ON VALIDATING PARALLEL ARCHITECTURES VIA GRAPH EMBEDDINGS

Arnold L. Rosenberg

COINS Technical Report 87-61

ON VALIDATING PARALLEL ARCHITECTURES VIA GRAPH EMBEDDINGS

*Arnold L. Rosenberg**
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

June 24, 1987

Abstract

A viable multi-purpose parallel architecture must exhibit many qualities, including: technological feasibility, low communication overhead, robustness in the face of faults, and programmability. The maturity and reliability of techniques for detecting and studying these desiderata decrease rapidly in the order of their listing. The issues of technological feasibility and communication overhead are relatively well understood; progress is being made on the problem of detecting and enhancing the robustness of proposed architectures; but the issue of programmability has not yet even been formulated in a generally accepted way. We extrapolate here from joint work with S. N. Bhatt, F. R. K. Chung, L. S. Heath, and F. T. Leighton, to propose an avenue for assessing the programmability of a proposed parallel architecture, using the formal tool of graph embeddings. As is common, we represent the communication structure of a parallel architecture as an undirected graph, and we seek efficient embeddings of a variety of specific graph families in the architecture graph. We choose the graph families to abstract the intertask dependency structures of popular algorithmic strategies, such as divide-and-conquer and convolution. We illustrate the approach by describing recent studies of optimal embeddings in the Hypercube architecture.

1. INTRODUCTION

There has been considerable interest for several years in an architectural style that is especially well suited for integrated circuit technology, namely arrays of identical processing elements (PEs). The literature is full of proposed instances of this architectural style. In most cases, the proposed array structure is defined in terms of an undirected graph whose

*This research was supported in part by NSF Grant DCI-87-96236.

vertices represent the PEs of the array and whose edges represent the inter-PE communication links. Many of these proposals seek to establish the feasibility of their proposed array structure by arguing that it enjoys:

1. *small communication overhead*

The most simple-minded measure of communication overhead is the diameter of the underlying graph; more sophisticated studies consider also the ease of finding short inter-PE routes, and the probability that routing paths will "block" one another.

2. *technological feasibility*

The most simple-minded measure of technological feasibility is the presence of small vertex-degrees in the underlying graph; more sophisticated studies consider other issues: for instance, in a VLSI implementation, one must be able to lay the proposed array structure out on a chip or wafer in small area, with short runs of wire. A significant subset of these proposals address also the issue of the

3. *robustness* of the array, i.e., its capacity for "working around" faults.

The issue of robustness is of particular importance when one contemplates implementing the array using large-scale VLSI or WSI technology since, at least for the foreseeable future, any aggressive use of these technologies exposes one to having a positive fraction of one's PEs not survive the fabrication process. One cannot predict where these faulty PEs will lie in the array, so the capacity to avoid faulty PEs is a major issue in the evaluation of a proposed architecture.

One final issue is addressed in only a small minority of these proposals, despite its importance for any proposed multi-purpose architecture, namely,

4. *programmability,* i.e., the demonstration that the array can efficiently perform a variety of tasks.

Despite the complications mentioned above, there would seem to be rather general agreement on how to study the first two of our four issues. The issue of robustness encounters less universal agreement, there being one school of thought that advocates *a priori* avoidance of faults [7, 18, 19], by incorporating reconfigurability (via a network of switches) in the design, and one school that advocates *a posteriori* avoidance of faults [8, 9, 14], by salvaging a working sub-architecture. But even with these divergent approaches and philosophies, one is likely to obtain some level of agreement about the ability of an architectural design to survive faults; the arguments would more likely center on how efficiently the robustness is achieved.

The final issue in our list, programmability, is the most problematical since, despite its importance, no standard approaches to establishing the programmability of a proposed architectures have evolved. It is our intention here to propose and illustrate one approach to validating the programmability of a proposed parallel architecture, by means of graph embeddings.

A word about the competitors of our proposal: One finds in the literature two main approaches to establishing the programmability of an array structure, exemplified by the following. In [16] the viability of the Cube-Connected Cycles (CCC) architecture is argued by presenting a programming scheme that can be implemented efficiently on the CCC and demonstrating that a number of important computations can be specified efficiently using that scheme. This approach does indeed establish the programmability of an architecture for a variety of tasks; it does not, however, suggest how one can ascertain whether or not one's desired task can be programmed efficiently using the indicated scheme; moreover, no one has come up with an analogous scheme for most other architectures, suggesting that this approach may not be generally applicable. In [13] the viability of the Mesh-of-Trees (MT) architecture is argued by demonstrating how a variety of important computational problems can be solved efficiently on an MT. Johnsson use the same technique to show that grid-based algorithms can be implemented efficiently on the Hypercube architecture. This technique is the ultimate test for the specific algorithms studied, but it often gives no hint about how to assess the efficiency of the architecture for even closely related algorithms.

The proposal we espouse here is to try to argue the programmability of an architecture by looking at its performance on *classes* of algorithms, rather than on individual ones. The informal vehicle we propose to develop here builds on the ***logical mapping problem*** for parallel architectures, namely the problem of realizing the intertask communication structure of one's parallel algorithm on the idealized parallel architecture one has access to [2, 6]. The word "idealized" here indicates that the logical mapper, who plays the role of the programmer in a sequential environment, does not worry about a variety of physical issues such as possible faults in, or efficiency-enhancing compromises in the fabricated architecture.

We represent both algorithms and arrays as undirected graphs, with the following notations and interpretations: For parallel algorithms, the vertices V_G of the algorithm-graph G represent the tasks in the algorithm; the edges E_G of G represent the interdependencies among the tasks. These interdependencies might reflect data dependencies, as with grid- or convolution-based algorithms, or control dependencies, as with divide-and-conquer algorithms. As noted earlier, for processor arrays, the vertices V_H of the array-graph H represent the PEs of the array; the edges E_H of H represent the inter-PE communication links.

