TEMPUS: Activity 2

Parallel Algorithms,
Chapter 1 Fundamentals
Section 2 Basic Techniques

István Erényi, KFKI-MSZKI

20th July 1997

Overview

Basic techniques
1 Introduction and General Consideration for Parallel Algorithm Design:

Much more difficult task than sequential algorithm design

Lack of well-defined methodology

Collection of techniques and paradigms may be effective for a wide range of tasks

These techniques may serve for general guidelines

Basic design techniques:

Balanced trees: *Prefix sums*
Parallel prefix

Pointer jumping

Divide and conquer: *Convex hull*

Partitioning: *Merging*

Pipelining: *Insertions into 2-3 trees*

Accelerated Cascading

Symmetry Breaking

Embedding a network (graph) into a different one

Basic techniques
1 Introduction
(continued)

Definition:

pardo*(parallel do)* statement:

for $l<=i<=u$ **pardo**

The statement(s) following the **pardo** corresponding to all values of i between l and u will be executed concurrently.

The prefix-sum algorithm:

$S_i = x_1 * x_i * ... * x_i,\ \ 1<=i<=n;$

*: denotes an associative operation

The **prefix sums** of this sequence are the n partial sums (or products) defined by the above equation.

A sequential algorithm works repeatedly applying the following operation:

$s_i = s_{i-1} * x_i$, for $2<=i<=n$.

Thus it's calculation takes $O(n)$ time.

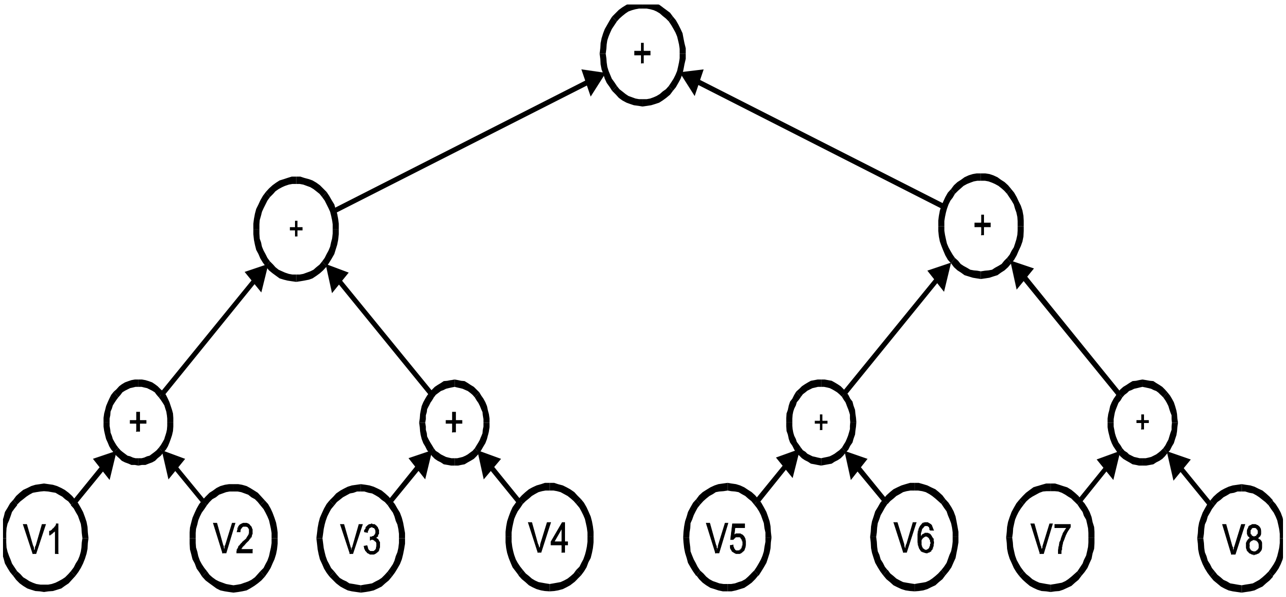
Basic techniques
2 Balanced trees:

The *Balanced Trees* method may be presented by the example of parallel summing n numbers. *The strategy is* building a balanced (binary tree) on the input elements and traversing the tree forward and backward to and from the root.

Internal nodes usually hold information concerning the data at the leaves of the subtrees rooted at these internal nodes.

The success of this strategy much depends on the existence of a method to determine the information stored in the internal nodes from the information stored at their children.

Prefix sums algorithm
with balanced tree



Balanced Tree strategy applied to summing up eight numbers
Each internal node contains the sum of the subtrees beginning at this node

Basic techniques
Balanced trees
(continued)

