GENERIC PROGRAM STRUCTURES INDUCED BY PARTITIONS OF A SYSTOLIC COMPUTATION GRAPH

Brian J. d'Auriol[*]
Dept. of Mathematics and Computer Science
The University of Akron
Akron, Ohio, 44325-4002, USA
bdauriol@cs.uakron.edu

Virendrakumar C. Bhavsar[†]
Faculty of Computer Science
University of New Brunswick
Fredericton, N.B., E3B 5A3, Canada
bhavsar@unb.ca

Abstract

We propose a technique which essentially constructs a set of program structures, 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 processes induce a specific processor topology dependent on the partitioning strategy used. The processing requirements based on the topology imply a specific program structure. Details of such resulting program structures are discussed. 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, Graph Partitioning and Mapping, Systolic Computations

1 Introduction

In this paper, we present a technique which essentially constructs a set of program structures, each representing a valid parallel implementation of a given systolic algorithm. By program structures we mean the definition and interrelation of program modules together with their interface and implementation descriptions. We consider that the program structures model different types of implementations similar to concurrent program archetypes [1].

Our approach is based on partitioning a reduced data dependency graph (often referred to as a systolic computation graph (SCG)) representing a systolic algorithm and the subsequent mappings of the computations involved to processors in a multicomputer environment. In this paper, we are primarily interested in the cause-effect relationship between partitioning the SCG and the resulting induced program structure. Note that we investigate a different problem than the more traditional one of partitioning the SCG onto an existing computational device (e.g. a VLSI device). In this work, we have been influenced by the occam 2 language and transputer architectures [2]. Furthermore, since the generation of SCGs is well known (see for example [3]), we assume that the SCG for a particular systolic algorithm is given.

We develop a graph-based notation to represent an SCG and note that there are similarities with the notation given in [4]. The principal advantage of our formalism is that it allows for both the definition and partitioning of the SCG in an integrated model. A discussion of the applicability of other formal models appears in [5].

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 program structure is presented while details resulting from the use of this technique are given 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 [4]). 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 $\deg(v)$. We denote by F the set of computations associated with all $v \in V$. The data sequencing of the inputs has both time and space dimensions. We denote a datum value by d and define the data sequencing as $I = (E^I, \sigma_+^I, \varsigma^I)$ where E^I is the set of directed edges not in E that 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 ς^I is a scheduling function which establishes a time-ordered sequence for the data items as-

[*]Partially supported by NSERC doctoral fellowship.
[†]Partially supported by NSERC grant, OGP0089.

sociated with each edge: $\varsigma^I(e \in E^I, d) \mapsto \{0, 1, 2, \ldots\}$.

We define an SCG, s, as $s = (G, I, F)$. This represents a generalization of previous definitions relating to systolic arrays (see for example [3]) In certain cases, the progression of computations in s will exhibit a wavefront property. This has often been referred to as a *wavefront array* [6].

Let $\mathcal{A}$ denote a given input systolic algorithm. Note that $\mathcal{A}$ has the usual systolic constraints [3]. There are several systolic array design methodologies which may be used to transform $\mathcal{A}$ into s (see for example [3, 7]); moreover, such synthesis techniques often result in a family of different SCGs.

We denote by $\hat{G}$ a restricted SCG as exemplified in Figure 1 and note that: (a) $\hat{G}$ has the wavefront property and (b) there is a single source and sink for data. These properties provide for a mechanism to allow parallel processing and to simplify the coding requirements, respectively.

Due to the impact on I and F by the single source and sink property of $\hat{G}$, we consider alternative definitions for I and F. Specifically, we denote by $\hat{I} = (E^{\hat{I}}, \sigma_+^{\hat{I}}, \varsigma_i^{\hat{I}})$ and $\hat{F}$, the newly transformed data sequencing and the set of computation functions, respectively. Therefore we define $\hat{s} = (\hat{G}, \hat{I}, \hat{F})$ and $\hat{G} = (\hat{V}, \hat{E}, \sigma_+, \sigma_-)$. Further details of the properties of $\hat{G}$ appear in [5].

3 Partitioning of the Systolic Computation Graph

In this section, several space and time partitioning schemes are presented. 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 $i, j \in \{0, 1, 2, \ldots\}$, 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)}. We denote by $f_{i,j}$ the computation associated with $v_{i,j}$, $f_{i,j} \in F$. A sequence of computations is denoted by $(f_{i_0,j_0}, f_{i_1,j_1}, \ldots)$ and the set of such sequences, such that each sequence is independent of any other sequence, is denoted by F^*. The arguments to a particular function, where necessary, are denoted as $f(d_0, d_1, \ldots)$.

