Conference Name: The 10th Annual International Conference on High Performance Computers (HPCS '96), Ottawa, June 1996

Paper Title: Multicomputer Implementations of Systolic Computations: A Unified Approach

Principal author: Brian J. d'Auriol
Department of Computer Science and Engineering
Wright State University
Dayton, Ohio
USA, 45435
Phone: (513)-873-5131, Fax: (513)-873-5133,
Email: bdauriol@valhalla.cs.wright.edu

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

Date: February 12, 1996

List of Figures

1 Unified process relationships. 5

2 Structure of $\hat{G}$ for a systolic array of 5×5 array size and examples of four characteristic partitions on $\hat{G}$. 12

3 Partitioning algorithm. 15

4 Example of Theorem 1's constructive proof. 16

5 The processor graph construction. 17

6 An example of diagonal partitioning (ζ_d); P_i denotes a processor to which the corresponding interpartition region is allocated to (see Figure 7). 21

7 Bidirectional linear processor network resulting from applying ζ_d on $\hat{G}$. 21

8 An example of orthogonal partitioning (ζ_o) for partition lines of the form $x = b$. 22

9 Linear processor network resulting from applying ζ_o on $\hat{G}$. 22

Multicomputer Implementations of Systolic Computations: A Unified Approach

Brian J. d'Auriol*

Department of Computer Science and Engineering,

Wright State University,

Dayton, Ohio,

USA, 45435

Virendrakumar C. Bhavsar †

Faculty of Computer Science,

University of New Brunswick,

Fredericton, New Brunswick,

Canada, E3B 5A3

Abstract

We propose a unified approach for implementing systolic computations on distributed memory multicomputers (DMMCs). Advantages of this approach include its applicability to a wide range of systolic algorithms, DMMCs and computer languages. It also incorporates the evaluation of expected execution times for different possible implementations of a given systolic computation and the generation of source code for a selected implementation. We present the details of the various stages of the unified approach including the generation of a systolic array, several partitioning schemes and related theorems, source code generation, execution time analysis and optimization based on performance models. The intended use of the approach is for building a compiler that would accept a systolic algorithm specification as input and would generate an efficient executable object program.

*Supported by NSERC doctoral fellowship.

†Partially supported by NSERC grant, OGP0089.

1 INTRODUCTION

Since its introduction in 1978 by Kung and Leiserson, systolic processing has been and continues to be an important research area. A significant reason for this research interest is the potential to exploit the intrinsic parallelism in computations lending themselves to systolic processing solutions. Another important reason is that there are many application areas of systolic processing including digital signal processing, matrix computations, symbolic processing, sorting and relational databases.

Implementations of systolic processing have typically been either in special purpose VLSI or on special purpose computers. Special purpose VLSI implementations suffer from limited programmability, higher cost, high input/output requirements and clock skew (see for example [1] and [2, Chapter 1]). Special purpose systolic computers, like the Warp, have addressed some of these concerns. However, these computers have higher costs and limited applications compared to general purpose computers.

Nowadays, distributed memory multicomputers (DMMCs) are readily available, general purpose scalable computers. Examples of DMMCs include IBM SP2 and multi-transputer systems [3]; the latter, in particular, offer low cost solutions. Software implementations of systolic computations on DMMCs are attractive due to their generality, portability, lower costs and flexibility.

The implementation of systolic computations on DMMCs is often difficult. The main reason is that systolic computations have fine grain parallelism whereas DMMCs are well suited for medium and coarse grain parallelism. This obvious mismatch in the granularity necessitates efficient partitioning and mapping of systolic computations onto DMMCs. Such partitioning may be difficult since more than one distinct implementation may be possible and the choice of the implementation may require evaluation of many different factors. In addition, program development on DMMCs is usually time consuming, error prone and may be limited to a specific architecture. Consequently, trained or experienced parallel programmers are often needed

thus increasing the cost of the development.

Megson in [4] gives a discussion of relative strengths of various multi-transputer implementations of systolic programs for the model reduction problem in control theory while Rice and Seidman in [5] present a multiple transputer implementation of a model for generalized systolic computation. Lengauer et al. in [6] proposes a two step compilation scheme for systolic algorithms while Megson and Comish in [7] presents a systolic algorithm design environment within which a systolic compilation scheme is discussed. The SDEF system [8] was designed to assist with systolic research, that is, to provide a tool which "... can be used to *express the results* of the analysis, synthesis and optimization performed by other tools".

None of these approaches generate different possible implementations which can then be compared in order to determine an efficient final implementation. Further, in the above work there is a lack of a general framework for the implementation of systolic computations on general purpose DMMCs. Lastly, efficiency issues of the implementations are not addressed in detail. In [9], we have outlined a general strategy that addresses these issues.

In this paper, we propose a unified approach for the implementation of systolic computations on distributed memory multicomputers. The intended use of the unified approach is to build a compiler which would accept a systolic algorithm specification as input and would generate an efficient executable object program.

This paper is organized as follows. The objectives of the proposed unified approach, some definitions and an overview of the stages of the unified approach are given in the following section. These stages as well as the necessary components required for the execution of these stages are detailed in Section 3. Finally, concluding remarks are given in Section 4.

2 THE UNIFIED APPROACH

2.1 OBJECTIVES

The main objectives of the proposed unified approach for the implementation of systolic computations on DMMCs are: (a) the applicability to *arbitrary* systolic algorithms, (b) the applicability to *general* purpose DMMCs, (c) the incorporation of execution time optimization, (d) the inclusion of necessary performance models, (e) the provision for a framework in which the techniques for the derivation of systolic arrays and for source code generation are included, and (f) the provision for as much automation as possible when fully implemented.

2.2 DEFINITIONS

In this section, we present a few definitions which are used in the rest of the paper.

Megson [2] defines the term *recurrence relation* as a mathematical equation which can be expressed in the following general form $f(p) = g(f_1(q_1), \ldots, f_k(q_k))$, where there are $k > 0$ terms of function g representing the $k > 0$ occurrences of the recurrence variable represented by the function f. The index space p and $q_1, q_2, \ldots, q_k$ are vectors which represent the dimensionality of the recurrence (with q being the set of dependencies on p). Additionally, a *uniform recurrence relation* is a uniform recurrence equation if and only if $q_i = p - w_i$ such that all w_i are constant vectors for $1 \leq i \leq k$.

Definition 1: A *systolic algorithm* is a set of uniform recurrence relations combined with the recurrence relation inputs (bases of the recurrence relation) and the lower and upper limits of each dimension in p and q.

Definition 2: A *systolic array* s has a defined geometry (represented as a graph) G combined with a data sequencing I (in space and time) and a set of computational functions F mapped to each vertex in G: $s = (G, I, F)$.

In this paper, *systolic design methodology* refers to methods and procedures used to transform a systolic algorithm into a systolic array. A *wavefront array* is a systolic

array with the additional characteristic that the data sequencing (I) is such that a wavefront is obtained in the progression of the computations in s. A *systolic implementation* is any valid computer program (coded in some language and executable on a particular target machine) which has the same outputs as the given systolic array.

2.3 THE APPROACH

We have identified six stages necessary to achieve the objectives of the unified approach. These stages are shown in Figure 1 and are described below:

1. *Systolic Array Generation*: A systolic array conforming to both Definition 2 and to a set of restrictions (see Section 3.1) is generated by one of the several existing methods of systolic array generation.

2. *Systolic Array Partitioning and Mapping to Architecture*: The partitioning and subsequent mappings of the systolic array in general lead to different source code representations (including the sequential program). Differences may include possible variations in the communication or computation program structures. Although partitioning of systolic arrays is not new, our approach is different from previous work in that we consider the partitions as cuts of G and I (refer to Definition 2) where G is embedded into the Cartesian coordinate system.