Recursive prefix sums algorithm

Input: array of $n=2^k$ elements $(x_1,...x_n)$

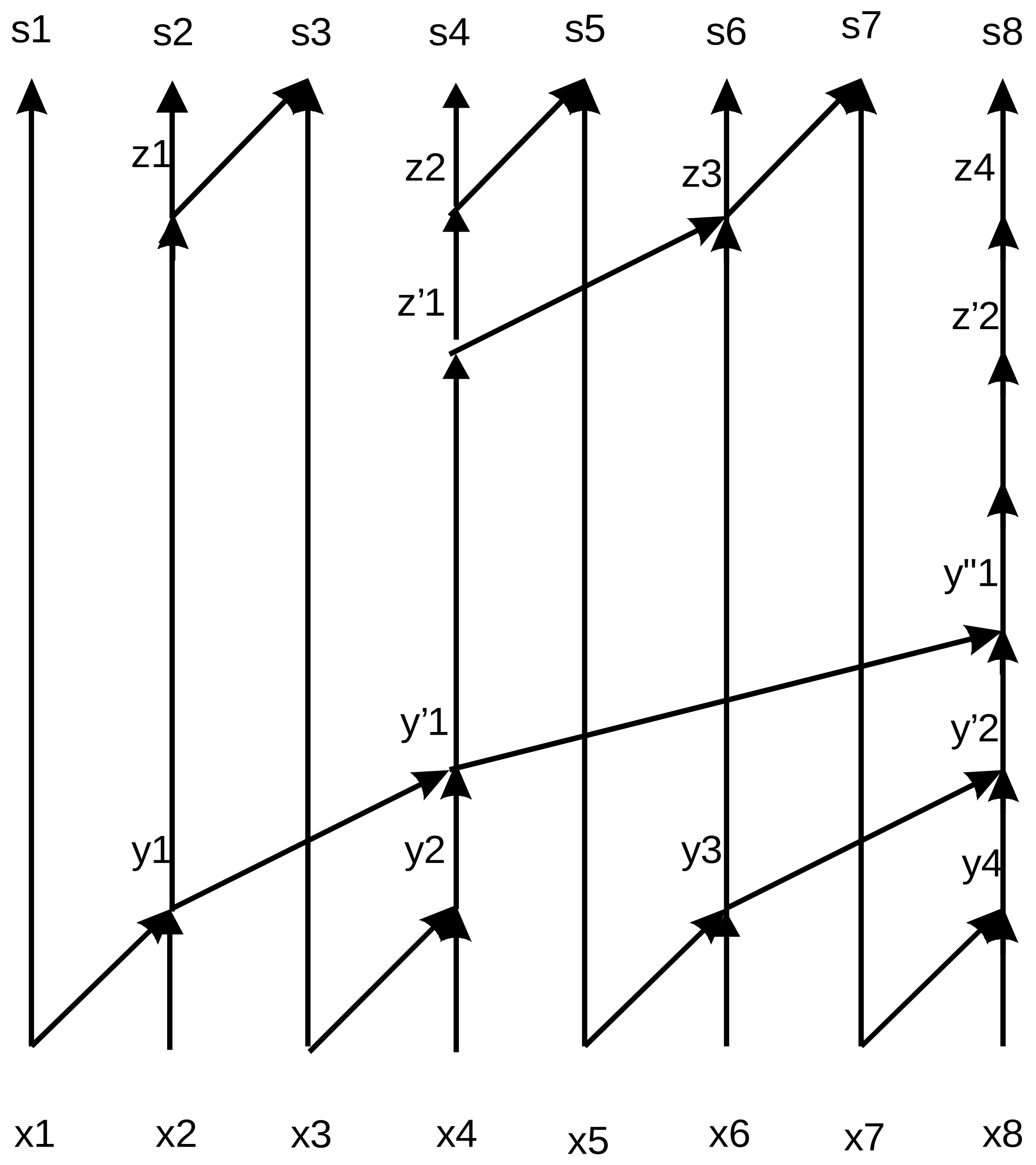
Output: the prefix sums

begin:

1. **if** $n=1$ **then** {$set\ s_1 = x_1$; **exit**}
2. **for** $1<=i<=n/2$ **pardo**
   $Set\ y_i = x_{2i-1} * x_{2i}$
3. Recursively compute the prefix sums of {$y_1,...y_{n/2}$} and store them in {$z_1,...z_{n/2}$}.
4. **for** $1<=i<=n$ **pardo**
   {i even : $set\ s_i := z_{i/2}$
   $i=1$: $set\ s_i := x_1$
   i odd>1 : $set\ s_i := z_{(i-1)/2} * x_i$}

end.

Optimal recursive prefix sums algorithm



Prefix sums of eight elements. Each node (arrow) represents a * operation on the data.

Basic techniques
2 Balanced trees
(continued)

Nonrecursive prefix sums algorithm

Input: array *A* of $n=2^k$ elements $(x_1,...x_n)$

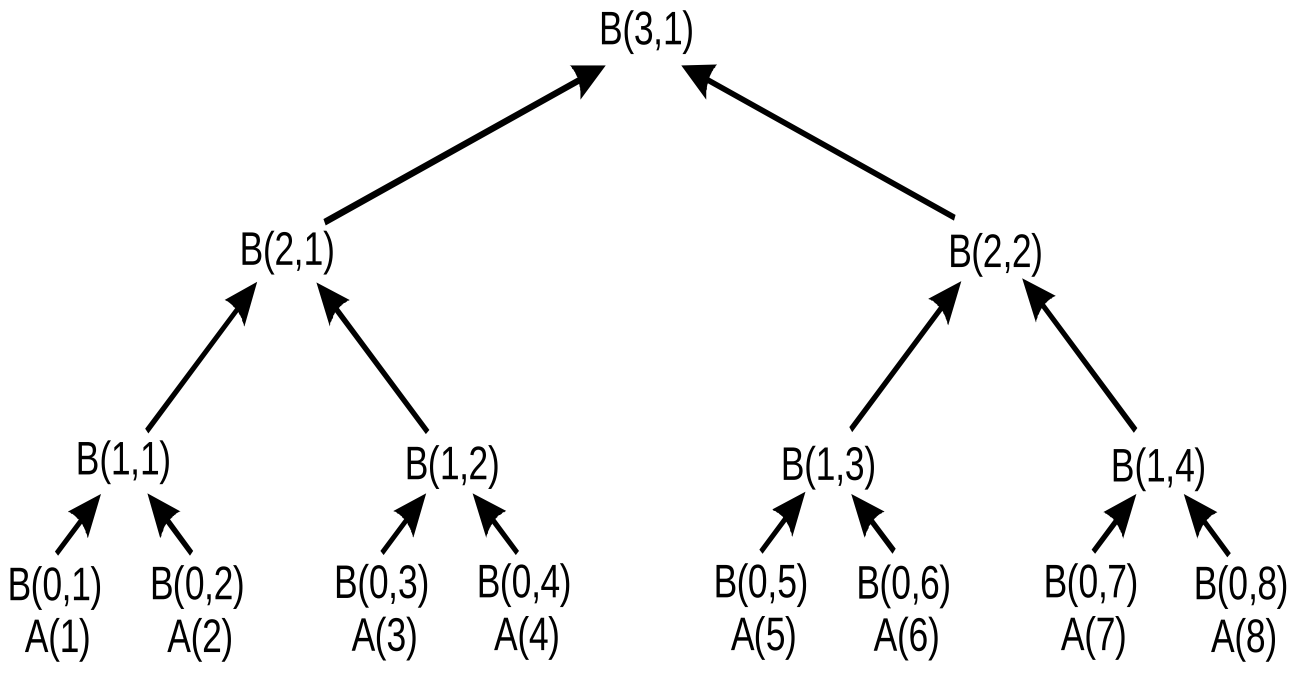
Output: the prefix sums, array *C*

begin:

1. **for** *1<=j<=n* **pardo**
   Set *B(0,j):=A(j)*
2. **for** *h=1* **to** *log n* **do**
   for $1<=j<=n/2^h$ **pardo**
   Set *B(h,j):=B(h-1,2j-1)*B(h-1,2j)*
3. **for** *h=log n* **to** *0* **do**
   for $1<=j<=n/2^h$ **pardo**
   {*j even* :*set C(h,j):=C(h+1,j/2)*
   j=1 :*set C(h,1):=B(h,1)*
   j odd>1 :*set C(h,j):=*
   *:=C(h+1,(j-1)/2)*B(h,j)*

