Lecture 2: Models of Parallel Computations

Lecture Outline

(Reading: Chapter 2 of textbook)

```
1. The RAM model for sequential computation
2. The PRAM model of parallel computation
3. PRAM Algorithms
4. Reducing the number of processors
5. Problems that defy fast solutions on PRAM
```

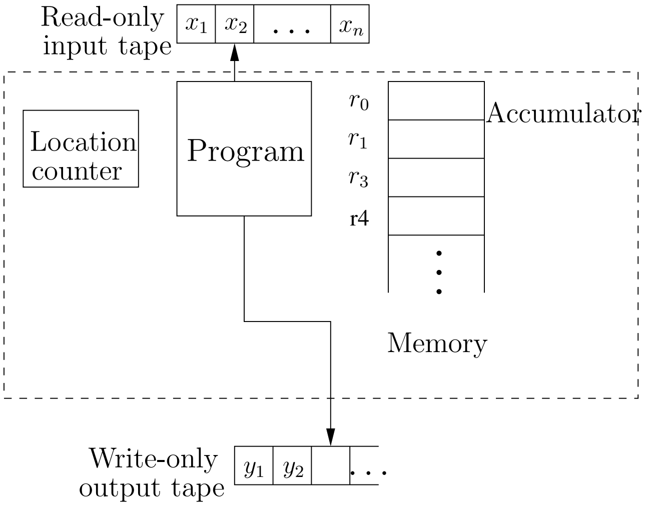


Figure 14: The random access machine (RAM) model of serial computation (Aho, Hopcroft, and Ullman 1974.) (Fig.2-1, p.26 of textbook)

1. The random access machine (RAM) – a model of sequential computation.

(1) A RAM consists of a memory, a location counter, a read-only input tape, a write-only output tape, and a program (Fig.2-1, p.26).

- Program is not stored in memory and can not be modified.
- The input tape contains a sequence of integers. Every time an input value is read, the input head advances one square.
- The output head also advances after writing an integer.
- The memory consists of an unbounded sequence of registers, $r_0, r_1, \cdots$. Each register can hold an integer.
- Register r_0 is the accumulator, where the computations are performed.

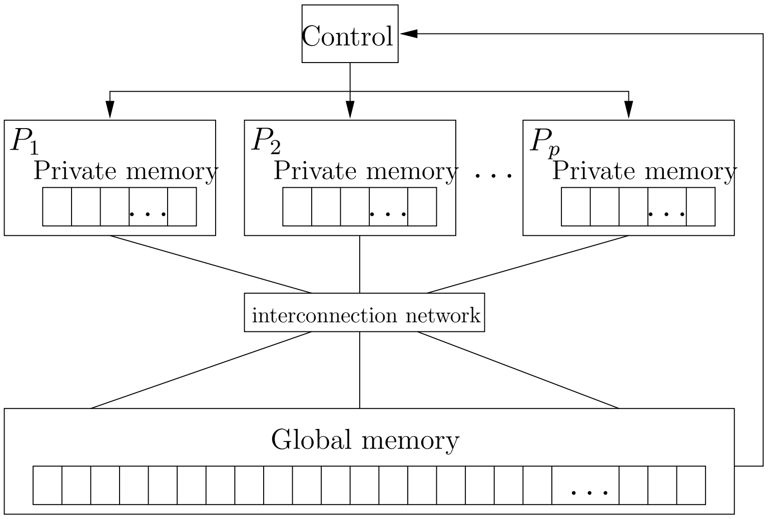


Figure 15: The PRAM model of parallel computation (Fig.2-2, p.27 of textbook)

(2) Example operations are: load, store, add, subtract, mulitiplication, division, read, write, test, jump, and halt.

Operation code	Address
1. LOAD	operand
2. STORE	operand
3. ADD	operand
4. SUB	operand
5. MULT	operand
6. DIV	operand
7. READ	operand
8. WRITE	operand
9. JUMP	label
10. JGTZ	label
11. JZERO	label
12. HALT	

(3) Computational complexity of RAM model

- **Worst-case time and space complexity** of a RAM program: the maximum time (space), over all inputs of size n, taken by the program to execute.
- **Expected time and space complexity** of a RAM program: the average time (space), over all inputs of size n, taken by the program to execute.
- **Uniform cost criterion:** Each RAM instruction takes 1 time unit, and each RAM register requires 1 unit of space.

- **Logarithmic cost criterion:** it takes into account that an actual word of memory has a limited storage capacity. The uniform cost criterion is appreciate if the values manipulated always fit into one computer word.

(4) Concepts of computational complexity: O, Ω, and Θ

- O. $O(f(n)) : \exists$, read "order at most $f(n)$", upper bound, means at most. $O(f(n))$ is the set S of all functions s.t. for each function $g(n) \in S$ there exist a constant $c > 0$, an integer $n_0 \geq 0$, s.t. $\forall n \geq n_0$, $g(n) \leq cf(n)$.
- Ω: $\Omega(f(n))$, read "order at least $f(n)$", lower bound, means at least. $\Omega(f(n))$ is the set S of all functions s.t. for each function $g(n) \in S$ there exist a constant $c > 0$, an integer $n_0 \geq 0$, s.t. $\forall n \geq n_0$, $g(n) \geq cf(n)$.
- Θ: $\Theta(f(n))$, read "order exactly $f(n)$", exact bound, means exactly. $\Theta(f(n))$ is the set S of all functions s.t. for each function $g(n) \in S$ there exist a constant $c_1, c_2 > 0$, an integer $n_0 \geq 0$, s.t. $\forall n \geq n_0$, $c_1 f(n) \leq g(n) \leq c_2 f(n)$.

2. The PRAM model of parallel computation

(1) A PRAM consists of (a) *a control unit*, (b) *global memory*, and (c) *an unbounded set of processors, each with its onw private memory* (Fig.2-2,p.27).

- Active processors execute identical instructions, and every processor has a unique index. The value of the index can be used to disable or enable the processor or influence which memory location it accesses.
- A PRAM begins with the input stored in global memory and a single active processor. During each step of the computation, an active and enabled processor may read a value from a single private or global memory location, perform a single RAM operation, write into one local or global memory location.
- Alternatively, during a computation step, a processor may activate another processor.
- All active and enabled processors must execute the same instruction, albeit on different memory locations.
- The computation terminates when the last processor halts.

(2) Cost of a PRAM computation (Definition 2-1, p.27).

