JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 4, 439–458 (1987)

On Mapping Parallel Algorithms into Parallel Architectures*

FRANCINE BERMAN

University of California at San Diego, La Jolla, California 92093

AND

LAWRENCE SNYDER

University of Washington, Seattle, Washington 98195

Received May 13, 1985

The *mapping problem* arises when the communication structure of a parallel algorithm differs from the interconnection architecture of the intended parallel machine (*topological variation*). This problem is compounded when the number of processes required by the algorithm exceeds the number of processors available in the architecture (*cardinality variation*). In this paper, we present a solution to the mapping problem when there are topological and cardinality variations for a commonly used class of parallel interconnection structures. This class includes, for example, shuffle-exchange networks, hypercubes, square meshes, linear systolic arrays, cube-connected cycles, and complete binary trees. We illustrate this solution by presenting and evaluating mappings from two-dimensional and linear systolic arrays into tree architectures, and from tree algorithms into shuffle-exchange networks.

1. INTRODUCTION

A fundamental task in the implementation of a parallel algorithm on a parallel computer is the allocation of processes and communication paths in the algorithm to processors and data channels in the given machine. Prob-

* The work described herein is part of the Blue CHiP Project. Funding was provided in part by the Purdue Research Foundation and Office of Naval Research, Contracts N00014-80-K-0816 and N00014-81-K-0360, Special Research Opportunities Task SRO-100. This material was presented in part at the 1984 International Conference on Parallel Processing, Bellaire, MI, August, 1984.

0743-7315/87 $3.00

lems may arise when the number of processes required by the parallel algorithm exceeds the number of processors available in the parallel device (*cardinality variation*) or when the communication structure of the parallel algorithm differs from the interconnection architecture of the parallel machine (*topological variation*). Both types of variation occur frequently in nonglobal memory parallel computing: Algorithms use a variety of communication structures but the wires of the device are soldered into fixed positions; additionally, the size of interesting problems frequently exceeds the number of available processors. Although there are solutions to both problems—mapping edges to paths solves topological variation, and mapping several processes to one processor solves cardinality variation—the complexity analysis of any solution must charge for the extra costs of multiplexing processes and for delays over extended data paths.

There is a growing body of work on the mapping problem for specialized environments, i.e., work which focuses on the creation of optimized mappings of specific algorithms on specific parallel machines. For broad- or general-purpose parallel machines, i.e., machines which execute a variety of different algorithms, previous work on the mapping problem has focused on either topological variation (layout problems) or cardinality variation (multiplexing problems). Successful approaches for resolving topological variation have included the creation of approximation algorithms for embedding (algorithm) communication structures into (device) interconnection architectures [B2, IB] and the development of classes of efficient networks into which interconnection structures may be embedded [D1, D2]. Such approaches generally assume that there is no cardinality variation, i.e., that processes in the algorithm may be mapped one per processor in the device. Quotient structures have been used to resolve cardinality problems [FF]; however, given an arbitrary family of interconnection structures, the user must provide the embeddings that contract large structures to smaller ones.

In this paper, we present a solution for solving both topological and cardinality variation problems for a class of parallel interconnection structures. More importantly, for many commonly used interconnection structures, this solution is achievable by an automatic procedure. We describe this procedure in Section 2 and illustrate its use and analyze its efficiency in Section 3.

2. A General Solution

To model the mapping of a parallel algorithm into a parallel architecture, we begin by providing a suitable abstraction for both the algorithm and the target machine. In the sequential model, algorithms are most often described in terms of high-level programming languages that include the usual looping,

branching, and procedural constructs [AHU]. Although such languages differ in detail, a high-level description provides a well-defined model for the behavior of the algorithm. Unlike the sequential environment, there is no universally accepted set of parallel constructs and programming paradigms. Hence we limit our perspective if we describe parallel algorithms within a given parallel programming language. As is frequently done in the literature, we model a parallel algorithm as a static communication graph in which the nodes represent processes, and the edges represent communication paths. The behavior of the parallel algorithm can then be represented as the flow of data on the communication graph. Since a parallel algorithm is really a collection of algorithms, one for each problem instance, we represent a parallel algorithm as a *family* of communication graphs $\{G_n\}$ where each graph G_n represents the communication graph of the algorithm for a given problem size. (We assume that all communication paths have the same length and bandwidth.)

The graph model is also appropriate for representing a target parallel computer with no shared memory. The computing power of such machines is a synthesis of the interaction of interprocessor communication and intraprocessor computation. In particular, the role of communication between processors is fundamental since each processor may have only limited computing power. The graph model provides an explicit representation of the processors and the communication structure of the parallel machine. We represent the interconnection architecture of the target parallel machine by an undirected interconnection graph H.

Using these abstractions, the family $\{G_n\}$ to represent a parallel algorithm and the graph H to represent the interconnection architecture of a nonshared memory parallel computer, we propose the following solution to the mapping problem. This solution can be automated for many parallel algorithms. The strategy is to separate cardinality variation problems from topological variation problems as follows.

To implement a parallel algorithm instance G_n on a parallel architecture H,

(1) Embed the communication graph G_n into a smaller communication graph G_k from the same graph family; i.e., *contract* the algorithm as if the architecture were of the same interconnection type. (This eliminates cardinality variation in the absence of topological variation.)

(2) Next, *lay out* G_k on interconnection architecture H; i.e., map G_k into H with an assignment of at most one process per processor. (This eliminates topological variation in the absence of cardinality variation.)

(3) Third, implement the modified algorithm G_n on the image of G_k in H by *multiplexing*.

This strategy provides a solution to the mapping problem. Preliminary research based on a more detailed development has shown this strategy to be largely successful in producing automatic mappings for a useful class of parallel algorithms and architectures whose interconnection structures include hex and square meshes, toruses, shuffle-exchange graphs, cube-connected cycles, butterflies, hypercubes, etc. We present this detailed development and upper bounds for the performance of our solution in this and the next section.

