

Generic Program Structures Induced by Partitions of a Systolic Computation Graph

Brian J. d'Auriol*

Department of Computer Science

The University of Manitoba

Winnipeg, Manitoba,

R3C 2N2, Canada

Virendrakumar C. Bhavsar†

Faculty of Computer Science

University of New Brunswick

Fredericton, New Brunswick, E3B 5A3, Canada

*Partially supported by NSERC doctoral fellowship.

†Partially supported by NSERC grant, OGP0089.

Corresponding Author:

Brian J. d'Auriol

Department of Computer Science

The University of Manitoba

Winnipeg, Manitoba, Canada

R3T 2N2

Phone: (204)474-8831

Fax: (204)269-9178

email: bdauriol@cs.umanitoba.ca

URL: <http://www.cs.umanitoba.ca/~bdauriol>

Second Author:

Virendrakumar C. Bhavsar

Faculty of Computer Science

University of New Brunswick

Fredericton, New Brunswick, Canada

E3B 5A3

Phone: (506)453-4566

Fax: (506)453-3566

email: bhavsar@unb.ca

URL: <http://www.cs.unb.ca/profs/bhavsar.html>

Abstract

We propose a technique which essentially constructs a set of concurrent program archetype definitions, each representing a valid parallel implementation of a sequentially specified program. Our approach is based on partitioning a systolic computation graph and the subsequent mappings of the computations involved to processors in a multicomputer environment. We show that the partitioning and mapping process induces a specific processor topology dependent on the partitioning strategy used. The processing requirements based on the topology imply a specific program structure. The details of resulting program structures are illustrated. The formalism contained in this paper is useful for the construction of automated tools to support the generation of parallel program codes.

Keywords

Generic Program Structures, Concurrent Program Archetype, Partitioning and Mapping, Systolic Computations

Nomenclature

<u>Symbol</u>	<u>Description</u>	<u>Special Instructions</u>
\mathcal{A}	The input algorithm	script
b	A constant	math-italics
c	An arbitrary partition curve	math-italics
c_j^i	A particular curve in \mathbf{C}_i	math-italics
\mathbf{C}	A set of related partition curves	bold
$\mathbf{C}(s)$	A partition \mathbf{C} applied to s	bold, math-italics
d	A data value	math-italics
$\deg(v)$	The degree of vertex v	math-italics
e	An arbitrary edge in a graph or data sequencing	math-italics
E	Edges in G	math-italics
\bar{E}	A special set of edges	math-italics
E^I	A set of edges in I	math-italics
$E^{\hat{I}}$	A set of edges in \hat{I}	math-italics
E^p	Edges in G^p	math-italics
$f_{i,j}$	An element of F which has been mapped to $v_{i,j} \in \hat{G}$	math-italics
F	Computations associated with s	math-italics
F^*	The set of ordered sets of computations $f_{i,j}$	math-italics
\hat{F}	Computations associated with \hat{G}	math-italics
G	A graph associated with s	math-italics
G^p	A processor graph	math-italics

Nomenclature (continued)

<u>Symbol</u>	<u>Description</u>	<u>Special Instructions</u>
\hat{G}, \hat{G}'	A restricted graph associated with s	math-italics
i, j, k, l	Indices and/or constants	math-italics
I	The data sequencing associated with s	math-italics
\hat{I}	The data sequencing associated with \hat{G}	math-italics
\mathbf{M}^s	Software system necessary to implement \mathcal{A}	bold, superscript math-italics
$\overline{\mathbf{M}}^s$	A basis definition of \mathbf{M}^s	bold, superscript math-italics
M_C^s	The configuration module	normal, sub/superscripts math-italics
M_{CM}^s	The communications module	normal, sub/superscripts math-italics
M_F^s	The computations module	normal, sub/superscripts math-italics
M_{IO}^s	The input/output module	normal, sub/superscripts math-italics
M_R^s	The communication router module	normal, sub/superscripts math-italics
N_i, N_j	The dimensions of \hat{G}	math-italics
s	A systolic graph	math-italics
t_i	The i^{th} time instant	math-italics
$v_{i,j}$	Arbitrary vertices in a graph	math-italics
V	Vertices in G	math-italics
V^p	Vertices in G^p	math-italics
W	An input wave	normal
x, y	Cartesian coordinates	math-italics
ζ	A set of characteristic partition curves	Greek
$\zeta(v)$	A set of partitions applied to s	Greek, math-italics

Nomenclature (continued)

<u>Symbol</u>	<u>Description</u>	<u>Special Instructions</u>
σ_+, σ_-	Functions which map edges to vertices in G	Greek
σ_+^I	Functions which map edges in I to vertices in G	Greek
$\sigma_+^{\hat{I}}$	Functions which map edges in \hat{I} to vertices in \hat{G}	Greek
σ_-^p, σ_+^p	Functions which map edges to vertices in G^p	Greek
ς_i^I	Scheduling function for I	Greek
$\varsigma_i^{\hat{I}}$	Scheduling function for \hat{I}	Greek
{ }	Set notation	
()	Cartesian coordinate, ordered set notation, paranthesis	
–	Set difference	
∈	Set membership	
:	Is defined as	
↦	Maps to	
	Such that	
∀, ∃	Logic quantifiers	

Figure Captions

- Figure 1 Structure of \hat{G} for a systolic array of a 5×5 array size and examples of three characteristic partitions on \hat{G} .
- Figure 2 Partitioning algorithm.
- Figure 3 An example for the partitioning algorithm.
- Figure 4 The graphs resulting from the application of the partitioning to the example.
- Figure 5 An example of diagonal partitioning (ζ_d).
- Figure 6 Bidirectional linear network resulting from applying ζ_d on \hat{G} .
- Figure 7 An example of orthogonal partitioning (ζ_o) for partition lines of the form $x = b$.
- Figure 8 Linear network resulting from applying ζ_o on \hat{G} .
- Figure 9 A derived program structure.
- Figure 10 occam 2 pseudocode showing a particular coding of M_{IO}^s for matrix multiplication with a farming implementation.
- Figure 11 occam 2 source of M_F^s for matrix multiplication with a farming implementation.
- Figure 12 occam 2 source of M_F^s for the diagonal partitioning implementation of the WLD.

1 Introduction

A program archetype has been defined by Chandy as “(a) a program design strategy appropriate for a restricted class of problems, and (b) a collection of program designs with (c) implementations of exemplar problems in one or more programming language...” [3]. Parallel program archetypes have been proposed as a method to reduce the effort required to develop and implement parallel programs [3]. In this paper, we present a derivation technique which essentially constructs a set of program archetype definitions, each representing a valid parallel implementation of a sequentially specified program source for systolic algorithms. Our other work includes additional aspects of program archetypes, for example, performance modeling of the generated parallel implementations [6, 7].

Our approach is based on partitioning a reduced data dependency graph (often referred to as a systolic computation graph) representing a systolic algorithm and the subsequent mappings of the computations involved to processors in a multicomputer environment. Data dependency graphs (see [23, 1] and the references therein) are well known graphical representations of the dependence relations between statements. Systolic computation graphs can be generated by a projection-like transformation of the data dependency graph onto, often, a two-dimensional surface [17], that is, systolic array synthesis. In this paper, we assume that the systolic computation graph is given, and consequently, are not concerned with the systolic array synthesis issues.