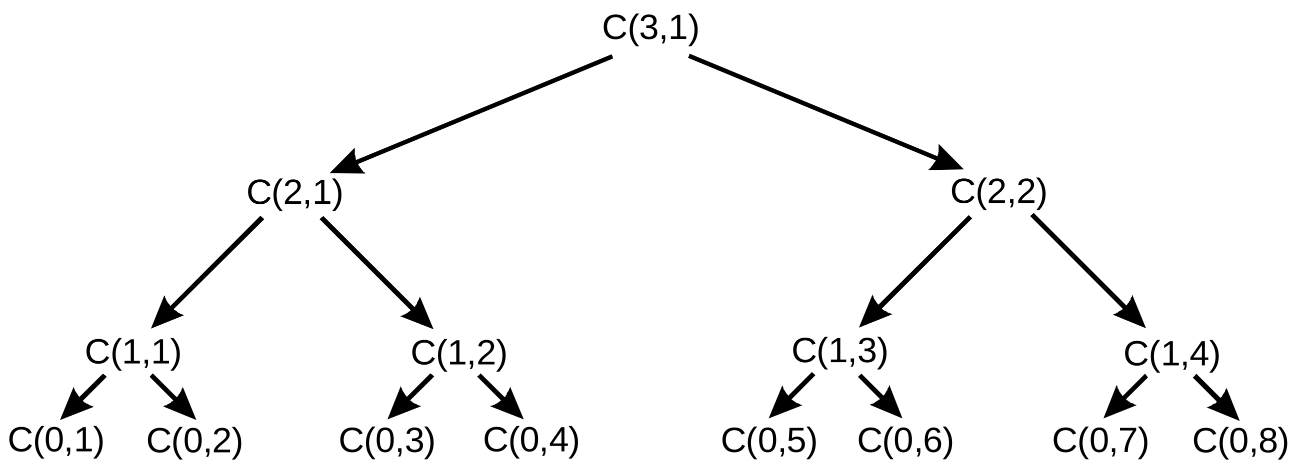
end.

2 Nonrecursive prefix sums algorithm with balanced trees
Forward traversal (bottom-up)
B(I,j) generation



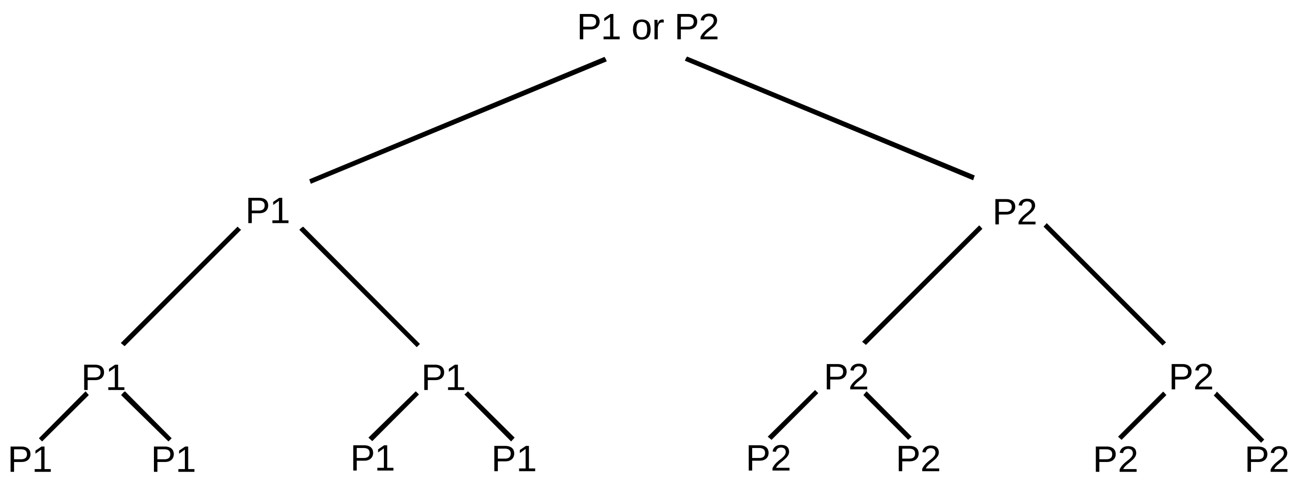
Balanced tree, nonrecursive prefix-sums algorithm.
The bottom-up traversal of the binary tree.

2 Nonrecursive prefix sums algorithm with balanced trees
Backward traversal (top down)
C generation



Balanced tree, nonrecursive prefix-sums algorithm. Elements of *C* as generated by the top-down traversal of the binary tree.

2 Nonrecursive prefix sums algorithm with balanced trees
A posible processor allocation scheme



Balanced tree, nonrecursive prefix-sums algorithm.
Processor allocation with two processors.
The processor is responsible for computing the data generated at the node where the processor is shown.

Basic techniques
3 Pointer Jumping

Presented on the example of:

Finding the roots of a forest

Forest F consists of a set of rooted trees. It is specified by an array of parent node P of length n. $P(i)=j$ if (i,j) is an edge in F; i.e. j is the parent of i in a tree. For the root $P(i)=i$.

The problem is determining the root for each j between 1 and n.

Sequential algorithm:

1. Determine all of the roots
2. Perform depth-first or breadth-first traversal of each tree from the root

The cost of the sequential algorithm is linear $O(n)$.

Basic techniques
3 Pointer Jumping
(continued)

Finding the roots of a forest

Parallel algorithm is based on finding *successor* *S(i)* of node *i*, that is its parent *P(i);* and on repeated updating the successor of each node by that successor's successor. The distance doubles at each repetition, unless the root is reached. It is the method of **pointer jumping** or **path doubling.**

Input: A forest of rooted directed trees

Output: for each *i*, root *S(i)* of the tree

begin:

1. **for** $1<=i<=n$ **pardo**

    Set $S(i):=P(i)$

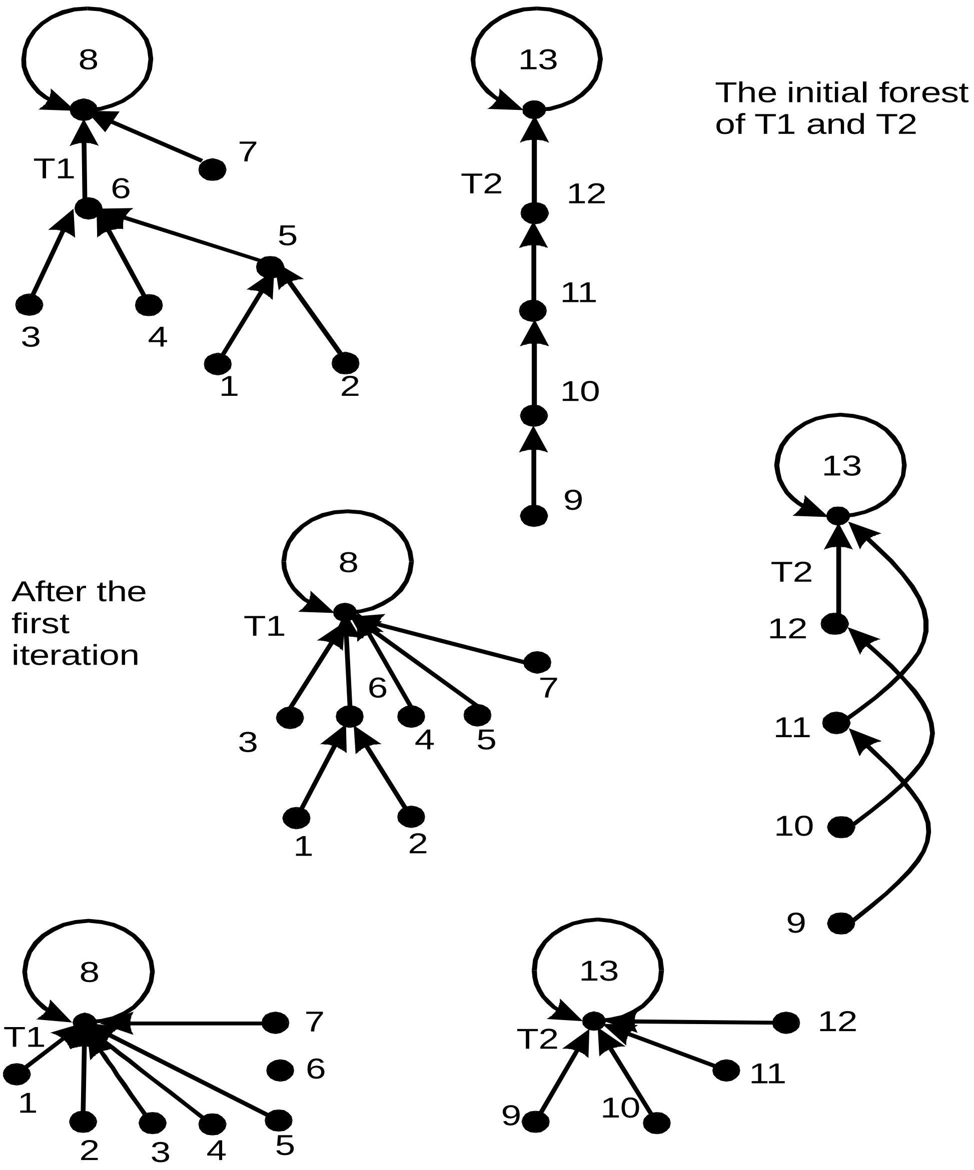
    while $(S(i))\neq S(S(i))$ **do**

    Set $S(i):=S(S(i))$

end.

3 Finding the roots of a forest
Pointer with jumping

Pointer Jumping, finding the roots of a forest



After the second iteration

Basic techniques
3 Pointer Jumping
(continued)

Sum of weights of the nodes of a forest

The algorithm is similar to the one that finds the roots of the forest, it is extended by parallel calculation of the prefix sums.

Input: A forest of rooted directed trees, with each vertex having its weight $W(i)$

Output: for each i, $W(i)$ is set to the sum of weights in the path from i to the root of the tree

begin:

1. **for** $1<=i<=n$ **pardo**
   - Set $S(i):=P(i)$
   - **while** $(S(i))\neq S(S(i))$ **do**
   - Set $W(i):=W(i)+W(S(i))$
   - Set $S(i):=S(S(i))$

end.

Basic techniques

4 Divide and Conquer

The strategy consists of three main steps:

1. partitioning the input into several parts of almost equal sizes,
2. recursive solution of the sub problem (sub problems may be solved concurrently)
3. combining and merging the sub-solutions into the final solution.

The success depends on the efficient solution of the first and third subsets. The strategy has been successful in several sequential algorithms.

The example for illustrating the strategy is the **convex hull** problem (see in Ch. 4).

Basic techniques
4 Divide and Conquer
(continued)

Element ranking problem:

Given a nondecreasing sequence $A=\{a_1,a_2,...,a_n\}$ of n elements, The rank of the element a, denoted by $rank(a:A)$ is the number of elements of A that are less or equal to a.

The *divide and conquer* method consists of comparing a with the middle element of A. Based on the outcome of comparison the new search can be restricted to the upper of lower half of A, let denote it A'.

Now a is compared with the middle element of A' and based on the result the new search is restricted to the upper or lower half of A', denoted by A''. The search is recursively repeated until a is isolated between two successive elements of A. The cost of the method is $O(logn)$.

Basic techniques
5 Partitioning

The **partitioning** strategy consists of

1. breaking up the given problem into *p* independent almost equal subproblems
2. solving the subproblems concurrently.

p is the number of processors.

Partitioning is a similar strategy to the *divide and conquer* one, however, there the main task is combining the subsolutions, while in this case the main task is partitioning the input.

The example to show the strategy is the merging algorithm.

Basic techniques

5 Partitioning

(continued)

The merging task is as follows:

Let $A(a_1,a_2,...,a_n)$ and $B(b_1,b_2,...,b_m)$ be two nondecreasing sequences. The task is merging them into an sorted sequence $C(c_1,c_2,...,c_{n+m})$.

An optimal merging algorithm:

1. Choose approximately *n/logn* elements of each of *A* and *B* that partition *A* and *B* into blocks of almost equal sizes.
2. Rank each of the chosen elements in the other sequence. This step reduces the task into merging pairs of subsequences, each of *O(logn)* elements.
3. Apply the sequential algorithms to these subsequences.
4. Combine the sorted subsequences.

Basic techniques
5 Partitioning
(continued)

Partitioning for the merging algorithm:

Input: arrays $A(a_1,...,a_n)$ and $B(b_1,...,b_m)$ (both *logm and m/logm* are integers).

Output: $k(m)=mlogm$ pairs (A_i, B_i) of subsequences , such that **a.)** $|B_i|=logm;$ **b.)** $\Sigma_i|A_i|=n;$ **c.)** each element of A_i and B_i is larger than each element of A_{i-1} or B_{i-1}.

begin

1. *Set* $j(0):=0$, $j(k(m)):=n$
2. **for** $1<=i<=k(m)-1$ **pardo**
   2.1. *Rank* b_{ilogm} *in A, and*
   let $j(i)=rank\ (b_{ilogm}:A)$
3. **for** $0<=i<=k(m)-1$ **pardo**
   3.1 Set $B_i:=(b_{ilogm+1},...,b_{(i+1)logm})$
   3.2 Set $A_i:=(a_{j(i)+1},...,a_{j(i+1)})$
   (A_i *is empty if* $j(i)=j(i+1)$)

end.

Basic techniques

5 Partitioning

(Continued)

An example for merging two sequences:

$A=(3,7,8,12,13,17,23,27);\ B=(2,10,20,30)$

$m=4;\qquad k(m)=m/logm=2;$

Rank b_{ilogm}*=in A,* i.e. *rank (10:A)=3,*
the following two pairs of sequences are obtained:

1. $A_0=(3,7,8)$ and $B_0=(2,10)$

2. $A_1=(12,13,17,23,27)$ and $B_1=(20,30)$

Note: each element in A_0, B_0 is smaller than any element in the second pair. Hence we simple can merge the pairs (A_0, B_0) and (A_1, B_1) separately, getting thus C_0 and C_1, then concatenating them into the result, C.

Basic techniques

Partitioning

Partitioning for the merging algorithm:

	B_0	B_1		B_i	
B	$b_1 \ldots$ $\ldots b_{logm}$	$B_{logm+1} \ldots$ $\ldots b_{2logm}$	•••	$b_{ilogm+1} \ldots$ $\ldots b_{(i+1)logm}$	•••
	$\downarrow$	$\downarrow$		$\downarrow$	
A	$a_1 \ldots$ $\ldots a_{j(1)}$	$a_{j(1)+1} \ldots$ $\ldots a_{j(2)}$	•••	$a_{j(i)+1} \ldots$ $\ldots a_{j(I+1)}$	•••
	B_0	B_1		B_i	

Basic techniques
6 Pipelining

The overall operation is subdivided into stages (subtasks). Subtasks can be viewed as a wave moving to the next stage and so on, up to the conclusion of the operation. When a stage finishes its subtask on one data item, it can get the next item for processing. The suboperations of the stages can be executed concurrently.