The goal of a logical mapping is to place interdependent tasks close to one another in the array, all the while utilizing array resources efficiently. Formally, we formulate a logical mapping as an *embedding* of the algorithm-graph G in the array-graph H:

- We associate tasks with PEs via a one-to-one mapping

$$\iota : V_G \rightarrow V_H$$

- We measure proximity via the *DILATION* of the embedding, namely, the farthest apart that interdependent tasks get separated in the array:

$$DILATION(\iota) = \max_{(u,v)\in E_G} \text{Dist}_H(\iota(u), \iota(v))$$

- We measure efficiency of resource utilization via
  - the *LOAD-FACTOR* of the embedding, namely, the largest number of intertask paths that traverse a single array edge – a measure of congestion
  - the *EXPANSION* of the embedding, namely, fraction of PEs that *do not* get used in the embedding; more conveniently:

$$EXPANSION(\iota) = |V_H|/|V_G|$$

The now-standard measures of DILATION and EXPANSION originated in [17].

An Aside. One might expect it to suffice to consider only one of DILATION and EXPANSION, since it is intuitive that minimizing one measure would minimize both. Such is not the case [4, 11]. For instance, one finds the following in [11]:

Theorem 1. *There is an embedding of the N-node complete ternary tree in the complete binary tree with*

$$\text{DILATION} = 2$$

and

$$\text{EXPANSION} = N^{.26}$$

Every embedding of the N-node complete ternary tree in the complete binary tree that has

$$\text{EXPANSION} < 2$$

also has

$$\text{DILATION} > (\textit{constant}) \log\log N$$

Thus, attempting to optimize either measure causes the other to grow without bound.

Validation Proposal. We propose to establish the programmability of a proposed array by seeking the best embeddings in the proposed array of a variety of families of algorithm-graphs whose structures abstract the intertask dependency structures of a variety of important algorithmic paradigms. If, on the one hand, we establish the existence of embeddings

with $O(1)$ DILATION, $O(1)$ LOAD-FACTOR, and $O(1)$ EXPANSION, then we shall conclude that the proposed architecture can be used to implement efficiently any algorithm in the class represented by the graph family: one simply uses that embedding as the logical mapping. If, on the other hand, we fail to find efficient embeddings, or even if we can prove that none exist, we shall not be able to draw any definitive conclusion therefrom, for our notion of embedding is static – tasks are preassigned to PEs and, once assigned, are never moved; even if no static assignment strategy exists, there might be some dynamic assignment strategy that leads to an efficient algorithm implementation. Thus the strengths of our proposed approach result from

- its forcing the array designer to focus on algorithms instead of programs
- its precision and rigor

The weaknesses of the approach are embodied in the problem of "false negatives," i.e., the fact that our proposal yields only a *semi*-decision procedure for the existence of efficient implementations.

To develop our proposal further, we begin a list of important graph families that would be of especial interest in any attempt to verify the programmability of a proposed multipurpose parallel architecture.

- *arbitrary binary trees:* (cf. Fig. 1)
  - For our purposes, a binary tree is a connected acyclic graph one of whose vertices is bivalent (the *root*) and all of whose other vertices are either univalent (the *leaves*) or trivalent. The tree is *complete* if all root-to-leaf paths have the same length (which is called the *height* of the tree). We denote by $T(h)$ the height-h complete binary tree.
  - Binary trees are paradigmatic for divide-and-conquer algorithms in the sense that each task of such an algorithm spawns either zero or two subtasks which never interact until they return control to their common parent.
- *FFT graphs:* (cf. Fig. 2)
  - Let m be a positive integer. The 2^m-*input FFT graph* denoted $F(m)$, is defined as follows. $F(m)$ has vertex-set

$$V_m = \{0, 1, \cdots, m\} \times \{0, 1, \cdots, 2^m - 1\}$$

    The subset $V_{m,\ell} = \{\ell\} \times \{0, 1, \cdots, 2^m - 1\}$ of V_m $(0 \leq \ell \leq m)$ is called the ℓ^{th} *level* of $F(m)$; vertices in $V_{m,0}$ are called *inputs*, and vertices in $V_{m,m}$ are called *outputs* (in deference to the algorithmic origin of the graph). The edges of $F(m)$ form *butterflies* (or, copies of the complete bipartite graph $K_{2,2}$) between consecutive levels of vertices. Each butterfly connects vertices

$$\langle \ell, \alpha 2^{\ell+1} + \beta \rangle \text{ and } \langle \ell, \alpha 2^{\ell+1} + \beta + 2^{\ell} \rangle$$

on level ℓ of $F(m)$ $(0 \leq \ell < m;\ 0 \leq \alpha < 2^{m-\ell-1};\ 0 \leq \beta < 2^{\ell})$ with vertices

$$\langle \ell + 1, \alpha 2^{\ell+1} + \beta \rangle \text{ and } \langle \ell + 1, \alpha 2^{\ell+1} + \beta + 2^{\ell} \rangle$$

on level $\ell + 1$. One can view $F(m)$, $m \geq 2$ ($F(1) = K_{2,2}$ being given), as being constructed inductively, by taking two copies of $F(m-1)$ and 2^m new output vertices, and constructing butterflies connecting the k^{th} outputs of each copy of $F(m-1)$, on the one side, to the k^{th} and $(k+2^{m-1})^{\text{th}}$ new outputs, on the other side. Thus, $F(m)$ has $(m+1)2^m$ vertices and $m2^{m+1}$ edges.

- The FFT graphs are paradigmatic for convolution-based algorithms since they reflect the data-dependency structure of the 2^m-input FFT algorithm – cf. [1, Ch. 7] – and related convolution-based algorithms.