3. *Source Code Representation*: In order to meet the objectives, a representation of the partitioned systolic array in a specific machine-dependent environment is required. A machine-dependent environment includes the language (e.g. any source code language, normally a high level language), configuration language and machine characteristics. The execution of this stage combines the partitioned systolic array with the machine-dependent environment thus representing the complete source code required for the implementation on the particular DMMC.

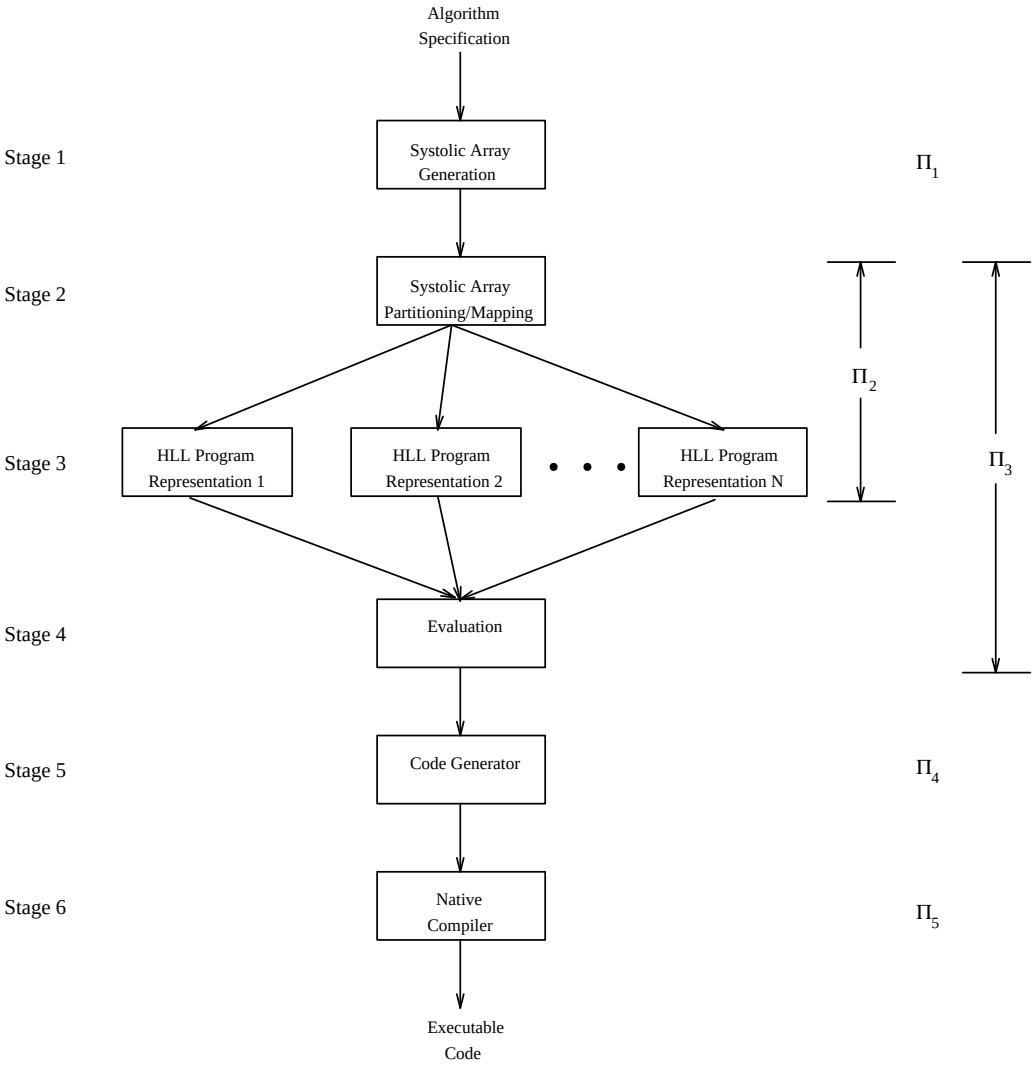

Figure 1: Unified process relationships.

4. *Evaluation*: This stage evaluates the expected execution time of each of the possible source code program representations and determines the optimal one. The following tasks are required for this purpose:

   (a) Performance modeling of the object (executable) program which would be generated from a specific source code representation. Note that since we are interested in generating an executable program, our performance modeling is based on the estimation of the expected execution time of the

object program.

(b) Analysis of both the computation and communication time expected for the source code representation.

(c) Dependency analysis of the processor mappings of the source code representation.

5. *Code Generation*: This stage generates the actual source code corresponding to the source code representation (from stage 3) that has been selected by the Evaluation stage.

6. *Compilation of the Source Code*: A native compiler on the target machine is employed to generate the object code.

Also shown to the right in Figure 1 are the five tasks (indicated by Π_i) which are responsible for executing the stages in the unified approach.

Given the specification for a systolic algorithm as well as a number of required parameters (e.g. number of processors available, computational and communicational aspects of the target DMMC), application of the stages of the unified approach would result in an object program for that algorithm specification. This object program would have been selected from many candidate implementations and consequently, represents an efficient program.

The algorithm specification used as input to Stage 1 should conform to Definition 1 and be in as simple (from the point of view of the programmer) a form as is possible. In this way, the user need not be concerned with identifying specific features of systolic processing, but rather, can concentrate on specifying the required tasks. Clearly, a specification language will need to be adopted.

For the purposes of this work, we assume that some algorithmic specification for the input is available since almost any language can be used in this manner. Stage 1, the generation of a systolic array, is a known process and details can be

found in [2, 7]. We place some restrictions on the systolic array that is derived (see Section 3.1); however, we allow any systolic design methodology to be used. We develop a notation based on the unified approach in which to represent systolic arrays. The principal advantage of our formalism is to concisely describe the partitioning process. There are several other formal models (see for example [7, 10, 11] and the references therein) which could possibly be extended so as to be used within our unified approach. Stage 5, Code Generation, is assumed to exist and consequently, the details of this stage are not addressed in this paper. As well, the last stage (the Compilation stage) assumes the existence of language compilers for the target machine. Since this is reasonable, we also do not include a detailed discussion of this stage in this paper. In this context, in this paper, we restrict the discussion to Stages 1-4 of the unified approach.

3 STAGES OF THE UNIFIED APPROACH

In this section, we detail Stages 1-4 of the unified approach presented in the previous section. Each of the four stages are discussed in the following order: (a) first new notations are introduced, (b) then, the execution of the stage is discussed, and (c) finally, any necessary components required for the execution of stage are discussed.

3.1 STAGE 1: SYSTOLIC ARRAY GENERATION

Since this stage consists of a known procedure, we very briefly discuss the execution of this stage; however, as mentioned in the previous section, we place restrictions on the systolic array that is derived.

Notation

Following the usual notation of representing the geometry of a systolic array as a directed graph (see for example the formal notation used in [10]), 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 input to Stage 1 is a given systolic algorithm conforming to Definition 1 and is denoted by $\mathcal{A}$. The method used in Stage 1 for systolic array generation is denoted by Π_1 and the set of all possible systolic arrays that can be generated by Π_1 is represented by $\mathbf{S} = \{s_0, s_1, \ldots, s_{\nu_\mathbf{S}}\}$. Each $s_i \in \mathbf{S}$ will have an associated systolic geometry denoted as G_{s_i}.

