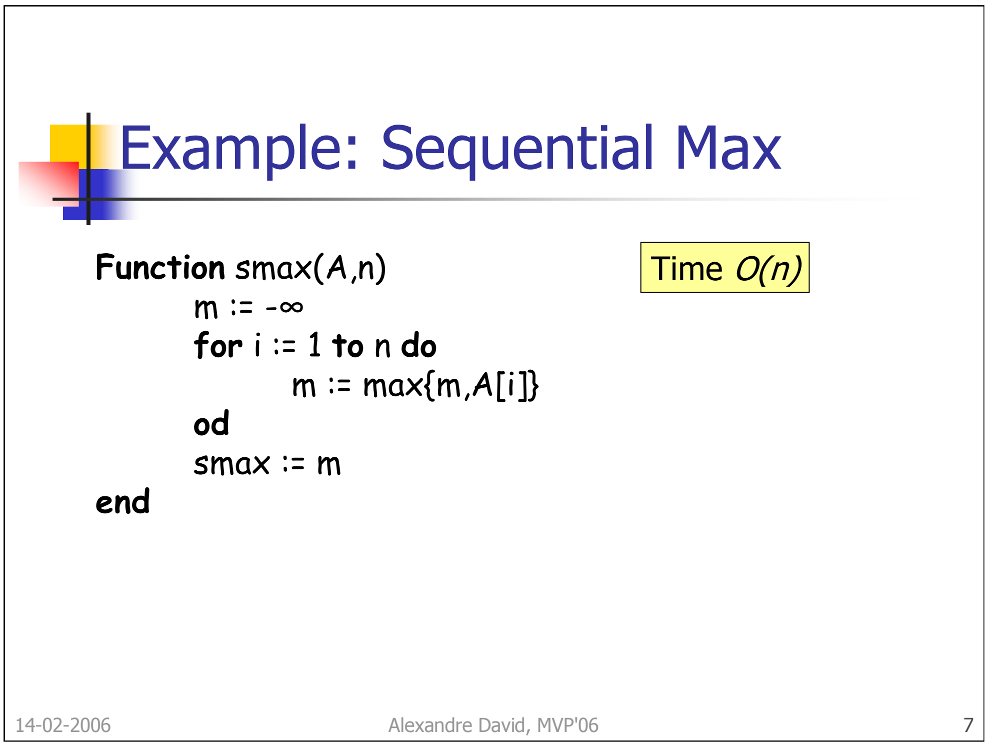
•Common: concurrent write allowed if the values are identical.

•Arbitrary: only an arbitrary processes succeeds.

•Priority: processes are ordered.

•Sum: the result is the sum of the values to be stored.

Exclusive write is exclusive with reads too.



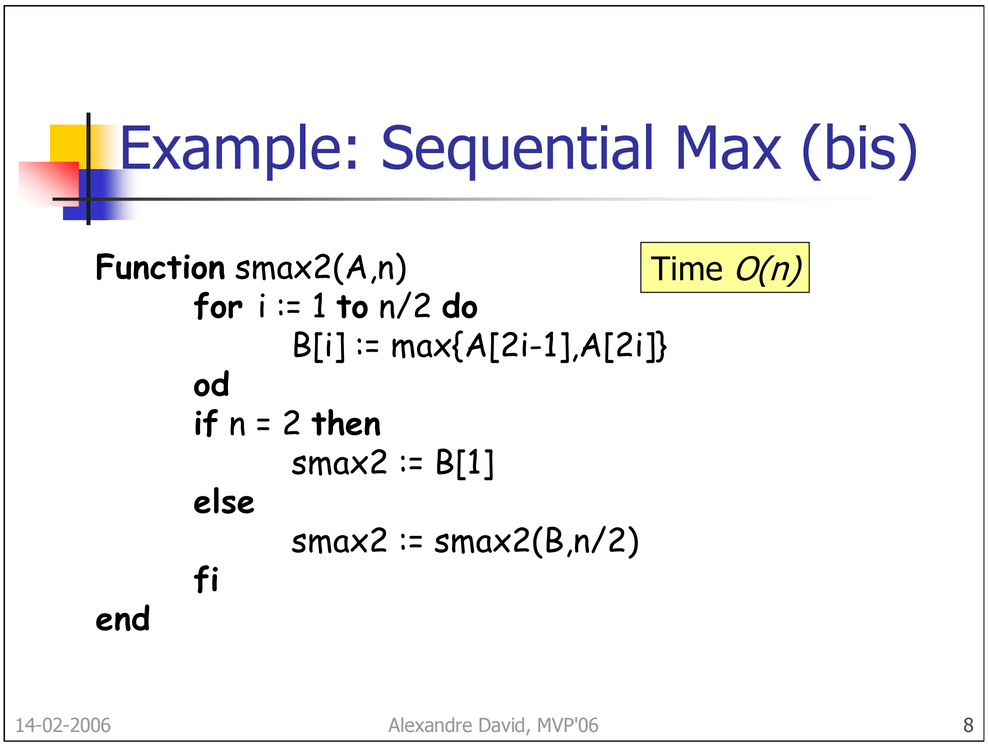
Example: Sequential Max

```
Function smax(A,n)
        m := -∞
        for i := 1 to n do
                m := max{m,A[i]}
        od
        smax := m
end
```

Time *O(n)*

Simple algorithm description, independent from a given language. See your previous course on algorithms. O-notation used, check your previous course on algorithms too.

Highly sequential, difficult to parallelize.



Example: Sequential Max (bis)

```
Function smax2(A,n)                    Time O(n)
    for i := 1 to n/2 do
        B[i] := max{A[2i-1],A[2i]}
    od
    if n = 2 then
        smax2 := B[1]
    else
        smax2 := smax2(B,n/2)
    fi
end
```

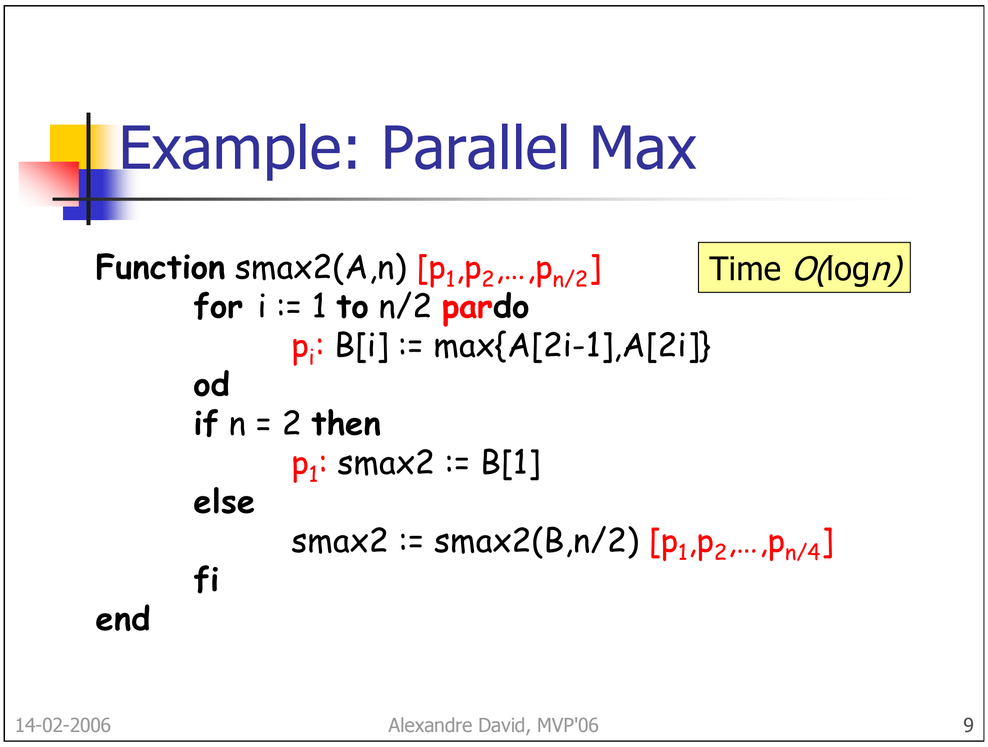
Remarks:

•Additional memory needed in this description

•B[i] *compresses* the array A[1..n] to B[1..n/2] with every element being the max of two elements from A (all elements are taken).

•The test serves to stop the recursive call – termination!

This is an example of the *compress and iterate* paradigm which leads to natural parallelizations. Here the computations in the for loop are independent and the recursive call tree gives the dependency between tasks to perform.



Example: Parallel Max

Time $O(\log n)$

```
Function smax2(A,n) [$p_1,p_2,…,p_{n/2}$]
        for i := 1 to n/2 pardo
                $p_i$: B[i] := max{A[2i-1],A[2i]}
        od
        if n = 2 then
                $p_1$: smax2 := B[1]
        else
                smax2 := smax2(B,n/2) [$p_1,p_2,…,p_{n/4}$]
        fi
end
```

14-02-2006 Alexandre David, MVP'06 9

EREW-PRAM algorithm.. Why? There is actually no contention and the dependencies are resolved by the recursive calls (when they return).

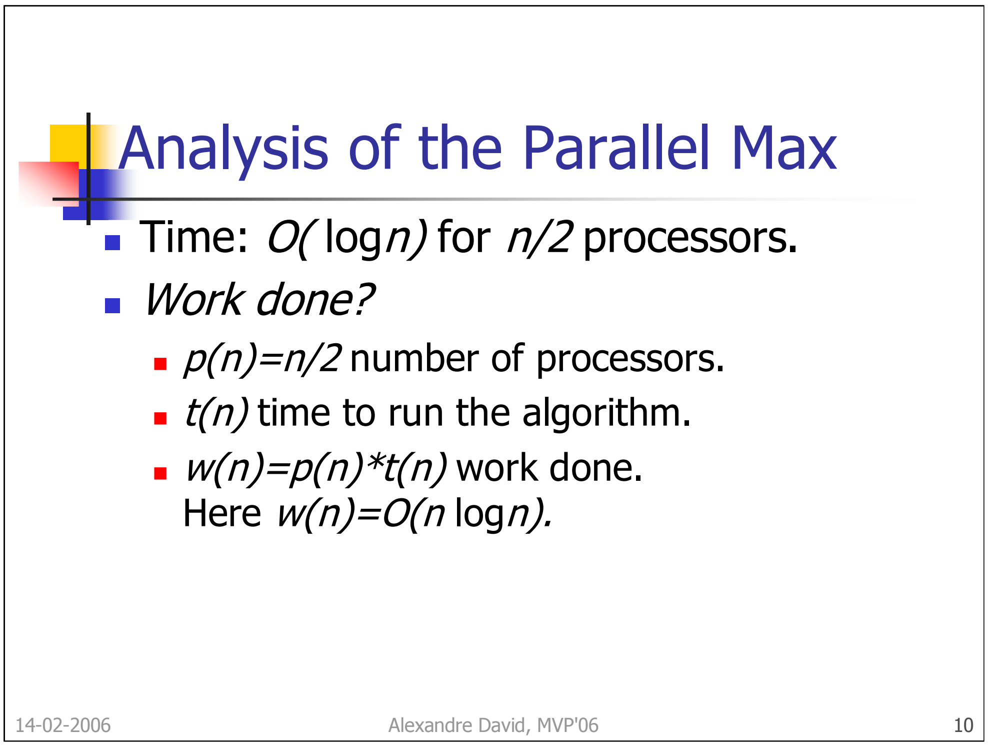
Here we give in brackets the processors used to solve the current problem.

Time *t(n)* to execute the algorithms satisfies *t(n)=O(1)* for *n=2* and *t(n)=t(n/2)+O(1)* for *n>2*. Why?

Think parallel and PRAM (all operations synchronized, same speed, p_i: operation in parallel). The loop is done in constant time on n/2 processors in parallel.

How many calls? How to solve *t(n)=f(n)*?

Answer: see your course on algorithms. Here simple recursion tree log*n* calls with constant time: *t(n)=O(*log*n)*. **Note:** log base 2. You are expected to know a minimum about log.