The first stage of the solution is to embed algorithm instance G_n into a smaller member G_k of the same graph family. We are most interested in embeddings that are adjacency-preserving, i.e., that do not expand communication paths, since propagation of data along expanded communication paths also increases time complexity. We call an adjacency-preserving embedding of a large member of a graph family $\{G_n\}$ into a smaller one a *contraction.* Although the programmer will specify the parallel algorithm (provide the graph family), we would like the compiler or mapping preprocessor to perform the contraction automatically, i.e., to create the embeddings. Hence the formalism we choose for defining graph families must promote automatic contraction. We use *edge grammars* [B1, BS] as a means of defining graph families. An operation on edge grammar-defined families called *truncation* will provide a means of automating the contraction.

The second stage of the solution is to embed graph G_k into architecture H. This can be done by using a general layout algorithm or by using a library of optimized layout routines. We discuss the layout stage further in Section 3.

The third stage is to multiplex G_n on the image of G_k in H. The effect of multiplexing is most predominant as a real-time phenomenon rather than in the evaluation of asymptotic upper bounds. An implementation on CHiP architectures of this solution strategy (including multiplexing) is given in [BGKRS].

The Contraction Stage

Edge grammars. Edge grammars define graph families by generating pairs of vertex labels (strings) using conventional formal language mechanisms. Each vertex label represents a node, and a pair of vertex labels represents an edge. We are most interested in edges whose vertex labels are strings of the same length. The collection of edges with same-length vertex labels can naturally be partitioned into a family of graphs $\{G_n\}$ where G_n is the graph formed by all the edges with length n vertex labels generated by the edge grammar. The edge grammar-generated graph family can then be used to represent the communication graph instances of a parallel algorithm. More formally,

DEFINITION. A *Type 2 edge grammar* is a context-free grammar $\Gamma = (N, T, S, P)$ in which N is a finite set of nonterminals, T is a finite set of terminal

pairs (a, b) over a fixed alphabet, G in N is the start symbol, and P is a set of productions. Each production in P has the form $A \rightarrow BC$ or $A \rightarrow B$ where A is in N and B and C are in $N \cup T$. We define the concatenation of two terminal pairs $(a, b)(c, d)$ to be the terminal pair (ac, bd).

A *Type 3* edge grammar is a Type 2 edge grammar all of whose productions have the form $A \rightarrow B(a, b)$, $A \rightarrow B$, or $A \rightarrow (a, b)$ where A and B are in N and (a, b) is in T.

(Following standard formal language conventions, a Type 0 edge grammar would have unconstrained production rules and a Type 1 edge grammar would have production rules of the form $uAv \rightarrow uwv$ where $A \in N$, $u, v, w \in (N \cup T)^*$ and w is not the empty string. For the present application, all useful edge grammars are of Type 2 or Type 3.)

DEFINITION. The graph family generated by Γ, $G(\Gamma)$, is the set $\{G_n\}_{n>0}$ where G is the start symbol of Γ and G_n is the undirected graph (V_n, E_n) in which

$$V_n = \{v | G \rightarrow^*(v, w) \text{ or } G \rightarrow^*(w, v) \text{ for some } w \text{ and } |v| = |w| = n\}$$

$$E_n = \{(v, w) | G \rightarrow^*(v, w), |v| = |w| = n \text{ and } v \neq w\}.$$

Edge grammar derivations produce pairs of strings (v, w) in the same way as conventional formal languages by applying a sequence of production rules to the start symbol. If the strings in the pair are of the same length n, we consider them to be labels of nodes connected by an edge in graph $G_n = (V_n, E_n)$. (If $v = w$, we include the node v in V_n but we do not include the self-loop (v, v) in E_n. In the parallel processing environment, this is representative of the fact that communication from a processor to itself can be implemented internally within a processor and without an explicit communication line.) Note also that each G_n is an undirected graph since communication paths in our algorithms are considered to be bidirectional.

We include an example of an edge grammar derivation of the graph family of complete binary trees.

EXAMPLE. The family of complete binary trees $\{B_n\}$ can be generated by a Type 3 edge grammar. Let Γ be the edge grammar with nonterminals B_0, B_1, B_R, B, and R, terminals $(0, 0)$, $(1, 1)$, $(2, 2)$, $(2, 0)$, and $(2, 1)$, start symbol B, and the following productions:

$$\begin{array}{lll} B \rightarrow R & B \rightarrow B_0 & B \rightarrow B_1 \\ B \rightarrow B_R & B_0 \rightarrow B(0, 0) & B_1 \rightarrow B(1, 1) \\ B_R \rightarrow R(2, 0) & B_R \rightarrow R(2, 1) & B \rightarrow (2, 0) \\ B \rightarrow (2, 1) & R \rightarrow (2, 2) & R \rightarrow R(2, 2) \end{array}$$

Consider a derivation within Γ. At each step, the derivation extends the length of the vertex labels in the pair being generated by applying one of the production rules. For example, the edge (20, 10) can be derived from B as follows:

$$B \rightarrow B_0 \rightarrow B(0, 0) \rightarrow (2, 1)(0, 0) = (20, 10).$$

Since 20 and 10 both have length 2, the edge (20, 10) is in B_2, the complete binary tree with length 2 labels. By similar derivations, it can be shown that the only other nonlooping edges derivable from B whose vertex labels are of length 2 are $\{(20, 00), (21, 01), (21, 11), (22, 20), (22, 21)\}$. (The only looping edge with vertex labels of length 2 derivable in this grammar is (22, 22).) The formation of these edges into a complete binary tree is illustrated in Fig. 1a.