The example algorithm to present the pipelining technique is **the insertion operation into a 2-3 tree**.

2-3 tree is a rooted tree with nodes having 2 or 3 children, and every path from the root to a leaf is of the same length.

Sorted lists can be represented by *2-3 trees*, where the items are stored at the leaves from left-to-right order. An internal node has the larger items in its right subtree.

Basic techniques

6 Pipelining shown by 2-3 trees

(continued)

Basic operations in sorted 2-3 tree lists are:
a.) *element search* b.) *element insertion*
c.) *element deletion*.

The structure has to be preserved after insert and delete operations

Insert operation consists of three phases: *search, insert, restructuring* (if needed).

Search: 2-3 ordered tree T has n elements $a_1<a_2<...<a_n$. Given and element b, the search is finding the leaf u with a_i that satisfies: $a_i<=b<a_{i+1}$.

The algorithm: search start at root r. Search goes on in one of the two or three subtrees $L[r]$, $M[r]$ or $R[r]$. If $b<= L[r]$ in the left, if $L[r]<b<=M[r]$ in the middle, otherwise in the right subtree.
It is repeated till leaves are reached.

Basic techniques

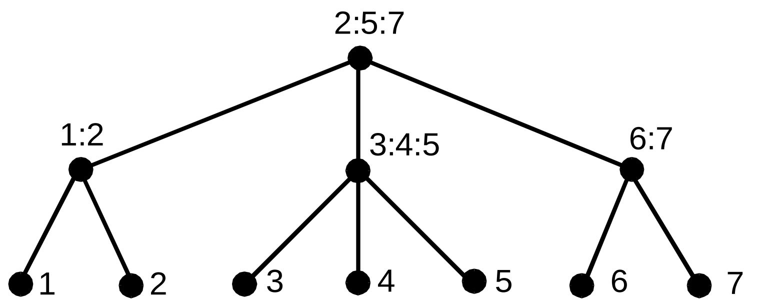
6 Pipelining shown by 2-3 trees:

(continued)

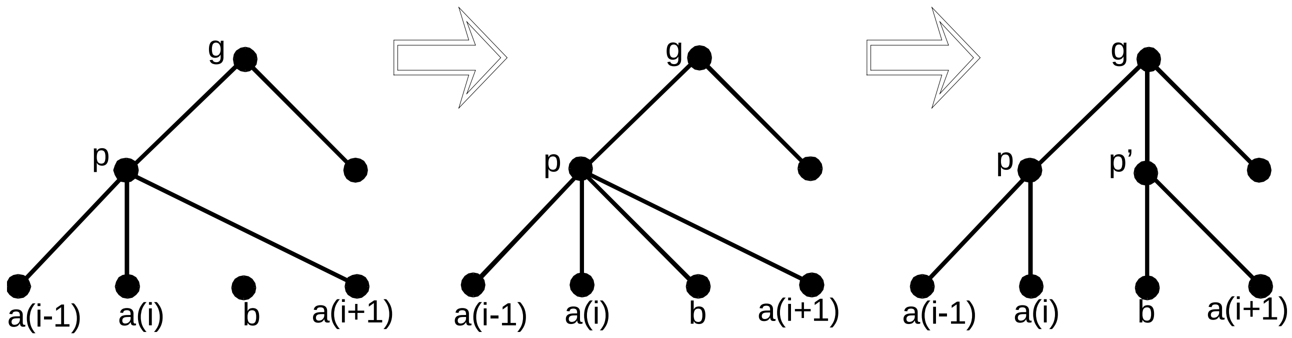
Insertion: After searching and locating *u* such as its item a_i satisfies $a_i<=b<a_{i+1}$. Now if $a_i \neq b$ we create a new leaf *u'* for *b* inserting it after *u* to the same parent *p*.
If *p* has got three children, the operation has been done, otherwise (in a four children case) we create a new parent *p'* and allocate the four children between them. Then for *p'* we can continue insertion in the same way until we reach three children. If we reach the root *r* that has got four children, we create a new root with two children, each of which has the appropriate two children of the root *r*.

Since the size *n* of the ordered sequence is between 2^h and 3^h, where *h* is the height of the tree, the operation cost is $O(logn)$.

A 2-3 tree
An example of *inserting*
an item into the tree



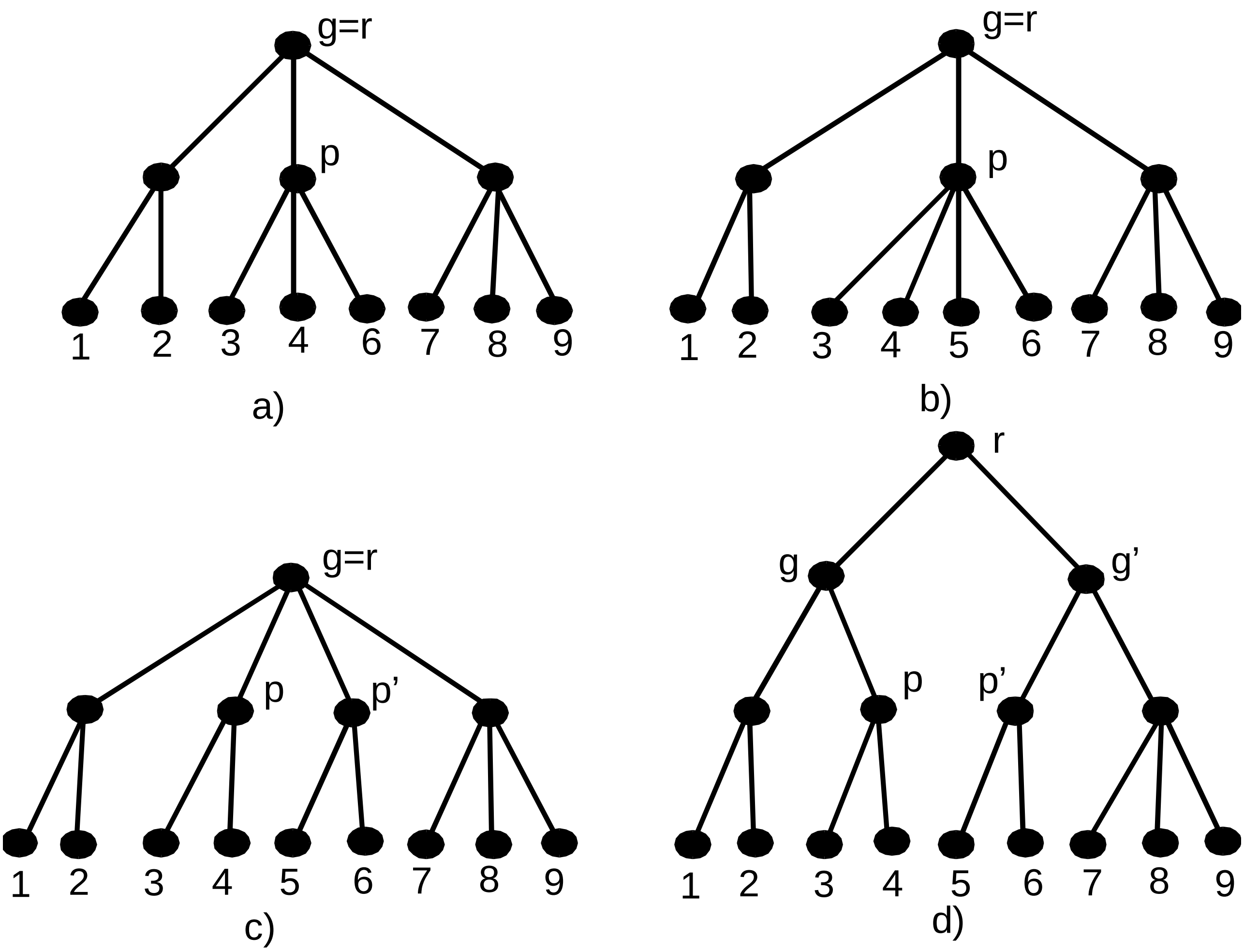
Example of a 2-3 tree. The notation L[v]:M[v]:R[v]
at the nodes denotes the larges elements
in the left, middle and right subtrees



