

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

MATRIX COMPUTATIONS ON MESH ARRAYS

Jaime H. Moreno

**August 1989
CSD-890046**

UNIVERSITY OF CALIFORNIA

Los Angeles

Matrix computations on mesh arrays

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

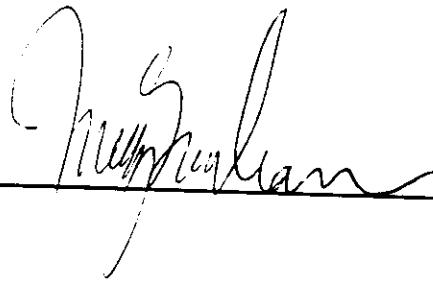
by

Jaime H. Moreno

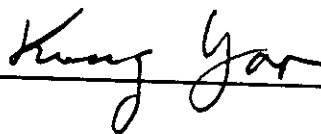
1989

© Copyright by
Jaime H. Moreno
1989

The dissertation of Jaime H. Moreno is approved.



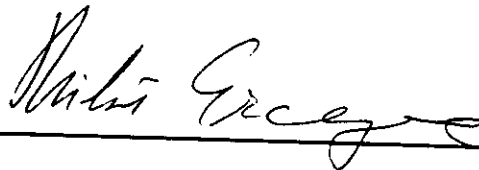
Tony Chan



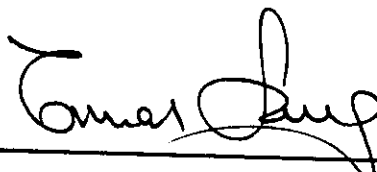
Kung Yao



Dave Rennels



Miloš Ercegovac



Tomas Lang, Committee Chair

University of California, Los Angeles

1989

To Marisa, who shared this experience
with love, understanding and support.

To my parents, who instilled in me the de-
sire to reach further.

TABLE OF CONTENTS

1	Introduction	1
1.1	Matrix computations and arrays	1
1.2	Description of research and summary of accomplishments	5
1.3	Organization of the dissertation	11
2	The design of arrays for matrix algorithms	13
2.1	Matrix algorithms	13
2.2	The architectural model of mesh arrays	14
2.3	Tradeoffs in throughput, cell storage and cell bandwidth	17
2.4	Realizing algorithms and mapping algorithms onto arrays	20
2.5	The range of applicability of arrays	22
2.6	Approaches to partitioning large problems	23
2.7	Issues in the design of arrays	29
3	Methods for the design of arrays	33
3.1	The classification of design methods by Fortes (et al.)	33
3.2	A framework to compare design methods	36
3.3	A review of other design methods	38
3.3.1	Algebraic descriptions	39
3.3.2	Descriptions using high-level languages	42
3.3.3	Graph-based descriptions	43
3.3.4	Discussion of other methods	45

3.3.5	Methods for partitioning	46
3.3.6	Mapping methods for class-specific arrays	47
3.4	Computer-aided design tools	48
3.5	Algorithm dependencies in methods for the design of arrays	51
3.6	Conclusions regarding methods for the design of arrays	54
4	Description of the graph-based method	57
4.1	Assumptions regarding matrix algorithms and arrays	57
4.1.1	Matrix algorithms	57
4.1.2	Cells and arrays	59
4.2	Summary of the data-dependency graph-based design approach	61
4.2.1	The regularization stage	63
4.2.2	The derivation of arrays	64
4.2.3	The performance and cost measures	65
4.3	Obtaining the fully-parallel data-dependency graph	66
4.4	Obtaining the multi-mesh data-dependency graph	69
4.5	Deriving G-graphs from a complete multi-mesh graph	75
4.5.1	The collapsing of a CMMG	76
4.5.2	The execution of a G-node	78
4.5.3	Cell properties that depend on the G-graph	78
4.6	Deriving arrays for fixed-size data from a CMMG	84
4.6.1	Two-dimensional arrays	84
4.6.2	Linear arrays	85
4.7	Deriving arrays for partitioned problems from a CMMG	85
4.7.1	The selection of G-sets	85
4.7.2	The scheduling of G-sets	86

4.7.3	Array properties	88
4.8	Using IMMGS	90
4.9	Deriving arrays for fixed-size data from an IMMGS	93
4.10	Deriving partitioned implementations from an IMMGS	95
4.10.1	The selection of G-sets	95
4.10.2	Array properties	96
4.11	Using MMGS with two flows of input data	97
4.11.1	Two-dimensional arrays for fixed-size data	98
4.11.2	Two-dimensional arrays for partitioned problems	100
4.12	Summary of performance measures of arrays	101
4.12.1	Fixed-size data	101
4.12.2	Partitioned problems	102
4.13	Arrays for the triangularization algorithm	102
4.13.1	Computational load	103
4.13.2	Problems with fixed-size data	103
4.13.3	Partitioned implementations	106
4.14	Type of array as a function of the design parameters	108
4.15	Tradeoffs between linear and two-dimensional arrays	109
4.16	A canonical linear array for partitioned problems	110
5	The formalization of the design method	113
5.1	Definitions	113
5.1.1	The canonical representation of matrix algorithms	114
5.1.2	The arrays	117
5.1.3	The model of execution in an array	118
5.1.4	The performance measures	118

5.1.5	The cost measures	119
5.1.6	The graphs	120
5.1.7	The equivalence of graphs	124
5.1.8	The realization of a graph	125
5.2	Step 4: Realizing a mesh data-dependency graph as an array	126
5.3	Step 3: Transforming a multi-mesh data-dependency graph into a mesh graph	128
5.4	Properties of cells that depend on the grouping process	135
5.4.1	Utilization of cells	135
5.4.2	Types of operations per cell	137
5.4.3	Storage per cell	137
5.4.4	Cell bandwidth	141
5.4.5	Cell pipelining	142
5.5	Step 2.2: Transforming a three-dimensional graph into an MMG	143
5.5.1	Eliminating data broadcasting	144
5.5.2	Eliminating bidirectional dependencies	144
5.5.3	Removing non-nearest neighbor dependencies	146
5.5.4	Synchronizing arrival of data to nodes	147
5.6	Step 2.1: Transforming the FPG into a three-dimensional graph	148
5.7	Step 1: Deriving the FPG of a matrix algorithm	162
5.8	Deriving arrays for matrix computations	165
5.9	The procedure to derive arrays for matrix algorithms	165
5.10	Partitioning	166
5.11	The application of the method to the LU-decomposition	167
5.11.1	Deriving the FPG	167
5.11.2	Deriving the MMG	167

5.11.3	Deriving the MGs	168
5.11.4	Realizing the G-graphs as two-dimensional arrays	170
5.11.5	Evaluation of the arrays	172
6	Mapping algorithms onto class-specific arrays	179
6.1	The regularization stage	179
6.2	The mapping stage and the specific target architecture	180
6.3	Mapping onto local-access arrays	183
6.3.1	The impact of large storage per cell	183
6.3.2	Coalescing the MMG	184
6.3.3	A heuristic approach to non-uniform coalescing	186
6.3.4	The allocation of data and the schedule of primitive operations	187
6.4	Example: mapping onto a memory-linked array	192
6.4.1	The target architecture	192
6.4.2	The performance evaluation measures	193
6.4.3	The mapping process	195
6.4.4	Uniform coalescing	195
6.4.5	Interleaved uniform coalescing	203
6.4.6	Non-uniform coalescing	204
6.5	Conclusions	206
7	A comparison with other methods based on dependencies	209
7.1	The regularization of algorithms	212
7.1.1	Index-dependencies	212
7.1.2	S.Y. Kung's method	218
7.1.3	MMG method	222

7.2	The derivation of arrays	229
7.2.1	Rao's method	229
7.2.2	S.Y. Kung's method	232
7.2.3	MMG method	234
7.2.4	Comparison of derivation of arrays	238
7.3	Conclusions regarding the comparison	238
8	Summary and further research	243
	References	247
A	Arrays for the Faddeev algorithm	261
A.1	Introduction	261
A.2	The modified Faddeev algorithm	262
A.3	Arrays for problems with fixed-size data	265
A.4	Evaluation of the arrays for problems with fixed-size data	270
A.5	The partitioning problem in the Faddeev algorithm	271
A.5.1	Partitioned structures previously proposed	272
A.5.2	Partitioning for linear arrays	273
A.5.3	Partitioning for two-dimensional arrays	274
A.5.4	Scheduling and I/O in partitioned Faddeev algorithm	275
A.5.5	Comparison of arrays for partitioned execution	277
A.6	Conclusions	284
B	Arrays to compute BA^{-1}	285
B.1	Introduction	285
B.2	Systolic arrays for computing BA^{-1}	285

B.3	Evaluation of the arrays	290
B.4	Conclusions	292
C	Arrays for LU-decomposition with neighbor pivoting	293
C.1	Introduction	293
C.2	The fully-parallel graph	293
C.3	Deriving the MMG	293
C.4	Deriving arrays for problems with fixed-size data	297
D	Algorithms with affine dependencies	299
D.1	The convolution algorithm	299
D.2	Deriving the MMG	300
D.3	Deriving arrays for problems with fixed-size data	302

LIST OF FIGURES

1.1	Parallel architectures for matrix computations	2
1.2	Matching fine-grain parallelism and architecture	3
1.3	Classes of application-specific arrays for matrix algorithms	4
1.4	Data-dependency graph-based design method	7
2.1	Examples of array structures	14
2.2	Cells for the different types of arrays	16
2.3	Computational model for multiple instances	17
2.4	Tradeoffs between local storage and cell bandwidth	19
2.5	The direct realization of a dependency graph	20
2.6	Partitioning an algorithm	24
2.7	Partitioning an algorithm through coalescing	24
2.8	Partitioning an algorithm through cut-and-pile	25
2.9	Partitioning an algorithm through coalescing and cut-and-pile	26
2.10	Partitioning an algorithm through decomposition into subalgorithms	27
2.11	Indirect and direct partitioning	28
2.12	Using the internal regular part of an algorithm in partitioning	29
3.1	Y-charts to describe transformational systems [Fort88]	34
3.2	Y-charts for two dependency-based methods [Fort88]	36
3.3	The stages in a design method	36
3.4	Index-dependencies	53
3.5	Data-dependencies	54

4.1	The canonical form of a matrix algorithm	58
4.2	Dependency graphs of vector and matrix operators	58
4.3	Mesh-connected array	60
4.4	Data-dependency graph-based design method	62
4.5	The triangularization algorithm by Givens' rotations	66
4.6	Symbolic evaluation of the triangularization algorithm	67
4.7	The FPG of the triangularization algorithm by Givens' rotations	68
4.8	Examples of multi-mesh data-dependency graphs	70
4.9	Removing properties not allowed in an MMG	72
4.10	Graph with no broadcasting for the triangularization algorithm	73
4.11	Multi-mesh dependency-graph for the triangularization algorithm	74
4.12	Examples of mesh data-dependency graphs (G-graphs)	75
4.13	The CMMG used to discuss the derivation of arrays	76
4.14	Deriving G-graphs from a CMMG	77
4.15	The schedule of primitive nodes within a prism	79
4.16	Local storage organization in a cell	82
4.17	Independent nodes in the flow of transmittent data	83
4.18	Realizing a G-graph as an array for fixed-size data	84
4.19	Dividing a G-graph into G-sets	86
4.20	Scheduling a G-graph in linear and two-dimensional arrays	87
4.21	The schedule of G-sets	88
4.22	I/O bandwidth in partitioned implementations	90
4.23	Drawing prisms in complete and incomplete MMGs	92
4.24	Schedule parallel to flow of transmittent data	93
4.25	Deriving G-graphs from an IMMIG	94

4.26	G-graphs derived from an IMMIG	96
4.27	A CMMG and G-graphs with two flows of input data	99
4.28	Array for G-graphs with two input flows	100
4.29	Scheduling G-sets with two input flows	101
4.30	G-graphs for the triangularization algorithm	103
4.31	Prisms for partitioned problems in the triangularization algorithm .	107
4.32	G-sets in the triangularization algorithm	107
4.33	The canonical linear array for partitioned problems	110
5.1	The canonical form of a matrix algorithm	115
5.2	Complete and incomplete mesh arrays	117
5.3	Example of a data-dependency graph	120
5.4	Examples of mesh dependency-graphs	121
5.5	Examples of multi-mesh dependency-graphs	123
5.6	Examples of equivalent graphs	124
5.7	Summary of design method's formalization	125
5.8	Realizing a mesh graph as an array	127
5.9	Collapsing a prism of primitive nodes onto a single node	129
5.10	Collapsing neighbor nodes from a mesh onto a single node	131
5.11	Grouping-by-prisms of size 1 by 1 by n in a CMMG	133
5.12	Grouping along directions other than axes	134
5.13	Cut-set of primitive nodes executed up to time $t = 12$ in a prism . .	138
5.14	Cardinality of a cut-set in complete and incomplete prisms	139
5.15	Determining local storage in a cell from a prism	140
5.16	Cell bandwidth and pipelining	142
5.17	Example of broadcasting and transmittent data	144

5.18	Bidirectional dependencies in a graph	145
5.19	Transforming bidirectional transmittent data	146
5.20	Connecting nodes' inputs and outputs that have the same name	149
5.21	Rearranging the flow of data in a plane	150
5.22	Fully-parallel graph of an algorithm with scalar operations	152
5.23	Flows of data and graph levels in a three-dimensional space	153
5.24	Routing data in one plane of the three-dimensional graph	154
5.25	Movements of flows of data in the three-dimensional space	155
5.26	Three-dimensional graph for matrix algorithm as scalar operations	156
5.27	Vector and matrix operators	157
5.28	Vector and matrix operators in neighbor planes	159
5.29	Directly dependent matrix operators in neighbor planes	160
5.30	Allocating directly dependent operators to the same plane	161
5.31	Fully-parallel data-dependency graph for the LU-decomposition	164
5.32	The LU-decomposition algorithm	167
5.33	Three-dimensional graph for the LU-decomposition algorithm	168
5.34	G-graphs for the LU-decomposition algorithm	169
5.35	Arrays for computing the LU-decomposition with fixed-size data	170
5.36	Decoupling I/O from computation in the LU-decomposition	171
5.37	Prism of primitive nodes along the <i>Z</i> -axis	173
6.1	The LU-decomposition algorithm and its MMG	181
6.2	A heuristic approach to perform non-uniform coalescing	186
6.3	The allocation of data to memory modules	187
6.4	Rotation in the square-root free algorithm	188
6.5	Schedule of primitive operations in the LU-decomposition	189

6.6	Scheduling primitive operations in an inner prism of the MMG	191
6.7	An hypothetic memory-linked array	192
6.8	A cell in the target class-specific array	193
6.9	Coalescing the LU-decomposition along the Z -axis	196
6.10	Coalescing the LU-decomposition along the X -axis	199
6.11	Coalescing the LU-decomposition along the Y -axis	202
6.12	Scheduling primitive nodes in interleaved uniform coalescing	205
7.1	A framework to compare design methods	210
7.2	Regular iterative algorithm for matrix multiplication	213
7.3	Regular iterative algorithm for a two-dimensional filtering problem	214
7.4	The regularization stage in Rao's method	214
7.5	Regular iterative algorithm for transitive closure	215
7.6	Reduced dependency graph for transitive closure	216
7.7	Algorithms for Gaussian elimination with partial pivoting	217
7.8	The regularization stage in the Signal Flow Graph method	218
7.9	Single assignment algorithm for transitive closure	219
7.10	The dependency graph for transitive closure in the SFG method	220
7.11	Reindexed dependency graph	221
7.12	Modified DG for the transitive closure algorithm	222
7.13	The regularization stage in the MMG method	223
7.14	The FPG for the transitive closure algorithm	224
7.15	Replacing broadcasting by transmittent data	225
7.16	Removing bidirectional transmittent data	226
7.17	Removing bidirectional transmittent data along X -axis	227
7.18	Unidirectional dependency graph	227

7.19	Multi-mesh dependency graph	228
7.20	The derivation of arrays in Rao's method	231
7.21	The array for transitive closure in Rao's method [Rao85]	232
7.22	The derivation of arrays in the SFG method	233
7.23	Array for transitive closure from S.Y. Kung's method [Kung87c]	234
7.24	The derivation of arrays in the MMG method	235
7.25	Projecting the MMG onto G-graphs	236
7.26	Systolic arrays for transitive closure	237
A.1	Matrix operations with the Faddeev algorithm	263
A.2	The fully-parallel graph without broadcasting	264
A.3	The multi-mesh dependency graph	266
A.4	The G-graph from grouping along the Z -axis	267
A.5	The systolic array from grouping along the Z -axis	268
A.6	The systolic array from grouping along the X -axis	268
A.7	The systolic array from grouping along the Y -axis	269
A.8	A bi-trapezoidal array	270
A.9	The G-graph from grouping along the Y -axis	274
A.10	Partitioned linear array for the Faddeev algorithm	274
A.11	Two-dimensional partitioning of the Faddeev algorithm	275
A.12	Scheduling G-graph into linear and two-dimensional arrays	276
A.13	I/O bandwidth in partitioning the Faddeev algorithm	278
B.1	The algorithm to compute BA^{-1}	286
B.2	The fully-parallel dependency graph	287
B.3	The multi-mesh dependency graph	288

B.4	Grouping nodes along the Y -axis	289
B.5	Grouping nodes along the X -axis	291
C.1	LU-decomposition with neighbor pivoting	294
C.2	Removing broadcasting	295
C.3	Removing bidirectional dependencies	296
C.4	The multi-mesh graph	297
C.5	A G-graph for LU-decomposition with neighbor pivoting	298
D.1	Convolution algorithm with affine dependency	299
D.2	Typical linear array for convolution algorithm	300
D.3	The fully-parallel graph of convolution	301
D.4	The multi-mesh dependency graph	302
D.5	Collapsing MMG along the Z -axis	303

ACKNOWLEDGMENTS

Many people have been an asset to this dissertation, and I wish to express my gratitude to them all.

Special thanks to my advisor, Prof. Tomás Lang. His dedication to this research, his constant encouragement and support, and his constructive suggestions and criticism have shown him to be an outstanding example of an ideal mentor in the true spirit of the ancient traditions. He will be a role model to guide my own research attitude. His advice has brought this dissertation to a level which I would have never reached on my own. Thanks, Tomás.

My appreciation also to the members of my committee, in particular to Prof. Miloš Ercegovac and Prof. David Rennels, for their interest in the topic, their ideas, and especially their support beyond the scope of this research.

I want to express my gratitude to IBM and its Graduate Fellowship Program, who sponsored me during the last two years. Its financial support has been essential for this research. Special thanks to Dr. Vojin Oklobdzija at IBM T.J. Watson Research Center, for his constant interest in my progress and well-being.

I also want to thank everyone in the Computer Science Department at UCLA with whom I have become acquainted and who helped me in different ways. In particular, I thank Dorab Patel, Verra Morgan and Doris Sublette. They were a great help, and I know they will continue helping others in the future; that's their nature. I'll treasure their kindness and friendship forever.

Last, my sincere thanks to the group of graduate students that made my daily life at UCLA a rewarding experience, in particular Leon Alkalaj, Miquel Huguet, T.M. Ravi, Frank Schaffa, Marc Tremblay, Jeong-A Lee and Paul Tu. Our enlightening discussions, our CIGAR meetings, and our good and hard times together will be a permanent memory of these years; our friendship will be part of my life. Thanks to you all.

VITA

- October 14, 1954 Born in Concepción, Chile
- 1978 Electrical Engineer Degree
Universidad de Concepción
Concepción, Chile
- 1978 – 1983 Lecturer, Assistant Professor
Department of Electrical Engineering
Universidad de Concepción
Concepción, Chile
- 1983 – 1985 UNDP Fellowship
- 1985 M.S. in Computer Science
University of California Los Angeles
Los Angeles, California
- 1985 – 1987 Teaching Assistant, Teaching Associate
University of California Los Angeles
Los Angeles, California
- 1987 – 1989 IBM Graduate Fellowship

PUBLICATIONS AND PRESENTATIONS

1. J. Moreno and T. Lang, "Comments on 'A systolic array for computing BA^{-1} ,'" *IEEE Transactions on Acoustics, Speech and Signal Processing*, November 1989.
2. J. Moreno and T. Lang, "A linear array for partitioned execution of matrix algorithms with high utilization," in *SPIE Real-Time Signal Processing XII*, August 1989.
3. J. Moreno and T. Lang, "Comparing design methods based on index dependencies and data-dependencies," in *International Conference on Systolic Arrays*, pp. 599–608, May 1989.
4. J. Moreno and T. Lang, "Arrays for partitioned matrix algorithms: trade-offs between cell storage and cell bandwidth," in *SPIE Real-Time Signal*

Processing XI, pp. 156–169, August 1988.

5. J. Moreno and T. Lang, "Partitioning algorithms for systolic arrays: application to transitive closure," in *International Conference on Parallel Processing*, pp. 28–31, August 1988.
6. J. Moreno and T. Lang, "On partitioning the Faddeev algorithm," in *International Conference on Systolic Arrays*, pp. 125–134, May 1988.
7. J. Moreno and T. Lang, "Design of special-purpose arrays for matrix computations: preliminary results," in *SPIE Real-Time Signal Processing X*, pp. 53–65, August 1987.
8. J. Moreno and T. Lang, "A multilevel pipelined processor for the Singular Value Decomposition," in *SPIE Real-Time Signal Processing IX*, pp. 100–112, August 1986.
9. J. Moreno and T. Lang, "Replication and pipelining in multiple instance algorithms," in *International Conference on Parallel Processing*, pp. 285–292, August 1986.
10. J. Moreno, "Analysis of alternatives for a Singular Value Decomposition processor," Master Thesis, Technical Report CSD-850035, Computer Science Department, University of California Los Angeles, October 1985.
11. J. Moreno and T. Lang, "Graph-based partitioning of matrix algorithms for systolic arrays," Technical Report CSD-880015, Computer Science Department, University of California Los Angeles, March 1988.
12. J. Moreno and T. Lang, "Designing arrays for the Faddeev algorithm," Technical Report CSD-880013, Computer Science Department, University of California Los Angeles, March 1988.
13. J. Moreno and T. Lang, "Reducing the number of cells in arrays for matrix computations," Technical Report CSD-880014, Computer Science Department, University of California Los Angeles, March 1988.
14. J. Moreno and T. Lang, "Removing algorithm irregularities in the design of arrays for matrix computations," Technical Report CSD-870040, Computer Science Department, University of California Los Angeles, August 1987.
15. J. Moreno, "A proposal for the systematic design of arrays for matrix computations," Technical Report CSD-870019, Computer Science Department, University of California Los Angeles, May 1987.

ABSTRACT OF THE DISSERTATION

Matrix computations on mesh arrays

by

Jaime H. Moreno

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor Tomás Lang, Chair

This dissertation addresses the systematic derivation of mesh arrays for matrix computations, in particular realizing *algorithm-specific* arrays and mapping algorithms onto *class-specific* arrays. A data-dependency graph-based transformational method is proposed in a design framework consisting of two stages, namely *algorithm regularization* and *derivation of arrays*. The first stage derives the *fully-parallel data-dependency graph* (FPG) of an algorithm and transforms this graph into a three-dimensional one with unidirectional nearest-neighbor dependencies (a *multi-mesh graph* MMG). The second stage transforms the MMG into a two-dimensional *G-graph*, which is realized as an algorithm-specific array or mapped onto a class-specific array. This stage allows the incorporation of implementation restrictions and the evaluation of tradeoffs in properties of cells, as well as the derivation of arrays for fixed-size data and partitioned problems, while performing optimization of specific performance/cost measures.

The proposed method is formalized by presenting a sufficient set of transformations and demonstrating the equivalence of graphs obtained from those transformations. Moreover, it is demonstrated that the MMG representation is always possible, due to the characteristics of the operators.

The method uses systolic, pseudo-systolic (an extension to systolic) and local-

access cells. *Pseudo-systolic* cells include two small FIFO buffers, require bandwidth that is a fraction of the computation rate, allow performing tradeoffs between memory size and cell bandwidth, and use pipelined functional units efficiently. In contrast, systolic cells have no local storage, while local-access cells have large local memory and low bandwidth.

The method has been applied to a collection of matrix algorithms, including matrix multiplication, convolution, matrix decompositions (LU, QR, Cholesky), transitive closure, the Faddeev algorithm, and BA^{-1} . The examples show that, in addition to the features listed earlier, this method is easy to apply. Moreover, the method is compared with other techniques, concluding that it is advantageous because it meets evaluation criteria and produces more efficient arrays.

A linear class-specific array for partitioned problems is proposed for which the method produces high cell utilization, low I/O bandwidth and low cell bandwidth.

The method has also proved useful for mapping algorithms onto local-access arrays, using coalescing combined with a heuristic approach to achieve load balancing.

CHAPTER 1

Introduction

1.1 Matrix computations and arrays

Matrix computations are characterized by having matrix operands and/or results. These computations are a frequently used mathematical tool in modern scientific and engineering applications, such as image and signal processing, systems theory, communications, pattern recognition, and graph theory [Spei88, Klem80, Andr76, Golu85, Brom81]. For example, in a review of parallel processing algorithms and architectures for real-time signal processing, Speiser and Whitehouse [Spei81, Spei83] have shown that the major computational requirements for many important real-time signal processing tasks (such as adaptive filtering, data compression, beamforming, and cross-ambiguity calculation) may be reduced to a common set of basic matrix computations. For these applications, they showed that the basic set of required matrix computations includes matrix-vector multiplication, matrix-matrix multiplication and addition, matrix inversion, solution of linear systems, eigensystems solution, matrix decompositions (LU-decomposition, QR-decomposition, singular value decomposition), and the Generalized SVD algorithm.

Matrix operations such as those mentioned above are compute-intensive. Consequently, they require high computation rates to achieve acceptable execution times and to meet real-time constraints of many applications. Such computational requirements have limited the adoption or even the comprehensive evaluation of new signal-processing algorithms, permitting them to be applied only to small problems in off-line computations, or to limited data sets [Spei81].

Achieving desired execution rates in matrix computations requires using parallel architectures. Several classes of such architectures have been used for these purposes, as depicted in Figure 1.1, exploiting different types of parallelism [Hayn82]. For example, vector computers [Kogg81] such as CRAY-1, CRAY-2 or NEC SX-1 use parallelism in matrix algorithms through vector instructions which are extracted from sequential programs using vectorizing compilers. These machines

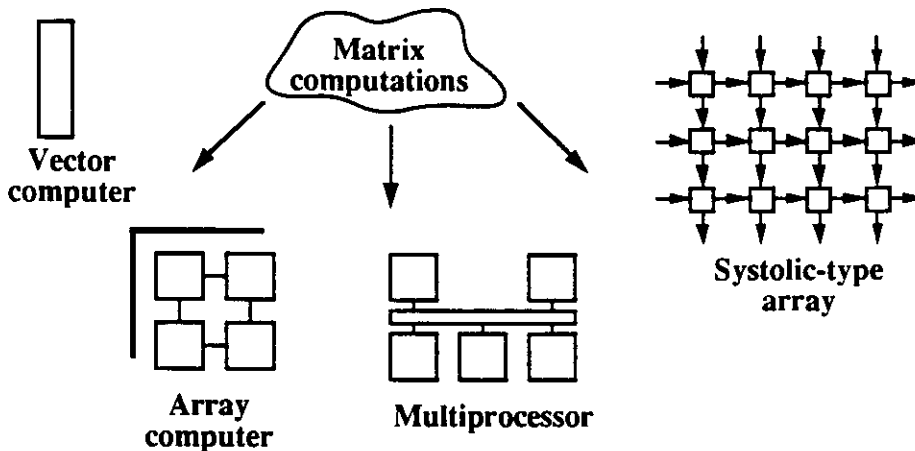


Figure 1.1: Parallel architectures for matrix computations

achieve high performance due to highly pipelined vector units. A similar type of parallelism is used in array computers, such as the historically important ILLIAC IV [Bouk72] and the recent Connection Machine [Hill85]. On the other hand, multiprocessor systems exploit parallelism at several levels: vector operations if they include vector processors (such as Alliant FX/Series [FXS87]), concurrent execution of several loop iterations (also in Alliant FX/Series), block methods that divide an algorithm into parallel tasks (as in Cedar, RP3, Hypercube, Butterfly) [Meie87, Fox88], and linear system solvers [Same85a, Same85b]. Moreover, matrix computations have become one of the preferred benchmarks for these architectures [Dong87a, Dong87b].

Although the above-mentioned parallel architectures have demonstrated that they are effective targets for matrix computations, they suffer from several degradation factors. These factors arise from the relative general-purpose character of those machines, and the need to adapt the algorithms to the specific hardware available in those implementations. Moreover, their general-purpose nature makes it necessary to include features that increase cost (for example, complex memory addressing schemes) and make the architectures less suited for very-large (VLSI) and wafer-scale (WSI) integration technology (for example, broadcasting or complex interconnection networks). These drawbacks have led to the introduction of *systolic-type arrays* [Kung79], architectures that seem very natural for matrix computations because they match well with the fine granularity of parallelism available in the computations, as depicted in Figure 1.2, and have very low



Figure 1.2: Matching fine-grain parallelism and architecture

overhead in communication and synchronization.¹ In addition, the regular nature and nearest-neighbor connections of systolic-type arrays match very well with the requirements for effective use of VLSI and WSI technology [Kung82].

Several *algorithms* may exist for a given computation. Some algorithms are suited for sequential execution (i.e., in a single processor), because they have a small number of operations, exclude complex operations, have locality characteristics that use the memory hierarchy efficiently, or have dependencies that preclude execution in parallel. Other algorithms are better suited for particular types of parallel architectures. Matrix computations have properties that make them attractive to all the above-mentioned classes and many algorithms have been developed [Golub85, SIAM87]. For execution in systolic-type arrays, the algorithm should exhibit sufficient fine-grain parallelism; in many cases the traditional algorithms used in sequential computers have this characteristic, while for others special algorithms have been developed [Luk87, Luk88]. Specific examples of algorithms devised for systolic-type arrays are reported in [Boja84, Boja86, Bren85b, Bren85a, John84, Kung83b, Hell83, Lewi88, Luk86, Núñez84, Núñez88, Torr88, Torr89] as well as in [Davi88, Ibar87, McWh83, Schr82, Rajo88b]. Collections of implementation of algorithms in arrays are found in [Robe86] as well as in [Kung88c].

Some applications require dedicated systems for specific matrix computations. In such cases, an application-specific array might be the most appropriate solution, as depicted in Figure 1.3, due to the possibilities of matching an array to the particular algorithm(s), and of fulfilling specific implementation requirements (such as speed, size, and power consumption.) If the application consists of matrix operations and other computations, the array should be combined with other modules to perform the complete task, composing a heterogeneous system. In addition, an application-specific array is usually connected to a host that performs input/output and control functions.

¹This is in contrast to dataflow computers [Ager82], which also use fine-grain parallelism but have large overheads.

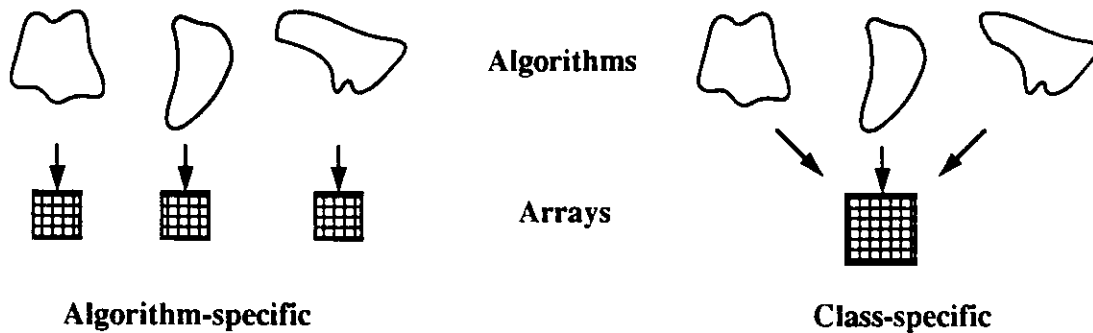


Figure 1.3: Classes of application-specific arrays for matrix algorithms

In contrast to highly-specialized systems for one algorithm, there are situations where the set of matrix computations is varied and not even predefined. For such cases, a general array that can be adapted (i.e., programmed) for a class of matrix algorithms is a suitable solution.

The design of algorithm-specific arrays implies determining the topology of such an array (i.e., triangular, linear, rectangular), the functionality of each processing element, the schedule of operations and data transfers, and the input/output. Such a design should meet the requirements of a particular application and optimize relevant criteria. On the other hand, programming a class-specific array consists of assigning operations to cells, scheduling these operations and data transfers, and specifying the input/output. Consequently, both activities have many aspects in common and similar techniques may be used for both purposes.

Another aspect of importance is related to the relative size of matrices and arrays, and two different cases may be identified. In the first one, the matrix size is fixed and the array is designed to make use of the maximum parallelism achievable with a mesh-type array (i.e., a problem with *fixed-size* data). Such an array is suitable to execute the computation for multiple sets of input data (i.e., a multiple-instance algorithm). In contrast, the second situation corresponds to a matrix that is much larger than the size of a cost-effective array; in this case, the computation has to be partitioned into subproblems and these subproblems executed in sequence (i.e., a *partitioned* problem) [Nava87, Hell84]. Consequently, the array is used many times with the different subproblems while operating on the solution of a single large problem.

The topic of this dissertation is a *method for the realization of matrix computations on mesh arrays*. A detailed description of the characteristics of these arrays and their computational model is given in Chapter 2. This chapter also describes

a generalization of systolic arrays, which consists of adding two small buffers to processing elements. These buffers are used to reduce the communication bandwidth between processing elements. Moreover, Chapter 2 presents a discussion of the requirements that an array has to satisfy and of the optimization criteria that are most significant.

Many methods have been proposed to design arrays [Fort88, Brom88, Fort87a]; some of them are reviewed in Chapter 3. Conclusions obtained from that review indicate that *proposed methods are not general enough* to accommodate a large variety of matrix algorithms, that they are *difficult to use*, and that they are *not able to take into account varying requirements nor incorporate optimization criteria as part of the design*. Moreover, most methods are oriented to the design of arrays for fixed-size matrices and are only indirectly applicable to the case of large matrices. To overcome these limitations, we have developed a data-dependency based method that is the main topic of this dissertation. Such a method is presented in Chapters 4 through 6, and is compared with the most popular previously proposed methods in Chapter 7. In these chapters, the method is illustrated with its application to important matrix algorithms, such as LU-decomposition, QR-decomposition, and transitive closure. More examples are given as appendices. These examples show not only the capabilities of the method but also the derivation of more efficient arrays than those previously proposed. Nevertheless, we concentrate on the capabilities of the method rather than on the arrays obtained.

1.2 Description of research and summary of accomplishments

The research described in this dissertation relates to techniques for the realization of algorithm-specific mesh arrays (i.e., systolic-type) of processing elements (PEs) for matrix algorithms, and for mapping these algorithms onto class-specific arrays. The contributions obtained throughout this research are summarized as follows:

Definition of pseudo-systolic cells

We extended the concept of a systolic cell to include a small local storage. The new type of cell operates in such a way that cell bandwidth is a fraction of the computation rate. This property is attractive for VLSI implementation, specially for cells that have a pipelined operation unit. We call this a *pseudo-systolic cell*.

Development of design method

We developed a general design technique that follows a transformational approach and is based on the data-dependency graph of algorithms. This method is summarized in Figure 1.4. Starting from the description of a matrix algorithm, a *fully-parallel data-dependency graph* (FPG) is derived; in such a graph, nodes represent operations and edges correspond to data dependencies. This graph is obtained by symbolic execution of a representation of an algorithm, and its structure is simplified by the existence of matrix and vector operators.

A regularization process transforms the FPG into a three-dimensional graph with unidirectional dependencies. We call this a *multi-mesh data-dependency graph* (MMG). We showed that this regularization is always possible, due to characteristics of the operations that compose a matrix algorithm. We identified the possible irregularities and developed transformations to eliminate them. This regularization process is aided by the visualization of the graph representations.

In a second stage of the method, arrays are derived from the MMG by collapsing the three-dimensional graph onto a two-dimensional one (a *G-graph*), and implementing this G-graph in arrays. Collapsing determines the scheduling of operations in cells and data transfers from/to other cells and from/to local storage.

The second stage incorporates implementation restrictions and evaluates tradeoffs in properties of cells. Moreover, this stage allows deriving arrays for fixed-size data problems, partitioned problems, and mapping onto class-specific arrays, while performing optimizations of given performance/cost measures.

This method shares with other previously proposed techniques, such as the Signal Flow Graph (SFG) method by S.Y. Kung [Kung87a, Kung88c], the property of using the data-dependencies in an algorithm as the description tool. However, the method proposed in this dissertation answers questions still open in the SFG method, such as how to derive the dependency graph and what is the form of this graph, how to incorporate implementation restrictions (i.e., limited storage and limited bandwidth per cell), how to perform tradeoffs between local storage and cell bandwidth, how to effectively use pipelined cells, how to consider performance and cost measures while applying the transformations, what are the tradeoffs between linear and two-dimensional arrays for partitioned problems, and how to map algorithms onto class-specific arrays. Moreover, our method is more systematic and simpler to use than the SFG technique.²

²Results from an extensive comparison between the method proposed in this dissertation, the

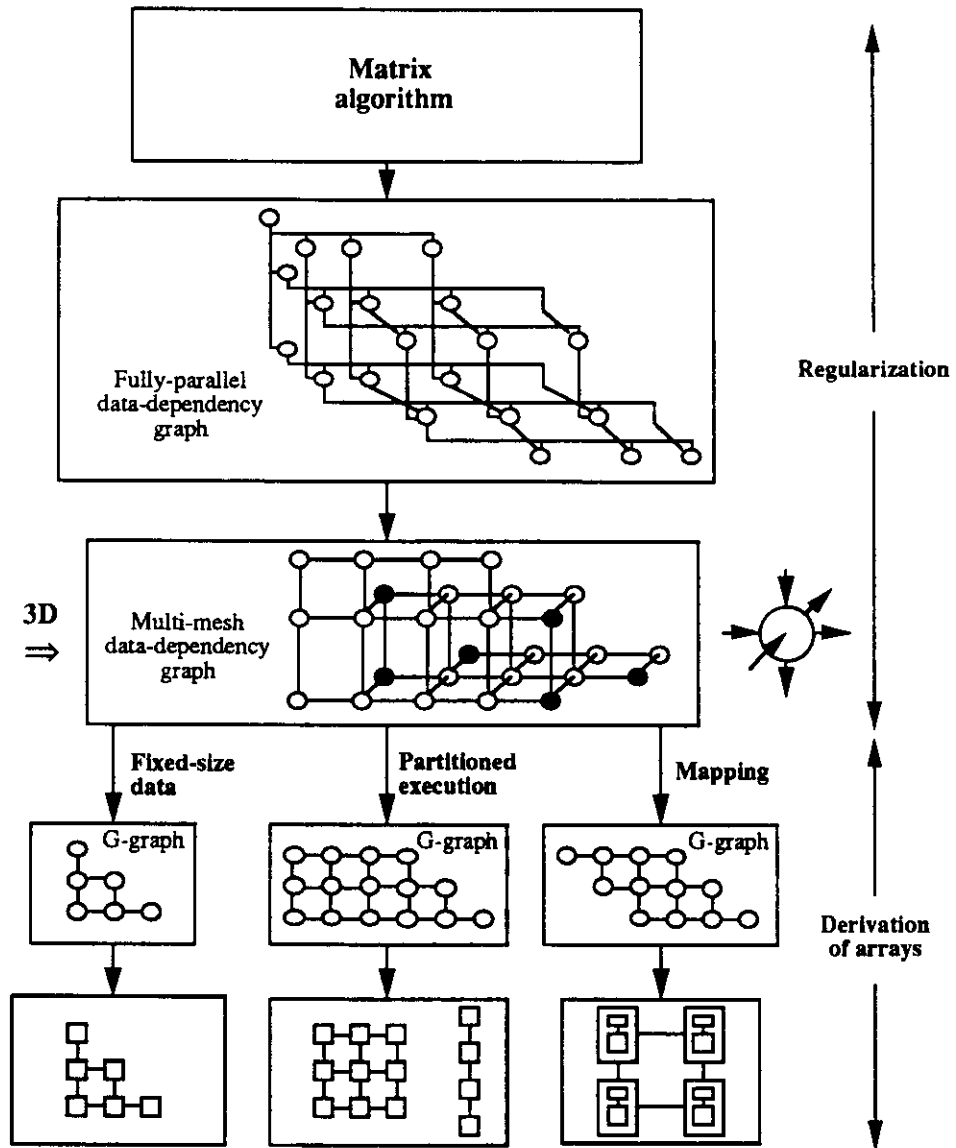


Figure 1.4: Data-dependency graph-based design method

Applicability of method to pseudo-systolic cells

The method developed is applicable to pseudo-systolic cells. Moreover, this method allows the organization of local storage in those cells as two FIFO buffers, performing tradeoffs between storage size and cell bandwidth, and the efficient use of pipelined operation units. Note that systolic cells are a particular case of pseudo-systolic ones, so that the method is applicable to them as well.

Preferability of linear arrays for partitioned execution

We showed that, for partitioned execution of matrix algorithms, linear arrays are advantageous over two-dimensional arrays (with the same number of cells) because while they have the same I/O bandwidth, they potentially provide a higher throughput, simplify the design process, and are more suitable to include fault-tolerance features.

Application of the method to a variety of algorithms

We applied the method to a variety of algorithms, including matrix multiplication, convolution, LU-decomposition (with and without pivoting), triangularization by Givens' rotations, Cholesky decomposition, transitive closure, Faddeev algorithm, and computation of BA^{-1} . Through these examples, we determined that the proposed method is easy to apply, incorporates implementation requirements, and optimizes selected performance measures. Moreover, the application of the method led to the derivation of new and more efficient arrays for specific algorithms (such as transitive closure and BA^{-1}) than what has been previously proposed in the literature, as well as to the systematic derivation of arrays obtained in an ad-hoc manner by other researchers.

Formalization of the method

The transformations that compose the method outlined above were formalized. For these purposes, a *canonical representation of matrix algorithms* was introduced, and the *equivalence of graphs* derived through the transformations was proved. In this process, it was demonstrated how the fully-parallel data-dependency graph of a SFG technique, and another approach are described later.

matrix algorithm is transformed into a three-dimensional graph with unidirectional dependencies, and conditions for achieving such a representation were determined.

Comparison with other design methods

We identified two stages of which any design technique is composed, namely *algorithm regularization* and *derivation of arrays*. The identification of these stages led us to devise a framework that allows evaluating design methods for application-specific arrays. Methods based on *index-dependencies* [Rao88, Quin84, Mold83, Capp84] and *data-dependencies* [Kung88c] were compared in this two-stage framework. The comparison uses the method for Regular Iterative Algorithms (RIA) [Rao88] as a representative example of index-dependencies, the Signal Flow Graph method [Kung88c] and our MMG method as data-dependencies based approaches. We concluded that *our method is advantageous*, because it meets the evaluation criteria and *produces arrays that are more efficient* than those obtained with the other techniques.

Linear array for partitioned execution

We proposed a *linear canonical array*, suitable for partitioned execution of matrix algorithms, that achieves high utilization, uses pipelined cells, and allows an off-cell communication rate lower than computation rate. All these characteristics are desirable for an implementation. Moreover, we described mapping algorithms onto such an array using our data-dependency based method. Cells of the proposed array consist of a pipelined functional unit, internal storage in the form of FIFO buffers, and queues attached to ports. The array is composed of a set of cells arranged in a linear structure, support for external I/O into the array, and memory modules external to cells.

As an example, we described a cell whose functional unit is composed of a conventional floating-point multiplier and an ALU. This cell was used in the mapping process, producing high utilization of resources and exploiting the internal pipeline in a simple manner. Mapping was illustrated using algorithms such as LU-decomposition, and triangularization by Givens' rotations. Performance estimates of the resulting implementations show that, for example, LU-decomposition of a 200 by 200 matrix computed in an array with 10 cells and 4-stage pipelines achieves about 90% utilization. Consequently, assuming a clock cycle of 50 [nsec], such an array delivers 360 [Mflops] out of a peak capacity of 400 [Mflops].

Mapping matrix algorithms onto local-access arrays

The suitability of the method for mapping problems onto local-access arrays was also addressed. Differences in mapping onto arrays with pseudo-systolic or systolic and local-access cells were discussed, as well as the impact that an architecture has on the mapping process and how the method can be adapted for a given architecture. In particular, having large storage per cell permits using coalescing as the partitioning approach. We described three strategies to perform coalescing, discussed their suitability in producing good load balancing, and devised a heuristic approach for such a purpose. To illustrate these issues, we considered a hypothetical memory-linked array, where cells are interconnected by memory modules of large capacity (an example of such an architecture is MWAP [Kung87b]).

Publications

The contributions listed above have originated the following publications:

1. J. Moreno and T. Lang, "Comments on 'A systolic array for computing BA^{-1} ,'" *IEEE Transactions on Acoustics, Speech and Signal Processing*, November 1989.
2. J. Moreno and T. Lang, "A linear array for partitioned execution of matrix algorithms with high utilization," in *SPIE Real-Time Signal Processing XII*, August 1989.
3. J. Moreno and T. Lang, "Comparing design methods based on index dependencies and data-dependencies," in *International Conference on Systolic Arrays*, pp. 599–608, May 1989.
4. J. Moreno and T. Lang, "Arrays for partitioned matrix algorithms: trade-offs between cell storage and cell bandwidth," in *SPIE Real-Time Signal Processing XI*, pp. 156–169, August 1988.
5. J. Moreno and T. Lang, "Partitioning algorithms for systolic arrays: application to transitive closure," in *International Conference on Parallel Processing*, pp. 28–31, August 1988.
6. J. Moreno and T. Lang, "On partitioning the Faddeev algorithm," in *International Conference on Systolic Arrays*, pp. 125–134, May 1988.

7. J. Moreno and T. Lang, "Design of special-purpose arrays for matrix computations: preliminary results," in *SPIE Real-Time Signal Processing X*, pp. 53–65, August 1987.
8. J. Moreno and T. Lang, "Graph-based partitioning of matrix algorithms for systolic arrays," Technical Report CSD-880015, Computer Science Department, University of California Los Angeles, March 1988.
9. J. Moreno and T. Lang, "Designing arrays for the Faddeev algorithm," Technical Report CSD-880013, Computer Science Department, University of California Los Angeles, March 1988.
10. J. Moreno and T. Lang, "Reducing the number of cells in arrays for matrix computations," Technical Report CSD-880014, Computer Science Department, University of California Los Angeles, March 1988.
11. J. Moreno and T. Lang, "Removing algorithm irregularities in the design of arrays for matrix computations," Technical Report CSD-870040, Computer Science Department, University of California Los Angeles, August 1987.
12. J. Moreno, "A proposal for the systematic design of arrays for matrix computations," Technical Report CSD-870019, Computer Science Department, University of California Los Angeles, May 1987.

1.3 Organization of the dissertation

Issues related to the design and implementation of arrays for matrix computations are discussed in Chapter 2. For these purposes, we consider architectures that include the systolic array model, as originally proposed, and introduce the pseudo-systolic extension with local storage. For those aspects that appear in the design of arrays, a classification is proposed which distinguishes between restrictions for a particular implementation and controlled/uncontrolled parameters.

In Chapter 3, we center our attention on systematic design approaches. A classification of design techniques due to Fortes et al. [Fort88] is reviewed first, and the limitations of that classification for the purposes of comparing design methods are discussed. As a way to overcome those limitations, a framework is proposed which allows comparing methods by identifying two stages in the application of a design technique: regularization of algorithms and derivation of arrays. Criteria to evaluate the suitability of methods under this framework are indicated, and

some of the approaches proposed in the literature are reviewed in terms of those criteria. Such criteria also constitute a set of guidelines for the development of a powerful design method. We highlight the convenience of using dependencies in an algorithm as the basis for a design technique, and discuss the properties of data-dependencies and index-dependencies for such purposes.

A description of our design method is given in Chapter 4. The different steps are presented, as well as the impact on array performance and cost implied by the transformations used in those steps. This description is illustrated with the triangularization algorithm using Givens' rotations. Moreover, this chapter describes a canonical linear array for partitioned execution of algorithms with high utilization, which is well suited for the application of the method. Chapter 5 presents a formalization of the method, including a canonical representation of matrix algorithms and proofs for the equivalence of graphs derived through the transformations that compose the method. This chapter also demonstrates how the regularization stage is achieved, that is, how the fully-parallel graph of a matrix algorithm is transformed into a multi-mesh graph. Conditions that guarantee achieving such a representation are given. Chapter 5 uses LU-decomposition without pivoting to illustrate the steps in the formalization of the method.

The suitability of our method to map algorithms onto linear local-access arrays is addressed in Chapter 6. We discuss mapping the LU-decomposition algorithm onto a memory-linked array such as MWAP [Kung87b]. This example illustrates that it is possible to obtain high utilization of hardware resources in a given array.

In Chapter 7, we compare our data-dependency based method with other techniques, in particular Rao's method for Regular Iterative Algorithms (RIAs) [Rao85, Rao88] as a representative example of index-dependency based approaches, and S.Y. Kung's Signal Flow Graph method [Kung87a, Kung88c]. In this comparison, we identify the outcome of the regularization stage and evaluate the ease of obtaining a regular description in each method. Then, we discuss the derivation of arrays and the ways that such a process achieves its objectives in each case. As an illustration, the application of the three methods to the transitive closure algorithm is presented and the results obtained are discussed.