In this paper, we are primarily interested in the cause-effect relationship between partitioning the systolic computation graph and the program structure, represented in a high level language, necessary to represent a parallel implementation. By program structure we mean the definition and interrelation of program modules together with their interface and implementation descriptions. Note that we investigate a different problem than the more traditional one of partitioning the systolic computation graph onto an existing computational device (e.g. a VLSI device or MIMD computer). The latter is a well researched area (see for example [17, 20, 5] and the references therein). Our interest in partitioning is confined to

such aspects that relate to the derivation of program structures. In this work, we have been heavily influenced by the occam 2 language [12] and transputer architectures [13]. Consequently, our model has been structured in a way so as to be able to represent the important features of this environment; however, the model is not limited to this environment.

We develop a graph-based notation to represent systolic computation graphs and note that there are similarities with the notation given in [18]. The principal advantage of our formalism is that it allows for both the definition and partitioning of systolic computation graphs in an integrated model. There are several other formal models (see for example [20, 18, 16] and the references therein) that could possibly be extended for the same purpose.

This paper is organized as follows. A particular systolic computation graph which we use to represent a given input algorithm is presented in the next section. In Section 3, the technique used to construct the archetypes is presented while program structure details resulting from the use of this technique are presented in Section 4. Concluding remarks are given in Section 5.

2 Systolic Computation Graph

Generally, the geometry of a systolic array is represented by a directed graph (see for example [18]). We denote the geometry of an arbitrary systolic array as $G = (V, E, \sigma_-, \sigma_+)$, where V is the set of vertices, E is the set of directed edges, and σ_-, σ_+ are functions which map E to V with $\sigma_-(e), \sigma_+(e)$ representing the source and destination nodes respectively for the edge $e \in E$. The degree of a vertex is denoted by $\deg(v)$. Each vertex in V has associated with it a computation; we denote the set of computations associated with all vertices as F . Another component of systolic arrays is the data sequencing of the inputs; this sequencing has both time and space dimensions. We denote a datum value by d and define the data sequencing as $I = (E^I, \sigma_+^I, \zeta^I)$ where E^I is the set of directed edges not in E which correspond to the paths of the data, σ_+^I is a function which maps E^I to V representing the destination node for each $e \in E^I$, and ζ^I is a scheduling function which establishes a time-ordered

sequence for the data items associated with each edge: $\zeta^I(e \in E^I, d) \mapsto \{0, 1, 2, \dots\}$.

We define a systolic computation graph, s , as $s = (G, I, F)$. This represents a generalization of previous definitions relating to systolic arrays, for example, the selection of common geometries that appear in [17] including the hexagonal, triangular and binary H-tree can all be represented by the notation. In certain cases, the progression of computations in s will exhibit a wavefront property. This has often been referred to as a *wavefront array* [14].

We denote by \mathcal{A} a given input algorithm that must conform to the usual constraints imposed by systolic algorithms, for example, a set of uniform recurrence relations combined with the recurrence relation inputs (sometimes referred to as the initial conditions of the recurrence) as well as the lower and upper bounds on the relations so as to constrain the domain of the indices of the recurrence relation to bounded regions [17]. We note that there are several systolic array design methodologies which may be used to transform \mathcal{A} into s (see for example [17, 20]); moreover, such synthesis techniques often result in a family of different systolic computation graphs.

We consider a graph which has the following properties and denote it as \hat{G} :

- (a) A two-dimensional planar digraph with nearest neighbor connections such that for all interior vertices, a 6-regular graph of indegree three and outdegree three, referred to as a hexagonal topology [17, page 16]. This graph has finite but arbitrary dimensions. This geometry is illustrated in Figure 1.
- (b) The set of directed arcs that are allowable by (a) are further restricted by imposing the wavefront property.
- (c) There is a single source and a single sink for data.

The property (a) implies that \hat{G} is a super-graph of many of the more common geometries in systolic computations [19]. The properties (b) and (c) simplify the coding requirements of the source code program representation (e.g. by eliminating global synchronization and imposing a single input and output point in the code, respectively). Furthermore, property

(b) also provides a mechanism to allow parallel processing (e.g. computation aligned along the wavefront can be carried out simultaneously). It is noted that property (c) poses little or no restrictions on its use in a general context since a direct application of the retiming lemmas of Leiserson and Saxe [15], as discussed by Megson [17], can perform the transformation of an arbitrary systolic computation graph with many inputs and outputs to \hat{G} .

However, the single source and sink requirement does impact both on I and F since the cardinality of E^I now equals one; also, the set of computations mapped to the boundary vertices will also need to be augmented to allow for the distribution of the data from the source to the upper and left boundary nodes. We denote by $\hat{I} = (E^{\hat{I}}, \sigma_+^{\hat{I}}, \zeta_i^{\hat{I}})$ and \hat{F} , the newly transformed data sequencing and the set of computation functions, respectively. Note that the original definitions of I and F remain valid with respect to the overall behavior of s despite the requirements imposed by \hat{I} and \hat{F} .

3 Partitioning of the Systolic Computation Graph

There are two principal forms of partitioning: locally parallel globally sequential (LPGS) and locally sequential globally parallel (LSGP) [5]. The former partitions a systolic array into blocks which are then computed sequentially in time while the latter partitions the array into blocks which are then allocated to a processor. We discuss several partitioning schemes in this section which are either LPGS or LSGP. We are interested here in those aspects of partitioning which induce program structures and not in the partitioning of the systolic graphs for purposes of mapping onto specific computational devices.

3.1 Notation

We embed \hat{G} into the Cartesian plane such that the origin coincides with the source vertex in \hat{G} . Vertices in \hat{G} are identified by $v_{i,j} \in V$ where $i, j \in \{0, 1, 2, \dots\}$ such that for specific values of i, j , (i, j) is a point in the x - y plane. Consequently, all $e \in E$ have an associated

direction vector from the set $\{(0,1), (1,0), (1,1)\}$ with a corresponding edge length of 1, 1, or $\sqrt{2}$ respectively. The dimensions of the graph are denoted by N_i and N_j in the x and y direction, respectively. We denote by $f_{i,j}$ an element from F which is associated with the vertex $v_{i,j}$. A sequence of computations is denoted by $(f_{i_0,j_0}, f_{i_1,j_1}, \dots)$ and the set of such sequences, such that each sequence is independent of any other sequence, is denoted by F^* . Where necessary, we show the arguments to a particular function as $f(d_0, d_1, \dots)$.

We denote by c a simple curve which induces a cut of \hat{G} such that the two sub-graphs of \hat{G} are connected by the cutset induced by c . Furthermore, c is restricted such that it does not pass through any vertex in \hat{G} . We group related curves together (where the relation is based on some set of characteristics, for example, lines having the same slope) and denote this set by \mathbf{C} . We denote by ζ a set of related sets of curves, $\zeta = \{\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C}_{\nu_\zeta}\}$, where each \mathbf{C}_i is a set of zero or more curves; each individual curve is denoted by $c_j^i \in \mathbf{C}_i$, for $i, j > 0$. Essentially, each \mathbf{C} represents a complete partitioning of s such that the induced program structure fully represents the input algorithm (we discuss the resulting program structures from this technique in Section 4). The collection ζ (as will be discussed later in this section as well as in the next section) essentially groups several \mathbf{C} sets together which result in the same program structure. The set of vertices in the sub-graphs induced by \mathbf{C}_i forms blocks of partitions. This type of partitioning is a *spatial* partitioning of \hat{G} and leads to an LSGP type of partition.