The process of inserting an item b into a 2-3 tree. 1. A node is created;2. It is assigned to parent p; 3. A new parent node is created and children assigned to it

2-3 tree

Inserting 5 among values stored in the tree



Steps for inserting 5 into a 2-3 tree. a.) The tree; b.) Inserting a new leaf c.) Inserting new parent node with leaves rearranged; d.) The final tree with new grandparents and a new root.

Basic techniques
6 Pipelining shown by 2-3 trees:
(continued)

Inserting a sequence of items:

Into a 2-3 tree T with n items $a_1<a_2<...<a_n$ an other sequence $b_1<b_2<...<b_k$ has to be inserted (k is much smaller than n).

The steps of the algorithm:

1. Sequential insertion of b_1 and b_k in order to find the two furthermost leaves of T between which all b_i-s fit.
2. Concurrently we locate all b_i-s and create a leaf for each of them.
3. Inserting k_i elements into the 2-3 tree. If a set of b_i-s has to be inserted in between two consecutive a_i-s, we call this set a **chain,** and denote its length k_i. This operation may be executed concurrently. This is the **parallel insertion algorithm.**

Basic techniques

6 Pipelining shown by 2-3 trees:

(continued)

The **parallel insertion algorithm.**

If all k_i-s are $k_i<=1$ then for each b_i a leaf l_i is created. If at height 1 a node ν had three children, at most three new children leaves may be created. So, at most another new node ν' should be created in order to restructure height 1. This step can be done concurrently, and then repeatedly until we reach the root.

If not all k_i-s are 0 or 1, let us denote B_i the part of B to be inserted between two consecutive a_i-s. In this case we divide B_i-s each into two parts and insert the middle elements concurrently. After we reorder the whole 2-3 tree. Repeatedly applying this, we may insert all the elements.

Basic techniques
6 Pipelining shown by 2-3 trees:
(continued)

The **parallel insertion algorithm.**

Pipeline stages:

A single element insertion of B_i-s may be called a **Simple parallel insertion algorithm.** It can be regarded as a stage of pipelining.

The parallel insertion algorithm consists of $<= \log k$ stages. During the *i-th* stage, a set of elements is inserted into the tree T_{i-1} to obtain T_i. Each T_i is processed bottom-up, height by height as a wave. Once the leaves are inserted, the wave moves up one level in order to let inserting the next leaves. Therefore the different stages are pipelined.

The running time of this method is $O(\log n)$ using a total of *O(klogn) operations*.

Basic techniques

7 Accelerated Cascading

The strategy is the following:

1. Start with an (optimal) algorithm until the size of the problem reduces.
2. Shift to a faster algorithm, (e.g. the Double-logarithmic time algorithm).

An example: calculate the maximum element of a sequence of *n* elements. This is a *prefix sums* problem, where the operation (denoted by *) is comparison, resulting in a Boolean value.

First algorithm runs using the *balanced trees method* for the prefix sums calculation, it takes *O(logn)* runtime.

The second algorithm is the **Doubly Logarithmic-Time Algorithm.**

Basic techniques
7 Accelerated Cascading
(continued)

Doubly Logarithmic-Time Algorithm.

<u>Definition:</u> *Doubly Logarithmic-Depth Tree*

Node-Level is the number of edges on the path between node *u* and the root *r*.

The *DLDT* tree has $n=2^{2^k}$ leaves. The root has $2^{2^{k-1}}$ children, each of its children has $2^{2^{k-2}}$ children, (the *i-th* level has $2^{2^{k-i-1}}$). Finally each node at level *k* will have two leaves as children.

The features of the DLDT:

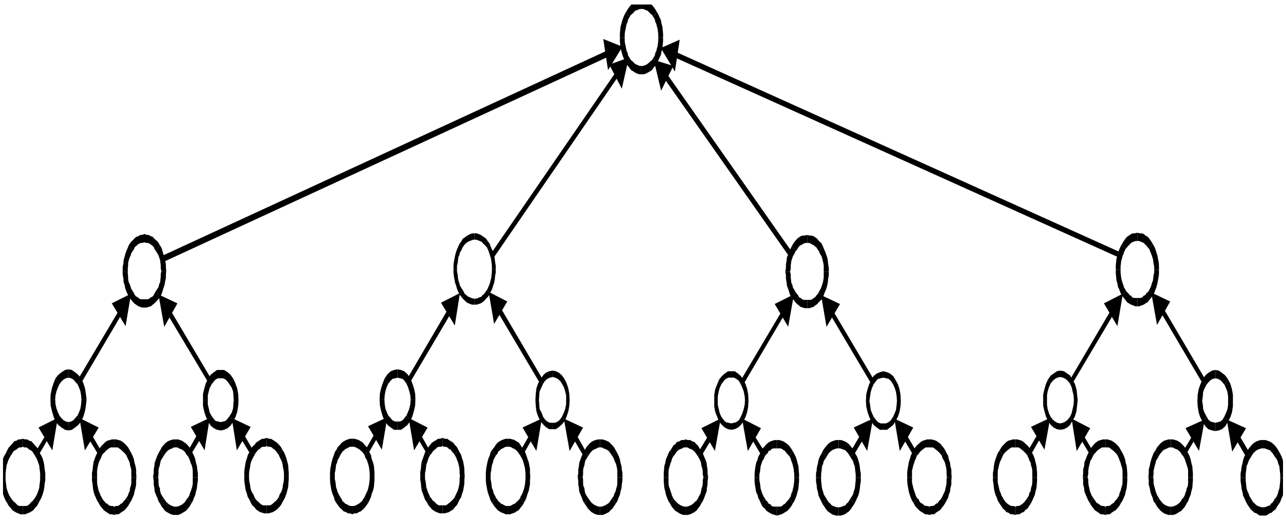
number of nodes at level *i* is $2^{2^k-2^{k-i}}$ for $0<=i<k;$

number of nodes at level *k* is $2^{2^{k-1}}=n/2$

The depth of the DLDT: is

$k+1= \log\log n + 1$

A 16-leaf Doubly Logarithmic-DepthTree



Doubly logarithmic-depth tree with 16 leaves

Basic techniques
7 Accelerated Cascading
(continued)

Doubly Logarithmic-Time Algorithm.