This dissertation ends by summarizing the results obtained as well as suggesting further work in the area. Additional examples of application of the method are given in Appendices, in particular the Faddeev algorithm, the computation of BA^{-1} , the LU-decomposition algorithm with neighbor pivoting, and a convolution algorithm with affine dependencies.

CHAPTER 2

The design of arrays for matrix algorithms

Since the introduction of the concept of *systolic arrays* [Kung78], much research has been performed on designing algorithms and architectures suitable for that model of computation. In this chapter, we concentrate on the design of mesh arrays (i.e., systolic-type) for matrix algorithms. This discussion considers architectures that include the systolic array model, as originally proposed, and extensions such as arrays with external memory and cells with local memory.

2.1 Matrix algorithms

A canonical description of matrix algorithms is given in Chapter 5. Briefly, matrix algorithms are characterized by the following properties:

- They constitute a compute-intensive class, with matrix operands and matrix results.
- They exhibit fine-grain parallelism suited for implementation in systolic-type arrays.
- They consist of primitive operations with up to three operands and up to two results.

We obtained the last property listed above as a conclusion from analyzing a large class of matrix algorithms: the execution of these algorithms consists of *unary, binary and/or ternary operations that produce one or two results each*. Consequently, these operations may be represented by nodes in a graph, with at most three inputs and two outputs per node, leading to graphs that are three-dimensional. This is an important characteristic of our design method, as is described in Chapter 4, and it determines several properties of arrays as discussed in this chapter.

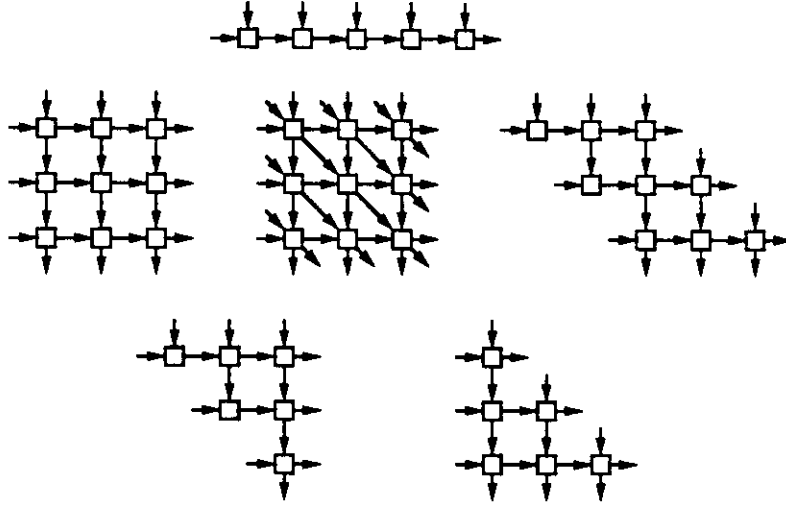


Figure 2.1: Examples of array structures

2.2 The architectural model of mesh arrays

There is no single formal definition of systolic-type arrays (i.e., mesh arrays) that is widely accepted. Moreover, a variety of architectural features (not necessarily compatible) have been considered key properties in defining this type of structure. In this section, we describe the properties of the architectural model of mesh arrays used in this dissertation.

Nearest-neighbor connections, no broadcasting

Arrays are collections of processing elements connected in a nearest-neighbor manner, with external I/O from a host only at the boundaries of the array. Moreover, these arrays have only local communications (i.e., there is no capability for broadcasting or routing data through cells without using that data).

Arrays can be linear, rectangular, hexagonal, trapezoidal or triangular structures, as shown in Figure 2.1.

Mesh arrays, unidirectional communications

We consider only mesh-connected arrays, either linear or two-dimensional (a linear array with K cells is a mesh of dimension K by 1). It will be shown in

Chapter 4 that matrix algorithms do not need higher connectivity, such as that available in hexagonal structures. Moreover, flow of data in the arrays considered here is unidirectional, that is, data flows from cell to cell in one direction only, without data counterflow. Consequently, cells of linear and two-dimensional regular structures have two input and two output ports.

Characteristics and types of cells

According to the properties of matrix algorithms given in Section 2.1, cells of a mesh array need to be able to read up to three operands and produce up to two results per operation. Moreover, outputs from a cell are either results computed within the cell or operands used in the cell and passed through without modification (i.e., transmittent data [Kung88c] (p. 118)). Since cells have only two input and two output ports, as indicated above, the third port is obtained by a *feedback loop* within the cell.

We assume that non-pipelined cells take the same amount of time to perform any operation. On the other hand, we assume that the stage time in pipelined cells is the same for all operations. These assumptions, which have usually been used for the design of application-specific arrays, are highly implementation-dependent, as recent studies of the design of application-specific cells have suggested [Erce87a, Erce87b, Cava87].

Cells of a mesh array belong to one of the following three types:

Systolic cell: *a cell with no local storage except for registers used to latch input operands for an operation.* Figure 2.2a depicts a systolic cell with its corresponding input/output ports. For ternary operations (those requiring three operands), two operands are received from outside a cell through ports and the third source of data is the feedback loop within the cell. For unary and binary operations, only one or two sources of data are active (i.e., carry data for such operations). Data flows through cells in such a way that *every operation in each cell requires one data transfer per active data source.*

Pseudo-systolic cell: *a cell with small, fixed-size storage (i.e., storage size is independent of the size of problems to be solved in the array).* This storage is composed of two separate FIFO buffers. Figure 2.2b depicts a pseudo-systolic cell with its ports and buffers. Ports and local storage provide two

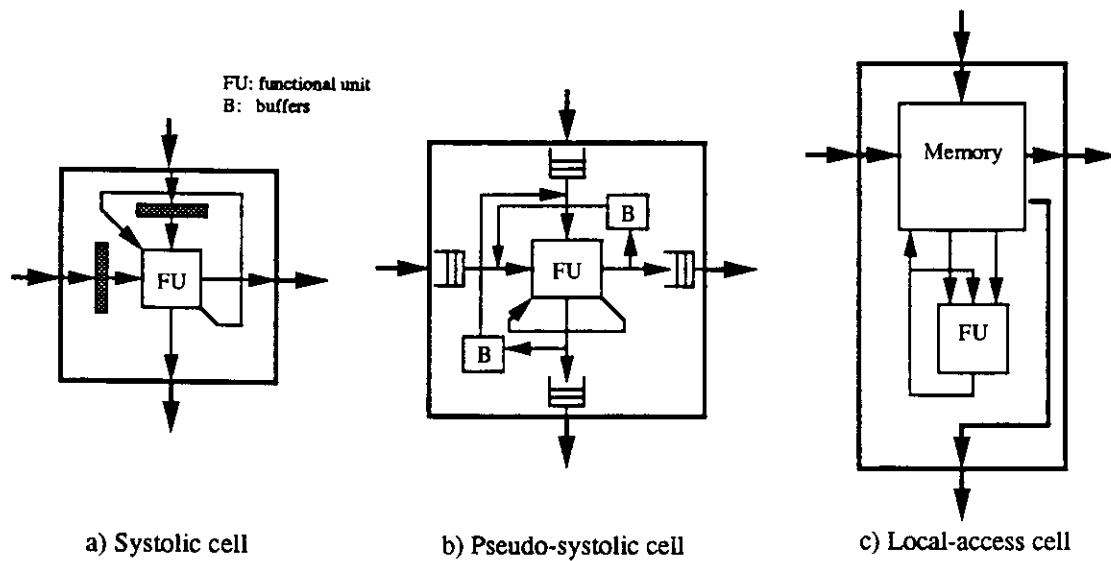


Figure 2.2: Cells for the different types of arrays

sources of data for every operation. Another source of data is the feedback loop within the cell.

Since the size of the FIFO buffers is fixed and small, we assume that access time to this local storage matches the functional unit execution rate (i.e., cell pipeline stage time or functional unit time) and that it is shorter than the time to transfer data among cells. This property is exploited by performing operations with data from the buffers. Consequently, pseudo-systolic cells do not need to receive data through ports at every cycle, so the *communication bandwidth of pseudo-systolic cells is lower than their computation rate*. This lower communication rate is adjusted to cell computation rate by FIFO queues attached to ports.

Local-access cell: *a cell with storage whose size is proportional to the size of problems to be solved in an array.* Figure 2.2c depicts a local-access cell. Operations are performed in each cell with up to two operands obtained from local storage, so that data received from neighbor cells is stored before it is used. Another source of data is the feedback loop within the cell.

Local-access cells have large local memory with the objectives of storing a large portion of data locally and reducing communication among cells. Consequently, *communication rate among cells is much lower than computation rate* (i.e., much less than one word per port per time-step).

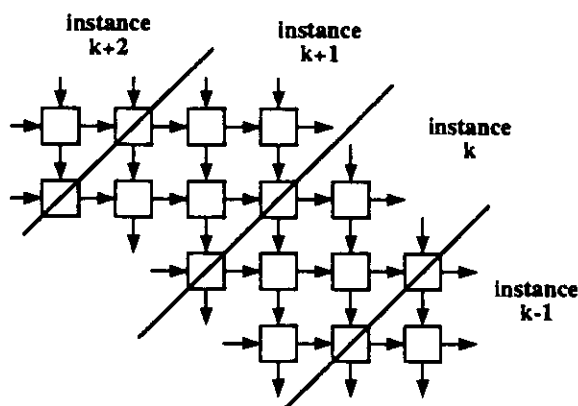


Figure 2.3: Computational model for multiple instances

Model of computation

The model of computation used in mesh arrays consists of a synchronized flow of data through cells, with operations performed in each cell. At each time-step, a cell reads operands from input ports, local storage, and/or the feedback loop, performs an operation, and delivers results to output ports, local storage, and/or the feedback loop. If the cell has a pipelined functional unit, the model of computation is also as just described except that the results delivered to ports, feedback loop and local storage are from an operation previously initiated in the pipeline.

This model of computation exploits parallelism and pipelining in the execution of an algorithm. Moreover, this model is suitable for the execution of *multiple-instance algorithms*, that is, algorithms that are executed repeatedly for different sets of input data. In such a case, one instance may use a cell during several time-steps, and different instances may be in concurrent execution throughout the array. Figure 2.3 depicts the flow of several instances through a mesh array.

In addition, this model is suitable for the partitioned execution of large problems in a small array. In this case, the different instances correspond to *subproblems* of the large problem.

2.3 Tradeoffs in throughput, cell storage and cell bandwidth

An important conclusion is readily available from the properties of cells described in the previous section: *if there are enough cells, systolic arrays can exploit*

all the parallelism in an algorithm. In a systolic cell, a data element is used in one time-step; at the next time-step, that element is immediately re-used in the same cell for another operation or transferred to another cell because there is no place to store data.¹ Consequently, every data element may be used in one cell at every time-step and the number of cells may match the size of the problem.

In contrast, cells with local storage hold data elements which are not used for an operation at a given time-step, so the use of parallelism is less than maximal.² As a result, the number of cells doesn't match the size of the problem. That is, for a given problem size, a systolic array may have more cells than a pseudo-systolic or a local-access array.

If the number of cells is less than the maximum determined by the size of the problem, then different arrays may provide the same throughput regardless of the type of cell used, as long as cells have the same step-time. The main differences between these arrays are the *tradeoffs between communication bandwidth and local storage*, and the *ability to use pipelined cells without increasing cell bandwidth* by operating with data from local storage. On one extreme, systolic cells require high communication bandwidth (equivalent to the computation rate) and no local storage. On the other end, local-access cells have a large local memory and low communication rate. Pseudo-systolic cells fall somewhere in between with lower bandwidth than systolic cells and little local storage. These properties are depicted in Figure 2.4 and described quantitatively in Table 2.1 for a matrix algorithm that consists of n^3 operations, where n is the dimension of the matrix. Using the method described in Chapter 4, realizing such an algorithm as an array with K cells leads to the results indicated in the table (the origin of those values are presented in Chapter 4). Table 2.1 shows that adding local storage to cells reduces communication bandwidth proportionally to the inverse square-root of the local storage size.

Table 2.1 also indicates the most suitable implementation for each type of cell, based on communication and storage requirements. For example, systolic cells are better implemented in WSI because that technology can provide communication bandwidth between cells of a similar magnitude as computation rate (as long as there is no need to go off-wafer). On the other hand, pseudo-systolic cells may be implemented in WSI or as one cell per chip. In the latter case, the cell may provide

¹Some cells may be performing delay operations, if dependencies in the algorithm do not allow the computation of a useful operation.

²We assume that no data are duplicated within the array, and consequently it is not possible to have a local copy of an element that has also been transferred to another cell. As far as we know, this is the case for all arrays proposed in the literature.

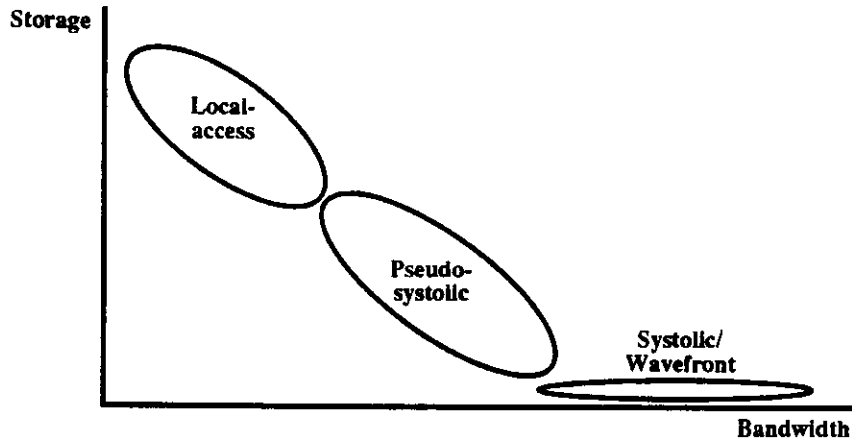


Figure 2.4: Tradeoffs between local storage and cell bandwidth

Table 2.1: Quantifying tradeoffs between local storage and cell bandwidth

	Systolic cell	Pseudo-systolic cell	Local-access cell
Storage per cell	None	$\approx S$	$\approx n^2/K$
Cell communication bandwidth per port [words/time-step]	1	$1/\sqrt{S}$	\sqrt{K}/n
Most suitable implementation	WSI	WSI or cell per chip (functional unit and buffer)	Board-level implementation

K : number of cells

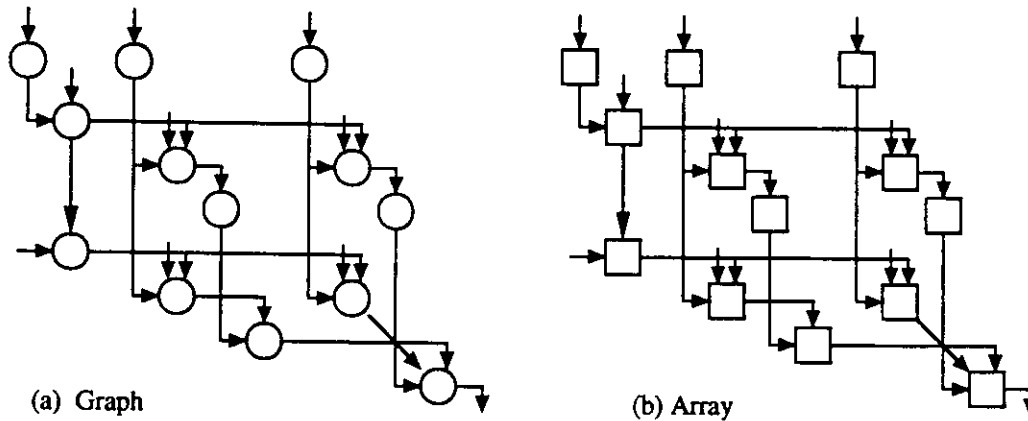


Figure 2.5: The direct realization of a dependency graph

high computation rate (using data from local buffers) and lower communication bandwidth between cells (which might require going off-chip). Finally, local-access cells are better implemented at the board level, because they require large memory modules that are less suited for single chip VLSI/WSI implementation.

2.4 Realizing algorithms and mapping algorithms onto arrays

A simple approach to derive an application-specific implementation for a matrix algorithm is to represent the algorithm as a graph (i.e., one node per operation and one edge per dependency) and perform a *direct realization* of this graph. That is, *each node of the graph (corresponding to an operation) is assigned to a different cell in the array, and each edge (dependency) is assigned to a different link*, as depicted in Figure 2.5. Such an implementation is suitable for pipelined execution of multiple instances of the algorithm, and exhibits the following characteristics:

Advantages

- Optimal utilization (for multiple instances), because each cell is used in a different instance of the algorithm at each time-step.
- Maximum throughput, given by the computation time of a node.
- Minimum computation time, given by the longest path in the graph.

Disadvantages

- Large number of cells, depending on the size of the problem.
- Possibly complex and irregular interconnection, determined by the dependencies in the algorithm.
- High I/O bandwidth, since all inputs appear simultaneously.
- Possibility of undesirable features for an implementation, such as data broadcasting or large fan-out.

Consequently, the objectives of the design process are to find a regular structure that requires fewer cells than the direct realization, and to remove the complex interconnections and other undesirable characteristics. Since there are fewer cells, it becomes necessary to map the operations in the algorithm onto array cells and time-steps. In turn, this process requires determining the characteristics, topology and interconnection of cells, specifying where and when each operation is performed, and specifying how data flows through cells. All of these steps in the design must be performed while preserving the algorithm dependencies.

From the discussion above, we can infer that the design of an array encompasses two aspects:

Architecture, that is, obtaining the characteristics of modules composing the array, their topology, and their interconnection (communication). These modules include processing cells, memories, and I/O ports. In devising an architecture, it is necessary to consider the characteristics of the algorithm (such as type of operations and dependencies), taking into account constraints arising from the technology used in the implementation.

Mapping (spatial and temporal) of the algorithm onto the architecture, that is, specifying what operations are performed in each cell and in what order, and specifying the flow of data through the array. This mapping must be such that no two operations are assigned to the same cell and expected to be executed at the same time. Consequently, mapping requires considering not only characteristics of the algorithm but also characteristics of the architecture.

These two aspects of a design exhibit different properties, depending on the range of applicability of the target array. For *algorithm-specific arrays* (i.e., those

that execute a single algorithm), devising architecture and mapping are inter-related tasks carried out simultaneously. Normally, there is no clear separation between the two; a designer considers both aspects as a single entity. We refer to this process as the *realization* of an algorithm as an array.

On the other hand, a *class-specific array* (i.e., an array suitable for a class of selected algorithms) requires only mapping, because in that case a generalized architecture is defined in advance. Consequently, an implementation on a class-specific array consists of devising a way to use the available resources. The results of performing the mapping are instructions (software) that specify the flow of data and the sequencing of operations in the different cells, so that mapping onto a class-specific array requires a suitable programming environment.

2.5 The range of applicability of arrays

As indicated in the previous section, arrays for matrix computations are classified into two groups depending on their range of applicability:

Algorithm-specific arrays (ASAs) are designed (and used) for one particular computation. Arrays of this type are reported in [Hein87, Kand88, Lopr88, Lack88, Lewi88, Schi86, Chou88].

Class-specific arrays (CSAs) are designed to execute a class of specific algorithms. Examples include SLAPP [Drak87, Syma86], devised to compute QR-decomposition, SVD, and GSVD, and the Hughes systolic/cellular system [Nash88, Przy88], originally devised for computing the Faddeev algorithm and SVD. More general arrays are Warp [Anna87], ESPRIT [Groo87], Matrix-1 [Foul87], VATA [Syma88], the arrays reported in [Avil83, Blac81], and the earlier two-dimensional systolic-array testbed [Syma83].

If a single matrix computation is to be performed in a system, an algorithm-specific array has the potential advantage of producing higher throughput and better utilization than a class-specific array, because the topology and the cells can match the particular characteristics of the algorithm. Moreover, little or no programming would be required, because most of the sequencing and control are embedded in the logic of the system. An ASA can also fulfill other requirements at a lower cost. On the other hand, using a CSA for the application might reduce

the design and fabrication cost of the system, because a CSA for algorithms of the same class may already exist.

If several predefined computations have to be performed, a CSA might have the advantage of reducing the cost of the system compared to having several ASA. However, the CSA might have a lower performance because it has to adapt to the characteristics of several algorithms.

Finally, for the cases in which the computations are not predefined, a class-specific array is preferable, because today's technology makes it easier to program a new algorithm in this type of system rather than designing and constructing a new system.

Fortes and Wah [Fort87b] state that there are two approaches in designing class-specific arrays:

- Adding hardware mechanisms to reconfigure the topology and interconnection pattern of the array, in order to emulate the requirements of an algorithm-specific design. Examples of this approach are the Configurable Highly Parallel computer (CHiP) [Snyd82], which has a lattice of programmable switches for reconfiguration purposes, and the Programmable Systolic Chip (PSC) [Fish83].
- Mapping different algorithms onto an array using software. This approach has been used in Warp [Anna87], SLAPP [Drak87] and ESPRIT [Groo87], and proposed for Matrix-1 [Foul87] and VATA [Syma88].

2.6 Approaches to partitioning large problems

In most applications it is necessary to perform a computation with large-size data in an array that is smaller than the size of the data, or map an algorithm with large variable-size data onto a small array [Nava87, Hell84, Kung88c]. The basic approach to solving both cases consists of decomposing (i.e., partitioning) the original problem into subproblems that fit the size of the target array. Such a decomposition is shown conceptually in Figure 2.6, where the dependency graph of an algorithm is represented by a parallelepiped (i.e., nodes in the dependency graph are distributed in a parallelepiped).

The basic approaches to achieve partitioning are:

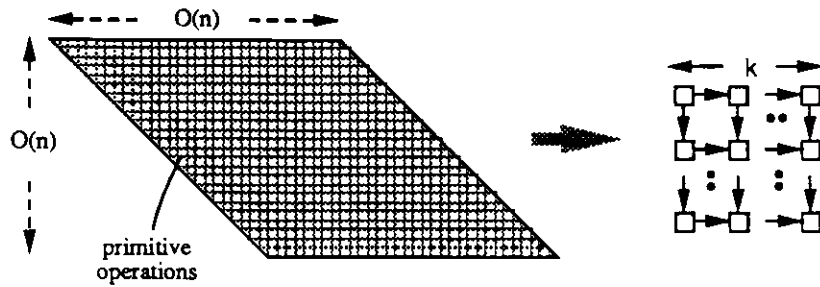


Figure 2.6: Partitioning an algorithm

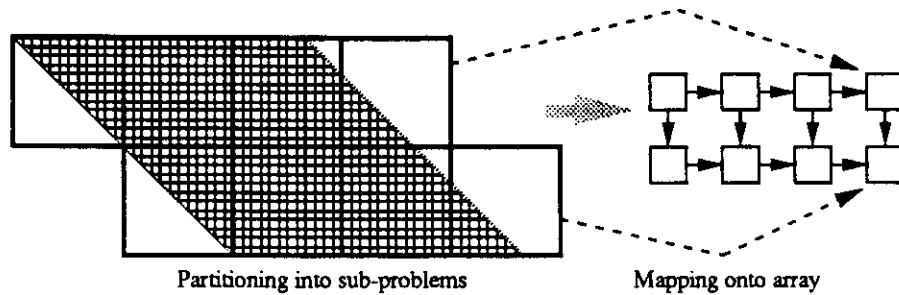


Figure 2.7: Partitioning an algorithm through coalescing

- a) **Coalescing.** This approach *partitions the algorithm into a number of subproblems equal to the number of cells available in the target array*. The dependencies between the subproblems (i.e., communication requirements) should match the interconnection structure of the array, and each subproblem is mapped onto one cell. Each cell sequentially executes the operations in its subproblem, according to a certain schedule. Figure 2.7 depicts this technique. This figure shows the dependencies in an algorithm (depicted as a parallelepiped), which are partitioned into a number of communicating subproblems that are mapped onto the array.

This type of partitioning has also been referred to as *locally sequential globally parallel partitioning (LSGP)* [Kung88c]. The more intuitive name *coalescing* has been frequently used as well [Nava87, Nash86a]. The scheme is attractive for its simplicity, generality and low communication bandwidth, but it requires large local storage within each cell (large enough to store all of the data of the corresponding subproblem). Consequently, such a scheme is suitable for implementations using local-access cells. Moreover, this scheme requires a careful selection of subproblems to achieve good load balancing.

- b) **Cut-and-pile.** This approach is actually a two-level scheme, where the *algorithm is decomposed into subproblems and each subproblem into components*.

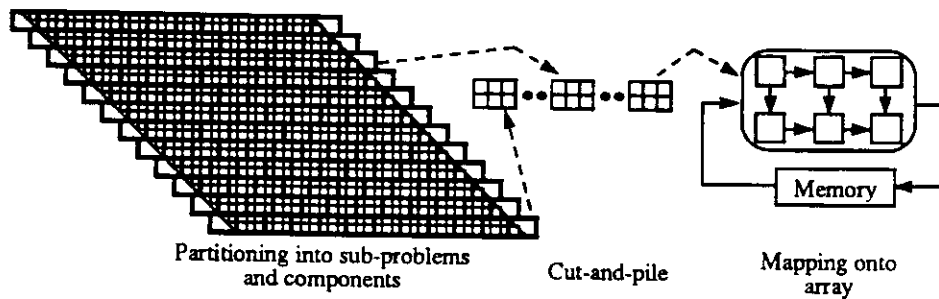


Figure 2.8: Partitioning an algorithm through cut-and-pile

A subproblem is mapped onto the entire array and each component is mapped onto a different cell. Subproblems are mapped sequentially, according to a certain schedule. Such a sequential mapping usually requires feedback of data and, in many cases, memory external to the array. Figure 2.8 illustrates this scheme, whereby an algorithm is partitioned into subproblems whose components exhibit communication requirements in a rectangular pattern (except for the boundaries of the algorithm). Consequently, these subproblems are mapped onto a rectangular array.

This type of partitioning has also been referred to as *locally parallel globally sequential partitioning (LPGS)* [Kung88c]. The more intuitive name *cut-and-pile* was coined in [Nava87]. This approach is attractive because it is general, it does not require memory in each cell, and it produces good load balancing. It is suitable for partitioning algorithms for execution in mesh arrays with systolic cells, which require high cell bandwidth. However, data must be fed back into the array.

- c) **Combination of coalescing and cut-and-pile.** An alternative that allows combining the benefits of the two techniques above was proposed in [More88a]. This approach consists of a three-level process, where the granularity of an algorithm is reduced by *applying coalescing to a limited extent*, partitioning the coalesced version into subproblems, and dividing these subproblems into components. Subproblems and components are mapped onto the array as in cut-and-pile.

This approach, illustrated in Figure 2.9, produces arrays with pseudo-systolic cells (which require a lower cell bandwidth than systolic cells). Moreover, pipelined cells can be used efficiently. Drawbacks are the need for data feedback, and some complexity in the control (sequencing) within cells.

This technique will be elaborated as part of the design method presented in Chapter 4.

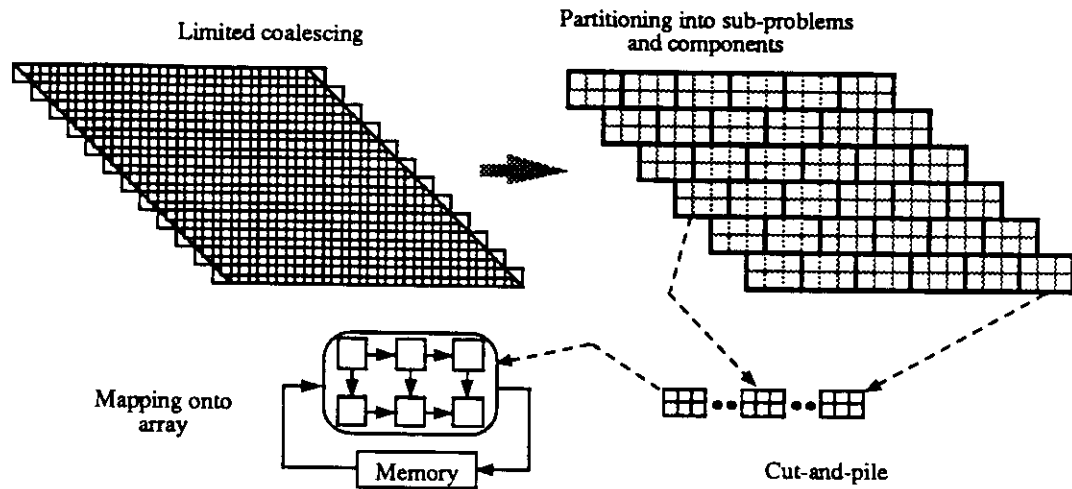


Figure 2.9: Partitioning an algorithm through coalescing and cut-and-pile

d) **Decomposing the algorithm into subalgorithms.** In this approach, the original algorithm is decomposed into subalgorithms. Such subalgorithms, which need not be the same as the original one, are executed sequentially in the array according to a certain schedule. Consequently, this approach transforms the original algorithm into a series of subproblems which, when executed, provide the same result as the original algorithm.

An example of this approach was proposed by Navarro et al. [Nava87] and is shown in Figure 2.10. In this case, an algorithm with large size dense matrices is transformed into an algorithm with band matrices and computed in an array tailored to the band size. This approach has the potential for high performance when applicable, but is less general than the ones discussed above because the decomposition depends on the algorithm.

Table 2.2 summarizes the advantages and disadvantages of the different partitioning approaches.

The first three partitioning approaches described above can be performed using either a *direct* or an *indirect* strategy, as shown in Figure 2.11. In the indirect approach, an algorithm is first realized as a large (i.e., virtual) array whose size depends on the size of the data. This array is then partitioned and mapped onto the small array. As a result, partitioning is performed not on the algorithm, but on an array that implements the algorithm for large-size data. This is in contrast to the direct approach, where the algorithm is directly partitioned for execution on a small array.

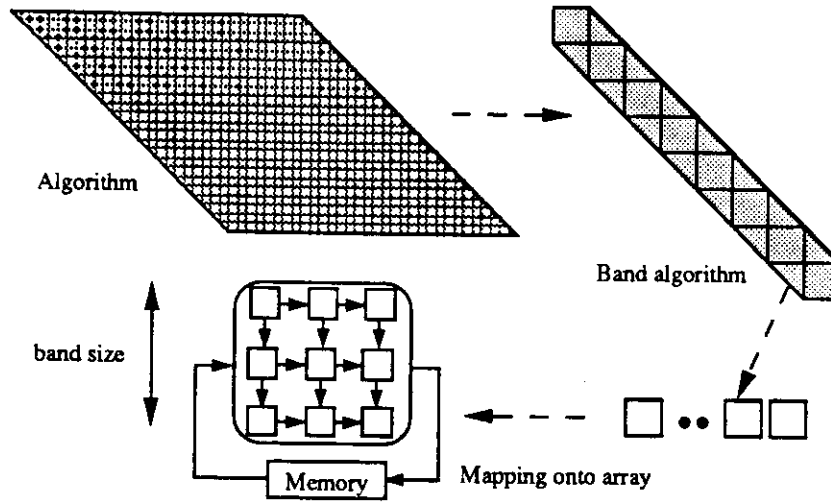


Figure 2.10: Partitioning an algorithm through decomposition into subalgorithms

Table 2.2: Comparison of partitioning approaches

Method	Advantages	Disadvantages
Coalescing	Simple. General. Low communication bandwidth.	Large storage per cell.
Cut-and-pile	General. Storage external to array.	Feedback of data. High communication bandwidth.
Coalescing/cut-and-pile	General. Small memory per cell. Lower communication bandwidth than cut-and-pile. Storage external to array. Allows pipelining within cells.	Feedback of data.
Decomposition into subalgorithms	Potentially good performance.	Lack of generality. Complex. Feedback of data.

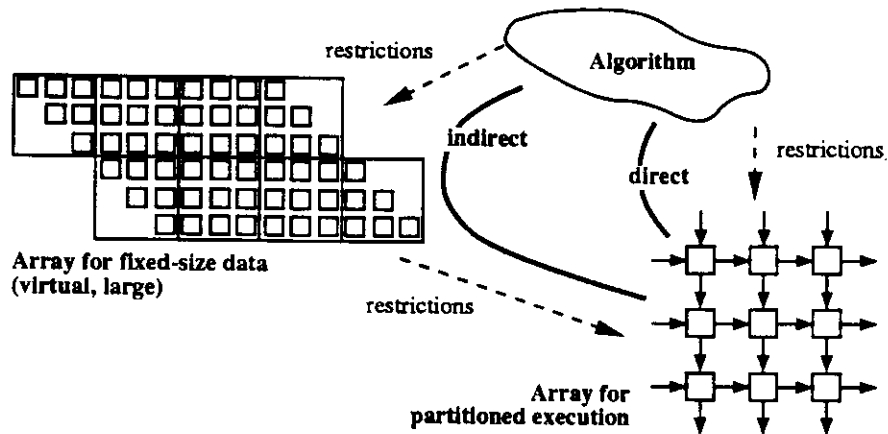


Figure 2.11: Indirect and direct partitioning

The differences between direct and indirect partitioning strategies can be stated in terms of flexibility to achieve the task. In the indirect approach, the techniques to obtain the virtual array are the same ones used to derive arrays for fixed-size data. Consequently, one first obtains an array suitable for fixed-size data; this imposes conditions on the array which might be detrimental for the partitioning step. On the other hand, partitioning the algorithm directly onto an array uses all the information available in the representation of the algorithm, and therefore can produce a better implementation.

In other words, direct partitioning is more flexible because it can use properties of the algorithm that are not suitable for fixed-size data (and are therefore eliminated in the indirect approach). The difference is partly due to the fact that at a given time, a partitioned design uses a very small portion of the parallelism available in the algorithm, while a fixed-size data design uses a much larger portion. Consequently, the irregularities in the algorithm have a much larger influence in the fixed-size data case. As shown in Figure 2.12, a partitioned design must center its attention on efficient execution of the internal regular part, because that part accounts for most of the computational load imposed by the algorithm. In contrast, the fixed-size data design has to cope with the irregularities.

In this dissertation, we deal only with direct partitioning techniques.

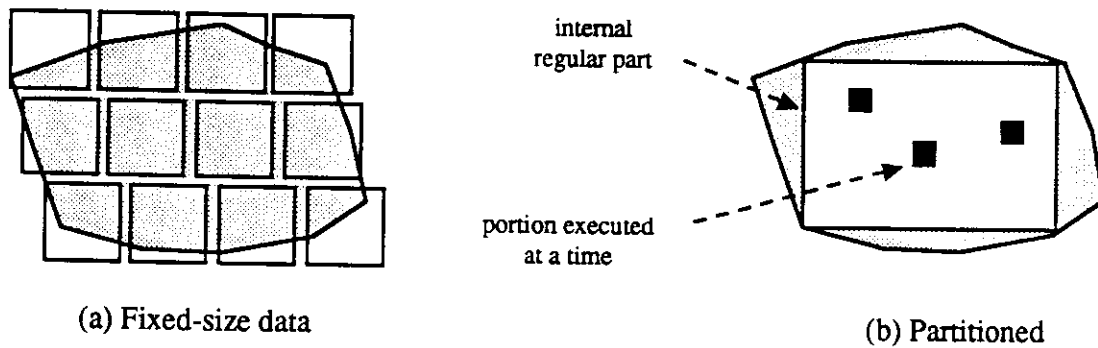


Figure 2.12: Using the internal regular part of an algorithm in partitioning

2.7 Issues in the design of arrays

The design of a digital system is usually accomplished with a structured decomposition process. A high-level specification of the system (behavior and structure) is refined through a top-down design procedure, leading from complex components to simpler subcomponents. These components are then implemented and the system is evaluated in terms of cost, performance, and other requirements. The design process is repeated if the resulting implementation is not satisfactory. Since there are many parameters to consider which influence the implementation in a complex way, the design process is time-consuming and costly. This is a very important consideration in the cost-effectiveness of a system, especially in the case of application-specific implementations where design cost is amortized with a small production volume. As a consequence, it is imperative for these systems to have a method and tools that reduce the design cost and time.

Mesh arrays are simpler to design than other systems, because it is possible to proceed faster and more directly to the design of lower-level components than in traditional design [Fort87b]. This is because a mesh array consists of a large number of a few types of modules interconnected in a predefined manner. Moreover, it is easier to evaluate the resulting implementations during the design process, and it is therefore possible to reduce the number of design iterations. These characteristics make mesh arrays especially attractive for application-specific systems in those cases where the mesh organization and computation model do not have a negative impact on cost and performance.

Typical requirements for mesh arrays are classified in Table 2.3. For a given implementation, some of these are actual requirements, others are used as opti-

Table 2.3: Typical requirements for mesh arrays

Performance:	Throughput Computation time
Cost:	Number of cells Utilization Overhead Types of Cells Complexity of cells Bandwidth
Other:	Domain of applicability Reliability Size Power dissipation Expandability

mization criteria, and the rest are ignored. The division into the three classes shown in Table 2.3 is dependent on the system being implemented. In one case utilization of processing elements might be the most important measure, while in other cases higher priority might be given to throughput or computation time. Consequently, cost and performance measures are items that have to be handled during the design process, both in defining the specific measures of interest and determining the values obtained.

Another important aspect is the integration of arrays into existing systems, which may be nontrivial [Fort87b]. Issues that may affect integration are:

- Extensive I/O bandwidth.
- Interconnection with the host.
- Memory subsystem supporting the array.
- Buffering and access of data to meet special input/output distributions.
- Multiplexing/demultiplexing data for insufficient I/O ports.

The design of an array must deal with these issues and devise suitable mechanisms to allow integrating array and host. This may require additional hardware support, such as queues for data transfers or memory external to the array. Specific solutions depend on characteristics of array and host.

Table 2.4: Typical parameters for mesh arrays

Class of admissible computations
Type of cell (systolic, pseudo-systolic, or local-access)
Dimensionality of the array (linear or two-dimensional)
Size of the array
Use of identical or specialized cells
Width of the communication path
Size of cell buffers
Cell bandwidth
Degree of cell pipelining
I/O bandwidth and data format
Cell pins

A particular mesh array is described by many parameters. The values of these parameters are determined during the design process in such a way that the implementation satisfies the requirements. Typical parameters for mesh arrays are given in Table 2.4. Since the number of parameters and their possible values is large, a manageable design process requires limiting the solution space. This is done by dividing the parameters into three classes as follows:

Restrictions. These parameters are fixed before a design starts. Some of these restrictions are mandated by the technology, while others result from a desire to use specific modules and/or to simplify the design. For a mesh array, the main restriction is in its organization as an array of processing cells with nearest-neighbor communication, with I/O only at the boundaries.

Controllable parameters. These are parameters for which values are obtained by directing the design process.

Uncontrolled parameters. These are parameters which receive values during the design process, but for which the process does not provide any control.

Which parameters belong to each class depends on the particular system to be implemented and on the design method. Ideally, a design method should be able to handle any division; of particular importance are the limitations imposed on the set of restrictions and on the parameters that can be controlled. Flexibility

in defining this division is a good measure of the power of a design method. A *closed* method has a predefined assignment of parameters to classes, while an *open* method allows performing such an assignment as directed by the application.

The research performed in this dissertation centers on a design technique which makes explicit to a designer the existence and impact of the above-mentioned classes of parameters, as well as the performance and cost measures. Moreover, the design technique is open in the sense that it allows the designer to select specific restrictions, controllable parameters, and measures.

In this chapter, we have discussed important issues in the design and implementation of arrays for matrix algorithms. The need for methods that make these tasks easier has been stated. The objective of design methods and tools is to help designers in the process of deriving arrays for specific algorithms or mapping algorithms onto arrays. Consequently, such methods should make the design issues visible and explicit to the user, as discussed in the next chapter.

CHAPTER 3

Methods for the design of arrays

Issues involved in designing arrays for matrix computations were discussed in the previous chapter. We center our attention now on methods that allow carrying out design tasks in a systematic manner. A classification of design methods due to Fortes (et al.) [Fort88] is reviewed first, and the limitations of this classification for the purposes of comparing methods are discussed. As a way to overcome those limitations, a framework is proposed which identifies two stages in the application of a design technique: *regularization of algorithms* and *derivation of arrays*. Criteria to evaluate methods under this framework are presented, and some of the approaches proposed in the literature are reviewed. The criteria also constitute a set of guidelines for the development of a powerful design method. We highlight the convenience of using dependencies in an algorithm as the basis for a design technique, and discuss the properties of data-dependencies and index-dependencies for this purpose.

3.1 The classification of design methods by Fortes (et al.)

Different techniques have been proposed for the design of arrays. The most successful approach for that objective has been a *transformational* paradigm, where the description of an algorithm is successively transformed into a form suitable for implementation [Fort88, Most84]. This approach may also be regarded as having an algebra, with suitable symbols and operators, where an algorithm is represented by expressions; a design method consists in manipulating the expressions with the objective of transforming them into a form that describes an architecture and the mapping onto such an architecture.

According to Fortes et al. [Fort88], transformational systems are characterized by how algorithms are described, what formal models are used, how systolic structures are specified, and what types of transformations are used on and between these representations. Consequently, Fortes et al. visualize transformational systems as three-dimensional spaces (i.e., *Y-charts*), as depicted in Figure 3.1, where

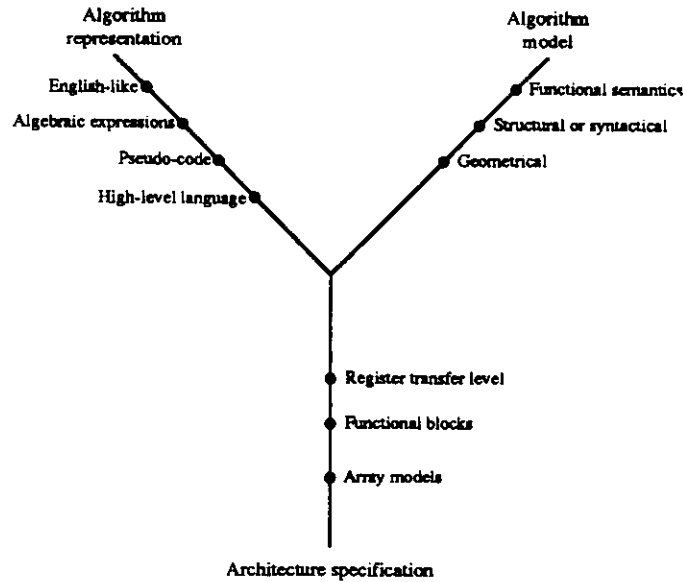


Figure 3.1: Y-charts to describe transformational systems [Fort88]

the dimensions (or axes) have the following meanings:

The algorithm representation axis, which indicates the different forms or levels to present an algorithm to the transformational system. Examples of points along this axis include representations in natural language, algebraic expressions, pseudo-code, and high-level language.

The algorithm model axis, which shows different levels of abstraction used to represent relevant features of the algorithm. Examples of points along this axis include algebraic model, computational graph, space-time representation and dependency graph.

The architecture specification, which is associated with the hardware model or level of design in which the systolic array is described. Points along this axis include processing element functions and interconnections, functional blocks, and register transfer level (RTL).

Directed arcs between points along and across the three axes of a Y-chart are used to illustrate transformations that map a given representation into another representation in the same or a different axis. From these Y-charts, Fortes et al. group the various methods that they reviewed into the following classes [Fort88]:

1. Methods that allow transformations to be performed at the algorithm-representation level and that advocate a direct mapping from this level to the architecture specification.
2. Methods that prescribe transformations at the algorithm-model level, requiring procedures for deriving the model from the algorithm representation and for mapping the model onto hardware.
3. Methods that transform a previously designed architecture into a new architecture.
4. Methods that abstract the function implemented by a given systolic architecture and use symbolic manipulations and transformations to prove the correctness of the design.

Fortes et al. associated each of the methods they reviewed with one of these four classes, and provided the corresponding Y-charts illustrating the properties of those methods. As a result, Fortes et al. identified fourteen techniques that perform transformations at the algorithm-model level. The popularity of transformations at such a level is basically due to their generality and applicability. The discussion in the remainder of this chapter is centered around this type of method. Moreover, the method proposed in this dissertation belongs to this class.

The classification devised by Fortes et al. has facilitated an understanding of the characteristics of the transformational systems. However, such a classification does not aid in comparing the capabilities of different methods or in selecting the adequate one for a given task. The concepts attached to the Y-chart axes are specific to each method, so that a comparison or evaluation of properties in terms of specific measures is not feasible.

To illustrate the limitations of this classification, let us consider two methods that use a similar tool at the algorithm-model level, namely dependencies. There are several techniques based on dependencies, such as those described in [Rao85, Mold83, Quin84, Kung88c, Capp84, Yaac88b]. Although these approaches have similar underlying principles to carry out their task, the corresponding Y-charts are not helpful for comparing and evaluating these techniques. For example, Figure 3.2 depicts the Y-charts for Quinton's method [Quin84] and for Moldovan and Fortes' technique [Mold83], as presented in [Fort88]. From this figure one can understand the type of transformations available in each approach, but it is not possible to decide which method is more suitable for a given task or how the two methods compare.

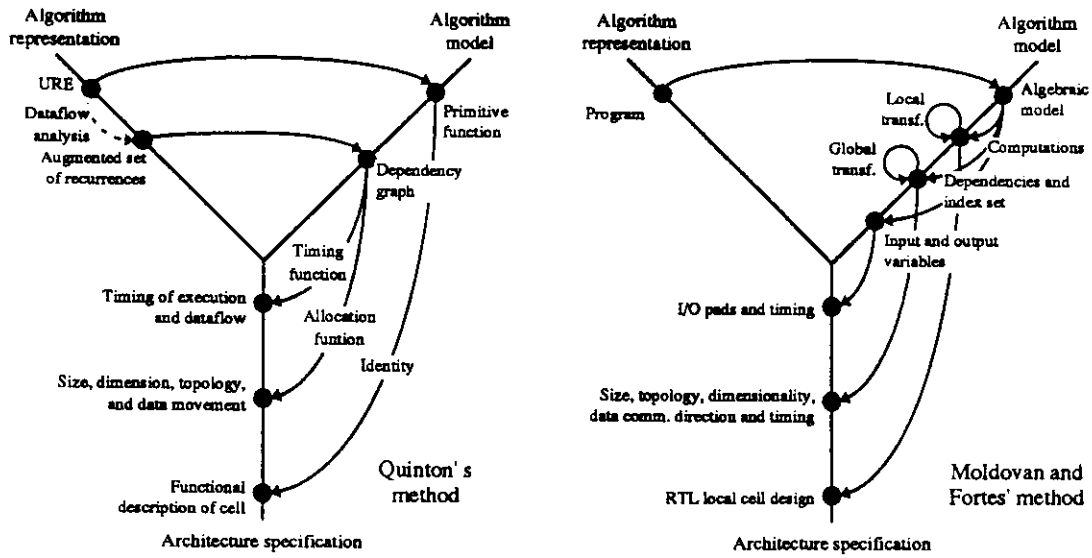


Figure 3.2: Y-charts for two dependency-based methods [Fort88]

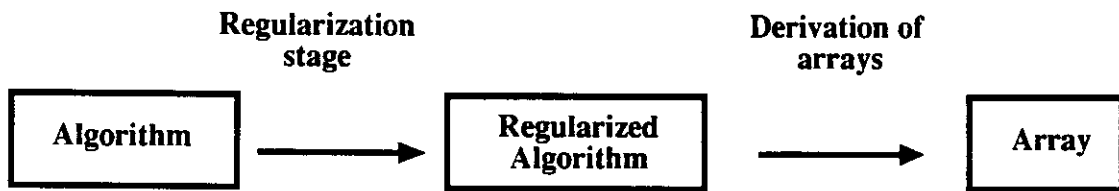


Figure 3.3: The stages in a design method

Due to the limitations described above in the classification of Fortes et al., in the next section we present a framework suitable for analyzing and comparing different design techniques.

3.2 A framework to compare design methods

The objective of a design method is to support the design process. We identify two stages in the application of a design technique, as shown in Figure 3.3:

Regularization, that is, the derivation of a canonical (regular) representation of an algorithm from an admissible form. The regularized form must provide an implicit or explicit description of parallelism, in a manner suitable for implementation in arrays. Moreover, the regularized representation must be

in a form suitable for manipulation in the remaining steps of the method.

Derivation of arrays using the regular description obtained above.

As will be discussed later, most methods proposed in the literature assume a regularized description of an algorithm but do not provide mechanisms to derive such a form. In other words, proposed methods have addressed only the second stage above, and have largely ignored the first one. Such a separation has allowed a concentration of effort on the derivation of arrays, but has neglected the impact that the regularized form has on implementations.

Analyzing methods in terms of the two stages given above, together with requirements and characteristics associated with each stage, allows a more precise comparison of the capabilities of different methods than that available with the Y-charts of Fortes et al. Within this framework, the suitability of a design method is evaluated in terms of the following criteria:

Regularization:

- The class of algorithms to which the method may be applied, that is, the degree of generality of the admissible form of algorithms.
- The set of transformations used to devise a regular description, suitable for derivation of arrays, from the admissible form of algorithms.
- The effectiveness of the regular description in conveying the properties of an algorithm suitable for implementation in arrays.

Derivation of arrays:

The capabilities of the method for:

- Performing transformations.
- Incorporating implementation constraints and restrictions in the design, such as limited local storage and limited bandwidth.
- Incorporating different attributes in the design of processing elements, such as pipelining, non-conventional arithmetic, and specialized functional units.

