# Chapter 1:

# Parallel Algorithms Fundamentals

## *Section 2:*

## *1.2 Basic Techniques for Parallel Algorithm Development*

Designing and developing parallel algorithms for parallel computers is an incomparably more difficult task then the development of sequential ones. Moreover, frequently a sequential algorithm does not provide a good starting point for the development of a parallel algorithm, because optimal versions of sequential algorithms often contain little part that may be directly parallelized. A better approach in this case is 'starting from the scratch' and trying to find algorithmic versions with internal natural parallelism even if the total cost (the total number of basic operations) of this algorithm becomes higher than the cost of the optimal sequential algorithm. Partitioning this new algorithm and creating concurrent operations for these parts often gives very promising results.

There is no well-defined methodology for parallel algorithm design. However a lot of techniques exist, that may be successfully applied, and thus the design of parallel algorithms becomes 'an art' in terms that experience, skills and also creativity of the researcher often ensure successful development of novel algorithms. The techniques may serve as general guidelines supporting researchers.

In this chapter we introduce the most popular techniques. Since it is the bets way to presenting a technique and showing its power, we use examples for various problems. Parallel algorithms developed with the help of these techniques offer solutions for these problems.

The techniques and problems we will discuss in the following chapters include:

* **balanced trees,** problems of prefix sums, parallel prefix sums,
* **pointer jumping**,
* **divide and conquer,**
* **partitioning,** merging problems,
* **pipelining**, insertion problems with 2-3-tree representation of data,
* **accelerated cascading**,
* **symmetry breaking,** graph coloring problem,
* **embedding graphs into other graphs.**

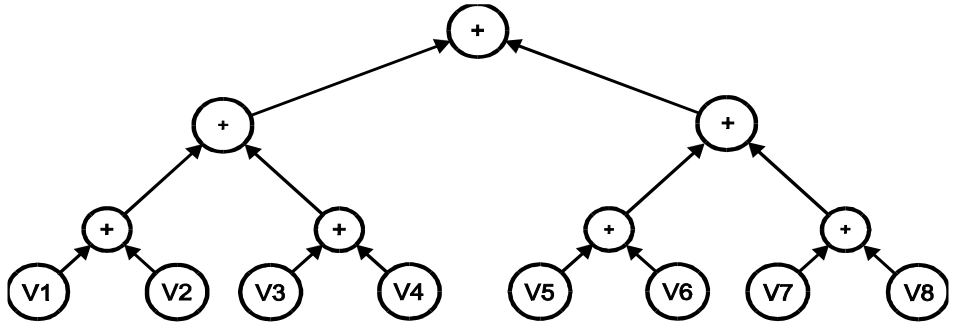
During the discussion of the techniques we will use the **pardo** statement. Here we define the statement:

**for** $l \leq u$ **pardo**

Statement(s) following this **pardo** and corresponding to $i$ between value $l$ and $u$ will be executed concurrently.

## 1.2.1. Balanced trees

The technique based on balanced trees can easily be presented selecting simple problems, like adding up eight numbers (Figure 1.2.1). However, the problem could be rather complex and finding the parallel partitions – in order to build a **balanced (binary) tree** – may not be so obvious.




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

**Figure 1.2.1**

The strategy with balanced trees is based on searching $l_1$ parallel executable partitions, whose output data may be joined and passed to the next level of processing. At this level the joined data are passed to $l_2 < l_1$ new partitions of the algorithm, and so on. Internal nodes usually hold the information related the data of their children, i.e. the subtrees originating at these internal nodes. In fact, during the execution of the algorithm the operations and data may traverse back and forward between the leaves and the root.

The success of the method depends on whether we are able to define the information of the nodes at internal levels from the information at their children or at the leaves of the tree.

The balanced tree technique will be presented on the example of the **prefix-sums algorithm.**

**The prefix sums:**

Given a sequence of $n$ elements $\{x_1, x_2,...,x_n\}$. An **associative** operation (addition, multiplication, etc.) denoted by * is applied to the elements of the sequence. The **prefix sums** are $n$ partial results (sums, products, etc.) defined by

$$s_i = x_1 * x_2 *,...,x_i; \quad 1 \le i \le n.$$

A sequential algorithm would be organized in cycles, repeatedly summing (here and below when speaking on prefix sums, summing means any associative operation denoted by *) the values: $s_i = s_{i-1} * x_i$, for $2 \le i \le n.$ This algorithm takes $O(n)$ time.

A fast parallel algorithm can use balanced trees for the prefix sums calculation. The internal nodes represent the * operation for their children, thus during traversal each node stores the sum of its subtree data. For getting the prefix sums, that is not enough to reach the root (it gives us the sum of all of the $n$ elements). During backward travel, we may calculate the prefix sums and get them at leaves, using up the sums of subtrees stored at the internal nodes.

Two versions of the parallel algorithm solution are given in the followings, the recursive and a nonrecursive one.

**The recursive prefix sums algorithm:**

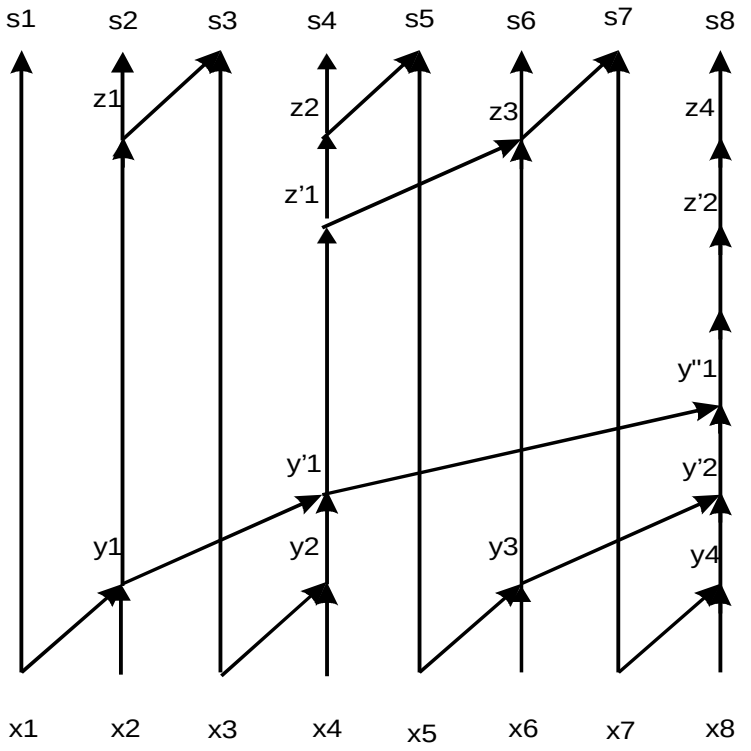
**Input:** *array of $2^k$ elements ($x_1, x_2,..., x_n$).*
**Output:** *prefix sums $s_i$.*
**begin**