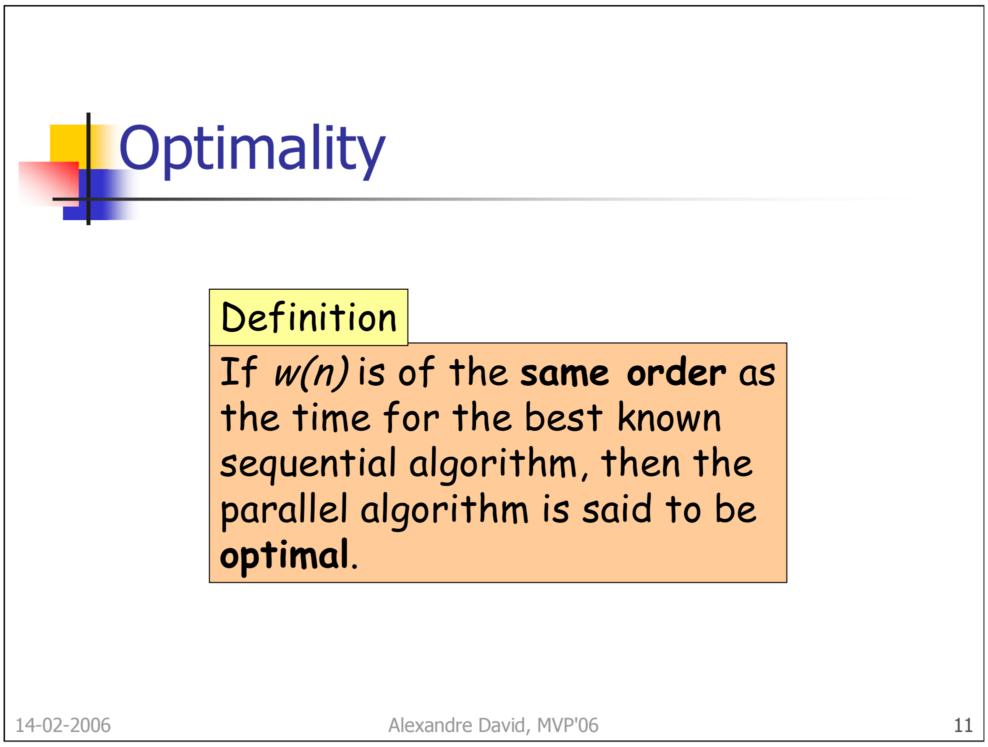
Analysis of the Parallel Max

- Time: $O(\log n)$ for $n/2$ processors.
- *Work done?*
  - $p(n)=n/2$ number of processors.
  - $t(n)$ time to run the algorithm.
  - $w(n)=p(n)*t(n)$ work done.
    Here $w(n)=O(n \log n)$.

14-02-2006 Alexandre David, MVP'06 10

Work done corresponds to the actual amount of computation done (not exactly though). In general when we parallelize algorithms, the total amount of computations is greater than the original, but by a constant if we want to be optimal.

The work measures the time required to run the parallel algorithm on one processor that would simulate all the others.



Optimality

Definition

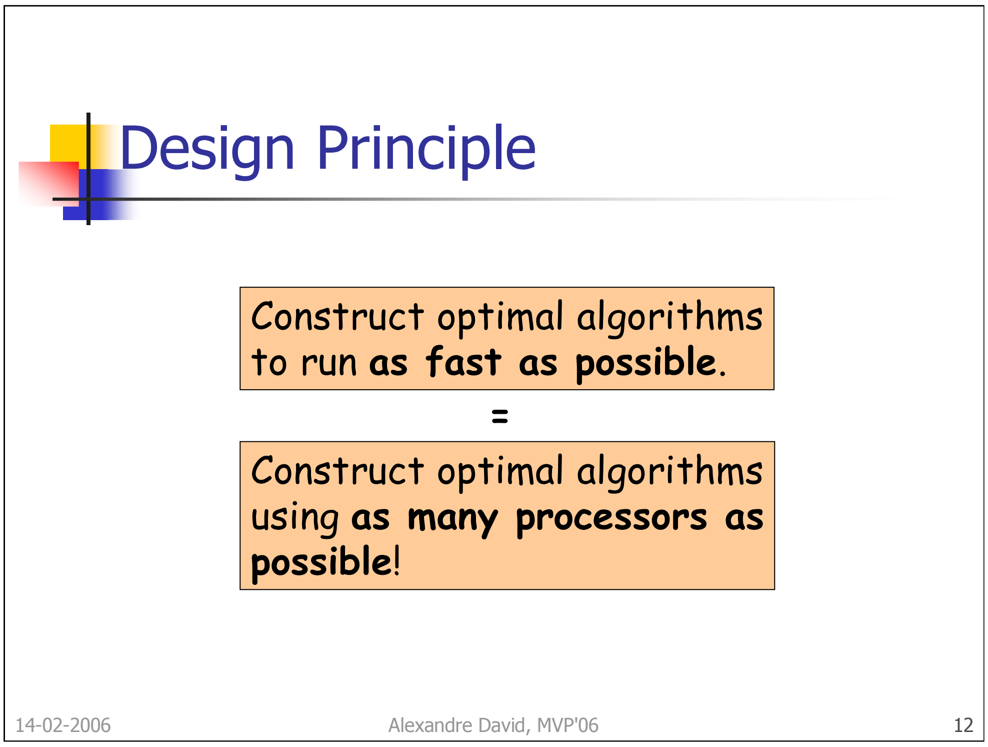
If *w(n)* is of the **same order** as the time for the best known sequential algorithm, then the parallel algorithm is said to be **optimal**.

What about our previous example?

It's **not** optimal. Why? Well, we use only *n/2*,*n/4*,…,*2*,*1* processors, not *n* all the time!

We do not want to waste time like that right?

Another way to see it is that you get a speed-up linear to the number of processors (though at a constant factor, which means sub-linear).



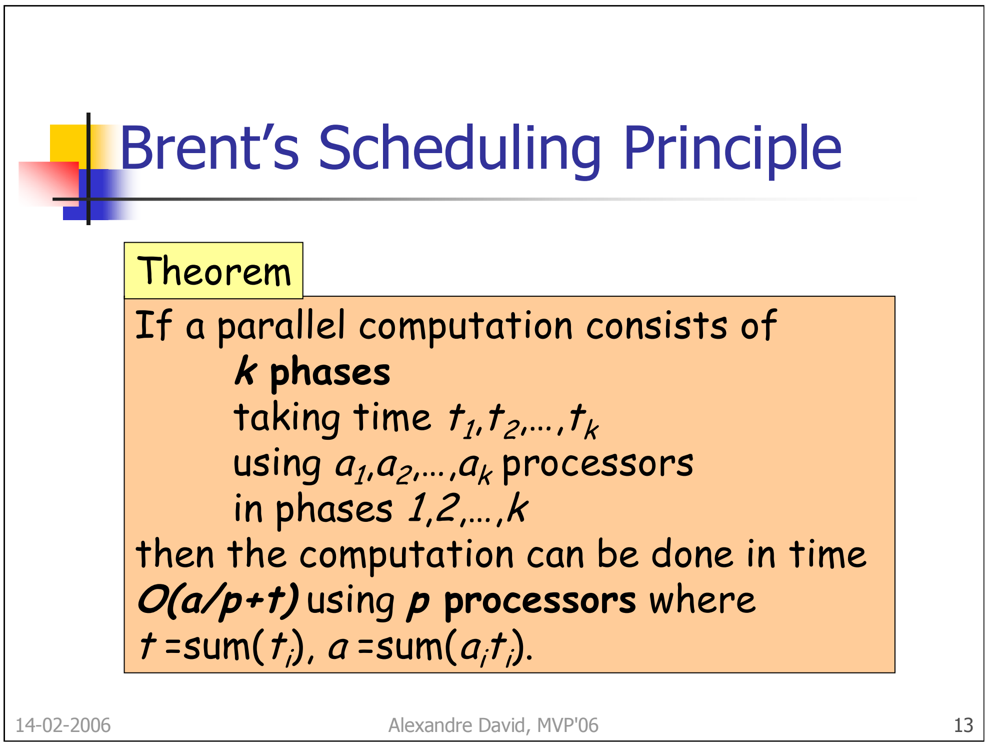
Design Principle

Construct optimal algorithms to run **as fast as possible**.

=

Construct optimal algorithms using **as many processors as possible**!

Note that if we have an optimal parallel algorithm running in time *t(n)* using *p(n)* processors then there exist optimal algorithms using *p'(n)<p(n)* processors running in time *O(t(n)*p(n)/p'(n))*. That means that **you can use fewer processors to simulate an optimal algorithm that is using many processors**! The goal is to maximize utilization of our processors. Simulating does not add work with respect to the parallel algorithm.



Brent’s Scheduling Principle

Theorem

If a parallel computation consists of

k phases

taking time $t_1, t_2, \ldots, t_k$

using $a_1, a_2, \ldots, a_k$ processors

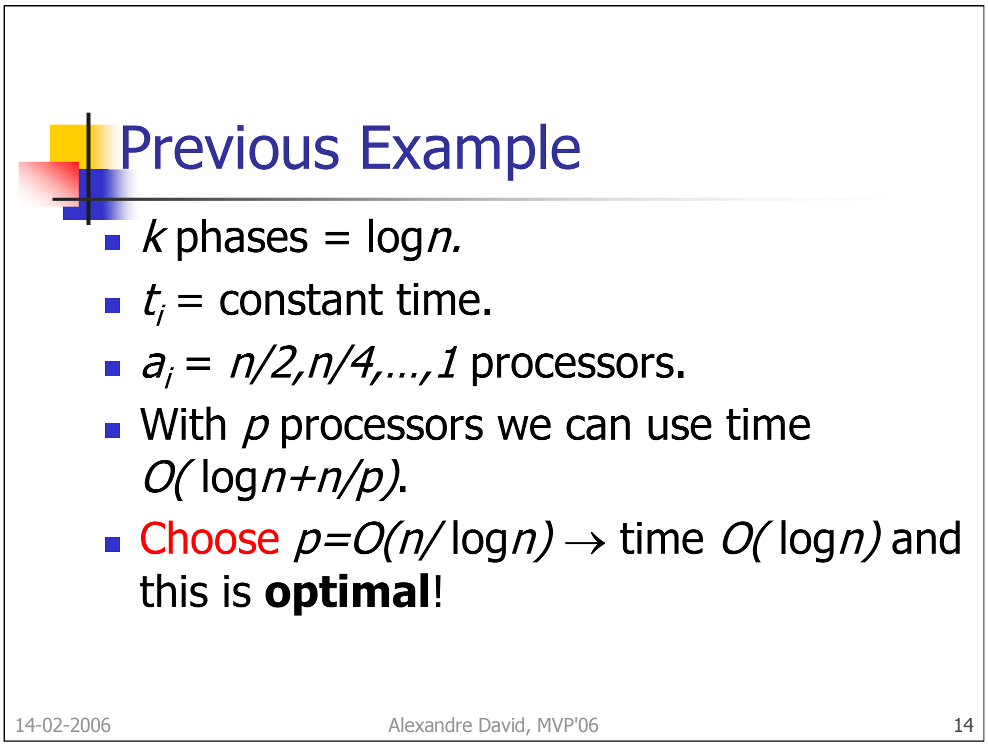
in phases $1, 2, \ldots, k$

then the computation can be done in time ***$O(a/p+t)$*** using ***p* processors** where $t = \mathrm{sum}(t_i)$, $a = \mathrm{sum}(a_i t_i)$.

What it means: same time as the original plus an overhead. If the number of processors increases then we decrease the overhead. The overhead corresponds to simulating the a_i with p. What it **really** means: It is possible to make algorithms optimal with the right amount of processors (provided that $t*p$ has the same order of magnitude of $t_{\text{sequential}}$). That gives you a bound on the number of needed processors.

It’s a *scheduling* principle to reduce the number of physical processors needed by the algorithm and increase utilization. It does not do miracles.

Proof: i’th phase, p processors simulate a_i processors. Each of them simulate at most ceil(a_i/p)≤$a_i/p+1$, which consumes time t_i at a constant factor for each of them.