- Performing optimization of specific performance measures as part of the design process, which requires performing tradeoffs among implementation parameters and evaluating them (i.e., moving around the space of solutions).
- Designing arrays for fixed-size data and partitioned problems.
- Realization and mapping, that is, designing algorithm-specific arrays and mapping algorithms onto class-specific arrays.

An additional factor, applicable to both stages, is the *ease of use and suitability for automation*.

Evaluating methods in terms of the factors above leads to the conclusion that a design technique must have *strong descriptive capabilities for ALGORITHM, ARCHITECTURE and TRANSFORMATIONS*. This is particularly true when one considers that a method should be able to cope with algorithms and architectures that may have very different characteristics. The need for such a descriptive power has not been explicitly identified before.

In the next section, we review some of the design methods proposed in the literature. We concentrate on the descriptive capabilities of these techniques, and identify the two stages in them (whenever possible).

3.3 A review of other design methods

In spite of all the research performed, realizing algorithms as arrays remains an open problem. Fortes et al. [Fort88] concluded that “from a global point of view, it is clearly indicated that the two greatest limitations in the state of the art of existing transformational systems are the non-existence of powerful systematic semantic transformations and the inability to systematically achieve optimality in the resulting designs.”

Significant and basic differences among design methods are found in the way that a regularized algorithm description is represented, and in the capabilities associated to that description. That is, methods differ in the *tool or language* used to describe a regularized algorithm, and in the suitability of such a tool or language to perform transformations to derive an array. Representations used for these purposes include algebraic expressions, graphical descriptions and high-level languages.

In this section, we briefly review how some of the methods proposed in the literature describe a regularized algorithm, how such a representation is derived, and how it is transformed into an architecture. A more extensive comparison of two specific methods with the one proposed in this dissertation is given in Chapter 7. It should be noted that most of the methods reviewed in this chapter have been devised for realizing only algorithm-specific arrays for problems with fixed-size data. Further discussion on the capabilities of these methods can be found in [Fort88], which also contains an extensive bibliography.

3.3.1 Algebraic descriptions

Several of the methods proposed use an algebraic approach. In such a context, the regularized description is given as a set of algebraic expressions to which transformations are applied. Examples are vector operators [Gann82], a canonical algebraic representation [Kung83a], recurrence equations [Mira84, Li84, Capp83, Quin84, Delo86, Guer86], regular iterative expressions [Rao88], indices of nested loops [Mold83, Fort85, Lee88], dependencies and identity [Ko88], space-time transformations [Capp84], and affine recurrent equations [Yaac88b, Rajo86].

For example, Kung and Lin [Kung83a] proposed an algebra for systolic computation. Algebraic transformations are applied on the algebraic model of an algorithm to obtain an algebraic representation of a systolic design. According to Fortes et al., “such a canonical algebraic representation consists of the two matrix expressions $v \leftarrow Av + bx$ and $y = c^T v$, where x represents the input, y represents the output, and v represents variables generated by implicit functions. A , b and c represent delay cycles between the availability and use of the variables.” In other words, the regularized version of an algorithm is given in terms of its canonical algebraic representation. However, there is no indication of how such a description is obtained for a particular algorithm. Algebraic transformations applied to the canonical representation consist of retiming and “ k -slowing.” This algebra is used to derive designs where broadcasting is replaced by distributing common data to different destinations at different times. The technique has been applied to FIR and IIR filters, matrix multiplication, and has been used to derive two-level pipelined arrays for LU-decomposition [Fort88].

Li and Wah [Li84] describe algorithms algebraically with three classes of parameters: velocities of data flows, spatial distributions of data, and periods of computation. Relationships among these parameters are represented as constraint vector equations that must be satisfied. Consequently, the regularized description

in this technique consists of parameters and constraint vectors. As in the case of Kung and Lin, this method does not include tools that allow obtaining the regularized form. Li and Wah formulate the design as an optimization problem, where the search space is polynomial on the problem size. They express completion time and hardware complexity in terms of the three classes of parameters above, and the optimization process consists of minimizing completion time T or area-time AT^2 . As a result, the method is suitable for considering only the issues mentioned; it does not have the facility to incorporate other implementation constraints. Results obtained include systolic arrays for FIR filters, matrix multiplication, discrete Fourier transform, polynomial multiplication, deconvolution, triangular matrix inversion, and tuple comparison [Fort88].

Weiser and Davis [Weis81] proposed a method that describes an algorithm as sets of data that are treated as wavefront entities, with transformations applied to the wavefronts. The regularized description consists of such wavefront entities, but no tool is included in the method to obtain them. The derivation of arrays is presented as operations on sets of data, using a *KM-function* on such data. Examples of applications of this method are string matching and band matrix multiplication.

Quinton [Quin84] uses a set of uniform recurrent equations as a description of the algorithm. As in the previous examples, the technique does not include facilities to derive this regularized algorithm. However, recent research has addressed this issue [Dong88b], although the effectiveness of those results is not yet clear. Regarding the derivation of arrays, the method first finds a timing function compatible with dependencies of the equations and then maps the equations into a finite set of coordinates, each representing a processing element of a systolic array. Architectures derived with this method include arrays for convolution and matrix product. Extensions of this technique allow the derivation of arrays for LU-decomposition and dynamic programming, as indicated in [Fort88].

Cappello and Steiglitz [Capp84] also use a set of uniform recurrence equations to describe the algorithm. Such a representation is expanded into a canonical form by the addition of an index representing time. Tuples of indices' values are associated to positions in a multi-dimensional space, where each index corresponds to one dimension. Points in such a space correspond to primitive computations. Arrays are obtained as the result of projections from the multi-dimensional space, which determine the topology and size of the architecture as well as timing and direction of data flows. Architectures for matrix-vector multiplication, convolution, matrix-matrix product and matrix transposition have been formally derived with

this technique, as indicated in [Fort88].

Recent work by Cappello et al. includes a more general form of algorithms, namely systems of affine recurrence equations [Yaac88a, Yaac88b]. The same type of algorithm description is also used in [Rajo86, Rajo88a].

Moldovan and Fortes [Mold83] derive an algebraic model of an algorithm by using techniques similar to those used in software compilers. This regularized form describes the algorithm as structured sets of indexed computations that operate on a set of inputs to obtain a set of outputs. Such a form is modified by local and global transformations. Local transformations affect the functional and structural specification of cells, while global transformations restructure the algorithm. Moldovan and Fortes have devised a software tool that aids in finding suitable transformations for deriving arrays. With such a tool the user still has to select, from among those valid transformations, the one that best fits specific needs; this selection was being done manually [Mold87]. Results derived with this method include matrix-matrix multiplication, LU-decomposition, dynamic programming, and partial differential equations [Fort88].

Rao and Kailath have proposed a method for a class of computations they refer to as Regular Iterative Algorithms (RIAs) [Rao85, Rao88]. They have proved that “a systolic array executes an RIA that has a uniform affine schedule and conversely, every RIA with a uniform affine schedule can be implemented on a systolic array,” given their definition of systolic array. Consequently, a regularized description consists of such an RIA, which is represented by a Reduced Dependence Graph (RDG) and a specification of the index space. This method, and later related research [Royc88a], have given attention to the issue of deriving a regularized description, but so far the results are heuristic in nature and not fully satisfactory. Regarding the derivation of an implementation, the method consists of projecting the index space onto a processor space and scheduling the computations mapped onto each cell. Projection and scheduling are obtained through algebraic operations on the dependency vectors that appear in the RDG, including solutions of linear programming problems. Examples of application of this method are matrix multiplication, two-dimensional filtering, sorting, Gaussian elimination, and transitive closure [Rao85]. This method has also been used for the Faddeev algorithm [Jain87], Kalman filtering [Linc88], algorithms with pivoting [Royc88a], and fault-tolerant digital filtering [Lev-88]. Moreover, compilation of RIAs described as high-level language programs is reported in [Atha88].

The work by Rao and Kailath provides a unifying framework for many of the

algebraic-based approaches. These techniques are derived from work by Karp, Miller and Winograd [Karp67]. In this context, Rao and Kailath state that “Moldovan [Mold82] essentially presupposes that the RIA on hand has a strongly separating hyperplane” [Rao86] with some added restrictions, so that “there exists a multiplicative transformation of the index space.” Moreover, Rao and Kailath state that “Quinton [Quin84] considers a very special sub-class of RIAs, referred to as uniform recurrent equations. Uniform recurrent equations are RIAs in which all variables, except for one, are propagating variables.” In other words, according to Rao and Kailath, their regularized description is more general than that used by other researchers and consequently its applicability is broader.

3.3.2 Descriptions using high-level languages

General-purpose high-level languages have also been used as the description tool for design methods. For example, Lam and Mostow [Lam85] use this approach to model the design process as a series of transformations on a high-level language description. They rely on human designers to decide which transformation to apply, instead of aiming towards a fully automated approach. A computer-aided transformational tool is used to assist designers. The process first performs software transformations on the description of the algorithm to prepare it for systolic implementation. This initial transformation converts the algorithm into a representation composed of highly repetitive computations, expressed in terms of nested loops and begin-end blocks. Consequently, the regularized description corresponds to this representation, and the method has steps to derive it. Such a regularization stage includes annotating the algorithm description with statements to indicate how subfunctions should be evaluated, such as “in parallel” or “in place” (i.e., sequentially within the same unit). The automated software tool is capable of mapping these statements onto hardware. Subsequently, a sequence of hardware allocation, scheduling and optimization phases are applied iteratively. The optimization phase is guided by the user, who selects the transformations to be applied [Lam85]. Results reported include previously known systolic arrays for polynomial evaluation, and a system for computing the greatest common divisor of two polynomials, as indicated in [Fort88].

Chen [Chen86] uses a general-purpose parallel programming language. Transformations are applied to algorithms described in such a language to remove broadcasting and limit the number of fan-ins and fan-outs. The output from this step can be regarded as the regularized algorithm. Another phase of transformations

incorporates pipelining to the algorithm and attempts to fully utilize hardware resources. These transformations are algebraic manipulations on expressions in the parallel programming language, so that the language must be amenable to algebraic modifications. Consequently, the approach is highly algebraic and has the same capabilities of other schemes based on algebraic expressions. Furthermore, it only incorporates the implementation issues indicated above, namely broadcasting and fan-in/fan-out. This technique was applied to the dynamic programming problem.

Chapman et al. [Chap85] use the OCCAM programming language for algorithm description, simulation and eventually implementation. Although the proposed approach yields programs which could be used on an array of Transputers, the objective of their work is the utilization of the OCCAM algebra to aid in the design process. Chapman et al. claim that an OCCAM program may be interpreted as an algebraic description of a regular array architecture that implements a given algorithm. The OCCAM program is transformed and, as long as the algebraic rules are adhered to, the designer may assume that the program will implement the same algorithm. However, there is no systematic way to perform those modifications, nor any mechanism to allow only valid transformations. Moreover, this technique does not define a regularized algorithm form.

3.3.3 Graph-based descriptions

A different line of research uses graphical notations to describe an algorithm. Examples include S. Y. Kung's Signal Flow Graph method [Kung87a], Schwartz and Barnwell's method [Schw84], the dataflow approach proposed by Ramakrishnan et al. [Rama83], and Koren and Silverman's technique [Kore83], among others.

S. Y. Kung's technique [Kung87a, Kung88c] starts by identifying a suitable algorithm and representing it with a dependency graph. Some transformations are applied on this graph to render it more suitable for later design steps, so that the resulting graph corresponds to the regularized version of the algorithm. However, the generation of the graph and the subsequent application of transformations are performed in ad-hoc manner, though some guidelines have been suggested. In any case, the process is not systematic, and it is not clear how to determine which transformations to use. The graph obtained is mapped (i.e., projected) onto a Signal Flow Graph (SFG), and there may be several SFGs, depending on the direction of projection. The method does not provide guidelines in selecting a specific SFG. Finally, the SFG is realized in terms of an array. Since the

choice of dependency graph, direction of projection and schedule greatly affects performance, Kung identifies two classes of mapping: canonical mapping for homogeneous (i.e., shift-invariant) dependency graphs, and generalized mapping for heterogeneous dependency graphs. The method allows removing data broadcasting by using *transmittent* data instead, but no other implementation restrictions, such as limited storage or bandwidth per cell, may be incorporated. Several algorithms have been studied with this method, including sorting, convolution, AR filter, matrix multiplication (dense and band), LU-decomposition, QR-decomposition, Gauss elimination, and transitive closure.

Ramakrishnan et al. [Rama83] proposed a formal model for a linear array of processing elements, as well as graph representations of programs suitable for execution on such a model. These graphs are defined as homogeneous graphs, which are a more limited class of program graphs than general data-flow graphs. In particular, homogeneous graphs have the same number of edges into and out of every node, excepting those nodes representing sources or sinks of data (i.e., inputs or outputs, respectively). This method may only be used to generate linear arrays, and it has been used for band matrix-vector multiplication, convolution, dynamic programming and transitive closure, as indicated in [Fort88].

Barnwell and Schwartz [Barn83, Schw84] have proposed another graphical approach. Their method starts with an algorithm described as a fully-specified flow graph, that is, a directed graph in which nodes represent operations and edges represent signal paths. Nodes are also used to represent delays explicitly, when those delays are part of the algorithm (e.g., digital filters). As a result of targeting the method to implementations in multiprocessors, this approach is suitable for arrays with identical cells. Barnwell and Schwartz claim that systolic arrays are characterized by synchronous data transfers, so that flow-graphs are constrained to have every output from a cell terminated by a delay node (or pipeline register). Hence, "the generation of systolic solutions for flow-graphs reduces to the distribution of delays nodes throughout the flow-graph." Their method consists of ad-hoc manipulations of the flow-graphs into systolic forms, using theorems from graph theory. Results obtained include previously derived and new architectures for FIR, IIR filters, and Markel-Gray lattice filters [Fort88].

Jover and Kailath [Jove84] proposed a pseudo-graphical approach. They introduced the concept of *lines of computation (LOCs)*, which are useful for determining whether a given topology is suitable for systolic computations. *LOCs* are a summary of an architecture with respect to time and space, and some properties of the architecture may be inferred from such *LOCs*. Jover and Kailath's work includes

the definition of *systolic-type arrays*, a generalization of systolic arrays which allows different cells not only at the boundary of the array but also inside. Reported results are three designs for matrix multiplication [Fort88].

3.3.4 Discussion of other methods

The design techniques reviewed in the previous subsections exhibit many important limitations in the following three aspects, which are discussed in more detail below:

- obtaining the regularized representation of an algorithm
- derivation of arrays (incorporating implementation restrictions and selecting suitable transformations).
- simplicity

Most methods do not provide tools to obtain the corresponding regularized representation of an algorithm, but rather assume that this representation is already available. For some simple algorithms, such as matrix multiplication, obtaining the regularized version (i.e., a uniform recurrence equation, a regular iterative algorithm or a dependency graph) is straightforward, so that the lack of a systematic procedure is not an issue. However, simple algorithms are relatively few; most matrix algorithms of importance (i.e., LU-decomposition, QR-decomposition, transitive closure, Gaussian elimination, Faddeev algorithm, among others) are not easily described in those regularized forms. Moreover, deriving the regularized form often adds complexity to the algorithm (in terms of additional operations that did not exist in the original version). Specific examples of these problems are described in detail in Chapter 7, where we compare Rao's method and S.Y. Kung's approach with the one developed in this dissertation.

Regarding the derivation of arrays, methods reviewed in this section are oriented towards the design of systolic arrays, that is, arrays of cells with no local storage. Moreover, some of these methods assume all identical cells in an array. Consequently, these methods have predefined the characteristics of cells and therefore are unable to incorporate other implementation constraints or restrictions in the design, such as type of cells, I/O bandwidth, and number of cells. In addition, these methods are unable to analyze tradeoffs among implementation parameters such as local storage and communication bandwidth of processing elements, and in many cases produce arrays with suboptimal cost and/or performance.

The last important aspect of transformational systems is simplicity in using the methods and guidance in selecting suitable transformations. Several of the techniques reviewed hide important properties of algorithms and implementations, in many cases leading to inadequate conclusions regarding features of an algorithm and their suitability for a particular array. That is, the tools used for these methods are not adequate to convey the characteristics of an algorithm that are relevant for an implementation. A significant example is the use of algebraic expressions that allow applying powerful transformations to an algorithm, and as a result have been considered attractive alternatives to automate the design of arrays. However, the complexity of such an approach obscures the process. In his pioneering work on systolic arrays, H.T. Kung [Kung79] concluded that “LU-decomposition, transitive closure and matrix multiplication are all defined by recurrences of the same ‘type.’ Thus, it is not coincidental that they are solved by similar algorithms using hexagonal arrays.” A similar statement was made by Moldovan in [Mold83], in the description of his algebraic design approach. However, it has been shown that the algorithms for these computations have quite different dependency structures, so that they are mapped efficiently only onto different arrays [More87, More88c].

3.3.5 Methods for partitioning

As stated earlier, most methods reviewed in this chapter are suitable only for the design of algorithm-specific arrays for fixed-size problems, and do not consider partitioning. Only a few methods use the partitioning techniques described in Section 2.6 in a systematic manner. Among those that consider them are the SFG method [Kung88c] (pp.374-382), Moldovan and Fortes’s technique [Mold86], and the approach by Navarro et. al [Nava87, Nava86b, Nava86a]. Excepting the technique by Navarro et al. the others are extensions to methods for the design of arrays for fixed-size problems; as discussed below, these extensions exhibit limitations.

Partitioning in S.Y. Kung’s method uses either coalescing or cut-and-pile (LSGP and LPGS respectively, in his notation), and requires separate steps for spatial mapping and scheduling. Such an approach first derives a large array, and then maps it onto a small array (i.e., it follows an indirect partitioning strategy). However, as we stated in Section 2.6, an indirect design of an array for partitioned execution is less convenient than direct partitioning.

Moldovan and Fortes’ partitioning technique is an extension of their method for the design of arrays for fixed-size data. As stated earlier, Moldovan and Fortes

use an algebraic approach. This method also realizes the algorithm as a large (i.e., virtual) array, which is later mapped onto a small array by partitioning it into bands that are processed successively (i.e., cut-and-pile). Consequently, as in Kung's scheme, this method cannot benefit from the properties of an algorithm which are suitable for direct partitioning.

Navarro et al. transform an algorithm with large-size dense matrices into an algorithm with band matrices and compute the resulting algorithm in an array that matches the size of the band. The original algorithm is decomposed into subalgorithms that are chained for execution in the target array. Although the approach is interesting, the decomposition, if possible at all, is dependent on the algorithm; as a result, this technique is not sufficiently general.

3.3.6 Mapping methods for class-specific arrays

A number of class-specific arrays have been recently proposed or built. Among these are Warp [Anna87], Esprit [Groo87], SLAPP [Drak87] and VATA [Syma88]. These arrays are intended to solve any of a class of algorithms. Out of these four architectures, only Warp has reported a systematic mechanism to compile algorithms for execution on it. According to the results published, SLAPP and Esprit have used ad-hoc mapping approaches. VATA is still under development, so it is not yet known what mapping approach it will use.

Warp [Kung88b] is a linear array of processing elements, able to implement different models of computation. Among them, a technique referred to as the *domain parallel model of computation* has been described in [Tsen88]. Mapping algorithms onto Warp according to that model of computation requires the user to specify how elements of a matrix are evenly allocated to cells (i.e., by rows or columns). The remaining mapping steps are automatically performed by a compiler. This approach does partitioning by coalescing, because the entire matrix is mapped onto the array at once. Moreover, Warp cells have large memory, which allows implementing such a partitioning technique. Consequently, all data elements and operations associated to one row (or column) of a matrix are allocated to the same processing element. The approach has the drawback that, for a large majority of matrix computations, the number of operations associated with a row (or column) of a matrix is not constant. As a result, the computational load is not evenly distributed throughout the array, leading to under-utilization of cells. Results reported in [Tsen88] indicate that, on the average, only about 33% utilization is achieved (i.e., 33 [Mflops] are delivered out of 100 [Mflops] peak capacity).

3.4 Computer-aided design tools

For some of the methods proposed in the literature, the issue of devising computer-aided tools that support those methods has been addressed. Among the tools built for this purpose are VACS [Kung88d, Kung89], PRESAGE [Dong88a], SYSTARS [Omtz88], DIASTOL [Fris86], ADVIS [Mold87], SDEF [Engs87], and HIFI [Anne88]. Poker [Snyd84], a more general tool than those cited above, has also been considered suitable for programming systolic arrays. We briefly review some of these tools in the paragraphs below.

SYSTARS

SYSTARS [Omtz88], developed at Delft University, employs a graphical approach for the analysis and synthesis of arrays for regular iterative algorithms (RIA, [Rao88]). SYSTARS displays the dependency graph of an RIA as a three-dimensional picture that can be rotated and zoomed-in by the user. Moreover, it allows investigating the properties of space-time mappings for fixed-size data and partitioned problems.

A SYSTARS design session starts by specifying a regular iterative algorithm in the form of a Reduced Dependency Graph (RDG, [Karp67]). SYSTARS uses the RDG to construct a multi-dimensional dependency graph, with one dimension per index. Nodes in this graph specify functional relationships between input and output variables, while directed edges between nodes represent the variables. A space-time mapping of the dependency graph is obtained by applying a linear transformation on the index space and the data dependencies of the RIA. Such space-time transformations lead to arrays for fixed-sized data. An additional design task is to partition such an array for large problems to fit on smaller size arrays. This partitioning is accomplished using a coalescing technique with an adaptive clustering algorithm.

The applicability of SYSTARS is limited by several factors. First, SYSTARS can only be used on RIAs that have three indices, so that they can be represented by three-dimensional dependency graphs. Moreover, the selection of the scheduling vector, which requires solving an integer programming problem, has not yet been implemented. As a result, this vector must be given by the user, although the system checks that the vector is a valid one. In addition, the partitioning approach followed by SYSTARS is *indirect*, because it first derives an array for large-size data

and then partitions that array to fit on a smaller one.

DIASTOL

DIASTOL [Fris86], developed in France, is based on the design method proposed by Quinton [Quin84]. This tool combines the design of the topological properties of an array with the design of the cells composing such an array. The design process starts from the equations of the algorithm, which are transformed within the system until they become *uniform recurrent equations* (UREs). The system then helps the designer to devise arrays by using a *dependency mapping* procedure that results in an abstract description of the design. This abstract specification is used as the starting point for the functional design of cells, including pipelined, skewed or bit-serial hardware operators. Timing and allocation functions allow the automatic computation of delays introduced by operators and the timing of the data (at the bit level).

The main limitations of this tool result from its suitability only for those algorithms that can be expressed as UREs, whose characteristics were discussed in Section 3.3.1, and the lack of support to perform partitioning of algorithms.

PRESAGE

PRESAGE [Dong88a] was developed at Philips Research Laboratory in Belgium. This is a tool for the development of systolic circuits also based on the method for uniform recurrence equations proposed by Quinton [Quin84]. Given a set of uniform recurrences, PRESAGE finds the linear timing and linear allocation functions that minimize the number of time steps and the number of processing elements in a system. Depending on the specification, pure or semi-systolic circuit designs are generated. Specifications on the connections are used to derive circuits with unidirectional data flow. When two directions of pipelining are allowed, PRESAGE finds the one that minimizes the cost of the system. The heart of the tool is an algorithm that solves some specific integer programming problems.

Similar to DIASTOL, the main limitation of PRESAGE is its suitability only for uniform recurrent equations.

ADVIS

ADVIS [Mold87], developed at University of Southern California, is a program that helps to transform a sequential algorithm into a parallel form suitable for implementation as an array. It considers both fixed-size data and partitioned problems.

ADVIS is based on the method proposed by Moldovan and Fortes [Mold83, Mold86]. Such a method uses an algorithm described with nested loops as the starting point. Dependencies among the variables composing the body of these loops are described as *difference vectors* of index points. These vectors, used as an original representation of the algorithm, are transformed into another representation where one index indicates time and the remaining indices are used to meet the space restrictions of an implementation. ADVIS aids in the complex task of selecting this transformation. The tool finds many valid transformations, from which a designer chooses the one that best fits the needs of an implementation. This selection is used in the remaining steps of the method.

The main limitations of ADVIS result from the underlying mapping method, as discussed in Section 3.3.1.

SDEF

SDEF [Engs87] is a programming environment for describing systolic algorithms developed at the University of California in Santa Barbara. It includes a notation for expressing the algorithms, a translator for the notation, and a systolic array simulator. SDEF accepts a special class of computations referred to as *atomic systolic computations*. Such computations are described by a set of four properties (S,D,E and F). Programming in SDEF is based on space-time representations of systolic computations [Capp84]. This tool is oriented towards programmable arrays, so that its capabilities are similar to those of Poker (described later). The main difference is that SDEF uses a higher level notation for programming systolic arrays instead of the message-based programming available in Poker. However, there is no support for the mapping process. As indicated in [Engs87], SDEF can be used to express the results of the analysis, synthesis, and optimizations performed by other tools.

VACS

VACS [Kung88d, Kung89] is a tool that implements the design method proposed by S.Y. Kung [Kung87a, Kung88c]. This tool accepts the description of an algorithm in terms of a dependency graph, and generates array structures for such an algorithm. The system is interactive and graphics-based. It allows the evaluation of several optimality criteria and the verification of design correctness by simulation. As a result, a designer can see the dependency graph displayed on a screen, can modify and simulate the graph, and even evaluate the numeric performance of an array with a finite word length.

The main limitations of VACS arise from the underlying method, as discussed in Section 3.3.3.

The Poker programming environment

Poker [Snyd84] is a general programming environment for parallel computers without shared memory. Although not specifically designed for systolic arrays, it has been described as a “system exhibiting many of the characteristics that one would expect of a programming environment specially built for systolic computation” [Snyd86]. A program written using Poker consists of five components that describe the communication structure of the problem, the processes running on the different cells, and I/O to cells and to array. Using an interactive interface, a user specifies each of these components, which are stored in a database, and accesses them through a set of views. Moreover, Poker provides debugging facilities that allow a variety of ways to control the running of a program, such as single-step and checkpoints.

As stated in [Snyd86], Poker exhibits many characteristics desirable in a programming environment. However, Poker has no support for the process of mapping an algorithm onto an array. It simply allows programming the operations that are executed in a cell, once it has been determined what those operations are.

3.5 Algorithm dependencies in methods for the design of arrays

As described in Section 3.3, transformational methods to map matrix algorithms onto arrays have used various principles to achieve their purposes: some use dependencies in the algorithm, some rely on parameters such as velocities of

data flow, data distributions and periods of computations, some use an algebra for systolic array design.

Using dependencies as a vehicle to perform transformations seems more advantageous than other approaches, because these dependencies describe the fine-grain parallelism available and dictate the communication requirements. Dependencies have been applied successfully to formulate and implement compiler optimizations [Kuck81], and to drive the dataflow model of computation [Ager82].

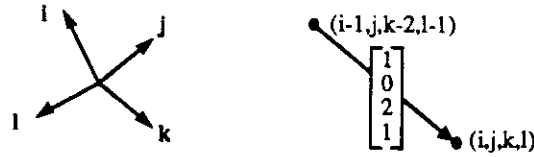
An attractive property of algorithm dependencies in the design of arrays is their suitability for obtaining a regularized description of an algorithm, and for deriving arrays based on such a description, paying attention to the target architecture (i.e., incorporating constraints/restrictions arising from the implementation). However, the effective use of dependencies for these purposes depends heavily on how those dependencies are expressed and manipulated, as we discuss below.

There are two ways to describe dependencies:

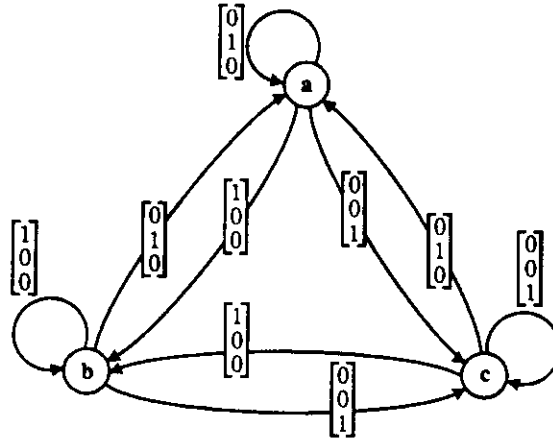
Index-dependencies are described by relations in the index space of an algorithm. That is, the computation of each instance of a variable is associated with a point in a multi-dimensional space defined by the indices of the algorithm; for such purposes, all variables must have the same number of indices. Dependencies among variables are related to the distance between those variables in the index space, and are represented as expressions with those indices. This type of dependencies is illustrated in Figure 3.4a. Examples are methods for regular iterative algorithms (RIAs) [Rao88], uniform recurrent equations (UREs) [Quin84], nested loop structures [Mold83], and affine recurrence equations [Yaac88b].

Index-dependencies are associated to specific types of algorithms whose structures are identical at every point in the index space. Consequently, these dependencies may be represented by a Reduced Dependency Graph (RDG). Nodes in such a graph correspond to variables, while edges represent the dependencies among variables. Edges are tagged with a dependency vector which corresponds to the difference among the indices of variables that appear in an expression. As an example, Figure 3.4b depicts the RDG for the transitive closure algorithm expressed as a RIA [Rao85].

Describing dependencies with index expressions has limited capabilities with respect to admissible algorithms. For example, UREs are algorithms expressed by a system of equations where one variable is computed as $x_i(z) =$



(a) Index-space and dependencies



(b) Reduced dependency graph for transitive closure

Figure 3.4: Index-dependencies

$f(x_1(z - d_1), x_2(z - d_2), \dots, x_p(z - d_p))$ (where z and d_j , $j = 1, \dots, p$ are vectors of indices) and all remaining $x_1(z), \dots, x_{i-1}, x_{i+1}, \dots, x_p(z)$ are just data transfers (i.e., $x_k(z) = x_k(z - d_k)$). RIAs, nested loop structures, and affine recurrence equations are more general than UREs because they allow more than one computed variable, but they still have important limitations on admissible algorithms, as will be discussed in more detail in Chapter 7.

Data-dependencies are described by a graph and are obtained by following the flow of data in an algorithm (such as a dataflow graph). Figure 3.5 illustrates this description of dependencies. Examples using this approach are the Signal Flow Graph method described in [Kung88c] and the method proposed in the following chapters of this dissertation.

The graph used to describe data-dependencies is an explicit graph (EDG), that is, a graph where there is one node to represent each operation and each edge corresponds to a data dependency among two operations. Figure 3.5b depicts the EDG for computing the transitive closure algorithm for a problem of size $n = 3$.

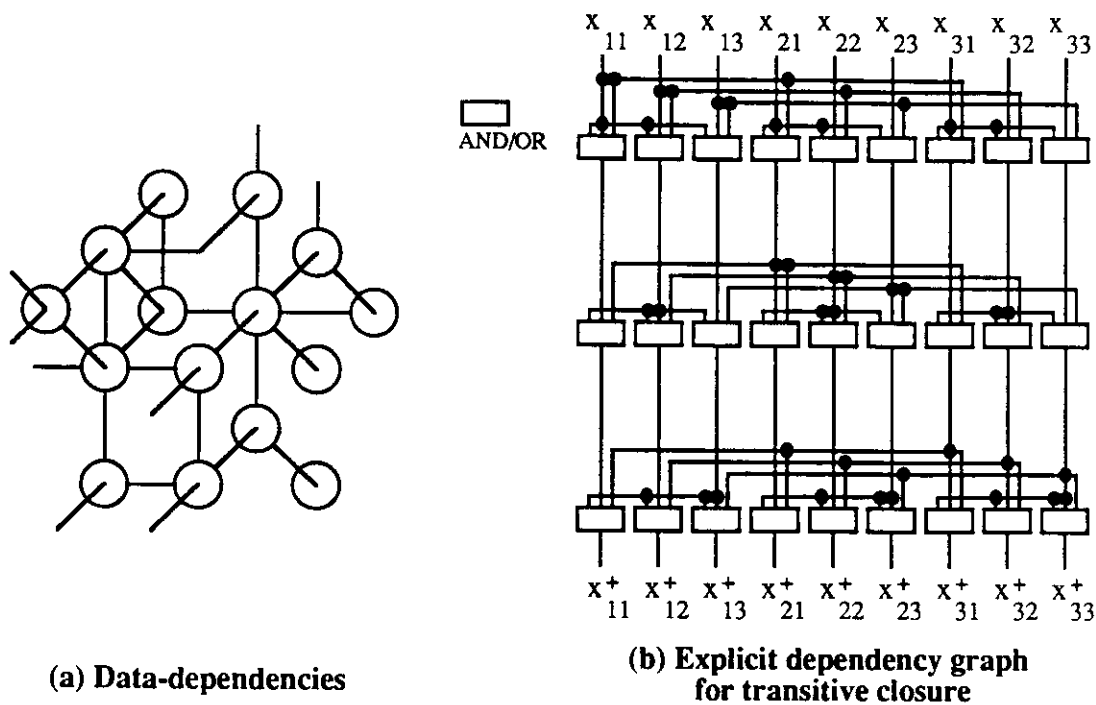


Figure 3.5: Data-dependencies

3.6 Conclusions regarding methods for the design of arrays

In this chapter, we have reviewed characteristics of methods for the design of arrays. It has been stated previously [Fort88] that the most convenient approach for this purpose is a transformational paradigm, in which an algorithm is successively transformed until reaching a form suitable for implementation. We have shown that such a transformational process consists of two stages: the first derives a regularized version of an algorithm, while the second uses the resulting form to derive arrays. It has been argued in this chapter that most existing methods provide insufficient tools, if any, to obtain the regularized representation. In fact, most methods assume that the desired representation is available, which is usually not the case for many matrix algorithms of interest. Moreover, it has been argued that the existing design techniques make many assumptions regarding the characteristics of cells and arrays, and that such techniques are unable to incorporate important implementation restrictions or perform tradeoffs among implementation parameters. Consequently, one may conclude that previously proposed design methods are not sufficient to accomplish the tasks required in the design of arrays.

We have argued that a successful method should use dependencies in algorithms as the basis for the transformational process. Moreover, performing these transformations requires strong capabilities which must cover both stages in a design, namely regularization and derivation of arrays. That is, desired capabilities in a method must comprise algorithm, transformations and architecture. These capabilities allow a design technique to take implementation restrictions into account, to perform tradeoffs among implementation parameters, and to preserve the dependencies in the algorithm, while paying attention to performance and cost measures.

Moreover, a successful design method should fulfill other requirements stated earlier, namely simplicity, generality, suitability for fixed-size and partitioned problems, and the realization and mapping of algorithms onto arrays. The combination of these factors leads to a design system that provides an integrated and unified framework for algorithm, architecture, and transformations. Such a framework has the potential of being a successful environment for realizing and mapping arrays for matrix computations, without the limitations that have characterized earlier methods.

CHAPTER 4

Description of the graph-based method

We describe now our data-dependency graph-based method for the design of application-specific arrays for matrix computations, and illustrate it using the triangularization algorithm by Givens' rotations. A formalization of this method is given in Chapter 5. This technique is oriented towards the execution of multiple-instance fixed-size data and partitioned matrix algorithms. Since application-specific implementations are normally devised for the successive execution of the same algorithm with different data sets, we believe that this orientation is proper.

4.1 Assumptions regarding matrix algorithms and arrays

The following subsections state the assumptions regarding the matrix algorithms, the operators and the arrays used in our method.

4.1.1 Matrix algorithms

As shown in Figure 4.1, matrix algorithms suitable for our method¹ are described recursively by an outermost loop and a loop-body that contains scalar, vector and matrix operators, and other matrix algorithms. A sequence of algorithms as those shown in Figure 4.1 is also a matrix algorithm. Operators in a matrix algorithm have the following characteristics:

Scalar (or primitive) operators are basic unary, binary or ternary operations whose computation time is data independent (such as add, multiply, rotation, sin). Consequently, a scalar operator may have up to three operands. Moreover, a scalar operator may produce up to two outputs. (In practice, scalar operators produce a single result, excepting cases such as rotation of a pair of elements which produces two outputs.)

¹A formal description of this canonical form of matrix algorithms is given in Chapter 5.

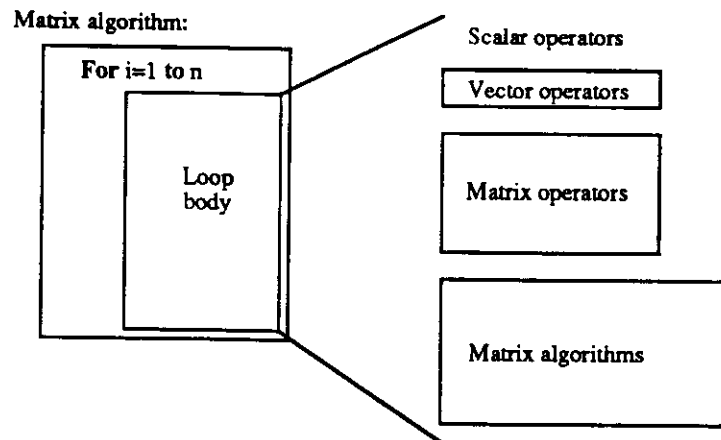


Figure 4.1: The canonical form of a matrix algorithm

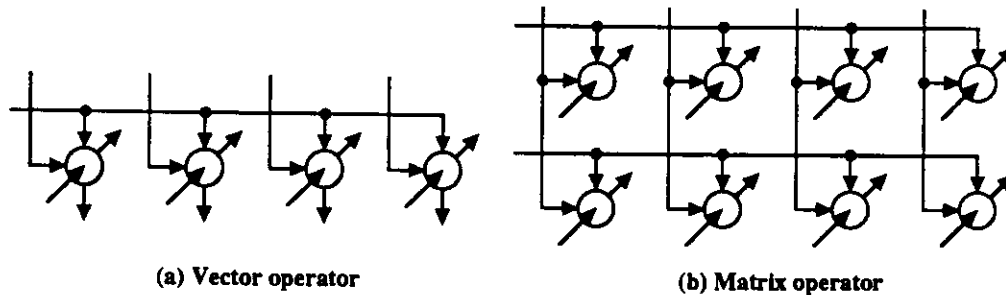


Figure 4.2: Dependency graphs of vector and matrix operators

Vector operators have up to two vector operands and produce up to two vector results. The same primitive operator is applied to each element of the vector operands to produce the vector results. An additional scalar operand may be common (i.e., broadcasted) to all the instances of the primitive operator. The dependency graph of a vector operator is shown in Figure 4.2a.

Matrix operators have one matrix operand, a vector operand common to rows of the matrix operand, and a second vector operand common to columns of the matrix operand. A matrix operator produces a matrix result. The same primitive operator is applied to each element of the matrix operand (and associated elements from the vector operands) to produce the matrix result. Figure 4.2b depicts the dependency graph of a matrix operator.

The form of a matrix algorithm above does not have any requirements on the way that variables are referenced, that is, on how loop indices are used to access elements of matrices and vectors. Two types of references are usually considered:

(1) *uniform expressions* and (2) *affine expressions*. Uniform expressions are of the form $(i - i_0)$ (i.e., an index plus/minus a constant), while affine expressions have the more general form $(i + j - k_0)$ (i.e., linear combination of indices and a constant). Uniform expressions are the more common type of references and appear in most matrix algorithms. Nevertheless, the method allows both types of references.²

In addition to input data, the output from one operator in a matrix algorithm may also be used as input for another operator. The limitations in number of inputs and outputs to/from the operators arises from the objective of realizing them in mesh arrays. Since these arrays have nearest-neighbor connections (i.e., no broadcasting) and external I/O only at the boundaries, they are suitable for applications where data elements flow through cells while being used for different operations. Consequently, realizing for example a matrix operator in a mesh array requires to eliminate broadcasting; this is achieved by transferring broadcasted data through the cells (i.e., transmittent data), so that only one independent input per cell is allowed (an element of the matrix operand, which is stored in the cell). Similar restrictions exist for vector operators. These aspects will be discussed in more detail as part of the method.

The properties of matrix and vector operators given above exclude operating on two matrices or on three vectors (i.e., adding two matrices or rotating elements of two vectors by corresponding angles contained in a third vector). These types of operators are not suitable for implementation in arrays, because they do not reuse input data. For our purposes, such cases correspond to sets of scalar operations.

From the discussion above, we observe that vector and matrix operators consist of primitive operations that are “tied” together by the common operand(s). Such operand(s) correspond to broadcasting data throughout the elements of the vector/matrix (in the case of operating on the elements of two vectors with no common operand, one may assume the existence of a “null” broadcasted value).

4.1.2 Cells and arrays

The following assumptions relate to characteristics of cells and arrays:

²Using uniform or affine expressions to access variables does not imply that the algorithm must be a uniform or an affine system of equations, as required by other methods. The form of admissible algorithms given here is more general than those restricted cases.

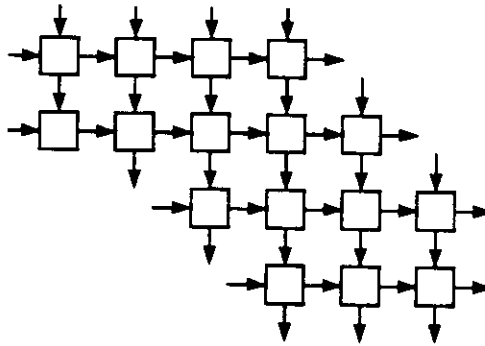


Figure 4.3: Mesh-connected array

- Arrays are either linear or two-dimensional mesh-connected structures, with external I/O from a host only at the boundaries of the array. Cells have two input and two output ports which are used to connect them in mesh structures. These arrays have only local communications (i.e., no capabilities for broadcasting or routing of data through cells without using it). Moreover, flow of data in these arrays is unidirectional, without data counterflow. Figure 4.3 depicts an array according to these characteristics.
- Cells are either systolic, pseudo-systolic, or local-access, as presented in Chapter 2.
- The model of computation consists of synchronized flow of data through cells, with operations performed in each cell. All operations have the same computation time. If cells are pipelined, the stage time is the same for all operations. Consequently, data flow rate and computation rate are determined by a basic *time-step*.
- At each time-step, a cell reads up to two operands (from input ports or local storage)³ and another operand from within the cell if required, performs an operation (or starts the operation if the cell is pipelined), and delivers results to output ports, local storage and/or for internal reuse.
- At each time-step, a cell produces up to two outputs for neighbor cells. Such outputs may be results computed within the cell or transmittent data.

From the discussion above, realizing an algorithm-specific array requires to:

³There may be occasions when the three operands must be brought into a cell from external sources. In such cases, one of the two ports is used for two operands. However, this situation is not frequent (once every $O(n)$ time steps), so that using a single port for two purposes imposes only a minor performance degradation on the array. Consequently, unless explicitly stated otherwise, we ignore this fact in the rest of this chapter.

- specify the characteristics of cells, such as size of local storage, bandwidth, operations performed, and number and type of functional units.
- specify the array topology and the number of cells.
- specify the flow of up to three data elements per operation.
- schedule the operations throughout the entire array.

In contrast, mapping a matrix algorithm onto a class-specific array requires only to

- specify the flow of data
- schedule the operations on the target array

4.2 Summary of the data-dependency graph-based design approach

A transformational design method based on the dependencies of algorithms is proposed in this dissertation. Starting from a fully-parallel data-dependency graph (FPG), in which nodes represent operations and edges correspond to data dependencies, the method applies transformations to the graph to incorporate implementation restrictions and handle design issues. Such transformations produce another graph suitable for direct realization as an algorithm-specific array or for mapping onto a class-specific array.

We suggest using a fully-parallel data-dependency graph as the description tool because this notation exhibits the intrinsic features of an algorithm. Such a graph could be used to derive an implementation by assigning each node to a different processing element (PE), and by adding delay registers to synchronize the arrival of data to PEs. The resulting structure (a pipelined realization of the graph) exhibits minimum delay (determined by the longest path in the graph) and optimal throughput (for multiple-instance computations), but might require non-neighbor and varying distance connections, large I/O bandwidth, and large number of units. The method presented in this chapter deals with these problems, while preserving the features inherent in the data-dependency graph.

As stated in Chapter 3, a design process consists of two stages: regularization and derivation of arrays. These two stages for our method, and the steps within them, are depicted in the high-level description shown in Figure 4.4. The method is summarized below, where we also indicate the suitability for automation of the different steps involved. Each step is described in detail afterwards.

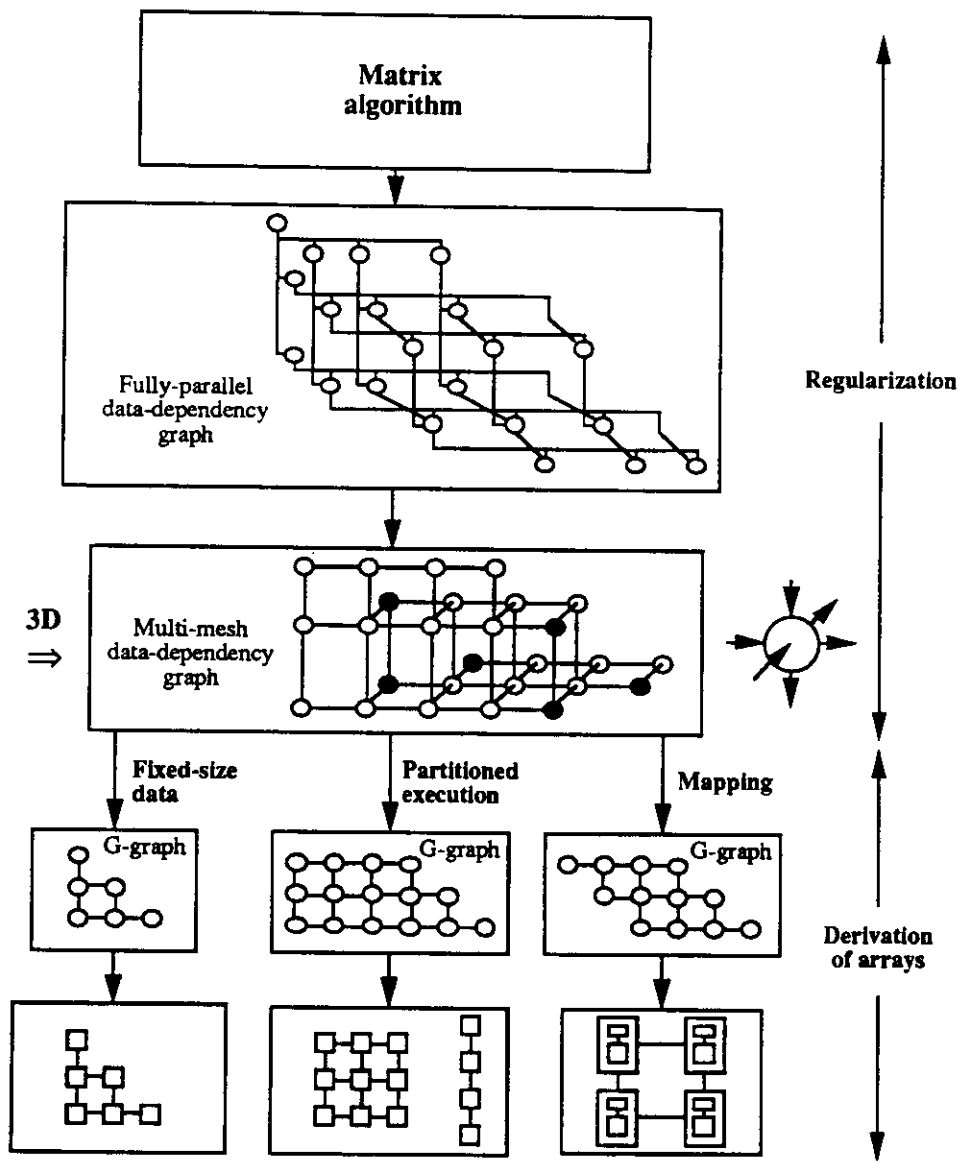


Figure 4.4: Data-dependency graph-based design method

4.2.1 The regularization stage

The regularization stage starts with the fully-parallel data-dependency graph and produces a three-dimensional graph that we call a *multi-mesh dependency graph* (MMG). This MMG is a unidirectional graph with nearest-neighbor dependencies, edges along axes of the three-dimensional space, and nodes at integer values of the axes. This regularization stage is performed as follows:

1. Draw the fully-parallel data-dependency graph (FPG) of the matrix algorithm. Such a graph is obtained by tracing the execution of the algorithm (i.e., outer-loop and loop-body). That is, symbolically execute the algorithm tracking which variables are used and when, and allocate operations to nodes and data references to edges of a graph.
2. Transform the fully-parallel data-dependency graph into a three-dimensional multi-mesh graph (MMG). To achieve this objective, perform transformations on the FPG to remove properties that are not allowed in the MMG, namely broadcasting, bidirectional flow of data, and non-regular dependencies. The set of transformations will be presented later.

These two steps could be performed automatically. The resulting MMG is used in the second stage of the method to realize algorithm-specific arrays for fixed-size data and partitioned problems, or to map algorithms onto class-specific arrays.

It should be noted that some researchers have regarded the dependency graph of a matrix algorithm as a multi-dimensional instead of a three-dimensional graph. Such a conclusion has been obtained from representing in a graph the *index-dependencies* in the algorithm. That is, the dimensionality of the graph has been defined by the number of indices that appear in an algorithm. In that approach, every variable is required to have all indices, so that each instance of a variable is associated with a point in the multi-dimensional index space [Rao88, Quin84, Mold83]. In contrast, the dimensionality of a *data-dependency* graph is defined by the number of inputs and outputs to/from primitive nodes that compose the graph, and matrix algorithms are characterized by primitive operators that have at most three operands. An extensive comparison between data-dependencies and index-dependencies, and their suitability for a design method, is given in Chapter 7.