1. **if** $n=1$ **then** {*set* $s_1 := x_1$; **exit**}
2. **for** $1 \le i \le n/2$ **pardo**
   *Set* $y_i := x_{2i-1} * x_{2i}$
3. *Compute recursively the prefix sums of* $\{y_1, y_2,..., y_{n/2}\}$, *and store them in* $z_1, z_2,...z/2_n$
4. **for** $1 \le i \le n$ **pardo**
   {$i = 1$ :*set* $s_1 := z_1$

*i even* :set $s_i := z_{i/2}$
*i odd > 1* :set $s_i := z_{(2i-1)/2} * x_i$}

**end**

Figure1.2.2 represents the algorithm data-flow diagram. The first basic operation is computing sums of the pairs of the input values in the order of their indices. This gives the partial results denoted by $y_1$,..., $y_4$. Recursively calling, first we get $y'_1$ and $y'_2$, then $y''_1$. At this moment we get the final sum, that is the eighth prefix sum, and naturally we have the first ($x_1$), the second ($x_1+x_2=y_1$), and fourth ($y'_1 = x_1, +...+x_4$); however the remaining prefix sum values have to be calculated. For this, a reverse traverse is required. Please note that *y*-s are now stored as *z*-s. The third sum we get using $x_1$and $z_1$. Following the diagram, it is easy to see how all the prefix sums are calculated.




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

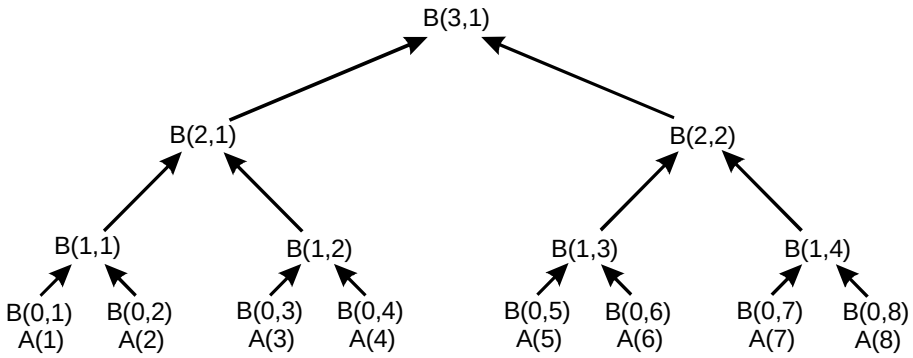
**Figure 1.2.2**

**The nonrecursive prefix sums algorithm**

In this algorithm we use three arrays: *A* is the input array, forward traverse prepares array *B,* while array *C* is calculated backward. In Figures 1.2.3 and 1.2.4 the forward and backward traverses are illustrated in order to help understanding the algorithm below. Figure 1.2.5 shows the possible processor allocation for the algorithm.

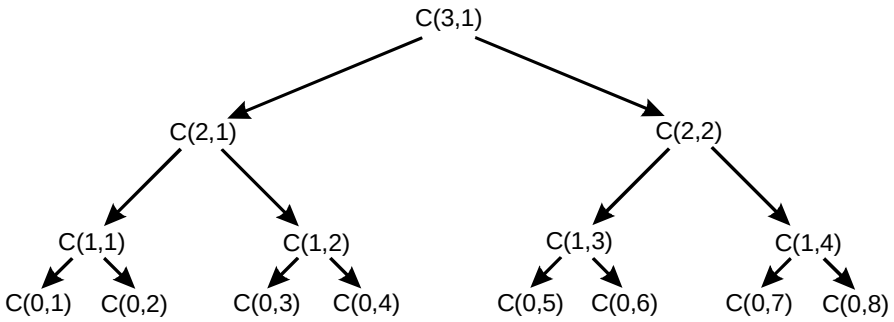
**Input:** *array A of* $2^k$ *elements* $(x_1, x_2,...x_n)$.
**Output:** *prefix sums in the array C, where C(0,j), is the jth prefix sum*
**begin**
1. **for** $1 \le j \le n$ **pardo**
   *Set B(0,j) : = A(j)*
2. **for** $h = 1$ **to** $\log n$ **do**
   **for** $1 \le j \le n/2^h$ **pardo**
   *Set B(h,j) := B(h-1,2j-1) * B(h-1,2j)*
3. **for** $h = \log n$ **to** $0$ **do**
   **for** $1 \le j \le n/2^h$ **pardo**
   {*j =1 :set C(h,1) := B(h,1)*
   *j even :set C(h,j) := C(h+1,j/2)*
   *j odd > 1 :set C(h,j) :=* C(h+1,(j-1)*/2) * B(h,j)}*

**end**



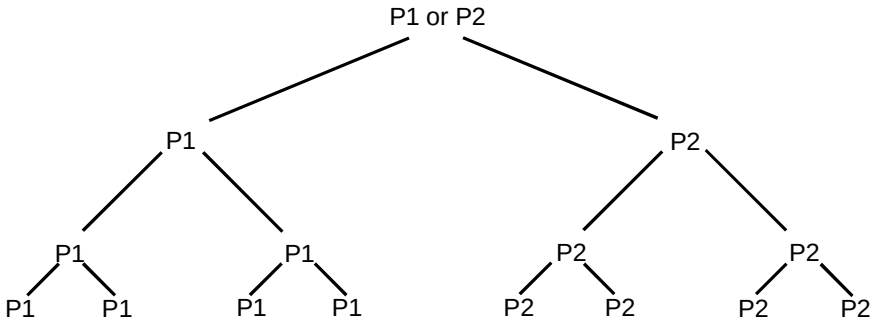

Balanced tree, nonrecursive prefix-sums algorithm.
The bottom-up traversal of the binary tree.