> **Definition**. The **cost** of a PRAM computation is the product of the parallel time complexity and the number of processors used. For example, a PRAM algorithm that has time complexity $\Theta(\lg p)$ using p processors has cost $\Theta(p \lg p)$.

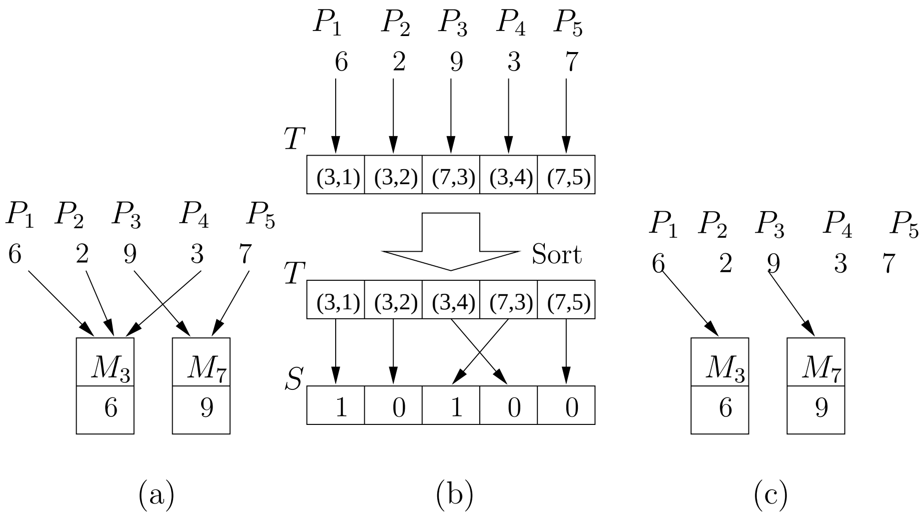


Figure 16: A concurrent write operation, which takes constant time on a p-processor PRIORITY PRAM, can be simulated in $\Theta(\log p)$ time on a p-processor EREW PRAM. (a) Concurrent write on the PRIORITY PRAM model. Processor P_1, P_2 and P_4 attempt to write values to memory location M_3. Processor P_1 wins. Processor P_3 and P_5 attempt to write values to memory location M_7. Processor P_3 wins. (b) To simulate concurrent write on the EREW PRAM model. each processor writes an (address, processor number) pair to a unique element of T. The processors sort T in time $\Theta(\log\ p)$. In constant time processors can set 1 to those elements of S corresponding to the winning processors. (c) Winning processors write their values. (Fig.2-3, p.29 of textbook)

(3) Various PRAM models: differing in how they handle read/write conflicts to the global memory.

- EREW (exclusive read, exclusive write): Read or write conflicts are not allowd.
- CREW (concurrent read, exclusive write): the default PRAM model.
- CRCW (concurrent read, concurrent write): Which write succeeds? a variety of CRCW models exist with different policies for handling concurrent writes to the same global memory location:
    * COMMON: All processors concurrently writing into the same global address must write the same value.
    * ARBITRARY: an arbitrary one is chosen as the "winner" – its value is written into the register.
    * PRIORITY: Some PPU's have higher priority (lower index) will succeed.

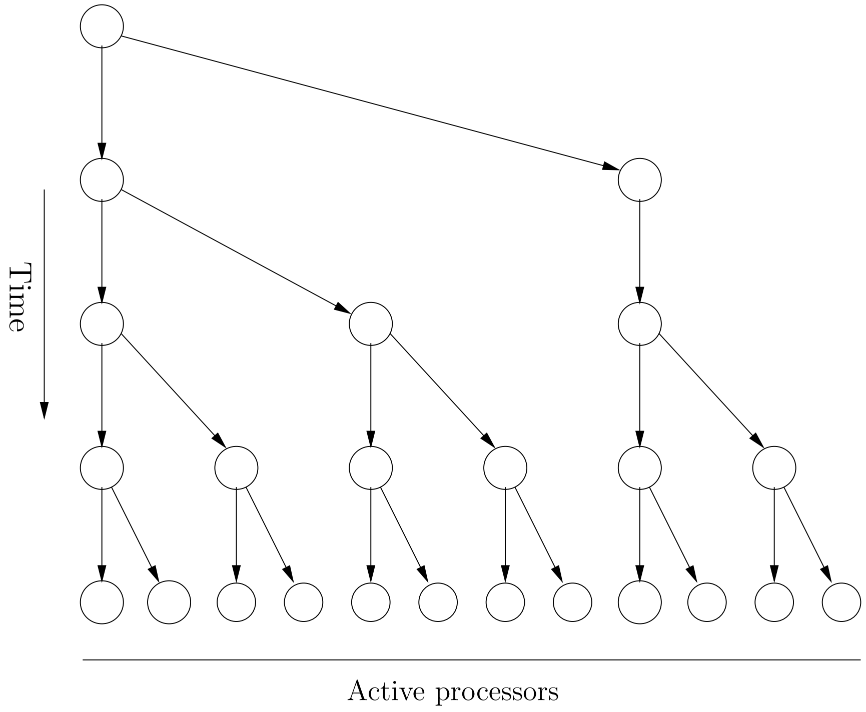


Figure 17: Exactly $\lceil \log p \rceil$ procesors activation steps are necessary and sufficient to change from 1 active processor to p active processors (Fig.2-4,p.30)

(3) Relative strengths of various PRAM models:

- EREW PRAM is weakest
- CREW PRAM is 2nd weakest
- CRCW PRAM is strong
- PRIORITY CRCW PRAM is the strongest

(4) **Lemma 2.1** (Cole [1988]).

A p-processor EREW PRAM can sort a p element array stored in global memory in $\Theta(\lg p)$ time.

(5) **Theorem 2.1** (Eckstein [1970]). p.29

A p-processor PRIORITY PRAM can be simulated by a p-processor EREW PRAM with the time complexity increased by a factor of $\Theta(\lg p)$.

Proof: p.29. Fig.2-3 for illustration.

a. Fig.2-3(a): Concurrent write on the PRIORITY CRCW MODEL. The whole operation takes one time unit.

b. Fig.2-3(b): five processors P_1, P_2, P_3, P_4 and P_5 initiate write operations simutaneously.

   (a) Each processor writes a $(memory\_location, process\_index)$ pair into the array T.

   (b) Array T is sorted in $\Theta(\log p)$ time.

   (c) For each memory cell location, the processor whose index appears first in the sorted array T wins the right to write its value to that location.