We also consider the temporal partitioning of \hat{G} . We define an *input wave*, W , as the collection of data inputs given to s at the same time: $W_i : I|_\zeta^I(\forall e \in E^I, d) \mapsto i$. Let c provide a cut of the time line. Consequently, c partitions the input waves into two collections of waves or *wave regions*. The three-level hierarchy of $c \in \mathbf{C} \in \zeta$ remains consistent in this partitioning case as well. This type of partitioning leads to an LPGS type of partition.

In this paper, we consider four particular characteristic sets of curves: *Diagonal* (ζ_d), *Orthogonal* (ζ_o), *Temporal* (ζ_t), and *Null* (ζ_s). The characteristic for ζ_d is that all lines in ζ_d have the form, $y = x + b$, while for ζ_o , the lines have either the form $y = b$ or $x = b$, but not

both. The null partition is the absence of any partition and may be considered as a special case of the others.

We define an *augmented* graph of \hat{G} as $\hat{G}' = (V, E \cup \bar{E}, \sigma_-, \sigma_+)$, where \bar{E} is a set of diagonal edges perpendicular to the set of diagonal edges in E such that $\sigma_-(e \in \bar{E}) = v_{i,j}$ and $\sigma_+(e \in \bar{E}) = v_{i+1,j-1}$. The augmented graph is used in the partitioning algorithm (discussed below) so that each vertex has an edge connected to one of its eight neighboring vertices. The incorporation of \bar{E} edges in the graph does not change the behavior of s in any way.

Lastly, we define the processor topology of a parallel machine as a graph $G^p = (V^p, E^p, \sigma_+^p, \sigma_-^p)$. A node $v_{i+1} \in V^p$ denotes a vertex such that $\exists e \in E^p | \sigma_+^p(e) = v_{i+1}$ and $\sigma_-^p(e) = v_i$. We are interested in deriving the necessary processor topology that a parallel machine should have in order to execute a program based on a given partitioning strategy. In subsequent discussions, we show that G^p is a graph consisting of supernodes and superedges, that is, each vertex in V^p itself contains a graph of vertices and edges while each edge in E^p itself contains one or more edges (actually, the graphs contained in the supernodes and superedges are sub-graphs of \hat{G}).

Figure 1 illustrates some of the notation introduced above for a 5 by 5 array size. The directions of the axes are shown in the upper left part of the figure and the actual origin is indicated by the filled in node. The input data is applied to the source, that is, to the single edge $e \in E^I$, $\sigma_+^I(e) = v_{0,0}$. This edge is identified by the dashed arrow pointing to the source or origin node. Two input waves, W_0 and W_1 , are shown by the light dashed lines. Edges in I are shown by light dashed arrows only for W_0 . The example curves c_d , c_o and c_t from ζ_d , ζ_o , and ζ_t respectively, are shown by heavy dashed lines.

3.2 Partitioning Process

In this subsection, a graph representing the topological requirements of a parallel machine is constructed by applying a partition to the systolic computation graph: $\mathbf{C}(s) \mapsto G^p$. The

algorithm we propose to construct G^p is given in Figure 2. Essentially, this algorithm maps each block of the partition to a unique vertex in V^p and edges in each cutset to a unique edge in E^p . In this algorithm, we use the augmented graph \hat{G}' . For all edges in \hat{G}' , if an edge intersects with a $c \in \mathbf{C}$ then source and sink vertices of that edge are added to the supernode set of V^p if required and that edge is added to the superedge set of E^p . This case represents Steps 1a-1c in Figure 2. However, if the edge does not intersect with any c , then the source and sink vertices of that edge are added to the same supernode graph if necessary; also, the edge itself is added to the supernode graph that contains the source node of that edge. Under certain conditions, the graphs of two supernodes may need to be merged together (refer to Example 1). The non-intersecting case represents Steps 2a-2e in Figure 2.

As a result of using the partitioning algorithm, each vertex in G^p represents a block of the partition defined by $\mathbf{C}(s)$. This is evident since each edge in G^p exactly corresponds to all edges in \hat{G} which intersect the corresponding cut. And, any vertex in G^p has been constructed such that at least one outgoing and one incoming edge cross a cut line (other than the two blocks of partition which contain the source and sink). Also, all vertices in \hat{G} which have at least one edge that does not cross a cut (i.e. an edge which is fully internal to block of the partition) are amalgamated into a single vertex. Consequently, all vertices in s are covered.

We consider that edges in E^p represent external (i.e. processor-to-processor) communication while edges in E which are contained in V^p represent internal communication. The following example illustrates the partitioning algorithm while several lemmas pertaining to the characteristics of the resulting processor graph are subsequently presented.

Example 1 Figure 3 shows a 3×3 graph and two partition curves, c_1, c_2 . All edges in E , but not in \bar{E} , have been explicitly identified; edges in \bar{E} are shown by dashed lines.

We select the edges in the order given in Figure 3. The edge e_1 is in the cut set defined

by the curve c_1 , thus, Step (1) of the algorithm is executed. Step (1) is also executed due to considering e_2 . However, only $v_{2,0}$ and e_2 are added to G^p . The resulting graph corresponding to these steps is shown in Figure 4(a). When edges e_3 and e_4 are treated similar to edge e_2 , the corresponding graph is shown in Figure 4(b).

Next, due to considering e_5 , Step (2) of the algorithm is executed and consequently, $v_{0,1}$ is added to the *supernode* $v_{0,0} \in V^p$. This procedure effectively combines the two vertices into one. Figure 4(c) shows the resulting graph after considering e_6 and e_7 . Note that the original vertices $v_{1,1}$ and $v_{2,1}$ in G^p no longer exist as single vertices; a different edge selection order could have resulted in these vertices not having been distinct at all. When all the edges in E have been considered, the final graph, G^p , consisting of three supernodes and two superedges, is obtained as shown in Figure 4(d). This graph represents a linear pipeline of three processors required in a parallel machine to implement the given systolic computation graph.

□

Note that in the above example, c_1 and c_2 do not belong to any ζ considered in this paper and are used only to demonstrate a general application of the algorithm. Also, although the edges in \bar{E} were not considered in this example, their use may be necessary in general. For example, if c_1 and c_2 are redefined as $c_1 : y = -x + 0.5$ and $c_2 : y = -x + 1.5$, they induce two cuts parallel to each other *and to edges in \bar{E}* . In this case, $v_{0,1}$ and $v_{1,0}$ could only be identified together by considering edges in \bar{E} .

Lemma 1 *G^p is a minimum graph satisfying the computation requirements given by s .*

Proof.

The proof is by contradiction. Assume G^p is not a minimum graph. Then, any edge in E^p can be removed. However, by the partitioning algorithm, each edge in E^p corresponds to an edge in E which intersects and thus crosses a cut line; also each vertex in V^p is a graph consisting of the vertices of a block of partition. The removal of an edge would clearly violate

the requirement that two blocks of partition are interconnected by an edge which crosses the cut line. The removal of a vertex from V^p would cause the associated graph of the supernode to disappear. Thus, G^p is a minimum graph. \square

Lemma 2 *Given s and ζ_o then $\zeta_o(s) \mapsto \{G_1^p, G_2^p, \dots, G_{v_G}^p\}$ where the family of graphs have the following characteristics:*

1. $\exists v_1, v_2 \in V_i^p$ such that $\deg(v_1) = \deg(v_2) = 1$
2. $\forall v \in V_i^p - \{v_1, v_2\}, \deg(v) = 2$
3. $\forall e \in E_i^p, \exists v_j \in V_i^p$ such that $\sigma_-^p(e) = v_j$ and $\sigma_+^p(e) = v_{j+1}$

Proof.