**Figure 1.2.3**




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

**Figure 1.2.4**




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.

**Figure 1.2.5**

## 1.2.2 Pointer jumping

The technique of **pointer jumping** consists of the following actions. A tree represents the task. In this tree, the method updates the successor of each node by the successor's successor. The successor of each node is its parent at the beginning, later – as the algorithm

moves on – an 'older' ancestor. Repeatedly applying the technique we get closer and closer to the root. The distance between the node and its successor doubles until the root is reached. After *k* iterations the distance becomes $2^k$ unless the root has been reached.

The technique is demonstrated on the example of **finding the roots** of a forest. Our original forest *F* consists of rooted directed trees. The forest is defined by an array of length *n.* Let denote the array *P*, and an array element (a parent) *P(i) = j*, if *(i,j)* is an arc (connecting the parent node *i*, and its child *i*). The root is denoted by *P(i) = i.* The task is to determine the root *R(j)* of the tree that contains node *j*.

A sequential algorithm would start from determining the roots, then perform a traversal (depth-first or breadth-first) from each root to the leaves. Thus the cost of the sequential algorithm is *O(n).*

In Figure 1.2.6 a forest containing two trees is shown. According to our **pointer jumping** or **path doubling algorithm,** at the beginning each node (which is not next to the root) 'jumps over' its parent to its grandparent as it is shown in Fig.1.2.6.b. Then this step will be repeated until the depth of the trees becomes 1. In our figure this is the second step. Now, we get the information which node the leaves originally belonged to. Our algorithm runs *O(log h)* time, where *h* is the maximum height of the forest.

Let us now try to formulate the algorithm:

**Input:** *A forest of rooted directed trees*
**Output:** f*or each i, root R(i) of the tree*
**begin:**
1. **for** $1 \le i \le n$ **pardo**
Set *R(i): = P(i)*
**while** *(R(i))≠R(R(i))* **do**
Set *S(i):=R(R(I))*
**end.**

The task can be extended to find the **sum of weights of the nodes** on the path from any node to the root**.** For this let us assume, that each node has a weight *W(i).* For calculating the weights along the paths we should generate 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** $1 \le i \le n$ **pardo**
Set *S(i):=P(i)*
**while** *(S(i))≠S(S(i))* **do**
Set *W(i):=W(i)+W(S(i))*
Set *S(i):=S(S(I))*
**end.**

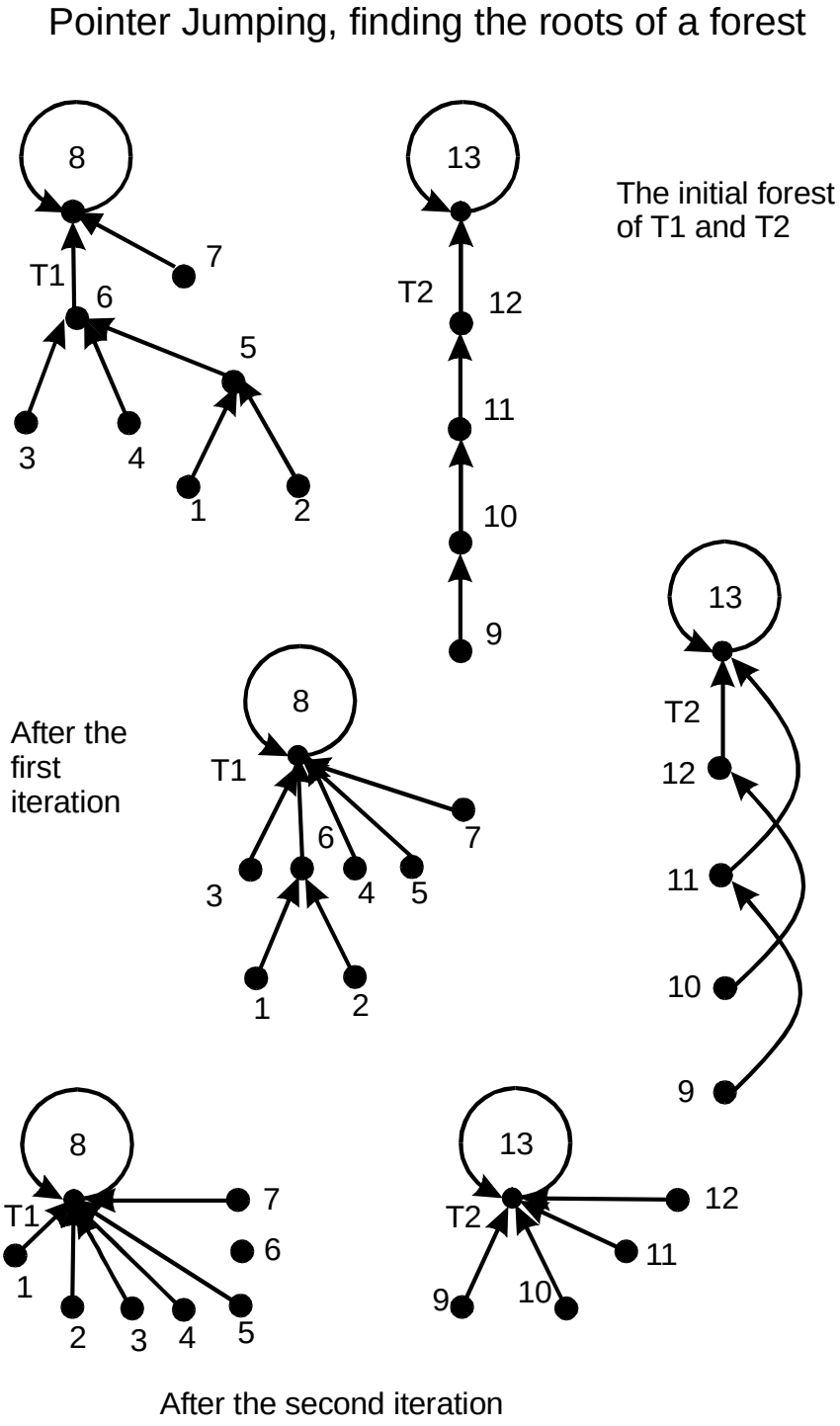

Pointer Jumping, finding the roots of a forest
8
7
T1
6
5
3
4
1
2
13
T2
12
11
10
9
The initial forest
of T1 and T2
After the
first
iteration
8
T1
7
6
3
4
5
1
2
13
T2
12
11
10
9
8
T1
7
6
1
2
3
4
5
13
T2
12
11
9
10
After the second iteration

**Figure 1.2.6**

## 1.2. 3 Divide and conquer

The basic idea of the strategy is simple. It consists of three steps.