   (d) The boolean array S is initialized as follows: if a processor i wins, S_i is set to value 1. Otherwise S_i has a value 0.

c. Fig.2-3(c): Each processor checks its values in array S. If S_i is 1, P_i writes its value.

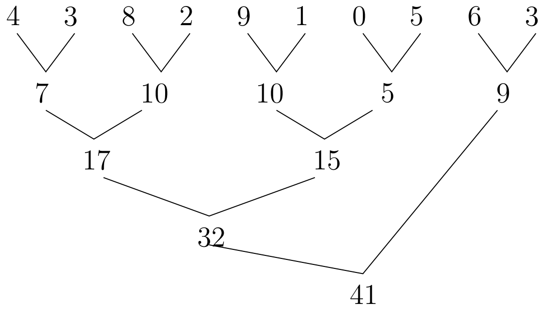


Figure 18: Parallel summation is an example of a reduction operation. Parallel reduction can be represented by a binary tree. A group of n values can be added in $\lceil \log n \rceil$ parallel addition steps (Fig.2-5,p.31)

3. PRAM Algorithms

(1) The Psuedo language (meta instructions)

- **spawn** (<processor name>)
- **for all** ... **do** ... **endfor**
- other constructs: **if**, **while**, **repeat**, assignments.

(2) Two characteristics of PRAM algorithms

a. There are two phases in PRAM algorithms: activation and full parallel computation. Logarithmic complexity (Fig.2-4,p.30).

b. Communication between processes are through the share global memory.

(3) **Parallel reduction** (Fig.2-5,2-6,2-7).

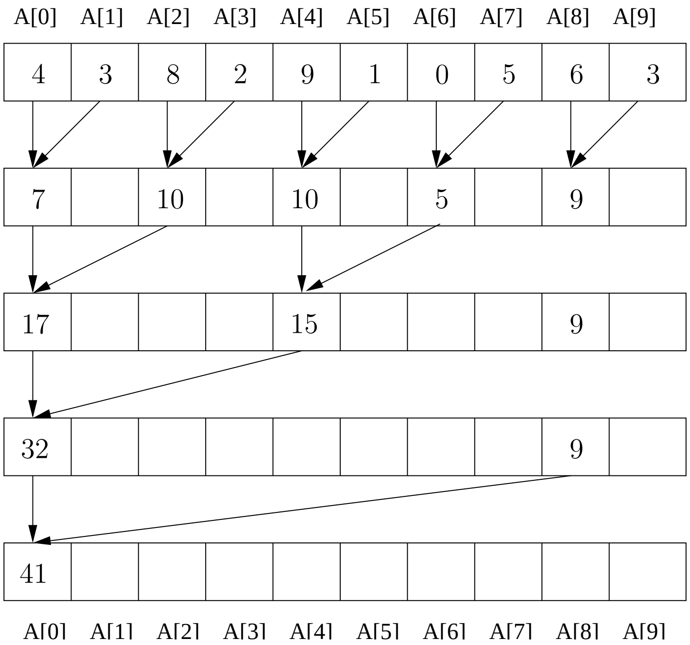

A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]
4 3 8 2 9 1 0 5 6 3
7 10 10 5 9
17 15 9
32 9
41
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9]

Figure 19: Example of operation of PRAM algorithm to find sum of 10 values (Fig.2-6,p.32)

a. Top-down: broadcast and divide-and-conquer

b. Bottom-up: reductions: parallel summation is an eg.

- ∗ Basic ideas of summarizing n integers by parallel reduction: Fig.2-5
- ∗ Illustration of array-based PRAM algorithm of summarizing n integers: Fig.2-6
- ∗ The algorithm (Fig.2-7)

```
SUM(EREW PRAM)
Initial condition: List of $n \geq 1$ elements stored in $A[0...(n-1)]$
Final condition: Sum of elements in $A[0]$
Global variables: $n, A[0...(n-1)], j$
begin
    spawn $(P_0, P_1, \cdots, P_{\lfloor n/2 \rfloor - 1})$.
    for all $P_i$ where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ do
        for $j \leftarrow 0$ to $\lceil \log n \rceil - 1$ do
            if $i$ modulo $2^j = 0$ and $2i + 2^j < n$ then
                $A[2i] \leftarrow A[2i] + A[2i + 2^j]$
            endif
        endfor
    endfor
```

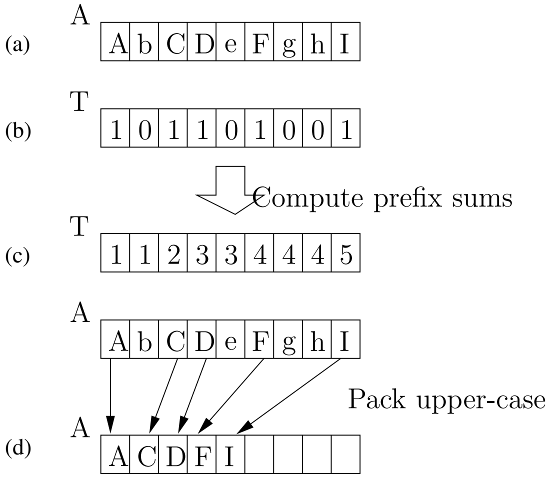


Figure 20: Packing elements is one application of prefix sums. (a) Array A contains both uppercase and lowercase letters. We want to pack uppercase letters into beginning of A. (b) Array T contains a 1 for every uppercase letter and a 0 for every lowercase letter. (c) Array T after prefix sums have been computed. For each element of A containing an uppercase letter, the corresponding element T is that element's index in the packed array. (d) Array A after packing (Fig.2-8,p.33)

end

Fig.2-7 EREW PRAM algorithm to sum n elements using $\lfloor n/2 \rfloor$ processors.

Note: The textbook assumes that the PRAM model is EREW. As just reading n numbers from global memory takes $\Theta(n)$ time under EREW model, the assumption is wrong. It should be CREW.

(4) **Prefix Sums** (Fig.2-8,2-9,2-10).

a. Problem description: given n values and an operator $\oplus$, compute all the sums

$$\begin{array}{l} a_1 \\ a_1 \oplus a_2 \\ \dots\ \dots \\ a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n \end{array}$$

The operator $\oplus$ can be addition $+$. Also called **parallel prefixes** and **scans**.

b. Applications: packing elements (Fig.2-8).

