Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

MAMACG: A TOOL FOR AUTOMATIC MAPPING OF MATRIX ALGORITHMS INTO MESH ARRAY COMPUTATIONAL GRAPHS

D. Le

March 1992
CSD-920009

UNIVERSITY OF CALIFORNIA

Los Angeles

MAMACG: A Tool for Automatic Mapping of Matrix Algorithms into Mesh Array Computational Graphs

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Dinh Lê

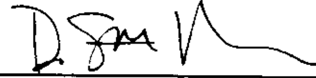
1992

© Copyright by

Dinh Lê

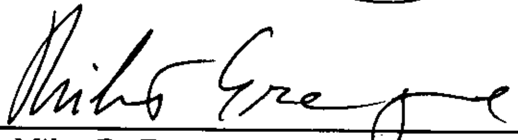
1992

The thesis of Dinh Lê is approved.



D. Stott Parker

Richard R. Muntz



Miloš D. Ercegovac, Committee Chair

University of California, Los Angeles

1992

To the boys and girls back home who don't know beans about my work, but who provided security, comfort and good food whenever I needed. They deserve my deep sincere thanks.

Table of Contents

1 Introduction 1

2 Transformation of MAC-algorithms into Array Computational Format 6

2.1 LU-decomposition 7

2.2 Warshall Algorithm for Transitive Closure 10

3 Derivation of the *ODG* 13

3.1 Symbolic Execution 13

3.2 Mapping of Symbolic Statements Into *ODG* 14

3.3 Derivation of *lu-decomp(3)'s ODG* 20

3.4 Derivation of *warshall(3)'s ODG* 21

4 Derivation of the *MMG* and *MAC-graph* 23

4.1 Definitions of Bidirectional Flows 23

4.2 Removal of Bidirectional Flows in *ODG* 26

4.3 Regularize the Last Plane 32

4.4 Generation of the Mesh Array Computational Graph 35

5 Implementation of MAMACG 41

6 A Survey of Related Work 44

6.1 Rao's Regular Iterative Algorithm (RIA) 45

6.2 MQRS's Afine Recurrence Equations (ARE) 46

6.3 VLSI Array Compiler System (VACS) 49

6.4 SDEF Programming System 51

6.5 ADVIS: Automatic Design of VLSI Systems 52

7 Summary and Further Research 54

A Running MAMACG 57

References 61

List of Figures

1.1 Steps taken to derive mesh array computational graphs 5

3.1 Steps taken in deriving *lu-decomp(3)'s ODG* 15

3.2 Steps taken in deriving *warshall(3)'s ODG* 22

4.1 *Warshall(3)'s ODG* 24

4.2 Graph generated after the removal of the negative nodes 29

4.3 *MMG* generated by *Warshall(3)* 34

4.4 *MAC-graph* generated by grouping along the Z-axis of *LU-decomp(3)'s MMG* 36

4.5 *MAC-graph* generated by grouping along the Y-axis of *LU-decomp(3)'s MMG* 36

4.6 *MAC-graph* generated by grouping along the X-axis of *LU-decomp(3)'s MMG* 37

4.7 *MAC-graph* generated by grouping along the Z-axis of *Warshall(3)'s MMG* 38

4.8 *MAC-graph* generated by grouping along the Y-axis of *Warshall(3)'s MMG* 39

4.9 *MAC-graph* generated by grouping along the X-axis of *Warshall(3)'s MMG* 39

List of Tables

Acknowledgments

Prof. Miloš Ercegovac set a high level of excellence early on, then ratcheted it up whenever I started to feel satisfied. Our journey together had its rough moments, due primarily to my impatience, but the satisfying results make it all worthwhile. Your guidance is greatly appreciated. Also appreciated is the time and advice of committee members Prof. Richard Muntz and Prof. Stott Parker.

Dr. Jaime Moreno's dissertation embarked me on this adventure. Not only did Jaime teach me everything I know about mesh array computational graphs, but his guidance, dedication, and wisdom transfigured this journeyman's work. I am in debt to you the most. Thank you Prof. Tomás Lang for all the long-distance assistance that reached me by way of Jaime.

Brad Pierce helped me to revise my prose until it communicated what I really meant and to highlight the salient results of my research. William Cheng's magical graphical editor (tgif) and his personal tutorial enabled me to generate those beautiful graphics automatically. Thank you kindly, friends.

Finally, thank you to my parents and family. We have not always seen eye to eye, but we have seen a lot together.

This research has been supported in part by the NSF Grant No. MIP-8813340 *Composite Operations Using On-Line Arithmetic for Application-Specific Parallel Architectures: Algorithms, Design, and Experimental Studies* and by the State of California Hughes Aircraft Company MICRO Project *Design of Mesh Arrays for Matrix Computations.*

Abstract of the Thesis

MAMACG: A Tool for Automatic Mapping of Matrix Algorithms into Mesh Array Computational Graphs

by

Dinh Lê

Master of Science in Computer Science

University of California, Los Angeles, 1992

Professor Miloš D. Ercegovac, Chair

The design of MAMACG, a software tool for automatically mapping an important class of matrix algorithms into mesh array computational graphs, is described. MAMACG is a concrete realization of Moreno's multi-mesh graph (MMG) method. The author's MAMACG implementation in Elk, a dialect of LISP with built-in X-graphics capability, is also described.

CHAPTER 1

Introduction

In [1], Moreno and Lang considered a class of matrix algorithms with the "loop-body structure" described below:

> Assume the following primitive syntactic sets: *numerals* (**Nml**), *identifiers* (**Id**), and *base types* (**BTypes**), where
>
> $$\mathbf{Nml} = \{0, 1, 2, \ldots\}, \mathbf{BTypes} = \{int, bool, double\}.$$
>
> From these, sets of *expressions* (**Exp**), *commands* (**Com**), *declarations* (**Dec**), and *types* (**Types**) are generated. The following notational conventions are used:
>
> $$x \in \mathbf{Id}, k, m \in \mathbf{Nml}, bt \in \mathbf{BTypes}, v \in \mathbf{Var}$$
>
> $$e \in \mathbf{Exp}, c \in \mathbf{Com}, d \in \mathbf{Dec}, ty \in \mathbf{Types}$$
>
> **Syntax Equations**
>
> *loop-body* ::= **for** x := e_0 **to** e_1 **do** c
>
> v ::= x | $x[e_1, e_2]$
>
> e ::= v | e_0 *bop* e_1 | *uop* e | (e)
>
> *bop* ::= + | − | * | / | ⊗ | ⋯

uop ::= $-$ | sin | cos | $\cdots$

c ::= v := e | $c_0; c_1$ | *loop-body*

d ::= **var** $x : ty$ | $d_0 ; d_1$

ty ::= bt | **array**$(k_1 : m_1 ; k_2 : m_2)$ **of** bt

Variable x is a *simple variable* and variable $x[e_1, e_2]$ is a *subscripted variable* where e_1 and e_2 are the elements of its *index* $[e_1, e_2]$. Note that only 2-tuple subscripted variables are accepted by this language.

This thesis considers those matrix algorithms which may be described by a single 2-tuple subscripted variable and up to three simple variables (used to index the subscripted variable). Many matrix algorithms such as transitive closure, LU-decomposition and BA^{-1} fall into this important subclass, called mesh array computational algorithm (*MAC-algorithm*).

MAC-algorithms can be rewritten in special formats, which, when symbolically executed, produce parallel code with fine-grain granularity suited for implementation on processor arrays. Indeed, extensive research related to effective mapping of matrix algorithms is available [2] but few of these approaches have resulted in practical design tools. This thesis describes the design and implementation of a tool, called MAMACG, which is based on the MMG method described in [1]. The principal feature of MAMACG is the use of Elk [3], a dialect of LISP with built-in X-graphics capability. Because of its high abstraction for fast implementation and interpretive nature for easy debugging, Elk is an ideal prototyping language. The automation of the mapping of MAC-algorithms onto mesh array computational graphs proceeds in the following three stages:

1. Derive the orthogonal data-dependency graph *(ODG)* of a MAC-algorithm.
2. Transform the *ODG* into a three-dimensional multimesh graph (*MMG*).
3. Transform the *MMG* into a two-dimensional mesh array computational graph (*MAC-graph*).

The second stage produces a unique solution—a three-dimensional graph with unidirectional nearest-neighbor dependencies (*MMG*). The third stage produces alternative solutions of mesh array computational graphs, based on the grouping of the nodes in the *MMG* parallel to the X, Y, or Z-axis and also on the size of groupings.

This thesis begins by discussing how MAC-algorithms are represented in a special formats which require all 2-tuple subscripted variables to be rewritten as 3-tuple subscripted variables, without changing the semantics of the algorithms. This special format is referred to as *array computational format* due to its suitability for implementation in arrays. This thesis then shows how the generated symbolic statements of the algorithm's array computational format are used to produce an orthogonal data-dependency graph (*ODG*) which associates every 3-tuple subscripted variable with a node and edges in the three-dimensional *ODG*. Next the *MMG* is derived by removing bidirectional flows existing in the *ODG*. Then the derivation of the mesh array computational graphs from the *MMG* is described. The development and implementation of MAMACG in Elk is discussed in Chapter 5. A survey of related works is then described in Chapter 6. The thesis concludes by summarizing the contributions of the research reported in the thesis and suggests future research directions. The use of MAMACG is described in Appendix A.

Note that all graphs shown in this thesis have been obtained using MAMACG. The mapping of MAC-algorithms to their computational formats requires user intervention, while the other steps are fully automated by MAMACG. Figure 1.1 shows the steps taken to map matrix algorithms to mesh array computational graphs. Steps 2 through 4 are based on the MMG method described in [1]. MAMACG refines and formulates the MMG method's high level description in terms of an Algol-like language in order to automate the transformation in Elk.

The regularization stage in the MMG method transforms a matrix algorithm into a fully-parallel data-dependency graph (*FPG*). This graph is then transformed into the multimesh graph (*MMG*). The approach followed by MAMACG does not use the *FPG*. Instead it derives the orthogonal data-dependency graph (*ODG*) which bears a closer resemblance to the *MMG*.