Edge grammars can be used to define many natural graph families (such as the family of shuffle-exchange graphs) which are not definable within existing formalisms such as graph grammars [CER], Lindenmayer systems [S1], and pair grammars [P]. Graph families definable by Type 2 and Type 3 edge grammars include the cube-connected cycles, hex and square meshes, hypercubes, toruses, shuffle-exchange graphs, linear systolic arrays, butterflies, and other interconnection structures commonly used in parallel computation. For a more thorough discussion of edge grammars, see [B1, BS].

Edge grammars not only provide a formalism for defining graph families but also a mechanism for mapping larger interconnection graphs into smaller interconnection graphs in a family. When the embeddings are adjacency-preserving, i.e., do not dilate communication paths, we call them *contractions.*

DEFINITION. Let J and K be undirected graphs. A *contraction* is an embedding $m : J \rightarrow K$, such that $m(V_J) \subseteq V_K$ and $\{(m(v), m(w)) | (v, w) \in E_J\} \subseteq E_K$.

In particular, a contraction is a mapping from graph J to graph K which identifies adjacent nodes in J with the same or adjacent nodes in K. If there

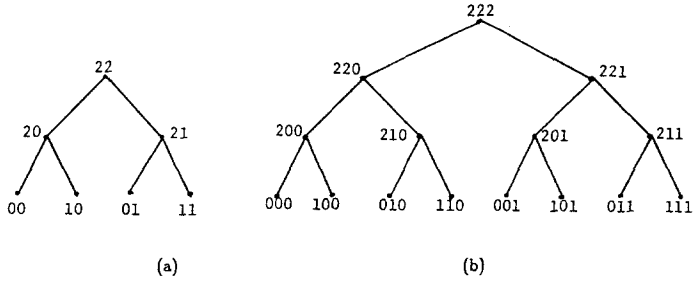

FIG. 1. The graphs B_2 (a) and B_3 (b) generated using the edge grammar Γ.

exist contractions $m_1 : J_1 \rightarrow J_2$, $m_2 : J_2 \rightarrow J_3, \ldots, m_k : J_k \rightarrow K$, we say that there is an *iterated contraction* from J_1 to K. When the context is clear, we denote this iterated contraction as $m : J_1 \rightarrow K$.

DEFINITION. Let $k > 0$ be an integer and let $\{G_n\}$ be a graph family. Then $\{G_n\}$ is *k-contractable* if for each n, there exists an iterated contraction $m : G_{n+k} \rightarrow G_n$.

DEFINITION. Let $k > 0$ and let $\{G_n\}$ be a graph family generated by an edge grammar Γ. Then $\{G_n\}$ is *k-truncatable* if for all n,

$$\{a_1 \cdots a_n | a_1 \cdots a_{n+k} \in V_{n+k}\} \subseteq V_n$$

and

$$\{(a_1 \cdots a_n, b_1 \cdots b_n) | (a_1 \cdots a_{n+k}, b_1 \cdots b_{n+k}) \in E_{n+k}\} \subseteq E_n.$$

K-truncation is an easily automated type of k-contraction. (Another type of contraction can be found in [FF]). As an example, consider the family $\{B_n\}$ of complete binary trees. This family is k-truncatable for $k > 0$. B_3 is the complete binary tree with length 3 labels illustrated in Fig. 1b. After 1-truncation, each of the nodes in B_3 is mapped to a node in B_2 (the complete binary tree with length 2 labels illustrated in Fig. 1a, and each edge is either coalesced or mapped to an edge in B_2). Pictorially, the 1-truncation of B_3 "folds" the left subtree onto the right subtree and coalesces the root and its left and right sons of B_3 into the root of B_2.

We can characterize many of the k-truncatable Type 3 graph families with the following theorem.

THEOREM 1 [B1]. *Let Γ be a Type 3 edge grammar for graph family $\{G_n\}$ and let k be a fixed integer. Assume that $|v| = |w| = k$ for every terminal pair (v, w) that appears in a production in Γ. If for each nonterminal A, there is a derivation $G \rightarrow {}^*A$ where G is the start symbol of Γ, then $\{G_n\}$ is k-truncatable.*

Type 3 graph families which can be shown to be k-truncatable by Theorem 1 include, for example, complete binary trees, cube-connected cycles, hex and square meshes, toruses, linear and multidimensional arrays, butterflies, and complete graphs.

We have briefly described a formalism, edge grammars, for generating graph families and have given a characterization of truncatable Type 3 graph families. Parallel algorithms whose graph families are characterized as such may be automatically contracted using truncation. We now need a way of evaluating the "goodness" of the embeddings produced by truncations and other types of contractions. In particular, we want to measure the distribution of the nodes and communication paths of the uncontracted graph over

the contracted graph. Contractions which distribute nodes and communication paths of the uncontracted graph roughly equally over the contracted graph minimize the amount of sequential processing to be done by each processor during multiplexing and maximize parallelism.

Weighting functions. Many graph families can be generated by more than one edge grammar and hence may be contracted by more than one mapping. (This is analogous to the formal language context in which a language may be generated by more than one grammar.) These contractions may differ in the allocation of nodes and edges of a larger graph G_{n+k} in a smaller graph G_n. We measure this allocation by defining the following weighting functions. These functions will measure the amount of parallelism preserved by a contraction by gauging the concentration of computing activity to be done sequentially at an individual processor during multiplexing. They also will provide a means of comparing different contractions for the same graph family.

DEFINITION. Let v be a unit node weight and e be a unit edge weight. Let m be an iterated contraction from graph G_{n+k} to graph G_n. For vertex u in V_n and edge s in E_n, let

$$w_m(u) = |\{x|m(x) = u\}|v + |\{(x, y)|m(x) = m(y) = u\}|e$$
$$w_m(s) = |\{(x, y)|(m(x), m(y)) = s\}|e;$$

that is, $w_m(u)$ measures the number of nodes and edges mapped to u by m, and $w_m(s)$ measures the number of edges mapped to s by m.