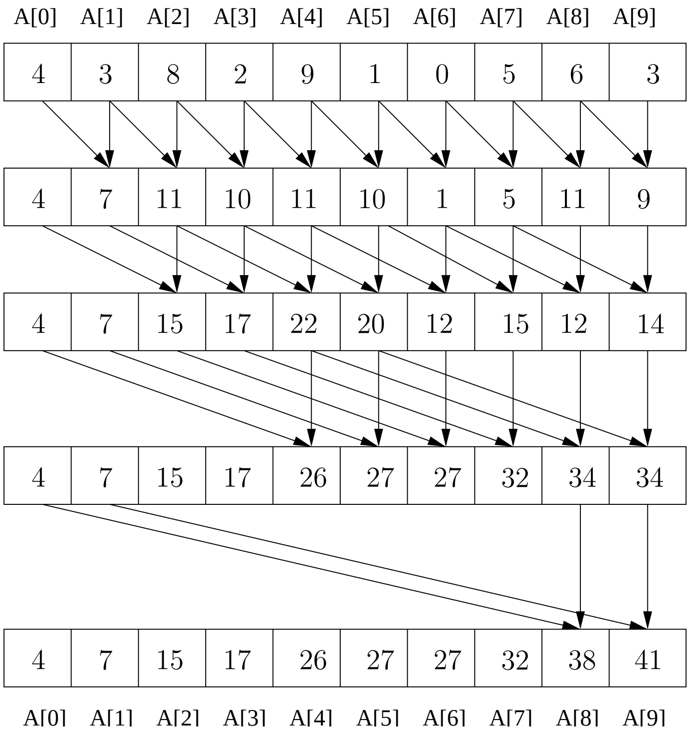


Figure 21: All prefix sums of a list of n values can be computed in $\lceil \log n \rceil$ addition steps on an EREW PRAM (Fig.2-10,p.34)

(a) An array A contains uppercase or lowercase letters. The goal is to move all uppercase letters to beginning of the array while maintaining their order.

(b) In Fig.2-8, a boolean array T is created $T[i] = 1$ if the value of $A[i]$ is an uppercase letter. Otherwise $T[i] = 0$.

   i. The prefix sums of array T are computed.

   ii. For each element $A[i]$ that contains an uppercase letter, the value of $T[i]$ is $A[i]$'s index in the packed array.

c. The algorithm: Fig.2-9.

```
PREFIX.SUMS(CREW PRAM):
Initial condition: List of $n \geq 1$ elements stored in $A[0...(n-1)]$
Final condition: Each element $A[i]$ contains $A[0] \oplus A[1] \oplus \cdots \oplus A[i]$
Global variables: $n, A[0...(n-1)], j$
begin
    spawn $(P_0, P_1, \cdots, P_{n-1})$.
    for all $P_i$ where $0 \leq i \leq n-1$ do
        for $j \leftarrow 0$ to $\lceil \log n \rceil - 1$ do
            if $i - 2^j \geq 0$ then
```

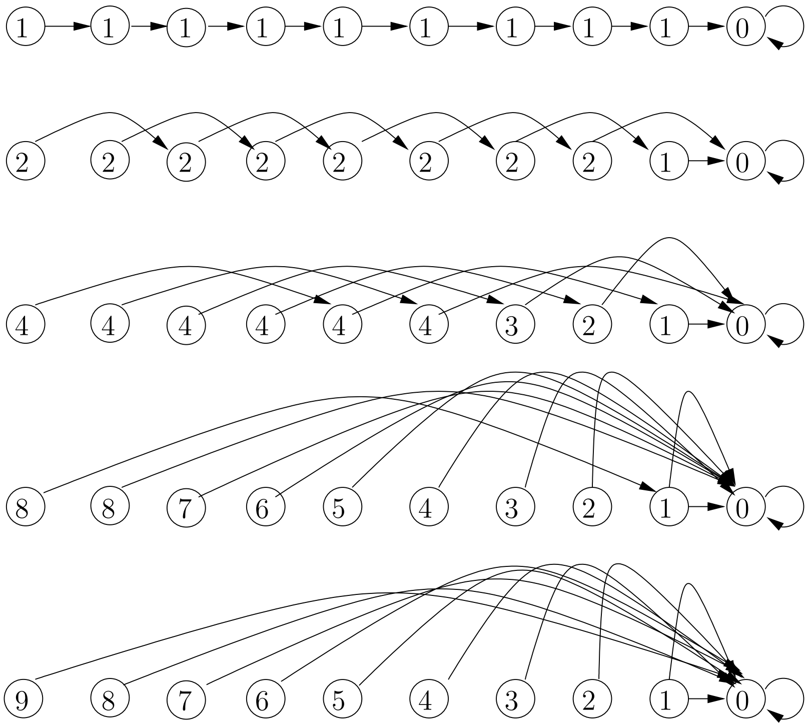


Figure 22: The position of each item on an n-element linked list can be determined in $\lceil \log n \rceil$ pointer-jumping steps (Fig.2-11,p.35)

```
            $A[i] \leftarrow A[i] + A[i - 2^j]$
         **endif**
      **endfor**
   **endfor**
**end**
```

Fig.2-9 PRAM algorithm to find prefix sum of an n-element list using $n - 1$ processors.

d. Illustration of the algorithm: Fig.2-10.

(a) The algorithm iterates $\lceil \log n \rceil$ times. The variable j goes from 0 to $\lceil \log n \rceil - 1$.

(b) Observation:

At the beginning of each iteration, $A[i]$ holds sum of 2^j elements:

$$A[i - 2^j + 1] + A[i - 2^j + 2] + \cdots + A[i]$$

(5) **List ranking** (Fig.2-11,2-12).

a. Problem description: **suffix sums** (similar to the prefix sums problem, using linked lists rather than arrays). A special case of the suffix sums (elements are either 0 or 1) is called the list ranking problem.

b. Example operations of a PRAM algorithm for listing ranking (Fig.2-11)

c. The algorithm: Fig.2-12.

```
LIST.RANKING(CREW PRAM):
Initial condition: Values in array $next$ represent a linked list
Final condition: Values in array $position$ contain original distance of each
                 element from end of list.
Global variables: $n, position[0...(n-1)], next[0...(n-1)], j$
begin
    spawn $(P_0, P_1, \cdots, P_{n-1})$.
    for all $P_i$ where $0 \le i \le n-1$ do
        if $next[i] = i$ then $position[i] \leftarrow 0$
        else $position[i] \leftarrow 1$
        endif
        for $j \leftarrow 1$ to $\lceil \log n \rceil$ do
            $position[i] \leftarrow position[i] + position[next[i]]$
            $next[i] \leftarrow next[next[i]]$
        endfor
    endfor
end
```

Fig.2-12 PRAM algorithm to compute, for each element of a singly-linked list, its distance from the end of the list.