4.2.2 The derivation of arrays

The second stage is as follows:

1. Collapse the MMG onto a two-dimensional graph (i.e., a *G-graph*,) by grouping primitive nodes onto *G-nodes*.

As is described later, this grouping determines properties of cells such as local storage, communication bandwidth, and cell pipelining. Moreover, grouping is driven by the target implementation (i.e., algorithm-specific array for fixed-size data, partitioned implementation, or mapping onto a class-specific array). Consequently, this step allows performing optimization of specific measures based on implementation constraints.

2. Schedule the order of execution of the primitive operations that compose a G-node. As is shown later, this scheduling impacts local storage and cell bandwidth.
3. *For problems with fixed-size data on two-dimensional structures*, realize the G-graph obtained in (1) as an array by allocating each G-node to a different PE and each edge to a different communication link.

For problems with fixed-size data on linear arrays, apply *cut-and-pile* to the G-graph obtained in (1). Each partition corresponds to a complete horizontal or vertical path of the G-graph. Nodes in a partition (i.e., a cut) are executed concurrently, while different partitions are scheduled (i.e., piled) for pipelined execution in the array. Each G-node in a cut is allocated to a different PE and each edge to a different communication link.

For partitioned problems:

- (a) Divide (i.e., cut) the G-graph obtained in (1) into sets of neighbor G-nodes (*G-sets*), where each G-set has as many nodes as there are cells in the array. Nodes in a G-set are structured in linear or two-dimensional manner, depending on the desired array topology, and are executed concurrently in the array.
- (b) Schedule (i.e., pile) G-sets for execution. G-sets are executed in overlapped (pipelined) manner and data flows between G-sets.

For mapping onto class-specific arrays:

- (a) Divide (i.e., cut) the G-graph obtained in (1) into sets of neighbor G-nodes (G-sets) whose characteristics (i.e., number and topology of nodes) are determined by characteristics of the specific cells and array.
- (b) Schedule (i.e., pile) G-sets for execution.

Most of the steps, both in the regularization stage and in the derivation of arrays, could be performed automatically. Exceptions are the selection of G-nodes and the mapping onto class-specific arrays, which would require input from a designer. These steps depend on the characteristics and restrictions of the implementation, so they are less amenable for complete automation. However, such steps could be aided by a CAD tool.

4.2.3 The performance and cost measures

To determine the performance of arrays derived with the method outlined above, we use the following measures (where N is the number of operations in the algorithm):

T	Throughput
K	Number of cells
U	Utilization ($U = N/KT^{-1}$)
$A_{I/O}$	Input/output bandwidth
C_{BW}	Cell bandwidth
C_w	Storage per cell

These measures are computed with information obtained from the dependency graphs, both the original FPG and the transformed graphs. Moreover, transformations used in the method affect such measures, so that one can study the impact of a particular transformation on cost and performance of the resulting array while carrying out the transformation.

In the following sections, we discuss each step of this method in detail.

```

For  $r$  from 1 to  $n - 1$ 
begin
  For  $i$  from  $(r + 1)$  to  $n$ 
  begin
     $\theta_{ri} = -\arctan\left(\frac{a_{ir}}{a_{rr}}\right)$  ,  $a_{rr} = \sqrt{a_{ir}^2 + a_{rr}^2}$ 
    For  $j$  from  $(r + 1)$  to  $n$ 
    begin
      
$$\begin{bmatrix} a_{rj} \\ a_{ij} \end{bmatrix} = \begin{bmatrix} \cos \theta_{ri} & -\sin \theta_{ri} \\ \sin \theta_{ri} & \cos \theta_{ri} \end{bmatrix} \begin{bmatrix} a_{rj} \\ a_{ij} \end{bmatrix}$$

    end
    
$$\begin{bmatrix} b_r \\ b_i \end{bmatrix} = \begin{bmatrix} \cos \theta_{ri} & -\sin \theta_{ri} \\ \sin \theta_{ri} & \cos \theta_{ri} \end{bmatrix} \begin{bmatrix} b_r \\ b_i \end{bmatrix}$$

  end
end

```

Figure 4.5: The triangularization algorithm by Givens' rotations

4.3 Obtaining the fully-parallel data-dependency graph

As indicated in Section 4.1, the input to the method is a matrix algorithm described by an outermost loop and a loop-body consisting of scalar, vector and matrix operators. For example, Figure 4.5 depicts the triangularization algorithm by Givens' rotations, which fulfills the admissible form. Note that an algorithm expressed by other means can be easily transformed into this type of description.⁴

The fully-parallel data-dependency graph (FPG) is obtained from a symbolic execution of the algorithm, which generates an ordered list of expressions. For example, Figure 4.6 depicts partially the list of expressions from symbolic execution of the triangularization algorithm shown in Figure 4.5, for a problem of size $n = 4$. This list contains, implicitly, the dependencies between operations that allow extracting the existing parallelism. Such a list is used to draw a graph where each operator is allocated to a node and references to variables are allocated to edges. This graph, which is equivalent to an unfolded dataflow graph, is a complete and accurate representation of the algorithm because there is a one-to-one correspondence between operators and nodes, and between dependencies and edges. We

⁴Strictly speaking, one could derive the fully-parallel data-dependency graph directly from a different algorithm description, without necessarily first transforming such an algorithm into a set of loops.

$$\begin{aligned}
\theta_{12} &= -\arctan(a_{21}/a_{11}) , \quad a_{11} = \sqrt{a_{21}^2 + a_{11}^2} \\
a_{12} &= \cos \theta_{12} a_{12} - \sin \theta_{12} a_{22} , \quad a_{22} = \sin \theta_{12} a_{12} + \cos \theta_{12} a_{22} \\
a_{13} &= \cos \theta_{12} a_{13} - \sin \theta_{12} a_{23} , \quad a_{23} = \sin \theta_{12} a_{13} + \cos \theta_{12} a_{23} \\
a_{14} &= \cos \theta_{12} a_{14} - \sin \theta_{12} a_{24} , \quad a_{24} = \sin \theta_{12} a_{14} + \cos \theta_{12} a_{24} \\
b_1 &= \cos \theta_{12} b_1 - \sin \theta_{12} b_2 , \quad b_2 = \sin \theta_{12} b_1 + \cos \theta_{12} b_2 \\
\theta_{13} &= -\arctan(a_{31}/a_{11}) , \quad a_{11} = \sqrt{a_{31}^2 + a_{11}^2} \\
a_{12} &= \cos \theta_{13} a_{12} - \sin \theta_{13} a_{32} , \quad a_{32} = \sin \theta_{13} a_{12} + \cos \theta_{13} a_{32} \\
a_{13} &= \cos \theta_{13} a_{13} - \sin \theta_{13} a_{33} , \quad a_{33} = \sin \theta_{13} a_{13} + \cos \theta_{13} a_{33} \\
a_{14} &= \cos \theta_{13} a_{14} - \sin \theta_{13} a_{34} , \quad a_{34} = \sin \theta_{13} a_{14} + \cos \theta_{13} a_{34} \\
b_1 &= \cos \theta_{13} b_1 - \sin \theta_{13} b_3 , \quad b_3 = \sin \theta_{13} b_1 + \cos \theta_{13} b_3 \\
&\vdots \\
\theta_{23} &= -\arctan(a_{32}/a_{22}) , \quad a_{22} = \sqrt{a_{32}^2 + a_{22}^2} \\
a_{23} &= \cos \theta_{23} a_{23} - \sin \theta_{23} a_{33} , \quad a_{33} = \sin \theta_{23} a_{23} + \cos \theta_{23} a_{33} \\
a_{24} &= \cos \theta_{23} a_{24} - \sin \theta_{23} a_{34} , \quad a_{34} = \sin \theta_{23} a_{24} + \cos \theta_{23} a_{34} \\
b_1 &= \cos \theta_{23} b_1 - \sin \theta_{23} b_2 , \quad b_2 = \sin \theta_{23} b_1 + \cos \theta_{23} b_2 \\
&\vdots
\end{aligned}$$

Figure 4.6: Symbolic evaluation of the triangularization algorithm

refer to this graph as a *fully-parallel data-dependency graph* (FPG), which uniquely describes the algorithm. Moreover, this graph corresponds to a *single assignment representation* of the algorithm.

To aid drawing the FPG, one can exploit the structure available within the algorithm in the form of vector and matrix operators. Vector operators are drawn as linear sets of nodes, while matrix operators correspond to two-dimensional sets of nodes, as illustrated in Figure 4.2. Moreover, the FPG consists of a sequence of subgraphs with similar structure, where each subgraph corresponds to one iteration of the outer loop. These aspects are illustrated in Figure 4.7, which shows the structure obtained in the FPG of the triangularization algorithm for a 4 by 4 matrix. In this case, each subgraph is composed of scalar and vector operations that are dependent. For example, at the top of the first subgraph there is one scalar operation that computes a rotation angle, and one vector operation that rotates the first two rows of A and b . Then, a new rotation angle is computed using data from row 3 and the updated row 1, and these two rows are rotated. Such a process is repeated throughout the graph.

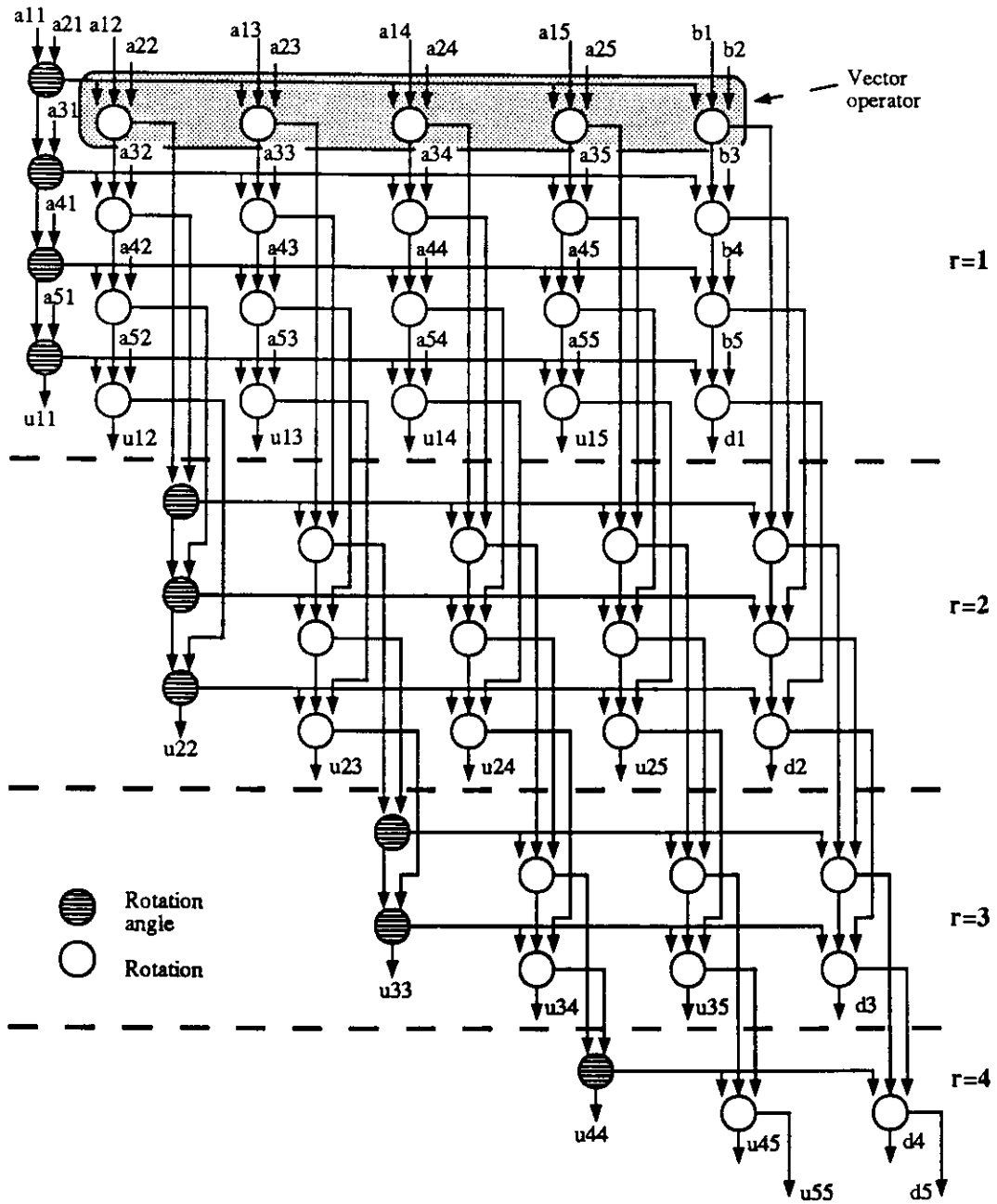


Figure 4.7: The FPG of the triangularization algorithm by Givens' rotations

Alleged drawbacks of FPGs for matrix algorithms are the complexity in their derivation and the size of such graphs (usually $O(n^3)$). However, as stated above, an FPG can be easily derived by symbolic execution of an algorithm. Moreover, because of the regularity of matrix algorithms, an FPG is derived for a small-size problem (i.e., a 3 by 3 to a 6 by 6 matrix) and the results extended to larger problems.

A formal procedure to derive an FPG is described in Chapter 5.

4.4 Obtaining the multi-mesh data-dependency graph

A *multi-mesh data-dependency graph* (MMG) has the following characteristics:

- Nodes only at points defined by integer values in a three-dimensional space.
- Unidirectional dependencies along axes of the three-dimensional space.
- Dependencies only between nearest-neighbor nodes (i.e., all edges have length one).

There are two types of MMGs: *complete* (CMMG) and *incomplete* (IMMG). A CMMG, as depicted in Figure 4.8a, has the structure of a cube because it is composed of meshes that have the same number of nodes. For a problem of size n , such a CMMG corresponds to an algorithm that has the maximum number of operations (n^3) and dependencies. In contrast, an IMMG has some nodes and edges missing at the outer portions of the graph. We restrict the number of missing nodes in an IMMG to monotonically increase or decrease along axes of the three-dimensional space. Figure 4.8b gives examples of IMMGS. In what follows, we use MMG to refer to both CMMG and IMMG.

Matrix algorithms always exhibit transmittent (i.e., broadcasted) data. As a convention, and unless stated otherwise, we assume that this transmittent data flows along the X -axis in an MMG.

We address now transforming an FPG into an MMG. In Chapter 5 we show that the FPG of a set of scalar operations, with up to three operands and three outputs each, is always representable in a three-dimensional space with unidirectional flow of data along axes of the space, and computing nodes distributed throughout the space with non-neighbor dependencies. Such a graph is transformed into an MMG by adding delay nodes between non-neighbor nodes. However, the resulting MMG

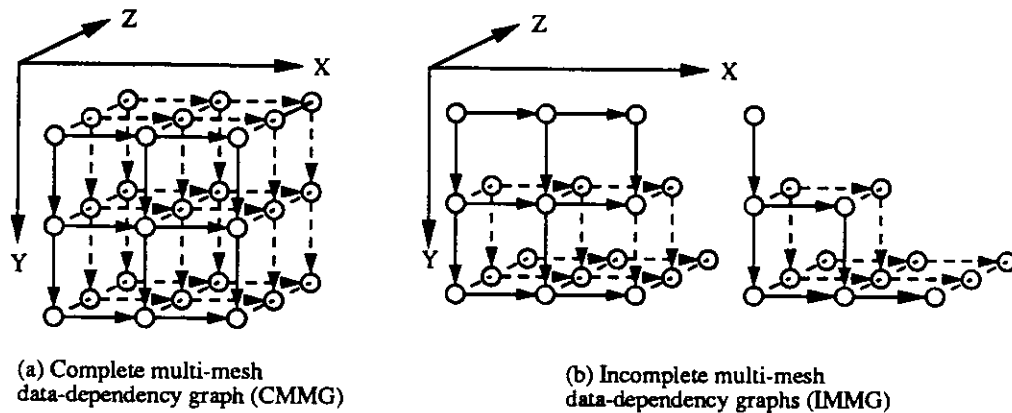


Figure 4.8: Examples of multi-mesh data-dependency graphs

has many delay nodes connecting computing nodes that are located far apart in the space, so that it is not suitable for efficient implementation in an array. In contrast, the structure of matrix and vector operators in matrix algorithms allows obtaining MMGs with most computing nodes at nearest-neighbor locations in the three-dimensional space and few delay nodes, as described in Chapter 5.

Transforming an FPG into an MMG (i.e., obtaining a regular description of an algorithm) consists of:

- Removing from the FPG those characteristics that are not allowed in the MMG. These are
 - broadcasting
 - bidirectional flow of data
 - non-regular dependencies
 - non-nearest neighbor dependencies
- Drawing the resulting graph as a three-dimensional structure.

Removing undesirable properties and drawing the graph in three dimensions are aided by the visual (i.e., graph) representation of the algorithm. For example, upon visually detecting in the FPG an undesirable characteristic, a suitable transformation is applied to the graph to remove it. Such transformations are formally defined in Chapter 5. Moreover, drawing the graph as a three-dimensional structure is achieved by allocating each iteration of the outer loop in the algorithm to one plane (or three-dimensional subgraph) in the space, and the different iterations are allocated to neighbor planes (or three-dimensional subgraphs). In Chapter 5,

we state the conditions required for allocating each iteration of the outer loop to a single plane (i.e., mesh) in the three-dimensional space.

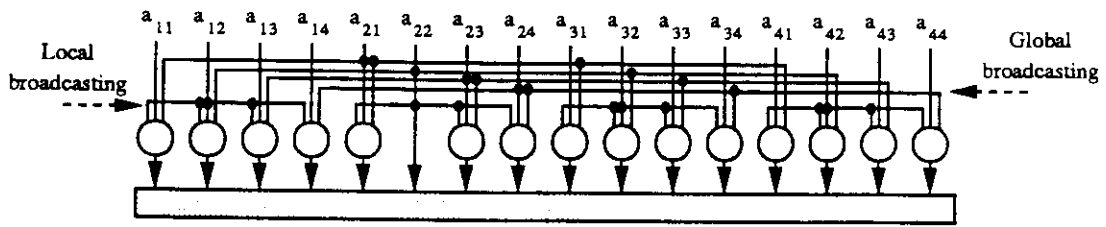
An example of the regularization process to obtain an MMG is shown in Figure 4.9. (This graph corresponds to a portion of that for the transitive closure algorithm.) Figure 4.9a shows one level of an FPG that contains several broadcasted data elements. Some of these elements are distributed throughout the entire level of the graph (global broadcasting), while others are broadcasted in a local manner (i.e., to neighbor nodes). This broadcasted data is replaced by transmittent data so that data flows through nodes, as illustrated in Figure 4.9b. Global broadcasting is replaced by transmittent data flowing along the X -axis, while locally broadcasted values become transmittent data flowing along the Z -axis.

The resulting graph in Figure 4.9b has the structure of one mesh but not all dependencies are between nearest neighbors. Moreover, such a graph exhibits bidirectional flow of data. These bidirectional dependencies are removed by flipping nodes at the left (outer) of the source of broadcasting to the right (inner) side of such a broadcasting, as shown in Figure 4.9c. In addition, a delay node has been added to make all dependencies between nearest neighbors. The resulting graph is a complete mesh, with unidirectional edges along each axis.

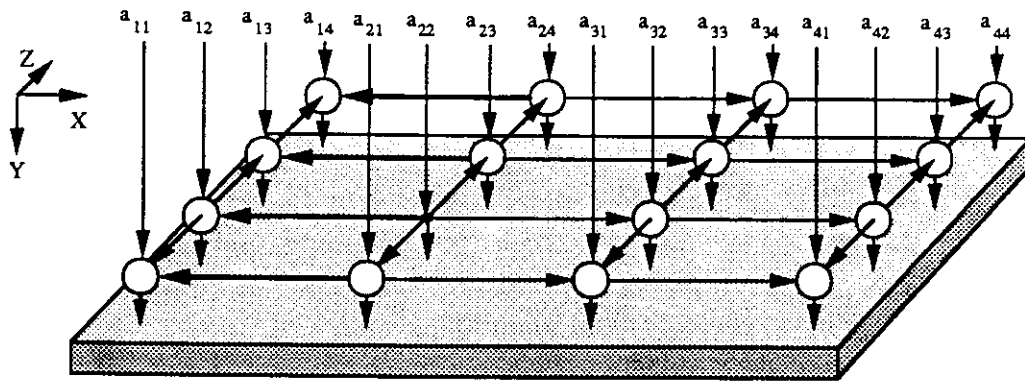
Similar transformations to those above are applied to the different levels of an FPG, so that each level is transformed into a mesh, and the entire graph becomes an MMG, either complete or incomplete.

Let us look into deriving the MMG for the triangularization algorithm. The FPG in Figure 4.7 exhibits data broadcasting, which corresponds to values of angles used to rotate pairs of rows of the matrix. This broadcasting is replaced by transmittent data, as shown in Figure 4.10. Moreover, nodes in the topmost section receive one external input, excepting nodes at the very top of the graph, which receive two external inputs. Such an "irregularity" is removed by adding a level of delay nodes that receives one external input, whose output become an input to nodes formerly with two external inputs, as shown in Figure 4.10.

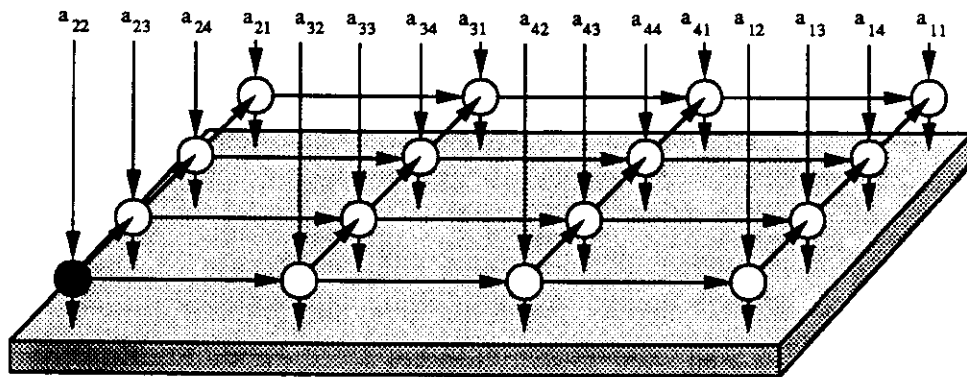
The graph in Figure 4.10 is now transformed into a three-dimensional structure by allocating each section (i.e., one iteration of the outermost loop) to a different plane of a three-dimensional space, as shown in Figure 4.11. Delay nodes have been added at the top of inner planes along the Z -axis, so that the resulting graph has dependencies only among nearest-neighbor nodes. The resulting graph has unidirectional flow of data, and corresponds to the multi-mesh data-dependency graph.



(a) One level of a fully-parallel dependency graph



(b) Replacing broadcasting by transmittent data



(c) Removing bidirectional flow of data and irregular dependencies

Figure 4.9: Removing properties not allowed in an MMG

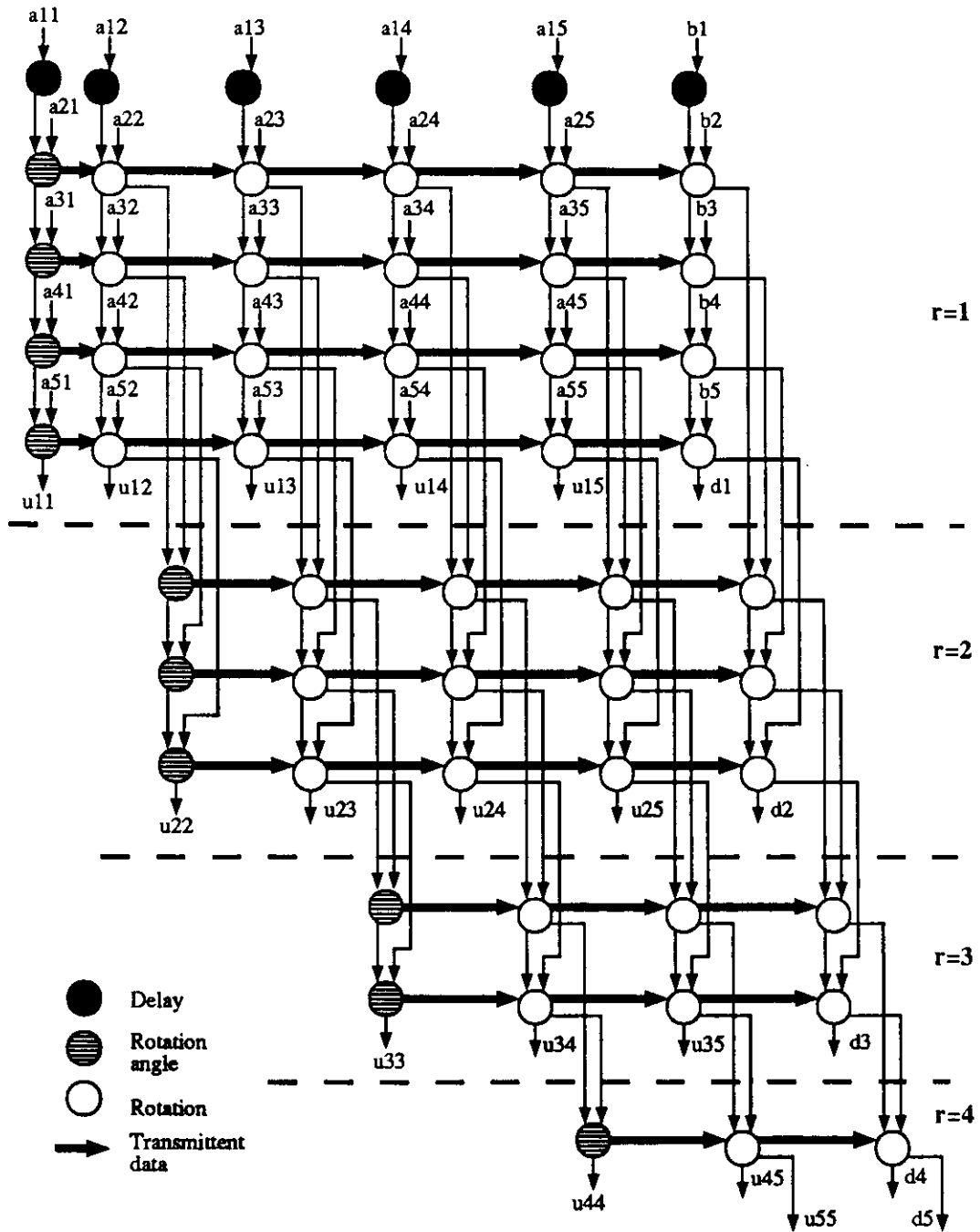


Figure 4.10: Graph with no broadcasting for the triangularization algorithm

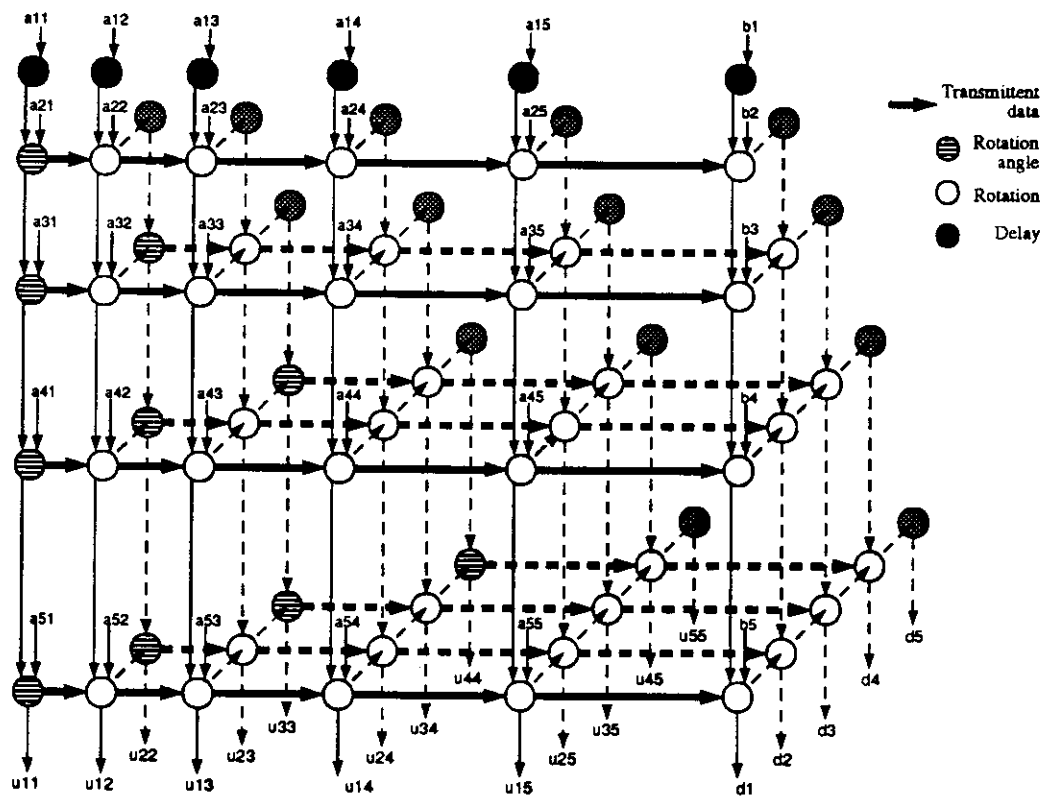


Figure 4.11: Multi-mesh dependency-graph for the triangularization algorithm

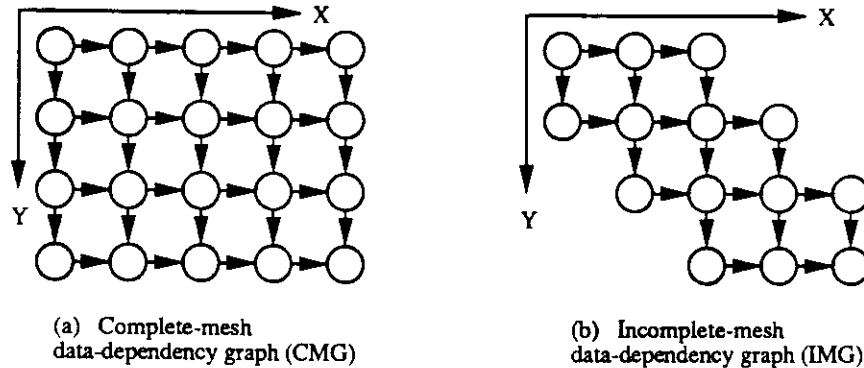


Figure 4.12: Examples of mesh data-dependency graphs (G-graphs)

4.5 Deriving G-graphs from a complete multi-mesh graph

We discuss now the process of deriving G-graphs from an MMG. This process consists of collapsing the MMG onto a two-dimensional graph (the *G-graph*) and scheduling the execution of the primitive operations that compose the nodes of such a G-graph (the G-nodes). This graph is later realized as (or mapped onto) an array.

A G-graph is a *mesh data-dependency graph* (MG) that has the following characteristics:

- Nodes only at points defined by integer values in a two-dimensional space.
- Unidirectional dependencies along axes of the two-dimensional space.
- Dependencies only between nearest-neighbor nodes (i.e., all edges have length one).

Similarly to MMGs, there are two types of MGs: *complete* (CMG) and *incomplete* (IMG). A complete MG consists of a rectangular mesh of nodes, while an IMG is obtained by removing some nodes and edges from the outer portions of a rectangular mesh. We restrict the number of nodes removed to either increase or decrease monotonically along axes of the two-dimensional graph. Figure 4.12 gives examples of G-graphs (i.e., MGs).

We first discuss deriving an MG from a *complete* multi-mesh data-dependency graph (CMMG) as the one shown in Figure 4.13, which has the following characteristics:

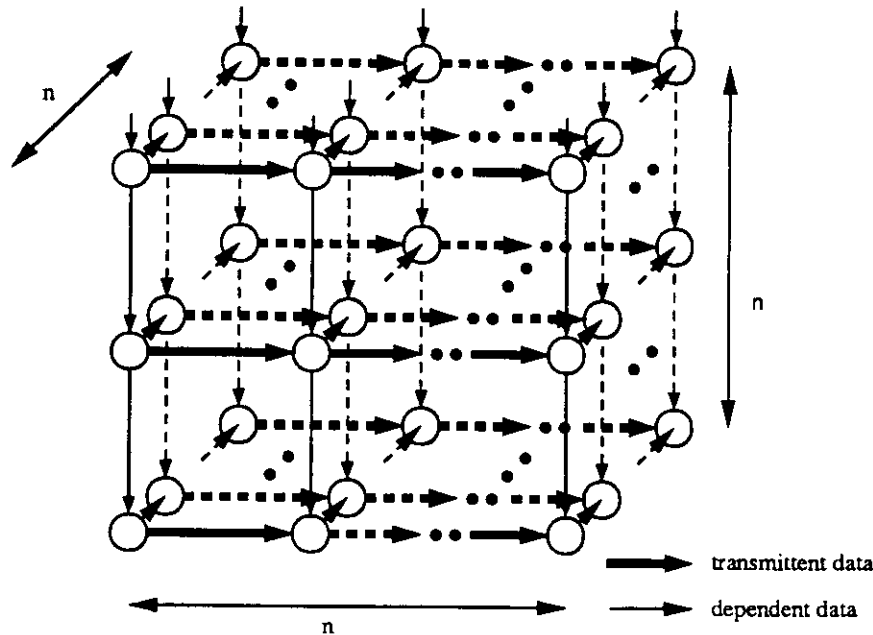


Figure 4.13: The CMMG used to discuss the derivation of arrays

- external input only from the top⁵
- only computing nodes (i.e., no delay nodes)

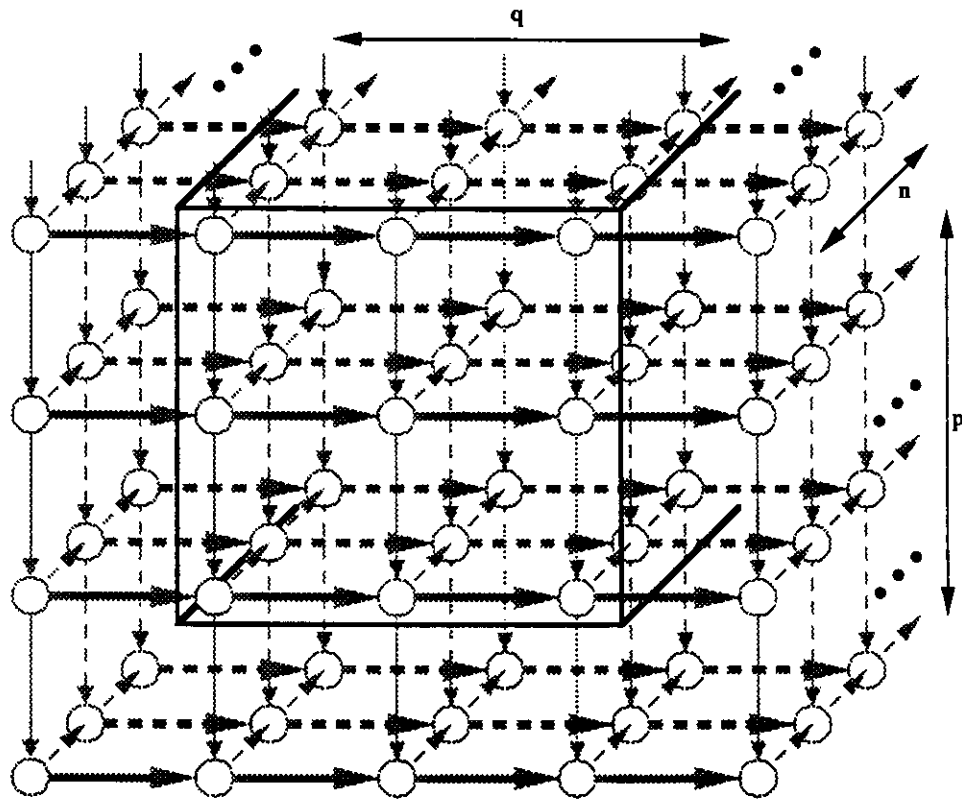
Later, we will address algorithms that are represented by IMMGS, as well as MMGS with two flows of external inputs. IMMGS are usually characterized by the existence of delay nodes in addition to computation nodes. The method is applied in the same manner to all cases, but the tradeoffs in the characteristics of cells and arrays are different.

4.5.1 The collapsing of a CMMG

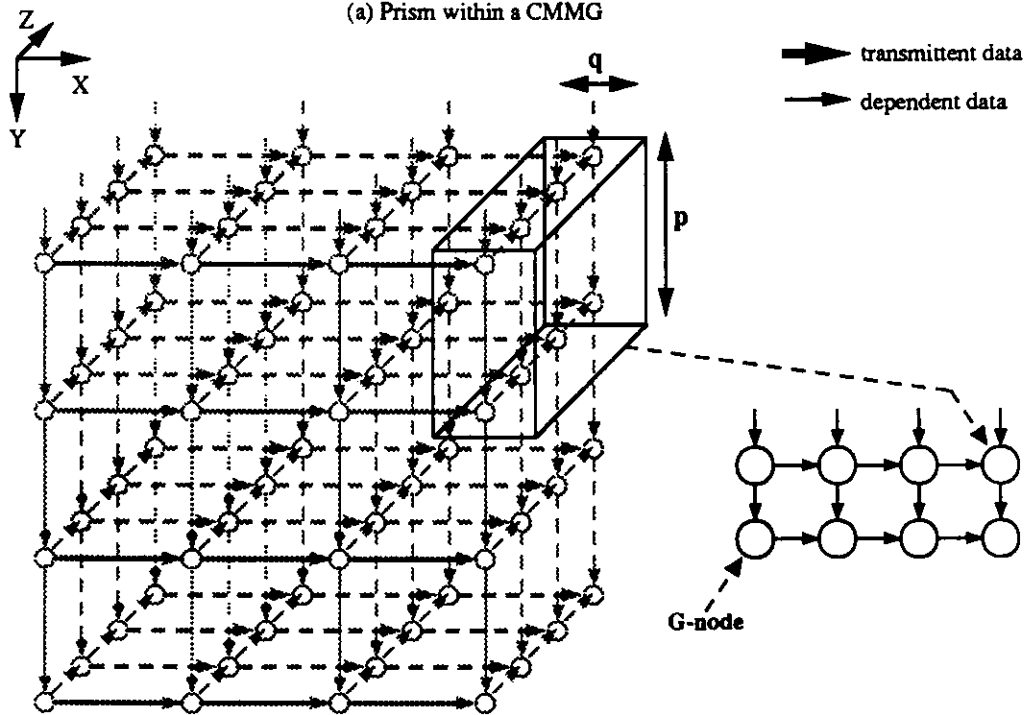
To transform a CMMG into a G-graph, we coalesce prisms of size p by q by n in the CMMG onto G-nodes, as depicted in Figure 4.14. Criteria to select the prisms depend on the target array.

Collapsing the CMMG requires to choose a suitable direction for the axis of the prism. As described in Chapter 5, grouping along axes of the three-dimensional space leads to simpler and more efficient implementations. Consequently, select-

⁵This is the case of many important matrix algorithms, such as LU-decomposition, QR-decomposition, Faddeev algorithm, transitive closure.



(a) Prism within a CMMG



(b) Collapsing CMMG onto G-graph

Figure 4.14: Deriving G-graphs from a CMMG

ing the direction of the prism axis may be limited to three alternatives, namely along direction X , Y , or Z . Figure 4.14 illustrates the case of prisms along axis Z . In what follows, unless explicitly stated otherwise, we consider prisms along axis Z , which is orthogonal to the flow of transmittent data along axis X . This orthogonality is necessary to use pipelined cells, as will be discussed later.

4.5.2 The execution of a G-node

The functionality of a G-node, as well as its computation time, are determined by the primitive nodes enclosed in a prism. Executing a G-node implies the sequential execution of the primitive nodes that compose it, which requires the selection of a schedule for such nodes. Moreover, the schedule and the size of the prisms' base determine several characteristics of the implementation obtained when realizing the G-graph as an array, such as the cell bandwidth, the size of local storage in a cell, and the ability to use pipelined cells.

The execution of an operation in a cell requires reading operands from input ports or from local storage, performing the corresponding operation, and delivering results to output ports or local storage. These operations correspond to primitive nodes of a G-node. Nodes at the boundaries of a prism access ports, while nodes inside the prism access local storage and the feedback loop. This characteristic impacts the properties of cells, as discussed below.

For reasons that will become apparent from the discussion that follows, we select to schedule primitive nodes by meshes of size p by q , as depicted in Figure 4.15, where nodes have been tagged with their scheduling time. That is, all nodes in one mesh of size p by q are executed before scheduling a node from the next mesh.

4.5.3 Cell properties that depend on the G-graph

We discuss now the characteristics of cells that are direct results from the selection of prisms and the schedule of primitive nodes.

Computation time of a G-node

The computation time of a G-node is given by

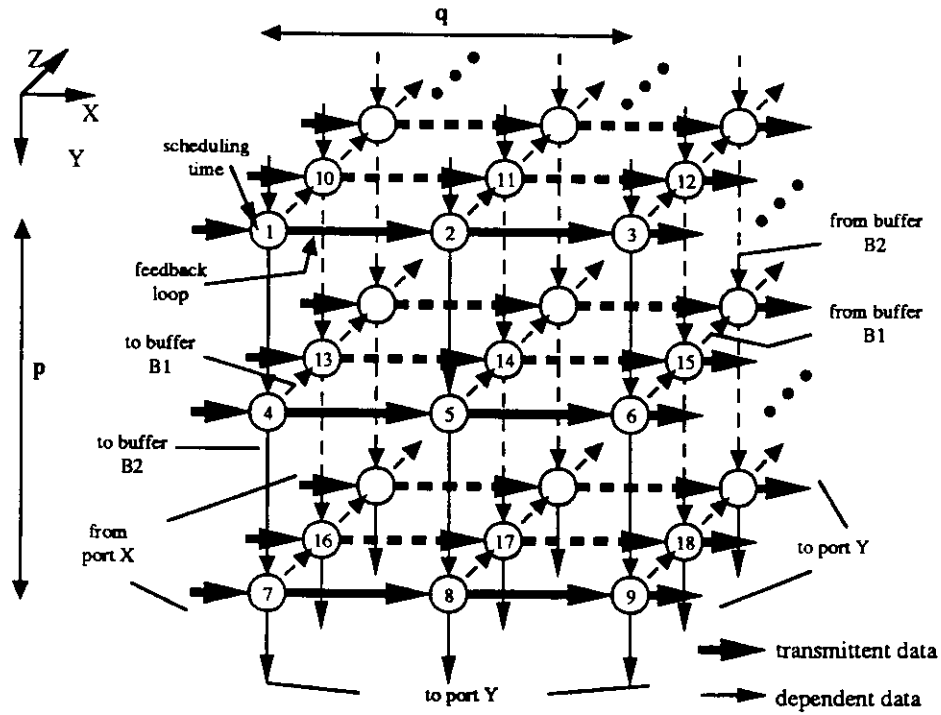


Figure 4.15: The schedule of primitive nodes within a prism

$$t_G = pqn$$

because each prism has pqn primitive nodes.

Number of G-nodes

The number of G-nodes in the G-graph is given by

$$N_G = n^2/pq$$

because there are n^3 primitive nodes that are coalesced into G-nodes of size pqn .

Cell communication bandwidth

There are two flows of data arriving to and leaving from a prism, namely along axes X and Y , as depicted in the Figure 4.15. When a G-node is realized as a cell,

these flows are assigned to ports in the same direction as that of arrival/departure. As a result, average cell communication bandwidth per port is determined by the total number of edges of the CMMG that are cut by the sides of the prism that defines a G-node. Consequently,

Average cell bandwidth, horizontal (X) port

$$C_{BW}^X = pn/pqn = 1/q \text{ [words/time - step]}$$

Average cell bandwidth, vertical (Y) port

$$C_{BW}^Y = qn/pqn = 1/p \text{ [words/time - step]}$$

because pn (qn) operations take an input from outside the prism throughout the entire execution of the prism.

Maximum cell bandwidth is determined by the schedule of primitive nodes. With the schedule indicated in Figure 4.15, maximum cell bandwidth is identical to the average. In such a case, the rightmost (lower) boundary of the prism is reached every q (p) primitive operations.

As stated in page 60, there may be occasions when a cell reads three operands from external sources. In terms of the G-graph, this means that the prism receives a third flow of data along the Z -axis. As also stated in page 60, such a flow is allocated to either one of the two cell ports, say the X -port. Using the schedule shown in Figure 4.15, the corresponding bandwidth required while reading such an input data (i.e., transferring data associated to the Z -axis at the outer mesh of the prism) is larger than the values above (i.e., $(pq + p)/pq = 1 + 1/q$). Instead of providing this large bandwidth, it is possible to maintain cell bandwidth of $1/q$ by transferring to a local storage the pq elements flowing along the Z -axis before starting the execution of primitive nodes of the prism. This approach requires $(pq)q = pq^2$ additional time-steps, increasing the G-node computation time to $t_G = pqn + pq^2 = pq(n + q)$ (and reducing utilization of cells). However, for $n \gg 1$, this effect is negligible.

Local storage per cell

Executing primitive nodes in a prism requires to store the output of one mesh (to be used as input to the next mesh) and the output of one horizontal path

within the mesh (to be used as input in the next horizontal path). When primitive nodes are scheduled by meshes of size p by q , as shown in Figure 4.15, the storage requirements are

$$C_w = pq + q = q(p + 1)$$

In this expression, pq locations are required to store the output of one mesh, and q locations are needed to store the output of one horizontal path.

Storage access and organization

Each input to a primitive node is associated with flow of data along one axis of the graph. These three flows of data (along the three axes in Figure 4.15) must be assigned to data paths inside a cell. Such an assignment determines the allocation of data flows to ports and local storage. Since there is at least one flow of transmittent data and up to two flows of dependent data within the prism, we allocate these flows as follows:

- To facilitate using pipelined cells, as discussed later, transmittent data within the prism (i.e., the X -flow in the figure) is allocated to the feedback loop within a cell.
- The Z -flow within the prism is allocated to a buffer B_1 , while the Y -flow is allocated to a buffer B_2 , as depicted in Figures 4.15 and 4.16.

Note that allocating a flow of transmittent data to the feedback loop allows replacing such a loop by a register that holds the transmittent data and provides it to the functional unit at every time-step. This data is also transferred to the neighbor cell, through the X port.

Consequently, local storage in a cell should be organized as two independent buffers that serve each of the two flows of data. Such buffers have size pq and q respectively, each with bandwidth of 1 [word/time-step]. In that case, buffers are accessed without conflicts as FIFOs, as dictated by the schedule of primitive nodes.

Cell bandwidth is $1/p$ (or $1/q$) [word/time-step]. In contrast, buffers are accessed at a rate of 1 [word/time-step] and cell computation rate is 1 [op/time-step].

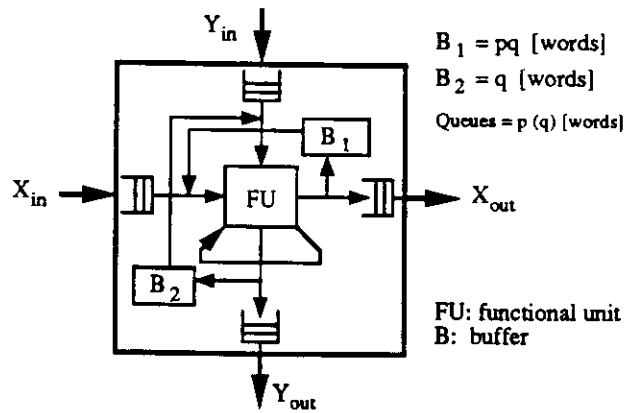


Figure 4.16: Local storage organization in a cell

The difference between these rates and cell bandwidth is adjusted by adding queues to cell ports, as shown in Figure 4.16. In this way, transfers in/out of a cell occur at the rate of $1/p$ (or $1/q$), while transfers between queues and functional unit have a maximum rate of 1 [word/time-step]. Such queues have size p and q , respectively.

Pipelined cells

Using pipelined cells requires data-independent operations scheduled at successive time-steps. In an MMG, nodes that are connected by transmittent data correspond to data-independent nodes, because their dependency in the graph arises from broadcasting, as shown in Figure 4.17. A schedule that follows the transmittent flow guarantees that nodes scheduled successively are data-independent, as long as the length of the pipeline (P_s) is shorter than the length of the transmittent path within the G-node (i.e., $P_s \leq q$). Consequently, we choose the schedule shown in Figure 4.17, which also corresponds to the one used in Figure 4.15.