We denote by c a simple curve which induces a cut of $\hat{G}$. 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, $\zeta = \{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_{\nu_\zeta}\}$. The collection ζ essentially groups several $\mathbf{C}$ sets together which result in the same program structure. Note that this collection has the same parallel implementation.

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: $W_i : I|\varsigma^I(\forall e \in E^I, d) \mapsto i$. Let c provide a cut of the time line. Consequently, c partitions the input waves into two sets of waves or *wave regions*.

In this paper, we consider two particular characteristic sets of curves: *Diagonal* (ζ_d) and *Orthogonal* (ζ_o). 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. Additionally, we also consider a *Temporal* partitioning (ζ_t). Lastly, the *Null* partition (ζ_s) represents 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 $\hat{G}' = (\hat{V}, \hat{E} \cup \bar{E}, \sigma_-, \sigma_+)$, where $\bar{E}$ is a set of diagonal edges perpendicular to the set of diagonal edges in $\hat{E}$ such that $\sigma_-(e \in \bar{E}) = v_{i,j}$ and $\sigma_+(e \in \bar{E}) = v_{i+1,j-1}$. The incorporation of $\bar{E}$ edges in the graph does not change the behavior of s in any way.

Finally, 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 $\sigma_-^p(e) = v_i$. In subsequent discussions, we show that G^p is a graph consisting of supernodes and superedges, and moreover represents the derived processor topology that a parallel machine should have in order to execute a program based on this graph.

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 $e \in E^{\hat{I}}$, $\sigma_+^{\hat{I}}(e) = v_{0,0}$. 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 processor graph is constructed by applying a partition to the SCG. $\mathbf{C}(\hat{s}) \mapsto G^p$. 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. The non-intersecting case represents Steps 2a-2e in Figure 2. Further discussion of the partitioning algorithm appears in [5].

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.

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. □

Lemma 2 *Given $\hat{s}$ and ζ_o, $\zeta_o(\hat{s}) \mapsto \{G_1^p, G_2^p, \ldots, G_{\nu_G}^p\}$, where each G_i^p has 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, after applying 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. □

Lemma 3 *Given $\hat{s}$ and ζ_d, $\zeta_d(\hat{s}) \mapsto \{G_1^p, G_2^p, \ldots, G_{\nu_G}^p\}$, where each G_i^p has 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 follows that for Lemma 2. □

We now turn our attention to the temporal partitioning case, 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}. In the following discussion, for simplicity, we do not show function arguments for $f_{i,j}$. Note that this may result in the appearance of duplicate function sequences. 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, $\hat{s}$, ζ_t and $v_{i,j} \in \hat{V}$ such that $0 \leq i, j \leq N$ and two waves W_i and W_{i+1} that are contained in the distinct wave regions induced by ζ_t do not violate the wave independence restriction, $\zeta_t(\hat{s}) \mapsto F^*$*

Proof.

Consider the wave W_0. At t_0, the computation of $f_{0,0}$ occurs followed by $f_{1,1}$ at t_1, thus the ordered set of operations $(f_{0,0}, f_{1,1}, \ldots, f_{N,N})$, 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, $(f_{0,0}, f_{1,1}, \ldots, f_{N,N})$ 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, $(f_{0,1}, f_{1,2}, \ldots, f_{N-1})$ and $(f_{1,0}, f_{2,1}, \ldots, f_{N-1})$ exist. We consider all groupings of waves and the subsequent collections of function sequences in similar manner. Thus, $F^* = \{(f_{0,0}, f_{1,1}, \ldots, f_{N,N}), (f_{0,0}, f_{1,1}, \ldots, f_{N,N}), (f_{0,1}, f_{1,2}, \ldots, f_{N-1}), (f_{1,0}, f_{2,1}, \ldots, f_{N-1}), \ldots, f_{\nu_{F^*}}^*\}$ □

We conclude [5] from Lemma 4 that F^* can be computed by the standard processor farming technique [2]. 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.

In summary, the processor topologies derived from Lemmas 2, 3 and 4 refer to the example partitions shown in Figure 1, and these are as follows: ζ_o (orthogonal) - linear uni-directional topology (illustrated in Figure 3), ζ_d (diagonal) - linear bidirectional topology (illustrated in Figure 4), ζ_t (temporal) - linear bidirectional topology (an implementation of a processor farm), and ζ_s (sequential) - single processor.