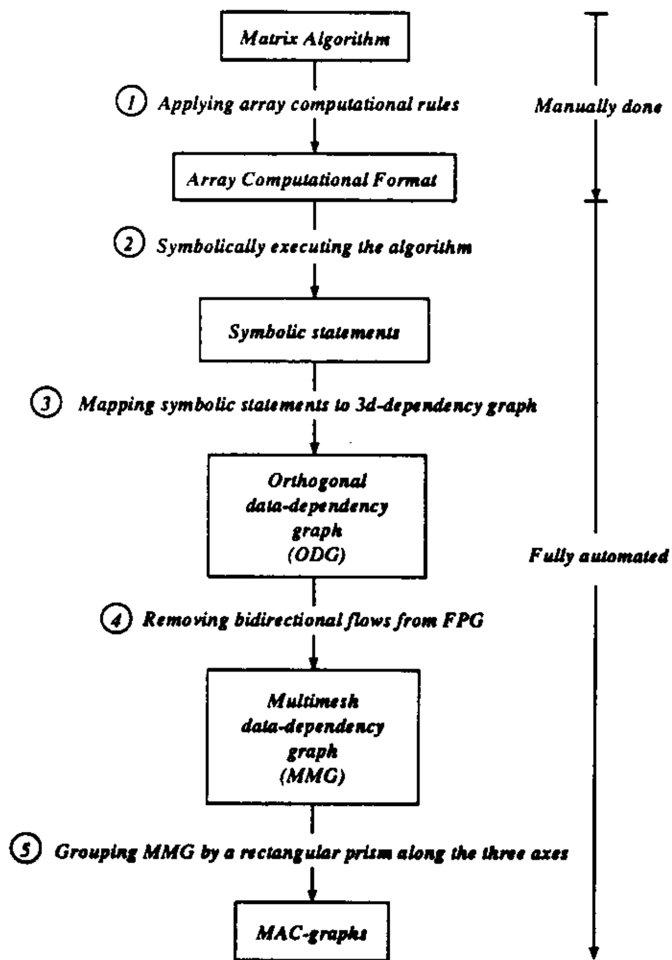


Figure 1.1: Steps taken to derive mesh array computational graphs

CHAPTER 2

Transformation of MAC-algorithms into Array Computational Format

To be used by MAMACG, a matrix algorithm must first be transformed from its conventional single thread form into an *array computational format* which is defined by the following rules:

1. At most three subscripted variables may appear in the right-hand-side expression of any assignment statement.

2. Each subscripted variable may appear on the left-hand-side expression only once (single assignment rule).

3. All 2-tuple subscripted variables must be expanded to 3-tuple subscripted variables without changing the semantics of the algorithm, where the added dimension reflects the order in time the subscripted variable is assigned a value. Suppose the following matrix algorithm were presented:

   for k := 1 **to** n **do begin**

   BODY

   end

   All 2-tuple subscripted variables inside *BODY* would be transformed to 3-tuple subscripted variables as follows:

(a) The variable $\alpha[i, j]$ appearing on the left-hand-side expression of a statement is expanded to $\alpha[i, j, k]$ since it refers to the variable $\alpha[i, j]$ of the k-th iteration being assigned a value.

(b) If a variable $\alpha[i, j]$ appears on the right-hand-side of a statement and appears on the left-hand-side expression of no previous statement of the current iteration, then it is expanded to $\alpha[i, j, k-1]$, since it refers to the value of the variable $\alpha[i, j]$ of the previous iteration.

(c) If a variable $\alpha[i, j]$ appears on the right-hand-side of a statement and it also appears on the left-hand-side expression of a previous statement of the current iteration, then it is expanded to $\alpha[i, j, k]$ since it refers to the value of the variable $\alpha[i, j]$ of the current iteration.

4. Input subscripted variables, $v[i, j, k]$, have $k = 0$.

5. Output subscripted variables are specified at the beginning of the algorithm.

The rest of this chapter demonstrates the transformations of the single thread forms of LU-decompostion and Warshall's transitive closure algorithms into their array computational formats. The transformations into AC formats are being done manually in this thesis but this step can also be automated. The next chapter describes how MAMACG maps array computational formats to three-dimensional orthogonal data-dependency graphs (*ODG*).

2.1 LU-decomposition

In the LU-decomposition computation, a given matrix **A** is decomposed into

$$\mathbf{A} = \mathbf{LU}$$

where **L** is a lower triangular matrix and **U** is an upper triangular matrix. The LU-decomposition algorithm can be written as:

```
\\ input: A[n,n]
\\ output: L[n,n], U[n,n]
var
  i, j, k: int;
begin
  for k := 1 to n do begin
    U[k,k] := 1/A[k,k];
    for j := k+1 to n do
      U[k,j] := A[k,j];
    for i := k+1 to n do
      L[i,k] := A[i,k] * U[k,k];
    for i := k+1 to n do
      for j := k+1 to n do
        A[i,j] := A[i,j] - L[i,k] * U[k,j];
  end
end
```

The tables below contain the transformations from the single thread computational format to the array computational format of the four statements appearing in the algorithm; all 2-tuple subscripted variables are expanded to 3-tuple subscripted variables using the technique described above.

single thread computational format

1.	$U[k,k] = 1/A[k,k]$
2.	$U[k,j] = A[k,j]$
3.	$L[i,k] = A[i,k] * U[k,k]$
4.	$A[i,j] = A[i,j] - L[i,k] * U[k,j]$

↓

array computational format

1.	$U[k,k,k] = 1/A[k,k,k-1]$
2.	$U[k,j,k] = A[k,j,k-1]$
3.	$L[i,k,k] = A[i,k,k-1] * U[k,k,k]$
4.	$A[i,j,k] = A[i,j,k-1] - L[i,k,k] * U[k,j,k]$

In statement 1, the left-hand-side expression $U[k,k]$ is transformed to $U[k,k,k]$ because it refers to the element $U[k,k]$ of the current iteration. The right-hand-side expression, $A[k,k]$, refers to the element of the previous iteration and, thus, is transformed to $A[k,k,k-1]$. Applying the current-previous iteration dependence relation to the remaining expressions results in the following code.

```
\\ input: A[n, n]
\\ output: L[n, n], U[n, n]
var
  i, j, k: int;
begin
  for k := 1 to n do begin
    U[k, k, k] := 1/A[k, k, k - 1];
    for j := k + 1 to n do
      U[k, j, k] := A[k, j, k - 1];
    for i := k + 1 to n do
      L[i, k, k] := A[i, k, k - 1] * U[k, k, k];
    for i := k + 1 to n do
      for j := k + 1 to n do
        A[i, j, k] := A[i, j, k - 1] - L[i, k, k] * U[k, j, k];
  end
end
```

2.2 Warshall Algorithm for Transitive Closure

Let G be a directed graph $G(V, E)$ where V is the set of vertices and E is the set of edges. The transitive closure of G, the graph G^*, has the same set of vertices as G. An edge (v_i, v_j) exists in G^* iff there exists a set of edges in G that forms a path connecting v_i to v_j.

The graph G is represented by an $n \times n$ matrix $\mathbf{A}$. The single thread computational format of the Warshall algorithm for transitive closure [4], shown below.

takes **A** as an input and produces $\mathbf{A}^*$—an $n \times n$ matrix that represents the graph G^*.

```
\\ input: A[n, n]
\\ output: A[n, n]
var
  i, j, k: int;
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        A[i, j] := A[i, j] ⊗ (A[i, k] ⊗ A[k, j]);
end
```

The variable $A[i, j]$ appearing in the left-hand-side expression of the single statement in the algorithm is expanded to $A[i, j, k]$ because it refers to the variable $A[i, j]$ of the current iteration being assigned a value. The variables $A[i, j]$, $A[i, k]$, and $A[k, j]$ appearing in the right-hand-side expression of the statement are expanded to $A[i, j, k-1]$, $A[i, k, k-1]$, and $A[k, j, k-1]$, respectively, because they refer to the values of $A[i, j]$, $A[i, k]$, and $A[k, j]$, respectively, of the previous iteration. The resulting array computational format of the Warshall algorithm for transitive closure is shown below.

```
\\ input: A[n, n]
\\ output: A[n, n]
var
  i, j, k: int;
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        A[i, j, k] := A[i, j, k − 1] ⊗(A[i, k, k − 1] ⊗ A[k, j, k − 1]);
end
```

CHAPTER 3

Derivation of the *ODG*

The automated transformation of a MAC-algorithm in array computational format to a three-dimensional *ODG* is performed by (i) symbolic execution of the algorithm to obtain symbolic statements over the given range of indices, and (ii) mapping of the symbolic statements onto the three-dimensional *ODG*.

3.1 Symbolic Execution

The symbolic execution of a matrix algorithm in array computational format consists of the following steps:

1. Call the procedure that represents the algorithm in a pseudo-Algol code given by the user. The arguments passed to the procedure are integer constants that describe the lower and upper bounds of the indices of the input matrices.

2. Bind the upper bound and lower bound variables to the integer constants passed from the caller.

3. Iterate the loop based on the values of the upper bound and lower bound variables.

4. Instantiate the 3-tuple index, $[i, j, k]$ where i, j, or k may be either a variable or an integer constant, to a 3-tuple constant index, $[i', j', k']$ where i', j', and k' are all integer constants.

5. Generate the symbolic statements.

Figure 3.1a shows the instantiation of the upper bound variable n to 3 in *lu-decomp(3)* and Figure 3.1b illustrates the corresponding symbolic statements.

Next we describe how symbolic statements are transformed into a three-dimensional *ODG*.

3.2 Mapping of Symbolic Statements Into *ODG*

Definition 1 An *ODG* is a three-dimensional Euclidean directed graph $G(V, E)$ in which each vertex represents a **computational node** (defined by Rule 1) and each directed edge represents a **data channel** (defined by Rules 3 and 4) carrying input or output data. ■

Figure 3.1c shows the *ODG* for *lu-decomp(3)*.

Rule 1 (Computational node) Each symbolic statement generates a unique **computational node** in the *ODG*. The left-hand-side expression of the symbolic statement is a subscripted variable $v[i, j, k]$ whose index, $[i, j, k]$, determines the coordinate where the node is situated in the *ODG*. The node is labeled $v[i, j, k]$. The right-hand-side expression of the symbolic statement determines the operation of the node. ■

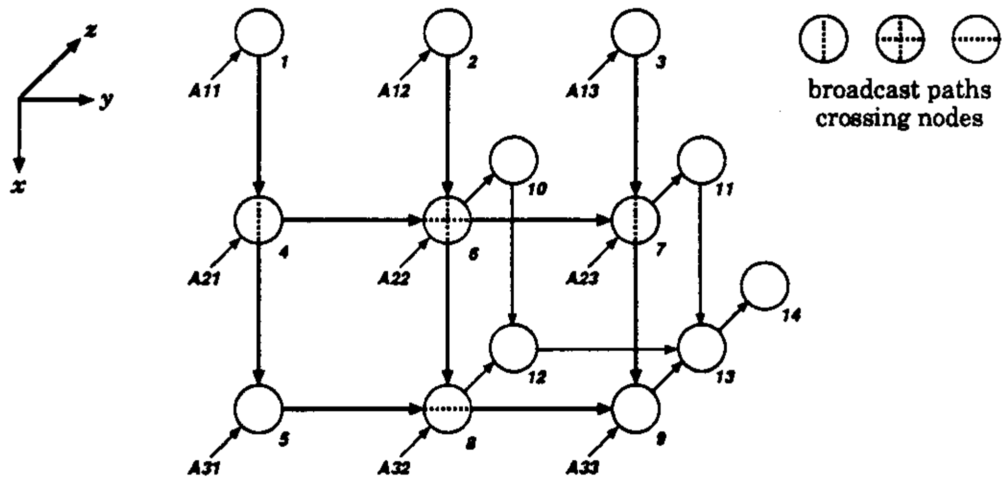
For example, the symbolic statement $L[2, 1, 1] := A[2, 1, 0] * U[1, 1, 1]$ generates the computational node $L[2, 1, 1]$ (labeled (4) in Figure 3.1c); it is placed at coor-

```
for k := 1 to 3 do
  begin
1.  U[k,k,k] := 1/A[k,k,k-1];
    for j := k+1 to 3 do
2.    U[k,j,k] := A[k,j,k-1];
    for i := k+1 to 3 do
3.    L[i,k,k] := A[i,k,k-1]*U[k,k,k];
    for i := k+1 to 3 do
      for j := k+1 to 3 do
4.      A[i,j,k] := A[i,j,k-1] - L[i,k,k]*U[k,j,k];
  end
```

a) lu-decomp code after instantiating n to 3

k = 1	k = 2	k = 3
(1) 1. U[1,1,1] := 1/A[1,1,0]	(10) 1. U[2,2,2] := 1/A[2,2,1]	(14) 1. U[3,3,3] := 1/A[3,3,2]
(2) 2. U[1,2,1] := A[1,2,0]		
(3) U[1,3,1] := A[1,3,0]	(11) 2. U[2,3,2] := A[2,3,1]	
(4) 3. L[2,1,1] := A[2,1,0] * U[1,1,1]		
(5) L[3,1,1] := A[3,1,0] * U[1,1,1]	(12) 3. L[3,2,2] := A[3,2,1] * U[2,2,2]	
(6) 4. A[2,2,1] := A[2,2,0] - L[2,1,1]*U[1,2,1]		
(7) A[2,3,1] := A[2,3,0] - L[2,1,1]*U[1,3,1]		
(8) A[3,2,1] := A[3,2,0] - L[3,1,1]*U[1,2,1]	(13) 4. A[3,3,2] := A[3,3,1] - L[3,2,2]*U[2,3,2]	
(9) A[3,3,1] := A[3,3,0] - L[3,1,1]*U[1,3,1]		

b) Symbolic statements generated after 1st, 2nd and 3rd iteration



c) Orthogonal data-dependency graph

Figure 3.1: Steps taken in deriving *lu-decomp(3)'s ODG*

dinate $\langle 2,1,1\rangle$ of the three-dimensional *ODG*. The node performs the operation $A[2,1,0] * U[1,1,1]$.