Execution

As mentioned in Section 2, there exist several systolic array design methodologies; the *Projection Method* [2] is seemingly the more popular. The result of the application of a systolic array design method is a set of systolic array solutions for the given systolic algorithm. For example, in the Projection Method, different systolic arrays result from different pairs of timing and allocation functions (see the discussion in [2]).

We compactly represent the systolic array generation process by the following:

$$\Pi_1(\mathcal{A}) \quad \mapsto \quad \mathbf{S} \tag{1}$$

Selection of a Systolic Array

From the preceding discussion, it is clear that there may be many possible systolic array solutions for a given systolic algorithm. We are interested in selecting a single systolic array solution; this would consequently provide for an invariant geometry, data sequencing and functions during the application of Stage 2 of the unified approach consisting of namely, the Partitioning and Mapping process.

From Definition 2, a systolic array has three principal parts: $s = (G, I, F)$. In this section, we are concerned with the geometry (G) and data sequencing (I) , but not

with the functions (F) of the systolic array. Our choice for the geometry of a systolic array is motivated by the following considerations: (a) its ability to accommodate many common systolic array geometries, (b) simple coding requirements, and (c) effective speedup in the computation.

We impose the following restrictions on G and denote the resulting graph as $\hat{G}$:

1. A two-dimensional mesh with two diagonals [12] consisting of interior nodes of degree six; this has been referred to as a hexagonal topology [2, page 16] and also as a *hexagonally mesh-connected* processor array [13, page 273]. This has been widely used for matrix multiplication [13, page 277]. For simplicity, we will henceforth refer to a graph which has this structure as a *hexagonal graph*. This graph has finite but arbitrary dimensions.

2. The set of directed arcs that are allowable from the first restriction are further restricted as follows. Asynchronous communication occurs when the directed arcs are considered as the communication pathways in the systolic array, resulting into a wavefront array.

3. There is a single source and a single sink for data.

As indicated in [14], the hexagonal graph is a super-graph of many of the more common systolic array geometries found in systolic computations. A sub-graph of $\hat{G}$, the square mesh, has been used in a similar context in [5]. A wavefront array both simplifies the coding requirements (for example, by eliminating global synchronization) and provides a mechanism to allow speedup. For example, by careful allocation of systolic cells to processors of the DMMC based upon the property of the wavefront progression, namely, that computation aligned along the wavefront can be carried out simultaneously, one can exploit intrinsic parallelism. A single source and sink requirements also simplify the coding requirements by imposing a single input and a single output point in the corresponding source code program representing the implementation. The single source-sink restriction of $\hat{G}$ poses little or no restrictions on its

use in the context of the unified approach, since a direct application of the retiming lemmas of Leiserson and Saxe as discussed by Megson [2, pp. 49-54], can perform the transformation of a systolic array $s_1 = (G, I, F)$ with many inputs and outputs along a boundary to a systolic array $s_2 = (\hat{G}, \hat{I}, \hat{F})$.

To summarize, the following restriction is placed on Π_1:

$$\Pi_1(\mathcal{A}) \quad \mapsto \quad s \in \mathbf{S} \tag{2}$$

where $s = (\hat{G}, I, F)$.

3.2 STAGE 2: PARTITIONING THE SELECTED SYSTOLIC ARRAY

Having now derived a systolic array $s = (\hat{G}, I, F)$ for the given systolic algorithm, we now consider the partitioning problem. Since there is a mismatch in the granularity of the systolic array and a DMMC, the systolic array needs to be partitioned into blocks of computations so that when these blocks are allocated to physical processors in the DMMC, the granularity mismatch is reduced or eliminated.

There are two principal forms of partitioning: locally parallel globally sequential (LPGS) and locally sequential globally parallel (LSGP) [15]. The former partitions an array into blocks which are then computed sequentially in time while the latter partitions an array into blocks which are then allocated to a processor. We discuss several partitioning schemes in this section which are either LPGS or LSGP.

Notation

We embed $\hat{G}$ into the Cartesian plane such that the origin coincides with the vertex representing the first computational node (i.e. the source) in $\hat{G}$ with all vertices in $\hat{G}$ lying in the positive x, y quadrant. A vertex in $\hat{G}$ is identified by $v_{i,j} \in V$ where i, j are positive integers. For all vertices in $\hat{G}$, the distance between $v_{i,j}$ and $v_{i+1,j}$ is unit distance and the directed edge, e, with $\sigma_-(e) = v_{i,j}; \sigma_+(e) = v_{i+1,j}$, has

associated with it a direction vector $(1,0)$. Similarly, the distance between $v_{i,j}$ and $v_{i,j+1}$ is unit distance, and for the edge e, $\sigma_-(e) = v_{i,j}$; $\sigma_+(e) = v_{i,j+1}$, has associated with it a direction vector $(0,1)$. The distance between $v_{i,j}$ and $v_{i+1,j+1}$ is $\sqrt{2}$. The edge e, $\sigma_-(e) = v_{i,j}; \sigma_+(e) = v_{i+1,j+1}$, has associated with it a direction vector $(1,1)$. The dimensions of the graph are denoted by N_i and N_j in the i and j directions, respectively.

We further define the data sequencing I as follows. Let I_i^d represent the data sequencing in the d direction for the i^{th} time instant, where d is a directional vector and an element of the set { (0,1), (1,0), (1,1) }. Thus $I_0^{(1,0)}$ would be the 0^{th} data element in the positive x direction; $I_2^{(1,1)}$ would be the second data element in the positive x, y direction (i.e. diagonal), etc.

Figure 2 illustrates both the geometry of $\hat{G}$ and the notation introduced above for a 5 by 5 array size (the heavy dashed lines in Figure 2 are discussed later). The directions of the x and y 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 at the vertex located at the origin. The top row and left column, in addition to the normal systolic computation, provide functions to appropriately distribute the data inputs. The data inputs, once distributed, move according to the established data sequencing. Although the restriction of single source does not impact upon the correctness of the computations, it does have impact on the execution time.

We denote by c either a line or a simple closed 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. We group related simple curves together (where the relation is based on some set of characteristics, for example, lines having the same slope) and denote this set of simple curves by $\mathbf{C}$. We denote by ζ a set of related sets of simple curves

$$\zeta \quad = \quad \{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_{\nu_\zeta}\}, \tag{3}$$

where each $\mathbf{C}_i$ is a set of zero or more simple curves. Consequently, each individual