- *rectangular grids:* (cf. Fig. 3)
  - The $d_1 \times d_2 \times \cdots \times d_k$ *rectangular grid* has vertices

$$\{1, \cdots, d_1\} \times \{1, \cdots, d_2\} \times \cdots \times \{1, \cdots, d_k\}$$

    and edges connecting each vertex of the form $\langle x_1, \ldots, x_i, \ldots, x_k \rangle$ with vertices $\langle x_1, \ldots, x_i \pm 1, \ldots, x_k \rangle$. Each d_i is called a *dimension* of the grid.
  - A large family of numerical algorithms naturally have data dependencies that are based on the rectangular grid [12].

Other families of graphs can be identified by perusing the literature. Just one more example are the so-called *refinable rectangular grids* [15], which appoximate the structure of finite-element algorithms.

We shall now present a case study to illustrate the application and implementation of our evaluation proposal. We focus here on one particular popular architecture, namely, the (Boolean) Hypercube. This architecture has been implemented commercially by BBN, Intel, N-cube, and Thinking Machines.

- Let d be a nonnegative integer. The *d-dimensional (Boolean) Hypercube* $C(d)$ is the graph whose vertices are all binary strings of length d and whose edges connect each string-vertex x with the d strings that differ from x in precisely one bit (position); cf. Fig. 4. Thus, $C(d)$ has 2^d vertices and $d2^{d-1}$ edges.

2. OPTIMAL EMBEDDINGS IN THE HYPERCUBE

2.1. A Useful Sample Embedding

In order to prepare the reader for the more complicated embeddings in subsequent sections, we present here a simple embedding of the complete binary tree $T(d)$ in the Hypercube

$C(d)$. This embedding, which comes from [3], has DILATION 2, LOAD-FACTOR 2, and EXPANSION 1. The embedding is "computed" in three steps.

1. Perform an *inorder* traversal of $T(d)$, numbering the nodes from 0 to $2^d - 2$ as one goes; cf. Fig. 5. Recall that inorder traversal:
   - visits the left subtree in inorder;
   - visits the root;
   - visits the right subtree in inorder.
2. Convert the inorder numbering from Step 1 to a string-labelling by converting each number to its length-d binary representation (padding with leading 0's where necessary); cf. Fig. 6.
3. Assign each vertex of $T(d)$ to the vertex of $C(d)$ indicated by the tree-vertex's string-label.

It is obvious that the proposed mapping is one-to-one, hence an embedding of $T(d)$ in $C(d)$, and that the embedding has unit EXPANSION. To show that the embedding has DILATION 2 and LOAD-FACTOR 2, one can verify easily that the left and right children of the level-ℓ vertex

$$\beta = \beta_0\beta_1\cdots\beta_\ell\beta_{\ell+1}\cdots\beta_{d-1}$$

of $T(d)$ (the left child being visited before β in the inorder traversal and the right child being visited after β) are, respectively,

$$\beta_0\beta_1\cdots\beta_\ell\overline{\beta}_{\ell+1}\cdots\beta_{d-1}$$

and

$$\beta_0\beta_1\cdots\overline{\beta}_\ell\overline{\beta}_{\ell+1}\cdots\beta_{d-1}$$

Since adjacencies in the Hypercube flip single bits, our claims about DILATION and LOAD-FACTOR follow.

For this embedding problem, optimal (i.e., unit) DILATION and EXPANSION are not simultaneously accessible. To wit, $T(d)$ and $C(d)$ are both bipartite (i.e., composed of red and blue vertices, with all edges connecting a red vertex with a blue one), but the ratios between the numbers of red and blue vertices are very different for the two graphs.

The illustrated embedding itself will be useful later. But there are two other lessons to learn from this example:

1. Embeddings in Hypercubes can be specified via labellings with binary strings.
2. One must often settle for nearly optimal embeddings if one wants to consider more than one cost measure.

2.2. Divide-and-Conquer Algorithms

This section is devoted to a result by Bhatt, Chung, Leighton, and Rosenberg [3] that shows that the divide-and-conquer control structure can be implemented on the Hypercube architecture efficiently. This assertion translates in our framework to the following theorem.

Theorem 2. *Every binary tree can be embedded in a Boolean hypercube with constant DILATION, constant LOAD-FACTOR, and constant EXPANSION.*

Proof Strategy: The theorem is proved in three stages, using two auxiliary graphs, the bucket-tree and the thistle-tree.

An *N-vertex bucket-tree* is an N-vertex complete binary tree whose vertices are *supernodes*: there is a constant c such that each vertex at level ℓ of the bucket-tree can, in an embedding, hold as many as

$$6 \log \frac{N}{2^\ell} + 18$$

vertices of the guest graph G.

A *height-h thistle-tree* is obtained from a height-h complete binary tree by appending (via edges) to each level-ℓ vertex of the complete tree $h - \ell$ new leaf vertices called *thistles*; cf. Fig. 7. The thistle-tree vertices that are inherited from the underlying complete binary tree are called its *primary* vertices.

The three stages of the embedding of a given binary tree T in a Hypercube are:

1. Embed the given binary tree T in a bucket-tree.
2. Embed the resulting bucket-tree in a thistle-tree.
3. Embed the resulting thistle-tree in a Hypercube.

A. Embed a Binary Tree T in a Bucket-Tree

Lemma 1. *An N-node binary tree can be embedded with DILATION 3 in an N-node bucket-tree.*

Proof Sketch.

1. Recursively node-bisect the binary tree T, thereby implicitly creating a decomposition tree for T: the root of the decomposition tree is T; the children of the root are the subgraphs T_1 and T_2 of T created by the first bisection; the children of the vertex representing T_i are the subgraphs T_{i1} and T_{i2} created by bisecting T_i; and so on.

2. Make the decomposition tree into a bucket-tree by converting each vertex into a super-node with the appropriate capacity.

3. Proceeding down the bucket-tree:

   - Place the bisector nodes from each bisection in the corresponding bucket (i.e., at the corresponding super-node).
   - Also place at that super-node any nodes of T that must be placed there in order to maintain the desired bound on DILATION.