(6) **Preorder tree traversal** (Fig.2-13,2-14,2-15).

a. Sequential preorder traversal (p.36)

```
PREORDER.TRAVERSAL($nodeptr$):
begin
    if $nodeptr \neq$ null then
        $nodecount \leftarrow nodecount + 1$
        $nodeptr.label \leftarrow nodecount$
        PREORDER.TRAVERSAL($nodeptr.left$)
        PREORDER.TRAVERSAL($nodeptr.right$)
    if
end
```

b. Investigating the potential parallelism in the sequential algorithm

   (a) Is this algorithm inherently sequential? On the surface, it appears to be. We cannot assign labels in the right subtree first.

   (b) Study the edge visiting order: each edge is visited twice, i.e. the sequential preorder traversal algorithm passes it twice, one downward and another upward.

   (c) Based on above observation, we can imagine dividing each edge into two edges, one corresponds to the downward traversal and another to the upward traversal. (Fig.2-13(b)). This change effectively transforms the preorder traversal problem to the problem of traversing a singly linked list, which can be solved by efficient parallel algorithm.

c. Illustration of the algorithm: four phases (Fig.2-13,p.37).

   Phase 1: constructs a singly-linked list (Fig.2-13(c));

   Phase 2: assign weights to vertices of the newly created linked lists: 1 with downward edges and 0 with upward edges. In the preorder traversal algo, a vertex (except the root) will be labeled as soon as it is encountered via a downward edge traversal.

   Phase 3: for each element of the singly-linked list, the rank of that element is computed by using the list-ranking algorithm.

   Phase 4: because in sequential preorder traversal nodes are only visited when traversing downward, processors associated with downward edges (those bold elements in Fig.2-13(d)) use the ranks to assign a preorder traversal number to their associated tree nodes. More specifically, if a downward edge, say (i, j) has a rank value k, then node j is the kth node from the end of preorder traversal list. If the tree has n nodes, then the preorder traversal lable for node C is $n - k + 1$.

d. Tree representation: three arrays (Fig.2-14).

   (a) For every tree node with index i, the node's parent, immediate sibling to the right, and the leftmost child, are stored in three arrays *parent, sibling*, and *child*.

   (b) Example: node B is the second node. Then $\mathit{parent}[2] = A$, the parent node of B; $\mathit{sibling}[2] = C$, the immediate sibling on the right of B; and $\mathit{child}[2] = D$, the leftmost child node of B.

e. PRAM code (Fig.2-15,p.39)

```
PREORDER.TREE.TRAVERSAL(CREW PRAM):
Global  $n$                          {Number of vertices in tree}
        $parent[1...n]$              {Vertex number of parent node}
        $child[1...n]$               {Vertex number of child node}
        $sibling[1...n]$             {Vertex number of sibling}
        $succ[1...2(n-1)]$           {Index of successor edge}
        $position[1...2(n-1)]$       {Edge rank}
        $preorder[1...n]$            {Preorder traversal number}
begin
    spawn (set of all $P(i,j)$ and $P(j,i)$ where $(i,j)$ is an edge)
    for all $P(i,j)$ and $P(j,i)$ where $(i,j)$ is an edge do
        {Put the edges into a linked list}
        if $parent[i] = j$ then
            if $sibling[i] \neq$ null then
                $succ[(i,j)] \leftarrow (j, sibling[i])$
            else if $parent[j] \neq$ null then
                $succ[(i,j)] \leftarrow (j, parent[j])$
            else
                $succ[(i,j)] \leftarrow (i,j)$
                $preorder[j] \leftarrow 1$   {$j$ is root of tree}
            endif
        else
            if $child[j] \neq$ null then $succ[(i,j)] \leftarrow (j, child[j])$
            else $succ[(i,j)] \leftarrow (j,i)$
            endif
        endif
        {Assign position values}
        if $parent[i] = j$ then $position[(i,j)] \leftarrow 0$
        else $position[(i,j)] \leftarrow 1$
        endif
        {Perform suffix sum on successor list}
        for $k \leftarrow 1$ to $\lceil \log(2(n-1)) \rceil$ do
            $position[(i,j)] \leftarrow position[(i,j)] + position[\mathrm{succ}(i,j)]]$
            $succ[(i,j)] \leftarrow succ[succ[(i,j)]]$
        endfor
        {Assign preorder values}
        if $i = parent[j]$ then $preorder[j] \leftarrow n + 1 - position[(i,j)]$
        endif
    endfor
end
```