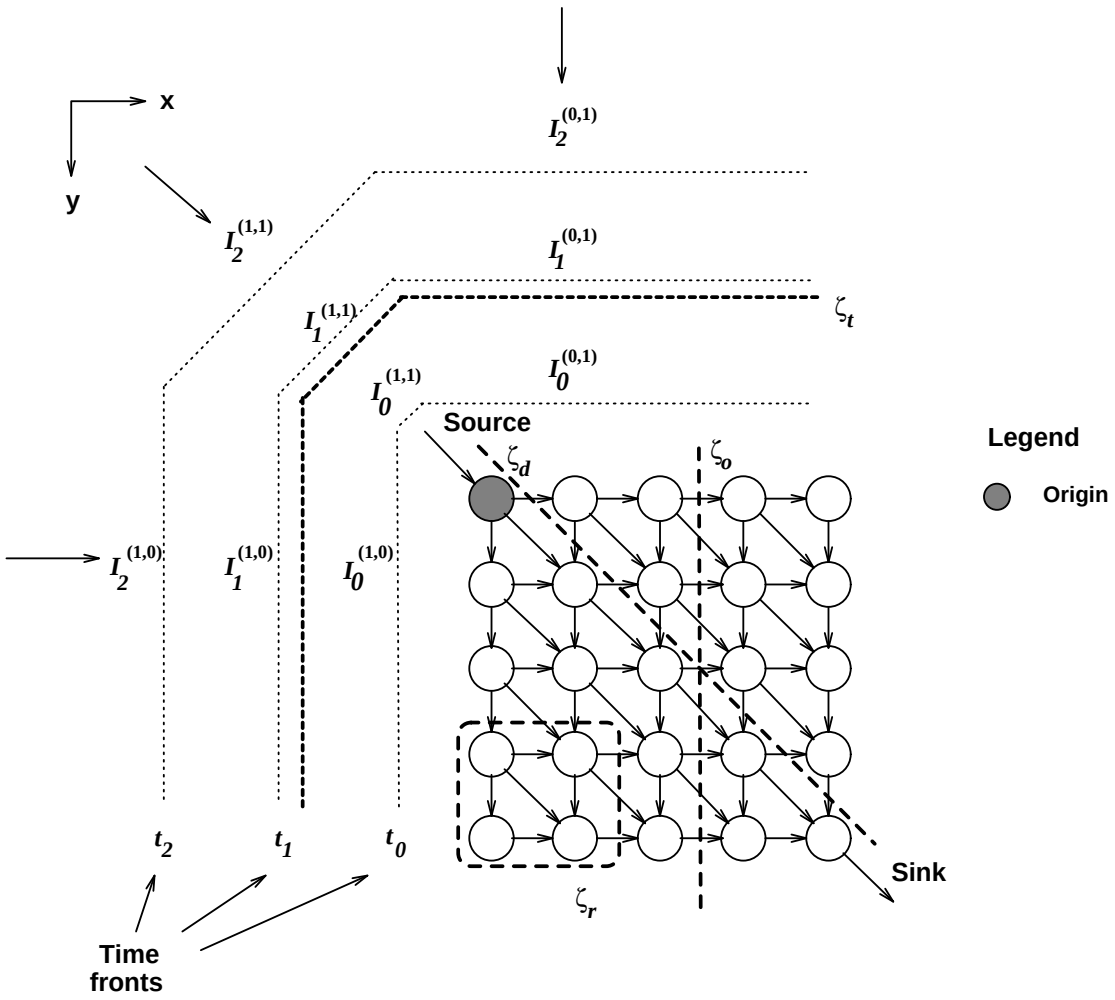

Figure 2: Structure of $\hat{G}$ for a systolic array of 5×5 array size and examples of four characteristic partitions on $\hat{G}$.

simple curve (which induces a cut on $\hat{G}$) is denoted by $c_j^i \in \mathbf{C}_i$, where $i, j > 0$. Each $\mathbf{C}_i$ defines a set of sub-graphs of $\hat{G}$. The set of vertices in such a sub-graph induced by $\mathbf{C}_i$ forms a block of a partition that is denoted by R; the number of vertices in R is denoted by $||R||$. Consequently, $\mathbf{C}_i(\hat{G}) \mapsto R$. Essentially, R represents a *region* of the systolic array, the regions having been defined by the partition curves on $\hat{G}$. Since ζ is a collection of related sets of simple curves, characteristics of the set ζ can be identified. We denote by $\mathbf{Z}$ the set of characteristic cuts on $\hat{G}$, that is, $\zeta \in \mathbf{Z}$. Lastly, we restrict all lines and simple closed curves in $\mathbf{Z}$ such that these curves do

not pass through any vertex in $\hat{G}$ (i.e. the curves do not pass through the points (i, j) for integer i, j). The type of partitioning discussed above is a *spatial* partitioning of the systolic array and leads to a LSGP type of partition.

We also consider the partitioning of the systolic array, s, along its time axis. That is, the data sequencing I consists of data elements moving in a particular direction within the systolic array at a particular time instant. We employ the notation developed above for spatial partitioning for the temporal partitioning case as well. Let c provide a cut of the linear time line. The rest of the definitions pertaining to $\mathbf{C}$ and ζ remain the same. This type of partitioning leads to a LPGS type of partition.

Examples of four sets of ζ are shown in Figure 2: *Diagonal* (ζ_d), *Orthogonal* (ζ_o), *Rectangular* (ζ_r) and *Temporal* (ζ_t). An example of a simple curve from each type of ζ is shown by a heavy dashed line in the figure. The characteristic for ζ_d is that all cuts are formed by straight lines of the form, $y = x + b$, while for ζ_o all cuts are formed by straight lines, all of the form $y = b$ or all of the form $x = b$ (combination of the two forms is disallowed since it would naturally lead to the type of partitions in ζ_r). The ζ_r consists of simple closed curves forming a rectangle while ζ_t (as discussed above) consists of cuts in the time domain.

We define an *input wave*, W, as a collection of the following data sequencing elements: $\{I_i^{(1,0)}, I_i^{(0,1)}, I_i^{(1,1)}\}$; the number of waves is denoted by $\overline{\mathrm{W}}$. In Figure 2, there are three input waves shown by the light dashed lines. 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 edges such that $\sigma_-(e \in \bar{E})$ is the vertex at position (i, j) and that $\sigma_+(e \in \bar{E})$ is the vertex at position $(i + 1, j - 1)$, where $0 \geq i \geq N_i, 0 \geq j \geq N_j$ (i.e. diagonal edges perpendicular to the set of diagonal edges in E). Additionally, an edge $e \in \bar{E}$ is defined so that it *does not intersect* any cut lines. The incorporation of $\bar{E}$ edges in the graph does not change the data sequences (I) or cell functions (F) in any way since no information is assigned to these edges.

Lastly, we define the processor topology of a target machine as a graph $G^p = (V^p, E^p, \sigma_-^p, \sigma_+^p)$. In the following subsection it is shown that G^p would consist of

supernodes and superedges constructed from the nodes and edges, respectively, of $\hat{G}$.

Execution

The execution of Stage 2 consists of applying a particular $\mathbf{C} \in \zeta$ to $s = (\hat{G}, I, F)$ so as to arrive at a particular G^p: $\mathbf{C}(\hat{G}) \mapsto G^p$. Figure 3 shows the algorithm used to determine G^p. Essentially, each R_i is mapped to a unique vertex V^p and each cutset defined by $\mathbf{C}$ to a unique edge E^p. An interpartition region is itself a graph and by this algorithm, each of these graphs is represented by a single node in V^p; also multiple edges are represented by a single edge in E^p. Consequently, G^p is a graph of supernodes and superedges. We then allocate each supernode to a processor and each superedge to a physical communication link between processors. Each R_i is identified by considering all $e \in E$ as follows. If an edge is in a cut set, then that edge represents external communication requirements for the corresponding processor, otherwise, that edge represents internal communication requirements. All vertices in a particular block of the partition that do not have any associated edges in the cut set are grouped together: this defines the supernodes. Edges in the cut set are also grouped together and form superedges.

Under certain situations, there may be insufficient information in $\hat{G}$ to determine a particular R_i by the above method. We introduce the following definition to address this problem and substitute $\hat{G}'$ for $\hat{G}$ during the application of the partitioning algorithm: $\mathbf{C}(\hat{G}') \mapsto G^p$. Since we are only concerned with identifying physically neighboring vertices in an interpartition region, we ignore the directions of the edges of G for the following definition.

Definition 3: A *satisfiable group of cuts* is a set of cuts $\mathbf{C} \in \zeta$ on G which satisfies either of the following two conditions for all blocks of the partition defined by $\mathbf{C}(G)$:

1. $||R|| = 1$, or
2. there exists at least one path (assuming each edge in G is undirected as stated before) which entirely lies within the subgraph (i.e. the path does not cross any c) between any two vertices in R.