In [3] it is shown how to accomplish Step 3 with super-nodes of the indicated sizes, using the *multi-color separator theorem for trees* [5]:

Theorem 3. *Let T be an N-node binary tree, with N_i nodes of color i*

$$1 \leq i \leq K; \quad \sum N_i = N.$$

By removing $\leq K \log N$ nodes, one can bisect T so that, for each $1 \leq i \leq K$, the two halves have equally many i-colored nodes.

B. Embed a Bucket-Tree in a Thistle-Tree

Lemma 2. *Every bucket-tree can be "realized" with DILATION $O(1)$ by a thistle-tree.*

Proof Sketch: Distribute the contents of a given super-node of the bucket-tree among the thistles at the corresponding vertex of the thistle-tree. This places multiple tree-vertices at the higher-level vertices of the thistle-tree, with a corresponding deficit at the lower-level vertices. In [3] it is shown how to "push" the excess vertices down a few levels without increasing DILATION more than a bounded amount.

As an immediate corollary we have:

Lemma 3. *Any N-vertex binary tree can be embedded with DILATION $O(1)$ in an N-vertex thistle-tree.*

C. Embed a Given Thistle-Tree in a Hypercube

We embed a thistle-tree with $N = 2^n - 1$ primary vertices in the Hypercube $C(n+1)$ in two stages.

1. First, we take the n-vertex complete binary tree underlying the thistle-tree, and inorder-label it with binary strings, just as we did in the previous section.

Note that for each vertex v of the complete binary tree, the string labelling v differs in exactly one bit-position from each of the strings labelling the vertices along the right spine of the subtree rooted at v's left child.

2. Map the primary vertices of the thistle-tree on the corresponding vertices of the complete binary tree. Map the thistles attached to each vertex v in any manner on the vertices along the right spine of the subtree rooted at v's left child.

The described mapping clearly has DILATION 2, but it is not an embedding, since it places two vertices of the thistle-tree at each image vertex of the Hypercube. Our final step fixes this problem, at the cost of increasing EXPANSION.

3. Take a second copy of $C(n)$ (which is connected to $C(n)$ by a matching). Use each vertex of this shadow-Hypercube to hold the thistle-vertex that resides at the corresponding vertex of the original copy of $C(n)$.

By using this shadow Hypercube, we have thus embedded a thistle-tree in a Hypercube with DILATION 2 and EXPANSION 2.

Composing the mappings in subsections A, B, and C completes the proof concerning DILATION and EXPANSION. A more detailed analysis of the steps of the proof establishes the claim about LOAD-FACTOR also.

3. CONVOLUTION-BASED ALGORITHMS

The Hypercube is a very congenial host for convolution-based algorithms also, as the following result of Heath and Rosenberg [10] indicates.

Theorem 4. *Every FFT graph is a subgraph of the smallest Boolean hypercube that will hold it; this gives an embedding with simultaneous optimal DILATION, optimal LOAD-FACTOR, and optimal EXPANSION.*

Proof Strategy. Let us focus on embedding the FFT graph $F(m) = (V_m, E_m)$ in the Hypercube $C(m + \lceil \log_2(m+1) \rceil)$. We specify the desired embedding by describing two labelling schemes.

- We assign each vertex $v \in V_m$ a unique d-bit label $L(v)$ (so that the labelling specifies an embedding).
- We assign each level ℓ of $F(m)$ a *bit-position pair* (a_ℓ, b_ℓ) so as to satisfy the following: For each edge (u, v), where u is at level ℓ of $F(m)$ and v is at level $\ell + 1$, the labels $L(u)$ and $L(v)$ differ either in bit-position a_ℓ or in bit-position b_ℓ.

For each $i \in \{1, 2, \cdots, m\}$, there is a *bit-pair* (a_i, b_i) of bit-positions that are used for assignments to edges between levels $i-1$ and i of $F(m)$; i.e., all such butterfly edges "flip" the same pair of bits.

One verifies by induction that when we assign the d-bit label $L(v)$ to any single vertex v of $F(m)$, the labels of all remaining vertices can be specified uniquely by specifying the *levelled bit-pair sequence* (*LBPS*) $S = (a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)$. We can, therefore, specify the desired embedding by labelling some input of $F(m)$ with the string $00\cdots0$ and using an appropriate LBPS to specify the labelling of the remaining vertices.

It remains to select an LBPS so that the labelling L is injective. To this end, call the LBPS $S = (a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)$ *proper* if, for each i, at least one of a_i or b_i does not occur in $\{a_j, b_j : j < i\}$, i.e., at an earlier level; call an a_i or b_i satisfying this condition *new*. We shall find a proper LBPS to effect the desired labelling by visiting $F(m)$ level by level.

The advantage to using a new bit-position in our labelling is, of course, that labels obtained by flipping that bit-position are guaranteed never to have appeared at earlier levels. Since $F(m)$ has $(m+1)2^m$ vertices, it is clear that we can always use one new bit-position at every level. Indeed, when the level we are visiting is a power of 2, then we can use two new bit-positions. Our problem reduces, therefore, to deciding how to choose the not-new bit-position at each level of $F(m)$ that is not a power of 2. We accomplish this by partitioning the levels into blocks between adjacent powers of 2. Assuming with no loss of generality that the a-position at each level is always new, we choose the b-position as follows.

- For $1 \leq i \leq 2^{k-1} - 1$, we set $b_{2^k+i} = a_{2^{k-1}+i}$.
- For $i = 2^{k-1}$, we set $b_{2^k+i} = a_{2^{k-1}}$.
- For $1 \leq i \leq 2^{k-1} - 1$, we set $b_{2^k+2^{k-1}+i} = b_{2^k+i}$.

The proof that the specified labelling is injective proceeds in two steps. First one shows that if the labelling is not injective, then it must fail to be so on the height-m complete binary tree rooted at vertex $00\cdots0$ (whose leaves are the level-m vertices of $F(m)$). Then one shows that if there are any repeated labels in this tree, then there must be an instance of repeated labels the first few levels of the tree. Finally, one verifies directly that such repetitions do not occur early in the tree.