The *Doubly Logarithmic-Depth Tree* for the calculation of the maximum of elements:

The algorithm:

1. Each node holds the maximum of the elements of its subtree. The algorithm proceeds bottom-up.
2. The algorithm of the nodes is the **Basic maximum** (BM) algorithm.

Since the algorithm reaches its result, when the calculation reaches the root of the DLDT, the running time is *O(log log n)* times the BM execution time.
(Below we prove, that BM execution time does not depend on *n*).

Basic techniques
7 Accelerated cascading
(continued)

The **Basic maximum** (BM) **Algorithm.**

Input: *An array A of p distinct elements (p is the number of processors)*

Output: *Boolean Array of M, M(i) =1 if and only if A(i) is the maximum element.*

begin

1. **for** $1<=i,j<=p$ **pardo**
   if $A(i)>=A(j)$ **then** *set* $B(i,j):=1$
   else *set* $B(i,j):=0$
2. **for** $1<=i<=p$ **pardo**
   Set $M(i):=B(i,1)\cap...\cap B(i,p)$

end.

Step 1. is done via simultaneous read operations

Step 2. via concurrent writes of the same value (CRCW PRAM model).

Basic techniques

8 Symmetry breaking

In a lot of computing problems parallel execution can not treat all the input items, that may be symmetric. In these cases, some mechanism should be introduced to partition the items into classes, and thus the problem solution may be started.

The symmetry breaking method is introduced on the example of the coloring problem:

The coloring problem (example)

Let have a directed cycle $G=(V,E)$, the indegree and outdegree of each of the vertices is 1. For any of the two vertices u and v a directed path exists. **k-coloring** of G assigns a color c to each vertex such that no neighboring vertices have the same color.

Basic techniques
8 Symmetry breaking
(continued)

The coloring problem:

Simple for a sequential algorithm: use two colors alternatively, and at the end (if the vertices number is odd) add a third color.

For parallel algorithms:

Basic coloring

Input: *a directed cycle, its arcs are specified by an array S. Color* c_i *= vertex number.*

Output: *New coloring c' of the vertices.*

begin

for *1<=i<=n* **pardo**

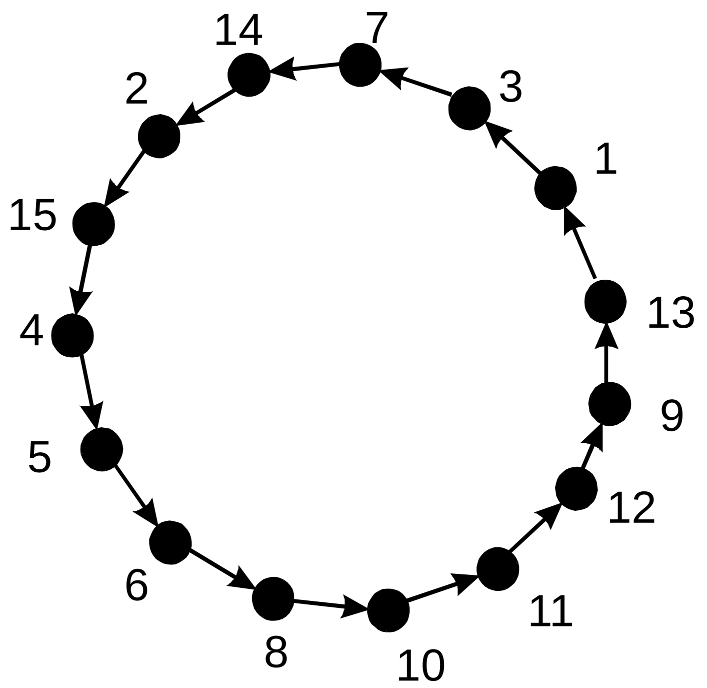
1. Set k to LSB where c(i) and c(i+1) disagree (i cyclically increased)

2. Set $c'(i):=2k+c(i)_k$

end.

It can be proven that *c'* is a valid coloring.

A directed cycle and the basic coloring algorithm



v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

The basic coloring algorithm on a directed cycle. The first two columns show original coloring, k is the LSB in which c(i) and c(i+1) differ, c'=2k+c(i)k, is the new color

Basic techniques
9 Embedding Networks into other networks

An algorithm may have been designed for one architecture, it may be needed to adapt it into an other one.

Let nodes correspond to processors, edges to communication links.

Given two graphs *G(V,E)* and *G'(V',E')*.

Embedding *G* into *G'* means mapping each vertex in *V* onto a vertex (or vertices) in *V'*, and each edge in *E* onto an edge (or set of edges) in *E'*.

For mapping three parameters are important:

If more edges are mapped onto one: **edge congestion**, (additional traffic); if one edge onto more: **edge dilation,** if the number of the vertices differs, then **expansion** of the mapping exist.

The strategy is shown on several examples.

Basic techniques
9 Embedding Networks into other networks
(continued)