Choose $\mathbf{C} \in \zeta$.
Let G^p be an empty graph initially.
For each edge, $e \in E \cup \bar{E}$ do
BEGIN

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

1. If an 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.
   END
2. Otherwise
   BEGIN
   (a) if $\sigma_-(e)$ is in a graph and if $\sigma_+(e)$ is not in a graph of a supernode, then add $\sigma_+(e)$ to the graph of a supernode containing $\sigma_-(e)$.
   Otherwise, if neither $\sigma_-(e)$ nor $\sigma_+(e)$ is an any graph of a supernode in V^p, then add them both to the same supernode.
   END

END

Figure 3: Partitioning algorithm.

Theorem 1 *Given $s = (\hat{G}, I, F)$, $G^p = (V^p, E^p, \sigma_-^p, \sigma_+^p)$ and any non-temporal group of cuts $\mathbf{C}$ on $\hat{G}$ then, $\mathbf{C}$ on $\hat{G}$ results in interprocessor communication requirements which can be satisfied by G^p: $\mathbf{C}(\hat{G}) \mapsto G^p$.*

Proof.

From the preceding discussion, $\mathbf{C}(\hat{G}')$ can be substituted for $\mathbf{C}(\hat{G})$. As a result of using the partitioning algorithm, each vertex in G^p represents a block of the partition defined by $\mathbf{C}$ on $\hat{G}$. 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 a block of the partition) have been amalgamated into a single vertex. Since the satisfiable group of cuts condition is given by using the augmented graph definition, all vertices in $\hat{G}$ have now been accounted for and the theorem follows.

□

Example 3.1 Figure 4 shows a 3×3 systolic array with geometry $\hat{G}$ and two partition curves, c_1, c_2. Note that c_1 and c_2 do not belong to any ζ considered in this paper and are used to demonstrate the application of the algorithm in a general context. All edges in E, but not in $\bar{E}$, have been explicitly identified; edges in $\bar{E}$ are shown by dashed lines.

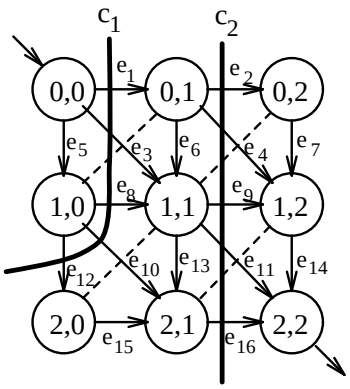

Figure 4: Example of Theorem 1's constructive proof.

The order of edges is as given in Figure 4 (the partitioning algorithm does not specify an order). Since edge e_1 is in the cut set defined by the curve c_1, the three steps in part 1 of the algorithm (see Figure 3) are executed. The second edge, e_2, is in the cut set defined by c_2, thus again the steps in part 1 are executed. In this case, however, only node $(0,2)$ and e_2 are added to the processor graph. The resulting processor graph corresponding to these steps is shown in Figure 5(a). Edges e_3 and

e_4 are treated similar to edge e_2; the corresponding processor graph is shown in Figure 5(b).

Since edge e_5 does not intersect with any partition curve, the steps in part 2 of the algorithm are applied. The first condition holds true and consequently node $(1,0)$ is added to the *supernode* $(0,0)$ in the processor graph. This procedure effectively combines the two nodes into one. The next two edges, e_6 and e_7, are considered similarly. Figure 5(c) shows the resulting processor graph from this sequence of steps. Note that the original nodes $(1,1)$ and $(1,2)$ in the processor graph no longer exist as single nodes; a different edge selection order could have resulted in these nodes not being distinct nodes at all.

When all the edges in E have been considered, the final processor graph consisting of three supernodes and two superedges is as shown in Figure 5(d), which represents a linear pipeline consisting of three processors.

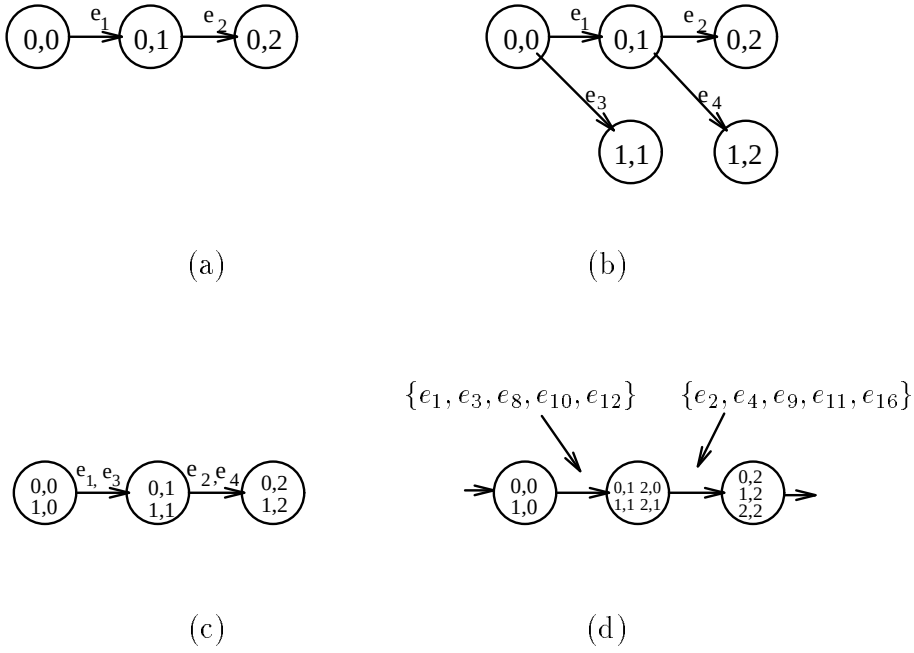

Figure 5: The processor graph construction.

□

Although the edges in $\bar{E}$ were not considered in the above example, in general,

their use is necessary. For example, two partition curves having equations $y = -x+0.5$ and $y = -x + 1.5$ define two cuts parallel to each other *and to edges in* $\bar{E}$. In the previous example, nodes $(0, 1)$ and $(1, 0)$ can only be identified together by considering edges in $\bar{E}$.

Corollary 1 *G^p is a minimum graph which satisfies the interprocessor communication requirements of a systolic array, 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 Theorem 1, 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.

□

Corollary 2 *Given $s = (\hat{G}, I, F)$, $G^p = (V^p, E^p, \sigma^p_-, \sigma^p_+)$ and $\zeta \in \{\zeta_d, \zeta_o, \zeta_r\}$ on $\hat{G}$ then $\zeta(\hat{G}) \mapsto \{G^p_1, G^p_2, \ldots, G^p_{\nu_G}\}$, where the resulting set of graphs differ only in the number of supernodes and superedges, but not in the nature of their connectivity.*

Proof.

Since ζ is a set of a finite number of sets of $\mathbf{C}$, we can enumerate all of the resulting G^p by applying Theorem 1 to each $\mathbf{C} \in \zeta$. Since the form of the lines (or simple closed curves) in ζ have been defined and due to the regularity of $\hat{G}$, each $\mathbf{C}$ will partition the systolic array such that the blocks of the partition will have identical structures, differing only in the number of such blocks.

□

The processor topologies implied by the partitioning (see Theorem 1 and the associated corollaries) may or may not be unique. This observation leads to the following three possibilities: (a) any two or more arbitrary partitions on the systolic array 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.

In the following theorem, the term *intermediate* results refers to results from any partial evaluation of a particular computation which are used in the evaluation of other computations. The most common type is the result of one systolic cell being used as an input to some *other* systolic cell.