- Partitioning or dividing input data into several partitions possibly of near equal sizes
- Solving the in this way obtained sub-problems recursively.
- Merging the solutions of the sub-problems into the solution of the whole problem.

Note: the sub-problems may be solved concurrently.

This strategy depends very much on the success of the problem partitioning and the subsolutins merging. The strategy is successfully applied to sequential algorithms too, since the number of partitioning steps is in logarithmic dependence with the input size, and if the problem size in sub-problems decreases linearly (i.e. faster), then the solution becomes more efficient. Famous examples for the strategy are the Fast Fourier Transformation (FFT) algorithms. The discussion of FFT algorithms is beyond the content of this material.

Another example may be the **convex hull problem** discussed by this material in the Chapter 4: Geometric algorithms.

For illustration of the technique we use the problem of *element ranking.* Given a sequence $A=\{a_1, a_2, a_n\}$ of $n$ ordered elements, the rank of an element $a$ (denoted by *rank(a:A)*) is the number of elements in $A$ that are less or equal to $a$. Suppose the task is to find the rank of a newly given value $a'$.

The **divide and conquer** strategy may be applied to this problem in a very simple way. We compare $a'$ with the element $a_m$ in the middle of the sequence $A$, and based on the result of the comparison, the new search may be restricted to the upper or the lower half of the sequence. Let denote this half $A'$. Now the comparison is continued with the middle element of this new $A'$ sequence. Recursively repeating this comparison until we may isolate $a$ in-between two successive elements of $A$. By definition for the rank, the index of the smaller element where we finish the process is the rank for $a'$.

## 1.2.4. Partitioning

According to this strategy:

- The whole problem is divided up into $p$ subproblems that can be handled independently.
- The $p$ sub-problems are solved concurrently.

Though the basic idea looks similar to the divide and conquer strategy, there the main problem is how to combine the results of the sub-solutions into the final solution. Here we mostly treat independent subproblems; the main task is the **partitioning** task itself.

An example, which serves for the demonstration of the strategy, is the **merging algorithm.**

In our example we have two nondecreasing sequences $A(a_1, a_2, ..., a_n)$ and $B(b_1, b_2, ..., b_m)$. The task is merging them into a sorted sequence $C(c_1, c_2, ..., c_{n+m})$. Applying the partitioning strategy, an optimal algorithmic solution could consists of the following steps:

- Choosing approximately *n/log n* elements of *A* and *B.* This splits both *A* and *B* into approximately equal blocks.
- Ranking each of the selected elements into the other sequence. This step reduces the task into merging parts of subsequences, each of which with *O(log n)* elements.
- Applying the known sequential algorithms of merging to the subsequences. Note that the subsequences can be independently and thus concurrently treated.
- Merging the subsequences.

Note that when an element *b* is ranked in sequence *A* and *A* is broken into parts according to this, one part *A'* of *A* contains elements smaller than *b*, the other one *A"* contains greater items. Similarly *b* splits the sequence *B* into such two parts *B'* and *B".* So, if merging independently the corresponding parts *A'* and *B'* into *C'* and *A" and B"* into *C",* then we get the resulting two parts of *C* such that all of the elements in *C'* are smaller (or at most equal) than the smallest element in *C".* So *C* is obtained simply by concatenating its two parts.

The **partitioning technique for the merging problem:**

**Input:** arrays $A(a_1,..., a_n)$ and $B(b_1,...,b_m)$ (both *log m and m/log m* are integers).
**Output:** $k(m)=m \log m$ pairs $(A_i, B_i)$ of subsequences, such that **a.)** $|B_i|=\log m$; **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** $1 \le i \le k(m)-1$ **pardo**
   2.1. *Rank* $b_{ilogm}$ *in A, and let* $j(i) = rank\ (b_{ilogm}:A)$
3. **for** $0 \le i \le k(m)-1$ **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.**

For illustration let us see the following example:

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

*m=4; k(m)=m/log m=2;*

*Rank* $b_{ilogm}$*=in A,* i.e. *rank (10:A)=3,*

Thus the following two pairs of sequences are obtained:

**1.** *A'=(3,7,8)* and *B'=(2,10)*

**2.** *A"=(12,13,17,23,27)* and *B"=(20,30)*

As it has been mentioned before each element in *A'* and *B'* is smaller then any element in the second pair of sequences. Hence we simple can merge the pairs to get the resulting *C* sequence of *M+N* elements.

$$C'=(2,3,7,8,10); \quad C''=(12,13,17,20,23,27,30)$$

$$C=(2,3,7,8,10,12,13,17,20,23,27,30)$$

Figure1.2.7 illustrates the merging algorithm.