4. GRID-BASED ALGORITHMS

Our treatment of rectangular grids is restricted to the important subcase wherein each dimension is a power of 2. This subcase occurs commonly since it often entails simplified programming.

For this special case, the following result is easily established.

Theorem 5. *For arbitrary integers $d_1, d_2, \ldots, d_k$, the $2^{d_1} \times 2^{d_2} \times \cdots \times 2^{d_k}$ grid is a subgraph of the Boolean Hypercube $C(d_1 + d_2 + \cdots + d_k)$; this gives an embedding with simultaneous optimal DILATION, optimal LOAD-FACTOR, and optimal EXPANSION.*

One proves the theorem easily by noting that if one folds the grid in half along any dimension, the edges connecting the two halves form a matching; but the same is true if one focusses on any one dimension of the Hypercube. Thus, successively folding the grid in half, and dedicating one dimension of the Hypercube to each fold, yields the sought embedding immediately; cf. Fig. 8.

5. CONCLUDING REMARKS

Despite all of its shortcomings, the proposed validation scheme has certain advantages, such as precision, rigor, and generality, not shared by the competitors that have thus far found their way into print. It should, therefore, be considered one useful implement in the toolbox of the architectural designer.

ACKNOWLEDGMENT: It goes without saying that this paper owes its existence to my friends and collaborators Sandeep Bhatt, Fan Chung, Lenny Heath, and Tom Leighton.

6. REFERENCES

1. A.V. Aho, J.E. Hopcroft, J.D. Ullman (1974): *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

2. F. Berman and L. Snyder (1984): On mapping parallel algorithms into parallel architectures. *Intl. Conf. on Parallel Processing.*

3. S.N. Bhatt, F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1986): Optimal simulations of tree machines. *27th IEEE Symp. on Foundations of Computer Science*, 274-282.

4. S.N. Bhatt and F.T. Leighton (1984): A framework for solving VLSI graph layout problems. *J. Comp. Syst. Sci. 28*, 300-343.

5. N. Blum (1983): An area-maximum edge length tradeoff for VLSI layout. *16th ACM Symp. on Theory of Computing*, 92-97.

6. S.H. Bokhari (1981): On the mapping problem. *IEEE Trans. Comp., C-30*, 207-214.

7. F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1983): DIOGENES – A methodology for designing fault-tolerant processor arrays. *13th Intl. Conf. on Fault-Tolerant Computing*, 26-32.

8. J.W. Greene and A. El Gamal (1984): Configuration of VLSI arrays in the presence of defects. *J. ACM 31*, 694-717.

9. J. Hastad, F.T. Leighton, M. Newman (1986): Reconfiguring a hypercube in the presence of faults. Typescript, MIT.

10. L.S. Heath and A.L. Rosenberg (1987): An optimal mapping of the FFT algorithm onto the Hypercube architecture. Tech. Rpt., Univ. of Massachusetts; submitted for publication.

11. J.-W. Hong, K. Mehlhorn, A.L. Rosenberg (1983): Cost tradeoffs in graph embeddings. *J. ACM 30*, 709-728.

12. L. Johnsson (1985): Basic linear algebra computations on hypercube architectures. Tech. Rpt., Yale Univ.

13. F.T. Leighton (1984): Parallel computation using meshes of trees. *1983 Workshop on Graph-Theoretic Concepts in Computer Science*, Trauner Verlag, Linz, pp. 200-218.

14. F.T. Leighton and C.E. Leiserson (1985): Wafer-scale integration of systolic arrays. *IEEE Trans. Comp., C-34*, 448-461.

15. R.J. Lipton, A.L. Rosenberg, A.C. Yao (1980): External hashing schemes for collections of data structures. *J. ACM 27*, 81-95.

16. F.P. Preparata and J.E. Vuillemin (1981): The cube-connected cycles: a versatile network for parallel computation. *C. ACM 24*, 300-309.

17. A.L. Rosenberg (1981): Issues in the study of graph embeddings. In *Graph-Theoretic Concepts in Computer Science: Proceedings of the International Workshop WG80*, Bad Honnef, Germany (H. Noltemeier, ed.) *Lecture Notes in Computer Science 100*, Springer-Verlag, New York 150-176.

18. A.L. Rosenberg (1983): The Diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. Comp., C-32*, 902-910.

19. L. Snyder (1981): Overview of the CHiP computer. In *VLSI 81: Very Large Scale Integration* (J.P. Gray, ed.) Academic Press, London, pp. 237-246.

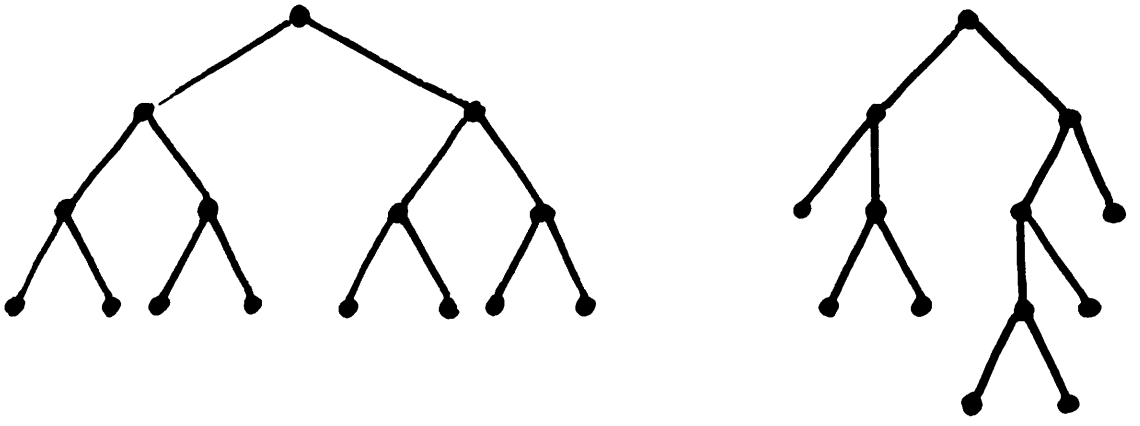

Figure 1. Sample binary trees; the lefthand one is *complete*.