Theorem 2 *When no* intermediate *results are used in the computation of s, the temporal partition ζ_t on a systolic array $s = (\hat{G}, I, F)$ results in $\overline{\mathrm{W}}^2$ individually distinct processes.*

Proof.

Let t_0 be associated with the first time front (which also corresponds with the time of the first activation in s) and t_i be associated with the i^{th} time front, $i \geq 0$. Consider the first region of t_0 with inputs I_0 individually. A cell at position (x, y) for $x = y$ in $\hat{G}$ requires one time unit for the data located at positions $(x-1, y)$ and $(x, y-1)$ to reach (x, y) and consequently x time units from the border. Thus, input data from I_0 arrives at cell (i, i) at the i^{th} time instant, t_i. Now consider a cell at position (x, y) with $x \neq y$. Let t_y be the required time units for $I_0^{(0,1)}$ inputs to arrive and t_x be the required time units for $I_0^{(1,0)}$ to arrive. By the previous argument, input data from I_0 will arrive at cell (x, y) at time instants t_x and t_y and since $x \neq y$, the inputs will not arrive during the same time step. Useful computation can only occur then in the main diagonal. A sequential ordering is imposed on all the cell activations with the number of cells activated equal to the number of inputs in $I_0^{(0,1)} = I_0^{(1,0)}$. The diagonal

components of I do not contribute to this discussion since it also requires one time unit for data located at position $(x-1, y-1)$ to reach (x, y).

We now consider the region of I_i. Similar to the I_0 case, the cells in the main diagonal (i.e. (x, y) for $x = y$) will activate (although the activations would normally be spatially displaced and start at cell $(x+i, y+i)$). Cells in the diagonals adjacent to the main diagonal (i.e. (x, y) for $|x-y| = 1$) will activate as inputs I_i and I_{i-1} arrive Again, a sequential ordering on the cell activations exist. In general, the cell (x, y) for $|x-y| = a$ will be activated in its respective diagonal whenever inputs I_i and I_{i-a} arrive. As previously stated, the contribution of the diagonal components may be ignored with no loss in generality.

From the above discussion, a sequential order is imposed on all cells in a *diagonal* during any I_i region. We can now regroup inputs according to their allocation to diagonals. For the main diagonal the groups are $I_i^{(0,1)}$, $I_i^{(1,1)}$ and $I_i^{(1,0)}$. For the diagonals ((x, y) for $|x-y| = a$), the groups are $I_i^{(1,0)}$, $I_{i-a}^{(0,1)}$ and $I_i^{(1,1)}$ or $I_{i-a}^{(1,0)}$, $I_i^{(0,1)}$ and $I_i^{(1,1)}$.

Since a sequential ordering is imposed on all cell activations within each diagonal, a diagonal is considered as a distinct computation. The number of diagonals to be computed and hence the number of distinct computations required is

$$\sum_{k=1}^{\overline{\mathrm{W}}} (2k-1) \;=\; \overline{\mathrm{W}}^2. \qquad (4)$$

□

By Theorem 2, the single systolic computation has been transformed into a set of distinct computations which can then be computed individually. We consider such a set of computations to be amenable to the standard processor farming technique. Consequently, we adopt a single linear chain of processors for illustration in this paper and note that such a processor chain is a part of a general star network (commonly used for processor farms).

The processor topologies derived from Theorems 1 and 2, that are associated with the partitions shown in Figure 2, are as follows: ζ_d (diagonal) - linear bidirectional,

ζ_o (orthogonal) - linear uni-directional, ζ_r (rectangular) - mesh or hexagonal uni-directional, 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 6 and 7, while a similar example is given in Figures 8 and 9 for the correspondence between the Orthogonal Partitioning strategy and the linear uni-directional processor topology.

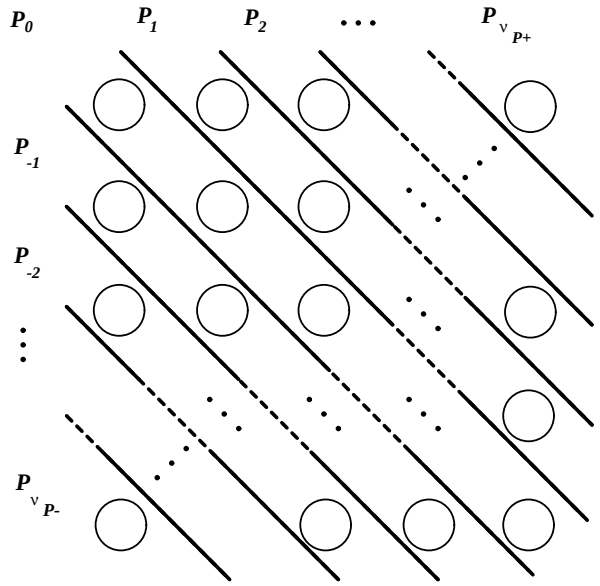

Figure 6: An example of diagonal partitioning (ζ_d); P_i denotes a processor to which the corresponding interpartition region is allocated to (see Figure 7).

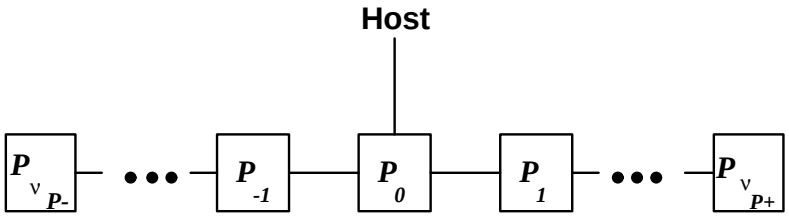

Figure 7: Bidirectional linear processor network resulting from applying ζ_d on $\hat{G}$.

Processor topology impacts upon the required high level language program representation of an algorithm. Two examples are given: (a) an implementation with

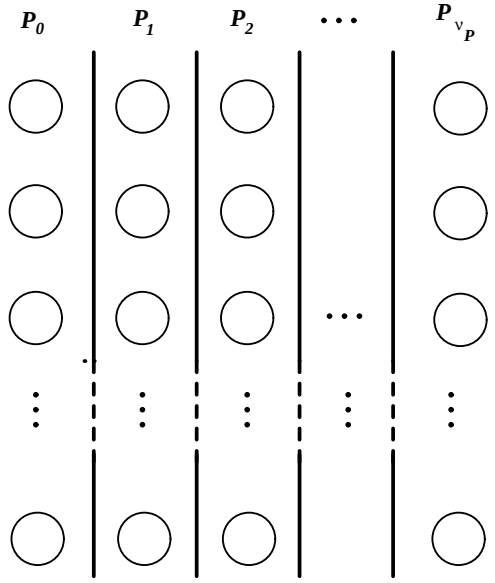

Figure 8: An example of orthogonal partitioning (ζ_o) for partition lines of the form $x = b$.

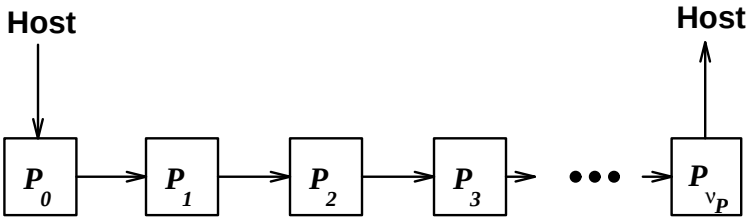

Figure 9: Linear processor network resulting from applying ζ_o on $\hat{G}$.

bidirectional data flow requires different high level language program representation than a program with uni-directional data flow, and (b) extending from a linear network of four processors to five requires at least one additional process instantiation (along with appropriate communication channels) so that the fifth processor can be used. In each of these examples, the differences in the high level language program representation are often subtle. However, when considering specific high level language and machine limitations, these differences may become very pronounced. Indeed, it has been our experience[16, 17, 18] that manually coding the implementation variations implied by the different partitions lead to, at times, widely different program structures, impacting on both the communication and computation segments.