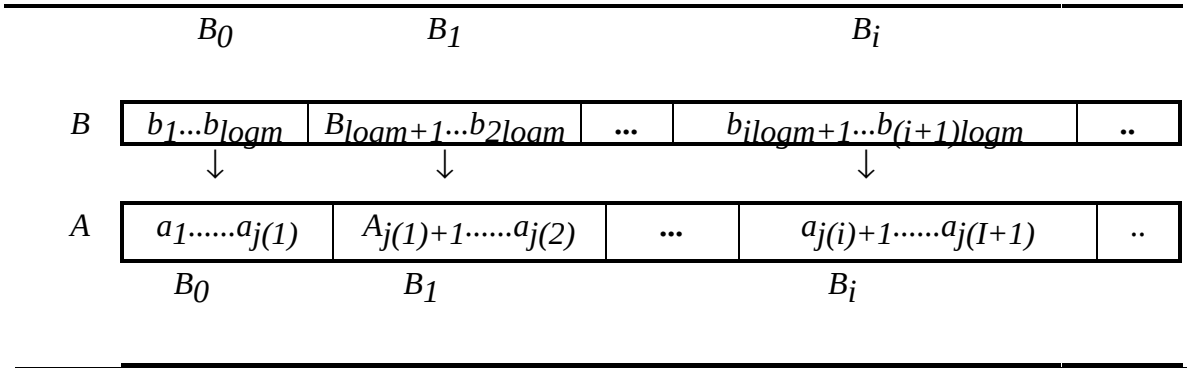



**Figure 1.2.7.**

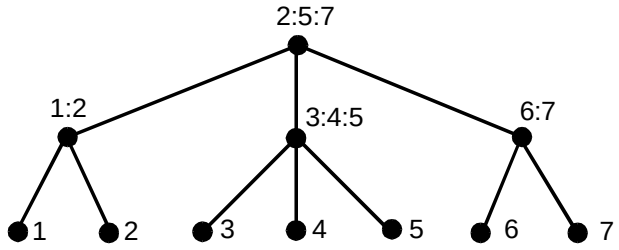
## 1.2.5 Pipelining

The technique of **pipelining** is a development strategy when the overall task is split into tasks, further each task is split into a sequence of subtasks. Processing is done in a way that when the first subtask of a task is completed, the second subtask starts a new task's first subtask can begin, in parallel with this. Thus the overall task is subdivided into subsequent tasks shifted in time and further, each task is divided into 'stages'. The operation of stages can overlap with the stages of other tasks. The operation is similar to the operation of an assembly line in an industrial production plant.

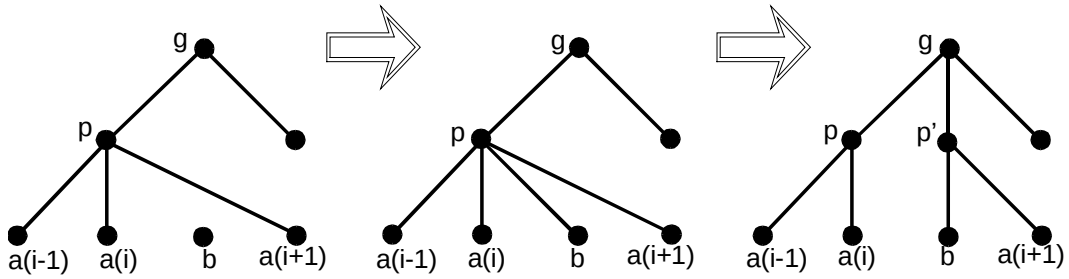
The whole operation of the given task can be viewed as a wave moving up to the next stage, and so on, up to the completion of the operation. Meanwhile the next task's wave follows the current task.

The strategy is illustrated with **insertion** operation using **2-3 trees.**

**A 2-3- tree** (Figure 1.2.8) 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, equal to the height *h* of the tree. The number of the leaves *n* is between $2^h$ (if all nodes have only two children) and $3^h$ (if all nodes have three children.




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



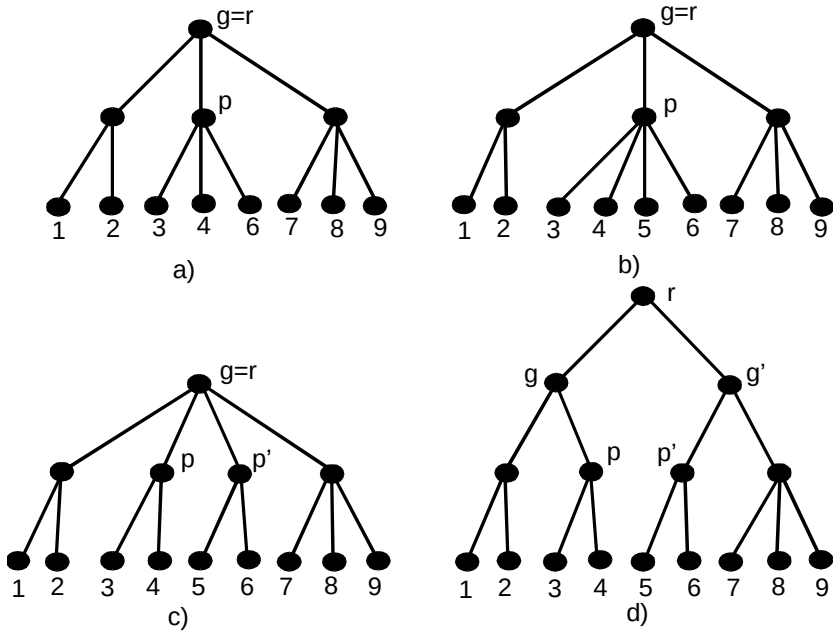

The process of inserting an item b into a 2-3 tree. 1. A node is created;2. It is assigned to parent p; 3. A new parent node is created and children assigned to it

**Figure 1.2.8**

2-3 trees are convenient structures to store sorted lists at their leaves in the order from left to the right. A sorted list *A* of items length *n* is defined: $A = (a_1, a_2, \ldots, a_n)$, such that $a_1 < a_2 < \ldots < a_n$. Let denote an internal node *v*, it holds two (or three) data item sets in its subtrees: *L[v]* with the smallest values, and *M[v]* (and *R[v]* with the greatest values).

The following *basic operations* are defined in linearly *sorted lists stored in 2-3 trees.* Naturally none of the operations can distort the order:

- searching for an element,
- inserting an element,
- deleting and element.




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.

**Figure 1.2.9**

For our example we have selected the insert operation. It consists of three phases:

- search (in order to know where to insert the element),
- inserting the element,
- restructuring the tree if anywhere 4 children appear at a node.

The **search operation** for an item $b$ to be inserted in a 2-3- tree $T$ which has $n$ sorted items $a_1 < a_2 < ... < a_n$ may be defined as finding the leaf $u$ with element $a_i$ that satisfies the following relation: $a_i \leq b < a_{i+1}$.

The search starts at the root, and always is restricted to one of the two (three) subtrees as the process proceeds toward the leaves. Please note the each node holds information on the items at the leaves of its subtrees. If $b \leq L[r]$, then we proceed in the left subtree, if $L[r] < b \leq M[r]$, in the middle tree, otherwise a right subtree exists, where the search has to go on, unless $b$ is greater than the greatest item in the sorted list. This process is repeated until the leaves are reached.