This function represents the sequential communication cost and the bandwidth requirements of m when the argument is a node or edge, respectively. For example, in Fig. 1, by contracting the complete binary tree B_3 onto the complete binary tree B_2, we get

$$w_m(22) = 2v + 3e$$

since nodes 222, 221, and 220 in B_3 are mapped into node 22 in B_2 as are edges (222, 221) and (222, 220). Also,

$$w_m((22, 21)) = 2e$$

since edges (221, 211) and (220, 210) in B_3 are mapped onto edge (22, 21) in B_2. Note that our model assumes a unit communication cost and a unit computation cost for the original parallel algorithm $\{G_n\}$.

By generalizing the weighting functions, we can obtain *average* and *maximum* edge weights $(\alpha_m(E_n), \omega_m(E_n))$ and *average* and *maximum* node weights $(\alpha_m(V_n), \omega_m(E_n))$ under contraction m as follows:

$$\alpha_m(E_n) = \frac{\sum_{s \in E_n} w_m(s)}{|E_n|}$$

$$\omega_m(E_n) = \max_{s \in E_n}\{w_m(s)\}$$

$$\alpha_m(V_n) = \frac{\sum_{u \in V_n} w_m(u)}{|V_n|}$$

$$\omega_m(V_n) = \max_{u \in V_n}\{w_m(u)\}.$$

Again using the example of the complete binary tree graph family $\{B_n\}$ generated by edge grammar Γ shown in Fig. 1,

$$\alpha_m(E_2) = \omega_m(E_2) = 2e$$

$$\alpha_m(V_2) = \frac{15v + 2e}{7}$$

$$\omega_m(V_2) = 3v + 2e.$$

A different edge grammar which also generates the graph family of complete binary trees is Γ' with nonterminals C, L, N, and I, terminals (2, 2), (0, 0), (2, 1), (1, 1), and (2, 0), start symbol C, and productions

$C \rightarrow (2, 0)$	$L \rightarrow N(2, 1)$	$N \rightarrow N(0, 0)$	$I \rightarrow (2, 1)$	$I \rightarrow L$
$C \rightarrow L$	$C \rightarrow I$	$N \rightarrow (0, 0)$	$N \rightarrow (1, 1)$	$I \rightarrow I(2, 2)$
$C \rightarrow (2, 1)$	$L \rightarrow N(2, 0)$	$N \rightarrow N(1, 1)$	$I \rightarrow (2, 0)$	

The complete binary trees with length 2 labels (C_2) and length 3 labels (C_3) are shown in Fig. 2. By Theorem 1, this grammar is 1-truncatable. When we 1-truncate the trees in Fig. 2, we obtain the weighting functions

$$\alpha_m(E_2) = \omega_m(E_2) = e$$

$$\alpha_m(V_2) = \frac{15v + 8e}{7}$$

$$\omega_m(V_2) = 3v + 2e.$$

These weighting functions can now be used to compare Γ to Γ' for generating this graph family. The maximum number of nodes which must be simulated sequentially during multiplexing at an individual processor is the same

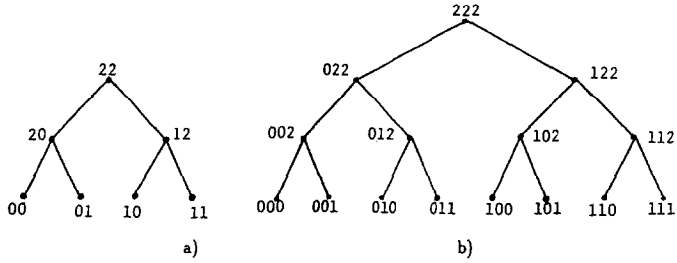

FIG. 2. Complete binary trees C_2 (a) and C_3 (b) for edge grammar Γ'.

for both edge grammars; however, more communications can be performed as intraprocessor computations with the mapping induced by Γ'. For a machine with small bandwidth (say one which could only pass one complete message at a time), Γ' would be a better choice to generate this graph family.

Table I shows the effect of truncating interconnection graph G_{n+1} into G_n for a sampling of interconnection families and edge grammars. These truncations are pictured in Fig. 3.

In mapping parallel algorithm instance G_n to architecture H, contractions and truncations are used to solve cardinality variation problems, i.e., to translate the original algorithm communication graph to a communication graph which is not "too big" for the architecture. To resolve the remaining topological variation problems, we can use known layout results. In the next section, we combine contraction methods and known layouts to achieve upper bounds for mapping truncatable graph families.

3. USING LAYOUT RESULTS TO PROVIDE GOOD BOUNDS

In this section, we give a cost measure for layout results and provide general upper bounds for mappings achieved by our three-stage strategy of con-

TABLE I
WORST CASE WEIGHTS IN SELECTED GRAPHS[a]

Graph family	Edge grammar type	$\omega_m(V_n)$	$\omega_m(E_n)$
(a) Lines	3	$2v + e$	e
(b) Complete binary trees	3	$3v + 2e$	$2e$
(c) Cube-Connected cycles	3	$4v + 3e$	$2e$
(d) Square meshes	3	$4v + 4e$	$2e$
(e) Hypercubes	3	$2v + 2e$	$2e$

[a] All families are Type 3 and k-truncatable.