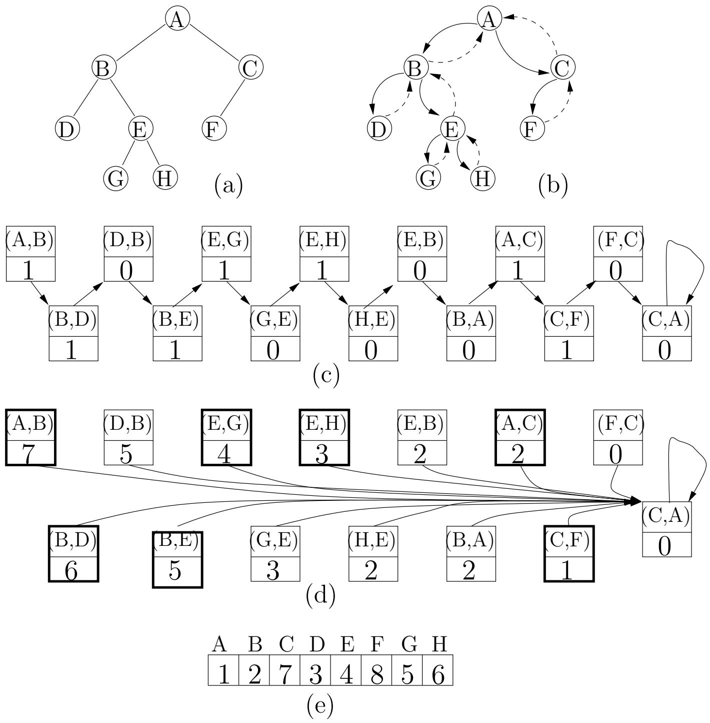


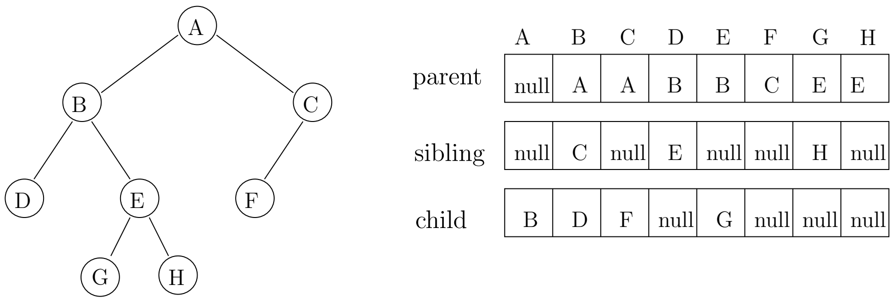
Figure 23: Preorder traversal of a rooted tree. (a) Tree. (b) Double tree edges, distinguishing downward edges from upward edges. (c) Build linked list out of directed tree edges. Associate 1 with downward edges and 0 with upward edges. (d) Use pointer jumping to compute total weight from each vertex to end of list. Bold elements of linked list correspond to downward edges. Processors managing these elements assign preorder values. For example, element (E, G) has weight 4, meaning tree node G is fourth node from end of preorder traversal. The tree has 8 nodes, so we can compute that tree node G has label 5 in preorder traversal. (e) Preorder traversal values. (Fig.2-13,p.37)

Fig.2-15 PRAM algorithm to label the nodes of a tree according to their position in a preorder traversal.

(7) **Merging two sorted lists** (Fig.2-16,2-17).

a. Many PRAM algorithms achieve low time complexity by performing more operations than optimal RAM algorithms – merging two sorted lists is a good eg.

b. Optimal RAM algorithm for merging: $\Theta(n)$ time (at most n-1 comparisons to merge two lists of size n/2).



	A	B	C	D	E	F	G	H
parent	null	A	A	B	B	C	E	E
sibling	null	C	null	E	null	null	H	null
child	B	D	F	null	G	null	null	null

Figure 24: We can represent a rooted tree with a data structure that stores, for every tree node, the node's parent, immediate sibling to the right, and the leftmost child (Fig.2-14,p.38)

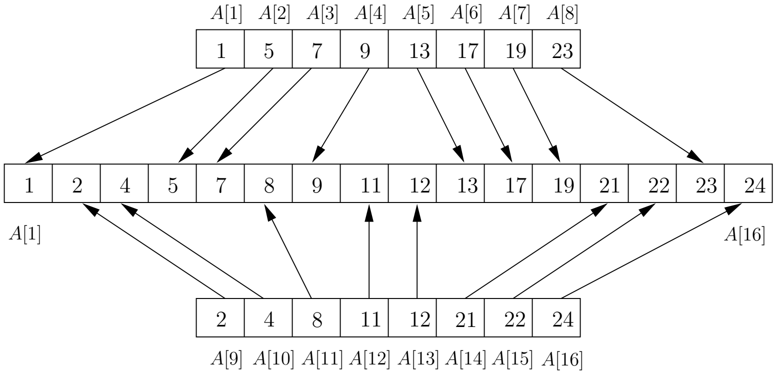


Figure 25: Two lists having $n/2$ elements each can be merged in $\Theta(n)$ time. (Fig.2-16,p.40)

c. PRAM algorithm: $\Theta(\lg n)$ complexity. Each element in the two lists is assigned a processor.

d. It tries to find the index of its element in the other list (it knows its index on its own list already) using binary search.

e. Assumption: values of the two lists are disjoint.

f. The PRAM code (Fig.2-17,p.41). The repeat loop implements the binary search.

g. Note: the final statement

$$A[high + i - n/2] \leftarrow x$$

will assign x, which holds the original value in $A[i]$ to array location $high + i - n/2$.

MERGE.LISTS(CREW PRAM):

```
Given:     Two sorted lists of $n/2$ elements each, sorted in
           $A[1] \cdots A[n/2]$ and $A[n/2+1] \cdots A[n]$
           The two lists and their unions have disjoint values
Final condition: Merged list in location $A[1] \cdots A[n]$
Global     $A[1...n]$
Local      $x$, low, high, index
begin
    spawn $(P_1, P_2, \cdots, P_n)$
    for all $P_i$ where $1 \leq i \leq n$ do
        {Each processor sets bounds for binary search}
        if $i \leq n/2$ then
            $low \leftarrow (n/2) + 1$
            $high \leftarrow n$
        else
            $low \leftarrow 1$
            $high \leftarrow n/2$
        endif
        {Each processor performs binary search}
        $x \leftarrow A[i]$
        repeat
            $index \leftarrow \lfloor (low + high)/2 \rfloor$
            if $x \leq A[index]$ then
                $high \leftarrow index - 1$
            else
                $low \leftarrow index + 1$
            endif
        until $low > high$
        {Put value in correct position on merged list}
        $A[high + i - n/2] \leftarrow x$
    endfor
end
```

Fig.2-17 PRAM algorithm to merge the two sorted lists and their unions have disjoint values

(8) **Graph coloring** (Fig.2-18,2-19).

a. Problem description: given a graph, color its vertices with a predefined colors c so that no two adjacent vertices have the same color.

b. Given: a graph with n vertices, represented by an $n \times n$ adjacency matrix, and a constant c.

c. A processor is created for each possible coloring. For example: $P(i_0, i_1, \cdots, i_{n-1})$ corresponds to a coloring of vertex 0 with color i_0 and so on. Initially each processor sets its value in the n-dimensional *candidate* array to 1. It then spends $\Theta(n^2)$ time to see if the coloring is valid (if two vertices have the same coloring value assigned). If $A[j, k] = 1$, and $i_j = i_k$, then the coloring is not valid.

d. Illustration: Fig.2-18.

   (a) $n = 3$ and $c = 2$. There are three nodes and two edges.

   (b) There are $c^n = 2^3 = 8$ possible combinations of colorings on the n nodes. These 8 possible colorings are listed as $(0, 0, 0), (0, 0, 1), (0, 1, 0), \cdots (1, 1, 1)$.

