1981

Programming Processor Interconnection Structures

Lawrence Snyder

Report Number:
81-381

PROGRAMMING PROCESSOR INTERCONNECTION STRUCTURES*

Lawrence Snyder

Department of Computer Sciences
Purdue University
West Lafayette, IN
47907

CSD-TR-381
October, 1981

ABSTRACT

Parallel computer architecture complicates the already difficult task of parallel programming in many ways, e.g., by a rigid interconnection structure, addressing complexity, and shape and size mismatches. The CHiP computer is a new architecture that reduces these complications by permitting the processor interconnection structure to be programmed. This new kind of programmming is explained. Algorithms are presented for several interconnection patterns including the torus and the complete binary tree and general embedding strategies are identified.

*The research described herein is part of the Blue CHiP Project. Funding is provided in part by the Office of Naval Research under Contract No. N00014-80-K-0816 and Contract No. N00014-81-K-0360, Special Research Opportunities Program, Task SRO-100.

PROGRAMMING PROCESSOR INTERCONNECTION STRUCTURES*

Lawrence Snyder

Department of Computer Sciences

Purdue University

West Lafayette, IN

47907

Introduction

Although it is a difficult task to design a sequential computer architecture that efficiently hosts sequential algorithms, it is perhaps even more challenging to design a parallel architecture that efficiently hosts parallel algorithms. The aspects of parallel computation that frustrate the harmonious match between algorithm and architecture are many:

Rigid interconnection structure: Parallel architectures tend to provide a fixed interconnection structure between processing elements (PE's). For example, ILLIAC IV is mesh connected; the Massively Parallel Processor [1] has a toroidal structure. But recently developed parallel algorithms use a variety of PE interconnection structures. For example, there are tree algorithms for everything from sorting to graph coloring [2] as well as applicative language expression evaluation [3], hexagonally connected pipelined algo-

*The research described herein is part of the Blue CHiP Project. Funding is provided in part by the Office of Naval Research under Contract No. N00014-80-K-0816 and Contract No. N00014-81-K-0360, Special Research Opportunities Program, Task SRO-100.

rithms for numeric problems [4], "double trees" for searching and data base operations [5], and many nonstandard interconnection graphs. (See Figure 1.) The problem is that the rigid interconnection structure biases the architecture towards a particular class of algorithms and makes it difficult to use for any other class of algorithms.

Problem shape and size mismatch: Parallel algorithms tend to require a particular number of PE's in a particular shape that is determined by the problem's input, but the architecture provides only one fixed size and shape. For example, an algorithm requiring an n/2 × 2n array of PE's does not "fit" on an $n \times n$ mesh connected architecture even though there are enough processors.

Addressing complexity: Certain parallel architectures, e.g., the Ultra Computer [6] and the Cube connected cycles [7], provide a "universal" interconnection structure in which a logical interconnection structure is implemented on the physical structure by means of packet routing operations. Time is wasted in unproductive packet switching. More seriously, the programs stored in the PE's are complicated by the need to compute target addresses.

Paucity of programming languages: Although languages such as APL and Concurrent Pascal have "parallel semantics," most parallel algorithms are specified in an *ad hoc* manner. Thus there is little guidance from the programming language as to what features to optimize for.

These and other complications explain in large measure why highly parallel computers have been difficult to program.

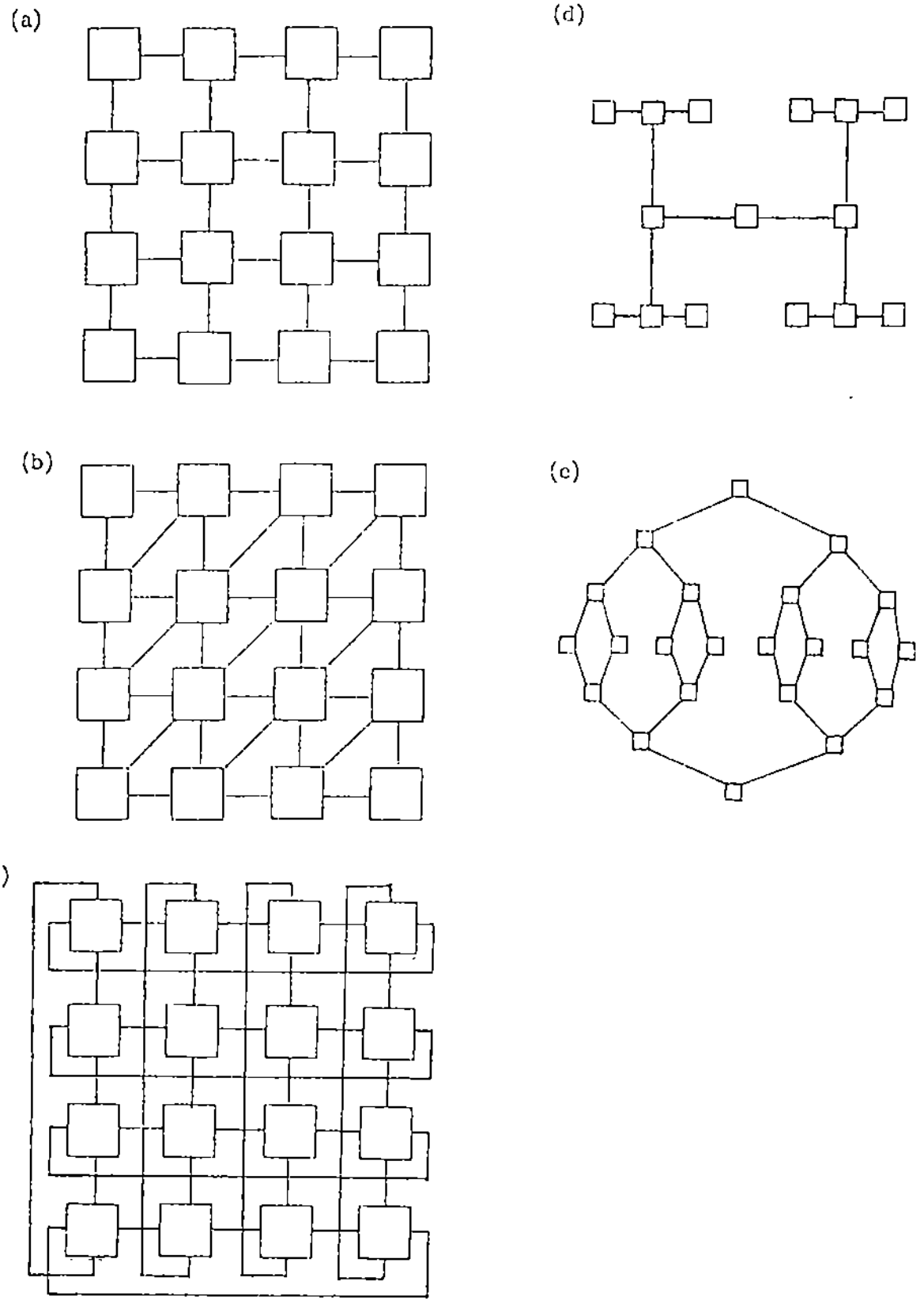


Figure 1. Interconnection patterns for parallel algorithms (a) mesh, (b) hexagonally connected mesh, (c) torus, (d) binary tree, (e) double tree.