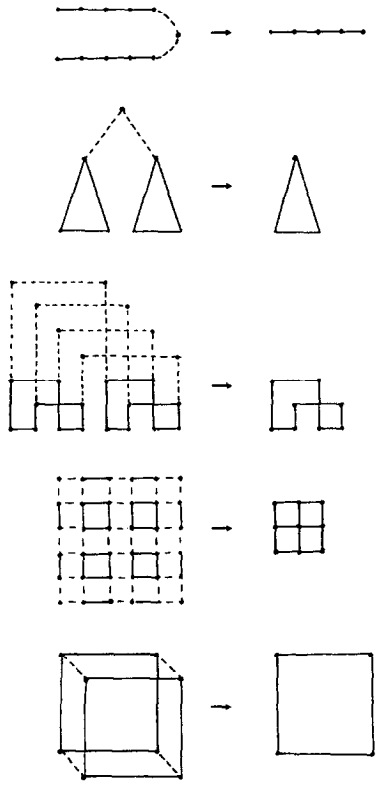

FIG. 3. Truncations of the graph families given in the table. Nodes incident to dashed edges are coalesced to a single point.

traction, layout, and multiplexing. We illustrate these upper bounds by mapping a two-dimensional array and a linear systolic array into a tree machine [B3] and by mapping a tree algorithm into a shuffle-exchange architecture [S4]. All of the graph families representing parallel algorithms in our examples have edge grammars that are truncatable by Theorem 1.

The Layout Stage

In the last section, we developed and evaluated contractions from large-sized graphs to small-sized graphs. The next step in our strategy is to lay out the contracted graph on the interconnection structure representing the target machine. This can be done using known optimal or near-optimal layout results or the result of an approximation algorithm. For the theoretical discussion, we will assume that the layout of contracted algorithm G_k on architecture H is the best known. In implementing our strategy, it is actually more desirable (for generality) to use an approximation algorithm.

Cost functions. Once the layout is defined, we need a means to measure its "goodness." We define a cost function by analogy to the weighting functions for contraction to measure edge expansion for a layout.

DEFINITION. [RS]. Let $G_k = (V_k, E_k)$ and $H = (V_H, E_H)$ be undirected graphs and let f be a mapping of G_k into H. Then for each s in E_k, $f(s)$ is a path in H. Let $|f(s)|$ denote the length of the path $f(s)$. Then

$$\text{WCOST}(f) = \max_{s \in E_k} |f(s)|$$

and

$$\text{ACOST}(f) = \frac{\sum_{s \in E_k} |f(s)|}{|E_k|}.$$

Since by definition contractions are adjacency-preserving, if m is a contraction then $\text{WCOST}(m) \leqslant 1$ and $\text{ACOST}(m) \leqslant 1$.

ACOST and WCOST measure the efficiency of a layout by recording how far a communication path must be stretched to fit the interconnection structure of the target machine. In the extreme, long communication paths, reflected by large ACOSTs and WCOSTs, will cause propagation delay and increase time complexity. Hence the most desirable layouts will keep ACOST and WCOST short and, at best, constant.

The last stage is multiplexing. As noted before, multiplexing costs prevail as a real-time phenomenon rather than as an asymptotic phenomenon. For theoretical upper bounds, we can represent these costs as the computation and routing costs of implementing a parallel algorithm instance with communication graph X on a parallel machine with interconnection architecture X.

Upper Bounds

We use the contraction weighting functions α_m and ω_m in conjunction with the layout expansion costs ACOST and WCOST to give a measure of the efficiency of the entire mapping (contraction, layout, and multiplexing). Since the cost of communicating between processors may differ from the cost of computing at the processor sites, we determine the cost of routing data separately from the cost of computing data.

DEFINITION. A *routing* step is a unit which represents the maximum time it takes to pass a complete message from one processor to another in the target machine. Let $T_r(G_n, H)$ (respectively, $A_r(G_n, H)$) be the worst-case (respectively, average case) number of routing steps it takes to execute algorithm instance G_n on architecture H. A *computing* step is a unit which repre-

sents the maximum time it takes to execute a single basic operation at a processor in the target machine. Let $T_c(G_n, H)$ (respectively, $A_c(G_n, H)$) be the worst-case (respectively, average case) number of computing steps it takes to execute algorithm instance G_n on architecture H.

Note that $T_c(X, X)$ is the computing overhead for executing communication graph X on interconnection architecture X and $T_r(X, X)$ is the routing overhead for executing communication graph X on interconnection architecture X. Together these represent the costs associated with multiplexing.

To evaluate the computing and routing costs of implementing algorithm G_n on architecture H using our methods, the following theorem provides an upper bound for $T_c(G_n, H)$, $T_r(G_n, H)$, $A_c(G_n, H)$, and $A_r(G_n, H)$.

THEOREM 2 (Upper bounds). *Let f be a mapping of G_k into H. Let $n \geq k$ and let m be an iterated contraction from G_n into G_k. Then*

$$T_r(G_n, H) \leq (\text{WCOST}(f))(\omega_m(E_k))(T_r(G_k, G_k))$$
$$A_r(G_n, H) \leq (\text{ACOST}(f))(\alpha_m(E_k))(A_r(G_k, G_k))$$
$$T_c(G_n, H) \leq (\omega_m(V_k))(T_c(G_k, G_k))$$
$$A_c(G_n, H) \leq (\alpha_m(V_k))(A_c(G_k, G_k)).$$

Proof. Let $n \geq k$. Consider the algorithm which maps G_n onto H by contracting G_n into G_k and then laying out G_k in H. (See Fig. 4).