Since ζ_o is a set of a finite number of sets of \mathbf{C} , we can enumerate all of the resulting G^p by applying the partition algorithm to each $\mathbf{C} \in \zeta$. Assume all $c \in \mathbf{C}$ (where $\mathbf{C} \in \zeta_o$) are of the form $x = b$. Then, by the partition algorithm, all edges in E^p have direction vectors of $\{(1,0), (1,1)\}$. For all such edges, e , with direction vector $(1,0)$, $\sigma_-(e)$ is contained in a $v_k \in V_i^p$ and $\sigma_+(e)$ is contained in a $v_l \in V_i^p$ with $k \neq l$. Moreover, $\exists e^p \in E_i^p | \sigma_-^p(e^p) = v_k$ and $\sigma_+^p(e^p) = v_l$. Edges with direction vector $(1,1)$ result in the same v_k, v_l . The case for $y = b$ is similar. The three conditions given in the lemma summarize these results. \square

Lemma 3 *Given s and ζ_d then $\zeta_d(s) \mapsto \{G_1^p, G_2^p, \dots, G_{v_G}^p\}$ where the family of graphs have the following characteristics:*

1. $\exists v_1, v_2 \in V_i^p$ such that $\deg(v_1) = \deg(v_2) = 1$
2. $\forall v \in V_i^p - \{v_1, v_2\}, \deg(v) = 2$
3. $\forall e \in E_i^p, \exists v_j \in V_i^p$ such that $\sigma_-^p(e) = v_j, \sigma_+^p(e) = v_{j+1}, \sigma_-^p(e) = v_{j+1}$ and $\sigma_+^p(e) = v_j$

Proof.

The proof for Lemma 3 follows that for Lemma 2. □

The processor topologies implied by the partitioning algorithm may or may not be unique. For example, Lemma 2 and Example 1 both result in topologies with the same characteristic, namely, a linear pipeline of processors. This observation leads to the following three possibilities: (a) any two or more arbitrary partitions on s may lead to the same processor topology, (b) any two or more arbitrary partitions may lead to a set of processor topologies which are subgraphs of one of the processor topology graphs, or (c) any two or more arbitrary partitions may lead to distinctly different processor topologies.

We now turn our attention to the temporal partitioning case with ζ_t . The *wave independence* restriction is defined as follows: for two consecutive waves, W_i and W_{i+1} , all sequences in F^* that contain a function f_i operating on a data element from W_{i+1} cannot also contain a function f_j operating on a data element from W_i . For example, $f_{0,0}(d_1, d_2, d_3)$ will always have $d_1, d_2, d_3 \in W_j$ as would $f_{1,1}$, thus, the sequence $(f_{0,0}, f_{1,1})$, does not violate the wave independence restriction. However, the sequence $(f_{0,1}, f_{1,2})$ does violate this restriction since for $f_{0,1}(d_1, d_2, d_3)$, d_1 would be from W_i while d_2 and d_3 would be from W_{i+1} . Ostensibly, when multiple computations of \mathcal{A} are required, ζ_t is restricted in such a way that these multiple computations are carried out independently (e.g. multiple matrix multiplications for different sets of input matrices).

Lemma 4 *Given W, s with dimensions $N_i = N_j$ and ζ_t such that two waves W_i and W_{i+1} which are contained in distinct wave regions induced by ζ_t do not violate the wave independence restriction, then $\zeta_t(s) \mapsto F^*$*

Proof.

Let $N = N_i - 1$. Consider W_0 individually. At t_0 , the computation of $f_{0,0}$ occurs followed by $f_{1,1}$ at t_1 , thus the ordered set of operations: $(f_{0,0}, f_{1,1}, \dots, f_{N,N})$ where the function arguments for the functions in this set are data items from W_0 . Consider now W_1 together with W_0 . Again, the ordered set, $(f_{0,0}, f_{1,1}, \dots, f_{N,N})$ with function arguments from W_0 , exists. There is, additionally, a new ordered set consisting of the same operations, however, the function arguments are from W_1 . As well, the two new ordered sets, $(f_{0,1}, f_{1,2}, \dots, f_{N-1})$ and $(f_{1,0}, f_{2,1}, \dots, f_{N-1})$ exist. We consider all groupings of waves and the subsequent collections of function sequences in similar manner. Thus, $F^* = \{(f_{0,0}, f_{1,1}, \dots, f_{N,N}), (f_{0,0}, f_{1,1}, \dots, f_{N,N}), (f_{0,1}, f_{1,2}, \dots, f_{N-1}), (f_{1,0}, f_{2,1}, \dots, f_{N-1}), \dots, f_{\nu_{F^*}}^*\}$ (note that, although there appear to be duplicate sequences in F^* , the sequences operate on different data items, and are consequently, considered distinct — we do not show the function arguments for simplicity). \square

There are two interpretations of Lemma 4: (a) the set of all waves in a wave region can be input to s , or (b) the set of all computations in s can be reordered into distinct computations which may then be computed individually. In consideration of the restrictions imposed upon the temporal partitioning, the former interpretation essentially reduces to the null partition. In the latter case, we consider such a set of computations to be amenable to the standard processor farming technique [4]. Consequently, we adopt a single linear chain of processors for convenience in this paper and note that such a processor chain is a special case of a general star network commonly used for processor farms.

We informally consider the case when ζ_t is not restricted as above. In this case, there would exist sets of computations in F^* due to some W_i which require inputs from one or more of the previous waves. Such temporal dependences are bounded by a function of the number of previous waves and the size of s . With respect to the two interpretations of Lemma 4, in the first case, multiple copies of s exist where for each s , the input waves are contained in the wave regions defined by ζ_t . Note, that the data inputs for the computations

in F^* for each s would require additional specification, since these inputs are contained in waves allocated to some other s . Regarding the second case of Lemma 4, no change to F^* is required (clearly, however, the input data distributions require additional specification as well). Consequently, our observations regarding the implied topology are reasonable for unrestricted temporal partitioning.

In summary, the processor topologies derived from Lemmas 2, 3 and 4 refer to the example partitions shown in Figure 1, and are as follows: ζ_o (orthogonal) - linear uni-directional, ζ_d (diagonal) - linear bidirectional, ζ_t (temporal) - linear bidirectional (an implementation of a processor farm), and ζ_s (sequential) - single processor. An example of the correspondence between the Diagonal Partitioning strategy and the linear bidirectional processor topology is given in Figures 5 and 6, while a similar example is given in Figures 7 and 8 for the correspondence between the Orthogonal Partitioning strategy and the linear uni-directional processor topology.

4 Program Structures

Specific processor topologies were derived in the previous section based upon the partitioning and mapping process where subsets of F are mapped to specific processors in the topology. In this section, we present the program structure of a high level language program necessary to perform these computations in a distributed manner as implied by the topology. By program structure we mean the definition and interrelation of program modules together with their interface and implementation sections. We concentrate on defining aspects of the cohesion (a measurement of the closeness of the relationships between components internal to a particular module [22]) and coupling (a measurement of the strength of interconnections between modules [22]) of these modules (further discussion of cohesion and coupling may be found in, for example, [22, 11]).