We report on a new family of architectures, the Configurable, Highly Parallel (CHiP) computers, that respond to the demands of parallel algorithms, especially the need for locality and flexibility. The central concept is this:

> The processing elements are embedded into a programmable switch lattice that permits not only the programming of the PE's but also the direct programming of their interconnection structure.

This second kind of programming not only ameliorates the difficulties mentioned above, it also permits the convenient *composition* of parallel algorithms. It has even led to the development of entirely new parallel algorithms [8]. In this paper we give a synopsis of the CHiP architecture and then explore the consequences of this new kind of programming, interconnection structure programming. The main results are algorithms of programming various interconnection structures.

Synopsis of the CHiP Computer

[Readers familiar with the CHiP Computer may wish to omit this section.]

A CHiP Computer is composed of a set of homogeneous microprocessor elements connected at regular intervals to the switches of the switch lattice. The lattice is composed of programmable switches connected by data paths to each other or to the PE's. Perimeter switches are attached to external storage devices. Figure 2 illustrates two examples of this structure.* Each PE has its own local program and data memory and

*Notice that the pictures are not drawn to scale. The PE's are much larger than the switches.

each switch contains enough local memory to store several configuration settings.

(a) (b)

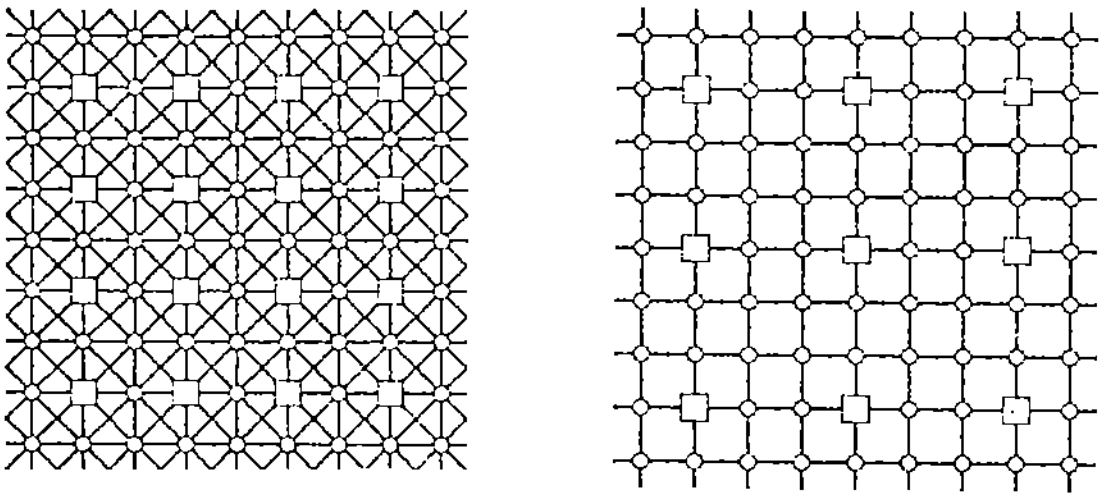


Figure 2. Two lattices. Circles represent switches; squares represent processing elements; lines represent data paths.

A configuration setting is an instruction which, when invoked, causes the switch to form a passive connection between any combination of its incident data paths. Notice that this is circuit switching rather than packet switching and that fan out is possible at the switches. Figure 3(a) shows the configuration settings for a mesh pattern for the lattice of Figure 2(a); Figure 3(b) shows the same lattice configured as a binary tree. To implement an interconnection pattern, the switches are loaded with configuration settings by an external control processor via a "skeleton" that is transparent to this discussion. This activity is usually performed in parallel with the controller's loading of the PE programs.

(a) (b)

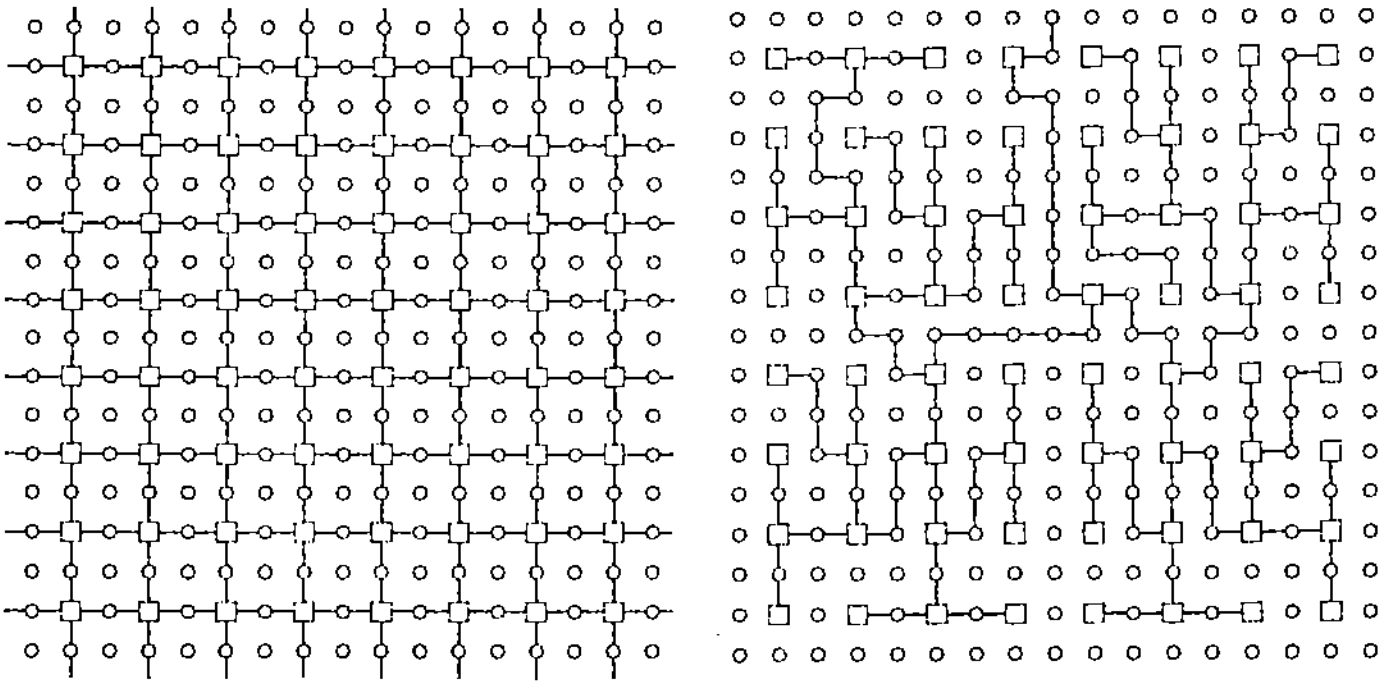


Figure 3. Two configurations of the lattice in Figure 2(a).