Using pipelined functional units reduces storage requirements in a cell to a value smaller than $C_w = q(p + 1)$. Since data takes P_s time-steps to become available at the output of the pipeline, for each flow of data there are P_s values in the pipeline that do not need to be saved in local storage. Consequently,

$$C_w = (pq - P_s) + (q - P_s) = pq + q - 2P_s$$

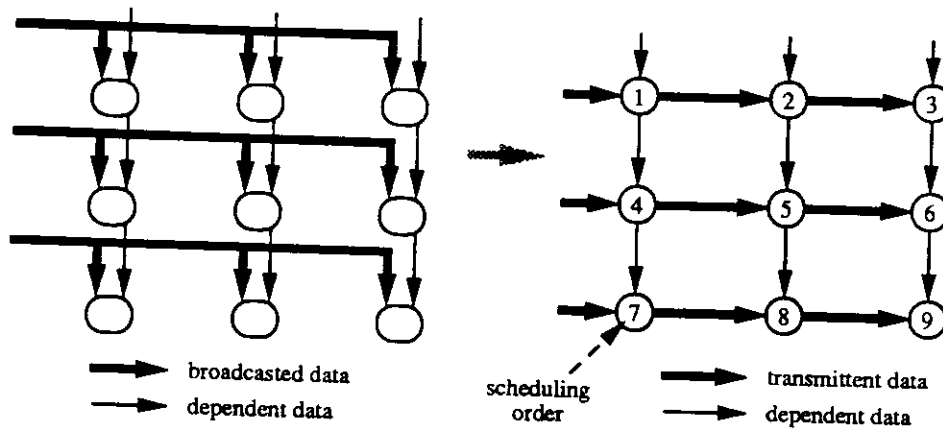


Figure 4.17: Independent nodes in the flow of transmittent data

Systolic cells

A particular case of G-graph is obtained when selecting prisms with base size 1 by 1 (i.e., $p = q = 1$). This grouping corresponds to *projecting* the CMMG onto a G-graph along one axis, leading to G-nodes whose execution needs no local storage, and cell bandwidth has the same rate as computation rate. (Values of $p = q = 1$ lead to $pq + q = 2$, which corresponds to two storage locations required to latch input operands.) Such nodes are suitable for implementation in systolic cells.

Note that increasing the degree of pipelining in systolic cells implies higher computation rate but requires higher cell bandwidth as well. This is not the case with pseudo-systolic cells, which may perform tradeoffs and adjustments of cell bandwidth and local storage.

Non-pipelined cells

Non-pipelined cells do not require to schedule primitive operations following the flow of transmittent data, because the results from one primitive node are available before starting the execution of the next node. Consequently, there is no relation between the direction of transmittent flow and the collapsing direction.

Moreover, in the case of non-pipelined systolic cells (which have no local storage) there is only one schedule possible which is determined by the dependencies in the MMG.

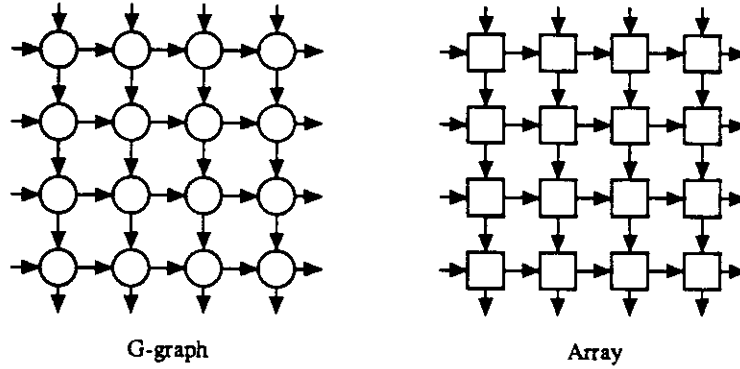


Figure 4.18: Realizing a G-graph as an array for fixed-size data

4.6 Deriving arrays for fixed-size data from a CMMG

We discuss now realizing the G-graph obtained from a CMMG as an algorithm-specific array. First, we look into problems with fixed-size data, and then partitioned implementations. We defer mapping onto class-specific arrays until Chapter 6.

4.6.1 Two-dimensional arrays

For problems with fixed-size data, the G-graph is directly realized as a two-dimensional array. That is, each G-node is realized as a different cell, and each edge in the G-graph is realized as a different communication link in the array. Figure 4.18 depicts an example of this process.

The performance of the array is directly obtained from the characteristics of the G-graph. In particular:

$$\begin{aligned}
 \text{Throughput} \quad T_f &= t_G^{-1} = [pqn]^{-1} \\
 \text{Number of cells} \quad K_f &= N_G = n^2/(pq) \\
 \text{Cell bandwidth} \quad C_{BW}^X &= 1/q \\
 &C_{BW}^Y = 1/p \\
 \text{Utilization} \quad U_f &= N/(K_f T_f^{-1}) = \frac{n^3}{(n^2/pq)pqn} = 1
 \end{aligned}$$

These measures indicate that, for a problem with fixed-size data, an algorithm represented by a CMMG is realized as a two-dimensional array with optimal utilization.

4.6.2 Linear arrays

Realizing the two-dimensional G-graph obtained in Section 4.5 as a linear array for problems with fixed-size data corresponds to a particular case of partitioning an algorithm for implementation in an array with $O(n)$ cells. This objective is accomplished by dividing the G-graph into sets of neighbor G-nodes organized in a linear arrangement, and executing such sets sequentially in the array (i.e., cut-and-pile, with “cuts” of size $O(n)$). This partitioning process is discussed in general in the following section.

4.7 Deriving arrays for partitioned problems from a CMMG

For problems with large and/or variable-size data (and for fixed-size data problems in linear structures), a G-graph is realized as an array with fewer cells (say K_p) than the number of G-nodes by applying the partitioning technique known as *cut-and-pile* [Nava87].⁶ This technique consists of two steps:

- divide the G-graph into sets of neighbor nodes (G-sets)
- execute the G-sets sequentially in an array

These steps are described next within the framework of our method.

4.7.1 The selection of G-sets

G-sets are sets with as many G-nodes as there are cells in an array. That is, an array with K_p cells requires G-sets with K_p G-nodes. Moreover, G-nodes in a G-set should have the following characteristics:

- Same computation time per G-node. Since G-nodes in a G-set are executed concurrently, identical computation time produces good cell utilization.
- Same dependency structure as communication links in the array. That is, linear arrays with K_p cells require that G-sets are linear sets of K_p G-nodes,

⁶The G-graph has been obtained from grouping prisms in the MMG, which corresponds to the application of coalescing to a limited extent. Consequently, the partitioning technique is actually a combination of coalescing and cut-and-pile.

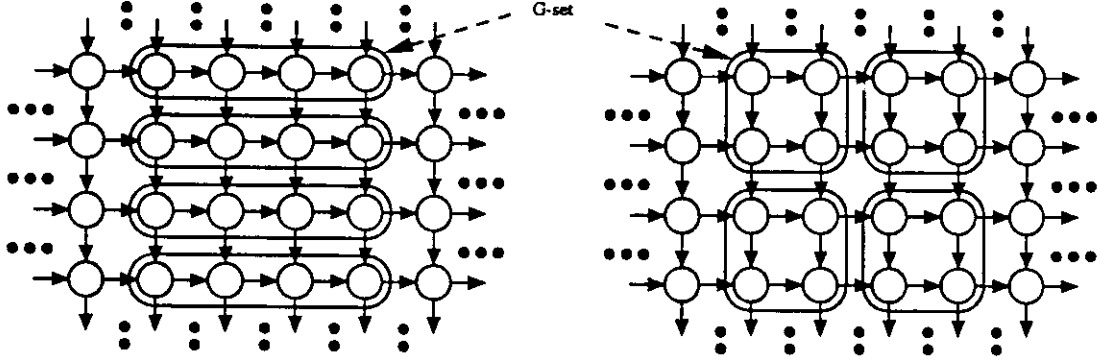


Figure 4.19: Dividing a G-graph into G-sets

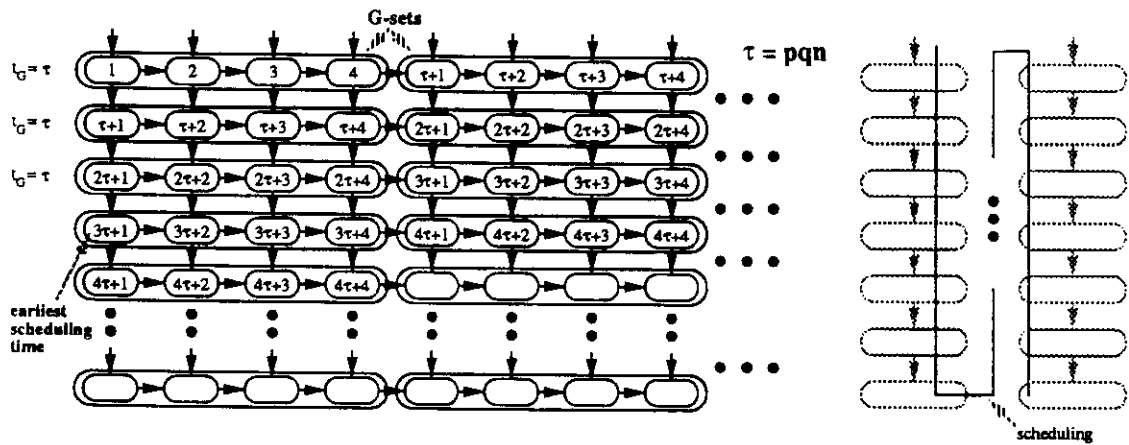
while two-dimensional arrays require two-dimensional sets of as many G-nodes as cells, as shown in Figure 4.19. Note that G-sets for a linear array may be composed of horizontal or vertical paths from the G-graph.

4.7.2 The scheduling of G-sets

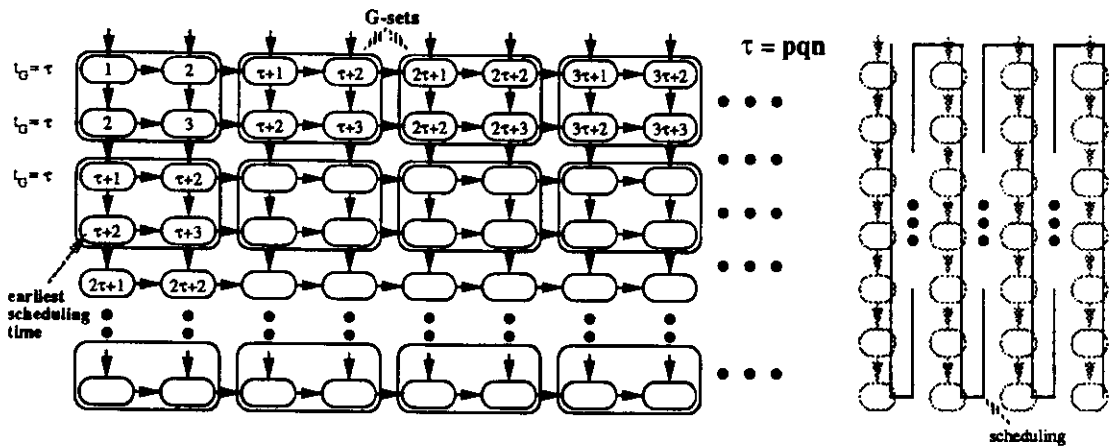
We discuss now the schedule of G-sets. To illustrate this scheduling, we use the G-graph for a CMMG shown in Figure 4.20, where all G-nodes have identical computation time $t_G = pqn$. Nodes in this figure have been tagged with their earliest possible scheduling time relative to a reference time.

The computation time of nodes in a G-set is $O(n)$, while the longest time required by a data element to traverse the array is $O(K_p)$, because there are K_p cells. Since $K_p \ll n$, *the data needed to begin executing the next G-set is available before the G-set in execution completes*. Consequently, scheduling needs to consider only the dependencies between G-sets and needs not worry about availability of intermediate data produced by previously scheduled G-sets.

Let us assume that the realization of the G-graph as a linear array is performed by composing G-sets with G-nodes in horizontal paths, as show in Figure 4.20a. Scheduling G-sets may be done by horizontal or vertical paths. For input bandwidth reasons discussed below, we choose to schedule G-sets by vertical paths, as also depicted in Figure 4.20a. G-sets are scheduled in pipelined mode, as depicted in Figure 4.21. Scheduling G-sets for execution in a two-dimensional array is similar to the linear array, as illustrated in Figure 4.20b and 4.21.



(a) - Scheduling G-graph into linear array



(b) - Scheduling G-graph into two-dimensional array

Figure 4.20: Scheduling a G-graph in linear and two-dimensional arrays

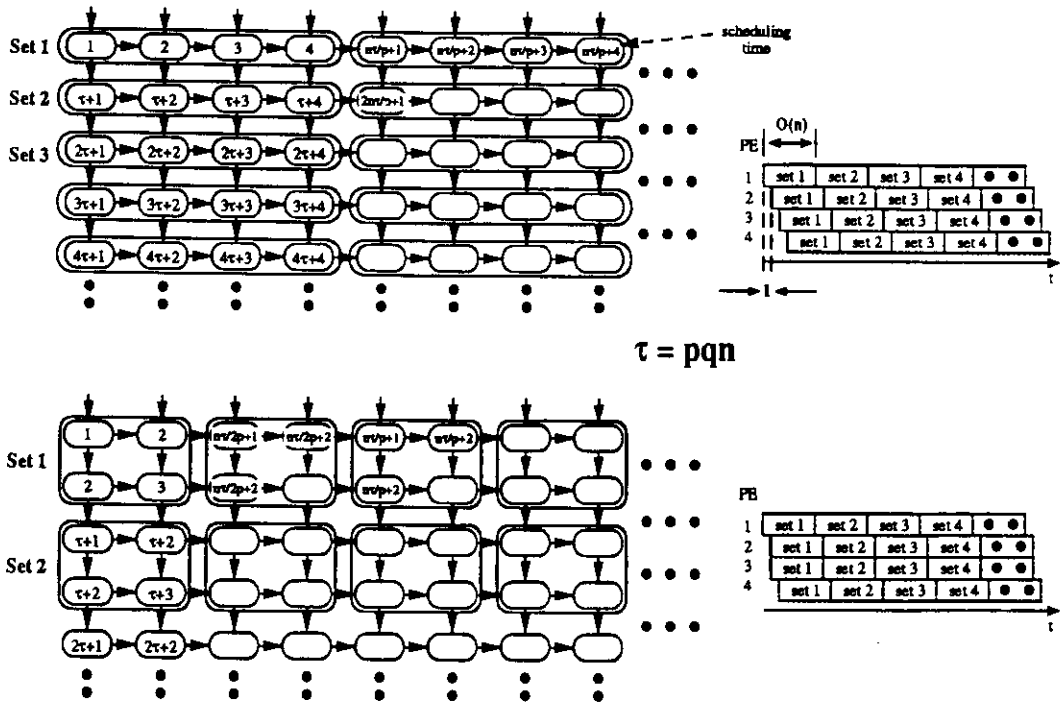


Figure 4.21: The schedule of G-sets

4.7.3 Array properties

Dividing a G-graph (obtained from a CMMG) into G-sets and scheduling the G-sets determines several characteristics of the resulting array, as described next.

Number of G-sets

The number of G-sets in a G-graph is

$$N_{GS} = n^2 / (pqK_p)$$

because $n^2 / (pq)$ G-nodes are divided into sets of size K_p .

Array throughput

Throughput of the array is given by

$$T_p = [t_G N_{GS}]^{-1} = \left[npq \frac{n^2}{pqK_p} \right]^{-1} = [n^3/K_p]^{-1}$$

because each G-set uses each cell for t_G time steps (t_G is the computation time of G-nodes).

Array utilization

Utilization of the array is computed as follows:

$$U_p = \frac{N}{K_p T_p^{-1}} = \frac{n^3}{K_p t_G N_{GS}} = \frac{n^3}{K_p (pqn)(n^2/pqK_p)} = 1$$

That is, the partitioned execution of a matrix algorithm represented by a CMMG leads to optimal utilization of cells in an array.

Input bandwidth

A host feeding input data to an array that executes a partitioned problem needs to provide only the data elements appearing as external input to G-nodes. Intermediate values are saved in, and retrieved from, *external* memories attached to the array, as those shown in Figure 4.22 (the M blocks). To reduce the peak rate at which data has to be provided from the host, the G-sets containing nodes that receive external inputs should not be scheduled consecutively. In such a case, the host needs to feed data to the array at a rate lower than one input per cell per time-step, and utilization of input connections may be increased by *decoupling computation from data transfers*.

The approach described above leads to the input structures shown in Figure 4.22, where the host feeds data to the array through a chain of registers (the R blocks in the figure). Each block R consists of a register and memory. Data from the host flows in pipelined mode through the registers and is stored in the memories. When a G-set from the top of the graph is scheduled for execution, data is read from memories in R into the PEs. New data is transferred from the host to the blocks R while cells use intermediate data from memories M.

Since each node at the top of the G-graph in Figure 4.21 receives qn data

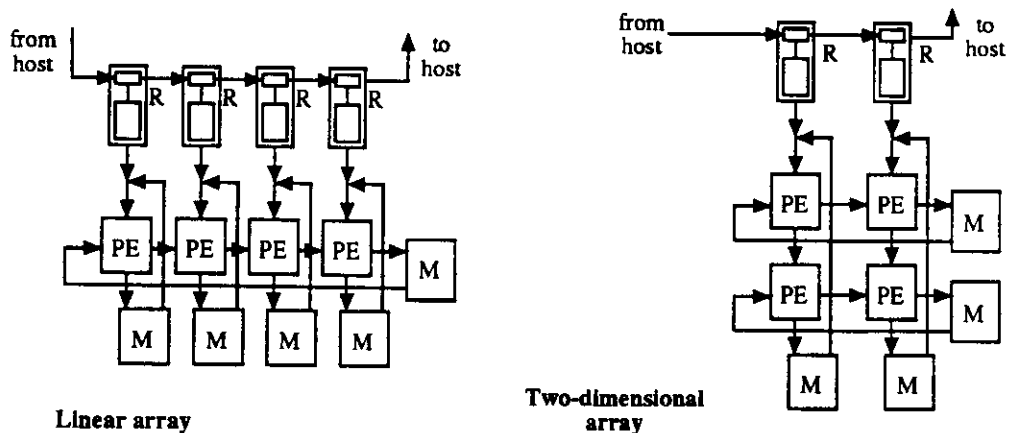


Figure 4.22: I/O bandwidth in partitioned implementations

elements from the host, input bandwidth of the array is given by

$$A_{IN} = \frac{(qn)K_p}{\sum_{j=1}^{n/p} t_G} = \frac{qnK_p}{(pqn)(n/p)} = \frac{K_p}{n}$$

Note that, under the conditions described above, *linear and two-dimensional arrays have the same input bandwidth from the host.*

Output bandwidth

If output from a G-graph appears only at the lowermost G-nodes, then output bandwidth follows the same pattern and may use the same structure indicated above for the input to an array.

In contrast, if results are obtained within the different G-sets, then those results appear distributed in time (i.e., every $O(n)$ time-steps, which corresponds to the computation time of G-nodes). Consequently, those results may also be transferred to the host through the same structure used for input, adding very little communication load (i.e., a rate of $1/O(n)$).

4.8 Using IMMGS

Matrix algorithms usually lead to incomplete multi-mesh dependency graphs instead of CMMGs. Examples are algorithms for LU-decomposition, inverse of

non-singular triangular matrix, triangularization by Givens rotations, Faddeev algorithm, among others.

The application of our method to an algorithm represented by an IMMIG follows the same procedure as in the case of CMMGs. However, there are two aspects related to the incompleteness of an MMG that complicate the derivation of G-graphs and influence the performance of the resulting arrays. These aspects, which are described below, are:

- Different number of nodes per prism, which influences utilization of cells.
- Flow of transmittent data orthogonal to direction of collapsing, which allows using pipelined cells.

Number of nodes per prism

Different number of nodes per prism implies different computation time of the corresponding G-nodes. Since G-nodes are realized in cells, this leads to cells with different computation time and, consequently, non-optimal cell utilization. This drawback does not appear in a CMMG, because in that case prisms of a given base size always have the same number of nodes, regardless of the grouping direction chosen.

The utilization of cells may be improved by *selecting prisms of varying base size*. In such a case, prisms that determine throughput are selected from the denser portion of an IMMIG. In contrast, the size of prisms' bases in the less dense part of the IMMIG is adjusted so that the total number of primitive nodes per prism is (roughly) equal to that in denser prisms. The price paid for the corresponding improvement in utilization is a more complex and non-uniform sequencing within cells than in the case with identical prisms bases.

Transmittent data and collapsing direction

The method presented in Section 4.5 assumes that the flow of transmittent data and the direction of collapsing an MMG are orthogonal. This is necessary to use pipelined cells in an array. In a CMMG, it is always possible to find such a combination of directions without restrictions, because prisms along any direction enclose the same number of primitive nodes. This fact is illustrated in Figure 4.23a.

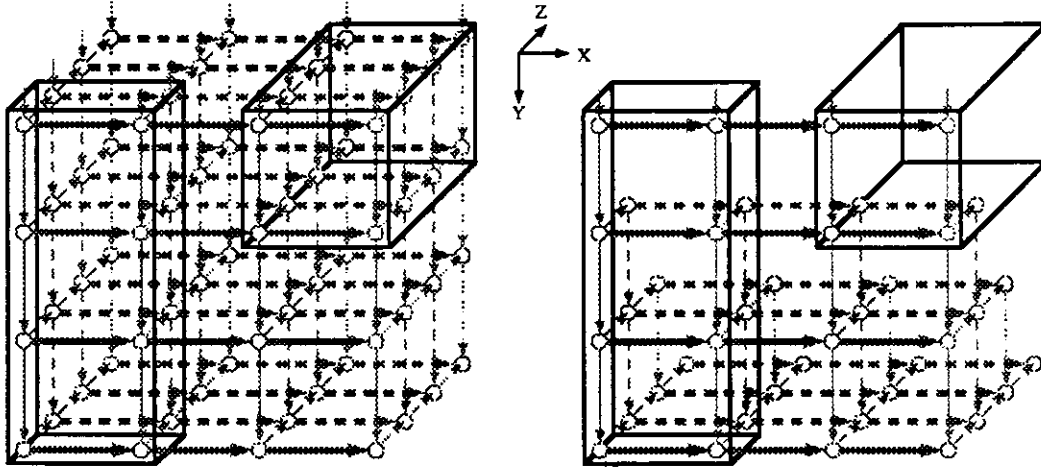


Figure 4.23: Drawing prisms in complete and incomplete MMGs

In contrast, a direction of collapsing orthogonal to the flow of transmittent data in an IMMIG may lead to prisms that enclose a varying number of primitive nodes, as it is inferred from Figure 4.23b. As indicated above, such groupings lead to G-nodes with varying computation time and potentially low utilization of cells in an array. Consequently, IMMIGs result in lower flexibility in selecting a direction of collapsing than CMMGs.

On the other hand, selecting a grouping direction parallel to the flow of transmittent data and scheduling primitive operations following that flow leads to a schedule by meshes of size p (or q) by n , requiring local storage of size $O(n)$ (according to the discussion in Section 4.5.3). Alternatively, one could schedule primitive operations by following the flow of transmittent data only up to p (or q) primitive operations, as depicted in Figure 4.24. The drawback of such an approach is the need for larger (but constant) local storage and more complex sequencing within cells than that obtained when the flow of transmittent data is orthogonal to the direction of collapsing.

Note that the aspects discussed here only apply when one wants to use pipelined cells in the array and selecting prisms with base size larger than 1 by 1 (so that cell bandwidth is lower than computation rate). As described earlier, for non-pipelined cells it is not necessary to schedule primitive operations following the flow of transmittent data; there is no relation between the direction of such a flow and the collapsing direction. Moreover, prisms of base size 1 by 1 have no local storage, so that cells may be pipelined but cell bandwidth is as large as computation rate.

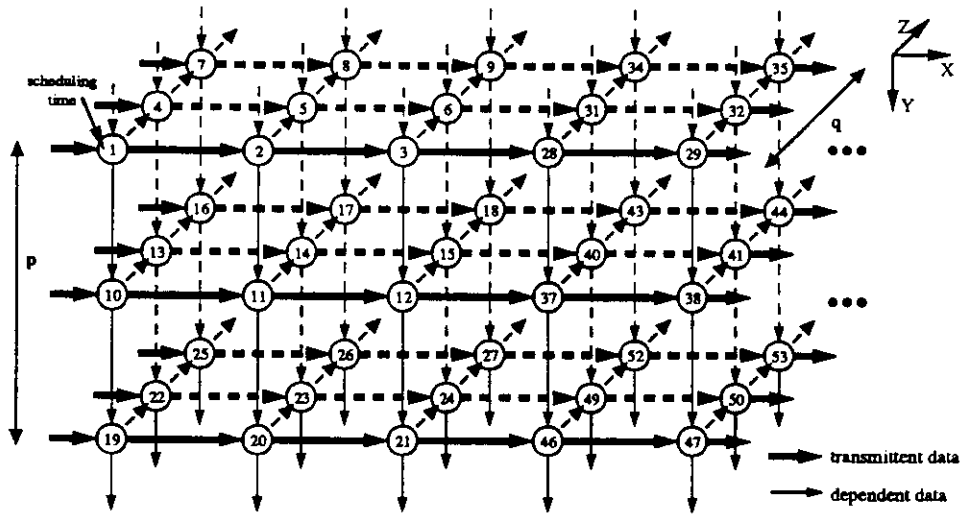


Figure 4.24: Schedule parallel to flow of transmittent data

4.9 Deriving arrays for fixed-size data from an IMMIG

We discuss now the derivation of arrays from IMMIGs. For this analysis, we assume that the IMMIG has a decreasing number of nodes per mesh along the Z -axis, as in the example shown in Figure 4.25a for a problem of size $n = 6, m = 4$. Moreover, we will perform groupings along all directions; in groupings along the X -axis, primitive nodes are scheduled as indicated in Figure 4.24. The same analysis presented here applies to graphs with other degrees of incompleteness.

The incompleteness of the MMG affects the performance of an array, as follows:

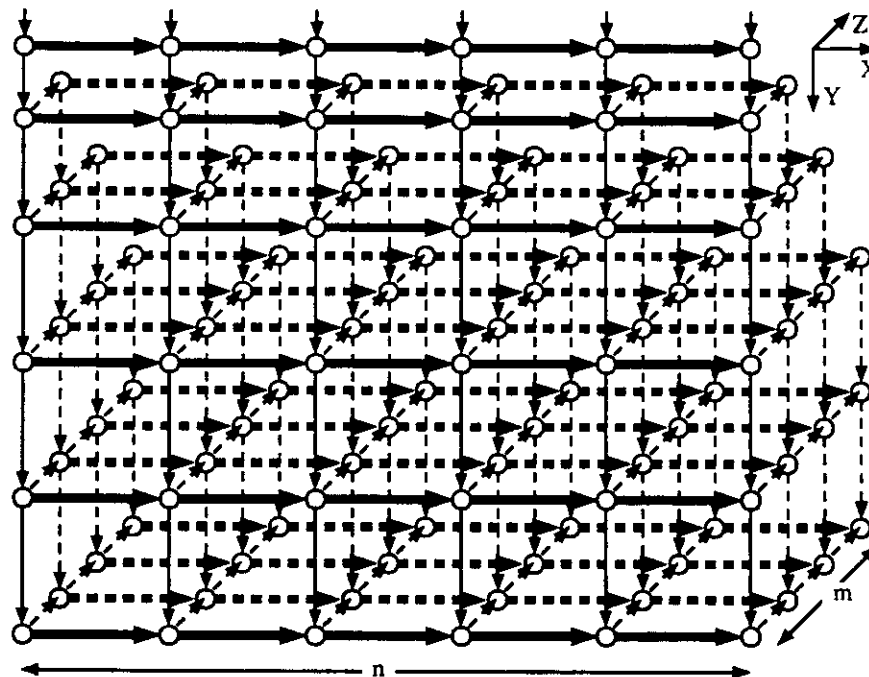
- Throughput is given by

$$T_f = [\max_i t_G^i]^{-1}$$

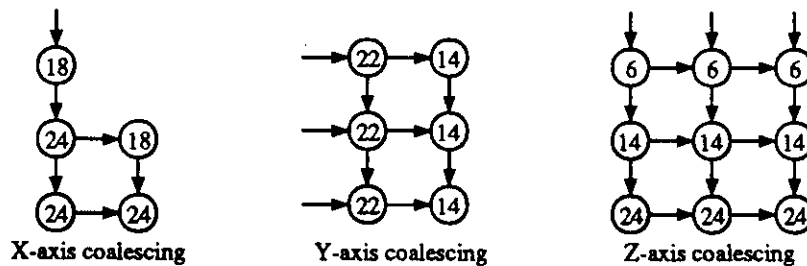
where t_G^i is the computation time of the i -th node in the G-graph. That is, throughput is determined by the G-node with longest computation time, or equivalently, by the prism with the largest number of primitive nodes.

- Number of cells (K_f) is equal to the number of G-nodes. Such a number depends on the direction of collapsing, and can be determined as

$$K_f = \sum_{i=1}^R N_G^i$$



(a) Incomplete multi-mesh dependency graph



(b) G-graphs

Figure 4.25: Deriving G-graphs from an IMM

where

$R =$ number of rows of G-nodes (i.e., $\lceil n/p \rceil$)
 $N_G^i =$ (maximum number of primitive nodes in a path of the MMG included in i -th row of G-nodes) divided by q

- Utilization is computed as

$$U_f = \frac{N}{K_f T_f^{-1}}$$

Figure 4.25b depicts the G-graphs derived from the IMM in Figure 4.25a assuming $p = q = 2$, where G-nodes have been tagged with their computation time. This figure shows that, for example, collapsing the IMM along the Z-axis leads to a rectangular G-graph where nodes in horizontal paths have the same computation time, but those in successive horizontal paths have increasing time. Similar results are obtained when coalescing along the Y-axis. In contrast, grouping along the X-axis leads to a trapezoidal G-graph where all G-nodes have the same computation time, excepting along the diagonal. Arrays obtained from realizing these G-graphs exhibit diverse cost and performance.

4.10 Deriving partitioned implementations from an IMM

We discuss now implementing arrays for partitioned problems represented by IMM. As stated in Section 4.7, realizing a G-graph as an array for partitioned execution implies selecting G-sets from the G-graph and scheduling them for sequential execution. Such G-sets consist of either linear or two-dimensional sets of G-nodes, depending on the structure of the array. The procedure to carry out this task is the same for both CMMs and IMM, but the tradeoffs in cost and performance are different in each case, as described next.

4.10.1 The selection of G-sets

To obtain optimal utilization of an array, *G-nodes in a G-set* should have the same computation time. This property is less stringent than for fixed-size problems, which require *all G-nodes in the entire G-graph* to have the same computation time. Consequently, partitioned implementations of an IMM have potentially higher utilization than arrays for fixed-size data.

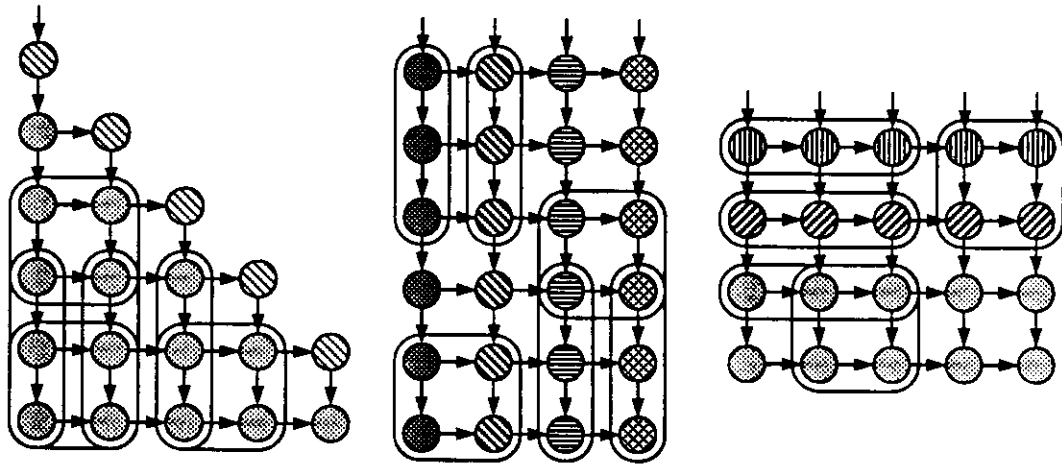


Figure 4.26: G-graphs derived from an IMM

To illustrate the issues involved, let us consider a large IMM that has the same structure as the one shown in Figure 4.25. In such an IMM, grouping leads to G-graphs as those shown in Figure 4.26. As indicated earlier, these are rectangular G-graphs where G-nodes in the same horizontal (or vertical) path have the same computation time, or a trapezoidal G-graph where all nodes have identical time (excepting at the diagonal). Consequently, the partitioned implementation in a linear array may choose among any of the three alternatives, without affecting performance of the resulting array. In contrast, selecting two-dimensional G-sets from the rectangular G-graphs will include G-nodes with different computation time, and utilization of the resulting two-dimensional array will not be optimal. Consequently, the G-graph obtained by grouping along the X -axis produces higher utilization in a two-dimensional array than the other two groupings.

Scheduling the G-sets obtained above is performed in the same way as in the case of CMMGs.

4.10.2 Array properties

As inferred from the discussion above, performance of an array that implements a partitioned problem depends on the amount of incompleteness in the IMM and on the collapsing direction. The performance measures are:

Array throughput

Throughput for partitioned problems is obtained as

$$T_p = \left[\sum_{j=1}^{N_{GS}} t_{max}^j \right]^{-1}$$

where t_{max}^j is the longest computation time of a G-node in the j -th G-set, and N_{GS} is the number of G-sets.

Array utilization

Utilization is given by

$$U_p = \frac{N}{K_p T_p^{-1}}$$

I/O bandwidth

Assuming the schedule of G-sets shown in Figure 4.21 and the I/O structure given in Figure 4.22, I/O bandwidth is given by

$$A_{I/O} = \frac{\max_i \left\{ \sum_{j=1}^{K_p} F_j^i \right\}}{\sum_{j=1}^{n/p} t_G^j}$$

where F_j^i is the maximum number of edges that arrive from outside the graph to the j -th G-node in the i -th G-set.

4.11 Using MMGs with two flows of input data

Throughout this chapter we have considered that MMGs have only one flow of input data. As indicated earlier, this is the case of many important matrix algorithms. However, there are some algorithms that have two flows of input data, such as matrix multiplication. In this section, we discuss how the second input flow is handled in our method. For such purposes, we use the MMG with external

inputs from left and top shown in Figure 4.27a. In this case, collapsing the MMG along the direction that has no input flow leads to a G-graph that has external inputs only at boundary nodes, as shown in Figure 4.27b for grouping along the Z -axis (prisms of base size 1 by 1). In contrast, collapsing the MMG along a direction that has input flow leads to a G-graph that has external inputs at one side of the graph and at every node throughout the graph, as illustrated in Figure 4.27c for grouping along the Y axis.

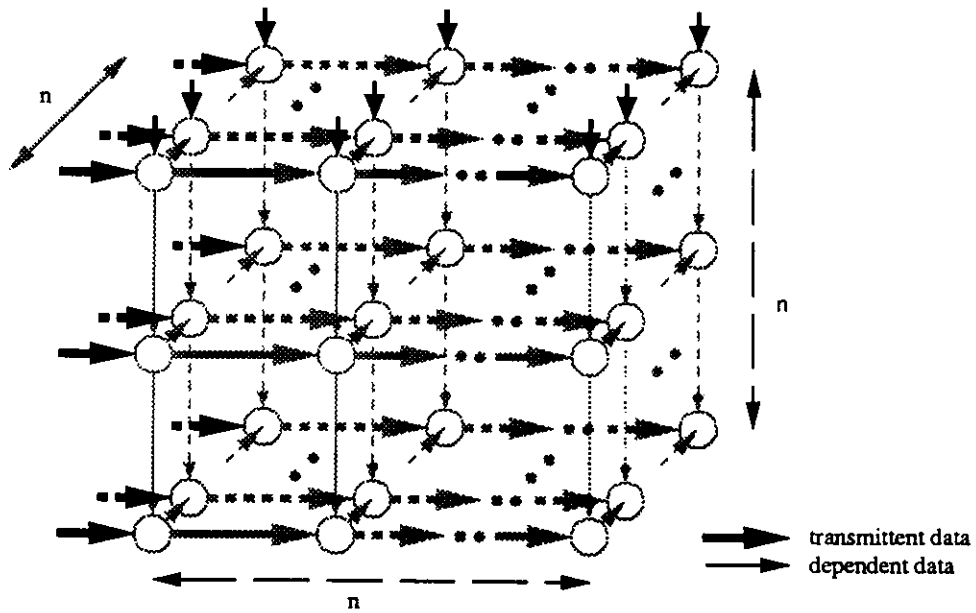
We now discuss the effects that the two flows of input data have on arrays for fixed-size and partitioned problems. We focus on two-dimensional arrays, since the problem and solution in a linear array correspond to that of a two-dimensional array with size K by 1.

4.11.1 Two-dimensional arrays for fixed-size data

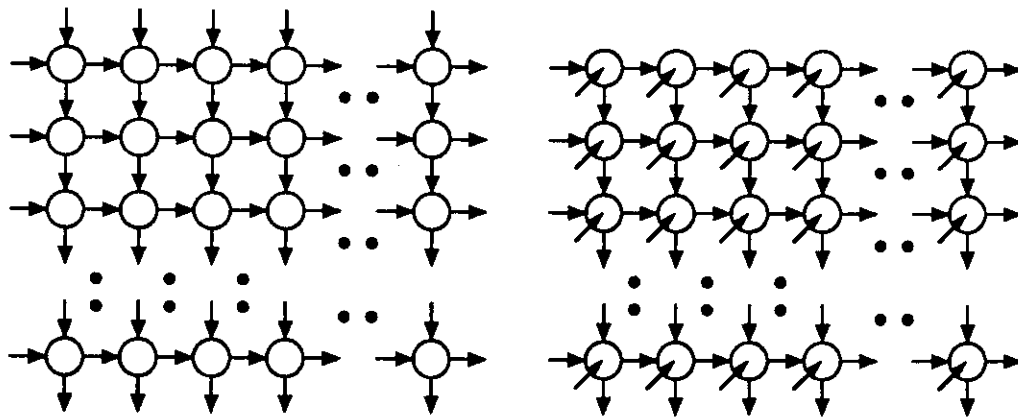
Let us consider first the realization of two-dimensional arrays for fixed-size data. Since the G-graph shown in Figure 4.27b has external input only at the boundaries, it is directly realized as a two-dimensional array without any problems. In contrast, since the arrays considered in this dissertation have external I/O only at boundary cells, direct realization of the G-graph shown in Figure 4.27c is not feasible unless pre-loading data into cells is allowed, because each G-node receives an external input. Pre-loading is achieved by transferring data through cells until it reaches its destination. Although this is a feasible solution that has been suggested/implemented in some cases, it has the disadvantage that cells are used for loading/unloading data without performing useful computations, so that array utilization and throughput decrease.

However, pre-loading data may be avoided. The computation time of G-nodes in Figure 4.27c is $t_G = n$, but each G-node receives only one external input along the Z -axis during the entire evaluation of the G-node (i.e., n time-steps). Consequently, the communication links associated to that data flow are under-utilized, because data are transferred only in one out of n time-steps. This property allows decoupling computation from data transfers, in a similar manner to the case of partitioned implementations discussed in Section 4.7. That is, data for one instance of an algorithm are transferred through a separate chain of registers, while a previous instance is computed.

The approach described above is depicted in Figure 4.28. The cells of this array have been augmented with a register; all those registers are connected in a chain,



(a) Multi-mesh graph with two input flows



(b) G-graph by grouping along Z

(c) G-graph by grouping along Y

Figure 4.27: A CMMG and G-graphs with two flows of input data

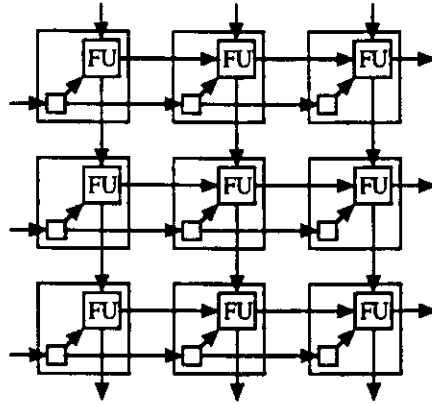


Figure 4.28: Array for G-graphs with two input flows

as shown in the figure. In such a case, input data arrives to the boundary of the array and flows through cells until it reaches its destination.

4.11.2 Two-dimensional arrays for partitioned problems

For partitioned problems, realizing the G-graph in Figure 4.27b as an array leads to large I/O bandwidth. In the case of one input flow, array I/O was decreased by not scheduling successively the G-sets that receive external inputs, as illustrated in Figure 4.21. That is, the execution of two G-sets with external inputs was spread out in time, allowing lower data transfer rate. However, this approach is not effective in the case of a G-graph as the one in Figure 4.27b. Executing G-sets of such a graph either vertically or horizontally implies the successive schedule of G-sets that have external inputs, so that the array must have the capability to provide the required bandwidth (i.e., one data element per time step). In other words, it is not possible to spread out in time the data transfers associated to G-sets with external inputs, unless a more complex schedule is implemented. An example of such type of execution order is illustrated in Figure 4.29, where G-sets are scheduled in diagonal order. Note that G-sets with external inputs are still executed successively at the beginning, while they spread out in time later. As a result, data for the first few G-sets has to be transferred in advance and stored in memories external to the array, so that it can be delivered at the required rate. Once the I/O bandwidth required by the execution of G-sets decreases to a value that is handled by the array/host connection, then the remaining inputs are transferred from host to array and stored in memories attached to inputs, as described in Section 4.7.

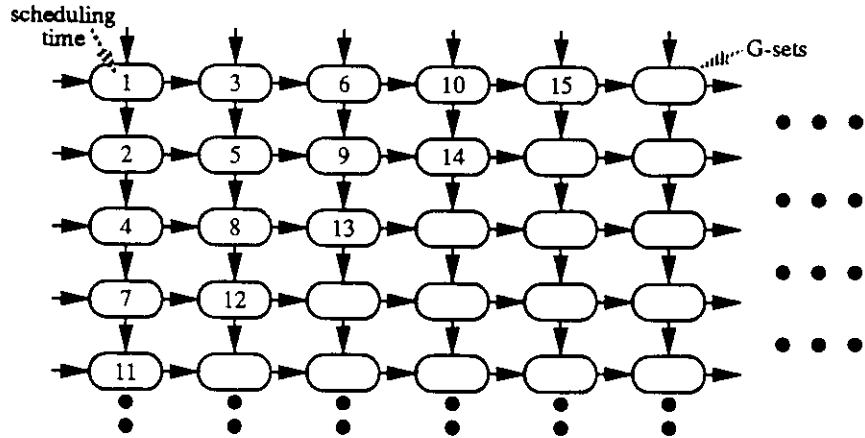


Figure 4.29: Scheduling G-sets with two input flows

In contrast to the approach outlined above, the G-graph with two flows of input data shown in Figure 4.27c is more amenable to partitioned execution. In such a case, the flow arriving to every G-node along the Z - axis has low bandwidth (as discussed earlier) and it can be implemented using the same technique presented for problems with fixed-size data, namely by transferring data in advance and augmenting cells with a path for such transfers. Moreover, in the case of a linear array, the same chain of R blocks shown in Figure 4.22 may be used for this purpose.

4.12 Summary of performance measures of arrays

In this section, we review the performance measures of arrays designed with our graph-based method. Since this method allows studying the impact of transformations on performance while carrying-out such transformations, and the corresponding issues were described together with the respective transformations in the previous sections, we just summarize the results at this time. Moreover, the terms used in this summary have been defined throughout this chapter.

4.12.1 Fixed-size data

For problems with fixed-size data, the values for performance and cost measures are (where N is the number of operations in an algorithm)

$$\text{Number of cells } K_f = N_G$$

$$\begin{aligned}
\text{Throughput } T_f &= [\max_i t_G^i]^{-1} \\
\text{Utilization } U_f &= \frac{N}{K_f T_f^{-1}} \\
\text{Storage per cell } C_w &= q(p+1) \\
\text{Cell bandwidth } C_{BW}^Y &= 1/p \\
&C_{BW}^X = 1/q \\
\text{Array I/O bandwidth } A_{I/O} &= K_B C_{BW}^B
\end{aligned}$$

where C_{BW}^B is the corresponding bandwidth of boundary cells (i.e., C_{BW}^X or C_{BW}^Y), and K_B is the number of such boundary cells.

4.12.2 Partitioned problems

The performance and cost measures for partitioned implementations are

$$\begin{aligned}
\text{Number of cells } K_p & \\
\text{Throughput } T_p &= \left[\sum_{j=1}^{N_{Gs}} t_{max}^j \right]^{-1} \\
\text{Utilization } U_p &= \frac{N}{K_p T_p^{-1}} \\
\text{Storage per cell } C_w &= q(p+1) \\
\text{Cell bandwidth } C_{BW}^Y &= 1/p \\
&C_{BW}^X = 1/q \\
\text{Array I/O bandwidth } A_{I/O} &= \frac{\max_i \left\{ \sum_{j=1}^{K_p} F_j^i \right\}}{\sum_{j=1}^{n/p} t_G^j}
\end{aligned}$$

In the expressions above, t_{max}^i is the longest computation time of a node in the i -th G-set mapped onto the array, and F_j^i is the maximum number of data elements that are brought from outside the graph into the j -th G-node of the i -th G-set (i.e., edges arriving into the graph).

4.13 Arrays for the triangularization algorithm

We illustrate now the derivation of arrays from an IMMIG by obtaining and evaluating implementations for the triangularization algorithm.

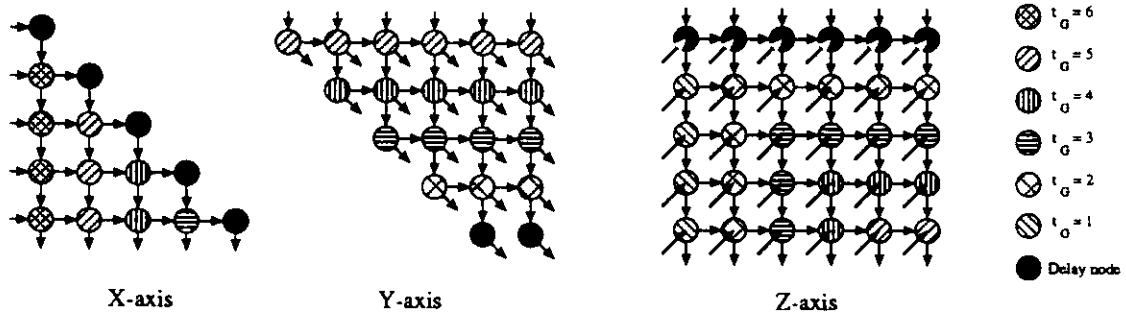


Figure 4.30: G-graphs for the triangularization algorithm

4.13.1 Computational load

The computational load imposed by the triangularization algorithm is obtained from the multi-mesh graph shown in Figure 4.11. This graph is composed of n dependent meshes. The number of operations can be determined by traversing those meshes from the innermost to outermost along axis Z . This number is

$$N = \sum_{i=1}^n (i-1)(i+1) = \frac{n}{6} [2n^2 + 3n - 5]$$

because there are $(i-1)(i+1)$ operations in mesh i . Note that the expression above does not include delay nodes in the graph, but only operation nodes.

4.13.2 Problems with fixed-size data

Let us look first into arrays for fixed-size data. Figure 4.30 depicts the results obtained from grouping prisms of base size 1 by 1 in the MMG shown in Figure 4.11. These G-graphs are directly realized as systolic arrays. The resulting implementations are evaluated below, for the general case of grouping by prisms of base size p by q .