Definition 2 The *size* of the *ODG* is the smallest cube enclosing all the computational nodes generated by the symbolic statements. ■

Rule 2 (Data flow) Data can flow parallel to the X axis, Y axis, or Z axis. Relative to a node, there are five types of flows parallel to the X axis, whose set is denoted by Ω_X,

$$\Omega_X = \{ \cdot , \cdot , \cdot , \cdot , \cdot \},$$

five types of flows parallel to the Y axis, whose set is denoted by Ω_Y,

$$\Omega_Y = \{ \cdot , \cdot , \cdot , \cdot , \cdot \}, \text{ and}$$

four types of flows parallel to the Z axis, whose set is denoted by Ω_Z,

$$\Omega_z = \{ \cdot , \cdot , \cdot , \cdot \}.$$

Let $x \in \Omega_X, y \in \Omega_Y$, and $z \in \Omega_Z$. A *data flow* relative to a node is defined as the superimposition of x over y over z centered at the nodes (denoted by circles) of x, y, and z. ■

Rule 3 (Inputs) An input to a node situated at coordinate $\langle i,j,k\rangle$ comes from a node situated at coordinate $\langle i',j',k'\rangle$ if a subscripted variable with index $[i',j',k']$

appears in the right-hand-side expression of the symbolic statement that generated the node situated at coordinate $\langle i, j, k \rangle$. The input is called

$$\textbf{x-input} \quad \text{if} \quad i \neq i', j = j', \text{and } k = k',$$

$$\textbf{y-input} \quad \text{if} \quad i = i', j \neq j', \text{and } k = k',$$

$$\textbf{z-input} \quad \text{if} \quad i = i', j = j', \text{and } k \neq k'.$$

A computational node may have at most three inputs. ■

Rule 4 (Outputs) A node situated at coordinate $\langle i, j, k \rangle$ sends output to the node situated at coordinate $\langle i', j', k' \rangle$ if a subscripted variable with index $[i, j, k]$ appears in the right-hand-side expression of the symbolic statement that generated the node situated at coordinate $\langle i', j', k' \rangle$. An output is called

$$\textbf{x-output} \quad \text{if} \quad i \neq i', j = j', \text{and } k = k',$$

$$\textbf{y-output} \quad \text{if} \quad i = i', j \neq j', \text{and } k = k',$$

$$\textbf{z-output} \quad \text{if} \quad i = i', j = j', \text{and } k' - k = 1.$$

Data on an output arc can be either generated within the node or transmitted from another node. ■

In Figure 3.1c, the node located at $\langle 2, 2, 1 \rangle$ (labeled (6) in the figure) has a x-input coming from the node located at $\langle 1, 2, 1 \rangle$ (labeled (2) in the figure), an y-input coming from the node located at $\langle 2, 1, 1 \rangle$ (labeled (4) in the figure), and a z-input coming from the external node located at $\langle 2, 2, 0 \rangle$.

Rule 5 (Data Flow Delay) Data are passed between adjacent nodes in unit time. Data flow from a computational node to the node(s) in the five adjacent locations—positive z-axis, positive and negative y-axis, and positive and negative x-axis. If data are required to pass through an adjacent location that is not occupied by a node, a **delay node** is created at that location to relay the data. ■

Definition 3 A node, located at $\langle i, j, k \rangle$, is called a **broadcast node** if it sends the same data to nodes located at $\langle i'_1, j'_1, k'_1 \rangle$, ..., $\langle i'_n, j'_n, k'_n \rangle$ where $n > 1$ and $k = k'_1 = k'_n$, that is, to more than one node located in the same plane of the *ODG*. A node that sends the same data to multiple destinations through the x-output is called an **x-broadcast node**; a node that sends the same data to multiple destinations via the y-output is called a **y-broadcast node**; and, similarly, a node that sends the same data to multiple destinations via the x-output and y-output is called an **xy-broadcast node.** ■

Figure 3.1c shows that the computational nodes (1), (2) and (3) are x-broadcast nodes that feed the same data to the set of nodes $\{(4),(5)\}$, $\{(6),(8)\}$, and $\{(7),(9)\}$, respectively; on the other hand, the computational nodes (4) and (5) are y-broadcast nodes that feed data to the set of nodes $\{(6),(7)\}$ and $\{(8),(9)\}$, respectively.

Definition 4 A path that carries the same data to several nodes along the X axis is called an **x-broadcast path**; a path that carries the same data to several nodes along the Y axis is called a **y-broadcast path**; otherwise, when it carries different data to each node in the path, it is called a **regular path.** ■

In Figure 3.1c, the broadcast paths going through the computational nodes are denoted by the broken lines crossing the nodes.

Definition 5 A node that lays on an x-broadcast path is called an **x-transmitting node**; a node that lays on an y-broadcast path is called an **y-transmitting node.** ■

Mapping Procedure

From the rules and definitions described so far, the procedure for mapping symbolic statements into the *ODG* consists of processing each symbolic statement, $\alpha[i,j,k] :=$ *rhs-exp*, into a computational node as follows:

1. Generate a node and place it at location $\langle i,j,k \rangle$.
2. For each subscripted variable $\beta[i',j',k']$ in *rhs-exp*, generate an appropriate x-input, y-input, or z-input to the node by applying Rule 3.
3. For each subscripted variable $\beta[i',j',k']$ in *rhs-exp*, generate an appropriate x-output, y-output, or z-output to the node located at $\langle i',j',k' \rangle$ by applying Rule 4. Note that the node located at $\langle i',j',k' \rangle$ can be either an external input or an existing computational node.
4. Detect and report violation of data flow by applying Rule 2.
5. Classify a node as a *broadcast node* if it satisfies Definition 3.

The derivations of the *ODGs* for *lu-decomp(3)* and *warshall(3)* is covered next.

3.3 Derivation of *lu-decomp(3)'s ODG*

Figures 3.1a, 3.1b, and 3.1c show the steps taken in deriving *lu-decomp(3)'s ODG*. In Figure 3.1a, the simple variable n, which determines the upper bound of the subscripted variables appearing inside the algorithm, is instantiated to 3. Figure 3.1b shows the symbolic statements generated after the first, second, and third iterations. Figure 3.1c shows *lu-decomp(3)'s ODG* after applying the procedure for mapping symbolic statements into *ODG*.

Rule 1 maps the symbolic statements generated after the first iteration into computational nodes $U[1,1,1]$, $U[1,2,1]$, $U[1,3,1]$, $L[2,1,1]$, $L[3,1,1]$, $A[2,2,1]$, $A[2,3,1]$, $A[3,2,1]$, and $A[3,3,1]$, which occupy coordinates $\langle 1,1,1\rangle$, $\langle 1,2,1\rangle$, $\langle 1,3,1\rangle$, $\langle 2,1,1\rangle$, $\langle 3,1,1\rangle$, $\langle 2,2,1\rangle$, $\langle 2,3,1\rangle$, $\langle 3,2,1\rangle$, and $\langle 3,3,1\rangle$, respectively. The other computational nodes are mapped similarly.

To simplify the discussion, all the nodes are relabeled $(1),(2),\ldots,(14)$ as shown in Figure 3.1b and Figure 3.1c. Rules 3 and 4 (**inputs** and **outputs**) are used to map the symbolic statements into input and output arcs. By rule 4, node (1) sends output to nodes (4) and (5) since its label, $U[1,1,1]$, appears on the right-hand-side expressions of the symbolic statements that generated computational nodes (4) and (5). Node (1), by definition 3, is an **x-broadcast** node since it sends data to multiple nodes parallel to the X axis. By definition 4, the path that carries data from node (1) to nodes (4) and (5) is called a **broadcast path** since it carries the same data to multiple nodes.

Similarly, node (2) sends its output to nodes (6) and (8) since its label appears on the right-hand-side expressions of the symbolic statements that generated computational nodes (6) and (8). Node (2) is also an **x-broadcast** node. The

path that carries data from node (2) to nodes (6) and (8) is a **broadcast path.**

$A[1,1,0]$ is an external input because the third element of its 3-tuple index has value 0. Node (1) receives input from the external input $A[1,1,0]$, by Rule 3, since $A[1,1,0]$ appears on the right-hand-side expression of the symbolic statement that generated the computational node (1). This particular input is called a **z-input** since it flows parallel to the Z axis.

The characteristics of the other arcs are derived in the same manner using the definitions and rules described in Section 3.2.

3.4 Derivation of *warshall(3)'s ODG*

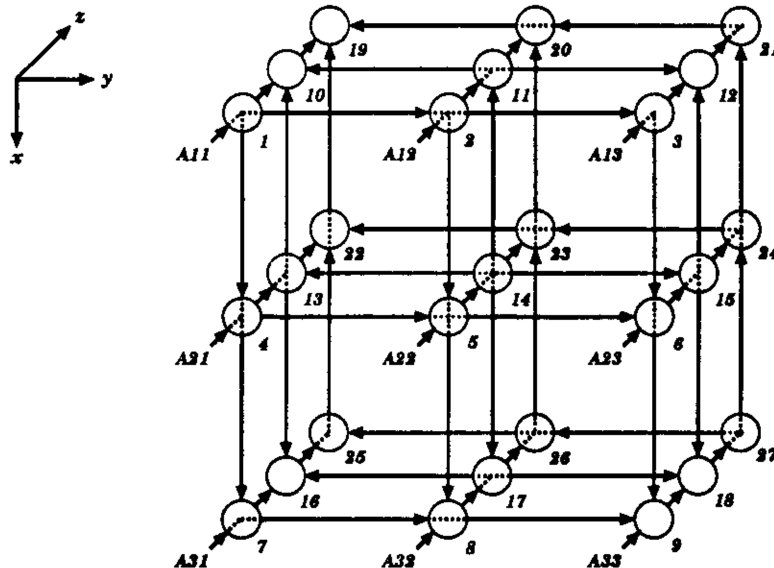
The derivation of *warshall(3)'s ODG* proceeds similarly to that in *lu-decomp(3)*; Figure 3.2a, 3.2b, and 3.2c show the corresponding steps.

```
for k := 1 to 3 do
  for i := 1 to 3 do
    for j := 1 to 3 do
      A[i,j,k] := A[i,j,k-1] xor (A[i,k,k-1] xor A[k,j,k-1]);
```