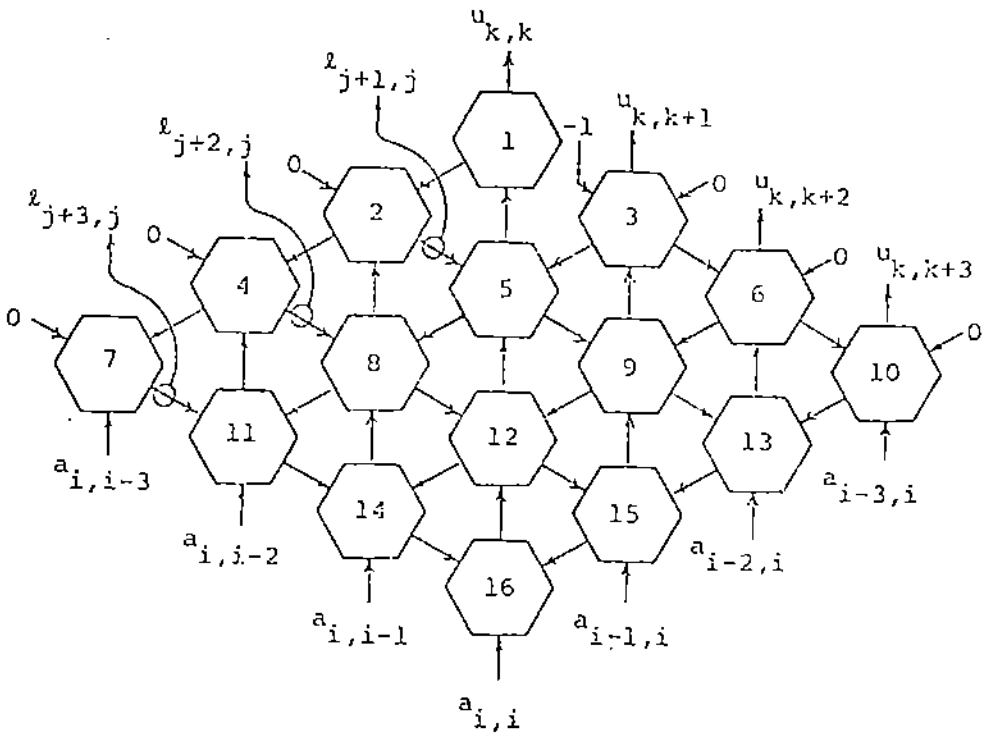
A parallel program is viewed as the composition of several parallel algorithms each with its own processor interconnection pattern. Each of these interconnection patterns and the associated PE code is called a "phase." The controller loads the PE's and switches with the instructions for several phases. Processing begins with a broadcast command from the controller to the switches to invoke a particular stored interconnection pattern. This also causes the PE's to begin synchronously executing their local programs. The interconnection structure remains static throughout the execution of the phase. When the phase completes, another broadcast command causes a different interconnection pattern

to be invoked and a new phase to be initiated. The action continues in this manner from phase to phase.

Several points are worthy of special emphasis. First, to implement an interconnection pattern requires that all configuration settings be stored in the same location in all of the switches. This is so that the broadcast command can take the simple form "invoke the setting in location x," thus making possible one step phase transitions. Second, switches can provide the ability for data paths to "crossover" one another, i.e., a setting can implement multiple data path interconnections. Third, the PE's need not know to whom they are connected; they simply execute instructions of the form READ EAST, WRITE NORTHWEST, etc. The interconnection pattern explicitly implements the routing. Fourth, the data paths are bidirectional.

Example: Consider the problem of finding the solution to a system of linear equations, $Ax=b$, where A is an $n \times n$ band matrix of width p and b is an n vector. To solve the problem we use the Kung-Leiserson LU decomposition pipelined (systolic) algorithm [4] and their lower triangular system (LTS) solver algorithm. The interconnection pattern (for $p=4$) is shown in Figure 4. The exact operation of the algorithms is unimportant except to say that they are pipelined and the data moves in the direction of the arrows. Phase 1 decomposes A into lower and upper triangular matrices, $A=LU$, and at the same time solves the lower triangular system, $Ly=b$. Figure 5 shows the embedding into the lattice of Figure 2(a) of these two algorithms -- the L matrix is transferred directly from the decomposition processor to the LTS solver. The x vector result can be formed by solving $Ux=y$, which is done by rewriting U as a lower triangular matrix and using

(a)



(b)

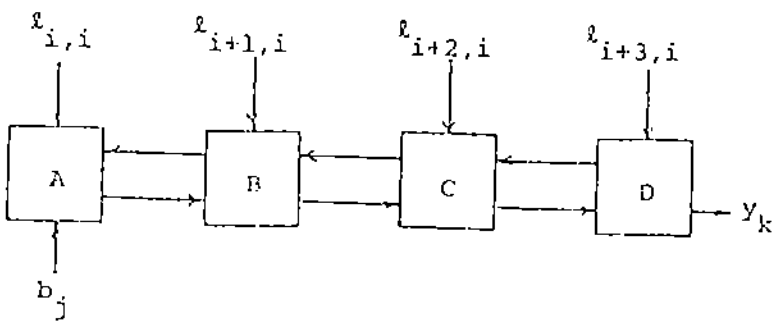


FIGURE 4. Kung-Leiserson Systolic arrays [4]. (a) LU-Decomposition; (b) Lower triangular systems solver.