After searching and finding $u$ for which $a_i \le b < a_{i+1}$, a leaf has to be inserted for $b$ (unless $a_i = b$ ) after $u$, the child of node $p$. Let denote this new leaf $u'$. At the first, it is the child of $p$. If now $p$ has three children, then the operation is finished, otherwise – the case of four children – a new parent $p'$ has to be created, and the four children have to be allocated among the two parents $u$ and $u'$. Then, as $p'$ is inserted at a level higher, the same process has to be repeated, and so on, until we either reach somewhere three children at a parent, or we reach the root. It may happen that the root will have four subtrees now. In this case the height of the tree has to be increased, and a new node and the root node have to be created (see Figures 1.2.8 and 1.2.9.). Because the whole process goes on vertically (the search going down, the insertion climbing up) the operation execution time is $O(\log n)$.

**Inserting a sequence of items into 2-3 trees**

A 2-3 tree $T$ is given with $n$ items: $a_1 < a_2 < ... < a_n$. The task is to insert another sequence of $k$ items: $b_1 < b_2 < ... < b_k$, where $k << n$, otherwise it is faster to rebuild the whole tree.

We can build up our algorithm based on the **pipelining** technique in the following steps:

1. Using the sequential insertion algorithm, we insert the smallest $b_1$, and the largest $b_k$ items in order to eliminate the further operations to the part of the tree which is between the corresponding two leaves of the original tree $T$. All the $b_i$-s will fall in between.
2. Concurrently we locate all the items $b_i$ applying the sequential search procedure for each, separately. For each $b_i$ we create a new leaf. This will take $O(\log n)$ parallel operations for each leaf.
3. The third step is inserting with pipelining technique. This step requires a more detailed explanation, which we give in the followings.
   It may happen that more than one $b_i$-s have to be inserted between two consecutive leaves of $T$. Let denote such an ordered list of $b_i$ -s a **chain,** and its length $k_i$. The insertion of such $k_i$ elements takes a bottom-up processing, very similarly to the sequential approach. However since we have more than one element, we may do processing for them concurrently.
   In the special case, when we do not have any chain with more than one element, at most three new children may be added to the three existing ones of the parent nodes at the first (from the leaves) level. This means that now the node has at most six children. For each parent node $v$ then we may add a new parent node $v'$ and reassign the children among them. This step again can be done concurrently for all of the nodes of level one that have more than three children. This is the **simple parallel insertion algorithm** (for the case when no chain has length exceeding one)**.** It runs $O(k \log n)$ time.

In a more general case, the size of the chains $k_i$ may be greater than one. In this case we select the middle element of the chain for insertion. This reduces the size of the chain to its half. The process then is repeated for the newly formed (half-sized) chains. After *O(log k)* iterations all the chain elements will be inserted. This process may be organized in a pipeline manner. We regard a *simple parallel insertion as a* **stage.** Each stage moves up from the bottom like a wave. Once the nodes at the leaves level are processed the wave may move up to the first level and a new wave may begin at the leaves, and so on. This pipelined operation runs *O(log n + log k)* time.

## 1.2.6 Accelerated cascading

In many cases we may apply two or more different algorithms to solve the same problem. Different algorithms perform differently depending on the size of the problem. Maybe an algorithm is optimal in terms that its cost increases slowly with the size of the problem, but for the problem up to a given size, we may define a better algorithm, (e.g. because the number of processors in this case fits the smaller problem better). If this is the case, then we can use the **accelerated cascading strategy.** It works as follows:

1. Start the optimal algorithm if the size of the problem exceeds a certain value. Reduce the problem below a given size.
2. After reducing the problem size, apply the faster, but not optimal algorithm.

The success of the strategy depends on two factors: the existence of a much faster version of the algorithm for smaller input sizes, and the existence of the first algorithm such that is capable to reduce the size of the problem and then pass it to the other algorithm.

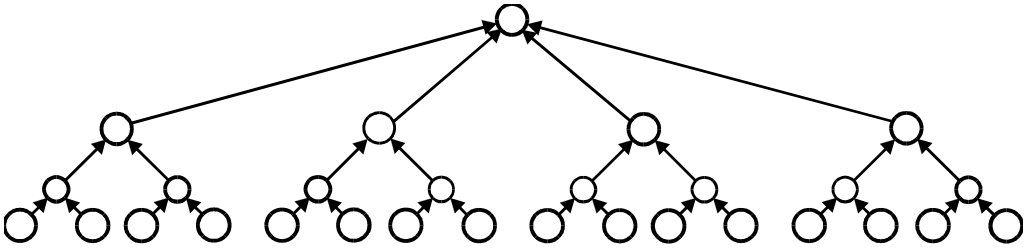
The use of the strategy of **accelerated cascading** may be demonstrated on the example of computing the **maximum element** $x_i$, in a set of elements $X = \{x_1, x_2, ..., x_n\}$, such that $x_i \geq x_j$, when $1 \leq j \leq n$.

*Maximum element search* is a similar problem to adding up numbers, where the operation – instead of addition – is comparison with a Boolean result and then selecting the maximum element for which the result of the Boolean operation is true (1). The technique based on balanced tree may be used. As we know, this method results in *O(log n)* run time. Let call this algorithm the *Basic Maximum (BM) algorithm.* It is given below.

**Input:** *An array A of p distinct elements*
**Output:** *Boolean Array of M, M(i) =1 if and only if A(i) is the maximum element.*
**begin**
1. **for** $1 \leq i,j \leq p$ **pardo**
   **if** $A(i) \geq A(j)$ **then** *Set B(i,j):=1*
   **else** *Set B(i,j):=0*
2. **for** $1 \leq i \leq p$ **pardo**
   *Set M(i):=B(i,1)∩...∩B(i,p)*

**end.**

Step 1 may be executed in a parallel computer via simultaneous read operations, while step 2, where the ∩ symbol denotes *Boolean AND* operation, may be realized by concurrent writes assuming that only the same value is allowed to write. Note that the result of comparison will be all 1-s only when the maximum value is met. A *CRCW* (concurrent-read concurrent-write) PRAM model with the *sum protocol* might be used to execute the algorithm.



Doubly logarithmic-depth tree with 16 leaves
**Figure 1.2.10**

˝

The second algorithm is the **Doubly Logarithmic-Time Algorithm.** It works on the principle of **Doubly Logarithmic-Depth Tree** - DLDT (Figure1.2.10). According to its definition the DLDT is a tree with $n = 2^{2^k}$ leaves, the root of the tree has $2^{2^{k-1}}$ children, each of the children of the root has $2^{2^{k-2}}$ children, and so on. Let us define the root's level as 0, and the node level is the number of edges on the path between the root and the node. Each node at each level (except the *k-th* level) will have $2^{2^{k-i-1}}$ children. At the *k-th* level each node has two leaves.