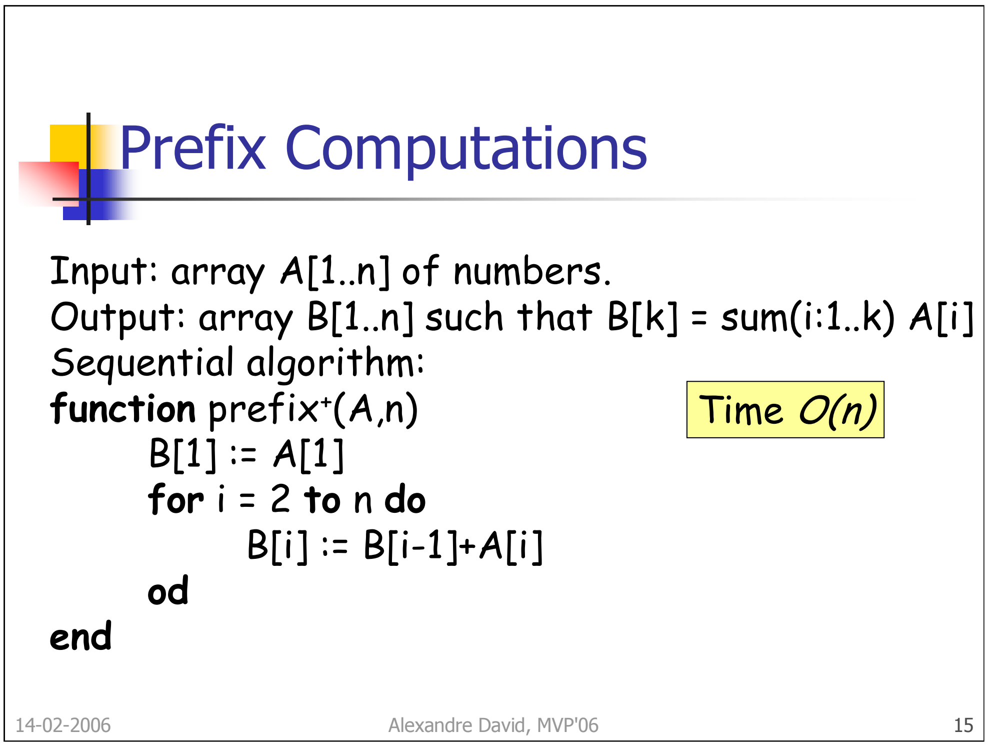


Previous Example

- k phases = log$n$.
- t_i = constant time.
- a_i = $n/2, n/4, \ldots, 1$ processors.
- With p processors we can use time $O(\log n + n/p)$.
- Choose $p = O(n/\log n) \rightarrow$ time $O(\log n)$ and this is **optimal**!

Note: n is a power of 2 to simplify. Recall the definition of optimality to conclude that it is optimal indeed. This does not gives us an implementation, but almost.

Typo p6 “using $O(n/\log n)$ processors”. Divide and conquer same as compress and iterate for the exercise.

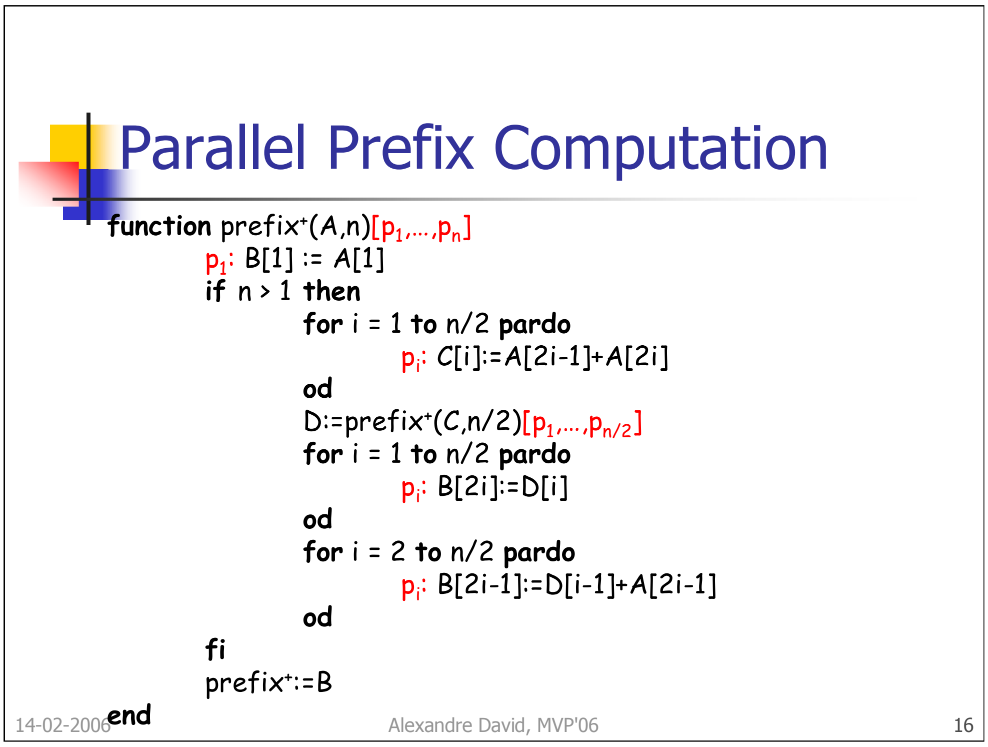


Prefix Computations

Input: array A[1..n] of numbers.
Output: array B[1..n] such that B[k] = sum(i:1..k) A[i]
Sequential algorithm:

```
function prefix$^+$(A,n)
        B[1] := A[1]
        for i = 2 to n do
                B[i] := B[i-1]+A[i]
        od
end
```

$\boxed{\text{Time } O(n)}$

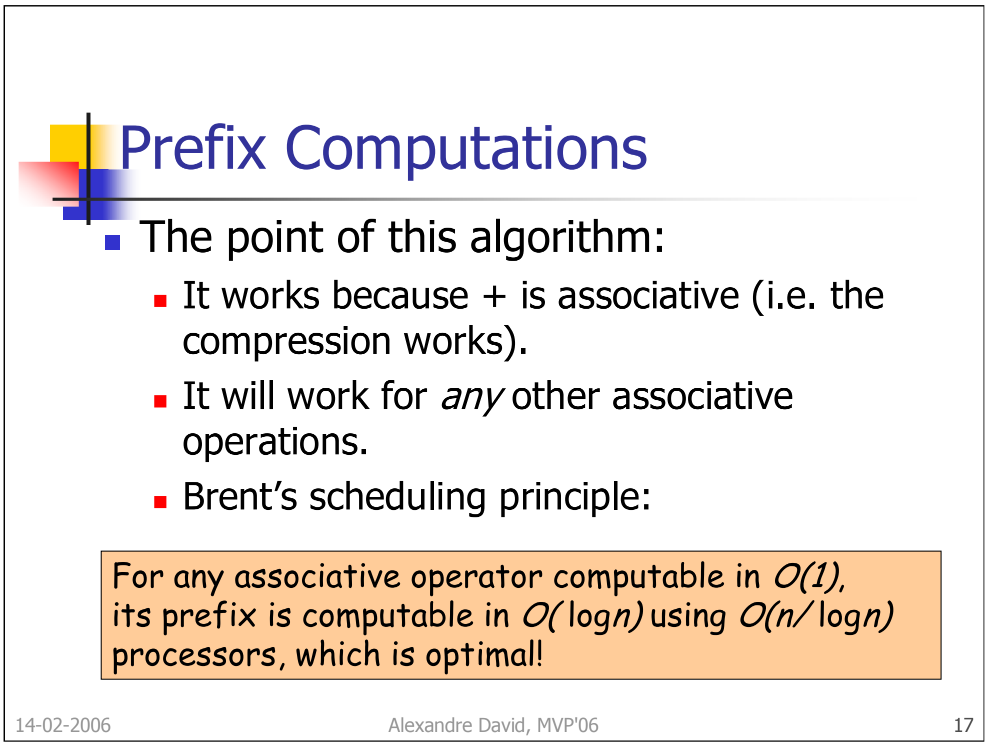


Parallel Prefix Computation

```
function prefix$^+$(A,n)[$p_1$,…,$p_n$]
    $p_1$: B[1] := A[1]
    if n > 1 then
        for i = 1 to n/2 pardo
            $p_i$: C[i]:=A[2i-1]+A[2i]
        od
        D:=prefix$^+$(C,n/2)[$p_1$,…,$p_{n/2}$]
        for i = 1 to n/2 pardo
            $p_i$: B[2i]:=D[i]
        od
        for i = 2 to n/2 pardo
            $p_i$: B[2i-1]:=D[i-1]+A[2i-1]
        od
    fi
    prefix$^+$:=B
end
```

14-02-2006 Alexandre David, MVP'06 16

Correctness: When the recursive call of prefix$^+$ returns then D[k]=sum(i:1..2k) A[i] (for $1 \le k \le n/2$). That comes from the compression algorithm idea.



Prefix Computations

- The point of this algorithm:
  - It works because + is associative (i.e. the compression works).
  - It will work for *any* other associative operations.
  - Brent's scheduling principle:

For any associative operator computable in *O(1)*, its prefix is computable in *O(logn)* using *O(n/ logn)* processors, which is optimal!

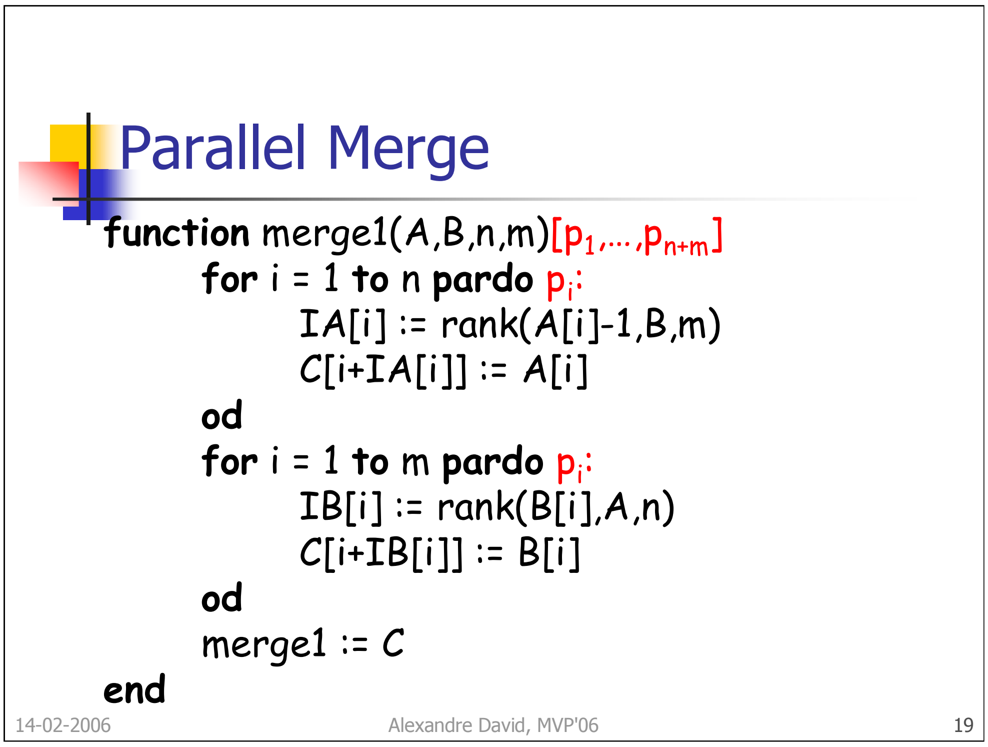
14-02-2006 Alexandre David, MVP'06 17

On a EREW-PRAM of course.

In particular initializing an array to a constant value…

Merging (of Sorted Arrays)

- Rank function:
  - rank(x,A,n) = 0 if x < A[1]
  - rank(x,A,n) = max{i | A[i] ≤ x}
  - Computable in time *O(* log*n)* by binary search.
- Merge A[1..n] and B[1..m] into C[1..n+m].
- Sequential algorithm in time *O(n+m)*.



Parallel Merge

```
function merge1(A,B,n,m)[$p_1$,…,$p_{n+m}$]
    for i = 1 to n pardo $p_i$:
        IA[i] := rank(A[i]-1,B,m)
        C[i+IA[i]] := A[i]
    od
    for i = 1 to m pardo $p_i$:
        IB[i] := rank(B[i],A,n)
        C[i+IB[i]] := B[i]
    od
    merge1 := C
end
```

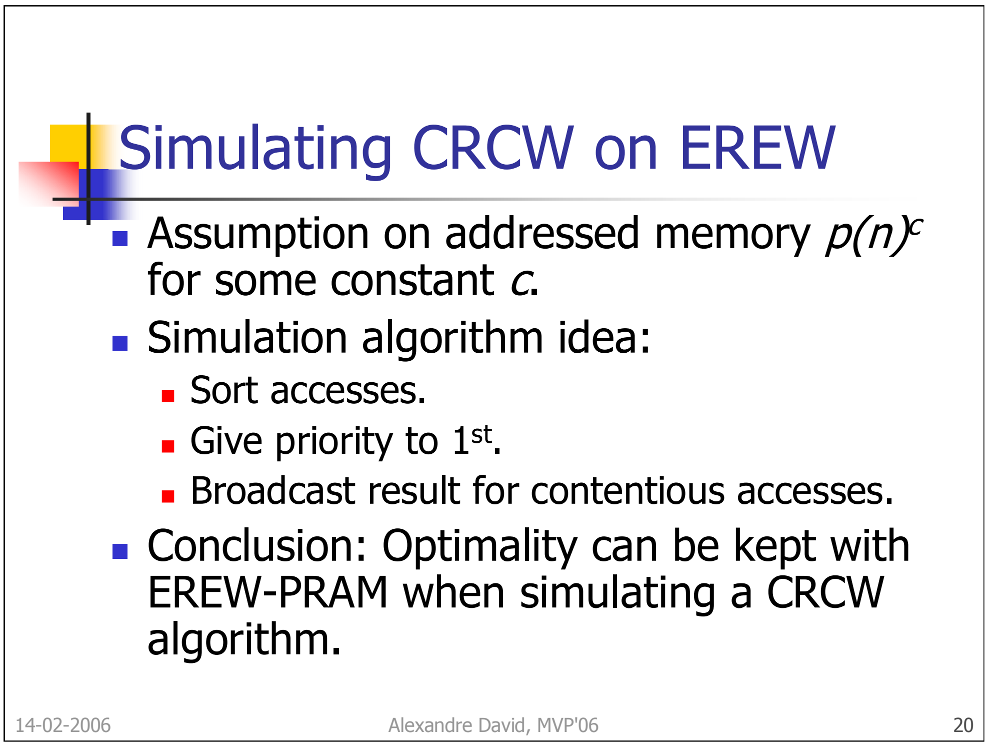
On CRCW-PRAM.