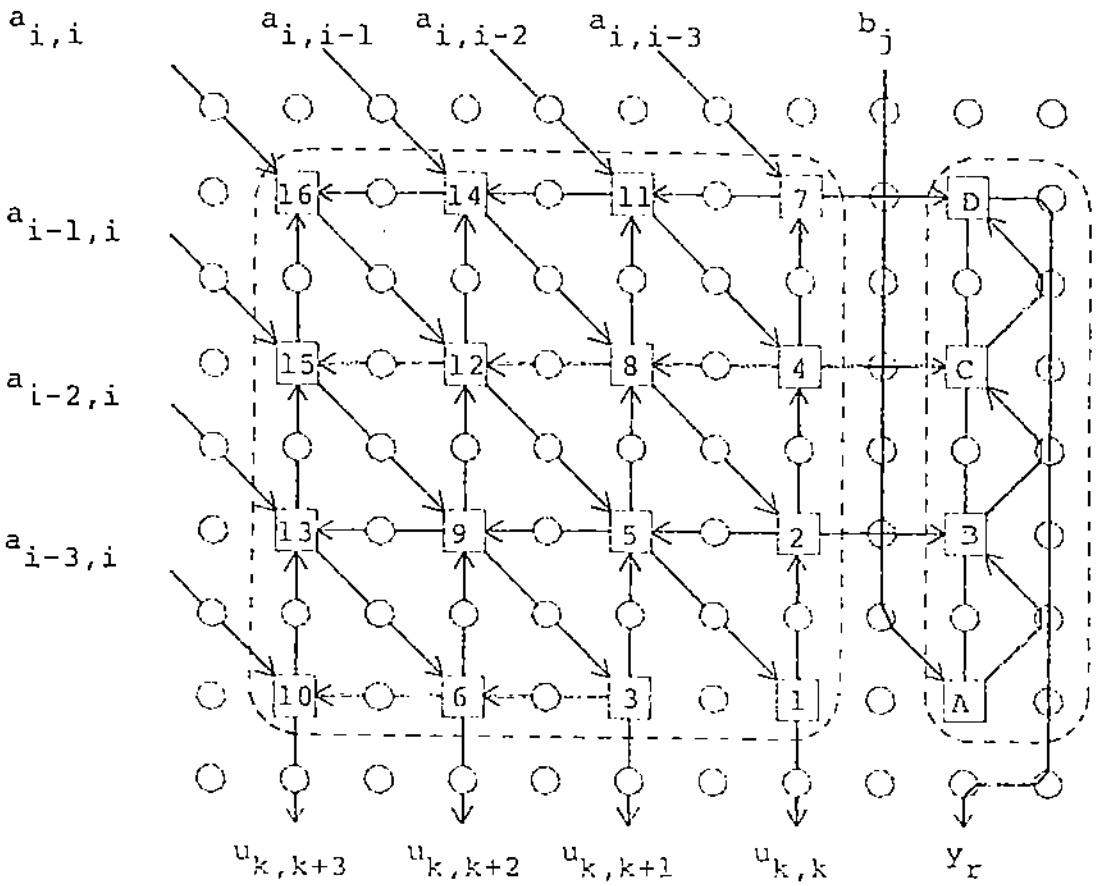


FIGURE 5. The embedding of the LU-Decomposition processors (1-16) and the lower triangular system solver (A-D) of Figure 4 in the lattice of Figure 2(a). The embedding appears in the North-West corner of the lattice.

the LTS algorithm, but U must be completely generated before being rewritten. Thus, phase 1 saves the U matrix and y vector values in preparation for phase 2 by threading them through the lattice. (See Figure 6.) In phase 2 the values are threaded back through the lattice in the opposite direction, which effects the rewriting operation, and they are input to another LTS solver. (See Figure 7.) The result exits from the array at the left end of the LTS solver.

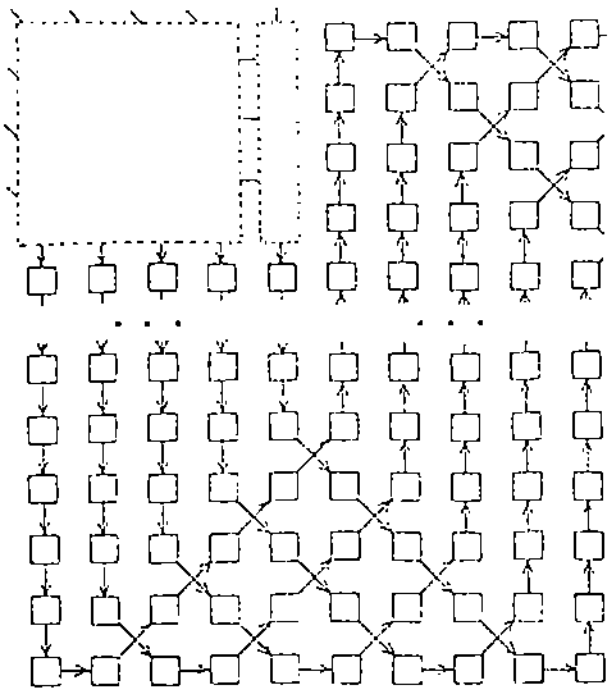


Figure 6. Threading the U matrix and y vector values of Phase 1. Switches not shown.

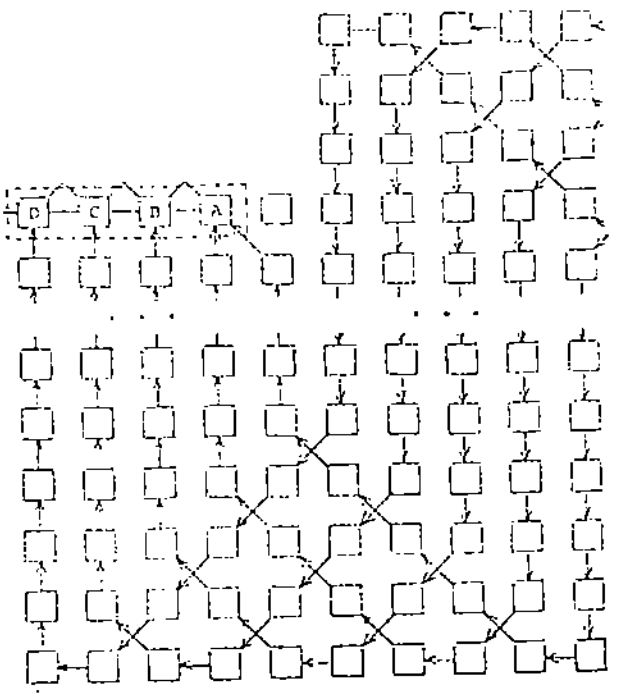


Figure 7. Reverse threading of the U and y values for Phase 2 together with a second LTS solver.

The example is specialized to a band matrix of width $p=4$. A general procedure that solves this problem for arbitrary width bands would differ only in the interconnection structure; the various PE programs required for an arbitrary width solution are all represented in this $p=4$ case. Thus, it is the programming of interconnection patterns that is of central importance.

Programming Interconnection Patterns

We will emphasize the specification of uniform rather than *ad hoc* interconnection patterns because they are of interest in their own right and they are often the building blocks that are used by the less regular patterns. First, we must consider the lattice that is to host the interconnection pattern.

As indicated in Figure 2, a variety of different lattices are possible, although any particular architecture will use only one. Lattices differ in complexity in several ways: corridor width, degree, and crossover capability. The corridor width, w, is the number of switches separating two adjacent PE's, e.g., the lattice of Figure 2(a) has $w=1$ and that of Figure 2(b) has $w=2$. Any lattice can embed an arbitrary graph, but to do so may require leaving some PE's unused [9]. A wider corridor width uses PE's more efficiently when embedding complex graphs. The degree, d, of a lattice is the number of data paths incident on a PE or a switch. (If these two numbers are different, d is the minimum.) For example, Figure 2(a) has $d=8$ while Figure 2(b) has $d=4$. Finally, the amount of crossover capability c is the number of distinct data paths that can intersect at one switch. A crossover capability $c=2$ permits a crossover while $c=1$ does not. In the interest of generality, we will assume the "simplest" lattice suitable for an interconnection pattern.

Programming an interconnection pattern requires that the configuration setting of each relevant switch be defined. For the present discussion it suffices that we give a logical specification of the setting since the actual bit configurations are irrelevant. Accordingly, we will code the compass points with single letters:

N(orth)	M(aine) i.e., Northeast
S(outh)	F(lorida) i.e., Southeast
E(ast)	A(rizona) i.e., Southwest
W(est)	O(regon) i.e., Northwest

and we will assign settings as pairs of these letters. For example, EW is a horizontal connection while ME is a 45 ° angle. The lattice will always be $n \times n$ where n is the number of processors on a side. We name the switches and PE's with a two value index corresponding to its matrix position. See Figure 8. We will name the lattice "L".