3.3 STAGES 3 AND 4: SOURCE CODE REPRESENTATION AND EVALUATION

In this section, we develop a model to represent the possible source code representations implied by the partitioning process discussed in the previous section. It is reasonable to expect that a unified model based on the unified approach presented in this paper is required to handle much (if not all) of the differences between various DMMCs and source code languages. In this work, we have been heavily influenced by the Occam 2 language and transputer architectures [3]. Consequently, the model which is developed in this section 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. In particular, we are interested in a model which incorporates the following: (a) source code representation in some (possibly arbitrary) computer language which is implied by the partition under study, (b) evaluation of the code representation in terms of its execution time on a specified target machine, and (c) allowance for any (possible arbitrary) target machines.

In this section we also discuss the Evaluation stage and the performance models therein since we consider both stages within the context of an implementational object (defined below).

Definition 4: An *implementational object*, Φ, represents a machine independent source code representation of an implementation (based on a specific type of topology as determined by an application of Theorems 1 or 2) of a systolic algorithm. Formally,

$$\Phi \quad = \quad (\boldsymbol{\Xi}, \mathcal{P}, \Upsilon)$$

where $\boldsymbol{\Xi}$ is a set of *prototype codes* together with a set $\mathcal{P}$ of *implementation parameters* and a *performance model* Υ.

The prototype codes are pre-defined source code templates which essentially are rules for generating source code in a language for which a native compiler is already available. The set of implementation parameters describes the *exact* characteristics

of the implementation. For the most part, these parameters are used in the associated performance model. The performance model evaluates the expected execution time for the given systolic algorithm. The performance model is parameterized so that it can provide a good estimate of the execution time while maintaining general applicability.

Lastly, Φ must be general enough that local program optimizations can be accommodated. For example, if the graph associated with a supernode (in the processor topology graph) has many vertices, then sets of these vertices may be identified via edge contraction. The consequence at the programming representation level is that the number of fine grain processes may be reduced while at the same time, the granularity of each of the remaining processes can be increased. The implementational object is dependent upon the topology and language. Consequently, each Φ must be a pre-defined entity. We denote the process required to produce the implementational objects as Π_2 and note that Π_2 deals with the generation of a systolic array, its partitioning and the construction of the components inherent in an implementation object. Below, we discuss the structure of an implementation object.

The set of prototype codes, $\mathbf{\Xi}$, consist of structured elements, $\mathbf{\Xi} = \{\xi_0, \xi_1, \ldots, \xi_{\nu_{\mathbf{\Xi}}}\}$, where each structured element, ξ_i, represents the rules for the generation of a complete program template in some specific language. Usually $\nu_{\mathbf{\Xi}}$ will be very small and, typically, it is equal to one or two. It is expected that supported languages will fall into two categories, those which are common (e.g. parallel versions of C and FORTRAN) and those which are specialized to an architecture (e.g. Occam 2 and W2 (a language developed for the WARP machine) — although there is a recent emphasis on extending the applicability of Occam 2 to an architecturally independent language). Formally,

$$\xi \quad = \quad (\hat{\mathbf{R}} \subset \mathbf{R}, \hat{\mathbf{L}} \subset \mathbf{L}) \tag{5}$$

where $\mathbf{R}$ represents the generation rules and $\mathbf{L}$ represents the supported languages.

There are two basic functions of ξ_i: firstly, to allow the computation and communication specific parts of the systolic algorithm under consideration to be included and secondly, to provide the 'harness' for such computation and communication. This harness will either be an interface to the operating system (where available) and/or will provide the configuration and allocation of physical processes and communication channels. In [16], we have presented an implementation of ξ for ζ_d for the Occam 2 language on a multiple transputer machine. We now generalize these ideas based on the above considerations. Consequently, ξ_i will consist of the following components.

1. Partition Specification: Details of the precise allocation of processes to processors including communication channel names and datatypes (where applicable) are given. A typical partition specification will consist of three sections: a declaration section containing computation and communication specific information (e.g. the number of systolic processes, machine size and communication datatypes); a declaration section containing a description of the physical network, and the allocation section for instantiating all processes and to enable communication between such processes to occur.

2. Input and Output Specification: Details concerning the procedure used to supply inputs to and extract results from the computational processes are given. An example of a straight forward procedure would be the packaging of the data inputs with initialization information and transmitting this to the source while waiting for output from the sink. More complex input/output controls may be required in other circumstances however.

3. Computation Specification: A computational unit consisting of a local input, a computation process and a local output is specified. The input and output in this context consists of appropriate communication primitives linking other computational units together.

4. Communication Specification: The communication specification has three principle components. The input and output sections must be updated with the exact time and sequence of data required. The communication of data is often to be embedded into an existing communication control structure, for example, a data packet.

5. Router Specification: This defines the nature of the communication between physical processes as distinct from the logical requirements of the computation (see Communication Specification). A typical router specification would detail the specification of a simple channel monitoring loop, shunting the input/output packets to a computation process or to another processor.

The performance model is intended to be applied to each prospective source code implementation which forms a possible solution to the given systolic algorithm. There is no need to generate the source code for each implementation until a particular implementation has been chosen. As such, the performance model must take into account factors from several domains: the flexibility in the generation of the prototype codes, architecture differences, the flexibility in the way the systolic array can be partitioned, the degree of local program optimization as well as the computation and communication requirements of the algorithm.

A performance model (Υ) is defined as

$$\exists p_f \in \mathcal{P}_f,\ \forall p_v \in \mathcal{P}_v \quad | \quad \Upsilon(p_f, p_v, \zeta, s) \rightarrow T(\text{expected execution time}). \tag{6}$$

where $\mathcal{P}_f$ and $\mathcal{P}_v$ are mutually exclusive subsets of $\mathcal{P}$; $\mathcal{P}_f$ refers to the set of fixed parameters while $\mathcal{P}_v$ refers to the set of variable parameters. Fixed parameters reflect the structure of Φ, that is, particular language and run-time environments. Variable parameters reflect the flexibility of Φ (e.g. local optimizations, number of blocks of the partition, algorithmic details, etc.).

This definition for Υ allows the performance model to be a good estimator since

characteristics from all three domains, implementation, algorithmic and machine, are taken into account. We have given a few performance models in [17].

Execution

The implementational objects must be determined prior to their use in the execution of this stage. Once a set of implementational objects exist, the Source Code Representation stage can be executed. The application of this stage essentially combines the prototype codes (from the implementational objects) with the systolic algorithm (which, from the application of the previous stage, is now in a partitioned form of the systolic array).

An analysis of $s = (\hat{G}, I, F)$ to determine the required communication, computation and program structure is required. This analysis will provide: (a) appropriate information to the performance models, and (b) the information necessary to generate the final source code representation. It is natural to sub-divide this procedure into its three constituent parts, namely, communication, computation and program structure. Then each part can be accomplished by employing individual tools. For the execution of this stage, only the program structure task is required to be executed.

The fourth stage, Evaluation, requires the other two tasks, namely, the communication and computation analysis tasks. We have developed a tool called COMMAN to analyze the communication requirements of Occam 2 programs [19]. We have used COMMAN to analyze programs which have been generated by the execution of stage 3 and to provide estimates of the communication times for the Occam 2 language and transputer implementations of these programs.