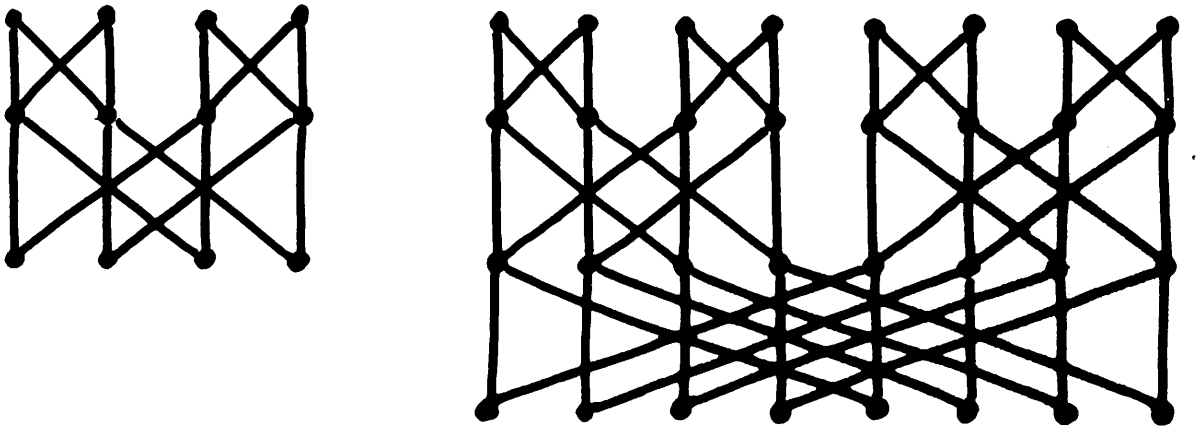

Figure 2. The FFT graphs $F(2)$ and $F(3)$.

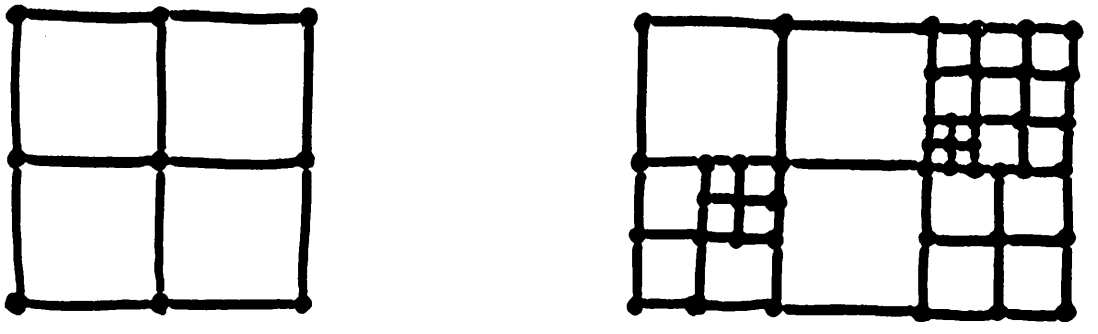

Figure 3. A sample rectangular grid (left) and refinable grid (right).