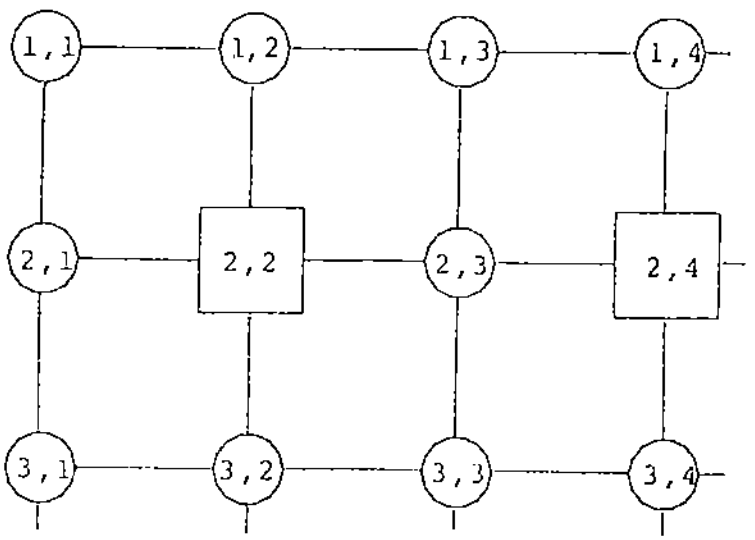


Figure 8. The two index coding scheme for a lattice.

As an example of this specification method, we observe that the mesh interconnection pattern (Figure 3(a)) can be defined* by the two conditions:

(i) i is odd and j is even implies $L[i,j] = NS$

(ii) i is even and j is odd implies $L[i,j] = EW$

*In our presentation of interconnection patterns, we will use a simple declarative specification. We are presently developing a configuration programming language, but until it is completed, we prefer the neutral declarative approach.

provided that the lattice is initially unconfigured. A hexagonally connected interconnection pattern requires the further condition

(iii) i is odd and j is odd implies $L[i,j] = OF$

and requires a lattice of degree $d=6$ or (for symmetry) $d=8$. Notice that this specification is somewhat more general than that used in Figure 5.

Torus Interconnection Patterns

Since the $n \times n$ torus interconnection pattern is simply an $n \times n$ mesh with the top row and bottom row PE's connected and the left column and right column PE's connected, (see Figure 1), one might expect a one corridor, degree 4, crossover capable ($c=2$) lattice to suffice to host this pattern. Surprisingly, it does not.

Theorem. Let L be a $w=1$, $d=4$, $c=2$ $n \times n$ lattice. L cannot be set to connect the PE's into an $n \times n$ torus.

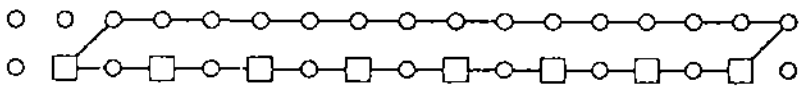
The proof involves arguing that the perimeter corridors must be used for two purposes - to support both the vertical and horizontal "wrap around" and thus cannot lead to an edge disjoint graph embedding.

Direct Torus Representation. Even when $d=8$, embedding the torus is not trivial if we are to avoid multiple use of data paths.

Lattice. $w=1$, $d=8$, $c=2$.

Settings for Crossover Level 1.

First we connect the PE's in the rows. Then we run a data path from the Northeast part of the first PE through the corridor above the row and finally down into the Northeast part of the last PE in the row. For example,



shows the construction for conditions (i) through (iii).

(i) [PE row connections] $1<i,j<2n+1$ and i is even and j is odd imply $L[i,j]=EW$.

(ii) [Northeast ports] $i<2n+1$ and i is odd imply $L[i,3]=AE$ and $L[i,2n+1]=AW$.

(iii) [Corridor above rows] $i<2n+1$ and i is odd and $3<j<2n+1$ imply $L[i,j]=EW$.

Settings for Crossover Level 2. A similar strategy is used for the columns.

(iv) [PE column connections] $1<i,j<2n+1$ and i is odd and j is even imply $L[i,j]=NS$.

(v) [Southwest ports] $j<2n+1$ and j is odd imply $L[3,j]=MS$ and $L[2n+1,j]=NM$.

(vi) [Corridor left of columns] $j<2n+1$ and j is odd and $3<i<2n+1$ imply $L[i,j]=NS$.

Figure 9 illustrates the entire construction.

The difficulty with this interconnection pattern, of course, is that it has long data paths that are subject to propagation delay. Some algorithms can accept such a delay, but generally we would like to reduce it. Accordingly, we prefer the following more intricate pattern that interleaves the row and column processing elements so that there is a fixed bound on the distance a signal must travel.

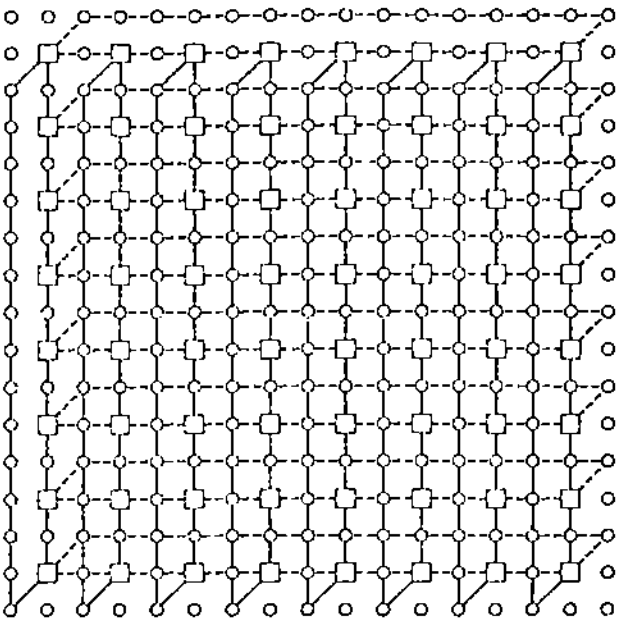


Figure 9. Direct embedding of the torus into the lattice of Figure 2(a). Edges of like color intersecting at a switch are connected.

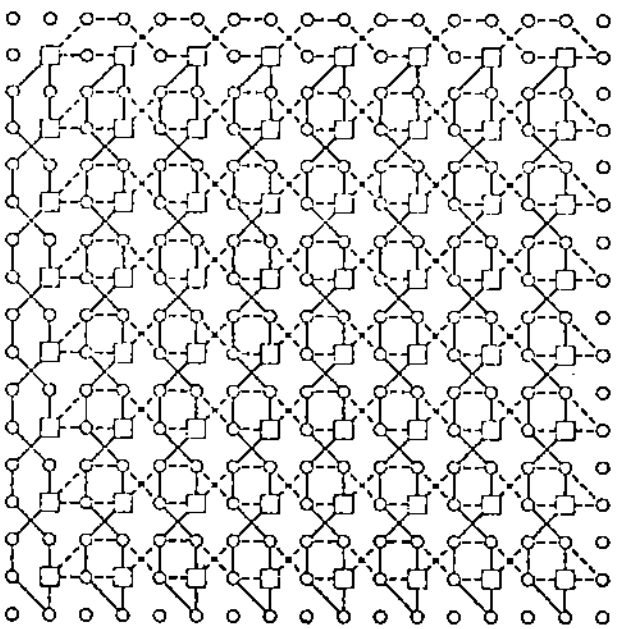


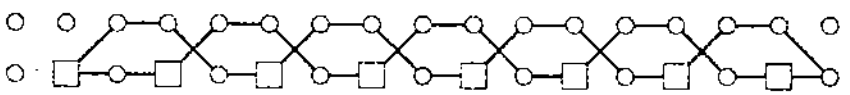
Figure 10. Interleaved embedding of the torus into the lattice of Figure 2(a).