4 Program Structures

In this section, we present the program structure of a given high level language program necessary to implement $\mathcal{A}$ on a multicomputer, for example, with topologies as described by Lemmas 2, 3 and 4. We concentrate on defining aspects of cohesion and coupling [8].

From Section 3, $\mathcal{A}$ is represented by $\hat{G}, \hat{I}$ and $\hat{F}$ where $\hat{G}$ and $\hat{F}$ are embedded in G^p. A $v \in V^p$ therefore specifies a subset of $\hat{F}$ together with the specification of any necessary internal communication requirements. We denote by M_F^s this computational process associated with a $v \in V^p$. Also, an $e \in E^p$ represents required communication between specific instantiations of M_F^s that is not specified by $\hat{F}$. Consequently, we denote by M_R^s the communication routing indicated by E^p.

Due to the single source and sink property of $\hat{G}$, 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 are 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: $\mathbf{M}^s = \{\mathrm{M}_F^s, \mathrm{M}_{IO}^s, \mathrm{M}_R^s, \mathrm{M}_C^s, \mathrm{M}_{CM}^s\}$. These five software modules are necessary components, since the removal of any one module makes the implementation infeasible.

The three relationships *IS-COMPONENT-OF*, *USES* and *INSTANTIATES* are defined 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 [9]. 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. Implicit in this discussion is the fact that one or more modules of $\mathbf{M}^s$ are generic modules. These relationships are shown in Figure 5.

The $\mathbf{M}^s$ represents 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 $\mathbf{M}^s$. The effect of the partitioning strategy is principally realized in the internal definition of each module, and consequently to some extent, also in the interfaces of the modules. This also impacts upon the precise definition of $\mathbf{M}^s$.

Since the primary purpose of M_F^s is to perform the computation, we consider the following three procedural objects in its interface (see Figure 5): `Compute`, `Accept-Work-Packet` and `Produce-WorkPacket`. The latter two provide the functions required to input and output the data to and from the computation process. The `Compute` process reflects computations in F or F^*; its details also depend upon ζ.

There are two procedural objects in M_R^s (see Figure 5) 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 depend on ζ.

There are four functions incorporated into M_{IO}^s (see Figure 5): `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 ζ. The latter two procedures are connected to, usually, the procedures in M_R^s.

Note that there are implementation details which are not strictly dependent upon the partitioning strategy as well as the input algorithm. Consequently, there may exist additional *external* specification of $\mathbf{M}^s$ which, for example, 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 in this paper.

We denote by $\overline{\mathbf{M}^s}$ the definition of $\mathbf{M}^s$ resulting from the partitioning strategy. From the above discussion the following lemma follows.

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

We have manually coded several implementations derived from the partitionings presented in this paper [5, 10]. One part of a particular implementation is discussed below as a specific example of the modules discussed herein.

Figure 6 shows the occam 2 pseudocode for M^s_{IO} resulting from ζ_t being applied to the matrix multiplication algorithm. 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 6. 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 the figure. 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.

5 Conclusion

We have shown that the partitioning and mapping processes of a systolic computation graph induce an associated set of program structures for a parallel implementation of the input algorithm. We have established this in two parts: firstly, we have shown that the partitioning and mapping processes induce a specific processor topology associated with each partitioning strategy (see Lemmas 2, 3 and 4) and secondly, we have shown that the program processing requirements based on the topology imply a specific program structure. The details of the resulting program structure have been provided. Furthermore, certain details of the resulting program structures are induced by the partitioning strategy (see Lemma 5).

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.

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, for example, see [11]. We have also applied the proposed technique in context of developing a compiler for high performance computers [10].

References

[1] K.M. Chandy, "Concurrent Program Archetypes," *Proc. of the 1994 Scalable Parallel Libraries Conference*, Mississippi State, Mississippi, USA, October, 12-14, 1994, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, pp. 1–9, October 1994.

[2] R.S. Cok, *Parallel Programs for the Transputer*. Englewood Cliffs, New Jersey, 07632: Prentice Hall, 1991.

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

[4] 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.

[5] B.J. d'Auriol and V.C. Bhavsar, "Generic Program Structures Induced by Partitions of a Systolic Computation Graph," Tech. Rep. 97/04, Department of Computer Science, The University of Manitoba, Winnipeg, Manitoba, Canada, R3T 2N2, March 1997.