   (c) $c^n = 8$ processors are spawned. Each processor $P(i_1, i_2, i_3)$ is responsible to verify if the coloring (i_1, i_2, i_3) is a valid coloring. The algorithm will output yes if at least one valid coloring is found.

e. Code: Fig.2-19, p.43.

```
GRAPH.COLORING(CREW PRAM):
Global  $n$                             {Number of vertices}
        $c$                             {Number of colors}
        $A[1...n][1..n]$                {Adjacency matrix}
        $candidate[1...c][1..c]...[1...c]$   {$n$-dimensional boolean matrix}
        $valid$                         {Number of valid colorings}
        $j, k$
begin
    spawn $(P(i_0, i_1, \cdots, i_{n-1}))$ where $0 \le i_v < c$ for $0 \le v < n$
    for all $P(i_0, i_1, \cdots, i_{n-1})$ where $0 \le i_v < c$ for $0 \le v < n$ do
        $candidate[i_0, i_1, \cdots, i_{n-1}] \leftarrow 1$
        for $j \leftarrow 0$ to $n - 1$ do
            for $k \leftarrow 0$ to $n - 1$ do
                if $A[j][k]$ and $i_j = i_k$ then
                    $candidate[i_0, i_1, \cdots, i_{n-1}] \leftarrow 0$
                endif
            endfor
        endfor
        $valid \leftarrow \Sigma candidate$ {Sum of all elements of $candidate$}
    endfor
    if $valid > 0$ then print "Valid coloring exists"
    else print "Valid coloring does not exist"
    endif
end
```

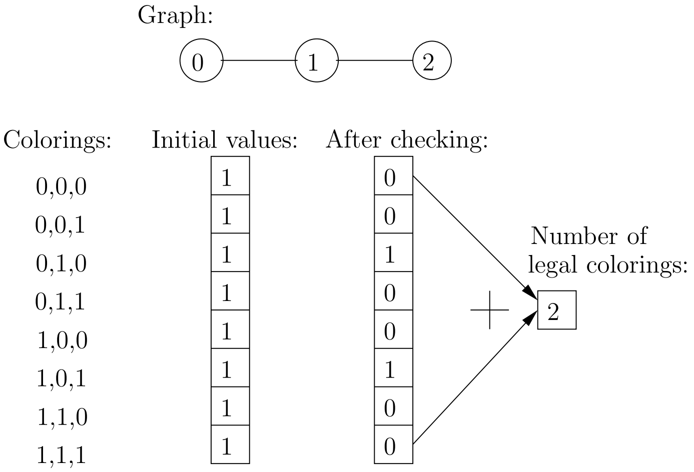


Figure 26: Example of CREW PRAM graph coloring algorithm. In this case the algorithm attempts to color a 3-vertex graph with 2 colors. time. (Fig.2-18,p.43)

Fig.2-19 CREW PRAM algorithm to determine if a given graph with n vertices can be colored with c colors.

f. Time complexity: $\Theta(\lg c^n)$ to spawn the c^n processors. Every processor executes the the doubly-nested for loop in time $\Theta(n^2)$ time. Summing the c^n answers requires time $\Theta(\lg c^n)$ time. The overall complexity thus is $\Theta(n^2 + n \lg c)$. I.e. $\Theta(n^2)$ because $c < n$.

4. Reducing the number of processors

(1) **Definition** (cost optimal parallel algorithm). A cost optimal parallel algorithm is an algorithm for which the cost is in the same time complexity class as an optimal sequential algorithm.

(2) None of the PRAM algorithms introduced so far is cost optimal. Eg: the reduction algorithm has time complexity $\Theta(\log n)$ with $\Theta(n)$ processors, given a cost $\Theta(n \log n)$, greater than the optimal sequential compexity $\Theta(n)$.

(3) To reduce the cost, we have to reduce the number of processors used.

(4) If the total number of operations performed by a parallel algorithm is in the same complexity class as an optimal sequential algorithm, then a cost optimal parallel algorithm does exist.
The parallel reduction problem is a good example. It performs n/2 additions in the 1st step, n/4 additions in the 2nd step, and so on. Total number of addition operations is n-1 over $\lceil \log n \rceil$ iterations. Hence a cost optimal PRAM algorithm for this problem does exist.

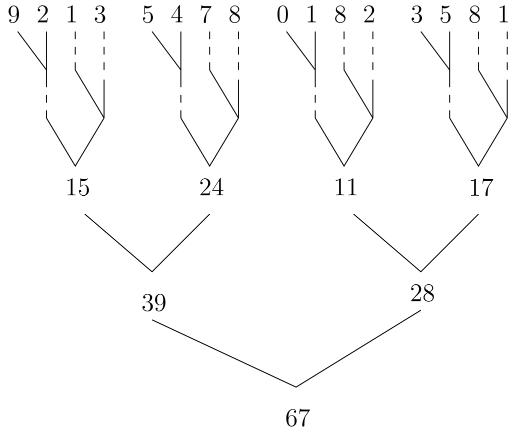


Figure 27: A PRAM can add n values in $\Theta(\log n)$ time using $\lfloor n/\log n \rfloor$ processors. During the first few iterations of the algorithm, each processor emulates a set of processors, adding to the execution time of the algorithm, but not increasing the overall complexity of the parallel algorithm beyond $\Theta(\log n)$. During later iterations, when no more than $\lfloor n/logn \rfloor$ processors are needed, the algorithm is identical to the original PRAM algorithm. time. (Fig.2-20,p.45)

(5) Is there a cost-optimal PRAM reduction algorithm that that has time complexity $\Theta(\log n)$? Minimum number of processors needed to achieve this complexity:

$$p = \frac{n-1}{\Theta(\log n)} \Rightarrow p = \Theta(n/\log n)$$

(6) Brent's theorem.

a. Significance of Brent's theorem: given the total computation time t of a given parallel algorithm A, the theorem tells relationship between t and m – the total number of operations, the theorem predicates the total time needed to execute A with p processors (the original algorithm A may use p' processors).

b. **Theorem 2.2 (Brent's Theorem)** (Brent 1974). Given A, a parallel algorithm with computation time t, if the parallel algorithm A performs m computational operations, there p processors can execute algorithm A in time $t + (m - t)/p$.