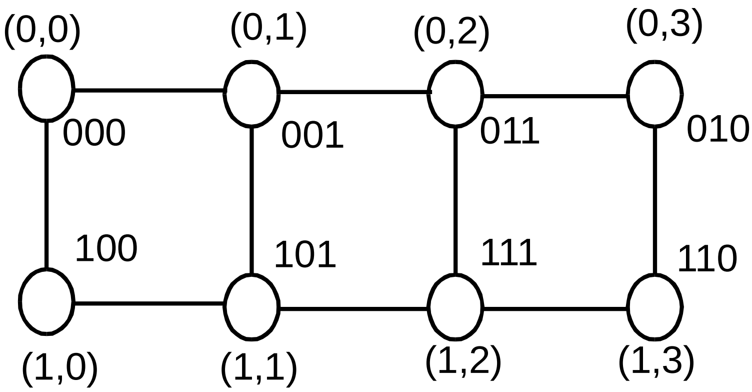
Embedding Linear array into a Hypercube

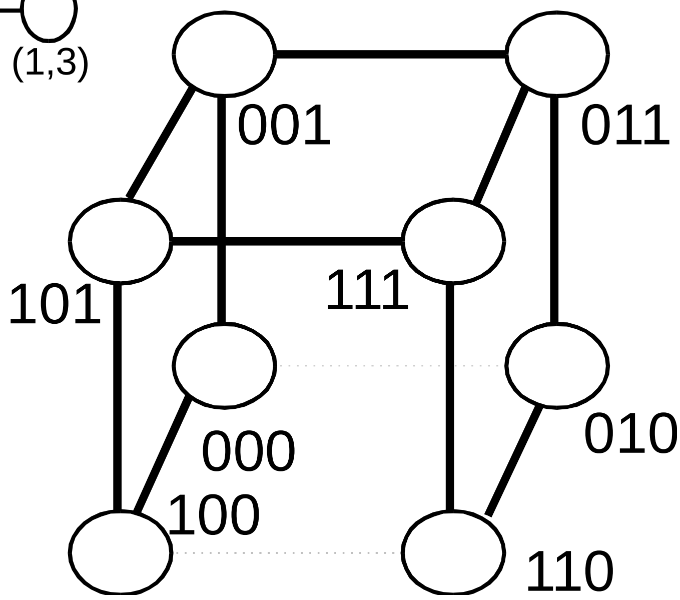
Linear array or ring of 2^d processors may be embedded into a *d*-D hypercube by mapping processor *i* onto the processor *G(i,d)* of the hypercube.

Embedding a Mesh into a Hypercube

This is a natural extension of embedding a ring into a hypercube. A $2^r \times 2^s$ mesh maps directly on a 2^{r+s} processor hypercube. Immediate neighbors in the mesh are mapped to hypercube processors with the label differing only in one bit. The dilation and congestion of this mapping are of one.

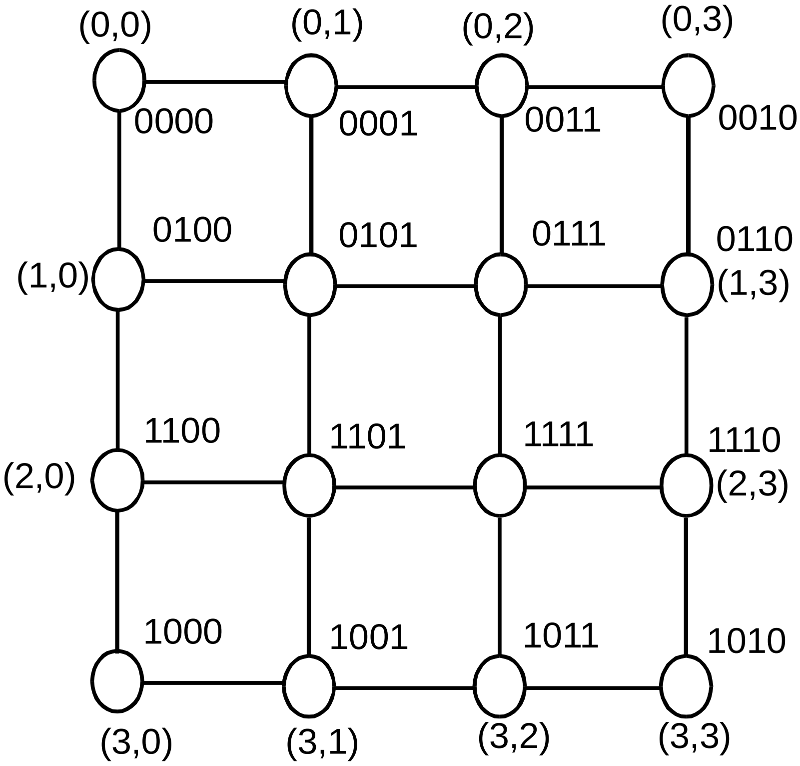
9 Embedding an 8-node mesh into a hypercube





A 2x4 mesh embedded into
a three dimensional hypercube.
Note: at the nodes of the mesh
we included both numberings
the 2-D mesh coordinates, and
the hypercube coordinates.

Numbering the 16 nodes of a 2-D mesh with the node labels of a 4-D hypercube for mapping



A 4x4 mesh with illustrations how its processors may be mapped to a four-dimensional hypercube

Basic techniques
9 Embedding Networks into other networks
(continued)

Embedding a Binary Tree into a Hypercube

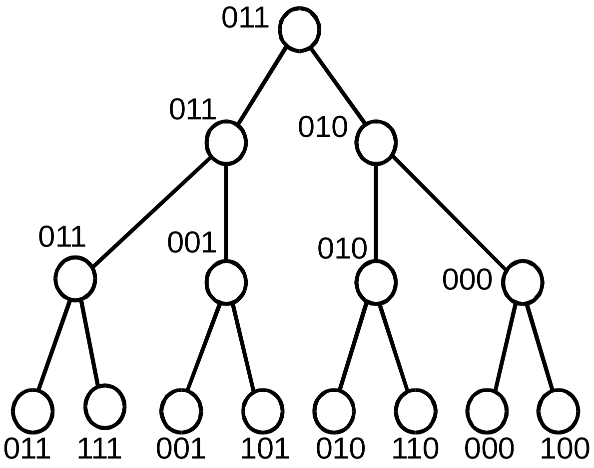
We may embed a binary tree of depth *d* with processors into a *d*-*D* hypercube using the following rules:

1. The root of the tree is mapped to any of processors.

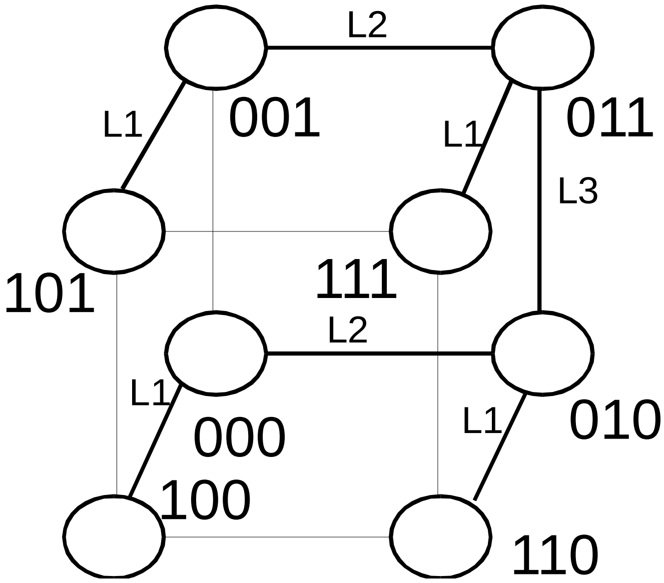
2 For node *i* at depth *j* the left child of *i* is mapped to hypercube processor *i*, and the right child is mapped to the processor whose label we obtain by inverting bit *j* of processor *i*'s label.

3. An edge between distinct nodes of the tree may be mapped into one single edge of the hypercube. This mapping has a congestion, expansion and dilation of one.

Embedding a binary tree with 8 leaves into a 3-D Hypercube



L3: edge at level 3
L2: edges at level 2
L1:edges at level 1



A tree embedded into a three-dimensional hypercube