Interleaved Torus Representation

Lattice. $w=1$, $d=0$, $c=2$.

Settings for Crossover Level 1.

First we connect alternate PE's in rows. For example,



The end connections are specified by

(i) [East port, end PE's] i is even implies $L[i,3]=EW$ and $L[i,2n+1]=WO$.

The westerly port connections of each PE are given by

(ii) [West port] i is even and $3<j<2n+1$ and j is odd imply $L[i,j]=OE$.

The connections in the corridor above the row are given by

(iii) [Northeast port] $i<2n+1$ and i is odd and $3\leq j<2n+1$ and j is odd imply $L[i,j]=AE$.

(iv) $i<2n+1$ and i is odd and $3<j$ and j is even imply $L[i,j]=WF$.

Settings for Crossover Level 2. The columns are connected in a manner analogous to the rows.

(i) [South port, end PE's] j is even implies $L[3,j]=NS$ and $L[2n+1,j]=ON$.

(ii) [Northport] j is even and $3<i<2n+1$ and i is odd imply $L[i,j]=OS$.

(iii) [Southwest port] $j<2n+1$ and j is odd and $3\leq i<2n+1$ and i is odd imply $L[i,j]=SM$.

(iv) $j<2n+1$ and j is odd and $3<i$ and i is even imply $L[i,j]=NF$.

The entire construction is shown in Figure 10.

Clearly the maximum number of switches that any data item must pass through is three. *We have increased the locality of the torus embedding*. It is, therefore, more amenable VLSI implementation and can be used in an arbitrarily large lattice with only a constant delay.

Complete Binary Trees

Although an efficient embedding of complete binary trees into the plane is known [10], its direct application to interconnection pattern programming is very wasteful. (See Figure 11.) In fact, since a complete binary tree of depth m has 2^m-1 nodes, we can expect a lattice with $2^k\times 2^k$ PE's to host a complete binary tree of depth $2k$ with one unused node. Call this node a "spare." We can expect that the simplest lattice hosting this pattern will not require crossover capability, since trees are planar, and will require only degree $d=4$, since trees have at most degree 3 connections. (The lattice then is given by $w=1$, $d=4$, $c=1$.) But if the reader attempts to develop an interconnection with these conditions, he will find it to be unexpectedly difficult.

The overall strategy is to begin with small, complete binary trees embedded in square regions of the lattice. To reduce propagation delay the root will be placed in the center of the block. Each block will contain a spare PE. We compose four such square blocks together to form a larger binary tree in a larger square block. Three of the four spare PE's will be used as nodes in the composed tree; the fourth spare will become

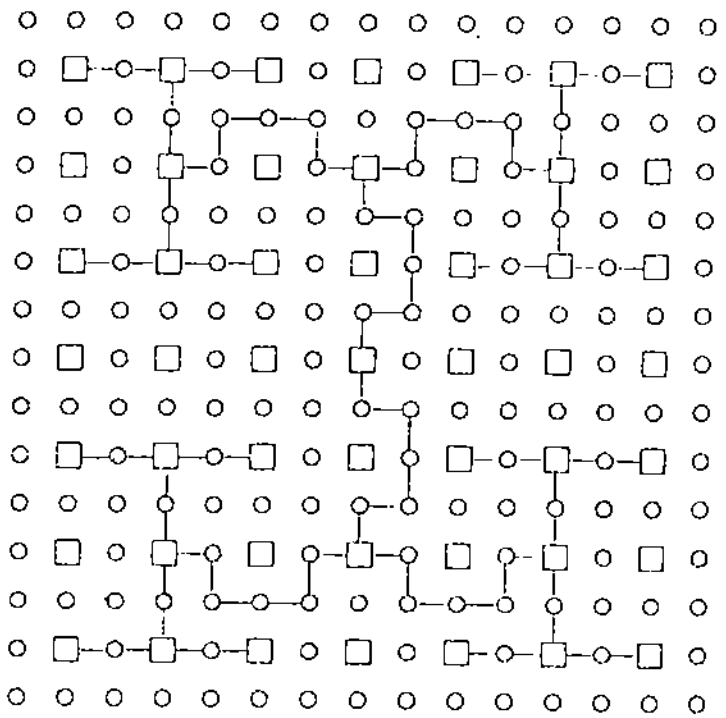


Figure 11. Hyper-H tree (Figure 1(d)) embedding [10]. Filled PE's are unused.

the spare of the new block. The goal is to place the spares so that they will be conveniently located for the compositon.

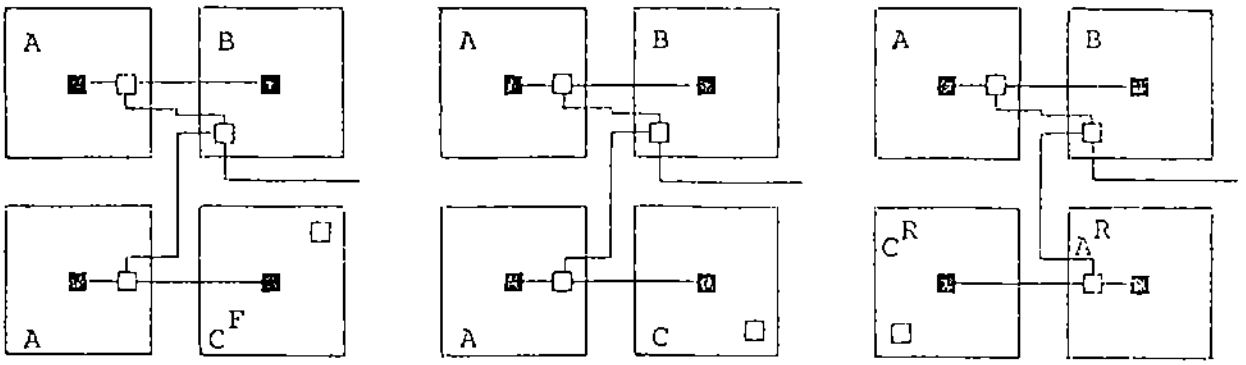
Define three types of tree embeddings:

Type A blocks have their spare PE midway along one side adjacent to the exiting edge from the block's root.

Type B blocks have their spare PE in the corner on the same side as the exiting edge from the block's root.

Type C blocks have their spare PE in the corner on the opposite side of the exiting edge from the root.

Figure 12 illustrates the three types of blocks and demonstrates that they can be inductively produced using blocks of these types.



Type A Type B^R Type C^R

Figure 12. Schematic of blocks composed to form larger blocks. Solid squares represent original roots; open squares represent spares. Superscript "F" means reflect with respect to horizontal axis (flip); superscript "R" means reflect with respect to vertical axis (reverse).