Proof: omitted.

c. Example. For the parallel reduction algorithm described previously, the time t is $\log n$. If we use $\lfloor \frac{n}{\log n} \rfloor$ processors, Brent's theorem tells the execution time is:

$$\lceil \log n \rceil + \frac{n - 1 - \lceil \log n \rceil}{\lfloor \frac{n}{\log n} \rfloor}$$

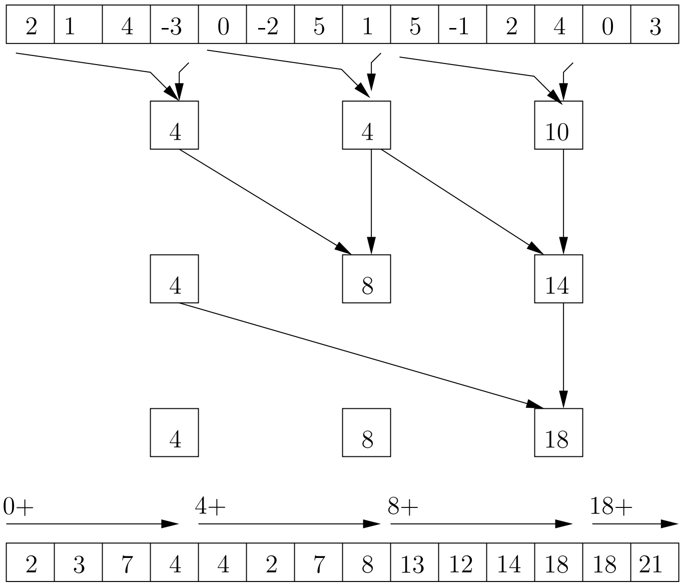


Figure 28: Illustration of a cost-optimal parallel algorithm to find prefix sums. (a) Set of 14 values is divided into 4 subsets, 1 per processor. First 3 processors find sum of their values using best sequential algorithm. (b) First three processors compute prefix sums in parallel using algorithm of Sec.2.3. (c) Each processor uses sum of values in lower processors' blocks as base for computing prefix sums in its own block, using best sequential algorithm (Fig.2-21,p.46)

$$= \Theta(\log n + logn - \frac{\log n}{n} - \frac{\log^2 n}{n})$$
$$= \Theta(\log n)$$

This indicates that, for the parallel prefix reduction algorithm previously discussed, if we reduce the number of processors previously used from n to $\lfloor n/\log n \rfloor$, the total time complexity will not change. Fig.2-20 shows how to do this.

(7) Another example – parallel prefix sum algorithm with optimal cost.

a. Given p processors and n values where $p < n - 1$.

b. Basic ideas

   (a) The n values are divided into p sets, $\lceil n/p \rceil$ values per set.

   (b) The first $p - 1$ processors compute sums of their $\lceil n/p \rceil$ values using the best known sequential algorithm. This takes $\lceil n/p \rceil$ steps.

   (c) The p processors then compute the prefix sums of these subtotals using the parallel algorithm described in Sec.2.3, in time $\lceil \log(p - 1) \rceil$.

(d) Finally, each processor uses the sums of the values in the lower blocks to compute, sequentially, the prefix of sums of its blocks of value. This takes $\lceil n/p \rceil$ steps.

(e) The total number of addition operations performed is $\Theta(n + p \log p)$. This means if p is very small, the described algorithm will be cost-optimal (because then $p \log p$ would be very small).

Therefore the execution time is:

$$\lceil \frac{n}{p} \rceil - 1 + \lceil \log(p-1) \rceil + \lceil \frac{n}{p} \rceil = \Theta(\frac{n}{p} + \log p)$$

c. Using Brent's theorem, substitute the variable t in the theorem with the execution time $\Theta(\frac{n}{p} + \log p)$ just obtained above, we have the following execution time for executing the above algorithm with processors:

$$\Theta((\frac{n}{p} + \log p) + \frac{n + p \log p - (\frac{n}{p} + \log p)}{p}) = \Theta(\frac{n}{p} + \log p)$$

The expression $\frac{n}{p} + \log p$ is minimized at $p = \Omega(n/\log n)$. Therefore the parallel algorithm has optimal cost and minimum execution time when $p = \Theta(n/\log n)$.

5. Problems that defy fast solutions on PRAM model

(0) The complexity classes $\mathcal{P}, \mathcal{NP}$, and $\mathcal{NP}$-complete.

(1) **Definition** (Polynomial functions) $T(n)^{O(1)}$ denotes polynomial functions of $T(n)$.

(2) **Definition** (Polylogarithmic functions) The set $(\log n)^{O(1)}$ is called the set of polylogarithmic functions.

(3) **Theorem 2.3** (Parallel computation thesis). The class of problems solvable in time $T(n)^{O(1)}$ by PRAM is equal to the class of problems solvable in work space $T(n)^{O(1)}$ by a RAM, if $T(n) > \log n$.

- This parallel computation thesis has been proven for those cases where $T(n)$ is a polynomial function of the problem size.
- A consequence of this theorem is that a PRAM can solve $\mathcal{NP}$-complete problems in polynomial time.
- An example: the graph color problem (which is an NP-complete problem). Solved by an $O(n^2)$ algorithm using an exponential number of processors.

(4) **Theorem 2.4**. If the number of processors in a PRAM is restricted to some polynomial function of the size of the input, then the problems solvable by the PRAM model in polynomial time is $\mathcal{P}$, the set of problems solvable in sequential polynomial time.

(5) Three definitions.

- **Definition 2.5**. A parallel algorithm has **polylogarithmic time complexity** if its time complexity is a polylogarithmic function of the problem size.
- **Definition 2.6**. $\mathcal{NC}$ is the class of problems solvable on a PRAM in polylogarithmic time using a number of processors that are a polynomial function of the problem size.
- **Definition 2.7**. A problem $L \in \mathcal{P}$ is **$\mathcal{P}$-complete** if every other problem in $\mathcal{P}$ can be transformed to L in polylogarithmic parallel time using PRAM with a polynomial number of processors.

(6) The class of $\mathcal{P}$-complete problems are those that seem to defy a fast solution (i.e. a polylogarithmic time) parallel solution.
Example problems: Maximum-flow problem

(7) The conjectured relationship between the five complexity classes: $\mathcal{P}, \mathcal{NP}, \mathcal{NC}, \mathcal{P}$-complete, and $\mathcal{NP}$-complete (Fig.2-22, p.48).

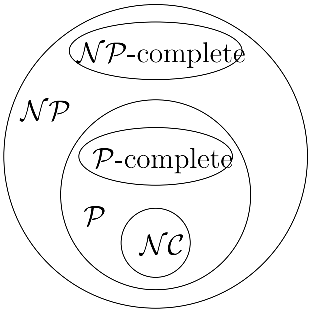


Figure 29: Conjectured relationships between the complexity classes NC, P, NP, P-complete, and NP-complete (Fig.2-22,p.48)