Compute indices for A[i] and compute indices for B[i] in parallel. Indices found by computing the rank of the elements. Dominating factor is the rank so this runs in *O(* log*(n+m))*. Not optimal, you see why?

However we could use processors p_{i+n} for the 2nd loop (and we would have to rewrite this so that we have all processors doing something), which is not suggested by the report but it does not change much (we still have (n+m)*log(n+m)).

The more complicated version proposed in the report is optimal, which means it's possible to merge arrays optimally.

Being more careful here we see that it's actually CREW-PRAM. If it is CRCW then it would write fewer elements than n+m and it would be wrong.

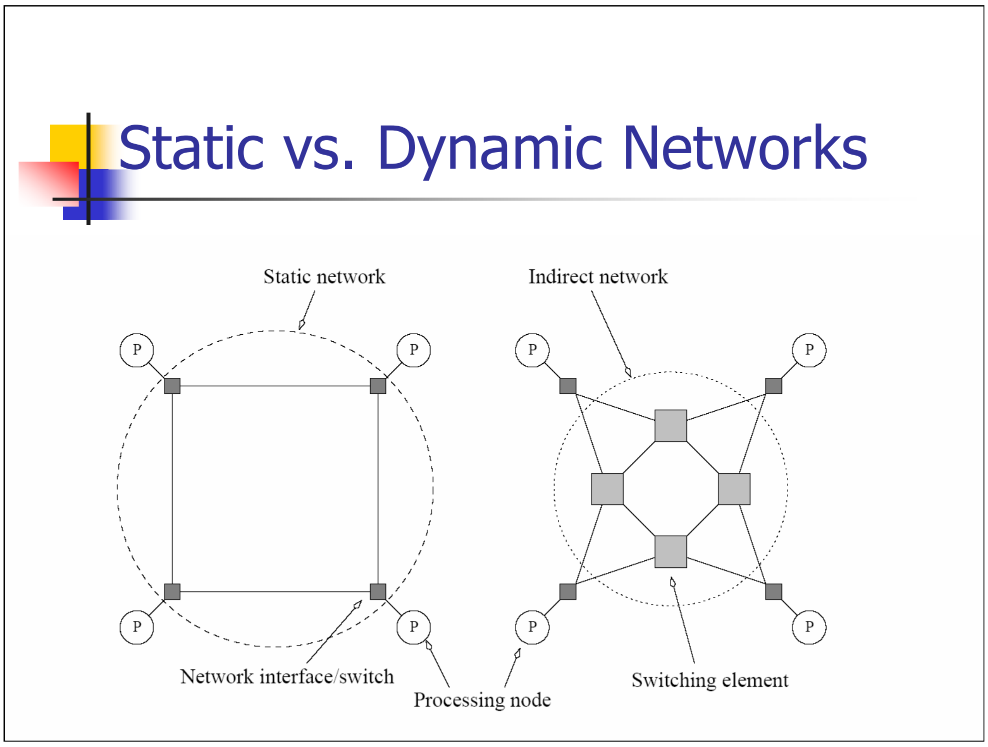


Simulating CRCW on EREW

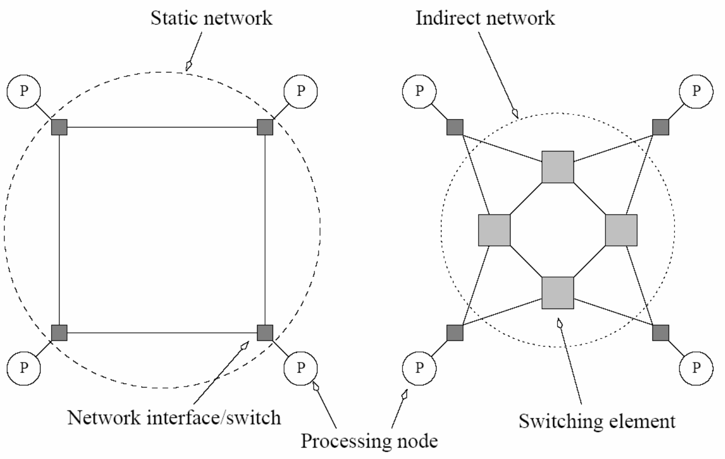
- Assumption on addressed memory $p(n)^c$ for some constant c.
- Simulation algorithm idea:
  - Sort accesses.
  - Give priority to 1st.
  - Broadcast result for contentious accesses.
- Conclusion: Optimality can be kept with EREW-PRAM when simulating a CRCW algorithm.

14-02-2006 Alexandre David, MVP'06 20

Read the details in the report. Remember the idea and the result.



Static vs. Dynamic Networks



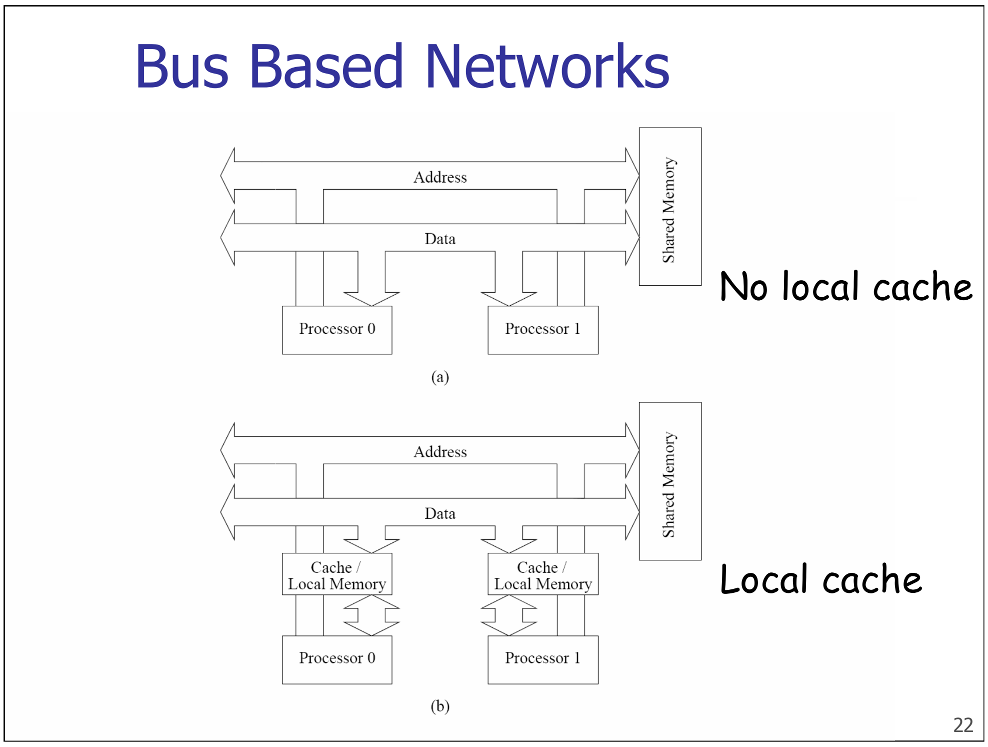
Interconnection networks built using links and switches. How to connect:

•Static networks, or direct networks, have p2p *static* communication links.

•Dynamic networks, or indirect networks, have switches to *route* the communication.

Number of (output) ports on a switch = degree. Mapping input -> output implemented by different technologies.

Static networks to connect processors <-> processors, dynamic networks to connect processors <-> memory.



Bus Based Networks

Good:

•Cost scales linearly with the number of nodes.

•The distance between all the nodes is constant.

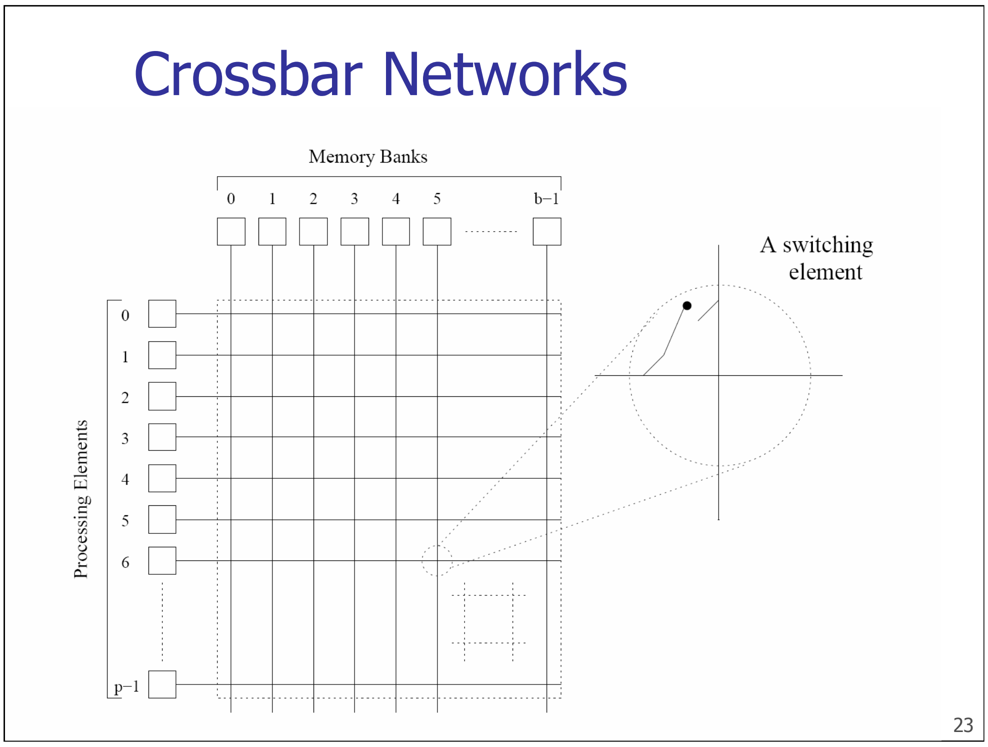
•It is ideal for broadcasting.

Bad:

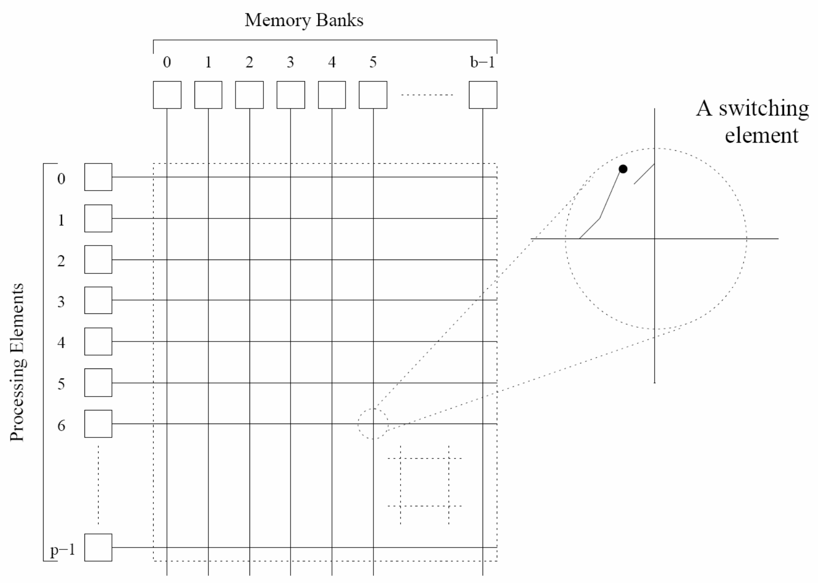
•Shared bandwidth between all the nodes -> bottleneck in performance.