4.13.2.1 Grouping along the X-axis

Grouping along the X -axis leads to a triangular array. The number of cells in such an array is (assuming that n/q , q/p are integers)

$$K_f^X = \sum_{i=1}^{\lceil n/q \rceil} \left\lceil \frac{iq}{p} \right\rceil = \sum_{i=1}^{n/q} \frac{iq}{p} = \frac{n(n+q)}{2pq}$$

The throughput of this array is determined by the bottom leftmost cell, because this cell computes more primitive operations than other cells, as it is inferred from the MMG. Assuming that $(n-p) \geq n/2$, $(n-q) \geq n/2$, this throughput is given by

$$T_X^{-1} = t_X^{\max} = pq(n+1) - \sum_{i=1}^q p(i-1) = pq(2n-q+3)/2$$

Utilization is

$$\begin{aligned} U_X &= \frac{N}{K_f^X T_X^{-1}} = \frac{N}{K_f^X t_X^{\max}} \\ &= \frac{n(2n^2 + 3n - 5)/6}{(n/2pq)(n+q)(pq/2)(2n-q+3)} \\ &= \frac{2(2n^2 + 3n - 5)}{3(n+q)(2n-q+3)} \end{aligned}$$

For large n , we obtain

$$U_X \rightarrow \frac{4n^2}{6n^2} = \frac{2}{3}$$

4.13.2.2 Grouping along Y-axis

The number of G-nodes (and consequently the number of cells) obtained by grouping along the Y-axis is computed as (assuming that p/q and n/p are integers)

$$K_f^Y = \sum_{i=1}^{\lceil n/q \rceil} \left\lceil \frac{iq+1}{p} \right\rceil = \sum_{i=1}^{n/q} \frac{iq+1}{p} = \frac{n}{2pq}(n+q+2)$$

The throughput of the resulting array is determined by the computation time of the top-rightmost cell. Assuming that $(n-p) \geq n/2$, $(n-q) \geq n/2$, this throughput is

$$T_Y^{-1} = t_Y^{\max} = pqn - \sum_{i=1}^p q(i-1) = \frac{pq}{2}(2n-q+1)$$

Utilization is

$$\begin{aligned}
U_Y &= \frac{N}{K_f^Y T_Y^{-1}} = \frac{N}{K_f^Y t_Y^{max}} \\
&= \frac{n(2n^2 + 3n - 5)/6}{(n/2pq)(n + q + 2)(pq/2)(2n - q + 1)} \\
&= \frac{2(2n^2 + 3n - 5)}{3(n + q + 2)(2n - q + 1)}
\end{aligned}$$

For large n , this results in

$$U_Y \rightarrow \frac{4n^2}{6n^2} = \frac{2}{3}$$

4.13.2.3 Grouping along the Z-axis

Grouping along the Z-axis leads to a rectangular array. The number of cells in such an array is

$$K_f^Z = \left\lceil \frac{n}{p} \right\rceil \left\lceil \frac{n+1}{q} \right\rceil \approx \frac{n^2}{pq}$$

The throughput of this array is determined by the bottom rightmost cell. Such a cell computes more primitive operations than other cells, as it is inferred from the MMG. Throughput is given by

$$\begin{aligned}
T_Z^{-1} = t_Z^{max} &= \sum_{i=1}^p \sum_{j=1}^q (\text{nodes along } Z) \\
&= [(n - q + 2) + (n - q + 3) + \dots + (n - 1) + 2n] \\
&\quad + [(n - q + 2) + (n - q + 3) + \dots + 3(n - 1)] \\
&\quad + [(n - q + 2) + (n - q + 3) + \dots + 4(n - 2)] \\
&\quad \vdots \\
&= \sum_{i=1}^p \left[\sum_{j=n-q+2}^{n+1} j - \sum_{j=1}^i j \right] \\
&= \sum_{i=1}^p \left[\frac{(n+1)(n+2)}{2} - \frac{(n-q+1)(n-q+2)}{2} - \frac{i(i+1)}{2} \right] \\
&= \frac{pq}{2} [2n - q + 3] - \frac{p}{6} [p^2 - 1]
\end{aligned}$$

Utilization is

$$\begin{aligned}
U_Z &= \frac{N}{K_f^Z T_Z^{-1}} = \frac{N}{K_f^Z t_Z^{\max}} \\
&= \frac{\frac{n}{6}(2n^2 + 3n - 5)}{\frac{n(n+1)}{pq} \left[\frac{pq}{2}(2n + 3 - q) - \frac{p}{6}(p^2 - 1) \right]} \\
&= \frac{pq(2n^2 + 3n - 5)}{6(n + 1) \left[\frac{pq}{2}(2n + 3 - q) - p(p^2 - 1) \right]}
\end{aligned}$$

For large n

$$U_Z \rightarrow \frac{2n^2 pq}{6n^2 pq} = \frac{1}{3}$$

4.13.2.4 Comments on arrays for fixed-size data

The expressions in the previous subsections indicate that collapsing the multi-mesh graph along the different axes leads to arrays with diverse cost and performance measures. For example, grouping the MMG along the X -axis leads to a triangular systolic array (with delay registers in the diagonal) that triangularizes a matrix of size 5 by 5 with utilization $U = 0.83$ and throughput $T = 0.17$. In contrast, collapsing the G-graph along the Y -axis leads to $T = 0.2$ in a triangular systolic array with more cells than the previous one, whose utilization is $U = 0.69$. Consequently, the selection of the most suitable architecture is determined by the relative weight of the different performance and cost measures.

4.13.3 Partitioned implementations

For partitioned implementations, we select the grouping direction that produces G-sets with nodes of identical computation time. To achieve this objective, we may select prisms along the X -axis or the Y -axis in the multi-mesh graph (grouping along the Z -axis is not a good choice, as inferred from Figure 4.30). Figure 4.31 illustrates coalescing by prisms of cross-section size 2 by 2 along the Y -axis. The resulting G-graph is shown in Figure 4.32a (this G-graph corresponds to a multi-mesh graph larger than the one shown in Figure 4.31; due to space constraints, the larger multi-mesh graph has not been shown).

The G-graph is now decomposed into G-sets, and G-sets are scheduled for execution in the array. G-sets consist of linear or two-dimensional sets of G-nodes,

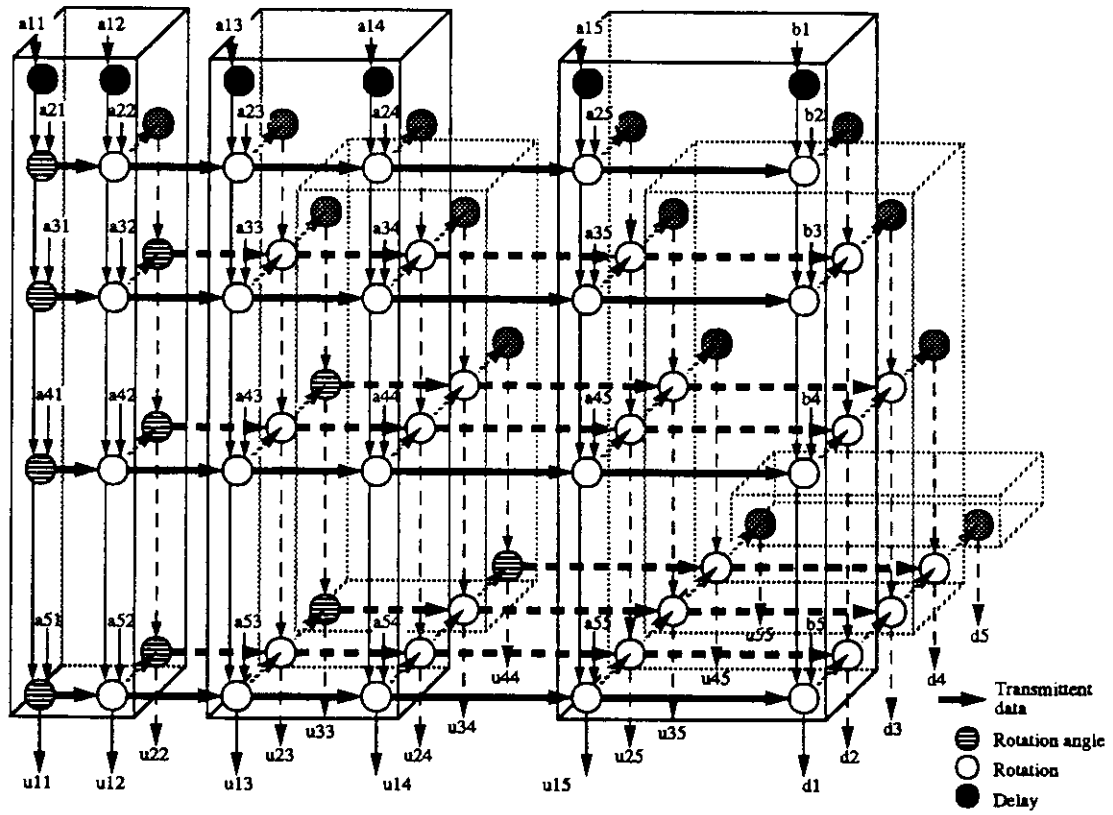


Figure 4.31: Prisms for partitioned problems in the triangularization algorithm

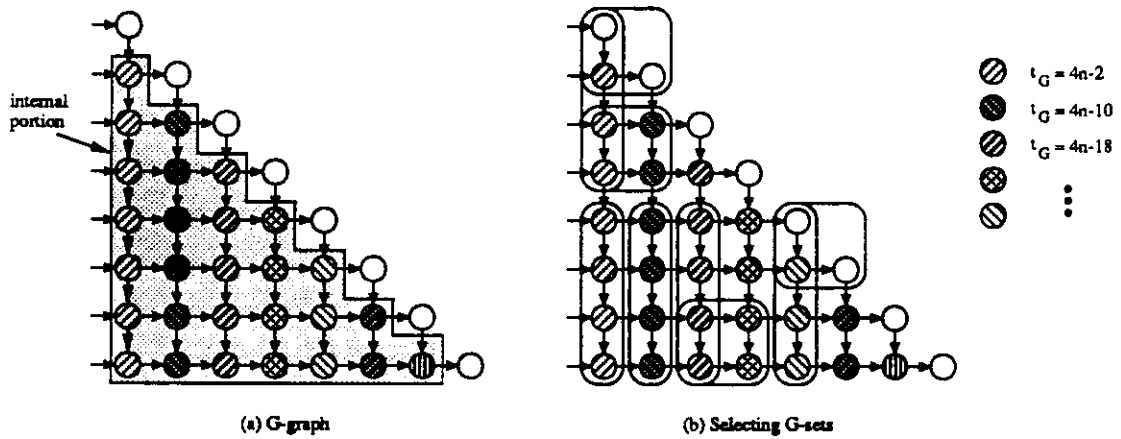


Figure 4.32: G-sets in the triangularization algorithm

as shown in Figure 4.32b. Notice that two-dimensional G-sets include G-nodes with different computation time, so that utilization of the resulting array will not be optimal. On the other hand, it is possible to select linear G-sets where all nodes have the same computation time, leading to better utilization of the corresponding linear array. Such linear G-sets are vertical paths of the G-graph, as shown in Figure 4.32b.

Scheduling the G-sets derived above is done by considering the array I/O bandwidth. Since input data appears only at leftmost G-nodes, we schedule G-sets in horizontal order. That is, we first schedule the leftmost G-set and then all G-sets to the right of that one. Upon reaching the right end of the G-graph, the next G-set at the left of the graph is scheduled, and the process continues in the same manner. G-sets are piled for execution in an array such as the one shown in Figure 4.22.

The evaluation of the partitioned implementation is not included here. An example of such an evaluation process and the issues involved in it are given in Appendix A.

4.14 Type of array as a function of the design parameters

The method described in this chapter allows determining the type of array based on the size of the prisms, that is, based on values of p and q . The following table gives some examples:

Array	Systolic	Pseudo-systolic	Local-access (two-dimensional)
Values of p, q	$p = q = 1$	$p = q > 1$	$p = q = n/\sqrt{K}$
No. of G-sets	n^2/K	$n^2/(pqK)$	1
Storage per cell	2	$p(p+1)$	$n^2/K + n/\sqrt{K}$
Cell comm. bandwidth	1	$1/p$	\sqrt{K}/n

These expressions led to the values in Table 2.1, where $p = q = \sqrt{S}$ in the case of pseudo-systolic arrays. As indicated earlier, the two words of storage per cell in a systolic array corresponds to registers required to latch input operands, so that actually there is no storage per cell in that implementation. Note that adding local storage to a cell reduces cell bandwidth proportionally to the inverse square-root of the size of such a local storage.

4.15 Tradeoffs between linear and two-dimensional arrays

In this chapter, we have discussed tradeoffs that arise when designing an array for a matrix algorithm. In particular, we have analyzed the relationship between local storage in a cell and cell communication bandwidth, and we have also shown that it is not always possible to achieve maximal utilization in a two-dimensional array for partitioned execution of an IMMIG.

Consequently, given a number of cells, an important issue is to determine the most suitable structure to interconnect those cells in an algorithm-specific array. Since we consider linear and two-dimensional mesh arrays, this question amounts to analyzing tradeoffs between linear and two-dimensional structures.

For a problem with fixed-size data, it is necessary to go from a linear array to a two-dimensional one when all parallelism available in one dimension has been exhausted. That is, for problems of size $O(n)$, an array with more than $O(n)$ cells must be a two-dimensional mesh.

On the other hand, for partitioned implementations the number of cells is much smaller than the size of problems, so that deciding between linear or two-dimensional structures is not so simple. Since the number of cells is fixed (say K_p), *a two-dimensional architecture does not exploit more parallelism than a linear one with the same number of cells*. Earlier, we have shown that partitioned implementations in linear and two-dimensional arrays have the same I/O bandwidth to/from a host. Moreover, a linear structure has potentially higher utilization than its two-dimensional counterpart (it is easier to find linear G-sets whose nodes have the same computation time), and is simpler to build.

In addition, a linear configuration is more suited to incorporate fault-tolerance features, because it can include the ability of bypassing a faulty cell. This is in contrast to two-dimensional arrays, in which the mesh has to be reconfigured around a faulty cell, either by software or hardware mechanisms. Moreover, the linear structure loses only the faulty cell, while the two-dimensional array may lose one row and one column of cells after reconfiguration.

The only drawback of a linear array is that it requires $(K_p + 1)$ external memory modules, while a two-dimensional structure needs only $2\sqrt{K_p}$. Given that memory systems are less expensive and simpler than the remaining modules in an array, we believe that such a drawback is minor.

Consequently, from this analysis it is possible to conclude that *a linear array*

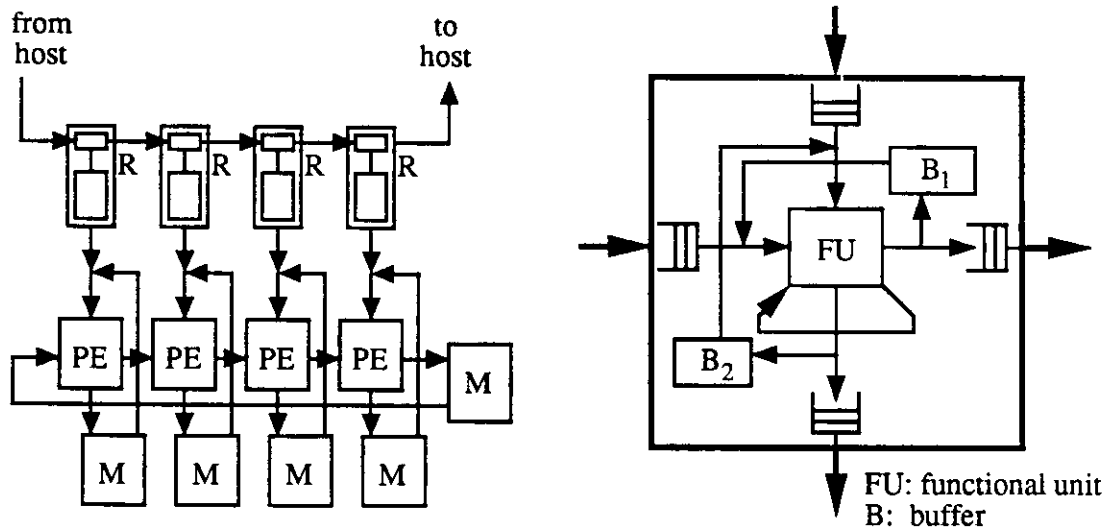


Figure 4.33: The canonical linear array for partitioned problems

is more suited for partitioned implementations than a two-dimensional structure.

The conclusions above are valid for large problems, that is, when the size of a problem is much larger than the number of cells in an array. In such a case, it is possible to derive a two-dimensional G-graph that will be “piled” onto an array in several G-sets. On the other hand, if the G-graph is not large enough, then the amount of parallelism available in such a G-graph (i.e., the number of G-nodes along one dimension of the graph) cannot be exploited successfully with many cells. Specifically, the conclusions stated above are valid if the G-graph has at least K_p by $\sqrt{K_p}$ G-nodes, because in this case the G-graph can be divided into linear or two-dimensional G-sets with K_p G-nodes.

4.16 A canonical linear array for partitioned problems

In Section 4.7, we introduced arrays that are well suited for the application of the partitioning approach used in our method. Although such arrays were presented in the context of algorithms described by CMMGs, they are equally applicable to algorithms described by IMMGS. Since in the previous section we have shown that a linear array for partitioned problems has advantages over a two-dimensional structure, the linear architecture shown in Figure 4.33 represents a good target implementation for the partitioned execution of matrix algorithms.

The array in Figure 4.33 constitutes a canonical structure. Its overall architecture is determined by the design method presented in this chapter, but the characteristics of cells are determined by a specific algorithm and associated implementation constraints (such as operations per cell, type of arithmetic, and numeric precision). This architecture has the following characteristics:

1. Linear collection of cells.
2. Support for external I/O into the array.
3. Memory modules external to cells.
4. Pipelined cells.
5. Small FIFO buffers in each cell.
6. Small queues attached to ports.
7. Cell bandwidth lower than computation rate.

An example of this canonical array is described in [More89a], where cells have a functional unit composed of a conventional floating-point multiplier and an ALU. The realization of algorithms in such an array produces high utilization of resources and uses the pipeline effectively. For example, performance estimates for the LU-decomposition algorithm indicate that a 200 by 200 matrix is processed in an array with 10 cells and 4-stage pipelines with utilization on the order of 90%. Consequently, with a clock cycle of 50 [nsec] the array delivers 360 [Mflops] out of a peak capacity of 400 [Mflops].

CHAPTER 5

The formalization of the design method

In this chapter, we present a formalization of the graph-based design method described in Chapter 4. Several terms used in this discussion are introduced first, including a canonical representation of matrix algorithms, and then we provide proofs for the equivalence of graphs derived through the transformations that compose the method. The process follows a bottom-up approach, by first discussing mapping a mesh dependency-graph onto an array, and later building the formalism up to the fully-parallel data-dependency graph.

5.1 Definitions

For the formalization of the method, we first introduce several terms that are adaptations of known concepts. Many of these definitions have already been used in preceding chapters.

Instances of an algorithm

A *single-instance algorithm* is an algorithm that is executed for a single set of input data. In contrast, a *multiple-instance algorithm* is executed for multiple sets of input data. Consequently, an *instance* of a multiple-instance algorithm consists of executing the algorithm for one set of input data.

Pipelined execution of multiple-instance algorithm

Pipelined execution of a multiple-instance algorithm is the overlapped execution of successive instances of the algorithm. That is, execution of a new instance is started before execution of previous instance(s) is (are) completed. Moreover, *perfect pipelined execution of a multiple-instance algorithm* corresponds to pipelined execution with no delay (idle time) between the execution of successive instances.

5.1.1 The canonical representation of matrix algorithms

In this dissertation, we center our attention on *matrix algorithms that are described recursively by the following canonical representation*:

Matrix algorithm is a tuple

$$A = (i, S, V, M, A)$$

where

- i is the index of a *for-loop* whose range is data-independent but may be dependent on other indices (i.e., **For** $i = j + 1$ **to** $n - k$, where i, j, k are loop indices and n is a matrix dimension).
- S is a finite set of *scalar operators*, such that¹ $|S|$ is independent from the dimensions of matrices (i.e., a small and constant number).
- V is a finite set of *vector operators*.
- M is a finite set of *matrix operators*.
- A is a finite set of *matrix algorithms*.

Figure 5.1 depicts such a recursive description. Consequently, we focus on algorithms that are represented in terms of a loop-index and a loop-body composed of scalar, vector and matrix operators, and recursively in terms of other matrix algorithms. Moreover, a sequence of matrix algorithms as those above is also a matrix algorithm.

Characteristics of operators in a matrix algorithm are as follows:

Scalar (primitive) operator is a tuple $s = (I_s, O_s, f_s)$ where

- I_s is a set of scalar operands, such that $|I_s| \in \{1, 2, 3\}$.
- O_s is a set of scalar outputs, such that $|O_s| \in \{1, 2, 3\}$.
- f_s is a primitive operation performed using all elements of I_s to produce all elements of O_s . The computation time of this operation is data independent (such as add, multiply, rotation, sin).

In practice, scalar operators have $|O_s| = 1$, excepting cases such as rotation of a pair of elements which produces $|O_s| = 2$.

¹ $|A|$ denotes the cardinality of set A

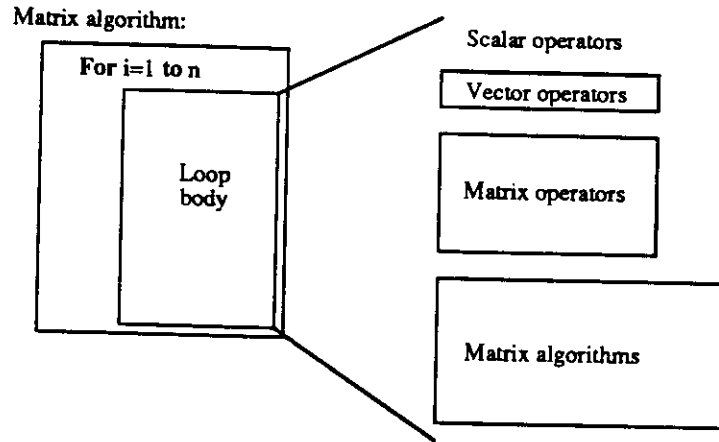


Figure 5.1: The canonical form of a matrix algorithm

Vector operator is a tuple $v = (I_v, O_v, B, F_v)$ where

- I_v is a set of vector inputs, such that $|I_v| \in \{1, 2\}$.
- O_v is a set of vector outputs, such that $|O_v| \in \{1, 2\}$.
- All vectors have the same length l_v (i.e., the same number of elements).
- B is a set of scalar inputs, such that $|B| \in \{0, 1\}$.
- F_v is a set of l_v scalar operators that perform the same primitive operation f_v . The i -th element of F_v , say scalar operator f_v^i , is such that

$$f_v^i = (\{I_v(i) \cup B\}, O_v(i), f_s)$$

where $I_v(i)$ and $O_v(i)$ are the i -th element of each vector in I_v and O_v , respectively.

Matrix operator is a tuple $m = (I_m, O_m, V_m, F_m)$ where

- I_m is a set of a single matrix input (i.e., $|I_m| = 1$).
- O_m is a set of a single matrix output (i.e., $|O_m| = 1$).
- Both input and output matrices have the same dimensions (m, n) .
- $V_m = \{v_h, v_v\}$ is a set of two input vectors, such that their lengths are $L(v_h) = m$ and $L(v_v) = n$.
- F_m is a set of mn scalar operators that perform the same primitive operation f_m . Each scalar operator is identified as $f_m(i, j)$, $i = 1, \dots, m$, $j = 1, \dots, n$; moreover,

$$f_m(i, j) = (\{I_m(i, j) \cup v_h(i) \cup v_v(j)\}, O_m(i, j), f_s)$$

where

- $I_m(i, j)$ and $O_m(i, j)$ are the (i, j) -th element of I_m and O_m , respectively.

- $v_h(i)$ and $v_v(j)$ are the i -th and j -th element of vectors v_h and v_v , respectively.

As indicated in Chapter 4, the limitations in number of inputs and outputs to/from the operators above arises from the objective of realizing those operators in mesh arrays, and from the need to implement broadcasting by transmittent data.

The canonical form of matrix algorithms excludes the existence of branches or loops with data-dependent range. These properties are not present in most matrix algorithms of interest, in particular those algorithms that are suitable for parallel implementation, so that this limitation is minor. The only usual case of data-dependent loop arises when testing for some termination (convergence) condition. However, that case may be handled by treating the algorithm as a loop with a sufficiently large range. On the other hand, the canonical form is sufficiently general to cover algorithms that are frequently used, such as LU-decomposition, transitive closure, Faddeev algorithm, QR-decomposition, Gaussian elimination.

As indicated in Chapter 4, the canonical form above does not have any restriction on the way that variables appearing in an algorithm are referenced, that is, on how loop indices are used to access elements of matrices and vectors. The method accepts, for example, the two common types of references usually considered: (1) *uniform dependencies* and (2) *affine dependencies*. Uniform dependencies are of the form $(i - i_0)$ (i.e., an index plus/minus a constant), while affine dependencies have the more general form $(i + j - k_0)$ (i.e., a linear combination of indices and a constant). Uniform expressions are the more common type of references, and appear in all the algorithms listed in the previous paragraph. Nevertheless, the method allows both types of dependencies. Using affine expressions to access variables may lead to a fully-parallel graph that requires more work to transform it into a multi-mesh graph, and/or to an MMG with more delay nodes, than algorithms with uniform dependencies.² Appendix D gives an example of an algorithm with affine dependencies.

²As also indicated in Chapter 4, using uniform or affine expressions to access variables does not imply that the algorithm must be a uniform or an affine system of equations, as required by other methods.

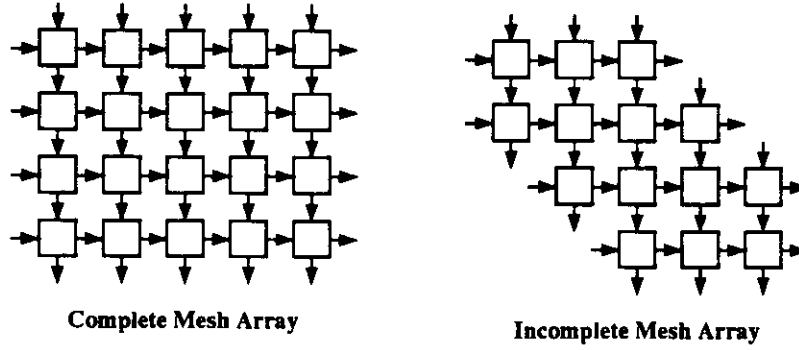


Figure 5.2: Complete and incomplete mesh arrays

5.1.2 The arrays

Definition 1 : Complete mesh array. *Collection of processing elements (PEs) or cells with the following properties:*

- *PEs are distributed on a two-dimensional grid structure such that each PE is identified by a pair of coordinates as $PE(i, j)$, where i, j are integers, $1 \leq i \leq L$ and $1 \leq j \leq M$, and L, M are the dimensions of the array.*
- *Communication links are such that, excepting at the boundaries of the mesh array, $PE(i, j)$ communicates with $PE(i, j + 1)$ and $PE(i + 1, j)$; that is, PEs are connected in nearest-neighbor fashion and with unidirectional links.*
- *External input/output to the mesh array is available only at boundary cells, that is, $PE(i, j)$ such that $(i = 1 \text{ or } j = 1)$ or $(i = L \text{ or } j = M)$.*

If there are positions in the two-dimensional grid structure that do not have a cell or there are cells without all links, then the array is an **incomplete mesh array**. Figure 5.2 depicts complete and incomplete mesh arrays. We restrict missing cells in an incomplete mesh array to fulfill the following requirements:

- Empty locations left by missing cells appear only at the ends of rows of cells. That is, a missing cell is identified by coordinates (i_m, j_m) such that $j_m > j$ or $j_m < j \forall j$, where j is used to identify an existing cell.
- The number of missing cells either increases or decreases monotonically along axes of the two-dimensional space.

In the remaining of this chapter, we will use the term *array* to imply a *mesh array*.

5.1.3 The model of execution in an array

We assume that the model of execution used by an array is synchronous, that is, all cells operate synchronously and all data transfers are performed simultaneously. Moreover, data flows through cells, and external input and output occurs only at boundary cells. In each unit of time (i.e., a time-step) a cell reads operands (from off-cell and/or local storage), performs an operation, and delivers results (off-cell and/or to local storage). Consequently, data flows in pipelined manner throughout the array, because it may advance one cell per time-step.

The execution of an algorithm in an array is characterized by the following parameters:

t_c : Computation time. Such a time corresponds to the interval between the first and last operations performed in the array to execute one instance of the algorithm.

$t(i, j)$: Computation time of PE(i, j). Continuous interval that PE(i, j) is dedicated to the execution of one instance of an algorithm, including idle time.

$t(i, j)_{max} = \max_{i,j}[t(i, j)]$: Computation time of the busiest cell in the array.

5.1.4 The performance measures

Performance measures to evaluate the implementation of algorithms in arrays are defined as follows:

Throughput of an array

Let us call t_i the interval between the initiation of two consecutive instances of a multiple-instance algorithm in an array. The throughput of such an array is

$$T = 1/t_i$$

For perfect pipelined execution, the throughput is

$$T_{pp} = 1/t(i, j)_{max}$$

Utilization of cells

The utilization of cells is the average fraction of time that array cells are busy performing operations. Utilization is computed as follows:

Let N be the number of primitive operations in an algorithm, such that all operations have the same computation time τ .

- For a single-instance algorithm

$$U_{single} = \frac{N\tau}{Kt_c}$$

- For a multiple-instance algorithm (pipelined execution)

$$U_{mult} = \frac{N\tau}{K/T}$$

5.1.5 The cost measures

Cost measures are defined as follows:

Number of computing cells. We distinguish two cases:

- K_f : number of cells in an array for a problem with fixed-size data. Such a number depends on the dimensions of matrices.
- K_p : number of cells in an array for a partitioned problem. Such a number is independent from matrices' dimensions.

Array input/output data bandwidth. Maximum number of data words that need to be transferred into and out of the array per time-step.

Storage per cell. Amount of storage in a cell that has access time compatible with the execution rate of the functional unit in such a cell.

Cell bandwidth. Maximum number of data words that need to be transferred into or out of a cell per time-step.

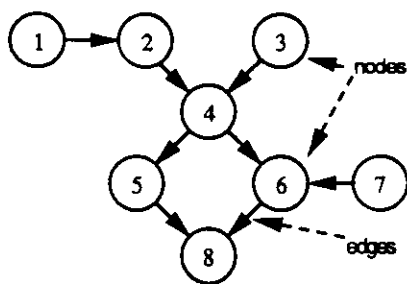


Figure 5.3: Example of a data-dependency graph

5.1.6 The graphs

We define now the different types of graphs used in our method.

5.1.6.1 Data-dependency graph

Definition 2 : *Data-dependency graph of an algorithm (DG) is a tuple $P = (D, E)$ that describes operations and data dependencies of the algorithm. D is a set of N nodes and E is a set of directed edges. Nodes correspond to operations and edges correspond to data dependencies among such operations. Each directed edge $e_{ij} \in E$ connects a pair of nodes $[n(i), n(j)] \in D$ and implies that node $n(i)$ precedes node $n(j)$. Such nodes are referred to as directly dependent nodes. Node $n(i)$ is characterized by its computation time $\tau(i)$ and its operation $\phi(i)$.*

Figure 5.3 depicts an example of a data-dependency graph. A DG is similar to an unfolded dataflow graph [Ager82], in the sense that nodes are evaluated as soon as data is available at their inputs.

In addition to operation nodes, a DG may have the following type of nodes:

Delay node: node that takes data arriving through incoming edges and delivers it to outgoing edges after a specified delay, without performing any operation on the data.

5.1.6.2 Mesh data-dependency graphs

Definition 3 : *Complete mesh data-dependency graph (CMG) is a DG with the following properties:*

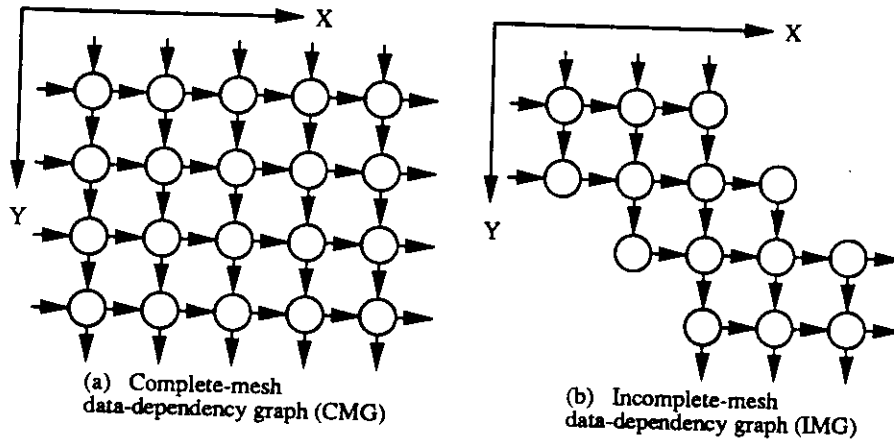


Figure 5.4: Examples of mesh dependency-graphs

- Nodes are distributed on a two-dimensional grid structure such that each node is identified by a pair of coordinates as node $n(i, j)$, where i, j are integers.
- Dependencies between neighbor nodes are unidirectional. Consequently, dependencies between nodes are as follows:

$n(i, j)$	depends directly on $n(i - 1, j)$ and $n(i, j - 1)$	for $i, j > 1$
$n(1, j)$	depends directly on $n(1, j - 1)$	for $j > 1$
$n(i, 1)$	depends directly on $n(i - 1, 1)$	for $i > 1$
- Node $n(i, j)$ has computation time $\tau(i, j)$

Figure 5.4a depicts an example of a CMG.

Definition 4 : Incomplete-mesh data-dependency graph (IMG) is a DG obtained after removing some nodes and/or edges from a CMG. Missing nodes are required to fulfill the following requirements:

- Empty locations left by missing nodes appear only at the ends of horizontal paths. That is, a missing node is identified by coordinates (i_m, j_m) such that $j_m > j$ or $j_m < j \forall j$, where j is used to identify an existing node.
- The number of missing nodes either increases or decreases monotonically along axes of the two-dimensional space.

Figure 5.4b depicts an example of an IMG. This figure shows how dependencies are affected when there is a node missing. For example, node $n(i, j)$ depends only on node $n(i, j - 1)$ if node $n(i - 1, j)$ does not exist.

In what follows, we use MG to refer to both complete and incomplete mesh data-dependency graphs. Moreover, unless stated otherwise, we assume that a CMG has size n by n , while an IMG has size n at least along one dimension.

5.1.6.3 Multi-mesh data-dependency graphs

Definition 5 : Complete multi-mesh data-dependency graph (CMMG) is a DG with the following properties:

- Nodes are distributed on a three-dimensional grid structure such that each node is identified by a triple of coordinates as node $n(i, j, k)$, where i, j, k are integers.
- Dependencies among nodes are as follows:

Node	depends directly on
$n(i, j, k)$	$n(i - 1, j, k), n(i, j - 1, k)$ and $n(i, j, k - 1)$ for $i, j, k > 1$
$n(1, j, k)$	$n(1, j - 1, k)$ and $n(1, j, k - 1)$ for $j, k > 1$
$n(i, 1, k)$	$n(i - 1, 1, k)$ and $n(i, 1, k - 1)$ for $i, k > 1$
$n(i, j, 1)$	$n(i - 1, j, 1)$ and $n(i, j - 1, 1)$ for $i, j > 1$

- All nodes have computation time $\tau(i, j, k) = \tau$.

Figure 5.5a depicts an example of a CMMG. Such a CMMG is composed of CMGs with dependencies among nodes in the same position in the component meshes. Consequently, a CMMG may be regarded as composed of parallel (dependent) CMGs.

Definition 6 : Incomplete multi-mesh data-dependency graph (IMMG) is a DG graph obtained after removing some nodes and/or edges from a CMMG. Missing nodes are required to fulfill the following requirements:

- Empty locations left by missing nodes appear only at the ends of paths along one dimension. For example, a missing node along the dimension represented by the second index is identified by coordinates (i_m, j_m, k_m) such that $j_m > j$ or $j_m < j \forall j$, where j is used to identify an existing node.
- The number of missing nodes either increases or decreases monotonically along axes of the three-dimensional space.

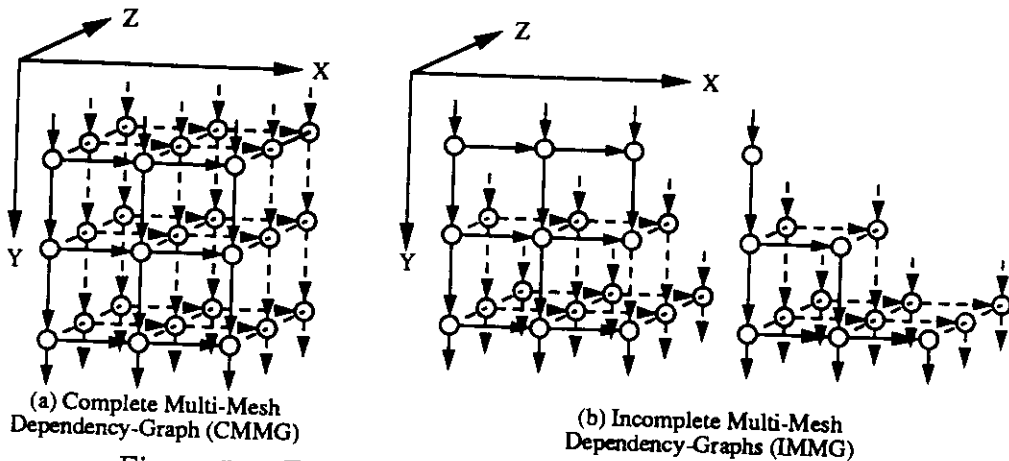


Figure 5.5: Examples of multi-mesh dependency-graphs

Figure 5.5b depicts examples of IMMGS. Since an IMMGS has parallel meshes along any of the three dimensions, an IMMGS might have parallel meshes with different number of nodes and/or edges along one dimension and the same number of nodes and/or edges along some other dimension, as is inferred from the figure.

In what follows, we use MMGS to refer to both complete and incomplete multi-mesh data-dependency graphs. Moreover, unless stated otherwise, we assume that a CMMGS has size n by n by n , while an IMMGS has size n along at least one dimension.

5.1.6.4 Fully-parallel data-dependency graph

Definition 7 : Fully-parallel data-dependency graph of a matrix algorithm (FPG) is a data-dependency graph with the following characteristics:

- FPG is a directed acyclic graph.
- Nodes correspond to primitive operators (unary, binary or ternary). Consequently, every node $n(i)$ has at most three incoming edges and at most three outgoing edges, which are named left, center and right inputs, and left, center and right outputs, respectively.
- All nodes have the same computation time $\tau(i) = \tau$.

Using FPGs to represent matrix algorithms requires that loops be unfolded, because FPGs are directed acyclic graphs. Moreover, FPGs are not capable of representing branches or data-dependent loops. However, as we stated earlier,

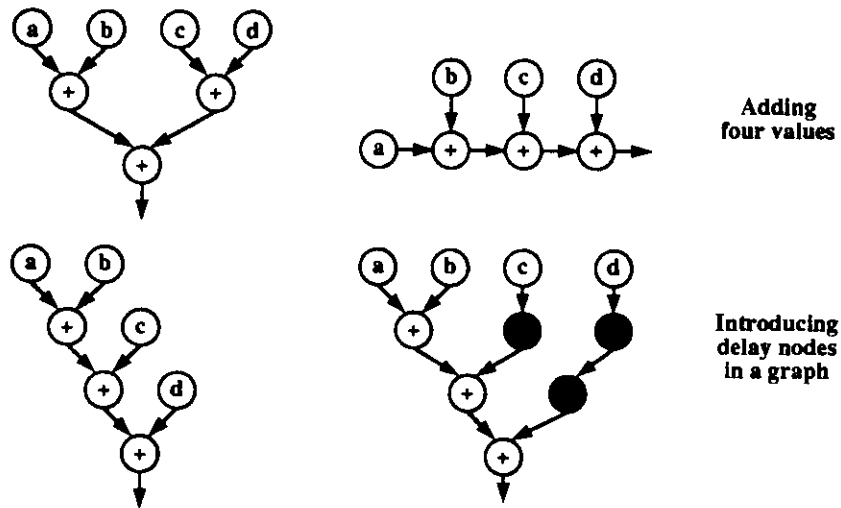


Figure 5.6: Examples of equivalent graphs

branches do not appear in matrix algorithms of interest, while data-dependent loops can be handled by considering a loop with a range sufficiently large.

5.1.7 The equivalence of graphs

The equivalence between two graphs is determined by the following definition.

Definition 8 : Graph equivalence. *Two data-dependency graphs P_1 and P_2 are equivalent if they describe the same computation. That is, the same set of input values in both DGs produces the same set of output values in both DGs.*

The equivalence among two data-dependency graphs as defined above is verified by writing the expressions for outputs in both graphs. Such a definition of equivalence does not include any requirements in terms of path length or timing properties in a graph. Examples of equivalent graphs are shown in Figure 5.6, which depicts alternative ways to add four elements. Note that adding delay nodes to a graph does not change the corresponding computation.

The definition of equivalence above is used as the basis for the graph transformations devised in our design method. Consequently, a transformation on a graph is a valid one as long as it fulfills such a definition.

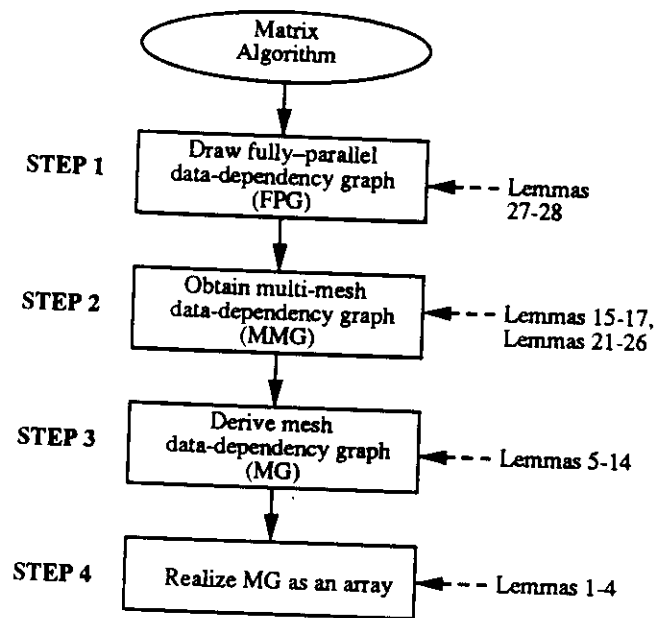


Figure 5.7: Summary of design method's formalization

5.1.8 The realization of a graph

Direct realization of the DG of an algorithm is a one-to-one mapping between the DG and an implementation, such that each node of the graph is realized as a different PE and each edge of the graph is realized as a different communication link. Consequently, the resulting structure has the same number of cells and interconnection links as there are nodes and edges in the graph, respectively.

Perfect realization of the DG of an algorithm is a direct realization that achieves perfect pipelined execution of the algorithm.

Now that the terminology used in the formalization of our technique to design arrays has been presented, we can center our attention on the specific steps of the method. As stated earlier, we follow a bottom-up approach by discussing first mapping an MG onto an array. The overall organization of this formalization is shown in Figure 5.7.

5.2 Step 4: Realizing a mesh data-dependency graph as an array

In our method, the step closer to an implementation corresponds to realizing a mesh data-dependency graph (MG) as an array. We present now the conditions for such a realization and the impact of those conditions on the utilization of cells in the resulting array. As will be inferred from the discussion here, the mesh dependency-graph is equivalent to the G-graph used in Chapter 4.

Lemma 1 *Consider a mesh data-dependency graph that describes one instance of a multiple-instance algorithm. Such an MG is perfectly realized as an array if input data for different instances of the algorithm is accessible as needed (i.e., always available).*

Proof: A CMG or IMG has nodes distributed on a two-dimensional grid structure with dependencies in unidirectional nearest-neighbor fashion. Consequently, such an MG is directly realized as an array by assigning node $n(i, j)$ to processing element $PE(i, j)$ and edge $e_{(i,j),(k,l)}$ to an interconnection link between $PE(i, j)$ and $PE(k, l)$. Since dependencies in the CMG or IMG are unidirectional, communication links in the array are unidirectional.

Moreover, due to the dependencies in the MG, each cell of the array initiates computation of its associated node delayed a number of time-steps with respect to neighbor cells upstream the flows of data (the delay depends on the size of local memory in cells). Consequently, computation advances through the array as a wavefront that moves at the same rate along the two dimensions, as illustrated in Figure 5.8b. Data is input to the array in skewed manner following the computation wavefront, as also depicted in Figure 5.8b, and intermediate data flows in pipelined mode through the array.

Since the computation wavefront for one instance of the algorithm and the associated intermediate data move through the array, a new instance is initiated without delays as long as input data is available. Such a case is illustrated in Figure 5.8c. Consequently, the MG is perfectly realized as an array. ■

Lemma 2 *An array that implements a perfectly realized mesh data-dependency graph achieves utilization $U = 1$ if and only if all nodes have the same computation time.*

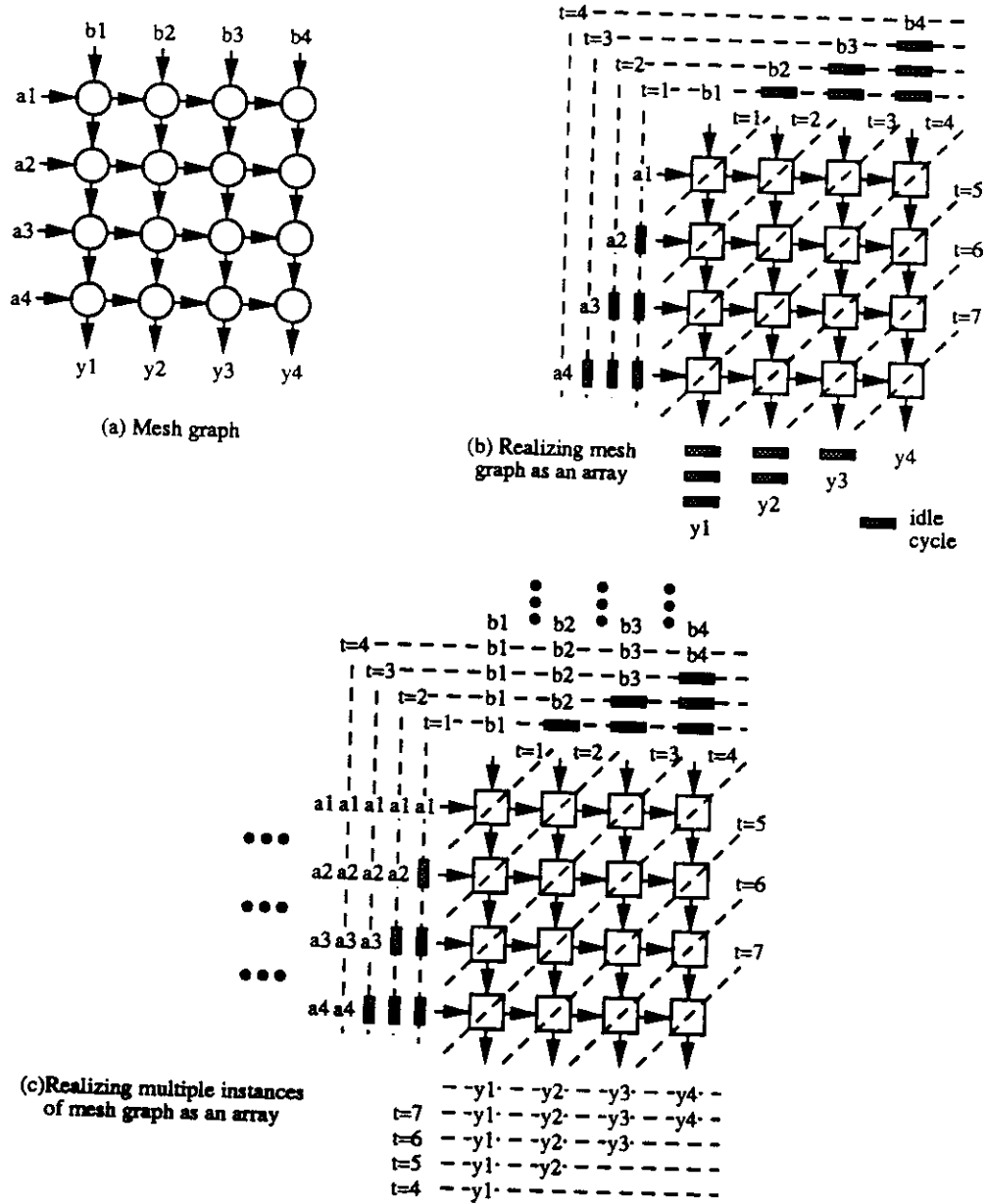


Figure 5.8: Realizing a mesh graph as an array

Proof: Necessity: Consider an MG with Υ nodes, where each node has computation time $\tau(i, j) = \tau_o$. Since each node is realized as one cell of the array, the number of cells is $K = \Upsilon$, and throughput is $T_{pp} = \tau_o^{-1}$. Consequently,

$$U = \frac{\sum_{j=1}^{\Upsilon} t(j)}{K/T_{pp}} = \frac{\Upsilon \tau_o}{K/T_{pp}} = \frac{\Upsilon \tau_o}{\Upsilon \tau_o} = 1$$

Sufficiency: If an array has utilization $U = 1$ it implies that all cells are used all the time while executing multiple instances of the algorithm. Consequently, all cells must perform operations of the same duration. Since each cell corresponds to one node in the MG, then all nodes have the same computation time. ■

We have described the realization of a mesh dependency-graph as an array. Moreover, we have identified the condition under which such a realization produces optimal utilization of cells in the resulting array. The MG used here is equivalent to the G-graph used in Chapter 4. The only apparent differences are that nodes in a G-graph are composed of many primitive nodes and produce many outputs, while nodes in an MG have been described as single operations with a single output in each direction. However, if we schedule primitive operations in the G-graph and regard the resulting sequence of operations as a single one with a set of outputs, then G-graph and MG become the same description. Such an equivalence will be discussed in detail in the next section.

Given the formalization developed so far, we can move one step higher in our method and look into mechanisms that allow deriving a mesh-dependency graph.

5.3 Step 3: Transforming a multi-mesh data-dependency graph into a mesh graph

The next step in our method considers that an algorithm is described by a three-dimensional data-dependency graph. We state here the process to convert such a three-dimensional graph into a mesh dependency graph. Moreover, we indicate the impact of graph properties on characteristics of the array obtained, according to the discussion in the previous section.