From previous discussions in Section 3, we consider \hat{G} , \hat{I} and \hat{F} as the only components

of \mathcal{A} needed to define a parallel implementation. However, by the partitioning and mapping process, \hat{G} and \hat{F} are transformed and embedded into the structure, G^p . The graph of each supernode in G^p (which includes the subset of computations mapped to that node as well as any of the internal communication requirements given by the edges contained in the supernode) gives the specification for the computational process. We denote by M_F^s this computational process.

The edges in G^p represent required communication between specific instantiations of M_F^s ; however, specifics of how that communication is to be conducted is not a part of \hat{F} and hence, not defined in M_F^s . Therefore, a new module denoted by M_R^s which provides for the communication routing is required.

Due to the single source and sink property of \hat{G} , there is a single point of data input to G^p represented as an arc to a vertex in G^p , and also there is a single point of data output from G^p represented as an arc from a vertex in G^p . Consequently, the effect of \hat{I} in G^p reduces to the abstraction of ‘input/output’ module (i.e. that which allows input to and from \hat{F}). Additionally, there is the requirement for data input and output to and from secondary storage (e.g. file storage). We denote this module by M_{IO}^s .

The definitions of the communication specific data types including both variable and channel typing needed by the previously discussed modules is contained in the ‘communications module’ denoted by M_{CM}^s . Process instantiation and allocation of processes to processors is provided by the ‘configuration module’ denoted by M_C^s .

We denote by \mathbf{M}^s the software system necessary to implement \mathcal{A} in parallel on a topology derived by the application of the partitioning algorithm. From the above discussion, \mathbf{M}^s is composed of a set of distinct software modules: $\mathbf{M}^s = \{M_F^s, M_{IO}^s, M_R^s, M_C^s, M_{CM}^s\}$.

These five software modules are necessary components, since the removal of any one module makes the implementation infeasible. This is evident from the following discussion. By Lemma 1, G^p is the minimum graph required and since the above construction of \mathbf{M}^s is based on G^p , \mathbf{M}^s represents the minimum program structure required to implement \mathcal{A} . Fur-

thermore, the module abstractions correspond to the associated components of \mathcal{A} as follows: M_F^s when fully instantiated across all processors implements \mathcal{A} , M_R^s when fully instantiated across all processors implements external communication, M_{IO}^s when instantiated provides for both the input/output of the data to and from the parallel implementation as well as input/output of the data to and from one or more of the instantiations of M_F^s , M_{CM}^s provides the definition of communication data types and M_C^s provides for the instantiation of the other modules and their corresponding allocation to processors.

Our discussion has focused on establishing a *required basis set* of software modules necessary to implement \mathcal{A} . Variations in the software design may be accommodated by corresponding variations in the definition of \mathbf{M}^s .

In the following discussion, we define three relationships *IS-COMPONENT-OF*, *USES* and *INSTANTIATES* to refer to the types of module interconnections in \mathbf{M}^s . The first two are well known relationships in software design, for example, our use is similar to the *include* and *use* relationships in HOOD [2] (refer to [21] for a detailed discussion of HOOD). We define *INSTANTIATES* as the relation whose domain and range is \mathbf{M}^s such that M_1 *INSTANTIATES* M_2 means that M_1 must execute so that a copy of M_2 is instantiated. Instantiated modules exist on a particular processor with required communication and channel variables given through the action of the instantiation process. Implicit in this discussion is the fact that one or more modules of \mathbf{M}^s are generic modules.

It is implied in our previous discussions that M_{CM}^s is incorporated into the other four (i.e. M_{CM}^s *IS-COMPONENT-OF* modules M_F^s, M_R^s, M_{IO}^s and M_C^s). Also, the following module relationships hold: M_F^s *USES* M_R^s , M_R^s *USES* M_R^s (this case may exist after M_R^s is instantiated and corresponds to the situation that a transmitter located on one processor outputs data to a receiver located on a different processor), M_R^s *USES* M_F^s , M_{IO}^s *USES* M_R^s and, M_R^s *USES* M_{IO}^s . M_C^s has interesting properties compared to the other modules: M_C^s is *executed* once prior to any of the other modules. Thus, M_C^s *INSTANTIATES* modules M_F^s, M_R^s and, M_{IO}^s . The number of instantiations of M_F^s and M_R^s are based on the number

of processors in the network. Thus network descriptions such as that provided by Lemmas 2 and 3 can be used to establish this parameter. In the particular case of Lemmas 2 and 3, the number of processors required can be obtained by a simple calculation. There is only one instantiation of M_{IO}^s which may be on a processor not explicitly defined as a part of the parallel computer, that is, for example, not defined by Lemmas 2 and 3. Figure 9 illustrates the relationships between these modules (further details of the figure are discussed below).

The previous discussion has focused on the definition of M^s ; the differences due to each partitioning strategy are now discussed. The effect of the partitioning strategy is principally realized in the internal definition of each module, and consequently to some extent, also in the modules' interfaces. This also impacts upon the precise definition of the previously discussed module relationships.

Since the primary purpose of M_F^s is to perform the computation, we consider the following three procedural objects in its interface: **Compute**, **Accept-Work-Packet** and **Produce-Work-Packet**. The latter two provide the functions required to input and output the data to and from the computation process. Essentially, these two procedures consist of reformatting the data to match the data formats used by M_R^s and the **Compute** process. In some cases, these two may be included into **Compute**. The **Compute** process reflects computations in F or F^* ; its details also depend upon ζ . These procedures are shown abbreviated in Figure 9.

There are two procedural objects in M_R^s corresponding to the transmitting and receiving functions of a communications router. These procedures are named **Transmitter** and **Receiver** (although their functionality may not be strictly restricted to transmission and receiving only). The internal functions of these procedures strongly depend on ζ . For example, with regards to ζ_t , it is defined that the transmitter instantiated on $v_i \in V^p$ can send information to the *transmitter* instantiated on $v_{i+1} \in V^p$ or to the **Compute** process instantiated on $v_i \in V^p$ (for the farming implementation, the transmitter provides the data to the worker processes while the receiver returns the results back to M_{IO}^s). However, it may receive data from only the *transmitter* instantiated on $v_{i-1} \in V^p$. The receiver for

this partitioning strategy is defined similarly, however, for communication in the opposite direction. The functions with respect to ζ_d are different. In this case, the transmitter instantiated on $v_i \in V^p$ receives its input from one or more **Compute** processes instantiated on v_i and transmits the communication to a receiver instantiated on v_{i+1} which, in turn, passes the communication onto the correct **Compute** process on v_{i+1} . These procedures are shown abbreviated in Figure 9.

There are four functions incorporated into M_{IO}^s : **Secondary-Storage** provides for the input/output to the user or host and is, to a large extent, invariant of a partitioning strategy. The **Control-Work-Packet** procedure controls the nature of the input/output to the computing worker processes while **Work-In** and **Work-Out** perform the actual input and output of the data. The implementation details of the second procedure are dependent on the partitioning strategy, for example, in ζ_t , usually, the number of outgoing data packets is bounded by the number of free worker processes (processors). The latter two procedures are connected to, usually, the procedures in M_R^s . These procedures are likewise shown abbreviated in Figure 9.