[6] 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*, Vol. 31, pp. 1054–1066, November 1982.

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

[8] I. Sommerville, *Software Engineering, Fifth Ed.* Addison-Wesley Publishing Company, 1996.

[9] J. Buxton and J. McDermid, "HOOD (Hierarchical Object Oriented Design)," in *SoftwareEngineering A European Perspective*, C. Richter, (ed.), ch. 4, pp. 222–225, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA, 90720-1264: IEEE Computer Society Press, 1993.

[10] 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, Fredericton, N.B. Canada, June 1995.

[11] B.J. d'Auriol and M.B. Dugar, "A Systolic Array Graph Partitioning System," *Proc. of the Eighth IASTD International Conference on Parallel and Distributed Computing and Systems (PDCS'96)*, Chicago, Illinois, USA, October, 16-19, 1996, K. Li, T. Abdelrahman, and E. Luque, (eds.), IASTED/ACTA Press, pp. 356–358, October 1996.

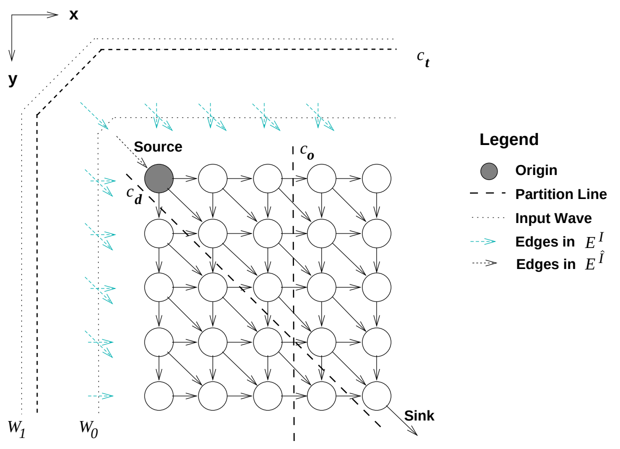

Figure 1: Structure of $\hat{G}$ for a 5×5 systolic array size and examples of three characteristic partitions on $\hat{G}$.

```
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 2: Partitioning algorithm

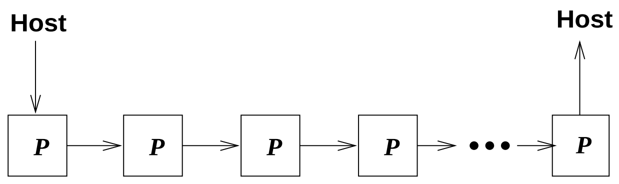

Figure 3: Linear network of processors resulting from applying ζ_o on $\hat{G}$.

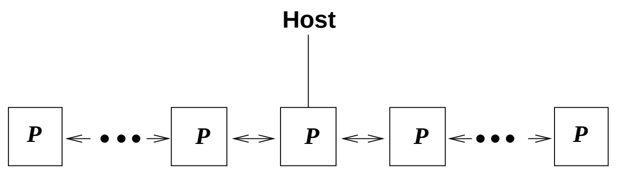

Figure 4: Bidirectional linear network of processors resulting from applying ζ_d on $\hat{G}$.

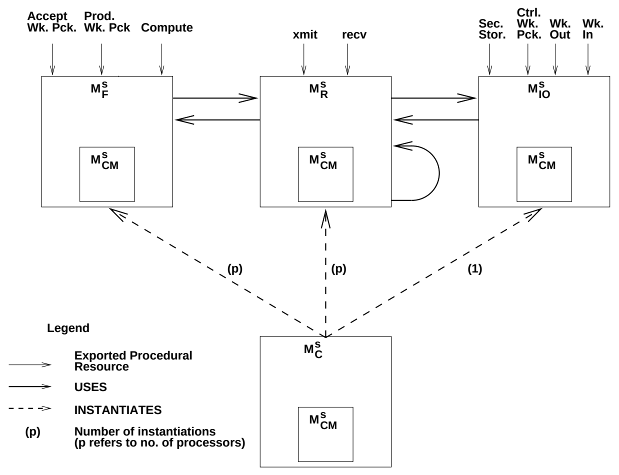

Figure 5: A derived program structure.

```
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 6: The occam 2 pseudocode showing a particular coding of M^s_{IO} for matrix multiplication with a farming implementation.