To compute the worst-case routing time, $T_r(G_n, H)$, note that $\omega_m(E_k)$ is the maximum number of edges mapped onto each edge in G_k and that WCOST(f) is the maximum expansion cost for laying out each edge in G_k on H. Hence the worst-case time for routing G_n on H is bounded by (WCOST(f))($\omega_m(E_k)$) times the routing overhead for G_k. Similarly, the worst-case computing time for G_n on H is bounded by the computing over-

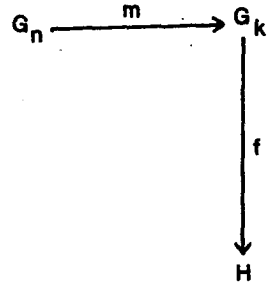

FIG. 4. The mapping procedure used in Theorem 2.

head for G_k times the maximum number of processors $\omega_m(V_k)$ to be simulated by an individual PE. The average case complexities are analogous.

COROLLARY 1. *There is a mapping of the size* $2^n - 1$ *linear systolic array* L_n *into the* $(2^k - 1)$*-node complete binary tree* B_k *with*

$$T_r(L_n, B_k) \leq 3(2^{n-k}e)T_r(L_k, L_k)$$

$$A_r(L_n, B_k) \leq \left(2 - \frac{2}{2^k - 1}\right)(2^{n-k}e)A_r(L_k, L_k)$$

$$T_c(L_n, B_k) \leq (3[2^{n-k-1}]v + 2^{n-k}e)T_c(L_k, L_k)$$

$$A_c(L_n, B_k) \leq \frac{(2^n - 1)v + (2^{n-k+1} - 2)e}{2^k - 1} A_c(L_k, L_k).$$

Proof. The Type 3 edge grammar Γ with start symbol L and rules

$$L \to L(0, 0) \qquad L \to L(1, 1)$$

$$L \to E_I(0, 2) \qquad L \to E_I(2, 1)$$

$$E_I \to E_O(0, 0) \qquad E_O \to E_O(1, 1)$$

$$L \to (0, 2) \qquad L \to (2, 1)$$

$$E_I \to (0, 0) \qquad E_O \to (1, 1)$$

$$L \to E_O \qquad L \to E_I$$

generates the family of lines $\{L_n\}$ where L_n is the line with $2^n - 1$ nodes (Fig. 5). This family is 1-truncatable by Theorem 1. Using truncation to contract L_n onto L_k, we get

$$\omega_m(E_k) = \alpha_m(E_k) = 2^{n-k}e$$

$$\omega_m(V_k) = 3(2^{n-k-1})v + 2^{n-k}e$$

$$\alpha_m(V_k) = \frac{(2^n - 1)v + (2^{n-k+1} - 2)e}{2^k - 1}.$$

By a theorem of Rosenberg and Snyder [RS], there is an embedding f of the length $2^k - 1$ line into the $(2^k - 1)$-node tree with

$$\text{WCOST}(f) \leq 3$$

$$\text{ACOST}(f) \leq 2 - \frac{2}{2^k - 1}.$$

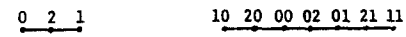

FIG. 5. Lines L_1 and L_2 as defined by the edge grammar Γ.

Hence Theorem 1 can be instantiated to provide upper bounds for embedding a linear array into a complete binary tree.

Note that slightly different upper bounds can be obtained by laying out L_n on B_n (Fig. 1) and then contracting B_n down to B_k. (This inverts our usual ordering of contraction and then layout.) This inversion gives us upper bounds

$$T_r(L_n, B_k) \leq 3(2^{n-k}e)T_r(B_k, B_k)$$

$$A_r(L_n, B_k) \leq \left(2 - \frac{2}{2^k - 1}\right)2^{n-k}eA_r(B_k, B_k)$$

$$T_c(L_n, B_k) \leq ((2^{n-k+1} - 1)v + (2^{n-k+1} - 2)e)T_c(B_k, B_k)$$

$$A_c(L_n, B_k) \leq \frac{(2^n - 1)v + (2^{n-k+1} - 2)e}{2^k - 1} A_c(B_k, B_k).$$

Corollary 1 shows that any algorithm defined on a $(2^n - 1)$-processor linear systolic array may be implemented on a $(2^k - 1)$-processor tree machine with a $O(2^{n-k})$ increase in computing and routing time. This demonstrates that a tree machine evenly distributes processes and communication paths from the linear systolic array and does not lose efficiency (i.e., increase computing or routing time disproportionately) over a nonmultiplexed linear systolic array.

COROLLARY 2. *There is a mapping of the depth n complete binary tree B_n into the size k shuffle-exchange graph SE_k with*

$$T_r(B_n, SE_k) \leq 2^{n+1-k}eT_r(B_k, B_k)$$

$$A_r(B_n, SE_k) \leq \frac{3}{2}(2^{n-k}e)A_r(B_k, B_k)$$

$$T_c(B_n, SE_k) \leq [(2^{n+1-k} - 1)(v + e) - e]T_c(B_k, B_k)$$

$$A_c(B_n, SE_k) \leq \frac{(2^{n+1-k} - 2)e + (2^n - 1)v}{2^k - 1} A_c(B_k, B_k).$$

Proof. The procedure given by Theorem 2 can contract the $(2^n - 1)$-node complete binary tree B_n into the $(2^k - 1)$-node complete binary tree B_k using the edge grammar Γ given in Section 2. The last step is to lay out B_k on SE_k, the shuffle-exchange graph with 2^k nodes. This can be accomplished efficiently using the following lemma.

LEMMA. *There is a mapping f from B_k into SE_k which is injective and has* ACOST $= \frac{3}{2}$ *and* WCOST $= 2$.

Proof of Lemma. Construct the mapping f as follows: Assign the root of the tree B_k to the node labeled $0^{k-1}1$ in SE_k. For each node v in B_k, if $f(v) = x$ in SE_k, then let f(left son of v) be the shuffle (S) of x and f(right son of v) be the shuffle and then an exchange (SE) of x. This mapping is illustrated for B_4 and SE_4 in Fig. 6.