We note that there are implementation details which are not strictly dependent upon the partitioning strategy as well as the input algorithm. Continuing the above example regarding the farming implementation arising due to ζ_t , the number of outgoing data packets is also dependent on corresponding implementational details of, mainly, M_R^s , where multiple buffering and guaranteed intermediate data store versus non-guaranteed data store are two specific issues of implementation. Consequently, there may exist additional *external* specification of M^s which specify the nature of buffered communications (single, double or triple buffering) in the farming implementation. This additional specification is a programming implementation issue and we do not consider it further. We denote by \overline{M}^s the definition of M^s resulting from the partitioning strategy.

From the above discussion the following lemma follows.

Lemma 5 *Given s and ζ , $\zeta(s) \mapsto \overline{M}^s$.*

We have manually coded implementations derived from both the temporal and diagonal partitioning strategies in the occam 2 language. The coded implementations were implemented as part of a set of experimental investigations into aspects pertaining to a model for a compiler for multicomputers [6]; they are included here as specific examples of the modules discussed herein. Figures 10 and 11 pertain to the temporal partitioning strategy for matrix multiplication while Figure 12 pertains to diagonal partitioning for the Weighted Levenshtein Distance computation (WLD) [9].

Figure 10 shows the occam 2 pseudocode for M_{IO}^s ; the **Secondary-Storage** procedure is shown as the comment `... User input/output` near the top of the figure (a folding editor has been used, consequently, this is a fold comment; the fold contains the program code).

The **Control-Work-Packet** procedure is shown in its two parts in Figure 10. Step 1 is a folded comment wherein the initial set of computations is farmed out to all of the available worker processes and Step 2 is a folded comment wherein the **Control-Work-Packet** procedure waits until one of the worker processes becomes available due to the completion of a prior scheduled computation. The **Work-Out** and **Work-In** procedures are not explicitly shown in Figure 10. However, the calls to these functions are contained in the I/O Transmitter and I/O Receiver sections of the code, respectively. Clearly, there is a very high coupling between the latter three procedures; consequently, it may be convenient to disregard this module definition and consider an alternative, for example, one which merges these three procedures together.

Figure 11 shows the occam 2 code for the **compute** procedure of M_F^s . Note that the **Accept-Work-Packet** and **Produce-Work-Packet** are embedded into the **compute** procedure. Moreover, M_{CM}^s is incorporated by the **#INCLUDE** statement at the top of the figure (i.e. the file `farming.inc` represents M_{CM}^s in this case). Figure 12 also shows the occam 2 code for the **compute** procedure (the computation section has been commented out); note the similarity in the structure of the procedure between both implementations of the **compute** procedure. However, in the latter example, the investigation was also concerned with representing the

computations at a finer granularity level, that is, the computations associated with *each* vertex in \hat{G} were represented as a definition of the computation. Figure 12 consequently also illustrates that variations in the definition of the computations to be performed may be accommodated by the derived program structures described herein. Further details of these implementations may be found in [6, 9].

5 Conclusion

We have shown that the partitioning and mapping process of a systolic computation graph induces an associated set of program structures in a high level language representation for a parallel implementation of the input algorithm. We have established this in two parts: firstly, we showed that the partitioning and mapping process induces a specific processor topology associated with each partitioning strategy (e.g. Lemmas 2, 3 and 4) and secondly, we showed that the program processing requirements based on the topology imply a specific program structure. We have provided details of the resulting program structure and have shown that certain details of the resulting program structures are induced by the partitioning strategy (i.e. Lemma 5).

We have also shown that the program structures derived by our method constitute a basis parallel implementation, and certain implementation specific information may be later added to the derived program structures. In [6], we have considered the role of code templates in this context.

The formalism contained in this paper assists in the construction of automated tools to support the generation of program code according to the proposed methodology. Details of a preliminary tool to perform the partitioning and mapping process (i.e. the first part of this paper) may be found in [10]. We have applied the proposed derivation technique in context of developing a compiler for high performance computers. Further details of this application may be found in [6, 7, 8].

References

- [1] Utpal Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, MA, USA, 02061, 1997.
- [2] J. Buxton and J. McDermid. HOOD (Hierarchical Object Oriented Design). In C. Richter, editor, *Software Engineering A European Perspective*, chapter 4, pages 222–225. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA, 90720-1 264, 1993.
- [3] K.M. Chandy. Concurrent program archetypes. In *Proc. of the 1994 Scalable Parallel Libraries Conference*, pages 1–9, Mississippi State, Mississippi, USA, October 1994. IEEE Comput. Soc. Press, Los Alamitos, CA, USA.
- [4] R.S. Cok. *Parallel Programs for the Transputer*. Prentice Hall, Englewood Cliffs, New Jersey, 07632, 1991.
- [5] A. Darte, T. Risset, and Y. Robert. Synthesizing systolic arrays: Some recent developments. In *Proceedings of the International Conference on Application Specific Array Processors (Cat. No.91TH0382-2)*, pages 372–386, Barcelona, Spain, Sept. 1991. Los Alamitos, CA, USA: IEEE Compute. Soc. Press 1991.
- [6] Brian J. d’Auriol. *A Unified Model for Compiling Systolic Computations for Distributed Memory Multicomputers*. PhD thesis, Faculty of Computer Science, University of New Brunswick, Fredericton, N.B. Canada, June 1995.
- [7] Brian J. d’Auriol and Virendra C. Bhavsar. Generic program representation and evaluation of systolic computations on multicomputers. H.R. Arabnia, editor, In *Proc. of the*

- International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96)*, Vol. III, pages 1213–1224, Sunnyvale, California, USA, August 1996.
- [8] Brian J. d'Auriol and Virendra C. Bhavsar. Multicomputer implementations of systolic computations: A unified approach. In *Proc. of the 10th Annual International Conference on High Performance Computers (HPCS'96)*, Ottawa, Ontario, Canada, June 1996. Published on CD-ROM by IEEE Canada Electronic Services, distributed by Carleton University Press.
- [9] Brian J. d'Auriol, Virendra C. Bhavsar, and L. Goldfarb. Systolic array implementations for reconfigurable learning machines on transputers. V.C. Bhavsar and U.G. Gujar, editors, In *Proc. of Supercomputing Symposium '91*, pages 105–119, Fredericton, N.B., Canada, June 1991. University of New Brunswick Press, Fredericton, N.B.
- [10] Brian J. d'Auriol and Meera B. Dugar. A systolic array graph partitioning system. K. Li, T.S. Abdelrahman, and E. Luque, editors, In *Proc. of the Eighth IASTD International Conference on Parallel and Distributed Computing and Systems (PDCS'96)*, pages 356–358, Chicago, Illinois, USA, October 1996. IASTED/ACTA Press.
- [11] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 07632, 1991.
- [12] Inmos Limited, Prentice-Hall Ltd., Hertfordshire, UK. *Occam 2 Reference Manual*, inmos document no. 72 occ 45 01 edition, 1988.
- [13] Inmos Limited, Prentice-Hall Ltd., Hertfordshire, UK. *Transputer Reference Manual*, isbn 0-13-929001-x edition, 1988.
- [14] S.Y. Kung, K.S. Arun, R.J. Gal-ezer, and D.V. Bhaskar Rao. Wavefront array processor: Language, architecture, and applications. *IEEE Transactions on Computers*, 31(11):1054–1066, November 1982.