The performance models associated with each of the possible implementations under consideration are now executed. Consequently, a set of predicted execution times is formed and the final implementation can now be selected.

We have mentioned that local optimizations on the source code representation may be considered. The blocks of the partition defined by a partitioning that have

been considered in this paper are composed of groups of systolic array cells which are allocated to the same processor. There is no reason not to consider amalgamating these cell functions. Issues in such a process include granularity analysis (both of the algorithm and machine) and communication dependences (i.e. amalgamating cells may impact upon the overall communication progression in the system). Additional tasks may be defined for the execution of these local optimizations; however, since these tasks represent optional components in the model, we do not deal with them further in this paper.

The result of the Evaluation Stage then, is the determination of which possible implementation would be the most efficient for the given systolic algorithm.

4 CONCLUSION

We have proposed a unified approach for the compilation of systolic computations for distributed memory multicomputers that is generally applicable to a wide range of systolic algorithms, DMMCs and computers languages. Furthermore, in [17], we have developed a model based on the unified approach. The application of the model is expected to eliminate or significantly reduce existing problems with soft-systolic implementations.

The unification in the proposed approach is three-fold. Firstly, we have developed a framework whereby we incorporate several existing models and procedures in systolic processing, namely, systolic array design methods and models. Secondly, we have considered the efficient software implementation of these algorithms — a unification of the common ideas in VLSI solutions with the generality of programming solutions. Thirdly, we have incorporated algorithmic, implementational and machine characteristics into our approach — a unification of three fundamental computing concepts. The approach we have proposed is two-layered with the first layer specifying the unified abstractions while the second layer details the structure of objects representing

the source code generation and performance modeling of implementations.

We believe that this is the first unified approach that deals with the aspects of generality, optimizations, prediction and efficiency from specification to implementation within a unified context. We expect that an application of the approach will result in a more productive environment for systolic processing.

Details concerning the derivation of two (ζ_d and ζ_t) implementational objects are given in [17, Chapters 4 and 5] and in [18] while in [16, 20] we have further discussed the role of prototype codes. Some additional details relating to partitioning strategies also appears in [20]. The development of a unified model, based on the unified approach discussed in this paper can be found in [17]. We have also conducted many experiments to investigate various key issues and concerns relating to the application of the unified model [17, Chapter 7]. In particular, we have considered the application of the model to two different systolic algorithms, the Weighted Levenshtein Distance algorithm and the matrix multiplication algorithm. Our results have demonstrated the usefulness of the unified approach and model.

References

[1] K. Johnson, A. Hurson and B. Shirazi, "General-Purpose Systolic Arrays," *IEEE Computer*, pp. 20–31, Nov. 1993.

[2] G.M. Megson, *An Introduction to Systolic Algorithm Design*. New York, USA: Oxford University Press, 1992.

[3] Inmos Limited, Prentice-Hall Ltd., Hertfordshire, UK, *Transputer Reference Manual*, ISBN 0-13-929001-X ed., 1988.

[4] G.M. Megson, "Transputer Implementation of Systolic Arrays for Model Reduction," *IEE Proceedings*, Vol. 137, pp. 343–352, Sept. 1990.

[5] M.D. Rice and S.B. Seidman, "A Multiprocessor Testbed for Generalized Systolic Computation," *Proceedings of Transputing '91*, 1991, P. W. et al., (ed.), IOS Press, Amsterdam, pp. 281–295, 1991.

[6] C. Lengauer, M. Barnett and D.G. Hudson, "Towards Systolizing Compilation," *Distributed Computing*, Vol. 5, pp. 7–24, . 1991.

[7] G. Megson, (ed.), *Transformational Approaches to Systolic Design*. 2-6 Boundar Row, London SE1 8HN, UK: Chapman & Hall, 1994.

[8] B.R. Engstrom and P.R. Cappello, "The SDEF Systolic Programming System," in *Concurrent Computations, Algorithms, Architecture, and Technology*, B. D. S.K. Tewksbury and S. Schwartz, (eds.), pp. 263–301, Plenum Press, New York, 1988.

[9] B.J. d'Auriol and V.C.Bhavsar, "Systolic and Wavefront Array Algorithms on Distributed Memory, Multiprocessor Computers," *Proc. of Supercomputing Symposium '93-High Performance Computing: New Horizons (SS93)*, Calgary, Alberta, Canada, June, 6-9, 1993, L. Bauwens, (ed.), University of Calgary, Calgary, Alberta, pp. 47–54, June 1993.

[10] R.G. Melhem and W.C. Rheinbold, "A Mathematical Model for the Verification of Systolic Networks," *SIAM Journal on Computing*, Vol. 13, pp. 541–565, August 1984.

[11] D.I. Moldovan, *Parallel Processing, From Applications to Systems*. 2929 Campus Drive, Suite 260, San Mateo, CA, USA, 94403: Morgan Kaufmann Publishers Inc., 1993.

[12] X. Zhong and S. Rajopadhye, "Systematic Generation of Linear Allocation Functions in Systolic Array Design," *Journal of VLSI Signal Processing*, Vol. 4, pp. 279–93, Nov. 1992.

[13] C.A. Mead, *Introduction to VLSI Systems.* Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1980.

[14] W.L. Miranker, "Spacetime Representations of Computational Structures," *Computing*, pp. 93–114, 1984.

[15] A. Darte, T. Risset and Y. Robert, "Synthesizing Systolic Arrays: Some Recent Developments," *Proceedings of the International Conference on Application Specific Array Processors (Cat. No.91TH0382-2)*, Barcelona, Spain, Sept., 2-4, 1991, Los Alamitos, CA, USA: IEEE Compute. Soc. Press 1991, pp. 372–386, Sept. 1991.

[16] B.J. d'Auriol, V.C. Bhavsar, L. Goldfarb, "Systolic Array Implementations for Reconfigurable Learning Machines on Transputers," *Proc. of Supercomputing Symposium '91*, Fredericton, N.B., Canada, June, 3-5, 1991, V. Bhavsar and U. Gujar, (eds.), University of New Brunswick Press, Fredericton, N.B., pp. 105–119, June 1991.

[17] B.J. d'Auriol, *A Unified Model for Compiling Systolic Computations for Distributed Memory Multicomputers.* PhD thesis, Faculty of Computer Science, University of New Brunswick, June 1995.

[18] B.J. d'Auriol and V.C. Bhavsar, "Evaluation of the Multi-Transputer Implementations of the Weighted Levenshtein Distance Computatation," *Proc. of International Conference on Parallel Computing and Transputer Applications '92 (PACTA'92)*, Barcelona, Spain, Sept., 21-25 Sept. Published as 'Parallel Computing and Transputer Applications', Part II, 1992, M. V. et al., (ed.), IOS Press, Amsterdam, pp. 879–888, Sept., Sept. 1992.

[19] B.J. d'Auriol and V.C. Bhavsar, "COMMAN — A Communication Analyzer for Occam 2," Tech. Rep. TR94-086, Faculty of Computer Science, University of

New Brunswick, P.O. Box 4400, Station A, Fredericton, N.B., E3B 5A3, June 1994. Accepted for publication in Transputer Communications with minor modifications in May 1995.

[20] B.J. d'Auriol, V.C. Bhavsar, L. Goldfarb, "Multi-Transputer Implementations of the Metric Approach to Pattern Recognition Using Weighted Levenshtein Distance," *Applications of Transputer 3: Volume II*, August 1991, e. a. T.S. Durrani, (ed.), IOS Press, Amsterdam, pp. 388–393, August 1991.