a) Warshall code after instantiating n to 3

```
(1) A[1,1,1] := A[1,1,0] xor (A[1,1,0] xor A[1,1,0])
(2) A[1,2,1] := A[1,2,0] xor (A[1,1,0] xor A[1,2,0])
(3) A[1,3,1] := A[1,3,0] xor (A[1,1,0] xor A[1,3,0])
(4) A[2,1,1] := A[2,1,0] xor (A[2,1,0] xor A[1,1,0])
(5) A[2,2,1] := A[2,2,0] xor (A[2,1,0] xor A[1,2,0])
(6) A[2,3,1] := A[2,3,0] xor (A[2,1,0] xor A[1,3,0])
(7) A[3,1,1] := A[3,1,0] xor (A[3,1,0] xor A[1,1,0])
(8) A[3,2,1] := A[3,2,0] xor (A[3,1,0] xor A[1,2,0])
(9) A[3,3,1] := A[3,3,0] xor (A[3,1,0] xor A[1,3,0])
(10) A[1,1,2] := A[1,1,1] xor (A[1,2,1] xor A[2,1,1])
(11) A[1,2,2] := A[1,2,1] xor (A[1,2,1] xor A[2,2,1])
(12) A[1,3,2] := A[1,3,1] xor (A[1,2,1] xor A[2,3,1])
(13) A[2,1,2] := A[2,1,1] xor (A[2,2,1] xor A[2,1,1])
(14) A[2,2,2] := A[2,2,1] xor (A[2,2,1] xor A[2,2,1])
(15) A[2,3,2] := A[2,3,1] xor (A[2,2,1] xor A[2,3,1])
(16) A[3,1,2] := A[3,1,1] xor (A[3,2,1] xor A[2,1,1])
(17) A[3,2,2] := A[3,2,1] xor (A[3,2,1] xor A[2,2,1])
(18) A[3,3,2] := A[3,3,1] xor (A[3,2,1] xor A[2,3,1])
(19) A[1,1,3] := A[1,1,2] xor (A[1,3,2] xor A[3,1,2])
(20) A[1,2,3] := A[1,2,2] xor (A[1,3,2] xor A[3,2,2])
(21) A[1,3,3] := A[1,3,2] xor (A[1,3,2] xor A[3,3,2])
(22) A[2,1,3] := A[2,1,2] xor (A[2,3,2] xor A[3,1,2])
(23) A[2,2,3] := A[2,2,2] xor (A[2,3,2] xor A[3,2,2])
(24) A[2,3,3] := A[2,3,2] xor (A[2,3,2] xor A[3,3,2])
(25) A[3,1,3] := A[3,1,2] xor (A[3,3,2] xor A[3,1,2])
(26) A[3,2,3] := A[3,2,2] xor (A[3,3,2] xor A[3,2,2])
(27) A[3,3,3] := A[3,3,2] xor (A[3,3,2] xor A[3,3,2])
```

(b) Symbolic statements generated at iterations 1, 2 and 3



x-bidirectional nodes: (2) and (8)
y-bidirectional nodes: (4) and (6)
xy-bidirectional node: (5)

negative-x nodes: (11) and (12)
negative-y nodes: (13) and (16)
negative-xy node: (10)

x-bidirectional nodes: (12) and (15)
y-bidirectional nodes: (16) and (17)
xy-bidirectional node: (18)

negative-x nodes: (21) and (24)
negative-y nodes: (25) and (26)
negative-xy nodes: (19), (20), (22) and (23)

(c) Orthogonal data-dependency planes generated at iterations 1, 2 and 3

Figure 3.2: Steps taken in deriving *warshall(3)'s ODG*

CHAPTER 4

Derivation of the *MMG* and *MAC-graph*

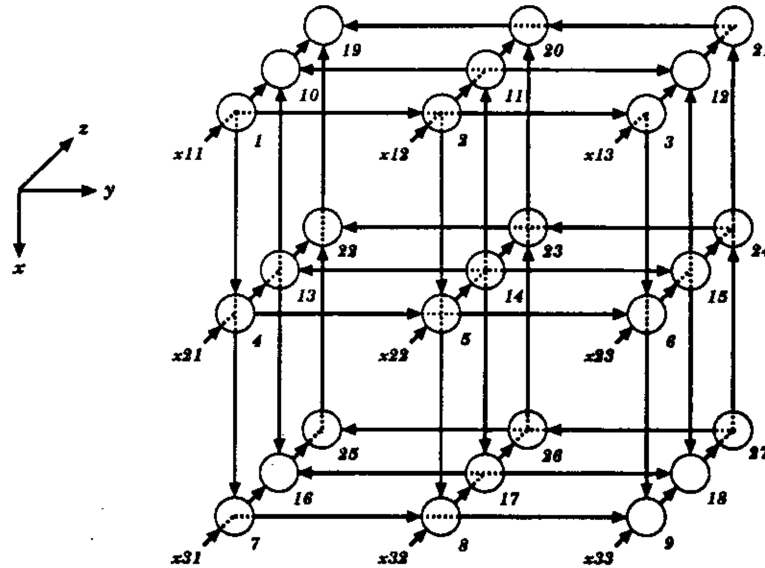
As the basis for efficient mesh array designs, the MMG method [1] introduces a three-dimensional multimesh graph representation—the *MMG.* To obtain the *MMG* from the corresponding *ODG*, bidirectional flows must be removed. Since *lu-decomp(3)'s ODG* does not contain bidirectional flow, its *MMG* is its *ODG.* On the other hand, since *warshall(3)'s ODG* contains bidirectional flows a technique for removing bidirectional flows must be applied to transform its *ODG* into its *MMG.* The rest of this chapter defines the properties of bidirectional flows and describes a technique for removing them from the *ODG.* The generation of mesh array computational graph (*MAC-graph*) from the resulting multimesh graph (*MMG*) is then described.

4.1 Definitions of Bidirectional Flows

Definition 6 An x-broadcast node is a **x-bidirectional node** $n[i,j,k]$ if one of its outputs is directed to a computational node $o[i',j,k]$ located on its negative x side ($i' < i$). ■

Definition 7 A y-broadcast node is a **y-bidirectional node** $n[i,j,k]$ if one of its outputs is directed to a computational node $o[i,j',k]$ located on its negative y side ($j' < j$). ■

Definition 8 An xy-broadcast node $n[i,j,k]$ is an **xy-bidirectional node** if one of its outputs is directed to a computational node $o'[i',j,k]$ located on its negative x side ($i' < i$) and another output is directed to a computational node $o''[i,j'',k]$ located on its negative y side ($j'' < j$). ∎



x-bidirectional nodes: (11) and (17)
y-bidirectional nodes: (13) and (15)
xy-bidirectional node: (14)

negative-x nodes: (11) and (12)
negative-y nodes: (13) and (16)
negative-xy node: (10)

x-bidirectional nodes: (21) and (24)
y-bidirectional nodes: (25) and (26)
xy-bidirectional node: (27)

negative-x nodes: (21) and (24)
negative-y nodes: (25) and (26)
negative-xy nodes: (19), (20), (22) and (23)

Figure 4.1: *Warshall(3)'s ODG*

Examples of bidirectional nodes are shown in Figure 4.1. Node (11) located at $\langle 1,2,2\rangle$ is a y-bidirectional node since one of its outputs goes to node (10) at $\langle 1,1,2\rangle$ which appears on its negative y side; node (13) at $\langle 2,1,2\rangle$ is a x-bidirectional node since one of its output goes to node (10) which appears on its negative x side; node (14) is a xy-bidirectional node since one of its output goes to node (11) which appears on its negative x side and another output goes to

node (13) which appears on its negative y side.

Definition 9 A node located at $\langle i,j,k\rangle$ that receives data from two nodes located at $\langle i',j,k\rangle$ and $\langle i,j'',k\rangle$ and resides on the negative x side ($i < i'$) and negative y side ($j < j''$) of these nodes, respectively, is called an **xy-negative node.** ■

Definition 10 A node located at $\langle i,j,k\rangle$ that receives data from another node $\langle i',j,k\rangle$ and resides on its negative x side ($i < i'$) is called an **x-negative node.** ■

Definition 11 A node located at $\langle i,j,k\rangle$ that receives data from another node $\langle i,j',k\rangle$ and resides on its negative y side ($j < j'$) is called a **y-negative node.** ■

Examples of negative nodes are shown in Figure 4.1. Node (11) located at $\langle 1,2,2\rangle$ is an x-negative node because it receives data from node (14) and appears on the negative x side of that node; node (13) is an y-negative node because it receives data from node (14) and appears on the negative y side of that node; node (10) is a xy-negative node since it receives data from node (11) and node (13) and appears on the negative x side of (13) and on the negative y side of (11).

Observation 1 A three-dimensional *ODG* has no bidirectional flows if it contains no x-negative node, no y-negative node, and no xy-negative node; otherwise, it has bidirectional flows. ■

As stated earlier, the *ODG* for *lu-decomp(3)* has no bidirectional flows, and, therefore, it also is the *MMG*. The *ODG* for *Warshall(3)*, on the other hand,

has bidirectional flows and must be transformed into an *MMG*. A technique for removing bidirectional flows is discussed next.

4.2 Removal of Bidirectional Flows in *ODG*

To remove bidirectional flows in the *ODG*, MAMACG moves all x-negative nodes to the positive x side, all y-negative nodes to the positive y side, and all xy-negative nodes to the positive xy side. The procedure used for these purposes is discussed next, where we let the size of the *ODG* be $\alpha \times \beta \times \gamma$.

Removal of X-negative Nodes

An x-negative node located at $\langle i, j, k \rangle$ of the *ODG*, can be moved to the new location $\langle \alpha + i, j, k \rangle$ if its x-input, y-input and z-input can be redirected to the new location.

For example, the x-negative node (11) located at $\langle 1, 2, 2 \rangle$ in Figure 4.1 can be moved to the new location $\langle 4, 2, 2 \rangle$ if its x-input (from node (14) located at $\langle 2, 2, 2 \rangle$), y-input (there is none) and z-input (from node (2) located at $\langle 1, 2, 1 \rangle$) can be redirected to the new location $\langle 4, 2, 2 \rangle$.

Redirect the X-input

The x-input that flows from the node located at $\langle i', j, k \rangle$ can be directed to the new location $\langle \alpha + i, j, k \rangle$ if the following condition holds:

- The node located at $\langle i', j, k \rangle$ is an x-transmitting node and the path from $\langle i', j, k \rangle$ to $\langle \alpha + i, j, k \rangle$ is a broadcast path.

In Figure 4.1, the x-input of node (11) comes from an x-transmitting node (14) and the path is a broadcast path. Therefore, it can be redirected to the new location $\langle 4, 2, 2 \rangle$.

Redirect the Y-input

The y-input that flows from $\langle i, j', k \rangle$ can be directed to the new location $\langle \alpha + i, j, k \rangle$ if one of the following conditions holds:

- The node located at $\langle i, j', k \rangle$ is a constant and has only the y-output.
- The node located at $\langle i, j', k \rangle$ was already moved to the new location $\langle \alpha + i, j', k \rangle$.
- The node located at $\langle i, j', k \rangle$ is an xy-broadcast node. In this case, a delay node is created at location $\langle \alpha + i, j', k \rangle$ with the x-input transmitted from the node located at $\langle i, j', k \rangle$, with no y-input, no z-input, and with the y-output transmitted to the node located at $\langle \alpha + i, j, k \rangle$. There must exist broadcast paths from the node located at $\langle i, j', k \rangle$ to the node at $\langle \alpha + i, j', k \rangle$ and from the node at $\langle \alpha + i, j', k \rangle$ to the node at $\langle \alpha + i, j, k \rangle$.
- The node located at $\langle i, j', k \rangle$ can be moved to the new location $\langle \alpha + i, j', k \rangle$.