- [15] C.E. Leiserson and F. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.
- [16] N. Ling and M.A. Bayoumi. Systolic temporal arithmetic: A new formalism for specification and verification of systolic arrays. *IEEE Transactions on Computer-Aided Design*, 9(8):804–820, August 1990.
- [17] G.M. Megson. *An Introduction to Systolic Algorithm Design*. Oxford University Press, New York, USA, 1992.
- [18] R.G. Melhem and W.C. Rheinbold. A mathematical model for the verification of systolic networks. *SIAM Journal on Computing*, 13(3):541–565, August 1984.
- [19] W.L. Miranker. Spacetime representations of computational structures. *Computing*, pages 93–114, 1984.
- [20] D.I. Moldovan. *Parallel Processing, From Applications to Systems*. Morgan Kaufmann Publishers Inc., 2929 Campus Drive, Suite 260, San Mateo, CA, USA, 94403, 1993.
- [21] P.J. Robinson. *Hierarchical Object-Oriented Design*. Prentice-Hall, Englewood Cliffs NJ, 1992.
- [22] Ian Sommerville. *Software Engineering, Fifth Ed*. Addison-Wesley Publishing Company, 1996.
- [23] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 390 Bridge Parkway, Redwood City, CA, 94065, 1996.

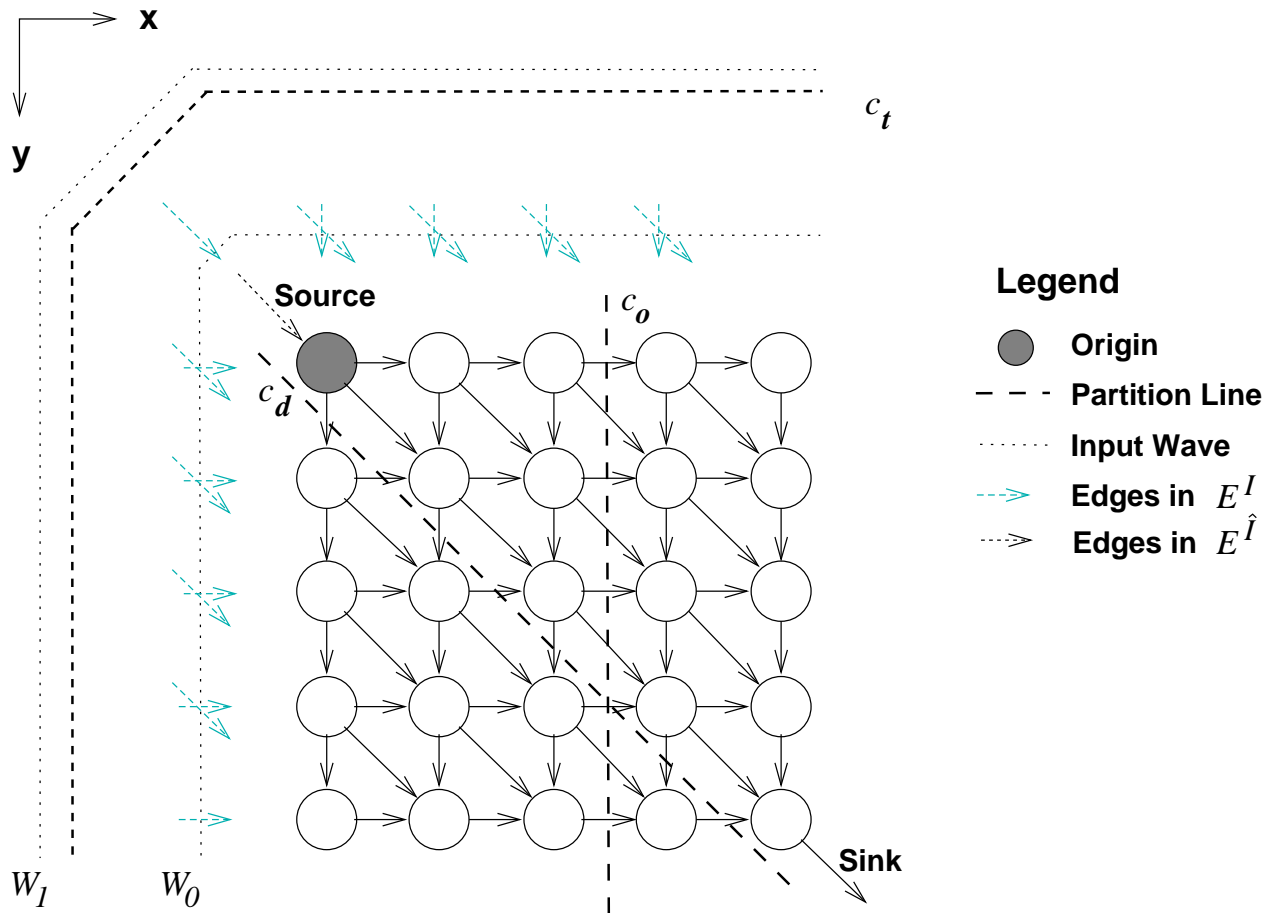


Figure 1: Structure of \hat{G} for a systolic array of a 5×5 array size and examples of three characteristic partitions on \hat{G} .

Choose $\mathbf{C} \in \zeta$.

Let G^p be an empty graph initially.

For all edges e in \hat{G}' do

BEGIN

There are two possibilities: either e intersects with a $c \in \mathbf{C}$ or it does not.

1. If e intersects with c then

BEGIN

(a) if $\sigma_-(e)$ is not already part of the graph of a supernode, then

$$V^p \leftarrow V^p \cup \sigma_-(e),$$

(b) if $\sigma_+(e)$ is not already part of the graph of a supernode, then

$$V^p \leftarrow V^p \cup \sigma_+(e),$$

(c) add e to the superedge $e^p \in E^p$ such that $\sigma_-(e^p)$ contains $\sigma_-(e)$ and $\sigma_+(e^p)$ contains $\sigma_+(e)$.

END

2. Otherwise

BEGIN

(a) if $\sigma_-(e)$ is in the graph of a supernode and $\sigma_+(e)$ is not in the same graph, then add $\sigma_+(e)$ to the graph of a supernode containing $\sigma_-(e)$,

(b) if $\sigma_+(e)$ is in the graph of a supernode and $\sigma_-(e)$ is not in the same graph, then add $\sigma_-(e)$ to the graph of the supernode containing $\sigma_+(e)$,

(c) if neither $\sigma_-(e)$ nor $\sigma_+(e)$ is in any graph of a supernode in V^p , then add them both to the same supernode,

(d) if $\sigma_-(e)$ and $\sigma_+(e)$ are contained in two different vertices in V^p , then merge the subgraphs of both vertices in V^p together along with all associated edges of those vertices,

(e) add e to the supernode containing $\sigma_-(e)$.

END

END

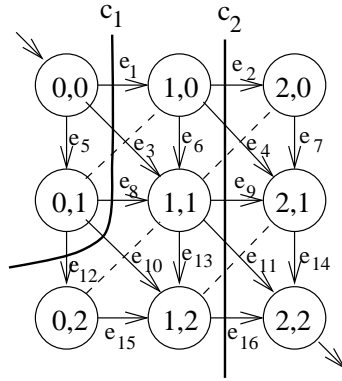
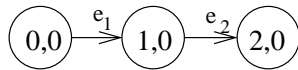
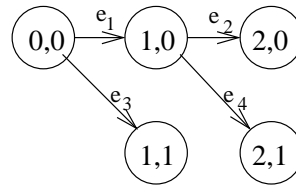


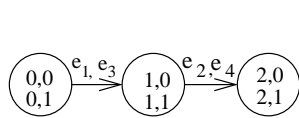
Figure 3: An example for the partitioning algorithm.