Notice, that as part of the inductive hypothesis, we must argue that the perimeter switches are available for routing the new edges. This is obviously true if they are available in the basis blocks. The smallest blocks that we have been able to find with this property are 4×4 blocks embedding 15 node binary trees. These are illustrated in Figure 13.

The conceptual algorithm is clear. Refer to Figure 14. Begin with an objective block type, e.g., Type B, and a lattice of size $2^k \times 2^k$ PE's. Recursively embed the four subtrees in lattices of size $2^{k-1} \times 2^{k-1}$ such that the proper block types are selected. In the basis cases ($2^2 \times 2^2$), use an explicit embedding. Notice that the results may require reflection. Connect the three spares by appropriate switch settings. This latter operation is always possible based on an inductive argument that depends upon two facts:

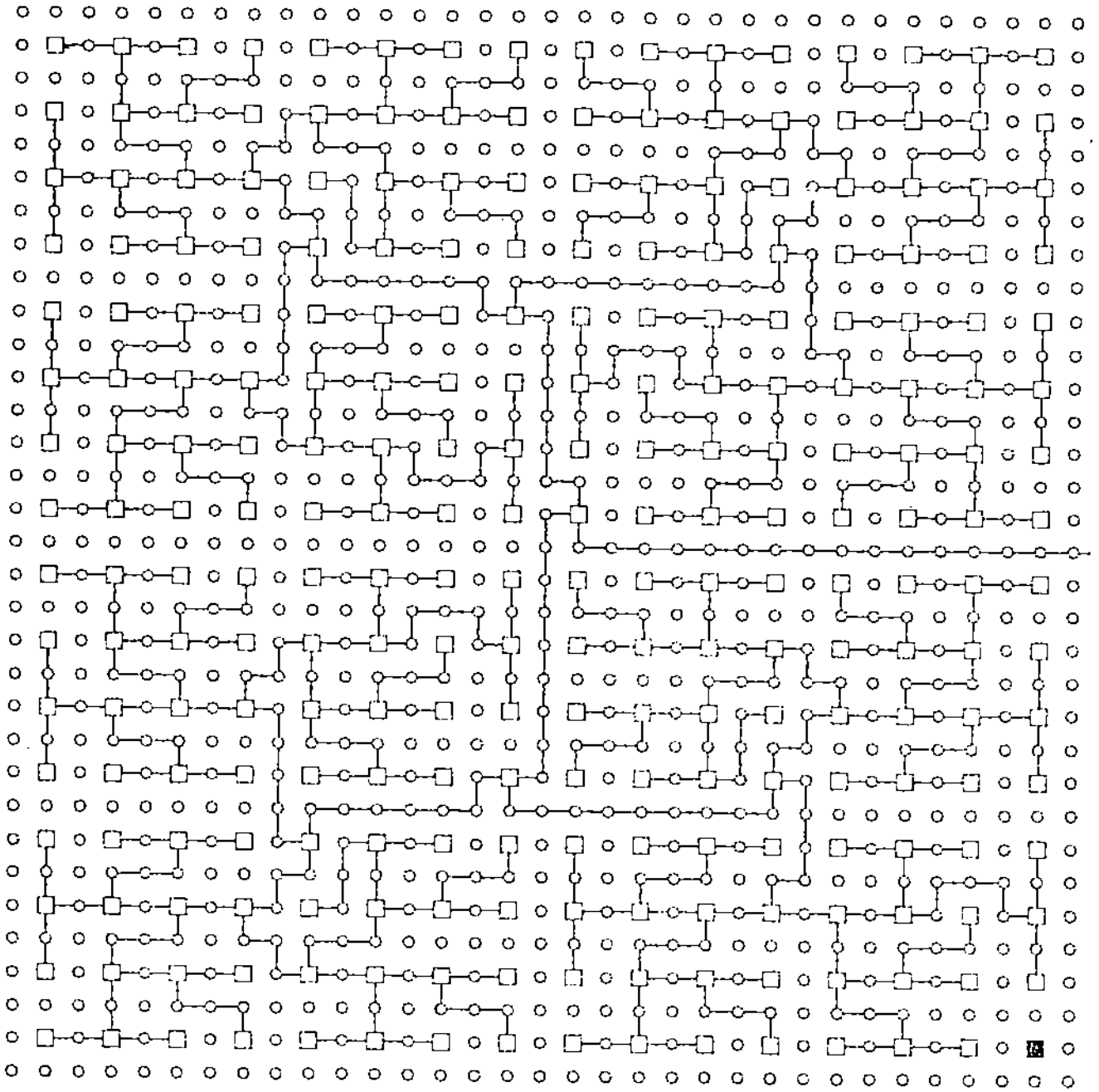


Figure 14. Embedding of a Type B, 255 node binary tree.

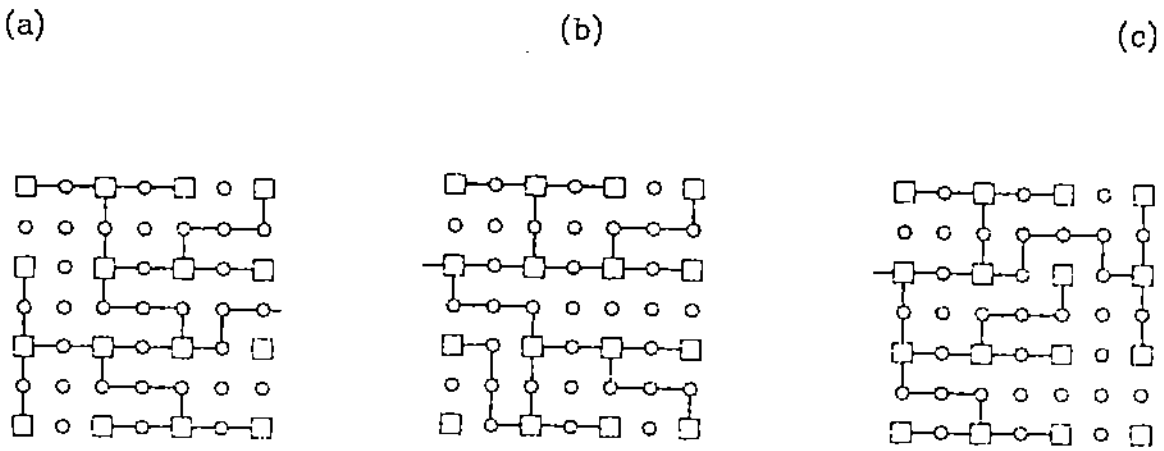


Figure 13. Basis blocks for planar binary tree embedding.

(a) After the basis connection, all spares have their origin as Type C basis block elements, and

(b) None of the switches surrounding a Type C basis block spare is used and so there are three directions of access.

This guarantees that the three data paths can always be assigned. The detailed program is omitted.

Clearly, we have achieved our goal of complete PE usage of this simple lattice. If the available lattice were more complex, e.g., had degree 8 or multiple corridors, then the same embedding would work and some minor optimizations would be possible.