Redirect the Z-input

The z-input that flows from $\langle i, j, k - 1 \rangle$ can be directed to the new location $\langle \alpha + i, j, k \rangle$ if one of the following conditions holds:

- The node located at $\langle i, j, k - 1 \rangle$ is a constant and has only the z-output.
- The node located at $\langle i, j, k - 1 \rangle$ was moved to the new location $\langle \alpha + i, j, k - 1 \rangle$.

- The node located at $\langle i, j, k-1\rangle$ x-broadcasts the z-input. In this case, the node is moved to $\langle \alpha + i, j, k-1\rangle$, a delay node is created at the location $\langle i, j, k-1\rangle$ inheriting the z-input and transmitting it to the node located at $\langle \alpha + i, j, k-1\rangle$.

- The node located at $\langle i, j, k-1\rangle$ can be moved to the new location $\langle \alpha + i, j, k-1\rangle$.

In Figure 4.1, the z-input of node (11) came from node (2) (located at $\langle 1,2,1\rangle$) and this node can be moved to the new location $\langle 4,2,1\rangle$ by creating a delay node at location $\langle 1,2,1\rangle$ to relay the external input from node $\langle 1,2,0\rangle$ along the x-broadcast path to node (2) at the new location $\langle 4,2,1\rangle$.

Summary of rules

We now summarize the rules used by MAMACG in removing x-negative and y-negative nodes.

Let *node′* be the new location, and *z-input′*, *y-input′*, and *x-input′* be the new set of inputs and *output′* be the new output. If

$$
\begin{aligned}
\textit{node} &= \langle i, j, k\rangle \\
\textit{z-input} &= \langle i, j, k-1\rangle \\
\textit{y-input} &= \langle i, j', k\rangle \\
\textit{x-input} &= \langle i', j, k\rangle \\
\textit{output} &= \{\langle i_1, j_1, k_1\rangle, \ldots, \langle i_n, j_n, k_n\rangle\}
\end{aligned}
$$

then

$$
\begin{aligned}
node' &= \langle \alpha + i, j, k \rangle \\
z\text{-}input' &= \langle \alpha + i, j, k-1 \rangle \\
y\text{-}input' &= \langle \alpha + i, j', k \rangle \\
x\text{-}input' &= \langle i', j, k \rangle \\
output' &= \{\langle \alpha + i_1, j_1, k_1 \rangle, \ldots, \langle \alpha + i_n, j_n, k_n \rangle\}
\end{aligned}
$$

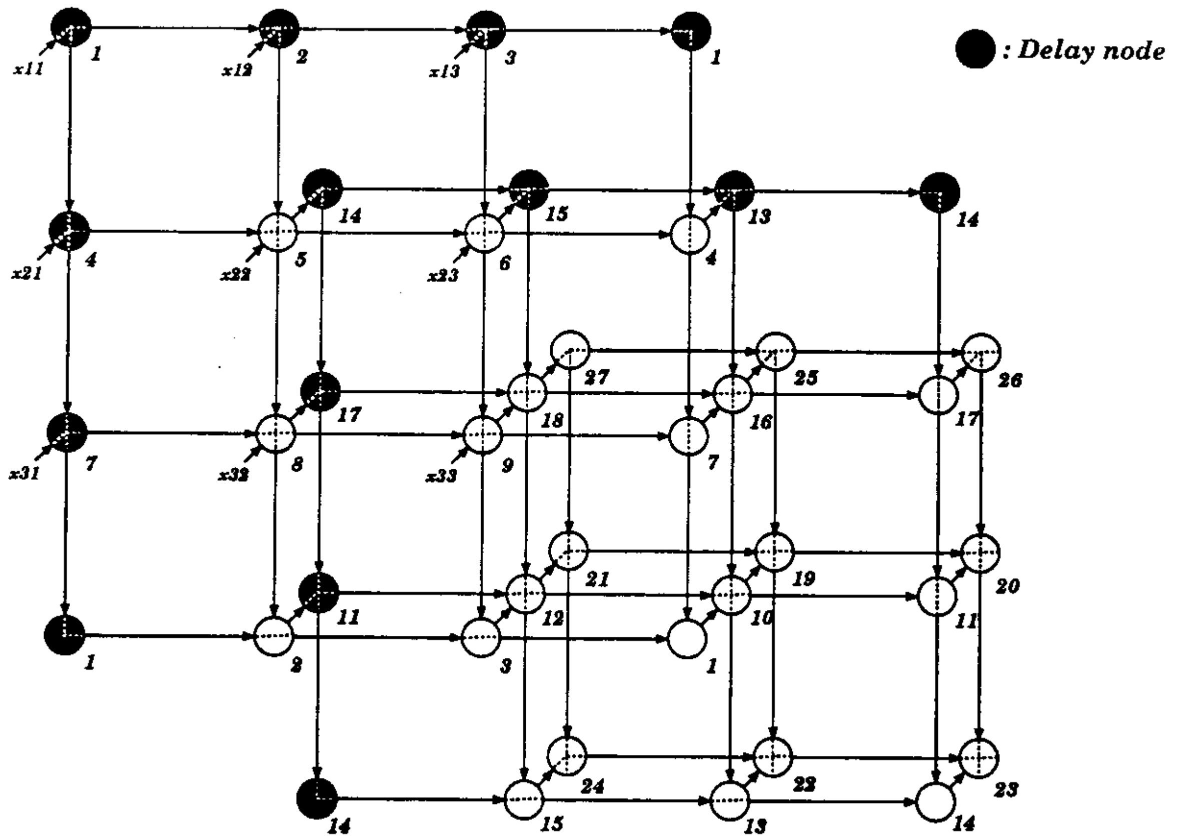


Figure 4.2: Graph generated after the removal of the negative nodes

As the removal of y-negative nodes is isomorphic to the removal of negative-x nodes, the corresponding discussion is omitted.

In Figure 4.1, the y-negative nodes (13) (located at $\langle 2, 1, 2\rangle$) and (16) (located at $\langle 3, 1, 2\rangle$) must be moved to the new locations $\langle 2, 4, 2\rangle$ and $\langle 3, 4, 2\rangle$, respectively. When node (13) is moved to the new location, it requires that node (4) be moved to the new location $\langle 2, 4, 1\rangle$. In moving node (4) to its new location, a delay node is created and placed at the old location where node (4) resided so that data coming from the external input from $\langle 2, 1, 0\rangle$ can be relayed to node (4) at the new location.

Removal of XY-bidirectional Flow

An xy-negative node located at $\langle i, j, k\rangle$ can be moved to the new location $\langle \alpha + i, \beta + j, k\rangle$ if its inputs—x-input, y-input and z-input—can be directed to the new location $\langle \alpha + i, \beta + j, k\rangle$.

Redirect the X-input

The x-input located at $\langle i', j, k\rangle$ can be directed to the new location $\langle \alpha + i, \beta + j, k\rangle$ if either of the following conditions holds:

1. The node located at $\langle i', j, k\rangle$ is an x-transmitting and the node located at $\langle i', j, k\rangle$ was moved to the new location $\langle i', \beta + j, k\rangle$.

2. The node located at $\langle i', j, k\rangle$ is a xy-broadcast node. In this case, a delay node is created at location $\langle i', j, k\rangle$ with no x-input, the y-input transmitted from the node located at $\langle i', \beta + j, k\rangle$, no z-input, and one x-output transmitted to the node located at $\langle \alpha + i, \beta + j, k\rangle$.

Redirect the Y-input

As the redirection of y-input located at $\langle i, j', k\rangle$ to the new location $\langle \alpha+i, \beta+j', k\rangle$ is isomorphic to the redirection of x-input located at $\langle i', j, k\rangle$ to the new location $\langle \alpha + i', \beta + j, k\rangle$, the corresponding discussion is omitted.

Redirecting the Z-input

The z-input that flows from $\langle i, j, k-1\rangle$ can be directed to the new location $\langle \alpha + i, \beta + j, k\rangle$ if one of the following conditions holds:

- The node located at $\langle i, j, k-1\rangle$ is a constant and only has the z-output.
- The node located at $\langle i, j, k-1\rangle$ xy-broadcasts the z-input. In this case, the node is moved to the new location $\langle \alpha + i, \beta + j, k-1\rangle$, three delay nodes are created at locations $\langle i, j, k-1\rangle$, $\langle \alpha + i, j, k-1\rangle$, and $\langle i, \beta + j, k-1\rangle$. The delay node located at $\langle i, j, k-1\rangle$ receives input from the node located at $\langle i, j, k-2\rangle$ and transmits it to the delay nodes located at $\langle \alpha + i, j, k-1\rangle$ and $\langle i, \beta + j, k-1\rangle$ which are then transmitted to the node located at $\langle \alpha + i, \beta + j, k-1\rangle$.
- The node located at $\langle i, j, k-1\rangle$ can be moved to the new location $\langle \alpha + i, \beta + j, k-1\rangle$.

Summary of rules

If the node located at $\langle i, j, k\rangle$ cannot be moved to a new location, then MAMACG cannot map the algorithm to a mesh array computational graph. Otherwise, let *node'* be the new location, and *z-input'*, *y-input'*, and *x-input'* be the new set of

inputs and *output'* be the new output, and if

$$\begin{aligned} node &= \langle i, j, k \rangle \\ z\text{-}input &= \langle i, j, k-1 \rangle \\ y\text{-}input &= \langle i, j', k \rangle \\ x\text{-}input &= \langle i', j, k \rangle \\ output &= \{\langle i_1, j_1, k_1 \rangle, \ldots, \langle i_n, j_n, k_n \rangle\} \end{aligned}$$

then

$$\begin{aligned} node' &= \langle \alpha + i, \beta + j, k \rangle \\ z\text{-}input' &= \langle \alpha + i, \beta + j, k-1 \rangle \\ y\text{-}input' &= \langle \alpha + i, j', k \rangle \\ x\text{-}input' &= \langle i', \beta + j, k \rangle \\ output' &= \{\langle i_1 + \alpha, j_1 + \beta, k_1 \rangle, \ldots, \langle i_n + \alpha, j_n + \beta, k_n \rangle\} \end{aligned}$$

4.3 Regularize the Last Plane

The graph resulting from application of the technique for removal of negative nodes might have an irregular last plane, as shown in Figure 4.2. The different pattern of the upper and left bordering nodes of the last plane and that of the upper and left bordering nodes of the other planes makes it more difficult to schedule and analyze the graph.

The regularizing process involves regularizing the bordering nodes, which include the corner top-left node, the leftmost nodes, and the topmost nodes of the last plane, so that they follow the same pattern:

- If the corner top-left node of the first plane was moved to the new xy location, then move the corner top-left node of the last plane to the new xy location. Likewise, if the corner top-left node of the first plane was moved to the new y location then move the corner top-left node of the last plane to the new y location. If the corner top-left node of the first plane was moved to the new x location then move the corner top-left node of the last plane to the new x location. Otherwise, stay at the same location.