Clearly, half of the edges in B_k are mapped into paths of length 1 and half of the edges are mapped into paths of length 2. Hence the average cost of the layout is $\frac{3}{2}$ and the worst-case cost is 2. It is left to show that f maps distinct nodes in B_k into distinct nodes in $f(B_k) \subseteq SE_k$. To do this, we prove two claims.

CLAIM 1. *For $0 \leq j \leq k$, the images of all vertices in B_k at depth j are distinct.*

Proof of Claim 1. By definition, all vertices at depth $j > 0$ in B_k are mapped into labels in SE_k of the form $x10^{k-j-1}a$, where $x \in \{0, 1\}^{j-1}$ and $a \in \{0, 1\}$. Since the path from the root to every vertex on level j uniquely determines the sequence of S and SE moves (i.e., the sequence of 0- and 1-bits), these labels have 2^{j-1} distinct prefixes and each prefix has two distinct suffixes. Hence the images of all nodes at depth j in B_k are distinct.

CLAIM 2. *Let v be the father of a subtree T of B_k. Then for all w in T, $f(v) = f(w)$ iff $v = w$.*

Proof of Claim 2. Let T be a subtree of B_k with root v. Let w be a descendant of v in T. Then $f(w)$ can be reached from $f(v)$ in SE_k by a sequence of S moves or a sequence of S and SE moves. If $f(w)$ can be reached from $f(v)$ by a sequence of S moves, then $f(v)$ and $f(w)$ are distinct since each path of S moves represents a nonoverlapping part of a necklace (cycle) in SE_k because the root in B_k is assigned $0^{k-1}1$. If $f(w)$ can be reached from $f(v)$ by a sequence of S and SE moves, then $f(w)$ has more 1-bits than $f(v)$. Hence $f(v)$ and $f(w)$ are distinct.

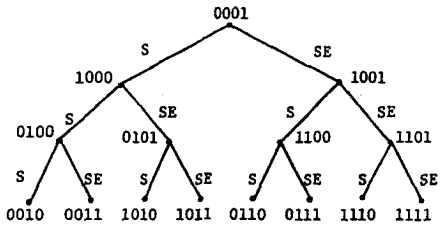

FIG. 6. Mapping of a 15-node complete binary tree into a 16-node shuffle-exchange graph. Edges labeled by S are mapped into shuffle edges; edges labeled by SE are mapped into paths of shuffle and then exchange edges.

We prove the main lemma by induction on the depth of B_k. We show, "If $0 \leq j \leq k$ and T is a subtree of $f(B_k)$ of depth j, then for all nodes v and w in $T, f(v) = f(w)$ iff $v = w$."

Let $j = 0$. Clearly the statement holds for one-node trees.

Now assume that the statement holds for all subtrees with depth less than or equal to $j-1$. Consider a subtree T of B_k with depth j and root r. Then $f(r) = w10^{k-h-1}a$ where $a \in \{0, 1\}$. By construction, $0 \leq h \leq k - j$. Consider the subtree T' of T with depth $j - 1$ and root r (Fig. 7). By the induction hypothesis, every node in the left subtree of T has distinct images under f and every node in the right subtree of T has distinct images under f. Also, the image under f of every node in T' is distinct (all three subtrees have depth $j - 1$). By Claims 1 and 2, the images of all leaves of T are distinct and $f(r)$ is distinct from the image of any node in the right or left subtree of T'. It is left to show that the image under f of any leaf in the left (right) subtree of T is distinct from the image of any internal node in the right (left) subtree of T.

Assume without loss of generality that l is a leaf in the left subtree of T. Let v be an internal node in the right subtree of T'. By construction, $f(l)$ is $uaw10^{k-h-j-1}0$ or $uaw10^{k-h-j-1}1$. Also by construction, $f(v)$ is $xaw10^{k-y-h-1}0$ or $xaw10^{k-y-h-1}1$ where $y < j$. Since the suffixes of $f(l)$ and $f(v)$ are different, $f(l) \neq f(v)$. Consequently, the image under f of every node in T is distinct.

Hence the embedding f is injective.

Corollary 2 shows that we can find an efficient upper bound for mapping the complete binary tree B_m into the shuffle-exchange graph SE_k by contracting the tree and then using the layout given in the lemma. In particular, this demonstrates that it is advantageous to run tree algorithms on devices with the shuffle-exchange topology, for example, the original ultracomputer of Schwartz [S2]. Since we can simulate $2^n - 1$ processors on 2^k processors with only a $O(2^{n-k})$ increase in routing and computing time complexity, both routing and computing time achieve full speedup with our methods.

COROLLARY 3. *There is a mapping of the size $2^n \times 2^n$ square mesh M_n into the $(2^{2k+1} - 1)$-node complete binary tree B_k with*

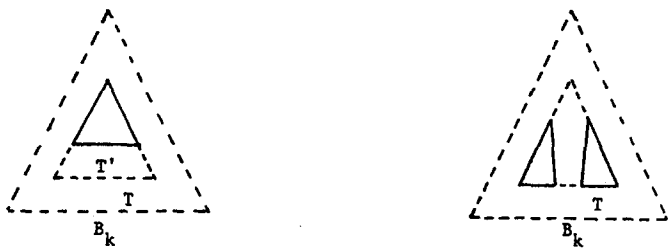

FIG. 7. Left: Subtree T' of T. Right: The left and right subtrees of T.

$$T_r(M_n, B_{2k+1}) \leq (4k)2^{n-k}eT_r(M_k, M_k)$$
$$A_r(M_n, B_{2k+1}) \leq (8)2^{n-k}eA_r(M_k, M_k)$$
$$T_c(M_n, B_{2k+1}) \leq (4^{n-k}v + 2e(4^{n-k} - 2^{n-k}))T_c(M_k, M_k)$$
$$A_c(M_n, B_{2k+1}) \leq (4^{n-k}v + 2e(4^{n-k} - 2^{n-k}))A_c(M_k, M_k).$$

Proof. The family of $2^n \times 2^n$ square meshes $\{M_n\}$ can be given by the following 1-truncatable Type 3 edge grammar.