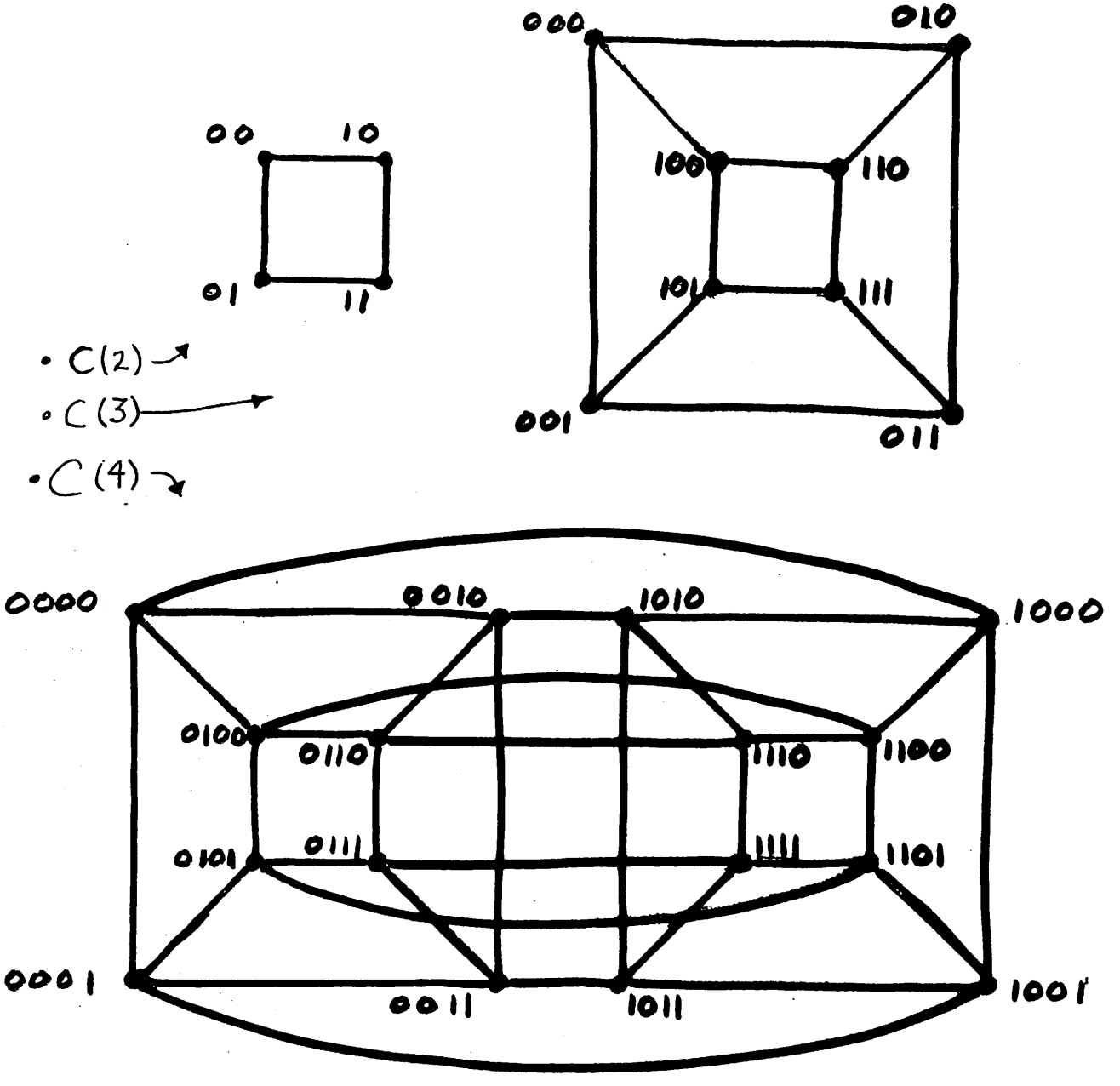

Figure 4. The Boolean Hypercubes
$C(2)$, $C(3)$, $C(4)$.

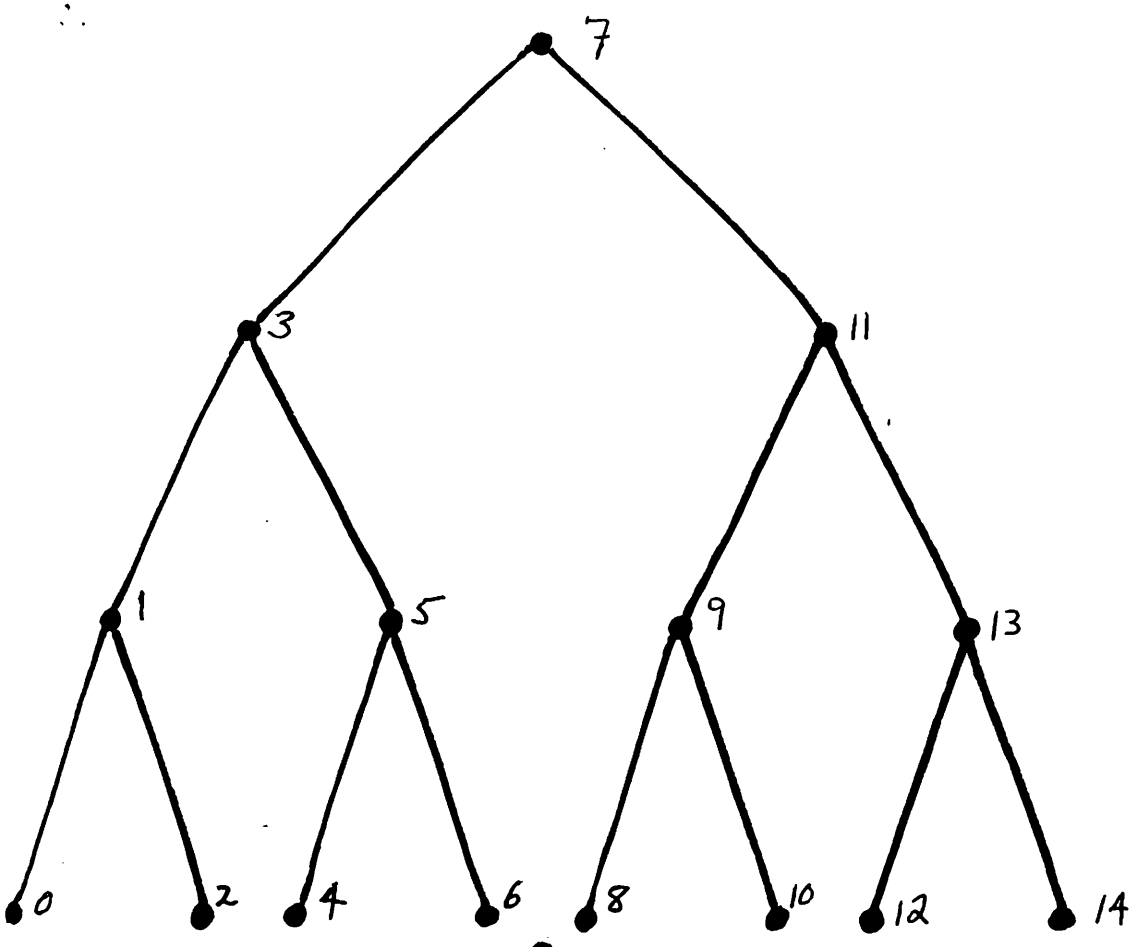

Figure 5. The inorder numbering of the complete binary tree.