- If the leftmost nodes of the first plane were moved to the new x location, then move all the leftmost nodes of the last plane to the new x location. If the leftmost nodes of the first plane stay at the same location then stay at the same location. Anything else triggers an error since no other case is possible.

- If the topmost nodes of the first plane were moved to the new y location, then move all the topmost nodes of the last plane to the new y location. If the topmost nodes of the first plane stay at the same location then stay at the same location. Anything else causes an error since no other case is possible.

In Figure 4.2, the first plane's corner top-left node (1) was moved from location $\langle 1,1,1 \rangle$ to location $\langle 4,4,1 \rangle$ and the last plane's top-left node (27) remained at location $\langle 3,3,3 \rangle$. Figure 4.3 shows node (27) after being moved to location $\langle 5,5,3 \rangle$.

The first plane's leftmost nodes, (4) and (7), of Figure 4.2 were moved to the new locations $\langle 2,4,1 \rangle$ and $\langle 3,4,1 \rangle$, respectively, while the last plane's leftmost nodes, (21) and (24), were not moved. Figure 4.3 shows nodes (21) and (24) after

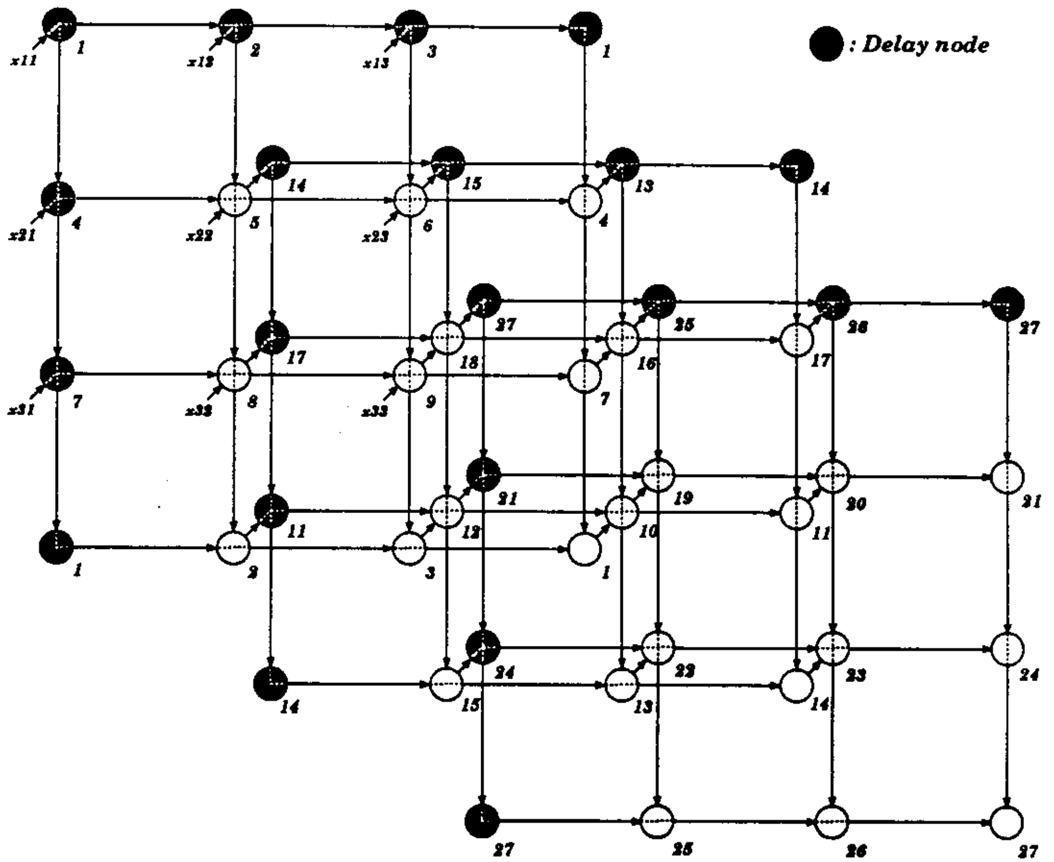


Figure 4.3: *MMG* generated by *Warshall(3)*

being moved to $\langle 4,6,3\rangle$ and $\langle 5,6,3\rangle$, respectively.

The first plane's topmost nodes, (2) and (3), of Figure 4.2 were moved to the new locations $\langle 4,2,1\rangle$ and $\langle 4,3,1\rangle$, respectively, while the last plane's topmost nodes, (25) and (26), were not moved. Figure 4.3 shows nodes (25) and (26) after begin moved to $\langle 6,4,3\rangle$ and $\langle 6,5,3\rangle$, respectively.

The resulting *Warshall(3)'s MMG* is shown in Figure 4.3. The generation of mesh array computational graphs (*MAC-graphs*) of a *MMG* is illustrated next.

4.4 Generation of the Mesh Array Computational Graph

The three-dimensional *MMG* is transformed into one of many possible mesh array computational graphs by grouping the primitive nodes along one of the three axes by rectangular prisms of base size $p \times q$, where p is the factors of the projected width size and q is the factors of the projected length size. Currently, MAMACG only considers a 1×1 prism.

The set of primitive nodes and links within one group is called a mesh array computational node (*MAC-node*) and the set of *MAC-nodes* is called the *MAC-graph.*

The resulting *MAC-graphs* for *lu-decomp(3)* are shown in Figure 4.4, Figure 4.5, and Figure 4.6.

In Figure 4.4, nine MAC-nodes, labeled $M1, \ldots, M9$, were formed after the projection of a 1×1 prism along the Z axis. $M9$ contains computational nodes located at $\langle 3,3,1\rangle$, $\langle 3,3,2\rangle$ and $\langle 3,3,3\rangle$; $M8$ contains nodes located at $\langle 3,2,1\rangle$ and $\langle 3,2,2\rangle$; $M7$ contains node located at $\langle 3,1,1\rangle$.

In Figure 4.5, six MAC-nodes, labeled $M1, \ldots, M6$, were formed after the

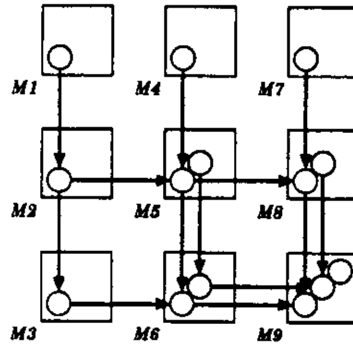


Figure 4.4: *MAC-graph* generated by grouping along the Z-axis of *LU-decomp(3)'s MMG*

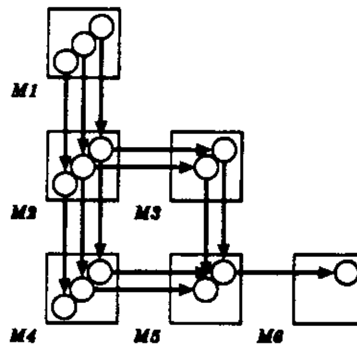


Figure 4.5: *MAC-graph* generated by grouping along the Y-axis of *LU-decomp(3)'s MMG*

projection of a 1×1 prism along the Y axis. $M1$ contains computational nodes located at $\langle 1,1,1\rangle$, $\langle 1,2,1\rangle$ and $\langle 1,3,1\rangle$; $M2$ contains nodes located at $\langle 2,1,1\rangle$. $\langle 2,2,1\rangle$, and $\langle 2,3,1\rangle$; $M3$ contains nodes located at $\langle 2,2,3\rangle$ and $\langle 2,3,2\rangle$.

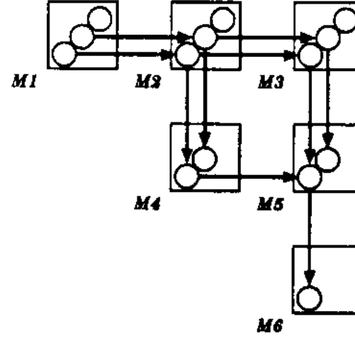


Figure 4.6: *MAC-graph* generated by grouping along the X-axis of *LU-decomp(3)'s MMG*

In Figure 4.6, six MAC-nodes, labeled $M1, \ldots, M6$, were formed after the projection of a 1×1 prism along the Y axis. $M1$ contains computational nodes located at $\langle 1,1,1\rangle$, $\langle 2,1,1\rangle$ and $\langle 3,1,1\rangle$; $M2$ contains nodes located at $\langle 1,2,1\rangle$, $\langle 2,2,1\rangle$, and $\langle 3,2,1\rangle$; $M3$ contains nodes located at $\langle 1,3,1\rangle$, $\langle 2,3,1\rangle$ and $\langle 3,3,1\rangle$.

The resulting *MAC-graphs* for *Warshall(3)* with base size 1×1 are shown in Figure 4.7, 4.8, and 4.9.

The choice of grouping axis and the dimensions of the rectangular prism affect the performance and cost of the derived mesh array computational graph as discussed in [1]. For example, the MAC-graph generated by grouping along the Z axis of *Warshall(3)'s MMG*, shown in Figure 4.7, has no MAC-node that contains more than three computational nodes, while the MAC-graph generated by grouping along the Y axis of *Warshall(3)'s MMG*, shown in Figure 4.8, has MAC-node that contains four computational nodes. The number of computational nodes

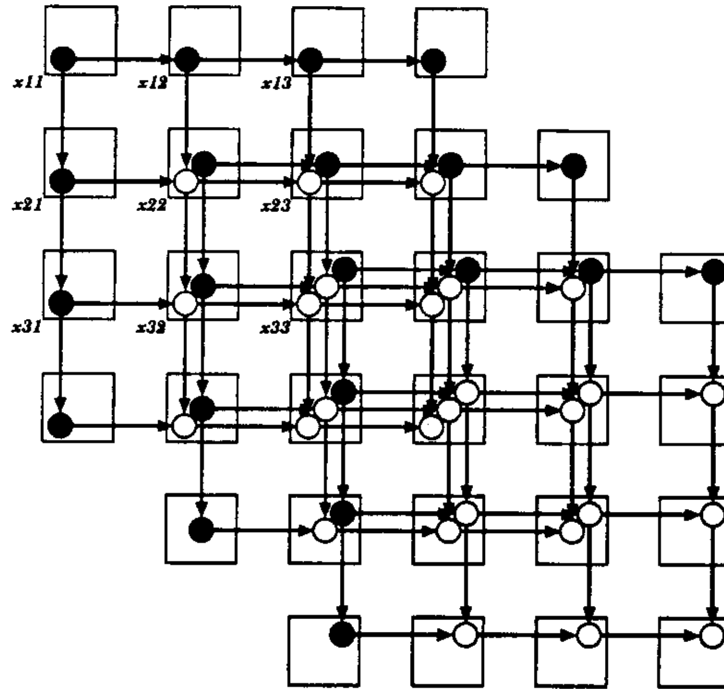


Figure 4.7: *MAC-graph* generated by grouping along the Z-axis of *Warshall(3)*'s *MMG*

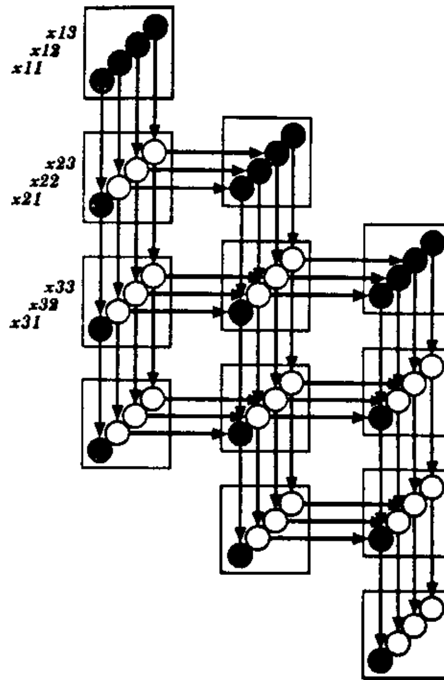


Figure 4.8: *MAC-graph* generated by grouping along the Y-axis of *Warshall(3)'s MMG*