In practice bus based only for small SMP (Intel). Caches are only a trick to reduce bandwidth consumption on the bus (not to reduce bandwidth as stated in the book).

Both for processors & memory.



Crossbar Networks

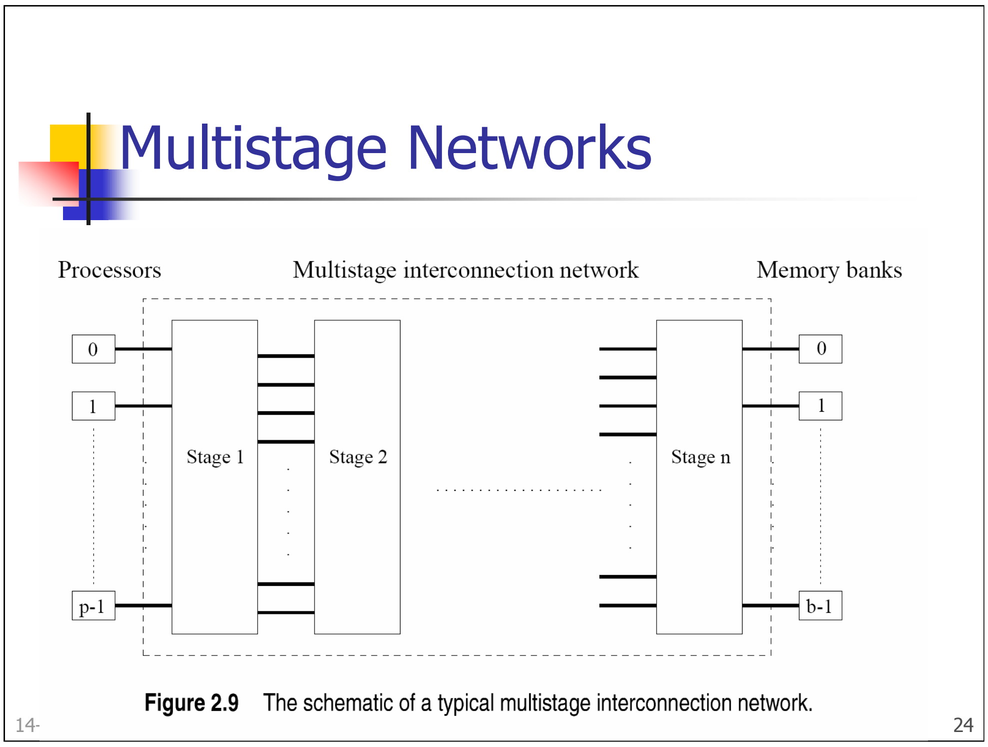


23

Grid to connect *p* processors to *b* memory banks. Non blocking in the sense that a connection (routing) does not block the connection of any other processing node, in contrast to multistage networks.

Good: scalable in performance (non blocking).

Bad: number of switches = *p*b*, not scalable in cost.



Multistage Networks

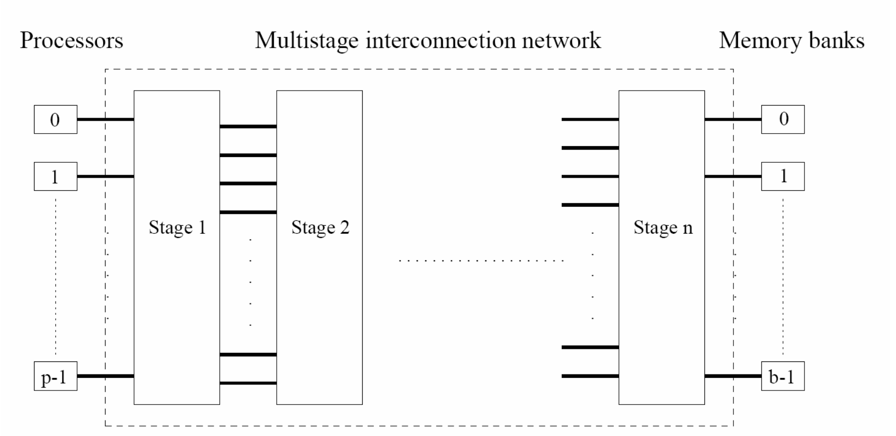


Figure 2.9 The schematic of a typical multistage interconnection network.

14-

24

Intermediate network, between crossbar and bus.

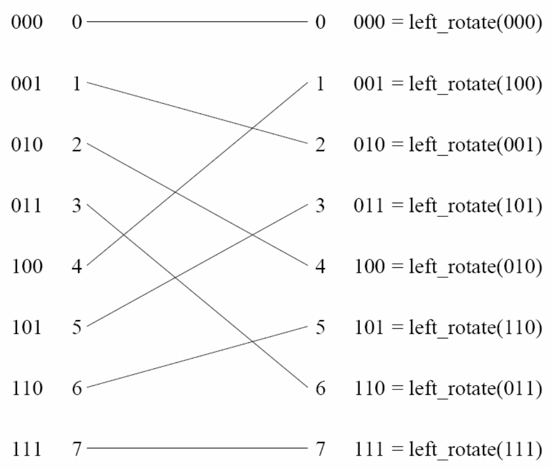
Again *p* processing nodes and *b* memory banks.

A common type is the **omega** network:

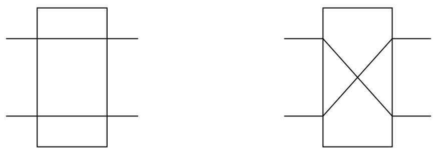
•It has log*p* stages (with matching number of inputs and outputs).

•It has a perfect shuffle interconnection pattern, easy with left rotate.

Perfect Shuffle Pattern


000 0 — 0 000 = left_rotate(000)
001 1 — 1 001 = left_rotate(100)
010 2 — 2 010 = left_rotate(001)
011 3 — 3 011 = left_rotate(101)
100 4 — 4 100 = left_rotate(010)
101 5 — 5 101 = left_rotate(110)
110 6 — 6 110 = left_rotate(011)
111 7 — 7 111 = left_rotate(111)

Switches in Omega Networks



Configurations: pass-through and cross-over.

p/2 * log p switching nodes:
log p stages, p/2 inputs & outputs.

Omega Network

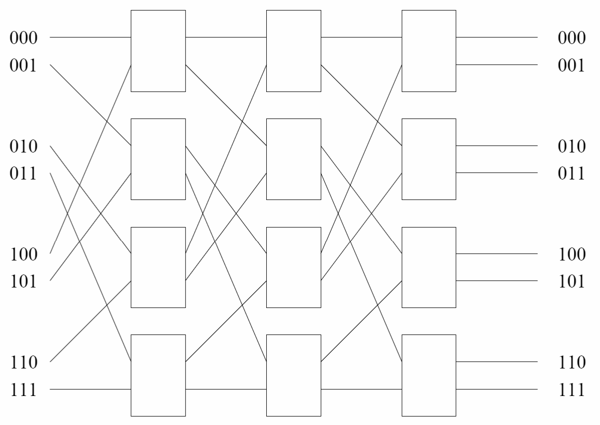


Figure 2.12 A complete omega network connecting eight inputs and eight outputs.

Blocking in Omega Networks

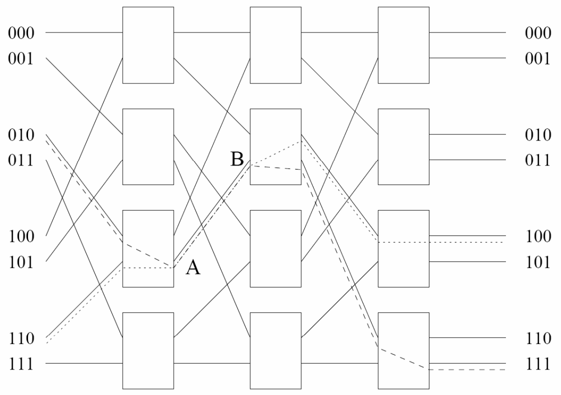
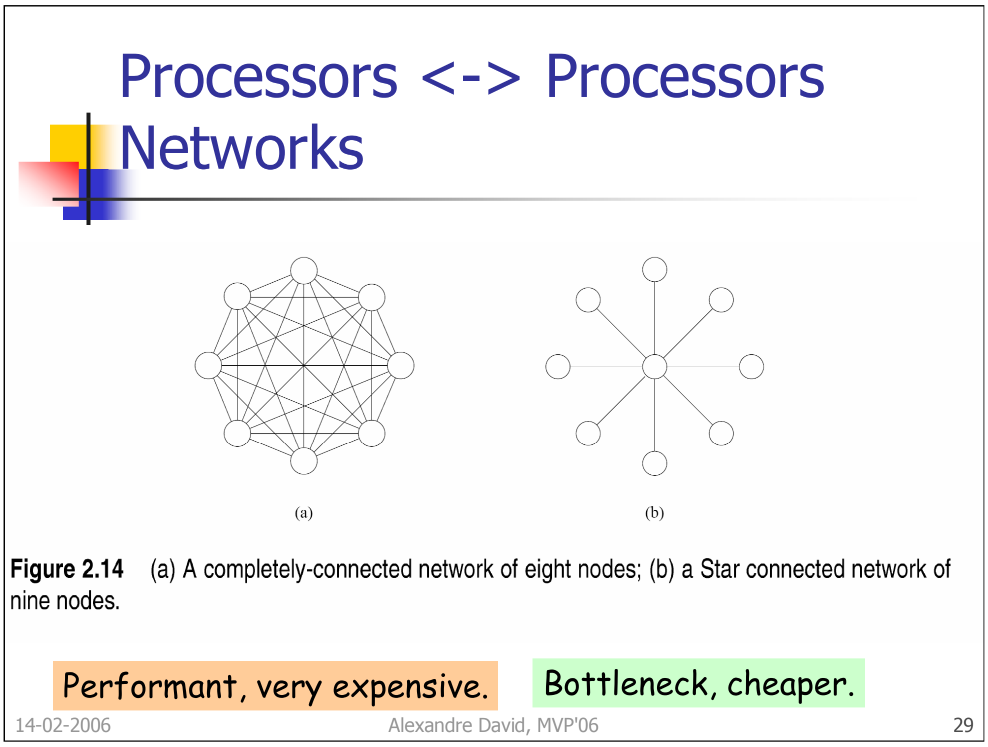


Figure 2.13 An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

Contention in the access, one is blocked. Such networks are called *blocking* networks.

So far processor <-> memory.



Processors <-> Processors Networks

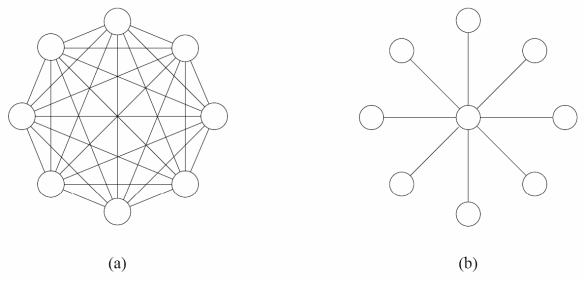


Figure 2.14 (a) A completely-connected network of eight nodes; (b) a Star connected network of nine nodes.

Performant, very expensive.

Bottleneck, cheaper.

14-02-2006 Alexandre David, MVP'06 29

Number of edges: n(n-1)/2 vs. n-1.

Linear Arrays and Meshes

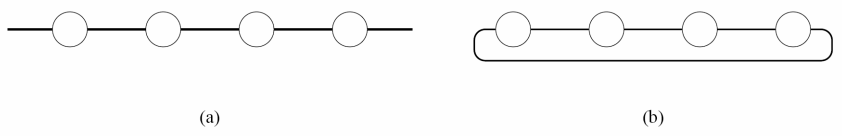


Figure 2.15 Linear arrays: (a) with no wraparound links; (b) with wraparound link.

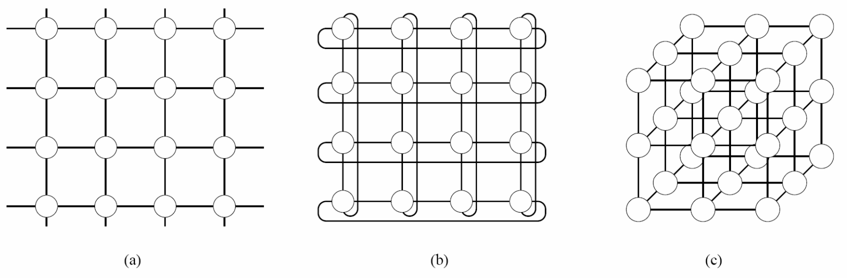


Figure 2.16 Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

Wrap around changes the number of neighbors and distance for some nodes.