Definition 9 : A prism of size p by q by n in an MMG is a prism originating at location (i_o, j_o, k_o) in the three-dimensional space that encloses all

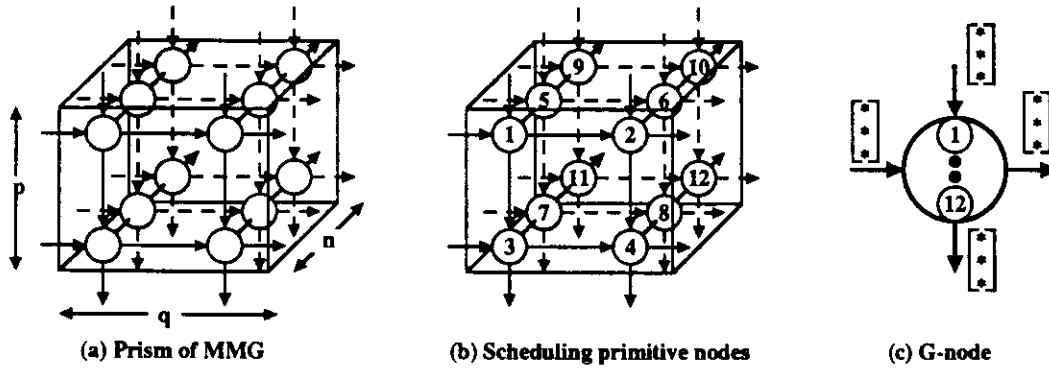


Figure 5.9: Collapsing a prism of primitive nodes onto a single node

primitive nodes identified by indices (i, j, k) such that $i \in \{i_o, i_o + 1, \dots, i_o + n\}$, $j \in \{j_o, j_o + 1, \dots, j_o + p\}$, $k \in \{k_o, k_o + 1, \dots, k_o + q\}$. In this prism, p is the height of the base while q is the width.

Figure 5.9a illustrates a prism of size $p = 2, q = 2, n = 3$. Primitive nodes in a prism are contained in dependent submeshes of size p by q .

Note that, for a particular multi-mesh graph, is possible to draw prisms along the three dimensions and not only along the Z -axis as indicated above. However, one can always rotate the multi-mesh graph (or rename the axes) so that prisms appear along the Z -axis. In the remaining of this chapter, and unless explicitly stated otherwise, we assume that prisms are drawn as defined above.

Lemma 3 *The primitive nodes enclosed by a prism in an MMG are equivalent to a single node (i.e., a G-node) whose functionality corresponds to the sequential execution of the primitive nodes. Such a sequencing is determined by a schedule that does not violate the dependencies between primitive nodes in the MMG.*

Proof: Consider dependent submeshes of an MMG, each one consisting of p by q primitive nodes that belong to the same mesh of an MMG, as illustrated in Figure 5.9a for $p = q = 2$. For those submeshes, select a schedule that does not violate the dependencies between primitive nodes, such as the one depicted in Figure 5.9b. In this schedule, all nodes in horizontal paths of the prism are executed before advancing to nodes in a lower horizontal path, and a complete mesh is executed before advancing to an inner mesh.

Let us analyze first one submesh, as illustrated in Figure 5.10a for $p = q = 2$. Any submesh is equivalent to a sequence of pq dependent nodes that follows the schedule of primitive nodes, as depicted in Figure 5.10b for the mesh in Figure 5.10a. In this sequence, $(p - 1)$ dependencies (i.e., edges) are added to enforce the schedule chosen. The new graph may be collapsed onto an equivalent single node whose computation time is pqn , has $(p + q)$ input edges plus $(p + q)$ output edges, as depicted in Figure 5.10c for the sample mesh considered. There are p incoming and p outgoing edges horizontally, while q incoming and q outgoing edges flow vertically. The resulting node is also equivalent to a node where all edges flowing along one direction are multiplexed through a single edge, both at inputs and outputs. This transformation is shown in Figures 5.10d for the node in Figures 5.10c. The multiplexing/demultiplexing process as well as the internal data transfers are control functions that may be hidden from outside, leading to a node such as the one depicted in Figure 5.10e. This node has a single input and a single output edge in each direction, which carry sequences of p and q value, respectively.

Similarly, each p by q submesh in the prism is equivalent to a single node whose functionality corresponds to the sequential execution of primitive nodes. All such nodes compose a sequence of directly dependent nodes which can be collapsed onto an equivalent single G-node, as the one depicted in Figure 5.9c.

The single node representation is equivalent to the set of primitive nodes enclosed by a prism in an MMG, because identical inputs deliver the same outputs although at different time. ■

Note that arrival of data to nodes in Figure 5.10b is not synchronized. For example, the output from node $n(1)$ arrives at the same time to nodes $n(2)$ and $n(3)$. However, according to the schedule chosen and the corresponding added dependency, $n(3)$ is executed after $n(2)$. Such synchronization problems are discussed in Section 5.5.4.

Corollary 1 *The computation time of a G-node is the sum of the computation times of primitive nodes in the corresponding prism. Moreover, if primitive nodes in a prism correspond to different operations, then the resulting G-node has to perform several operations.*

Proof: By Lemma 3. ■

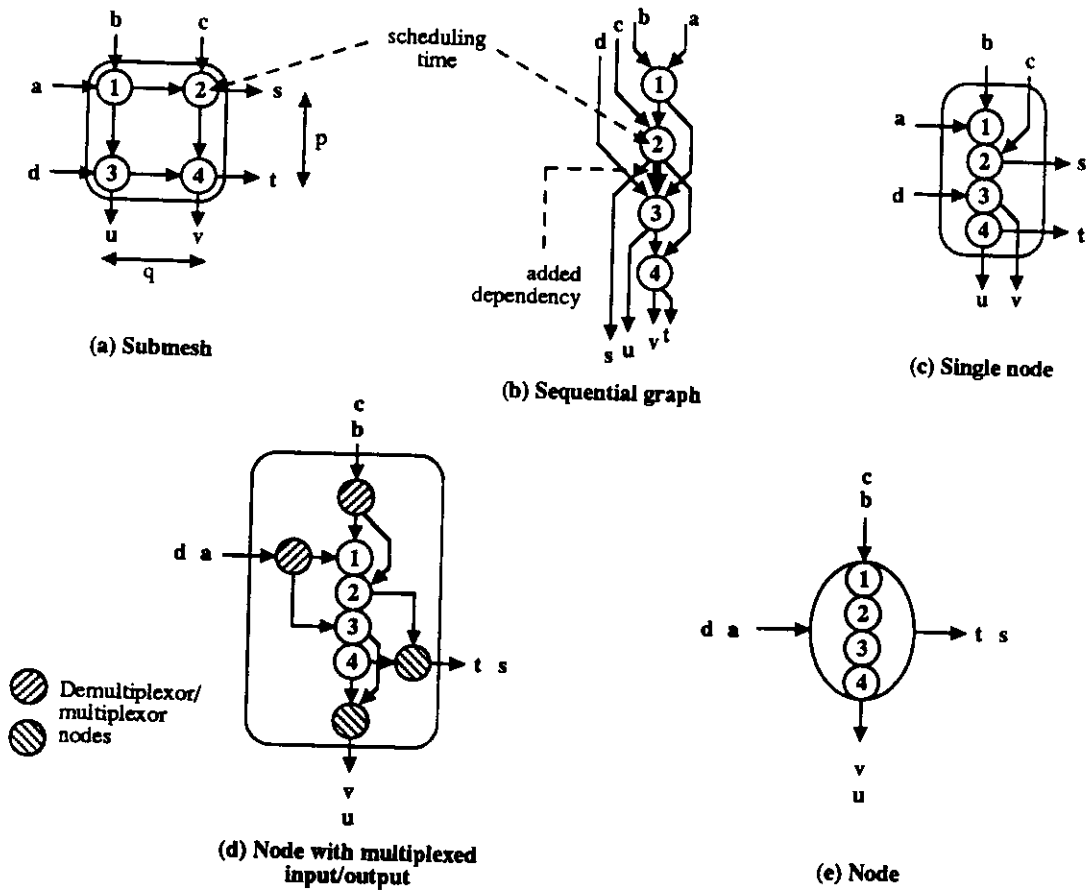


Figure 5.10: Collapsing neighbor nodes from a mesh onto a single node

From the lemma above, we can devise an important graph transformation, as discussed below.

Definition 10 : Grouping-by-prisms in an MMG (or grouping for short) is the process of reducing the number of nodes in the MMG by collapsing prisms of primitive nodes onto nodes. Each of such prisms is collapsed onto a single node (a G-node). Moreover, grouping-by-prisms reduces the MMG to a two-dimensional graph.

We use now Definition 10 to collapse an MMG onto an MG, as follows:

Lemma 4 A multi-mesh dependency graph (MMG) is equivalent to a mesh dependency graph (MG) obtained by the procedure grouping-by-prisms.

Proof: From Lemma 3, a prism of primitive nodes is collapsed onto an equivalent single G-node. Let us allocate all primitive nodes in the MMG to *parallel prisms*. That is, let us enclose all primitive nodes in the MMG with prisms of size p by q by n (where n is the size of the MMG), though prisms are not required to include the same number of primitive nodes in the case of IMMGS. Since prisms are parallel, they are distributed across the MMG in a nearest-neighbor mesh structure. Consequently, performing the procedure grouping-by-prisms reduces such prisms to G-nodes that are also placed in a nearest-neighbor mesh structure, and the resulting graph is a mesh graph. Moreover, the resulting graph is equivalent to the original MMG, because both deliver the same outputs. ■

As a particular case, grouping an MMG by prisms of size 1 by 1 by n corresponds to projecting the three-dimensional MMG onto a two-dimensional MG along one axis. Figure 5.11 depicts such a grouping along the three axes of an MMG.

Lemma 5 Grouping-by-prisms in a CMMG along any axis leads to a CMG where all G-nodes have the same computation time.

Proof: All paths of the CMMG along any axis have the same length. Consequently, G-nodes obtained by grouping are composed of the same number of primitive nodes. Therefore, all G-nodes have the same computation time (i.e., the same number of primitive operations to perform). ■

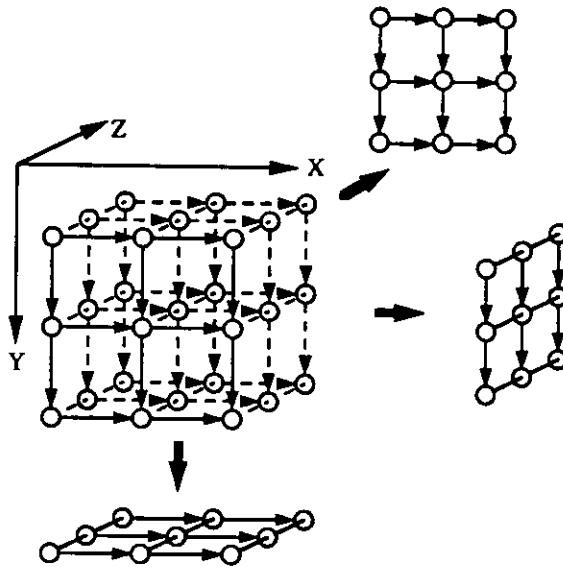


Figure 5.11: Grouping-by-prisms of size 1 by 1 by n in a CMMG

Corollary 2 *Grouping-by-prisms in an IMMIG may lead to an MG where G-nodes do not have the same computation time.*

Proof: The paths of the IMMIG along at least one dimension do not have the same length (otherwise the graph would be a CMMG). Consequently, grouping along such a dimension leads to G-nodes with different computation time. ■

The definition of grouping explicitly states that such a grouping should be performed along axes of the three-dimensional space. The following lemma discusses the drawbacks that appear when grouping along other directions.

Lemma 6 *Grouping-by-prisms in an MMG along a direction other than axes of the three-dimensional space produces a two-dimensional graph with bidirectional flow of data. Moreover, the resulting graph has G-nodes with different computation time when the original MMG is a CMMG.*

Proof: Consider grouping-by-prisms along a direction other than the axes. Let us assume that such a grouping is as depicted in Figure 5.12a, where nodes have been tagged with a number identifying the prism that encloses them. As a result, there are paths going, for example, from prism 1 to prism 3 and then back to prism 1 (such a path has been highlighted in Figure 5.12a). Consequently, grouping along a direction other than the axes leads to neighbor prisms connected by edges

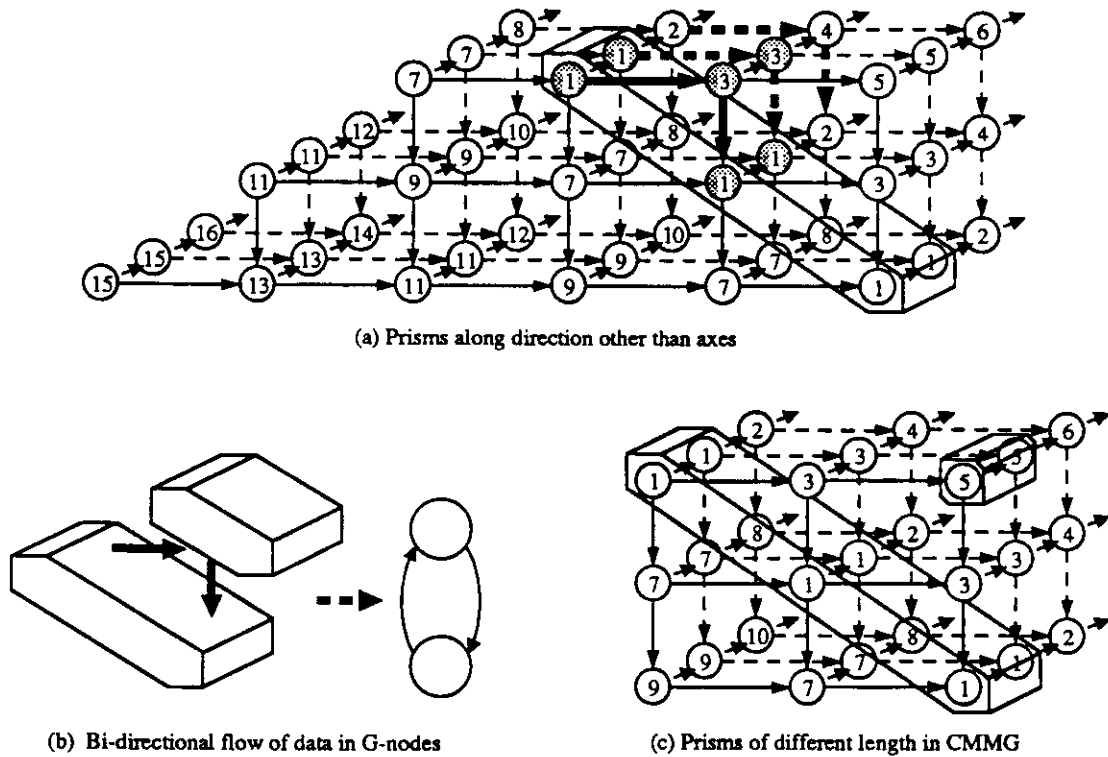


Figure 5.12: Grouping along directions other than axes

pointing in both directions, as illustrated in Figure 5.12b. When collapsing such prisms onto G-nodes, those edges become edges of a two-dimensional graph with opposite flow of data.

By Lemma 5, if the MMG is a CMMG then grouping along any axis implies selecting prisms of the same length, namely n . In contrast, grouping along other direction leads to prisms that do not have the same length, as shown in Figure 5.12c. Consequently, the number of primitive nodes enclosed by the prisms (i.e., p by q by length of prism) is not constant. Since the number of primitive nodes determines computation time of G-nodes, then G-nodes have different computation time. ■

Lemma 6 indicates that an MG derived from an MMG by grouping along a direction other than one axis is less convenient in devising an array. Drawbacks in such a case are implementation complexity, due to the need of more ports in a cell, and utilization of the resulting array, because even for a CMMG G-nodes have different computation time. The only cases where grouping along a direction other than an axis might be convenient are:

- When specialized cells are desired in an array for an algorithm that includes different primitive operations (i.e., cells that perform different operations). In such a case, it *might* be possible to select a direction for grouping that leads to prisms containing only one type of primitive nodes, at the expense of bidirectional flow of data in the G-graph and consequently more communication ports in array cells.
- Whenever is possible to find a direction that leads to prisms with the same number of primitive nodes in an algorithm represented by an IMMKG, or whenever is possible to find a better distribution of primitive nodes per prism than by grouping along an axis. Such cases would lead to better utilization of cells than grouping along axes.

Note that the two cases above are highly algorithm dependent, and one cannot develop a general procedure or technique based on them. In the remaining of this chapter, unless explicitly stated otherwise, we assume that an MG derived from an MMKG is obtained by grouping along axes of the three-dimensional space.

5.4 Properties of cells that depend on the grouping process

Grouping determines several important parameters of cells in an array derived from an MMKG, such as utilization, local storage, pipelining and operations per cell. We discuss these issues in the following sections.

5.4.1 Utilization of cells

Lemma 7 *A mesh dependency-graph (MG) derived from a complete multi-mesh dependency-graph (CMMKG) is perfectly realized as a complete array with utilization $U = 1$.*

Proof: Grouping-by-prisms in a CMMKG is done by selecting parallel prisms along any of the axes X , Y , or Z . By Lemma 5, grouping leads to G-nodes that have the same computation time. By Lemma 2, the resulting CMG is perfectly realized as a complete array with utilization $U = 1$. ■

Lemma 8 *A mesh data-dependency graph (MG) derived from an incomplete multi-mesh data-dependency graph (IMMKG) is perfectly realized as an incomplete array*

with utilization $U = 1$ if and only if all prisms have the same number of primitive nodes.

Proof: *Necessity:* If all prisms have the same number of primitive nodes, then grouping such prisms leads to G-nodes with the same computation time. However, since the graph is an IMMIG all such G-nodes do not constitute a complete mesh (otherwise it would be a CMMG). Therefore, the resulting graph is an IMG that has all nodes with the same computation time. By Lemma 2, such an IMG is realized as an incomplete array with utilization $U = 1$.

Sufficiency: If the incomplete array has utilization $U = 1$ then all cells in the array execute operations with the same computation time. Such operations correspond to G-nodes of a graph. In turn, G-nodes correspond to the sequential execution of a set of primitive nodes. Since all G-nodes have the same computation time, and since the computation time of all primitive nodes is the same, then G-nodes are composed of the same number of primitive nodes. ■

Corollary 3 *A mesh data-dependency graph obtained from an incomplete multi-mesh data-dependency graph by prisms with different number of primitive nodes is perfectly mapped onto an array with utilization $U < 1$.*

Proof: By Lemma 8. ■

From the two lemmas above we infer the condition for realizing an MMG as an array with utilization $U = 1$, namely the selection of prisms with the same number of primitive nodes. However, if such an optimal mapping is not feasible due to characteristics of the MMG, we would like to know how to perform grouping so that utilization is maximized even though it does not reach $U = 1$. The following lemma addresses this issue.

Lemma 9 *Let R be an array that implements an MMG with utilization $U_R < 1$ and throughput $T_R = 1/t_f$ (that is, the busiest cell computes t_f operations and all G-nodes have computation time $\tau(i, j) \leq t_f$). U_R is maximized when the number of G-nodes in the corresponding MG is minimized.*

Proof: Utilization is given by $U_R = N\tau/(Kt_f)$. For a given algorithm, the number of primitive operations N is fixed and all operations have the same computation time τ . Moreover, for a given throughput $T_R = 1/t_f$, the maximum computation

time per cell is t_f , also fixed. Consequently, maximizing U_R requires to minimize K , the number of cells in the array. Since realizing an MG as an array produces as many cells as G-nodes in the MG, maximizing utilization corresponds to minimizing the number of G-nodes. ■

5.4.2 Types of operations per cell

Lemma 10 *Let R be an array that realizes an MMG. The operations that a cell in R has to compute (i.e., the complexity of a cell) are given by the functionality of primitive nodes that are enclosed in a prism when performing the procedure grouping-by-prisms.*

Proof: From Lemma 3, a prism of primitive nodes is equivalent to a G-node whose functionality is given by the sequential execution of all primitive nodes in the prism. Since a G-node is realized as a cell, the different primitive operations in a prism determine the computing capabilities required in the cell. ■

Lemma 10 provides a criterion to select prisms based on cell complexity. For example, to devise an array where cells perform a single operation it is necessary to find prisms in the MMG that contain only one type of primitive node. It should be noted that such a choice may not always be possible.

5.4.3 Storage per cell

Lemma 11 *Let R be an array that implements an MMG. Define*

- $CS(t)$: *cut-set in a prism of primitive nodes that includes all nodes executed up to time t .*
- $E_{out}(t)$: *set of edges in $CS(t)$ that leave the prism (i.e., edges that reach a destination node outside the prism).*
- $|S|$: *cardinality of set S .*

Local storage required by a cell in R is given by

$$C_w = \max_{(t)} |CS(t) - E_{out}(t)|$$

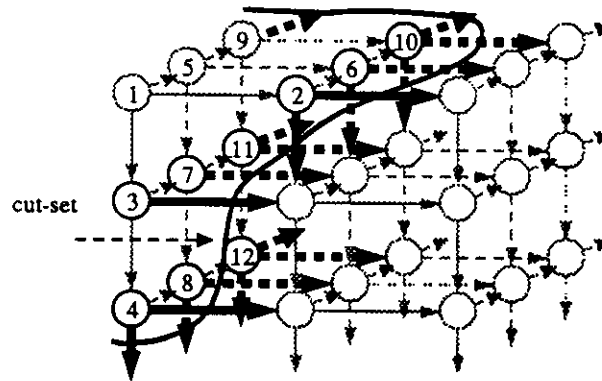


Figure 5.13: Cut-set of primitive nodes executed up to time $t = 12$ in a prism

Proof: A cut-set in a prism of an MMG that includes all nodes executed up to a given time t consists of edges directed from nodes already executed towards nodes that are scheduled for later execution, as depicted in Figure 5.13. All such edges carry values that are needed for the execution of the corresponding destination nodes within the prism, excepting those that leave the prism which correspond to data transferred to other cells.

At time $t + 1$, one primitive node from within the prism is executed and values carried by edges incident on such a node are used; values in all remaining edges must be saved for later use. Consequently, a cell that implements the prism must save in local storage all values from edges arriving to nodes within the prism, while edges leaving the prism do not need to be saved. The size of this storage is given by the maximum cardinality of a cut-set after removing from it those edges leaving the prism. ■

Corollary 4 *Grouping-by-prisms, with prisms of the same base size, leads to the same or higher storage requirements for a complete prism than for an incomplete one.*

Proof: Consider an incomplete prism with base size p by q (i.e., the maximum number of primitive nodes in one mesh within the prism is p by q), and also a complete prism with the same base size. Moreover, consider the same schedule of primitive nodes in both complete and incomplete prisms. The cut-set that includes all nodes executed up to time t , for both complete and incomplete prisms, is depicted in Figure 5.14. Since an incomplete prism is obtained by removing some nodes and edges from a complete one, the cut-set in the complete prism

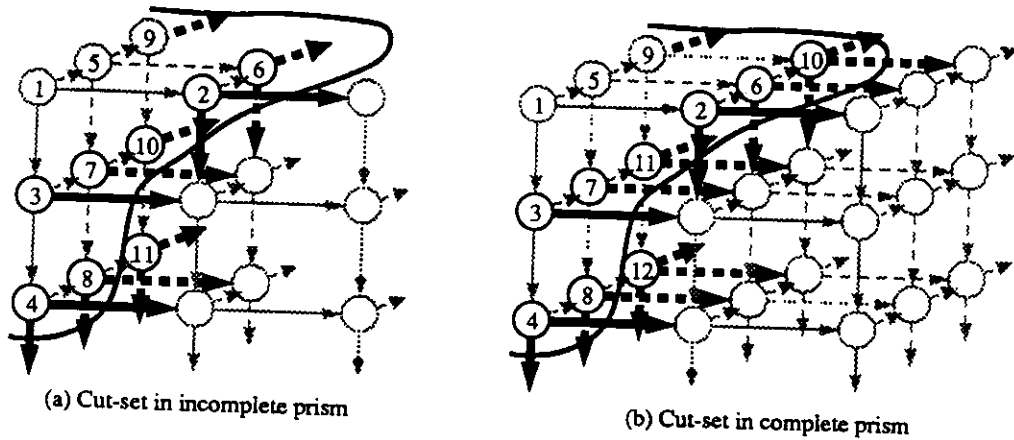


Figure 5.14: Cardinality of a cut-set in complete and incomplete prisms

has the same number or more edges than in the incomplete one (equal or larger cardinality). ■

An IMM has incomplete prisms but it may also have a complete (or almost complete) one in the denser portion of the graph. Moreover, the incompleteness may exist only at one end of the prism while the rest has the structure of a complete prism. Consequently, since a complete prism requires the same or more storage than an incomplete one, we can focus on determining the amount of storage required for a complete prism, as discussed below. The same situation is true for determining the bandwidth required by a cell, or the suitability of a cell for using a pipelined functional unit, as discussed later.

Lemma 12 *Let R be an array that realizes a CMMG. Scheduling primitive nodes by meshes of size p by q in the corresponding prism leads to the minimum amount of storage required in a cell. Moreover, this amount is*

$$C_w = q(p + 1)$$

and the contents of this storage are accessed in First-in, First-out (FIFO) manner.

Proof: From Lemma 11, the amount of storage required at time t is given by a cut-set in the prism that includes all primitive nodes executed up to t . Such a cut-set may be regarded as a *surface* that partitions the prism into two disconnected graphs, and storage locations are associated to edges of the prism that are incident on the surface. Surfaces parallel to a mesh of the prism lead to lower number of

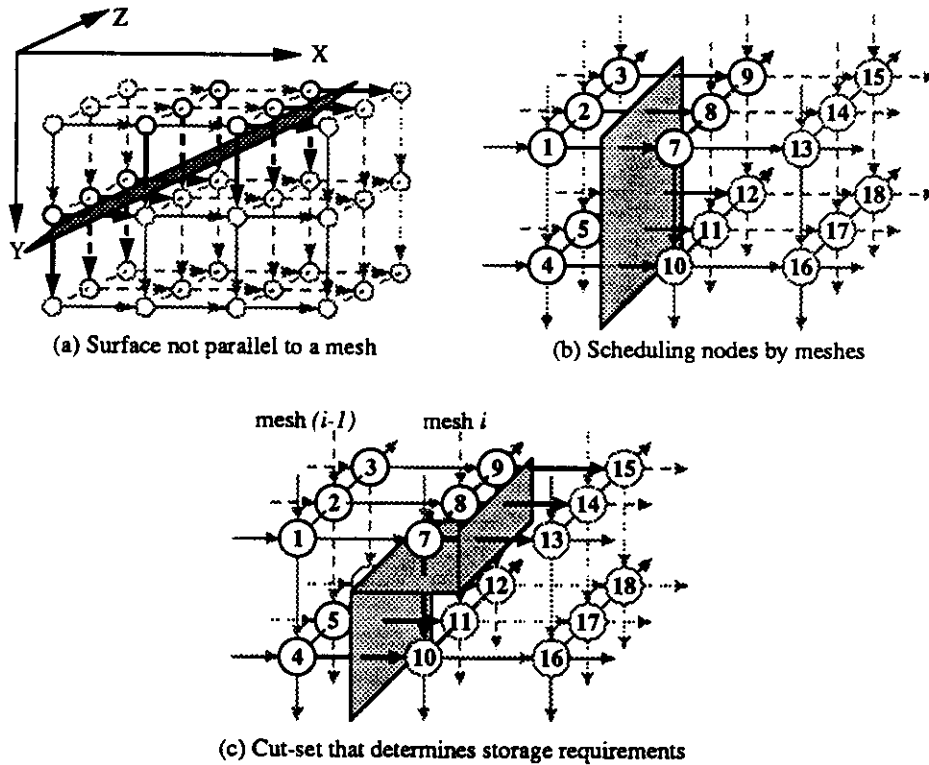


Figure 5.15: Determining local storage in a cell from a prism

edges incident on the surface, because only edges along one axis are incident on it. In contrast, surfaces that are not parallel to a mesh cut across edges that flow along two axes of the MMG, as shown in Figure 5.15a. Consequently, to reduce the amount of storage required one should schedule all primitive nodes in a mesh of the prism before moving to another mesh. Moreover, one should select meshes as small as possible.

Since the size of the prism is p by q by n , possible sizes of surfaces parallel to a mesh are p by q , p by n , and q by n . The smallest of these is p by q because $n \gg p, q$. Consequently, obtaining a cut-set corresponding to a surface parallel to meshes of size p by q can be accomplished by scheduling primitive nodes by meshes of the same size, as depicted in Figure 5.15b.

Assuming the schedule shown in Figure 5.15b, the storage required by a cell is given by a cut-set that includes all primitive nodes in an horizontal (vertical) path of the p by q mesh i , and all primitive nodes in the remaining horizontal (vertical) paths of the predecessor mesh $(i - 1)$, as shown in Figure 5.15c. This amount of storage corresponds to that needed to save data in edges flowing vertically

from nodes in mesh i , and in edges flowing horizontally from meshes i and $i - 1$. Scheduling any other node from mesh i removes as many edges from the cut-set as it adds to it, so that the storage requirements do not increase. Therefore, the amount of storage required is

$$C_w = pq + q = q(p + 1)$$

because there are pq edges flowing horizontally and q edges flowing vertically in the cut-set in Figure 5.15c.

Moreover, both horizontal and vertical flows of data are used separately so that local storage may be implemented as two independent modules of size pq and q , respectively. Since the scheduling order within meshes is the same throughout all meshes, data is sent to each local storage module in the same order that it is retrieved. Consequently, the addressing pattern for these storage modules corresponds to that of First-in, First-out (FIFO) buffers. ■

5.4.4 Cell bandwidth

Lemma 13 *Let R be an array that realizes a CMMG. Communication bandwidth through ports of a cell in R that executes a G-node scheduled as indicated in Lemma 12 is*

$$C_{BW}^X = 1/q \quad , \quad C_{BW}^Y = 1/p$$

Proof: A cell that implements a G-node executes primitive operations in the corresponding prism scheduled by meshes of size p by q , where p is the height of the prism base and q is the width. Edges going into a prism correspond to data that arrives to a cell. Such a cell reads an input value (i.e., an incoming edge to the prism) and performs q operations before reading another input value from the same side of the prism, as shown in Figure 5.16. Consequently, $C_{BW}^X = 1/q$.

Similarly, the cell reads an input value from the other side of the prism and then performs p operations before reading another value from such a side, leading to $C_{BW}^Y = 1/p$.

Expressions identical to those above are obtained for output ports. ■

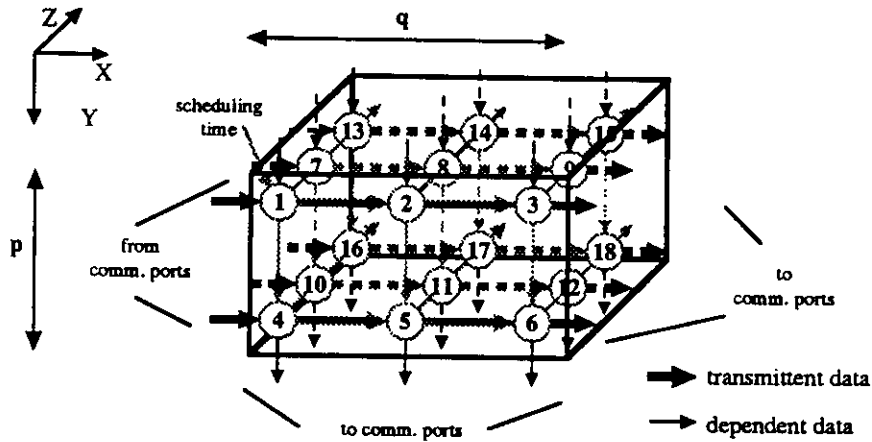


Figure 5.16: Cell bandwidth and pipelining

5.4.5 Cell pipelining

Lemma 14 *Let R be an array that realizes a CMMG. A cell in R that realizes a G-node scheduled as indicated in Lemma 12 may have a pipelined functional unit of up to q (p) stages if the flow of data parallel to the side of the prism that determines q (p) is transmittent data.*

Proof: We know that a cell that realizes a G-node executes primitive operations in the corresponding prism scheduled by meshes of size p by q . Such a cell may initiate operations in a pipelined functional unit at successive time-steps if there are data-independent operations ready for execution. Primitive operations connected by transmittent data are data independent and therefore suitable for pipelined execution. Consequently, scheduling operations by following the flow of transmittent data allows initiating operations in a pipelined functional unit at successive time-steps, as is inferred from Figure 5.16. ■

In this section, we have described transforming an MMG into an MG suitable for perfect realization as an array. We have also stated that such a realization leads to arrays with utilization $U = 1$ if all G-nodes have the same computation time. Moreover, we indicated that maximizing utilization when G-nodes have different computation time requires minimizing the number of G-nodes. In addition, we described how characteristics of cells in an implementation (i.e., operations, storage, bandwidth, pipelining) are inferred from the prisms that define the G-nodes. Consequently, if a matrix algorithm is described in terms of an MMG, then its perfect

realization as an array is straight-forward and the performance of the array may be evaluated in advance.

Therefore, the process of transforming an MMG into an MG is driven by implementation issues, such as utilization of cells, cell complexity, storage per cell, cell bandwidth, and cell pipelining. Moreover, selecting a direction for grouping and the size of prisms determines the values of those implementation parameters and impacts the performance and cost of an array. Consequently, the method allows evaluating different alternatives in searching for the one that better fulfills specific implementation constraints. Such constraints are incorporated into the method, as described in this section.

The next step in the formalization consists of a procedure to derive an MMG from a description of an algorithm that is less regular, namely an FPG.

5.5 Step 2.2: Transforming a three-dimensional graph into an MMG

So far we know how to perfectly realize an MMG as a mesh array. The remaining question is how to describe matrix algorithms of interest as MMGs. We address this issue now.

An MMG is characterized by specific properties, in particular unidirectional and nearest-neighbor dependencies in a three-dimensional space. On the other hand, a fully-parallel data-dependency graph does not necessarily exhibit such characteristics. Consequently, we need to provide transformations that will remove from an FPG properties that do not exist in an MMG. These undesirable characteristics, which often appear in the dependency graph of matrix algorithms, are:

- data broadcasting
- bidirectional dependencies
- non nearest-neighbor dependencies

We consider first transforming a three-dimensional graph into an MMG. In the next section, we discuss how the three-dimensional graph is obtained from the FPG.

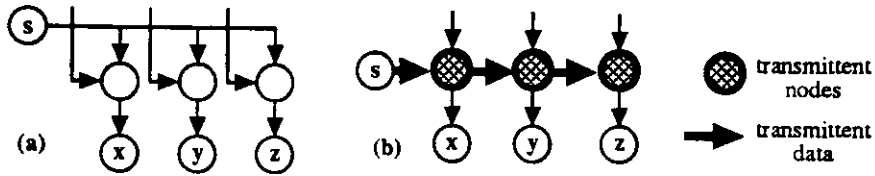


Figure 5.17: Example of broadcasting and transmittent data

5.5.1 Eliminating data broadcasting

The approach to deliver broadcasted data to several destinations in a graph consists of transferring data *through* nodes. Such an approach increases the latency of an algorithm, but it does not affect throughput. Removing data broadcasting is performed as indicated below.

Definition 11 : *Transmittent data* is data that propagates through nodes of a graph without being modified [Kung88c].

Definition 12 : *Transmittent node* is a node that uses a transmittent data element for local computation and delivers such a data in addition to the result of the computation within the node. Consequently, transmittent nodes produce more than one output.

Lemma 15 *Transmittent data and transmittent nodes are a suitable mechanism to implement data broadcasting.*

Proof: Broadcasting originates data in one node and delivers that data to several nodes, as depicted in Figure 5.17a. This graph is equivalent to the graph shown in Figure 5.17b, where highlighted nodes are transmittent nodes. Data going through transmittent nodes without being modified corresponds to transmittent data. The original graph exhibiting broadcasting has been transformed into an equivalent graph with transmittent data, so transmittent data and transmittent nodes are a suitable mechanism to implement broadcasting. ■

5.5.2 Eliminating bidirectional dependencies

An MMG has unidirectional dependencies between nearest-neighbor nodes. In contrast, a three-dimensional graph of a matrix algorithm with nearest-neighbor

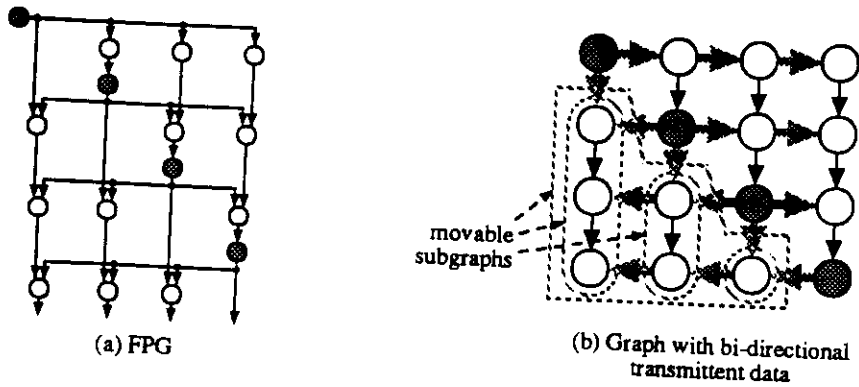


Figure 5.18: Bidirectional dependencies in a graph

edges may have bidirectional dependencies arising from transmittent data (i.e., after eliminating broadcasting). Note that these bidirectional dependencies appear only when one edge reaches more than one destination, that is, only with broadcasted data. Figure 5.18 depicts an example corresponding to one mesh in a three-dimensional graph. In this figure, nodes to the left of the main diagonal receive transmittent data from the right, while nodes at the right receive the same transmittent data from the left. Although the structure of the graph in Figure 5.18b is a mesh, dependencies do not fulfill the requirements of an MMG and consequently the entire graph is not an MMG.

To eliminate bidirectional dependencies, we take advantage of the property of a graph discussed below.

Definition 13 : *Movable subgraph* in a mesh graph is a subgraph where

- all horizontal incoming and outgoing edges carry transmittent data.
- all vertical incoming edges carry transmittent data that is also available as horizontal edges.

Figure 5.18b shows examples of movable subgraphs. Note that transmittent incoming edges in the vertical direction also appear as transmittent data in the horizontal direction.

Lemma 16 *A three-dimensional graph with nearest-neighbor connections and bidirectional dependencies may be transformed into an MMG if, in each mesh along one axis, all nodes at one side of the source of transmittent data are part of a movable subgraph.*

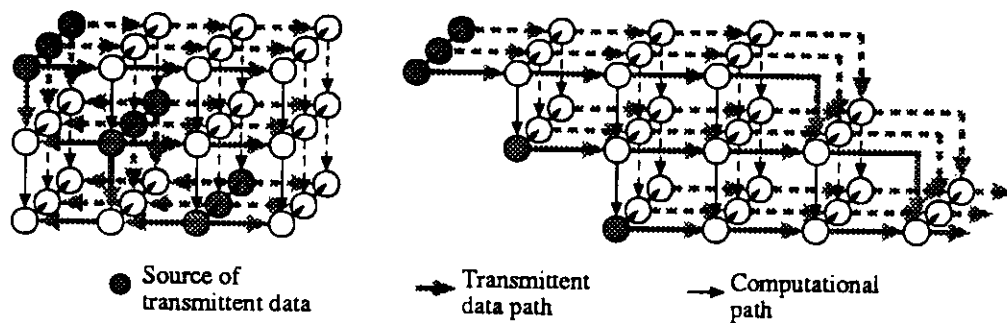


Figure 5.19: Transforming bidirectional transmittent data

Proof: If all nodes at one side of the source of broadcasting belong to movable subgraphs, then the only possible dependencies between those subgraphs arise from transmittent data. By definition, transmittent data is available elsewhere in the graph, in particular at the other side of the source of such a transmittent data. Consequently, it is possible to move the movable subgraphs (hence their name) to the other side of the source of transmittent data so that bidirectional dependencies are eliminated, as depicted in Figure 5.19. ■

5.5.3 Removing non-nearest neighbor dependencies

An MMG has dependencies only between nearest-neighbor nodes. In contrast, a matrix algorithm represented as a three-dimensional graph might have edges between nodes located distant from each other. This problem is solved by applying the following definition and lemma:

Distance between directly dependent nodes $n(i, j, k)$ and $n(p, q, r)$ in a three-dimensional graph is given by:

$$\delta[(i, j, k), (r, s, v)] = (r - i) + (s - j) + (v - k)$$

Since dependencies in MMGs are unidirectional, indices of a destination node have values larger or equal than a corresponding source node, and distance is always greater than zero. Moreover, dependencies in an MMG have $\delta = 1$, that is, nearest-neighbor dependencies.

Lemma 17 *Non-nearest neighbor dependencies in a three-dimensional graph are removed by adding delay nodes.*

Proof: Consider first the case of nodes $n(i, j, k)$ and $n(i, j, l)$ at distance $\delta = r$ in the same mesh of a three-dimensional graph. Consequently, there are $r - 1$ missing nodes in the path from $n(i, j, k)$ to $n(i, j, l)$. If the positions of missing nodes are filled with delay nodes, then the edge $e_{(i,j,k),(i,j,l)}$ becomes the path $[e_{(i,j,k),(i,j,k_1)}, e_{(i,j,k_1),(i,j,k_2)}, \dots, e_{(i,j,k_{r-1}),(i,j,l)}]$. Nodes now have $\delta = 1$. The same process is repeated to fill vacant positions when a dependency traverses different meshes along the remaining dimensions of the graph. By the definition of equivalence, the graph resulting after adding delay nodes is equivalent to the original graph. Since the resulting graph has dependencies with $\delta = 1$, it has been transformed into one with nearest-neighbor dependencies. ■

5.5.4 Synchronizing arrival of data to nodes

The correct execution of a matrix algorithm in an array requires that all operands for a given operation are available in a cell at the right time. That is, data arrival to cells must be synchronized. Since arrays are derived from the multi-mesh graph description of an algorithm, synchronized arrival of data to cells is a result of the distribution of nodes and edges in the three-dimensional space. We discuss such an issue now.

Lemma 18 *Arrival of data to nodes in a multi-mesh dependency graph is synchronized.*

Proof: Nodes in a multi-mesh graph are identified by three indices as $n(i, j, k)$. Assume that node $n(1, 1, 1)$ is executed at time $t_s(1, 1, 1) = 0$ (such a node does not depend on any other node), and that computation time of nodes is τ . Due to the dependencies in the CMMG, the time at which node $n(i, j, k)$ is ready for execution is determined by the distance between this node and node $n(1, 1, 1)$. That is,

$$t_s(i, j, k) = \delta[(i, j, k), (1, 1, 1)]\tau = [(i - 1) + (j - 1) + (k - 1)]\tau$$

because that is the length of the path from $n(1, 1, 1)$ to $n(i, j, k)$ and each node takes time τ .

Consider node $n(r, s, v)$ which, according to the expression above, is ready for execution at time $[(r - 1) + (s - 1) + (v - 1)]\tau$. Assume that node $n(r, s, v)$ is

a predecessor of $n(i, j, k)$ so that there is a path from $n(r, s, v)$ to $n(i, j, k)$. The length of this path is

$$\delta[(r, s, v), (i, j, k)] = (i - r) + (j - s) + (k - v)$$

so that a data element originating from $n(r, s, v)$ arrives at $n(i, j, k)$ at time

$$\begin{aligned} t_{arr}(r, s, v) &= [(r - 1) + (s - 1) + (v - 1)]\tau + [(i - r) + (j - s) + (k - v)]\tau \\ &= [(i - 1) + (j - 1) + (k - 1)]\tau \end{aligned}$$

This value corresponds to the time at which $n(i, j, k)$ is ready for execution. Consequently, all data elements needed for execution of node $n(i, j, k)$ arrive simultaneously (i.e., synchronized) at that node. ■

Lemma 18 above states that, in a multi-mesh graph, data arrives to nodes at the adequate time so that synchronized data arrival is guaranteed. Consequently, removing non-neighbor dependencies according to Lemma 17 also achieves data synchronization in a multi-mesh graph.

5.6 Step 2.1: Transforming the FPG into a three-dimensional graph

We prove now that a *matrix algorithm can always be represented as a three-dimensional data-dependency graph with unidirectional flow of data along axes of such a graph*. This three-dimensional graph is suitable for transformation into an MMG.

First, the existence of such a three-dimensional graph is proven for an algorithm composed only of scalar operations. This case corresponds to unfolding completely the loop-body of the matrix algorithm and ignoring the structure of matrix and vector operators. However, the three-dimensional graph for such a set of scalar operations may contain many delay nodes, so that it is not suitable for implementation in an array. Consequently, after the existence of the three-dimensional graph is proven, we discuss how considering vector/matrix operators as part of the algorithm simplifies drawing the three-dimensional graph and leads to regular representations with few delay nodes. As a result, only the latter part of the formalization is used afterwards as the mechanism to transform an FPG into a three-dimensional graph.

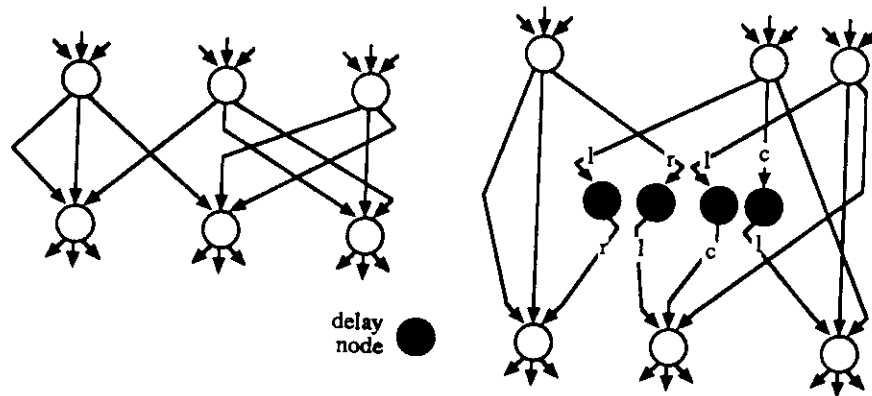


Figure 5.20: Connecting nodes' inputs and outputs that have the same name

Lemma 19 *It is always possible to transform the FPG of a matrix algorithm so that an output from one node goes to an input with the same name (i.e., left output to left input, and so on).*

Proof: Inputs and outputs to/from a node may be ordered in any manner. Let us call $o(n_1)$ and $i(n_2)$ an output from node n_1 that is also an input to node n_2 , respectively, where o, i may be either of $\{left, center, right\}$. Assume that $o(n_1)$ and $i(n_2)$ are different names. In that case, one can add a delay node d between nodes n_1, n_2 , an edge from n_1 to d and another edge from d to n_2 in such a way that

- $o(n_1)$ and $i(d)$ have the same names, where $i(d)$ is the input to node d
- $o(d)$ and $i(n_2)$ have the same names, where $o(d)$ is the output from node d

This process is graphically depicted in Figure 5.20. Consequently, resulting inputs and outputs always have the same name. From Definition 8, the resulting graph is equivalent to the original graph. ■

We refer to edges that go from output to input of the same name as *left-left*, *center-center* and *right-right edges*. Moreover, we draw graphs in such a way that a center-center edge always goes from one node to a node immediately below it, without crossing another center-center edge. In contrast, left-left and right-right edges may cross each other and may also cross center-center edges, as shown in Figure 5.20.

In the remaining of this chapter, unless explicitly stated otherwise, we assume

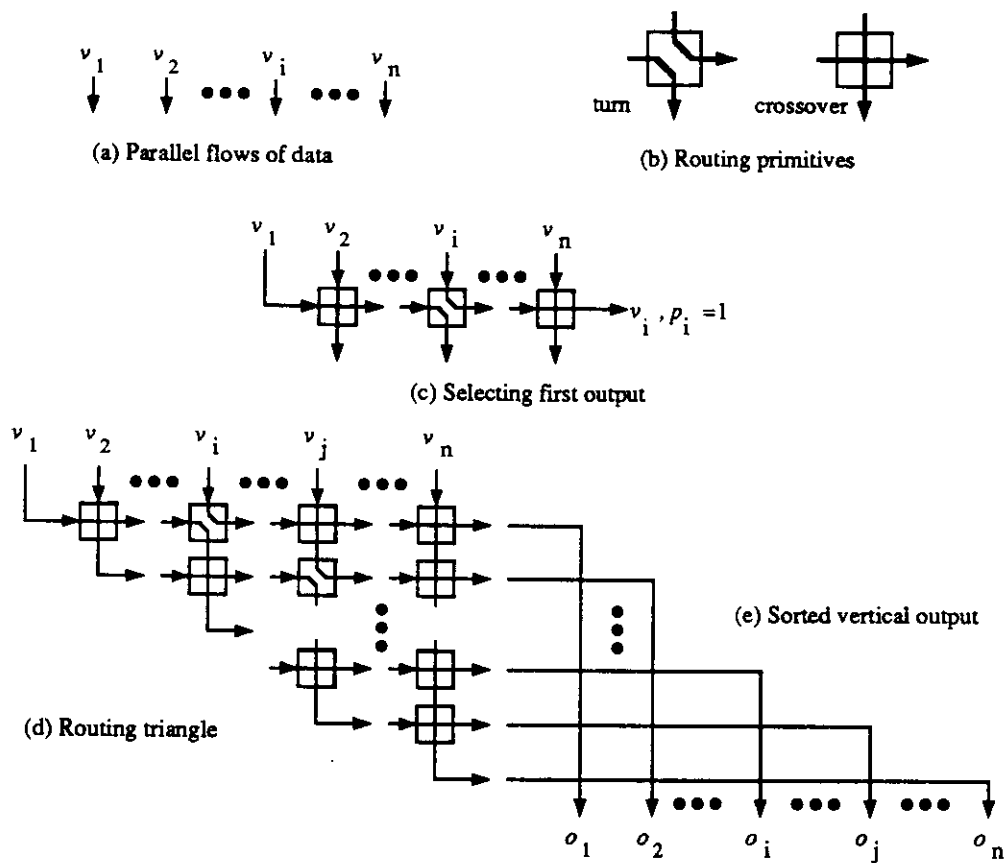


Figure 5.21: Rearranging the flow of data in a plane

that edges go from an output to an input that have the same names. Moreover, we assume that center-center edges do not cross each other.