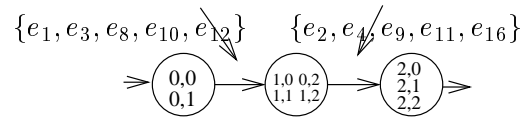
(a)



(b)



(c)



(d)

Figure 4: The graphs resulting from the application of the partitioning algorithm to the example.

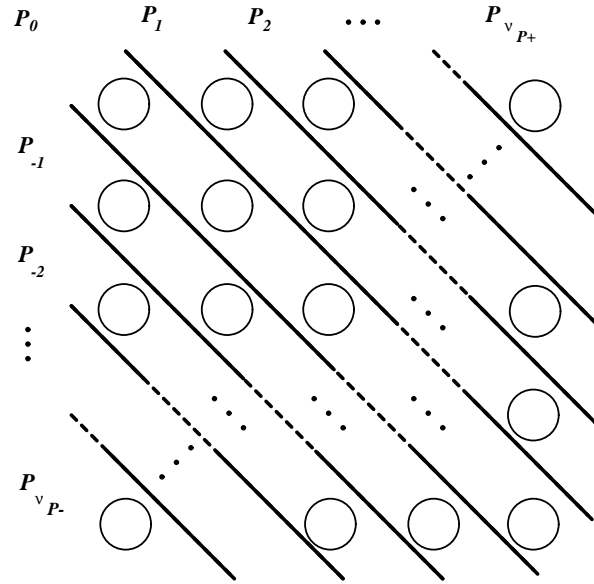


Figure 5: An example of diagonal partitioning (ζ_d).

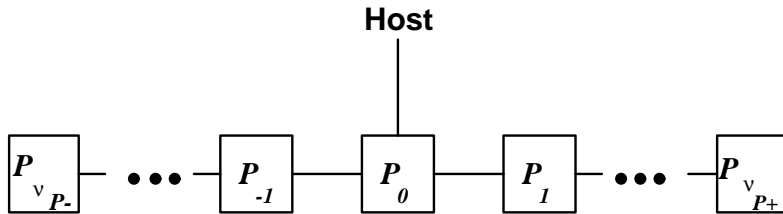


Figure 6: Bidirectional linear network resulting from applying ζ_d on \hat{G} .

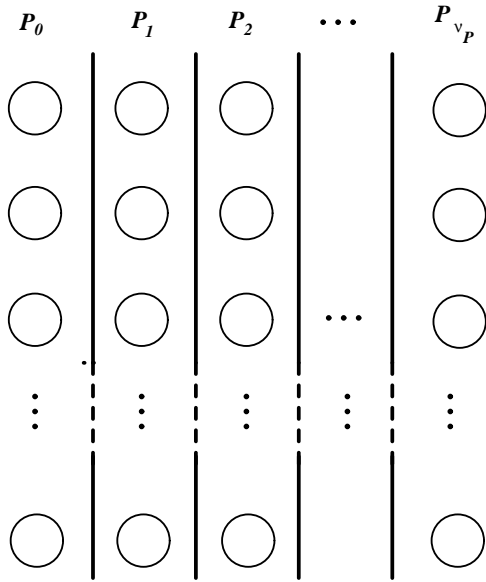


Figure 7: An example of orthogonal partitioning (ζ_o) for partition lines of the form $x = b$.

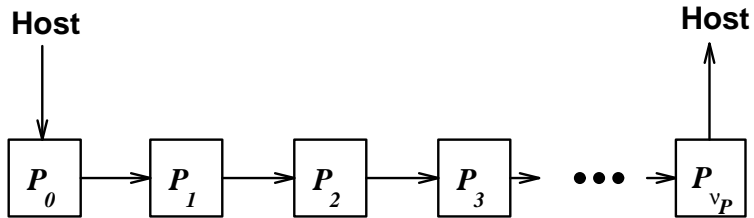


Figure 8: Linear network resulting from applying ζ_o on \hat{G} .

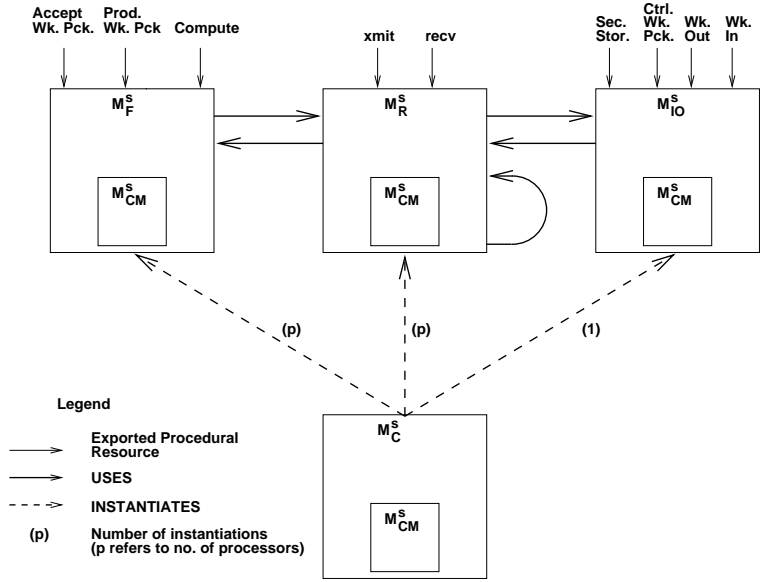


Figure 9: A derived program structure.

```

SEQ
... Local files and declarations used.
... User input/output
... Transpose matrix B. -- required by the input/output controls below.
PAR
-- I/O Transmitter
SEQ
... Step 1: Output work packets (one per available worker processors).
... Step 2: Wait for information from I/O Receiver; output a new work packet.
-- Receiver
SEQ
... Wait for a result
... Inform transmitter that a processor is available
:

```

Figure 10: occam 2 pseudocode showing a particular coding of M_{IO}^s for matrix multiplication with a farming implementation.

```

#include "farming.inc"

PROC compute(VAL INT processor.address,
             CHAN OF cell.com computation.load,
             computation.unload)

REAL32 c;
INT matrix.size;
[max.matrix.size]REAL32 V1;
[max.matrix.size]REAL32 V2;
INT i,j;

WHILE TRUE
  SEQ
    computation.load ? CASE
      computation.input; i;
        j;
        matrix.size::[V1 FROM 0 FOR matrix.size];
        matrix.size::[V2 FROM 0 FOR matrix.size]
      SEQ
        c := 0.0 (REAL32)
        SEQ i = 0 FOR matrix.size
          c := c + (V1[i] * V2[i])
        computation.unload ! computation.output;
          i;
          j;
          c
    :

```

Figure 11: occam 2 source of M_F^s for matrix multiplication with a farming implementation.

```

PROC computation(VAL INT address,
                CHAN OF cell.com.go in1, in2, in3, out1, out2, out3)

... Declarations specific to the systolic algorithm.

SEQ
  terminated := FALSE
  WHILE NOT terminated
    SEQ
      PAR
        in1 ? char1; cost1; prev.cell.value1
        in2 ? char2; cost2; prev.cell.value2
        in3 ? char3; cost3; prev.cell.value3

      ... Do cell computation

      PAR
        out1 ! char1; cost1; cell.value
        out2 ! null; 0.0 (REAL32); cell.value
        out3 ! char3; cost3; cell.value
    :

```

Figure 12: occam 2 source of M_F^s for the diagonal partitioning implementation of the WLD.