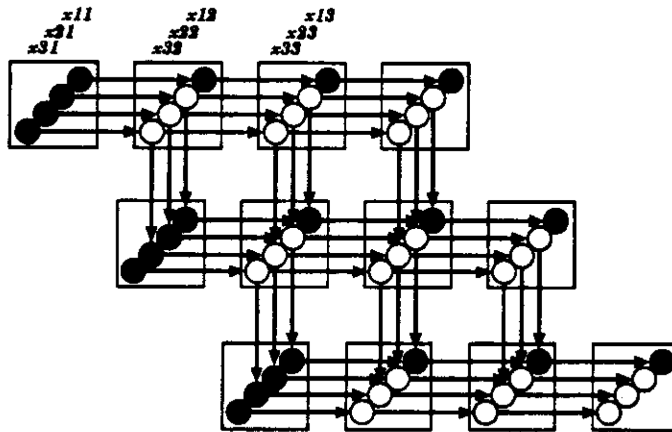


Figure 4.9: *MAC-graph* generated by grouping along the X-axis of *Warshall(3)'s MMG*

within a MAC-node dictates the amount of computation needs to be done by that MAC-node. Thus, the MAC-graph generated by grouping along the Z axis has better throughput than the MAC-graph generated by grouping along the Y axis because the first takes less time to compute (three computational nodes) and pass on the result than the latter (four computational nodes).

The next chapter describes the implementation of MAMACG.

CHAPTER 5

Implementation of MAMACG

Following object-oriented programming style, the current implementation of MAMACG has five objects—ODG, MMG, X-MAC-graph, Y-MAC-graph, and Z-MAC-graph—which are responsible for the formation of the *ODG*, *MMG*, *MAC-graphs* obtained by grouping along the X-axis, Y-axis, and Z-axis, respectively.

When an ODG object is created, it:

1. takes input, output, and local variables;
2. creates a hash table to contain all variables with each variable associated with a computational node;
3. creates a hash function to lookup and update information about a computational node; and
4. creates a mesh-3d-matrix with each entry corresponding to a label of a computational node.

The ODG object accepts the following messages:

assign! Take an assignment statement, process the statement according to Chapter 3 to create a computational node and insert it into the mesh-3d-matrix. During the processing of a statement, the following actions can occur: **insert-input!**, **insert-output!**, **input-cnt**, **x-output-cnt**, *etc.*

display-odg Display the *ODG*.

Upon its creation, the MMG object inherits from the ODG object the mesh-3d-matrix, and all the information about each computational node, including: **input-cnt**, **output-cnt**, **input-queue**, **x-output-queue**, **y-output-queue**, *etc.* The MMG object accepts the following messages:

remove-bi-flow! Apply the technique described in Chapter 4 to remove bidirectional flows in the existing mesh-3d-matrix.

display-mmg Display the *MMG*.

Upon its creation, the X-MAC-graph object inherits from the MMG object the transformed mesh-3d-matrix. Presently, X-MAC-graph object has only one function—to display the *MAC-graph* obtained by grouping the prism of size 1×1 along the x-axis. In the future, the code can be expanded to apply all prisms of size $p \times q$ with p being one of the factors of β' and q being one of the factors of γ', where $\alpha' \times \beta' \times \gamma'$ is the size of the *MMG*. Y-MAC-graph and Z-MAC-graph objects are treated in the same manner as the X-MAC-graph object.

Elk [3] has proven to be a valuable asset in the speedy developement of MA-MACG. To improve its single-view interface session restriction (only one session may exist at a time), we have partially implemented multiple-view interface sessions (one session is interpretive and the rest are graphical) so that all sessions can be operated simultaneously. Since the interpretive interface session operates under Scheme, it provides a complete and powerful procedural language to create, manipulate, and communicate with the various objects—ODG, MMG, X-MAC-graph, Y-MAC-graph, and Z-MAC-graph—in the system. The graphical

interface sessions allow the designer to see the various aspects of the objects in the system from many different angles and to manipulate the objects using the friendlier graphical interface (although much less powerful than the interpretive interface.)

The current Elk implementation, however, lacks the speed to deal with large problems. For example, solving the *MMG* for *Warshall(6)* takes approximately two minutes and the time complexity grows exponentially. An Elk compiler would be highly desirable.

Currently the main code of MAMACG has approximately 5000 lines of Scheme code (about 140 Kbytes).

CHAPTER 6

A Survey of Related Work

MAMACG derives the array computational graphs of the a given matrix algorithm by applying the five steps shown in Figure 1.1. Steps 2 through 4 are based on the MMG method described in [1]. MAMACG formulates the high-level description of the MMG method in terms of an Algol-like language to automate it in Elk.

In [1], the regularization stage transforms MAC-algorithms directly to orthogonal data-dependency graphs. However, the technique was not completely developed and, therefore, could not be implemented. Step 1 shown in Figure 1.1 sidesteps this problem by requiring matrix algorithms to be written in index-dependency formats which can then be directly realized to orthogonal data-dependency graphs. Index-dependency format, as defined in [1], is characterized by the following property:

> **"The computation of each instance of a variable in an algorithm is associated with a point in a multi-dimensional space defined by the indices."**

This chapter begins by describing the two index-dependency techniques—Rao's regular iterative algorithm [6] and MQRS's affine recurrence equations [10]—and pointing out their relations to MAMACG. It then covers the related

synthesis works which include VACS [13], SDEF [11], and ADVIS [12].

6.1 Rao's Regular Iterative Algorithm (RIA)

In [6], Rao's RIA is defined by the triple $\{I, X, F\}$ where:

"I is the index space which is the set of all lattice points enclosed within a specified region in S-dimensional Euclidean space,

X is the set of V variables that are defined at every point in the index space, where the variable x_j defined at the index point k is denoted by $x_j(k)$ and takes on a unique value in any particular instance of the algorithm, and

F is the set of functional relations among the variables restricted to be such that if $x_i(k)$ is computed using $x_j(k - d_{ji})$, then

> d_{ji} is a constant vector independent of k and the extent of the index space, and for every l contained in the index space, $x_i(l)$ is computed using $x_i(l - d_{ji})$ (if $x_j(k - d_{ji})$ falls outside the index space, then this is an external input to the algorithm)."

Rao's Regular Iterative Algorithm requires the algorithm to be written in the following restricted format:

1. statements must be written in single assignment form,
2. index-matching—every subscripted variable must associate its index with a unique three-dimensional coordinate—and
3. localization of dependencies—a variable may receive data only from another variable if their three-dimensional coordinates are adjacent.

In [1], Moreno points out the limitations of using RIA

> "RIAs seem attractive as regular descriptions of algorithms, due to their compactness and suitability for manipulation. However, an analysis of the process of obtaining RIAs indicates the following limitations:
>
> - Transforming an algorithm into a RIA might add computing load, in terms of additional variables and operations. Consequently, implementing the RIA implies performing more operations.
> - Currently, there is no systematic technique to obtain an RIA for a given algorithm."

MAMACG incorporates RIA's single assignment rule. Another index-dependency technique, called affine recurrence equations, is covered next.

6.2 MQRS's Afine Recurrence Equations (ARE)

A matrix algorithm represented in affine recurrence equations (ARE) is realized as a systolic array by mapping ARE to the reduced dependence graph (RDG), and then assigning timing-functions to the vertices in the RDG. ARE, RDG, and the timing-functions are described in [10] as follows:

> "In the following, Z denotes the set of integers. A system of ARE is a finite collection of equations of the form
>
> $$z \in D \rightarrow U(z) = f[V(I(z)), \ldots]$$

where:

- z is a point of Z^n.
- U and V are variables belonging to a finite set of V.
- D is the set of integral points of a convex polyhedron of Z^n, i.e. a set defined by a finite number of linear inequalities on z. D is called the domain of the equation.
- I is an affine mapping from Z^n to Z^l called index mapping. I has the form

$$I(z) = A.z + B$$

  where the constants A and B are integral matrices: $A \in Z^l \times Z^n$, and $B \in Z^l$.
- f is a single-valued function that depends strictly on its arguments; we assume that the function f has complexity $O(1)$.
- the $\ldots$ means that there can be other arguments of the same form as $V(I(z))$.
- the domains of two equations having the same variable in the left-hand side are disjoint. This hypothesis ensures that a variable is not defined twice.

Given an ARE, we say that a variable instance $U(x)$ depends directly on $V(y)$ if there exists an equation

$$z \in D \rightarrow U(z) = f[V(I(z)), \ldots]$$

such that $x \in D$ and $y = I(x)$. We say that $U(x)$ depends on $V(y)$, and we denote $U(x) \succ V(y)$, iff there exists a finite sequence

of directly dependent variable instances $U_1(x_1), \ldots, U_k(k)$ such that $U(x) = U_1(x_1)$ and $V(y) = U_k(x_k)$.

The reduced dependence graph (RDG) of an ARE is the graph whose vertices are the variable $U \in \mathbf{V}$ of the system, and whose edges are tuples (U, V, D, I), with origin vertex U, extremity vertex V, domain D and index I. There is such an edge for all pair U and V of equations $z \in D \rightarrow U(z) = f[V(I(z)), \ldots]$.

The scheduling problem is to find a function that associates each variable instance $U(z)$ with a given non negative instant of time t, in such a way that the arguments needed for the calculation of $U(z)$ are already calculated at time t. If such a mapping exists, the system is said to be explicit or computable.

A particular case of timing function is an *affine timing function*. Define for each variable U, a function t_U from Z^n to Z, where n is the index dimension of U, of the form:

$$t_U(z) = \lambda_U^1 z_1 + \ldots + \lambda_U^n z_n + \alpha_U$$

where $\lambda_U^1, \ldots, \lambda_U^n$ are integers independent of U. In the following, we let $\lambda_U.z = \lambda_U^1 z_1 + \ldots + \lambda_U^n z_n$.

Assume that the evaluation of each function of the system takes at least one unit of time. The following theorem gives a constructive means for obtaining the function t_U:

Theorem The numbers $\lambda_U, \alpha_U, U \in V$ define a timing function iff:

- for all edge (U, V, D, I) of the RDG, we have:

1. for all vertex σ of D, $\lambda_U.\sigma - \lambda_V.(I(\sigma)) + \alpha_U - \alpha_V > 0$,

2. for all ray ρ of D, $\lambda_U.\rho - \lambda_V.A.\rho \geq 0$

- for all variable U,

1. for all vertex σ of the domain D_U of U, $\lambda_U.\sigma + \alpha_U \geq 0$,

2. for all ray ρ of D_U, $\lambda_U.\rho \geq 0$."

Because ARE can only be applied to a limited number of applications and because of the complexities involved in deriving the timing-functions, we did not consider using it to derive the orthogonal data-dependency graph. Instead, MAMACG requires the designer to transform the algorithm into a format in which the order of time a variable is assigned a value is explicitly stated. This format enables MAMACG to directly realize the algorithm as an orthogonal data-dependency graph.