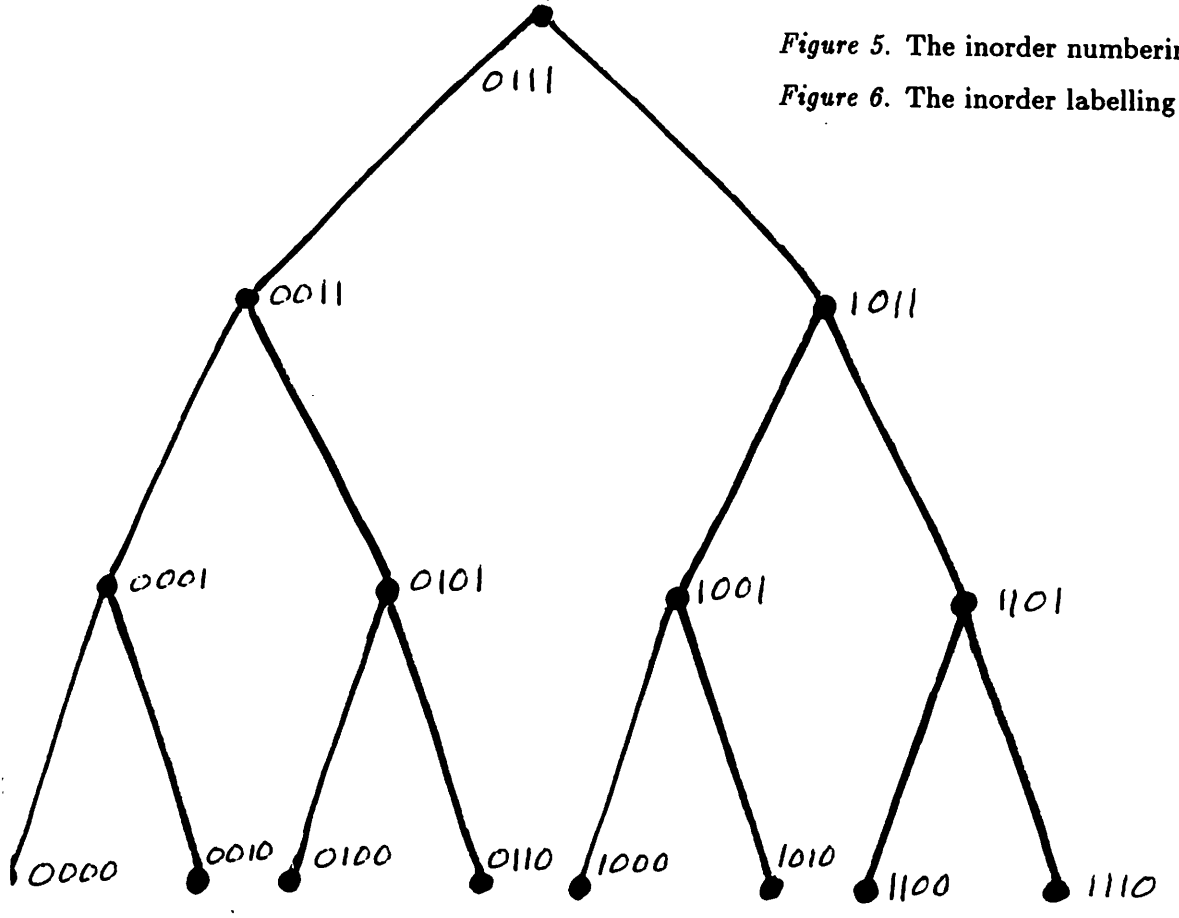

Figure 6. The inorder labelling of the complete binary tree.

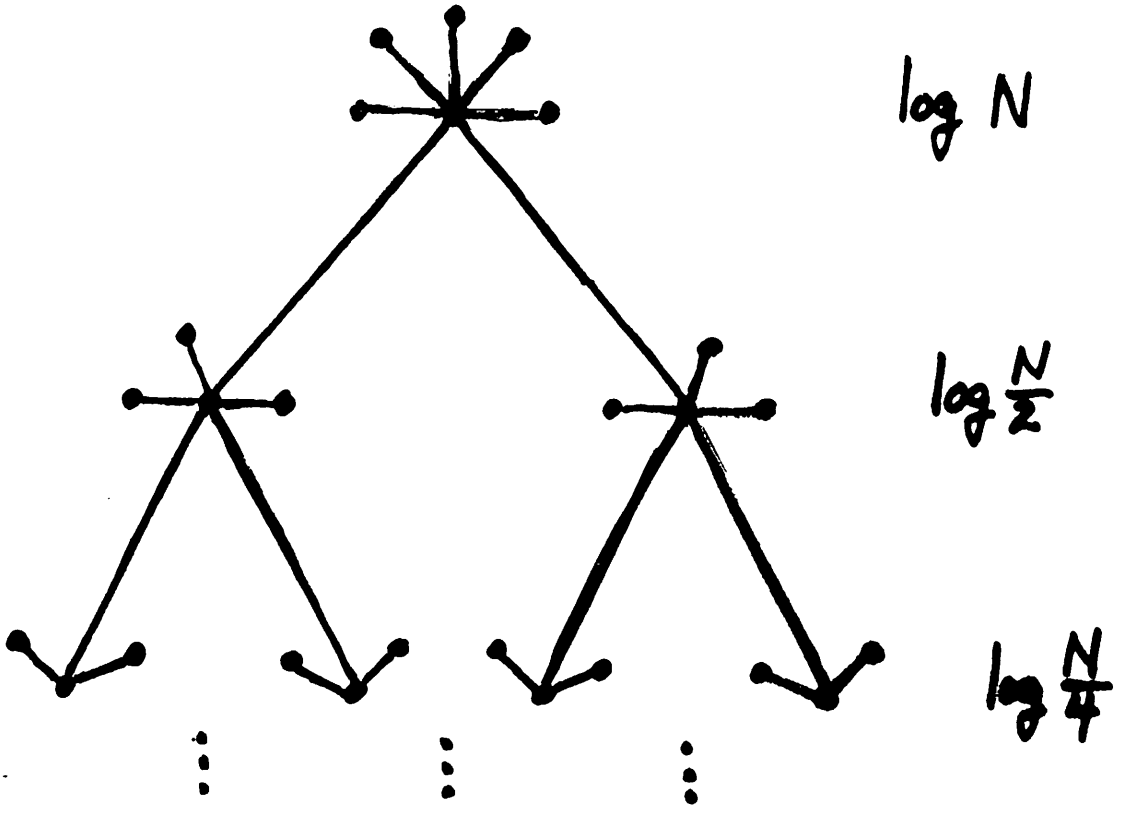

Figure 7. A thistle-tree.

Figure 8. Illustrating the folding of a rectangular grid and its embedding in the hypercube.