Lacing a Corridor

Although we could present many more of our embeddings - a broadcast tree, a double tree, leaves on a line tree, shuffle exchange, etc. - it is perhaps more instructive to illustrate a technique that gives unexpected power for programming complex graphs. It is called "lacing a corridor"

and it takes optimum advantage of a fixed architectural resource, the corridor width.

Suppose one is embedding an interconnection pattern and must move a large number of distinct data paths across a region of the lattice. By definition, the corridor width, w, is the number of switches separating adjacent PE's. Thus, if the degree $d=4$, then w distinct data paths can be routed between a pair of PE's. It would *appear* that for the degree $d=8$ lattice, w distinct data paths are still the maximum that can be routed down a corridor. But we can do much better.

The idea behind lacing is to begin with straight data paths down a corridor and then to add zig-zag paths that exploit the higher degree and the crossover capability of the switches. For example, Figure 15

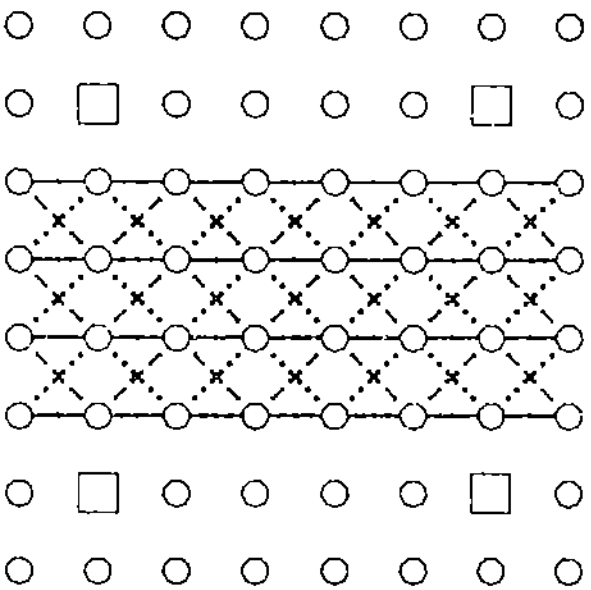


Figure 15. Lacing ten distinct data paths through four switches.

shows a $w=4$, $d=8$, $c=3$ lattice in which ten distinct data paths have been squeezed through the four available switches! This is the maximum possible since the bisection width of this portion of the lattice is ten. (Bisection width is a concept introduced by Thompson [11] referring to the minimum number of wires cut by a line bisecting a VLSI layout.) If we

expand our scope somewhat and include the switches that bound the corridor, then we can increase the number of distinct paths by two. (We will ignore this optimization in the lacing definition below.)

Lattice. $w>1$, $d=8$, $c=3$.

The construction is limited to a region bounded by four PE's. The upper left hand corner PE is $L[r,s]$.

Settings for *crossover level 1.* [Horizontal Path]

(i) $1\le i\le w$ and $0\le j\le w+1$ imply $L[r+i,s+j]=EW$.

Settings for *crossover level 2.* [Dotted Path]

(ii) $1\le i\le w-1$ and $0\le j\le w+1$ and j is even imply $L[r+i,s+j]=AF$.

(iii) $1\le i\le w-1$ and $0\le j\le w+1$ and j is odd imply $L[r+i+1,s+j]=OM$.

Settings for *crossover level 3.* [Dashed Path]

(iv) $1\le i\le w-1$ and $0\le j\le w+1$ and j is even imply $L[r+i+1,s+j]=OM$.

(v) $1\le i\le w-1$ and $0\le j\le w+1$ and j is odd imply $L[r+i,s+j]=AF$.

Notice that if the switches had even higher crossover capability $c=4$, which is the maximum for degree 8 switches, then we could even route vertical wires across the laces if they were needed.

Conclusions

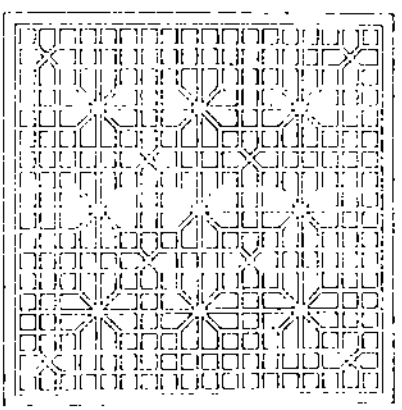
We have introduced the CHiP architecture and argued that its provision for interconnection pattern programming alleviates many of the difficulties encountered in parallel program development. This simplification is achieved in two ways. First, the rigidity of a fixed interconnection structure is no longer an obstacle when one wants to program an algorithm that uses a different interconnection pattern. And

secondly, there is a clean separation between routing the data and programming the activity of the PE's.

Additionally we have demonstrated that interconnection programming is an interesting and challenging activity. We wave shown that locality an be increased by careful study of the torus. We have shown that it is possible to embed the complete binary tree to achieve essentially complete PE utilization. The result involves an interesting assignment of spare PE's. And we have shown that there are general techniques (e.g., corridor lacing) to be found.

Acknowledgments

It is a pleasure to thank Ching C. Hsiao for his original use of lacing and Paul McNabb for developing the software to produce these embeddings and for stimulating discussions of the binary tree embedding. Thanks are due to Paul Morrissett for programming the torus and lacing figures and to Julie Hanover for excellent manuscript preparation.



Compound octagon-square lattice
Chengtu, Szechwan, 1825 A.D.

References

[1] P. A. Gilmore, K. E. Batcher, M. H. Davis, R. W. Lott and J. T. Burkley
Massively Parallel Processor
Technical Report GER-16684, Goodyear Aerospace Corporation, July 1979.

[2] Sally A. Browning
The Tree Machine: A Highly Concurrent Programming Environment
Ph.D. Thesis, California Institute of Technology, January, 1980

[3] Bart Locanthi
The Homogeneous Machine
Ph.D. Thesis, California Institute of Technology, 1980

[4] H. T. Kung and C. E. Leiserson
Systolic Arrays (for VLSI)
Technical Report CS-79-103, Carnegie-Mellon University, December 1979 (also in [10])

[5] Jon L. Bentley and H. T. Kung
A Tree Machine for Searching Problems
In *Proceedings of the 8th International Conference on Parallel Processing*, IEEE, pp. 257-266, 1979

[6] J. T. Schwartz
Ultracomputers
Transactions on Programming Languages and Systems, ACM, 1980

[7] F. P. Preparata and Jean Vuillemin
The Cube connected cycles: A Versatile Network for Parallel Computation
In *Proceedings of the 20th Annual Symposium on the Foundations of Computer Science*, IEEE October, 1979

[8] D. B. Gannon and Lawrence Snyder
Linear Recurrence Algorithms for VLSI: The Configurable, Highly Parallel Approach
In *Proceedings of the 10th International Conference on Parallel Processing*, IEEE, 1981

[9] L. Snyder
Overview of the CHiP Computer
In *VLSI 81*, Academic Press, 1981

[10] Carver Mead and Lynn Conway
Introduction to VLSI Systems
Addison Wesley, 1980

[11] C. D. Thompson
A Complexity Theory for VLSI
Ph.D. Thesis Carnegie-Mellon University, 1980