$M \to N$	$M \to H$	$M \to V$
$M \to (1, 2)$	$M \to (2, 3)$	$M \to (3, 4)$
$M \to (4, 1)$	$N \to (1, 1)$	$N \to (2, 2)$
$N \to (3, 3)$	$N \to (4, 4)$	$H \to (1, 2)$
$H \to (4, 3)$	$V \to (2, 3)$	$V \to (1, 4)$
$M \to N(1, 2)$	$M \to N(2, 3)$	$M \to N(3, 4)$
$M \to N(4, 1)$	$N \to N(1, 1)$	$N \to N(2, 2)$
$N \to N(3, 3)$	$N \to N(4, 4)$	$V \to V(4, 1)$
$V \to V(3, 2)$	$H \to H(2, 1)$	$H \to H(3, 4)$
$V \to N(1, 4)$	$V \to N(2, 3)$	$H \to N(4, 3)$
$H \to N(1, 2)$		

Contracting M_n with labels in this edge grammar coalesces squares in the graph as shown in Fig. 3. Using this contraction, we get

$$\omega_m(V_n) = 4^{n-k}v + 2^{n-k}(2e)(2^{n-k} - 1) = \alpha_m(V_n)$$
$$\omega_m(E_n) = 2^{n-k}e = \alpha_m(E_n).$$

By a theorem of DeMillo, Eisenstat, and Lipton [DEL], the layout of M_n into the tree B_{2k+1} can be done so that the average expansion of an edge is no larger than 8. This layout yields a worst-case expansion of $2k$. Although $2k$ appears long, DeMillo *et al.* have shown that any embedding of a $2^k \times 2^k$ square mesh M_k into a complete binary tree requires a worst-case expansion of at least $k - 3/2$ [DEL].

Corollary 3 shows that although some edges in the array may be expanded over long (up to $2k$) paths, the average routing cost of implementing array algorithms on tree architectures is considerably more efficient and achieves full PE utilization. Since many of the interprocess communication steps are converted into intraprocessor computations (reflected in T_c and A_c), we can

expect the actual running time of the algorithm to be slightly better than these complexity functions indicate.

We have described a general procedure for mapping parallel algorithms into parallel architectures. By Theorem 1, this procedure can be automated for a large class of commonly used parallel algorithms and architectures. In addition, Theorem 2 gives a measure of the upper bounds provided in general. Our experience with these methods on a variety of benchmark problems has been typical of the examples presented here: In general, these methods produce proportionally full processor utilization and full distribution of communication paths in the algorithm over data paths in the target architecture. In order to gauge the complexity of the multiplexing procedures and evaluate the actual performance of these solution techniques, the first author is designing a preprocessor for CHiP architecture [S3] based on this methodology. More details can be found in [BGKRS].

ACKNOWLEDGMENTS

We thank Dennis Gannon and Janice Cuny for their support and encouragement. We also thank the referees for their helpful comments and suggestions.

REFERENCES

[AHU] Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, Reading, MA, 1975.

[B1] Berman, F. Edge grammars and parallel computation. *Proc. 1983 Allerton Conference,* Urbana, IL, 1983.

[B2] Bokhari, S. On the mapping problem. *IEEE Trans. Comput.* **C-30,** 3 (Mar. 1981).

[B3] Browning, S. A. "The tree machine: A highly concurrent programming environment," Ph.D. thesis, California Institute of Technology, 1980.

[BGKRS] Berman, F., Goodrich, M., Koelbel, C., Robison, W., and Showell, K. A guide to the poker mapping preprocessor. Purdue Tech. Rep. 488, Aug. 1984.

[BS] Berman, F., and Shannon, G. Edge grammars: Decidability results and formal language issues. *Proc. 1984 Allerton Conference,* Urbana, IL, 1984.

[CER] Claus, V., Ehrig, H., and Rozenberg, G. *Graph Grammars and Their Application to Computer Science and Biology,* Lecture Notes in Computer Science, Vol. 73. Springer-Verlag, Berlin/New York, 1979.

[D1] DeGroot, D. Partitioning job structures for SW–Banyan networks. *Proc. 1983 International Conference on Parallel Processing.*

[D2] DeGroot, D. Expanding and contracting SW–Banyan networks. *Proc. 1983 International Conference on Parallel Processing.*

[DEL] DeMillo, R., Eisenstat, S., and Lipton, R. Preserving average proximity in arrays. *Comm. ACM* **21,** 3 (March 1978).

[FF] Fishburn, J., and Finkel, R. Quotient networks. *IEEE Trans. Comput.* **C-31,** 4 (1982).

[IB] Iqbal, M. A., and Bokhari, S. New heuristics for the mapping problem. Res. Rep. EECE-83-02, Department of Electrical Engineering, University of Engineering and Technology, Lahore, Pakistan, June 1983.

[P] Pratt, T. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Systems Sci.* **5,** 6 (1971).

[RS] Rosenberg, A., and Snyder, L. Bounds on the costs of data encodings. *Math. Systems Theory,* **12** (1978).

[S1] Salomaa, A. *Formal Languages,* Academic Press, New York, 1973.

[S2] Schwartz, J. Ultracomputers. *ACM Trans. Programming Languages Systems* **2,** 4 (1980).

[S3] Snyder, L. Introduction to the configurable, highly parallel computer. *Computer* **15,** 1 (Jan. 1982).

[S4] Stone, H. Parallel processing with the perfect shuffle. *IEEE Trans. Comput.* **C-20,** 2 (1971).