The rest of this chapter describes the related projects that tried to automate the process of realizing matrix algorithms to systolic arrays. They include VACS, SDEF, and ADVIS.

6.3 VLSI Array Compiler System (VACS)

VACS, developed by Kung and Jean [13], is based on the graph-based method developed by Kung [2], and accepts high-level behavior inputs and generates (optimal) array structures. It proceeds in the following three stages:

1. Obtain the Dependency Graph (DG). There is no systematic way of deriving the DG, although some guidelines have proven useful.

2. Map DG to Signal Flow Graphs (SFGs). As described in [2], "a complete SFG description should include both functional and structural description parts. The function description defines the behavior within a node, whereas the structural description specifies the interconnection (edges and delays) between the nodes. Theoretically, the structural part of an SFG can be represented by a finite directed graph, $G = \langle V, E, D(E) \rangle$. The vertices V model the nodes. The directed edges E model the interconnections between the nodes. Each edge e of E connects an output port of a node to an input port of some node and is weighted with a delay count $D(e)$. The delay count is the number of delays along the connection. Often, input and output ports are referred to as sources and sinks, respectively."

3. Realize SFG to processor array. The highly abstracted SFG can easily be transformed to a SIMD, systolic array, or wavefront array.

As stated in [13],

> "VACS accepts high-level behavior inputs in terms of dependence graphs and generates (optimal) array structures. VACS is interactive and graphic-based with several optimality criteria and design tradeoff being evaluated in advance. A complete system should have included original derivation of DG, DG/SFG transformation, mapping onto arrays, and interfaces with lower-level compiler and with host machine.
>
> Ultiminately, different input formats may be accepted by the VACS. It appears very feasible to translate the specific inputs into a DG expression. Some researchers have made very good progress toward this goal [9] [8]. With the awareness of such development, we have so far

focused on using DGs as input in the current VACS. The VACS generates array structures for a variety of algorithms, including those with non-shifting-invariant DGs and processors with time-varying functions. Graphic interface, criteria evaluator, and simulation tools are provided to facilitate the design process. A designer can see the DG graphically displayed, modify the DG, simulate the DG, evaluate different optimality criteria, select an "optimal" design, and even evaluate the numeric performance of the array with finite word precisions."

MAMACG shares a similar goal with VACS in displaying the orthogonal data-dependency graph. MAMACG, however, goes a step further in automating the derivation of the MMG and MAC-graphs (correspond to SFGs).

6.4 SDEF Programming System

SDEF is a systolic array programming system. As stated in [11],

> "It is intended to provide (1) systolic algorithm researchers/developers with an executable notation, and (2) the software systems community with a target notation for the development of higher-level systolic software tools.
>
> The SDEF system constitutes a programming environment for describing systolic algorithms. It includes a notation for expressing systolic algorithms, a translator for the notation, and a systolic array simulator with trace facilities.
>
> The translator generates a C program that performs the computation specified by the SDEF description. After being compiled, this C

> program can be run on the SDEF systolic array simulator.
>
> An SDEF program specifies both the computation and the communication requirements of a systolic algorithm. The SDEF program also specifies how the systolic algorithm is to be 'embedded' in spacetime.
>
> The SDEF notation is not intended to be the ultimate systolic programming language. Higher-level languages are contemplated. Much research into tools for analyzing systolic algorithms, as well as synthesizing and optimizing them, has been conducted. SDEF does not subsume these tools. Where automated, such tools can be connected to SDEF's front-end: SDEF can be used to express the results of the analyses, syntheses, and optimizations performed by other tools."

By using SDEF, a systolic array system can be made reusable, verifiable, and testable. However, SDEF is not a synthesis tool. MAMACG, in contrast, is a synthesis tool that automates the derivation of the mesh array computational graph, a basic representation of a systolic array system. However, the results automatically generated by MAMACG possess SDEF's positive characteristics—reusability, verifiability, and testability.

6.5 ADVIS: Automatic Design of VLSI Systems

ADVIS is a tool for partitioning and mapping matrix algorithms into systolic arrays. It does so in three stages:

1. Derive the mathematical models for VLSI arrays and algorithms. This formalism is necessary for mapping algorithms into architectures and for partitioning algorithms.

2. Map matrix algorithms into VLSI arrays using the formalism derived in stage 1.

3. Partition Algorithm.

One of the main problems of ADVIS is the manual derivation of the data dependence matrix in its formalism. This step could be automated, but more research would be needed. The results produced by ADVIS are algebraic expressions whose correctness can be hard to verify. MAMACG and ADVIS differ in approach.

CHAPTER 7

Summary and Further Research

We have described a tool and its implementation for automating the transformation of a class of matrix algorithms to multi-mesh graphs and projecting them onto mesh array computational graphs. The steps taken to derive the mesh array computational graphs include:

1. Transform a matrix algorithm into a special format called *array computational format.* Because the index of all subscripted variables appearing inside this format have 3-tuple indices, each subscripted variable can be associated with a three-dimensional coordinate based on its index.

2. Symbolically execute the algorithm for a fixed-size problem to generate symbolic statements.

3. Map the symbolic statements to a orthogonal data-dependency graph (*ODG*). Each symbolic statement generate a computational node that takes up a position in the three-dimensional coordinate based on its 3-tuple index.

4. Remove bidirectional flows within the *ODG* to obtain a multimesh graph (*MMG*) that has only unidirectional flows. This topology allows the input to enter on the left and top of the graph and the output to exit on the right and bottom of the graph which is a desirable feature for chip layout. Fur-

thermore, the flow of data from a node to one of its neighboring nodes does not collide with any other flow, a desirable feature for fast communication.

5. Map *MMG* onto mesh array computational graph *MAC-graphs* by projecting along the X, Y, and Z axes.

The results obtained by MAMACG are concrete and can be used to develop additional tools to ultimately derive the special-purpose and general-purpose mesh processor arrays.

The technique, however, can only be applied to a limited class of matrix algorithms with small fixed-size data. Futhermore, the first step in transforming matrix algorithms to array computational formats still requires user intervention.

To achieve the goal of full automation, a technique for mapping matrix algorithms to their array computational formats is being developed.

It is not feasible to derive *MAC-graphs* for large-size problems due to memory and time requirements. To solve this problem, a technique that recognizes the geometry of the three-dimensional *MMG* is being investigated. The recognition of the geometry of a small-size three-dimensional *MMG* enables MAMACG to predict the topology and content of the large size *MMG*, thus enabling MAMACG to solve a large-size problem using minimal amounts of memory and processing time.

The different *MAC-graphs* obtained by projecting along one of the three axes possess different characteristics. A technique for deriving the three important characteristics—cell communication bandwidth, local storage per cell, and storage organization—of the special-purpose mesh array computational system described in [1] can be automated so that the most appropriate (in terms of cost

measures and technology constraints) system for a given technology can be determined quickly and accurately.

An architecture that could take advantage of MAMACG's multiprocessing capability would provide the basic structure of a general-purpose mesh processor array. Based on such an architecture and on MAMACG's multiprocessing capability, a compiler could be developed to generate code that optimally schedules and executes matrix algorithms on mesh array architectures.

APPENDIX A

Running MAMACG

MAMACG runs on Elk, a dialect of Scheme with builtin X graphics capability. This appendix assumes that Elk exists on the system. MAMACG is divided into portions—main codes and algorithms

Main codes

G-gui.scm	MAC-graph user interface
adt.scm	basic abstract data type objects
math.scm	local math library
matrix.scm	matrix library
mesh-util.scm	MAC-graph utilities
mesh.scm	MMG object routines
object.scm	object programming routines
systolic.scm	ODG object routines
transform.scm	graphics transformation routines
util.scm	basic Scheme utilities
vector.scm	vector library

Algorithms

BAinv.scm	BA-inverse
given.scm	given rotation
mat-mul.scm	matrix multiplication
warshall.scm	Warshall transitive closure

To run an algorithm, says warshall.scm, do

```
% elk
> (load 'warshall.scm)
```

At this point, elk will generate the following messages before returning the prompt

```
[Autoloading systolic.scm]
[Autoloading mesh-gui.scm]
[Autoloading xlib]
[Autoloading xlib.o]
[Autoloading util.scm]
[Autoloading object.scm]
[Autoloading adt.scm]
[Autoloading vector.scm]
[Autoloading matrix.scm]
[Autoloading transform.scm]
[Autoloading math.scm]
```

```
[Autoloading pattern.scm]
[Autoloading mesh-util.scm]
[Autoloading G-gui.scm]
[Autoloading mesh.scm]
```

To obtain the *ODG* for the transitive closure problem of size 3, run

```
> (warshall 3)
```

An X window, called ODG-window, should pop up. Type L and the ODG of (*warshall(3)*) will appear on the ODG-window. The MMG of (*warshall(3)*) can be obtained by typing R on the ODG-window. A second X window, called MMG-window, will appear along with the MMG. The MAC-graphs can be obtained by typing G on the MMG-window. Three X windows—X-MAC-window, Y-MAC-window, and Z-MAC-window—will appear along with the X-MAC-graph, Y-MAC-graph, and Z-MAC-graph. The windows can be destroyed by clicking on the MMG-window and the ODG-window.

REFERENCES

[1] Jaime H. Moreno and Tomás Lang. Matrix Computations on Systolic-Type Meshes, *IEEE Computer*, **23**(4):32–51 (April 1990).

[2] S. Y. Kung. VLSI Array Processors, Prentice-Hall, 1988.

[3] Oliver Laumann. Elk Extension Language Interpreter.

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. pp 195–223.

[5] Veljko M. Milutinovic. Computer Architecture. pp 454–494.

[6] Shailesh K. Rao and Thomas Kailath, Fellow, IEEE. Regular Iterative Algorithm and their Implementation on Processor Arrays. *Proceedings of the IEEE*, Vol. **76**, No. 3, March 1988, pp 259–269.

[7] José A. B. Fortes, King-Sun Fu, and Benjamin W. Wah. Systematic Design Approaches for Algorithmically Specified Systolic Arrays. *Computer Architecture Concepts and Systems*, edited by Veljko M. Milutinović. North Holland, 1988.

[8] Marina C. Chen. A parallel language and its compilation to multiprocessor machines, *J. Parllel and Distributed Computing*, 1986, pp. 461–91.

[9] J. Bu and Ed F. Deprettere. Converting Sequential Iterative Algorithms to Recurrent Equations for Automatic Design of Systolic Arrays, in *Proc. IEEE ICASSP*, pp. 2025–28, 1988.

[10] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannich Saouter. Scheduling Affine Parametrized Recurrences by means of Variable Dependent Timing Functions. *Proceedings Application Specific Array Processors*. Edited by Sun-Yuan Kung, Earl E. Swartzlander, Jr., Jose A. B. Fortes, and K. Wojtek Przytula. IEEE Computer Society Press, September 1990.

[11] Bradley R. Engstrom and Peter R. Cappello. The SDEF Programming System. *Journal of Parallel and Distributed Computing* **7**, 201–231 (1989).

[12] D. I Moldovan. ADVIS: A software package for the design of systolic arrays. *IEEE Transactions on Computer-Aided Design*, January 1987, pp. 33–40.

[13] S.Y. Kung and Jack S. N. Jean. Array Compiler Design for VLSI/WSI Systems. *International Conference on Systolic Arrays held at Killarney, Co. Kerry, Ireland, 1989.* Edited by John McCanny, John McWhirter, and Earl Swartzlander Jr. Prentice Hall.