A DLDT has $2^{2^k-2^{k-i}}$ nodes at level *i* , at the *k-th* level $2^{2^{k-1}} = n/2$, and the depth of the tree is $k+1 = 1 + \log\log n$ .

The **algorithm** for calculating the maximum of elements on the base of DLDT is given in the following steps:

1. The maximum of the elements in each subtree is stored at the node where the subtree originates. The operations proceed bottom-up.
2. At the nodes we apply the Basic Maximum algorithm described above.

The algorithm gets its result when the calculation reaches the root of the tree. Thus we can calculate the maximum in $O(\log\log n)$ time.

According to our **accelerated cascading** strategy, we may start the calculation of the maximum value, using algorithm on the base of balanced tree. Moving up via the balanced tree we always compare the incoming two values at each node and the larger one is

forwarded further. Thus at each level of operations the size of the problem is reduced to its half. When we have already reduced the size to a treatable level (e.g. when we know that the maximum is among remaining *n' = 16 items),* then we may shift to the DLDT algorithm.

## 1.2.7 Symmetry breaking

Many problems have inherent natural symmetry. Treating symmetry may be a basic obstacle when applying parallel systems. Many processors try to work synchronously; many want to approach one part of the problem. How to start with? Which part to start with? How it effects the other parts if at the beginning one process or processor can take an arbitrary solution at a part, but the solution taken for one part is not independent of the others parts? Thus one arbitrary selected parameter may be introduced, but not more. Which processor is allowed to take the arbitrary value?

In these cases some mechanism should be introduced to break the 'symmetry' in the problem, and as a consequence the symmetry in the approach. Symmetry breaking in this case means that the inputs (or the subtasks) represented by vertices – and which are symmetrical in terms of their features – should somehow be put into different classes.

The **symmetry breaking technique** is presented via applying it to the **ring-coloring problem**. The problem looks trivial, a simple sequential algorithm would start at a vertex assigning a color to it, and then alternatively assigning another color to the next vertex. If at the last node the color becomes the same as the one the algorithm started with for the first node, then a third color will be set. For a parallel algorithm it is difficult to decide which color and which vertex it has to start with.

Suppose we have a directed cycle (graph form of a ring) *G(V,E),* both the indegree and the outdegree of the vertices is 1, and there exists a direct path from any vertex to any other one. The arcs of the graph are specified by an array of *S* of size *n. S(i) = j* if *(i,j)* is an arc. We want to determine the **n-coloring** of *G* such that no adjacent vertices will have the same color. The problem solution may be started with *k-coloring,* supposing that each vertex is given a color (e.g. the color index = the vertex index). Then our algorithm would reduce the number of different colors. The algorithm **Basic coloring** is given below. Color codes are binary numbers. Beginning with the first node, the algorithm finds the first least significant bit (LSB) position *k* in the codes of the adjacent vertices, and then assigns a new color code value equal to *2k plus the bit value of the current vertex color,* where the firs bit-difference is found: $c'(i) = 2k + c(i)_k$,

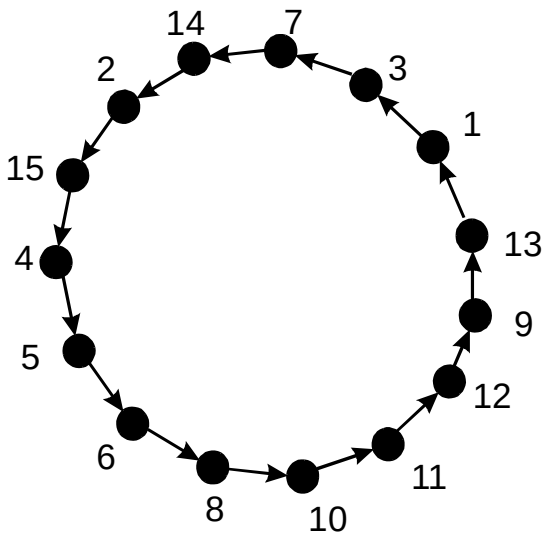
**Input:** *a directed cycle, its arcs are specified by an array S. Initial Colors* $c_i$ *= vertex number.*

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

**begin**
**for** $1 \le i \le n$ **pardo**
*1. Set k to LSB where c(i) and c(S(i)) disagree*
*2. Set c'(i): = 2k+c(i)$_k$*
**end.**

We do not show the proof, but it could be proven that the new codes would satisfy the true coloring condition in terms that colors of adjacent vertices become different, and the number of color codes reduces from the initial *n* to at *most twice the number of the most significant bit (MSB) plus 1*. (*2(log n/2)+ 1).* However, algorithm does not return the final 3-color result. After reducing the number of colors, another algorithm has to be applied to reach 3-coloring.

The coloring algorithm and its results with a ring are shown in Figure1.2.11.




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

| 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 |

**Figure 1.2.11**

## 1.2.8 Embedding networks into other network topologies

Parallel algorithms are designed with consideration to the parallel computer architecture and the topology of the interconnection network. If we want to apply a given parallel algorithm for an other parallel computer with a different communication network, then embedding the 'old' topology into the new topology may help us to solve the new communication problems, or simply show us where bottlenecks, other communication problems may appear in the new implementation.

When embedding topologies, nodes correspondent to processors and edges to communication links. With this modeling the parallel computers are represented by graphs and the problem is reduced to graph embedding.

Suppose we have two graphs: *G(V,E)* and *G'(V',E').* Our task is to map each vertex in *V* onto a vertex (or vertices) in *V',* and each edge in *E* onto an edge (or edges) in *E'.* Since during mapping not all vertices and edges are directly preserved, three parameters of the mapping process may be of importance:

- **Edge congestion** is the number of edges mapped onto a single edge in *E'.* Edge congestion means additional traffic on the communication lines. The maximum number of edges mapped into an edge is the **mapping congestion**.
- **Edge dilation** is the number of edges in *E'* that a single edge in *E* is mapped onto. Edge dilation means longer traffic paths that incur more delay in the communication. The maximum number of edges a single original edges has been mapped onto is the **dilation of the mapping**.
- **Expansion of the mapping.** *V* and *V'* may have different numbers of vertices. In this case a processor in a given architecture corresponds to more processors in the other one. The ratio of the number of processors in *V'* to the number of processors in *V* is called the expansion of the mapping.