Linear array: each node has 2 neighbors (except start & end). It becomes a ring (or 1-D torus) with wraparound.

2-D mesh has ***p*** processors so the dimension is given by *sqrt(p)*. Every node (except on the border) has 4 neighbors. Attractive from a wiring point of view. Adding wraparound links gives a 2-D torus.

3-D, similarly. Every time we add a dimension, we add 2 neighbors. 3-D meshes good for physical simulations because they correspond to the modeled problem and the way processing is distributed.

Hypercubes

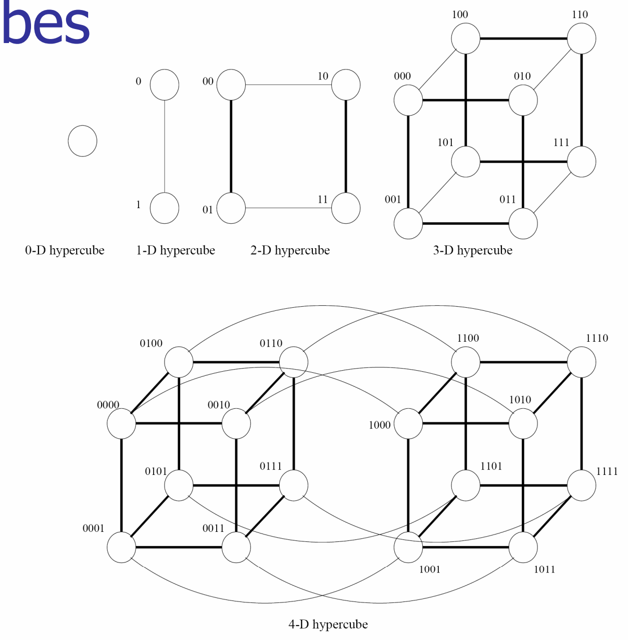


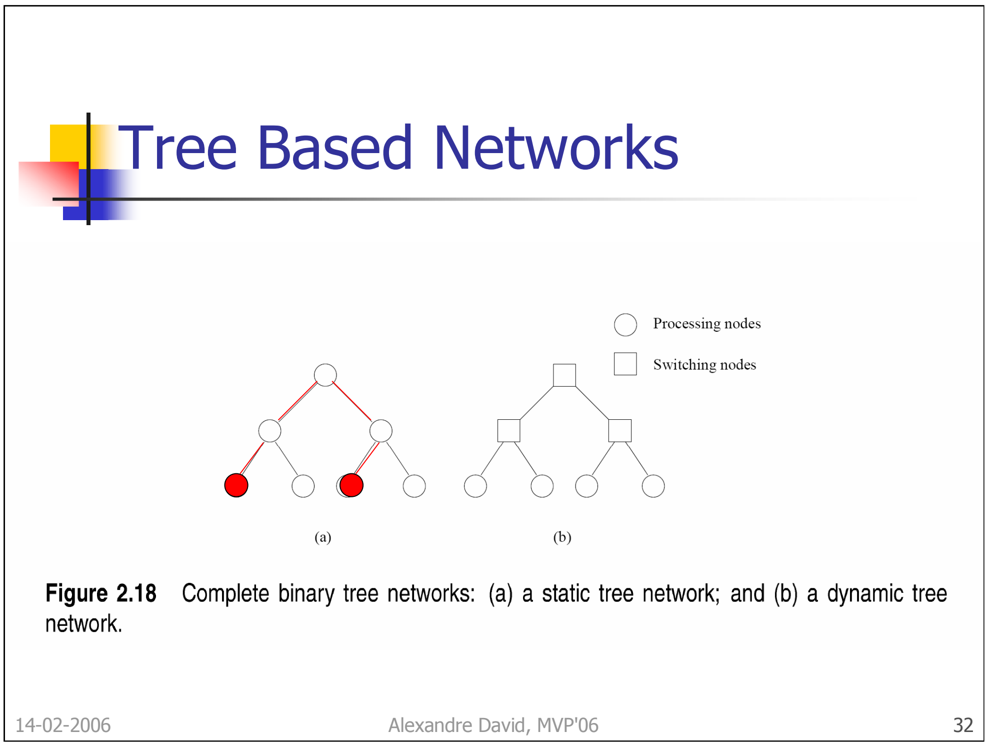
Figure 2.17 Construction of hypercubes from hypercubes of lower dimension.

14-02-2006 31

Hypercubes are the other extreme of linear meshes. 2 nodes per dimension and log *p* dimensions. Remember *p* processing nodes. Number of nodes for hypercube topology = $2^{\text{dimension}} = p$.

Very important: the clever way to distribute the indices. Remember this, it’s useful to derive parallel algorithms running on hypercubes.

Property: minimum distance between two nodes = number of different bits in the two indices (how many dimensions we need to cross).



Tree Based Networks

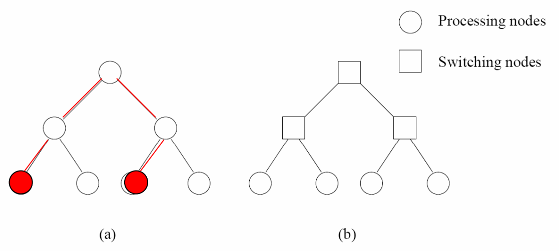


Figure 2.18 Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

14-02-2006 Alexandre David, MVP'06 32

(a) Some nodes share their connections for other nodes.

Routing for sending a message: Go up in the tree until it reaches a sub-tree that contains the destination, then go down. Performance in function of the height of the tree *O(* log *p)*.

Issue with communication: Nodes (or switches) up in the tree may be bottlenecks w.r.t. bandwidth. Fat trees: alternative to give more bandwidth to shared routes.

Fat Trees

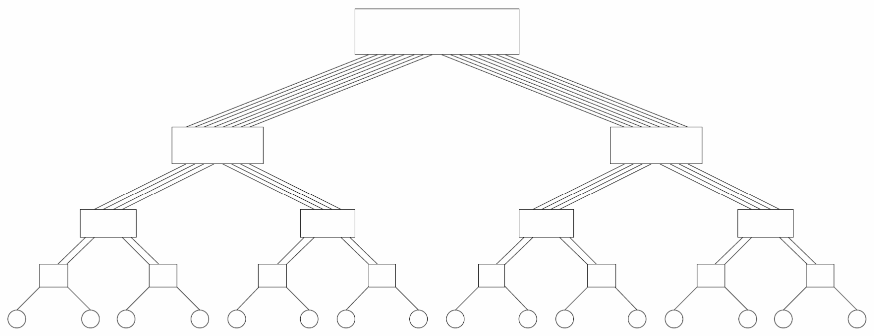
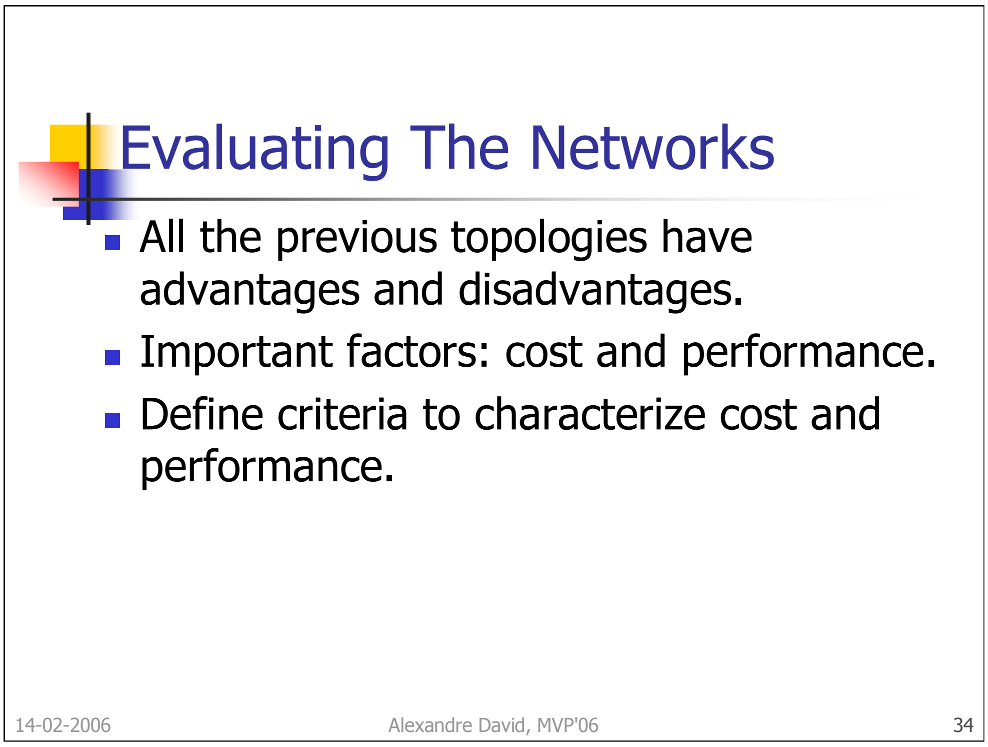


Figure 2.19 A fat tree network of 16 processing nodes.

More bandwidth where we need it.

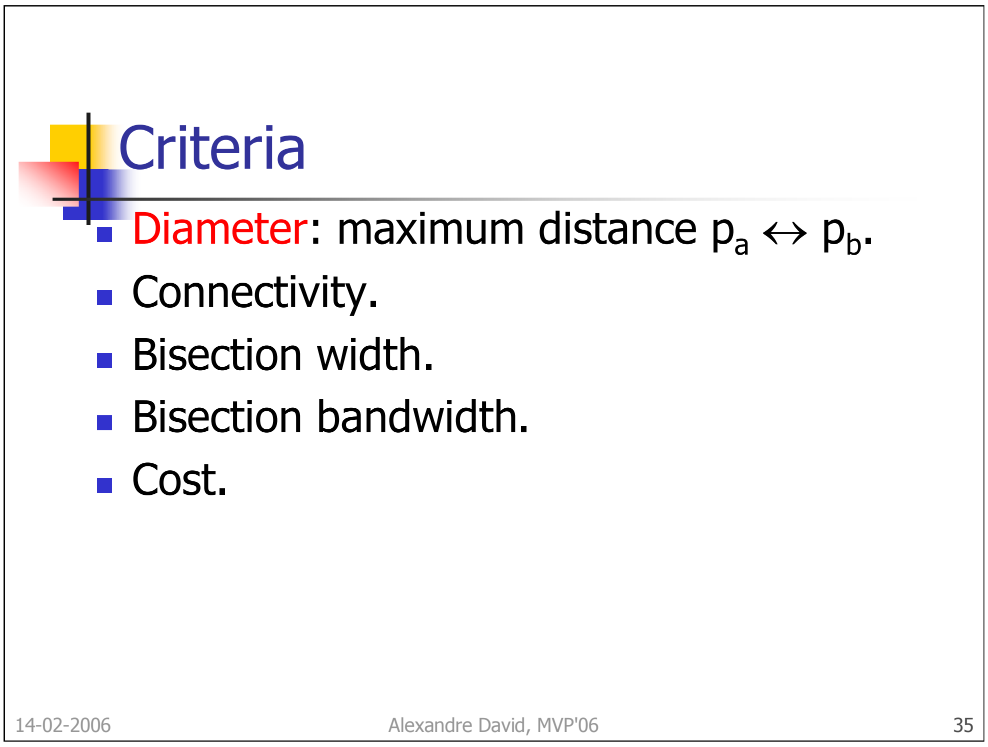


Evaluating The Networks

- All the previous topologies have advantages and disadvantages.
- Important factors: cost and performance.
- Define criteria to characterize cost and performance.

14-02-2006 Alexandre David, MVP'06 34

Your turn: Give suggestions on measure criteria.



Criteria

- Diameter: maximum distance $p_a \leftrightarrow p_b$.
- Connectivity.
- Bisection width.
- Bisection bandwidth.
- Cost.

14-02-2006 Alexandre David, MVP'06 35

Distance = shortest path between 2 nodes.
Diameter: How far 2 nodes may be.

•Completely connected: 1.

•Star connected: 2.

•Ring: floor(p/2).

•2-D mesh without wraparound: 2(dim-1). With wraparound: 2*floor(dim/2).
Note: dim = sqrt(*p*).

•Hypercube: dim (=log *p*).

•Complete binary tree: height=h, $p=2^{h+1}-1$, h =log($(p+1)/2$), travel $2h$.