In the followings we show a few examples for mapping. We have selected the Hypercube as the destination architecture because of several reasons. First the mapping is not trivial; secondly the architecture is very popular due to its advantageous ratio of the communication path length to the network size.

**Embedding a linear array into a hypercube**

The task is rather straightforward, a linear array or ring of $2^d$ processors may be embedded into a d-dimensional hypercube following the method given below.

Processor *i* of the array is mapped onto processor *H(i,d)* of the hypercube using the **binary reflected gray-code numbering,** for the hypercube (see also Figure1.2.14).

As an example for an 8-node ring the node numbers and the corresponding hypercube node indices (in fact these are the Gray codes) are enlisted in Table1.2.1.

**Table 1.2.1**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8-processor ring | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 3-D hypercube | 000 | 001 | 011 | 010 | 110 | 111 | 101 | 100 |

Note that not all edges of the hypercube are used, only eight of them like in the ring. So, the values of all three mapping parameters: congestion, dilation and expansion are equal to one.

**Embedding a mesh into a hypercube**

This is the natural extension of the ring embedding process. A $2^r$ by $2^c$ mesh maps directly onto a $2^{r+c}$ processor, i.e. (r+c)-dimensional hypercube. Reflected Gray code labeling also can be used if the upper bits are separated for row indices, the lower ones for column indices.

The mapping results are shown in Figures 1.2.12 and 1.2.13. Similarly to the ring mapping, the three basic parameters of the mapping are all equal to 1.

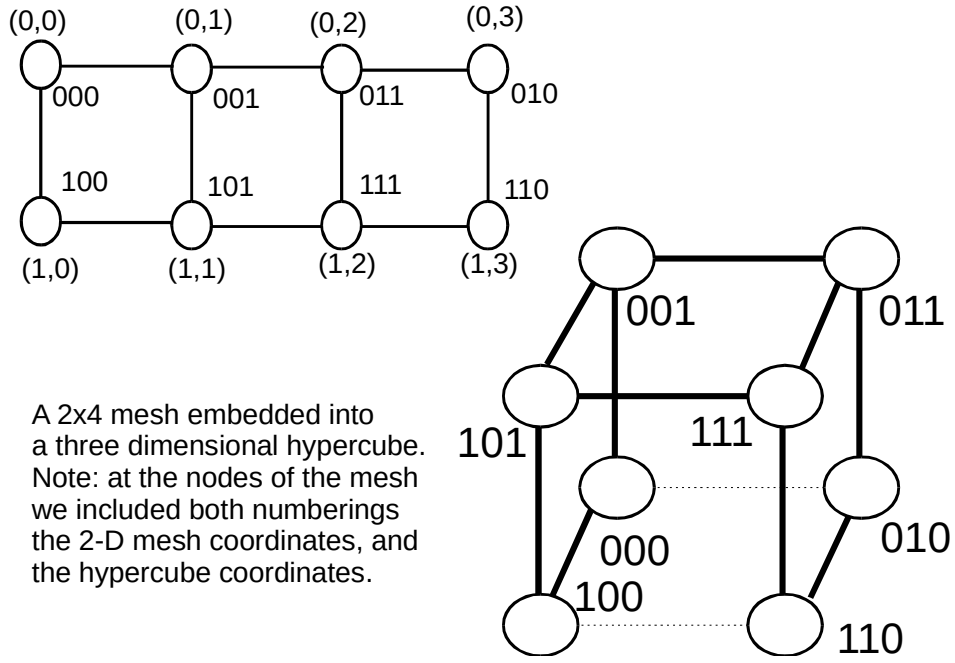
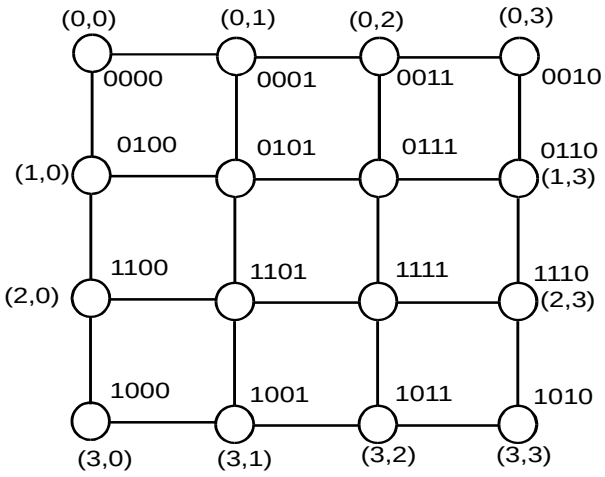



**Figure 1.2.12**




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

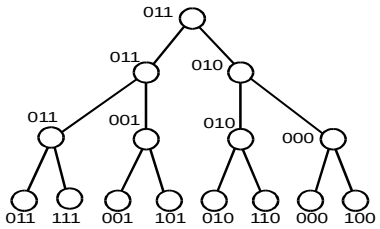
**Figure 1.2.13**

**Embedding a binary tree into a hypercube**

A binary tree of depth *d* with $2^d$ processors at its leaves may be embedded into a *d*-dimensional hypercube following the rules below:

1. The root of the tree is mapped to any processor in the hypercube
2. For the node *i* at depth *j* the left child of *i* is mapped to the hypercube processor *i,* while the right child is mapped to the processor whose index should be calculated by inverting the *j* bit in *i* index.
3. The edge between the distinct nodes of the tree will be mapped into a single edge of the hypercube.

The mapping is illustrated in Figure 1.2.14.




A tree embedded into a three-dimensional hypercube

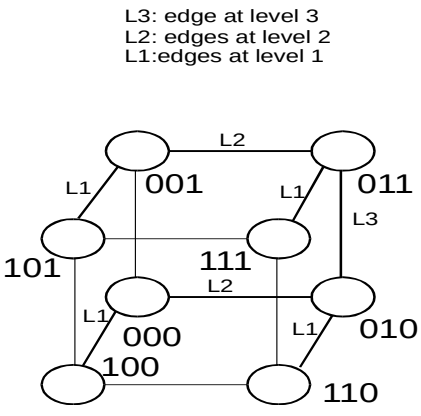



**Figure 1.2.14**