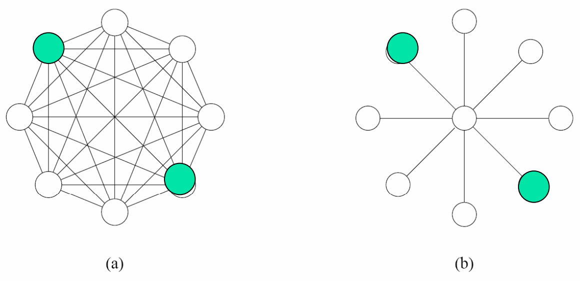


Figure 2.14 (a) A completely-connected network of eight nodes; (b) a Star connected network of nine nodes.

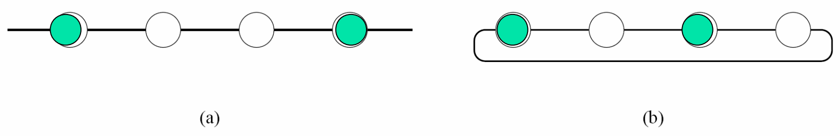


Figure 2.15 Linear arrays: (a) with no wraparound links; (b) with wraparound link.

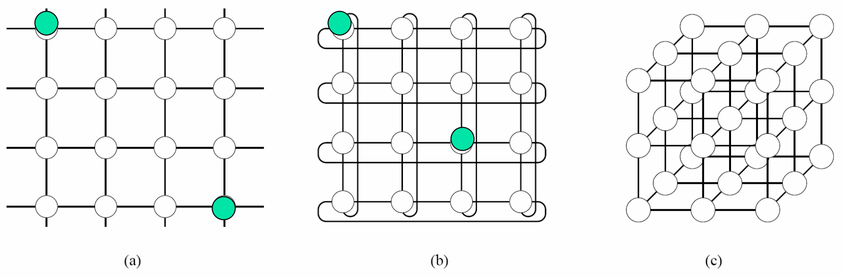


Figure 2.16 Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

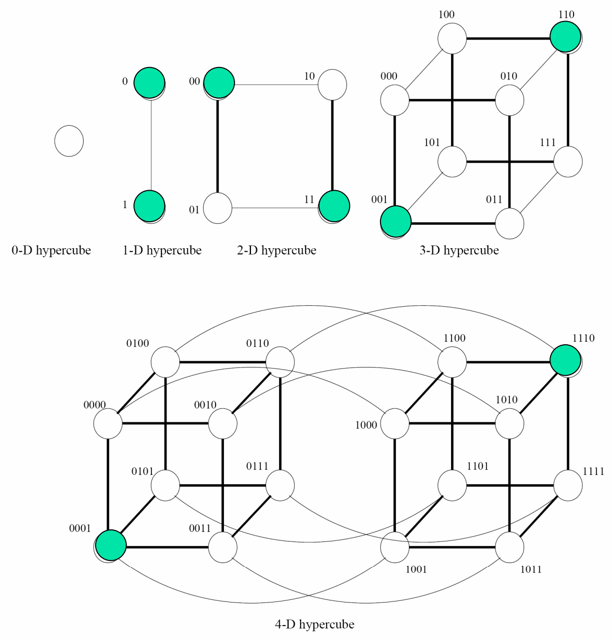


Figure 2.17 Construction of hypercubes from hypercubes of lower dimension.

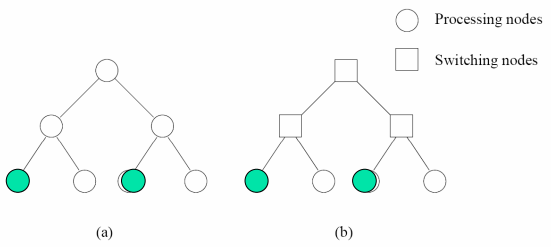
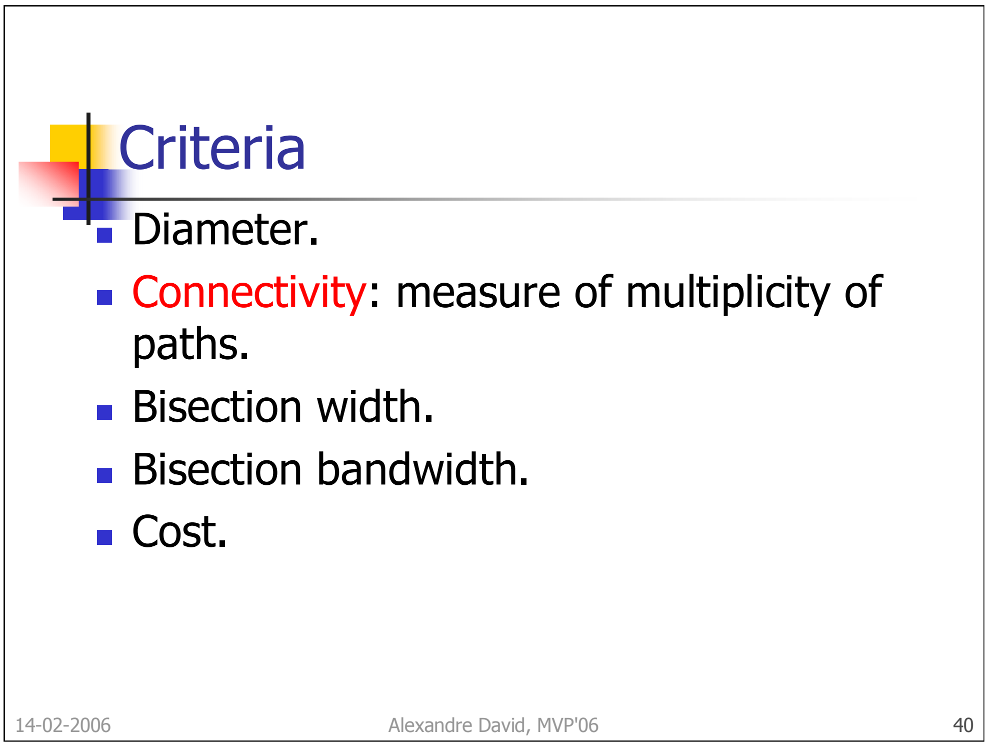


Figure 2.18 Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.



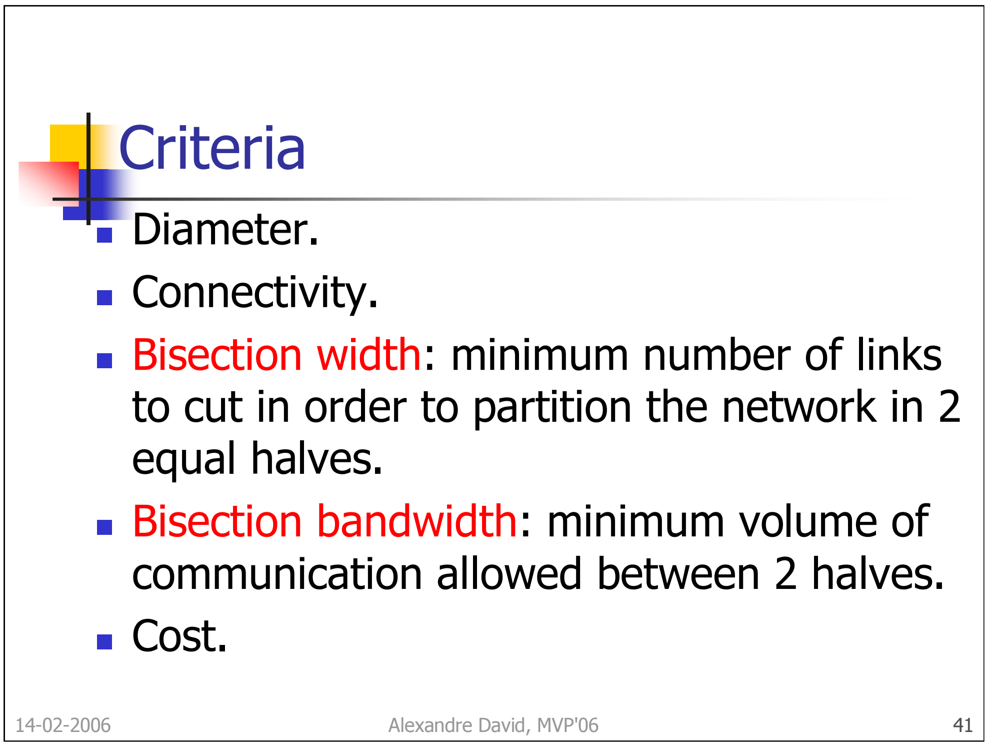
Criteria

- Diameter.
- Connectivity: measure of multiplicity of paths.
- Bisection width.
- Bisection bandwidth.
- Cost.

14-02-2006 Alexandre David, MVP'06 40

High connectivity to lower contention and avoid congested networks.

With or without wraparound gives different results (consider min). Interesting to remember: Connectivity is *d* for *d*-dimensional hypercubes.



Criteria

- Diameter.
- Connectivity.
- Bisection width: minimum number of links to cut in order to partition the network in 2 equal halves.
- Bisection bandwidth: minimum volume of communication allowed between 2 halves.
- Cost.

14-02-2006 Alexandre David, MVP'06 41

Bisection width measures the weakness/strength of the network, overall connectivity.

•Ring: 2.

•2-D mesh without wraparound: dim (=sqrt(p)); with wraparound: 2*dim.

•Completely connected network: needs to cut half of the edges = $p^2/2$.

•Hypercubes: how we construct… double nodes every time, so cut p/2 nodes.

Number of bits per link = channel bandwidth = channel rate (peak bit rate) * channel width (number of wires).

Bisection bandwidth also called cross-section bandwidth.

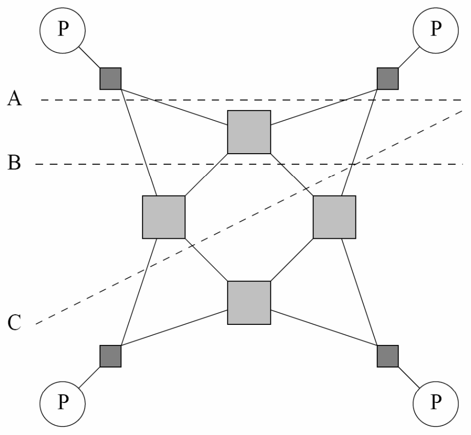
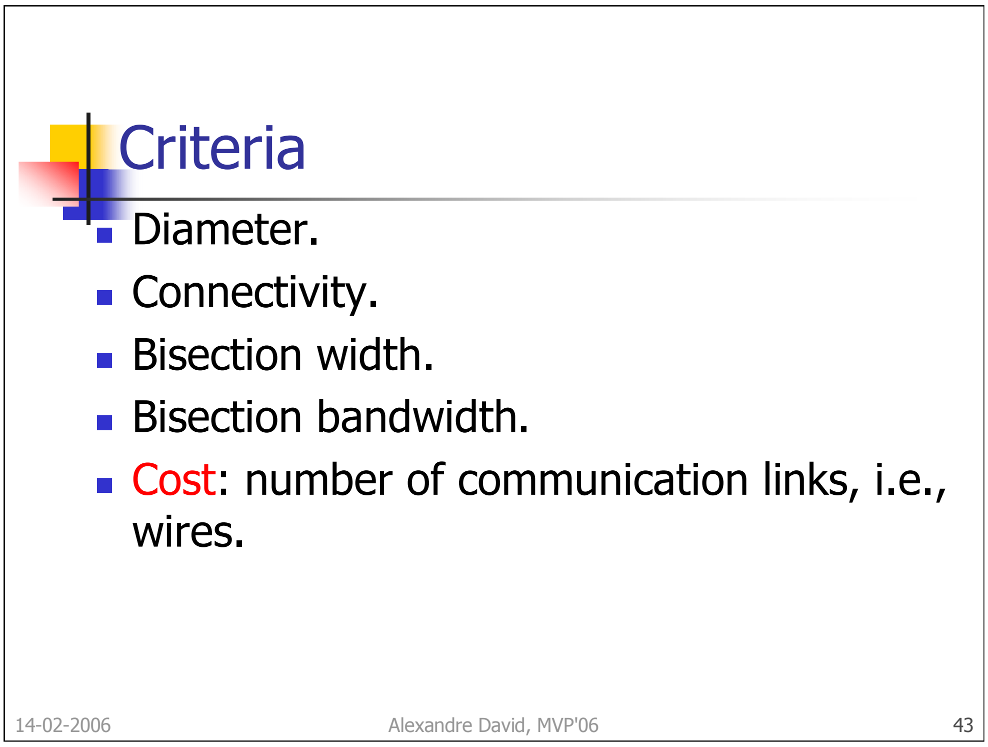


Figure 2.20 Bisection width of a dynamic network is computed by examining various equipartitions of the processing nodes and selecting the minimum number of edges crossing the partition. In this case, each partition yields an edge cut of four. Therefore, the bisection width of this graph is four.



Criteria

- Diameter.
- Connectivity.
- Bisection width.
- Bisection bandwidth.
- Cost: number of communication links, i.e., wires.

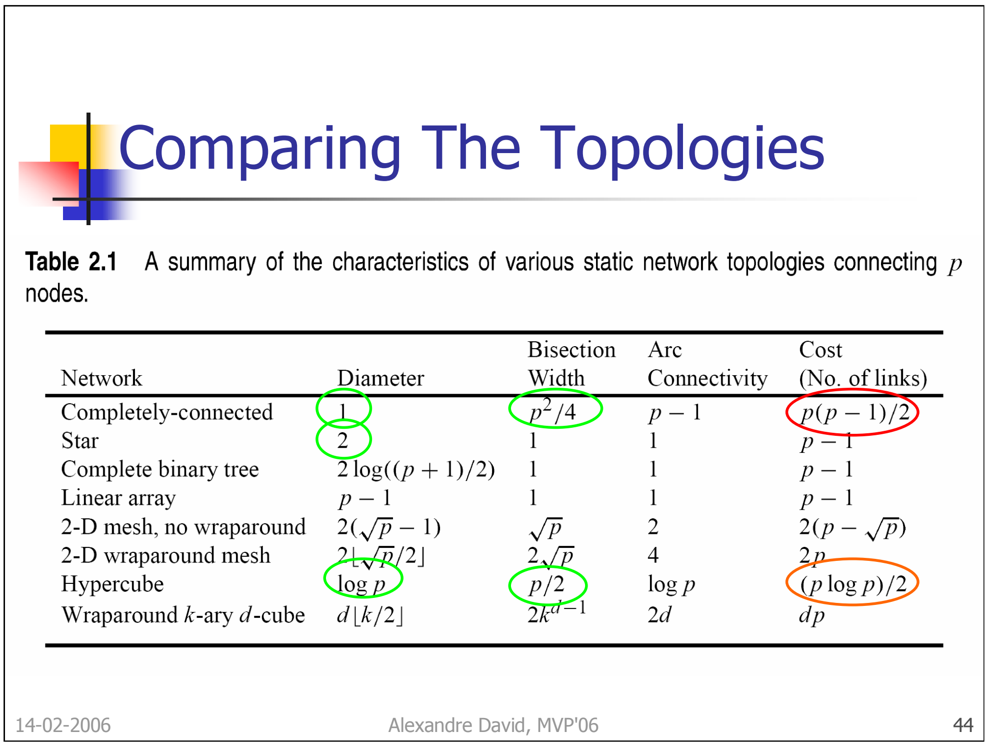
14-02-2006 Alexandre David, MVP'06 43

Number of wires:

•Linear arrays and trees: *p-1*.

•D-dimensional mesh: D**p*.

•Hypercube: *p**dim/2 (with dim = log *p*). Connectivity is dim.



Comparing The Topologies

Table 2.1 A summary of the characteristics of various static network topologies connecting p nodes.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Cache Coherence Protocols

- We need additional hardware to keep *multiple copies* of the same memory bank *consistent* with each other.
- We have seen that $$ is good but it does not come for free.
- Mechanism known as cache coherence protocol, usually described as state machines.

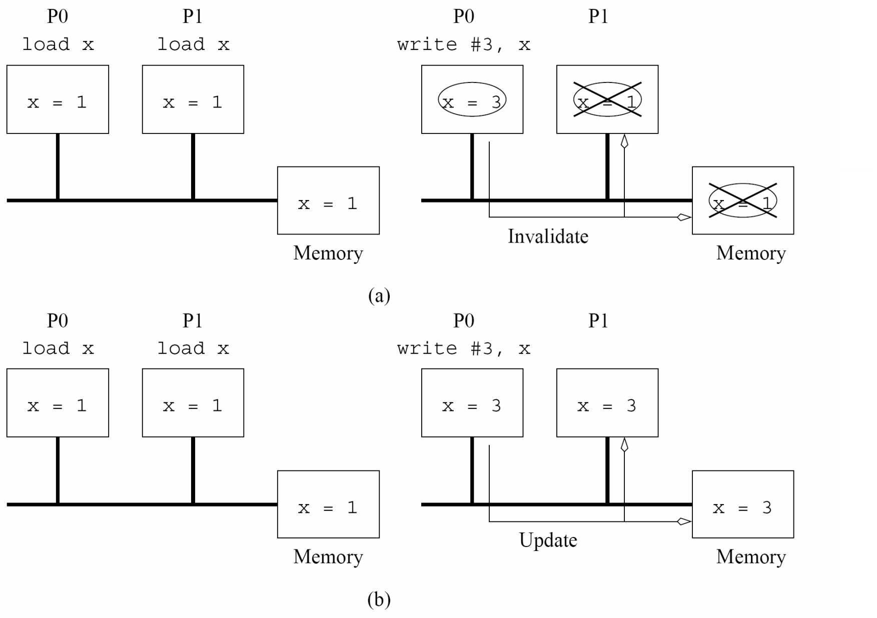


Figure 2.21 Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

46

2 principles: invalidate other copies or update other copies. 1st is cheap in terms of bandwidth.

Another factor that you don't see here: **false sharing**. Imagine y next to x on the same cache line. You will invalidate y as well. Degraded performance if different processors update different parts of the cache line: the cache line becomes shared although the variables are not.

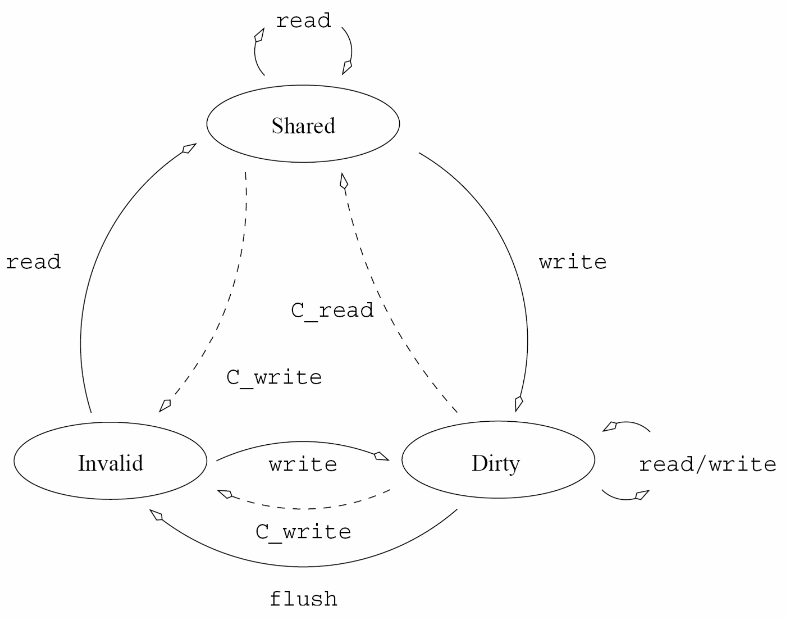
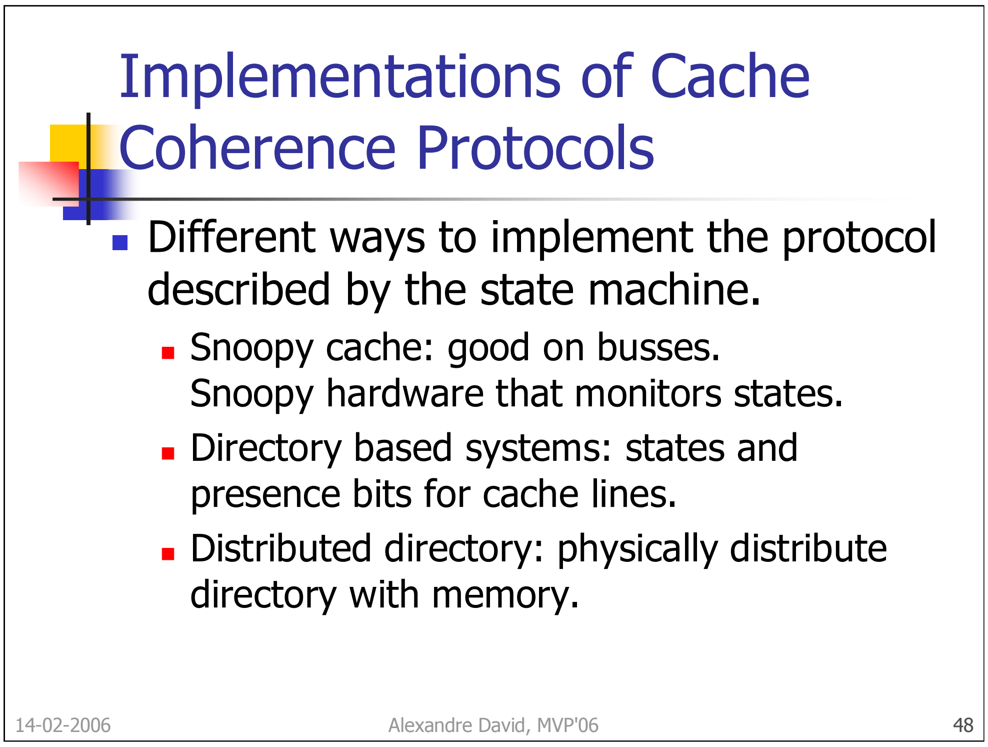


Figure 2.22 State diagram of a simple three-state coherence protocol.

47

One example, may define other types of machines with more states. 4 common.

UPPAAL demo.



Implementations of Cache Coherence Protocols

- Different ways to implement the protocol described by the state machine.
  - Snoopy cache: good on busses.
    Snoopy hardware that monitors states.
  - Directory based systems: states and presence bits for cache lines.
  - Distributed directory: physically distribute directory with memory.

Snoopy is popular.

Directory based is expensive.

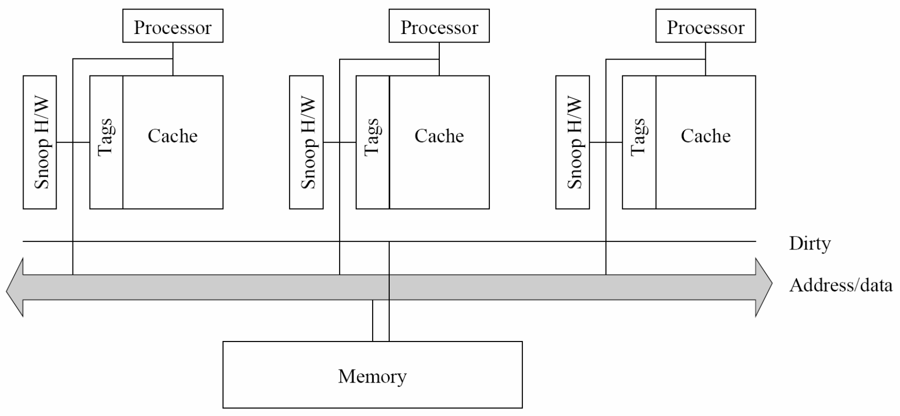


Figure 2.24 A simple snoopy bus based cache coherence system.

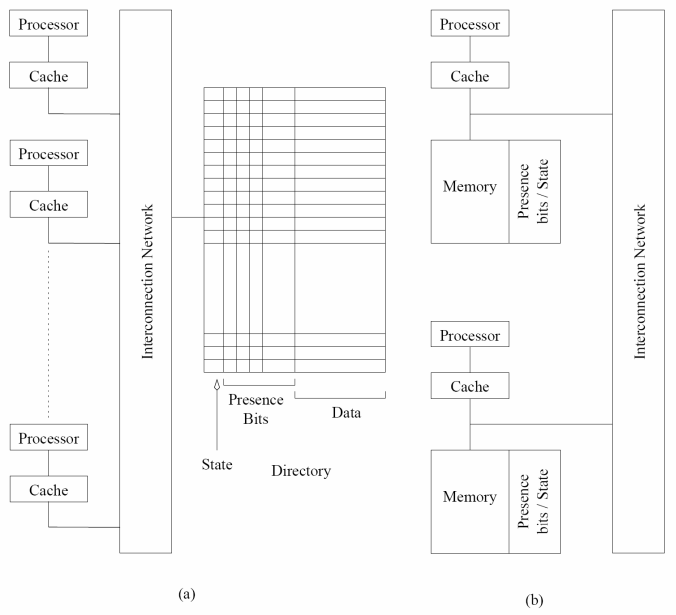


Figure 2.25 Architecture of typical directory based systems: (a) a centralized directory; and (b) a distributed directory.