Lemma 20 *A set of parallel flows of data is sorted in any order in a triangular orthogonal set of intersections with unidirectional data flow.*

Proof: Consider n flows of data along vertical parallel lines v_1, v_2, \dots, v_n of a plane, as shown in Figure 5.21a. Assume also that the desired output order is p_1, p_2, \dots, p_n (i.e., v_1 must appear in position p_1 in the output, and so on), where outputs are numbered from left to right. Moreover, consider two routing primitives: *crossover* and *turn*, as depicted in Figure 5.21b.

We first prove that $n - 1$ routing primitives placed in a row allow extracting data element v_i with $p_i = 1$ from the n data flows. For such purposes, let the leftmost element of the input data (i.e., v_1) flow horizontally until it intersects

with v_i flowing vertically, as depicted in Figure 5.21c. At this intersection, place a *turn* primitive so that v_1 flows vertically and v_i flows horizontally. Let v_i flow horizontally until it reaches the rightmost end of the parallel flows of input data. Moreover, let all remaining data flow vertically by placing *crossover* primitives in their intersection with v_1 or v_i flowing horizontally, as also shown in Figure 5.21c. This process leads to a row of $n - 1$ routing primitives, with data element v_i flowing horizontally at the end of such a row, and the remaining $n - 1$ data elements flowing vertically. In case that $v_i = v_1$, then the $(n - 1)$ routing primitives are crossover.

The process above is now repeated for the remaining $(n - 1)$ vertical flows of data, leading to a row of $(n - 2)$ routing primitives and data v_j (with $p_j = 2$) flowing horizontally, and so on until all data has been sorted in the desired order. The entire process leads to a set of routing primitives interconnected in a triangular unidirectional mesh structure that produces sorted data flowing horizontally, as shown in Figure 5.21d. We refer to such a collection of routing primitives as a *routing triangle*.

Data flowing horizontally can be made to flow vertically again, as illustrated in Figure 5.21e. Consequently, data has been sorted in an arbitrary order by a set of orthogonally connected routing intersections and unidirectional flow of data. ■

It should be noted that adding routing primitives according to Lemma 20 is done while deriving the multi-mesh graph of an algorithm. These routing primitives correspond to intersections in the three-dimensional space, so that they are related to data transfers in an array derived from an MMG. These routing primitives are *not* nodes or operations that need to be executed as part of the algorithm.

Lemma 21 *The fully-parallel dependency graph of a matrix algorithm, whose loop-body has been completely unfolded so that it consists only of scalar operations, has an equivalent three-dimensional graph with unidirectional flow of data only along axes of such a space.*

Proof: Consider the fully-parallel data-dependency graph of a matrix algorithm whose loop-body has been completely unfolded so that it consists only of scalar operations (i.e., we ignore the structure of matrix operators within the loop-body). Consider also the structure (or repetitiveness) among iterations which appears as a result of executing the same loop-body for as many times as the range of the outer loop. Figure 5.22a depicts the type of structure that is found in this case.

The loop-body consists of several levels of nodes, as shown in Figure 5.22a.

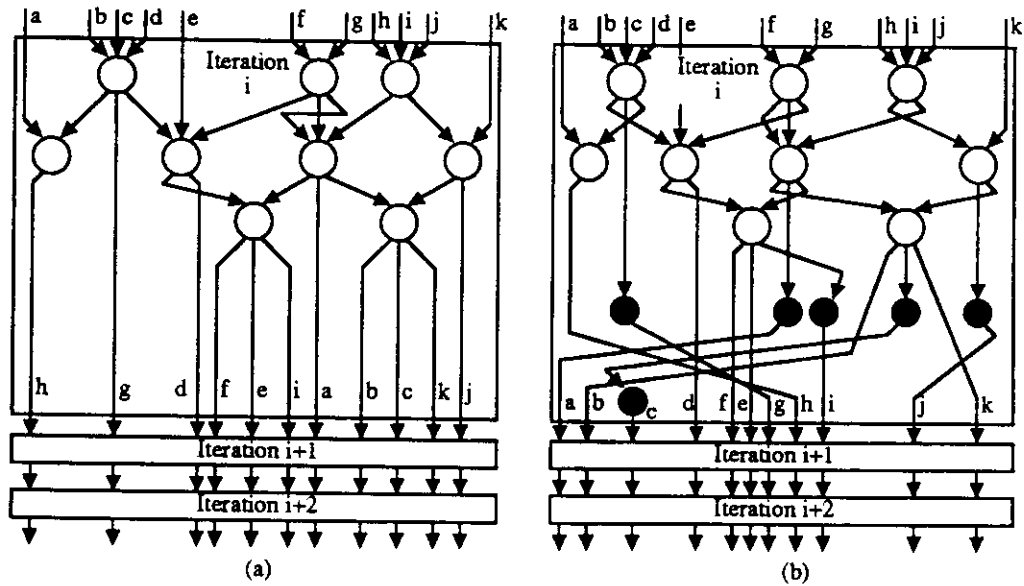


Figure 5.22: Fully-parallel graph of an algorithm with scalar operations

Applying the transformation described in Lemma 19 leads to a graph where nodes are connected by edges with the same names (i.e., left-left, and so on). Moreover, center-center edges do not cross each other. In addition, we consider that all edges between two iterations of the loop body do not intersect (i.e., intersections of edges occur inside the graph of the loop body). These aspects are illustrated in Figure 5.22b, which depicts the loop-body resulting from Figure 5.22a.

Allocate each named input/outputs flows to one axis in a three-dimensional space. That is, allocate left-left edges to flow along the X -axis, center-center edges to flow along the Z -axis, and right-right edges to the Y -axis, as illustrated in Figure 5.23. With this allocation of edges, each level of the graph is represented in the diagonal of a plane where the different flows of data intersect. Consequently, all levels together (i.e., all planes) compose a three-dimensional graph.

Consider now the flow of data from level to level of the graph (i.e., from plane to plane in the three-dimensional space). Outputs from one plane need to be routed towards a new set of intersections (i.e., the diagonal of a new plane), as shown in Figure 5.23. Moreover, outputs from one plane along axes X or Y may need to be routed towards a different position relative to other outputs along such axes (sorted in order) whenever the corresponding edges intersect in the FPG. However, edges along the Z -axis do not need to be sorted because they never intersect.

We use the results from Lemma 20 to achieve the required sorting of edges

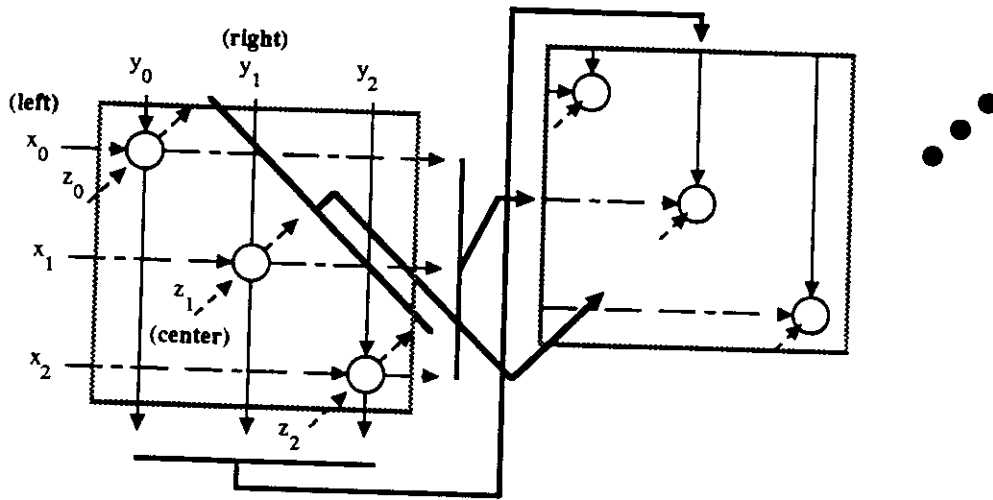


Figure 5.23: Flows of data and graph levels in a three-dimensional space

(i.e., reorder data between levels of the FPG). That is, we add routing triangles along axes X and Y so that edges in each plane are reordered as depicted in Figure 5.24. In this figure, data along the X -axis reaches the diagonal of a plane that contains the operations in one level of the FPG. This X -flow leaves computing nodes *along the Y -axis* in the same plane as operation nodes, and flows downwards until reaching the bottom of the area that contains the operations. At that point, Lemma 20 is applied to rearrange data, so that a triangular mesh of intersections for routing is introduced, and data exits this part flowing horizontally, as also shown in Figure 5.24.

A similar process as the one just described is used for data that reaches operation nodes flowing along axes Y , as also depicted in Figure 5.24.

Outputs from the reordering above are edges flowing along axes X and Y that intersect at the diagonal of the same plane containing the operations originating such outputs (say W_0), as depicted in Figures 5.25a and 5.25b. In contrast, edges flowing along the Z -axis are available in a different plane of the three-dimensional space (say W_1) and in a different (x, y) position, as shown in Figure 5.25c. To intersect the flows of data along the three axes, we make edges along X and Y flow towards an inner plane (say W_2), at the same time that edges along the Z -axis are routed through plane W_1 . That is, upon reaching plane W_1 the Z -flow switches direction towards the right along the X -axis, and then switches flow downwards along the Y -axis, all within plane W_1 . These movements are depicted in Figure 5.25c. When elements of the Z -flow reach their respective position in

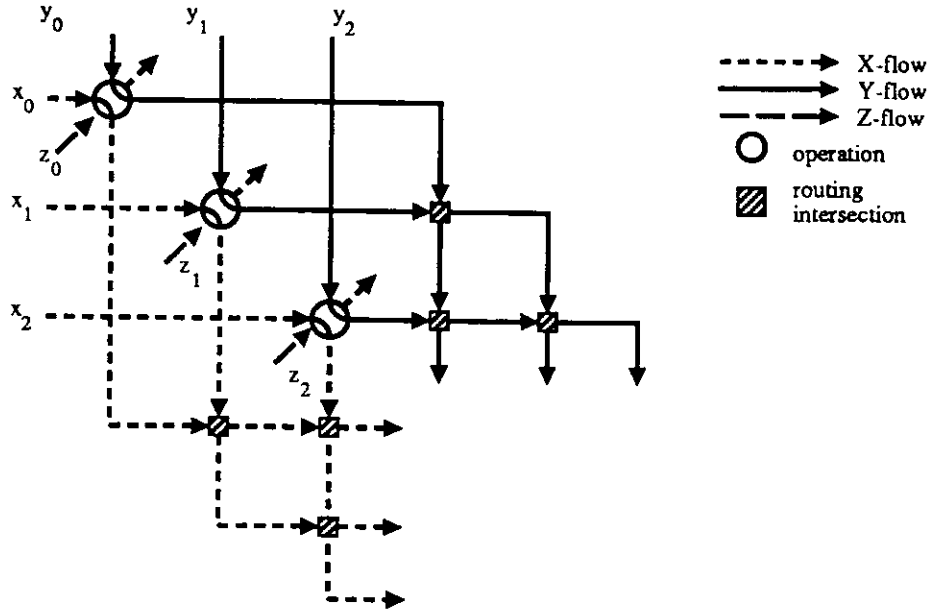


Figure 5.24: Routing data in one plane of the three-dimensional graph

front of the intersection between edges along axes X and Y , they switch direction once more along axis Z to meet the other two flows.

All data movements above are depicted together in Figure 5.26. Consequently, two levels of the graph are represented by three planes of a three-dimensional graph, with unidirectional flow of data only along axes of the three-dimensional space. Moreover, the entire graph is represented by a three-dimensional graph with unidirectional flow of data. ■

The three-dimensional graph in Figure 5.26 has computing nodes only in the diagonal of planes. Moreover, every other plane of the graph is used only for routing. Consequently, operation nodes are not in nearest-neighbor positions. Transforming this three-dimensional graph into a multi-mesh graph by adding delay nodes leads to a graph with more delay nodes than computing nodes, so that it is not suitable for implementation in an array. However, the graph in Figure 5.26 was derived ignoring the structure of matrix and vector operands in the algorithm. In fact, such a graph was obtained assuming only scalar operations. In what follows, we show that matrix and vector operands allow deriving a graph with nearest-neighbor operation nodes which can be converted into an MMG by adding just a few delay nodes.

Lemma 22 *Upon replacing data broadcasting by transmittent data, a vector op-*

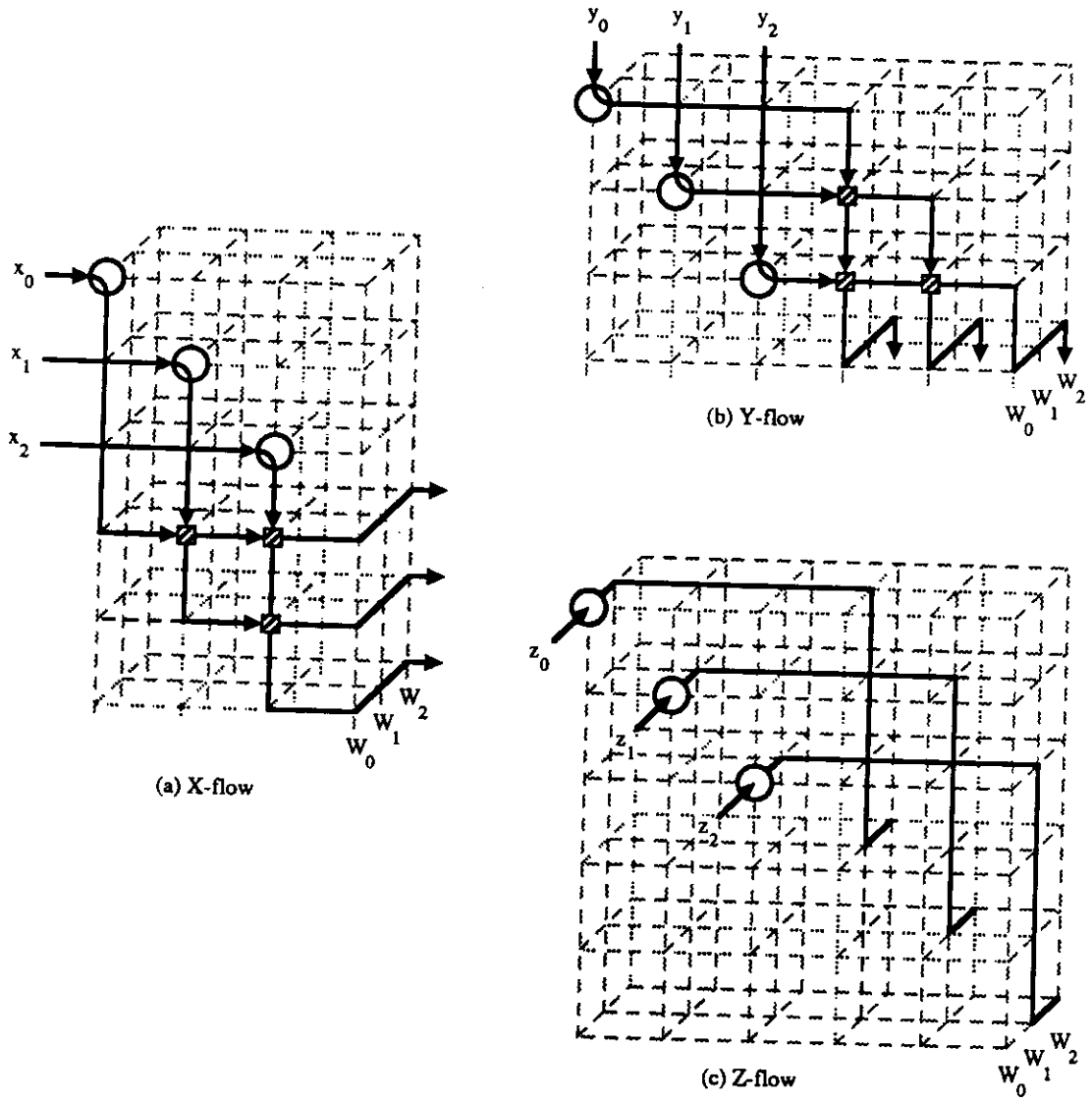


Figure 5.25: Movements of flows of data in the three-dimensional space

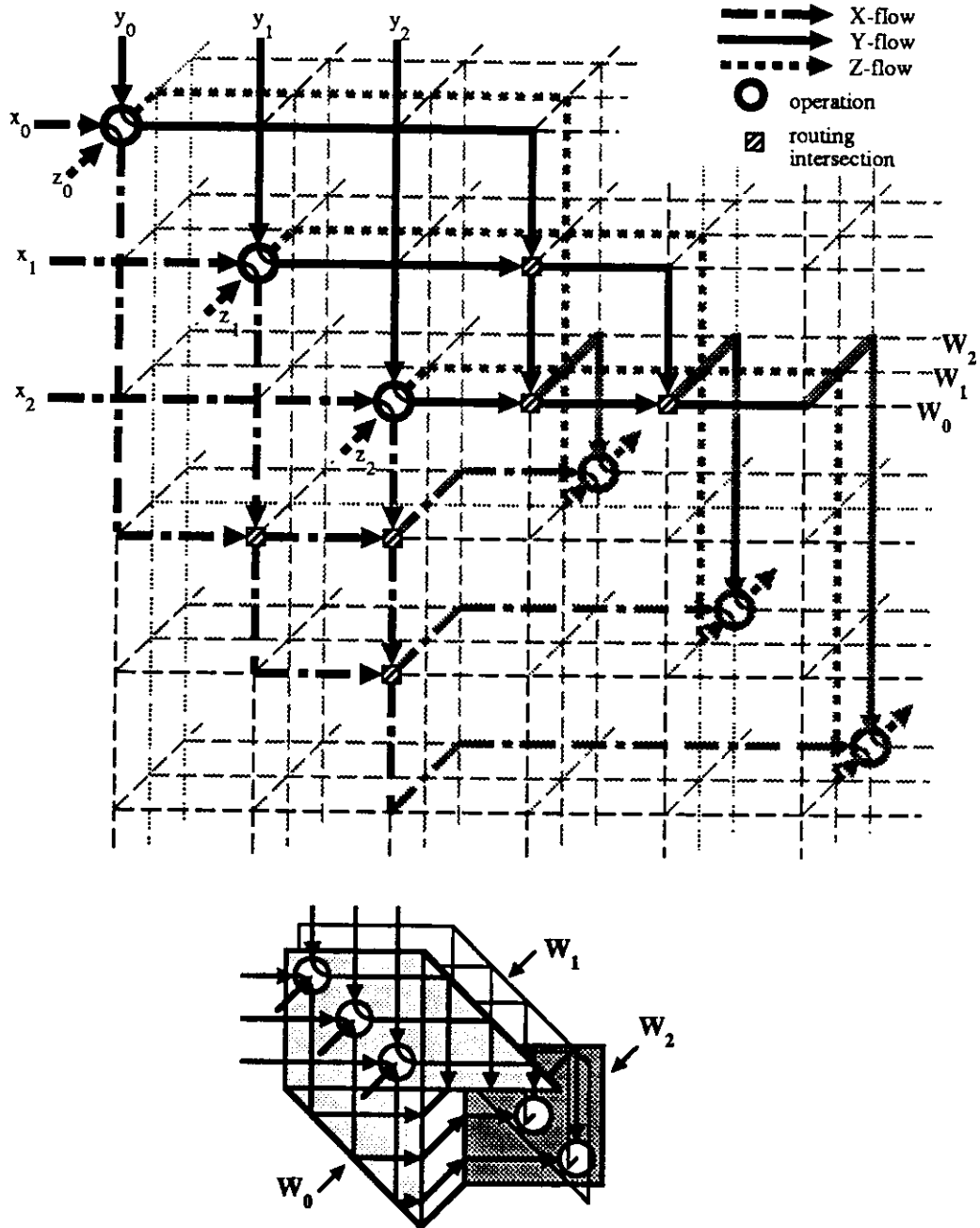


Figure 5.26: Three-dimensional graph for matrix algorithm as scalar operations

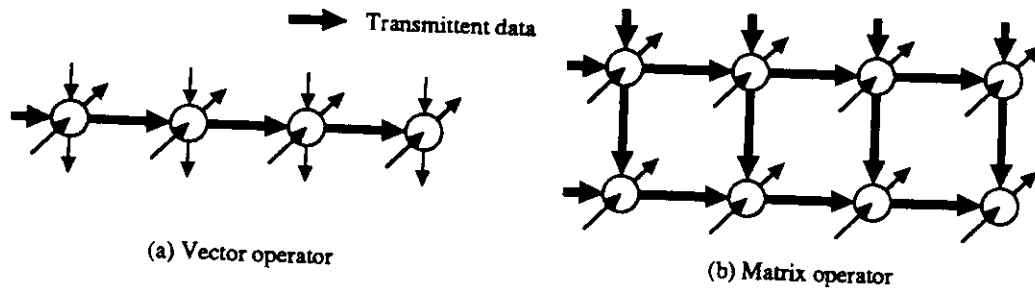


Figure 5.27: Vector and matrix operators

erator is represented as a linear set of nodes distributed along one axis of a plane.

Proof: By definition, a vector operator has up to two independent operands per primitive operator, and possibly one operand common for all instances of the primitive operation. Consequently, the common operand, if it exists, can be made to flow horizontally (or vertically) as transmittent data across primitive nodes organized as a linear structure in a plane, while the independent data values flow along axes Y (or X) and Z . Since one input and one output per node are used for transmittent data, the two remaining input and output are available for two vector operand inputs and two vector operand outputs, respectively. ■

Figure 5.27a depicts a vector operator of size 1 by 4, with transmittent data flowing along the X -axis.

Lemma 23 *After replacing data broadcasting by transmittent data, a matrix operator is represented as a two-dimensional set of nodes.*

Proof: By definition, a matrix operator has one operand common for all instances of the primitive operation on each row of the matrix, and one operand common for all instances of the primitive operation in each column of the matrix. Consequently, data broadcasted in a row can be made to flow horizontally, while data broadcasted in a column can be made to flow vertically, both as transmittent data; primitive nodes are placed at the intersection of these data flows. The resulting structure is a two-dimensional set of nodes. Since two inputs and two outputs per node are used for transmittent data, the remaining input and output along the Z -axis are available for one matrix operand input and one matrix operand output, respectively. ■

Figure 5.27b depicts a matrix operator of size 2 by 4.

All nodes in a vector or matrix operator are located at the same level of the FPG, because they are independent operations. Consequently, the two lemmas above state that primitive nodes of vector and matrix operators are not allocated to the diagonal of a plane in a three-dimensional space as it is the case for scalar operators, but to linear and two-dimensional structures respectively.

In the remaining of this chapter, unless stated otherwise, we assume that vector and matrix operators are represented by linear and two-dimensional sets of nodes, respectively.

From Lemma 21, we infer three interrelated factors that lead to placing nodes from different levels of the FPG at non-neighbor planes in the three-dimensional space:

- primitive nodes allocated to the diagonal of a plane
- the need to route three data flows so that they intersect at the next plane of computing nodes.
- reordering of data elements

However, from Lemmas 22 and 23, these reasons do not exist for matrix and vector operands, as we discuss next. We consider first the more stringent case when a vector operator has two outputs and both outputs are used as inputs to the same operator (vector or matrix) in the next level of the graph. Later, we address the more frequent case of a vector operator with a single output.

Lemma 24 *Consider a vector operator with two outputs that preceds another vector operator (or a matrix operator), and that both outputs from the preceding vector operator are used as input to the succeeding vector (or matrix) operator. These operators are allocated to adjacent planes of a three-dimensional graph if the elements of the two outputs from the preceding vector operator do not need to be reordered for use in the succeeding vector (or matrix) operator.*

Proof: Consider a vector operator in a plane of a three-dimensional graph. Without loss of generality, assume that transmittent data flows along the X -axis so that primitive nodes of the vector operator are placed in a linear structure along this axis, as shown in Figure 5.28. Consequently, data flowing along axes Y and Z correspond to vector operands and vector outputs to/from the preceding vector operator. Since, by hypothesis, both outputs from the preceding vector are not

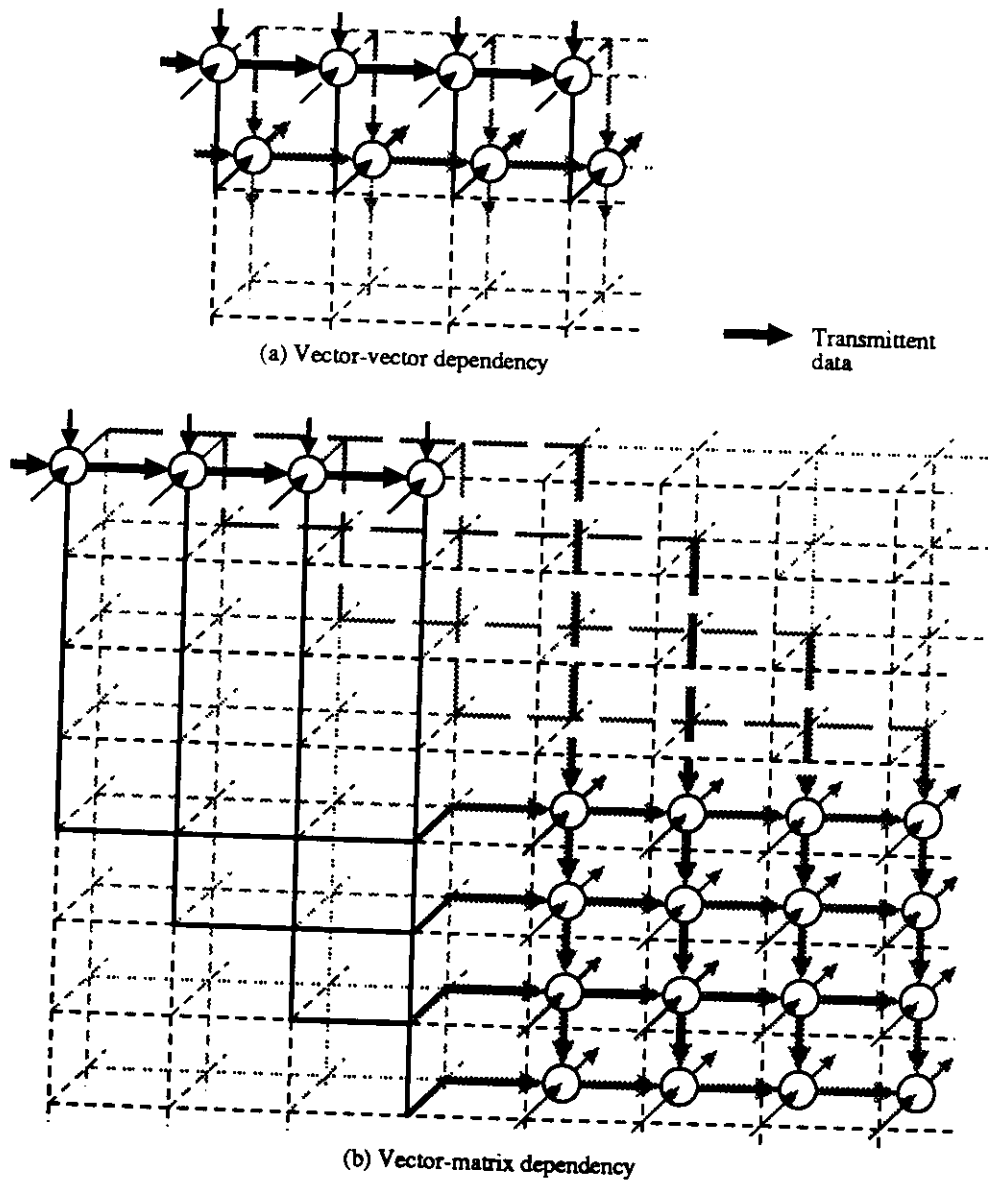


Figure 5.28: Vector and matrix operators in neighbor planes

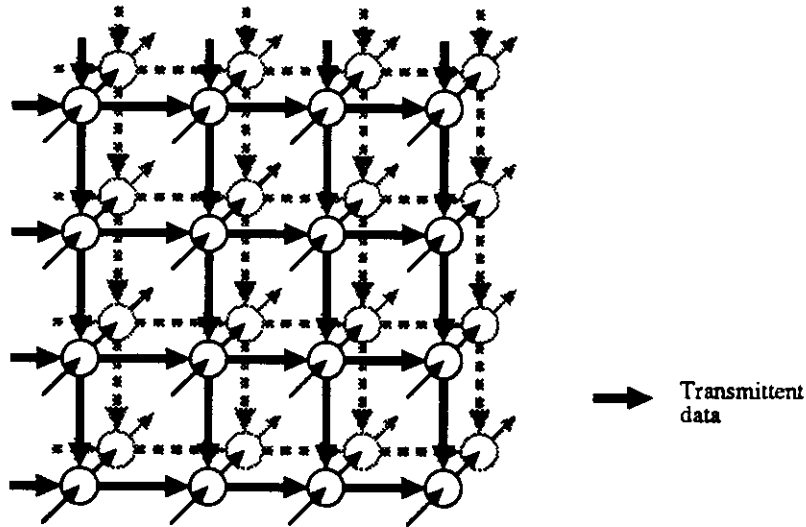


Figure 5.29: Directly dependent matrix operators in neighbor planes

reordered, they can be used as input to another vector operation (or to a matrix operation) placed in an adjacent plane of the three-dimensional space, as depicted in Figure 5.28. ■

Note that in the case of vector-vector dependencies, data leaving along axis Z (Y) from the first vector operator becomes a flow along axis Y (Z) when reaching the second operator. Similarly, in the case of vector-matrix dependencies, the change of flow is Z to Y and Y to X . Moreover, note that in the case of vector-matrix dependencies the graph requires space for changing direction in the same planes as operation nodes.

Lemma 25 *Two directly dependent matrix operations can be allocated to adjacent planes of a three-dimensional graph if elements of the output matrix from the preceding matrix operator are not reordered before being used as input to the succeeding matrix operator.*

Proof: Consider a matrix operator in a plane of a three-dimensional graph. The single matrix operand input and output flows along the Z -axis. Since elements of the output matrix operand are not reordered between the two levels, the succeeding matrix operator can be allocated to an adjacent plane of the three-dimensional space in such a way that the two matrix operators are aligned. This arrangement is depicted in Figure 5.29. ■

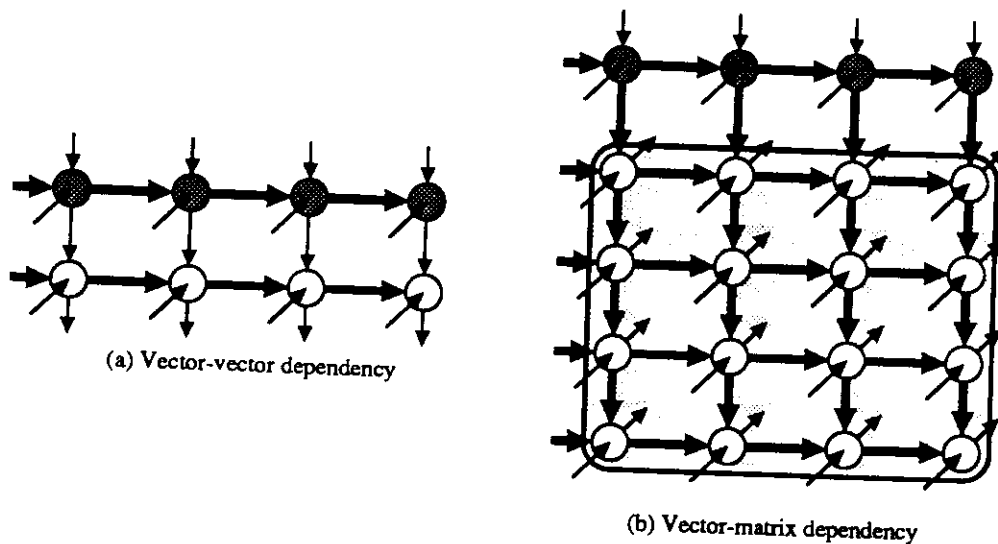


Figure 5.30: Allocating directly dependent operators to the same plane

We consider now the case of a *vector operator with a single output* that is used as input to a succeeding vector or matrix operator. This is the common case that appears in matrix algorithms of interest, including LU-decomposition, QR-decomposition, the Faddeev algorithm and Cholesky decomposition, among others.

Lemma 26 *Consider a vector operator with a single output that precedes another vector operator (or a matrix operator). These operators are allocated to the same plane of a three-dimensional graph if the elements of the output from the preceding vector operator do not need to be reordered for use in the succeeding vector (or matrix) operator.*

Proof: Assume that the predecessor vector operand is allocated to a linear set of nodes in one plane of the three-dimensional space. Moreover, assume that transmittent data through this operator flows along the X -axis. The single vector output produced by this operator can be delivered along axis Y in the same plane as operation nodes, as depicted in Figure 5.30. Such an output correspond to the input to the next vector operator, which is placed directly below the first vector operator, in the same plane. The same situation occurs for a vector operator that precedes a matrix operator, as also shown in Figure 5.30. (Note that the matrix operator has a second vector input that comes from a different source). ■

In this section, we have analyzed the representation of a matrix algorithm by a multi-mesh data-dependency graph. Such a graph is derived from the fully-parallel data-dependency graph. We have proved the existence of the three-dimensional representation. Moreover, we have also proved that two directly dependent levels of the FPG, composed of a vector with a single output and a succeeding vector or matrix operator, are allocated to the same plane in the three-dimensional space as long as the output elements from the first vector operator are not reordered between the two levels. Similarly, two directly dependent levels of the FPG composed of matrix operators are allocated to adjacent planes in the three-dimensional space, if the output elements from the first matrix operator are not reordered between the two levels.

Consequently, the three-dimensional graph obtained from an FPG has most of its operation nodes in nearest neighbor positions. Moreover, such a graph is transformed into an MMG by applying the necessary transformations from those presented in the previous section.

5.7 Step 1: Deriving the FPG of a matrix algorithm

The only step of our method that remains left in this formalization is deriving the FPG of a matrix algorithm. We address this issue now.

Lemma 27 *Any matrix algorithm can be represented by an FPG. Moreover, such a representation is unique for a given algorithm.*

Proof: Any matrix algorithm can be executed symbolically, that is, executed to determine what operations are performed on which variables. The output of that execution is an ordered list of operations with up to three operands and up to three results each. This ordered list can be traversed and mapped onto a graph as follows:

- Assign each operator to a node. Such a node has at most three incoming and three outgoing edges, since each operation has at most three operands and produces at most three results.
- Map references to variables in the algorithm to edges of the graph. Such edges go from the node representing the latest evaluation of a variable to the node using the variable.

The resulting graph is the fully-parallel data-dependency graph describing the algorithm. Since there is a one-to-one correspondence between operations of the algorithm and nodes of the graph, and between data dependencies and edges, the FPG uniquely describes the algorithm. ■

The FPG corresponds to a single assignment description of an algorithm. From the proof to Lemma 27 above, nodes of the FPG are generated when variables in the algorithm are created and every time that those variables are updated. New nodes to update a variable correspond to the generation of new variables. Since these new variables are generated on demand (i.e., as needed by the symbolic execution of the algorithm), only the minimum number of extra variables is generated.

Lemma 28 *The FPG of a matrix algorithm consists of a sequence of subgraphs with the same dependency structure, but potentially different number of nodes and/or edges in each subgraph. Each subgraph corresponds to one iteration of the outer loop in the algorithm.*

Proof: According to the canonical representation, a matrix algorithm consists of an outer loop and a loop-body which may recursively contain other matrix algorithms nested to any level. As stated in Lemma 27, the FPG is obtained by symbolic execution. This execution implies unfolding the outer loop, which in turn implies that the body of the loop is replicated as many times as the range of such a loop. Since the loop-body is fixed, all replications of the body have the same dependency structure, though the number of nodes and edges may change depending on the size of vector and matrix operands in each instance of the loop-body. Each replication of the loop-body can be regarded as being a subgraph, so that the entire FPG consists of a sequence of subgraphs with similar structure. ■

The two lemmas above provide necessary elements to draw the FPG of a matrix algorithm. From Lemma 27, symbolic execution of a matrix algorithm leads to a sequence of expressions which are used to draw the FPG. Moreover, from Lemma 28 we know that the FPG consists of a sequence of subgraphs with similar dependency structure. This structure is given by the presence of vector and matrix operators, so that one can aid the process of drawing the FPG by using the knowledge regarding the existence of those operators. In other words, the presence of a vector operator leads to a set of nodes arranged in a linear structure, while a matrix operator leads to nodes in a two-dimensional arrangement.

An example of the issues discussed above is shown in Figure 5.31, which depicts the FPG for LU-decomposition without pivoting of a 4 by 4 matrix A . Note that

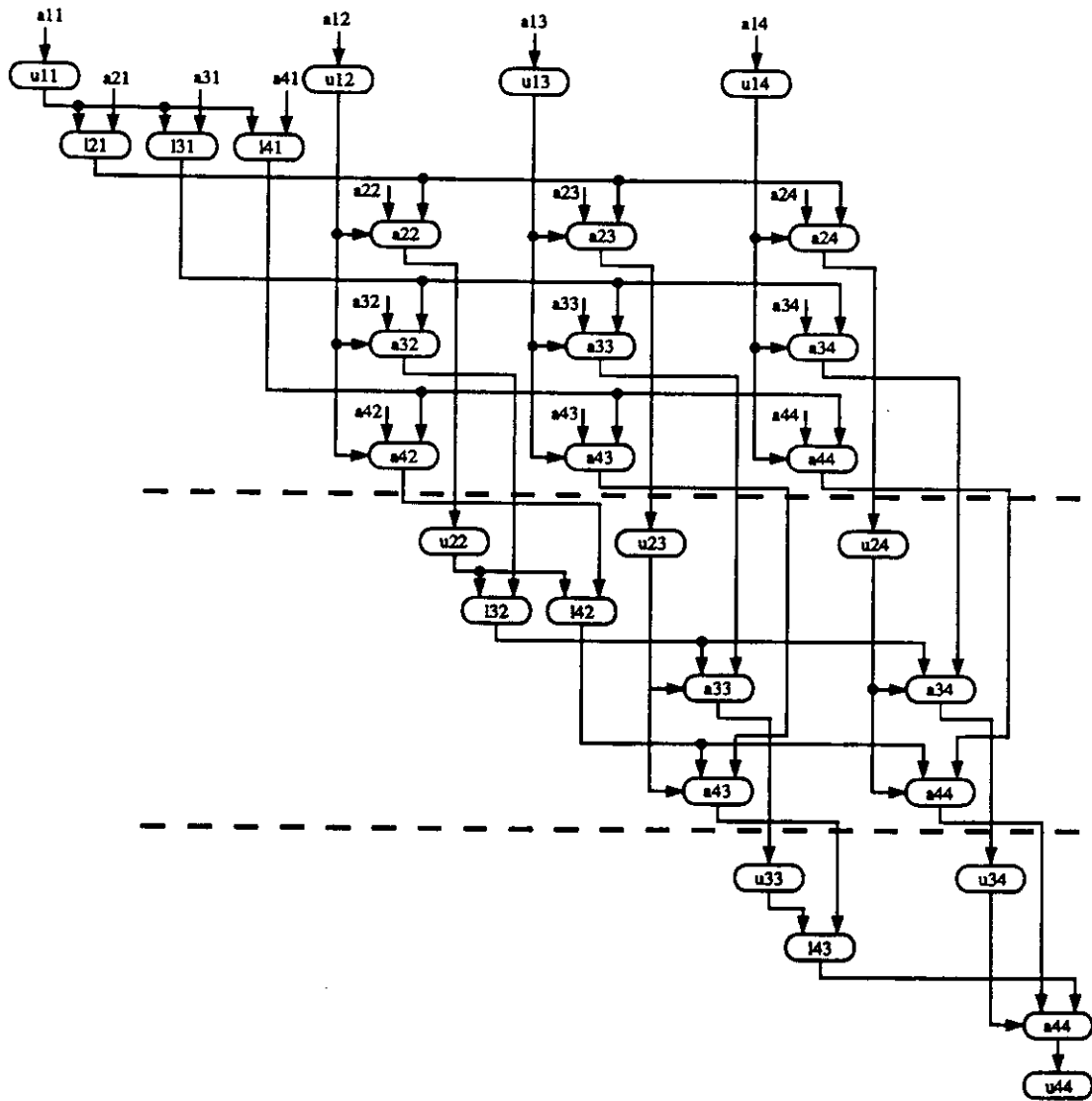


Figure 5.31: Fully-parallel data-dependency graph for the LU-decomposition

there are three subgraphs, each of them consisting of a set of nodes distributed in a linear structure (vector operators computing values of l_{ij} and u_{ij}), and a set of nodes in a two-dimensional structure (matrix operator updating values of a_{ij}). Moreover, each subgraph consists of levels that have vector operators with a single output preceding a matrix operator. Such subgraphs are allocated to one plane of a three-dimensional graph, according to the discussion in the previous section.

5.8 Deriving arrays for matrix computations

We have presented the components of a method for the realization of arrays for matrix computations. Such components can be combined in the following theorem:

Theorem 1 *A matrix algorithm is perfectly realized as an array.*

Proof: Lemma 27 stated that is possible to derive the FPG of a matrix algorithm by symbolic execution of such an algorithm. Moreover, this graph consists of a sequence of subgraphs with similar structure, as indicated by Lemma 28. Lemma 21 and Lemmas 15 through 17 allow transforming the FPG into an MMG. The resulting MMG can be reduced into an MG, as indicated in Lemma 4. This transformation is carried out taking into account issues such as performance, local storage per cell, cell bandwidth, cell utilization, and cell complexity, as stated in Lemmas 7 through 14. Finally, the MG can be perfectly realized as an array by Lemma 1. ■

5.9 The procedure to derive arrays for matrix algorithms

From the formalization in this chapter, we can extract the following procedure to design an array for a matrix algorithm:

1. Obtain the fully-parallel data-dependency graph (FPG) (Lemmas 27,28).
2. Transform the FPG into a multi-mesh data-dependency graph. This implies:
 - (a) Replace data broadcasting by transmittent data (Lemma 15).
 - (b) Draw the graph in a three-dimensional space (Lemma 21–26).
 - (c) Eliminate bidirectional transmittent data (Lemma 16).

- (d) Remove non nearest-neighbor dependencies (Lemma 17).

The resulting MMG corresponds to a *regularized* description of the matrix algorithm.

3. Collapse the resulting MMG onto an MG by performing the procedure grouping by-prisms (Lemma 4). This procedure is carried out taking into account implementation constraints (such as local storage per cell, cell bandwidth, cell pipelining) and the impact on performance (i.e., utilization of cells) arising from collapsing (Lemmas 5–14). Consequently, this step requires choices that determine characteristics of cells in an array and performance of the implementation.
4. The resulting MG is perfectly mapped onto an array by allocating each G-node from the MG to a cell, and each edge to a link in the array (Lemma 1).

5.10 Partitioning

The previous sections in this chapter have presented a formalization of our data-dependency graph-based method for the design of mesh arrays, which has addressed the derivation of structures for problems with fixed-data. The formalization of partitioning is an extension to the case of fixed-size data. In particular, the derivation of the MMG and the process of collapsing the MMG onto a G-graph are identical in both cases, as well as the determination of the schedule of primitive nodes within a G-node.

The last steps in the second stage of our method, namely the selection of G-sets and the process of scheduling G-sets, does not exist in the case of problems for fixed-size data. However, these steps are not that different from the process of grouping primitive nodes of the MMG into G-nodes. In other words, selecting and scheduling G-sets can be regarded as transforming the G-graph into a new graph that consists of as many nodes as there are cells in the array. These new nodes are composed of G-nodes. Consequently, the corresponding formalization follows trivially from the discussion in this chapter, and therefore is not included here.

<pre> For $k = 1$ to n $u_{kk} = 1/a_{kk}$ For $j = (k + 1)$ to n $u_{kj} = a_{kj}$ For $i = (k + 1)$ to n $l_{ik} = a_{ik} * u_{kk}$ For $i = (k + 1)$ to n For $j = (k + 1)$ to n $a_{ij} = a_{ij} - l_{ik} * u_{kj}$ </pre>	<pre> $n = 4$ $u_{1,1} = 1/a_{1,1}$ $u_{1,2} = a_{1,2}; u_{1,3} = a_{1,3}; \dots$ $l_{2,1} = a_{2,1} * u_{1,1}; l_{3,1} = \dots$ $a_{2,2} = a_{2,2} - l_{2,1} * u_{1,2}$ $a_{2,3} = a_{2,3} - l_{2,1} * u_{1,3}$ $a_{2,4} = a_{2,4} - l_{2,1} * u_{1,4}$ $a_{3,2} = a_{3,2} - l_{3,1} * u_{1,2}$ $a_{3,3} = a_{3,3} - l_{3,1} * u_{1,3}$ \vdots </pre>
(a) Algorithm	(b) Symbolic execution

Figure 5.32: The LU-decomposition algorithm

5.11 The application of the method to the LU-decomposition

We illustrate now the formalism for problems with fixed-size data developed in this chapter by applying it to the derivation and evaluation of arrays for the LU-decomposition algorithm. Additional examples are given in the appendices.

5.11.1 Deriving the FPG

The LU-decomposition algorithm is shown in Figure 5.32a. By Lemma 27, This algorithm can be executed symbolically, leading to a list of expressions as the one depicted in Figure 5.32b. The FPG constructed from this list was shown in Figure 5.31. As stated before, this graph consists of several sections that have the same dependency structure although different number of nodes per section (Lemma 28).

5.11.2 Deriving the MMG

The FPG in Figure 5.31 is transformed into an MMG by removing data broadcasting (Lemma 15) and by drawing the graph in a three-dimensional space (Lemma 21). The body of the outer loop in the algorithm in Figure 5.32a consists of two vector operators, with a single output each, that are succeeded by a matrix opera-

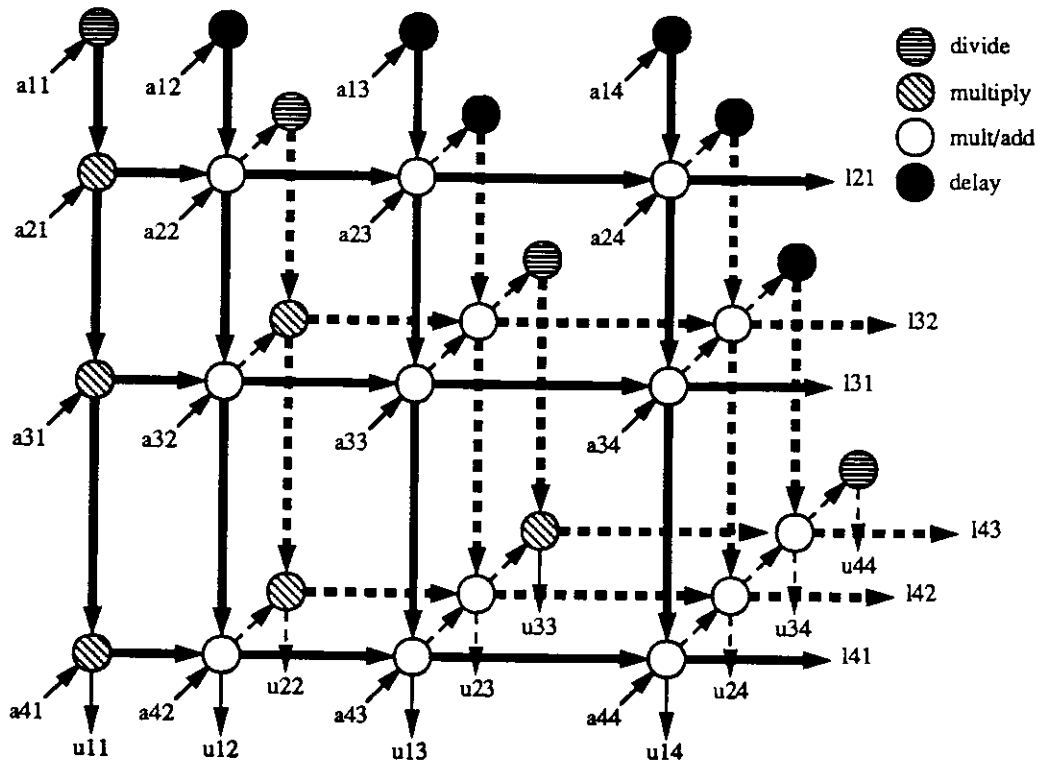


Figure 5.33: Three-dimensional graph for the LU-decomposition algorithm

tor that uses the output from both vector operators. Each iteration of the loop is allocated to a single plane in the three-dimensional space (Lemma 26), and the entire algorithm is represented by the three-dimensional graph shown in Figure 5.33 for a matrix of size 4 by 4.

5.11.3 Deriving the MGs

The MMG shown in Figure 5.33 can be collapsed onto two-dimensional G-graphs by grouping primitive nodes into G-nodes (Lemma 4). The G-graphs resulting from grouping along the three axes are shown in Figure 5.34, where, for simplicity in the drawing, we have assumed prisms of base size 1 by 1.

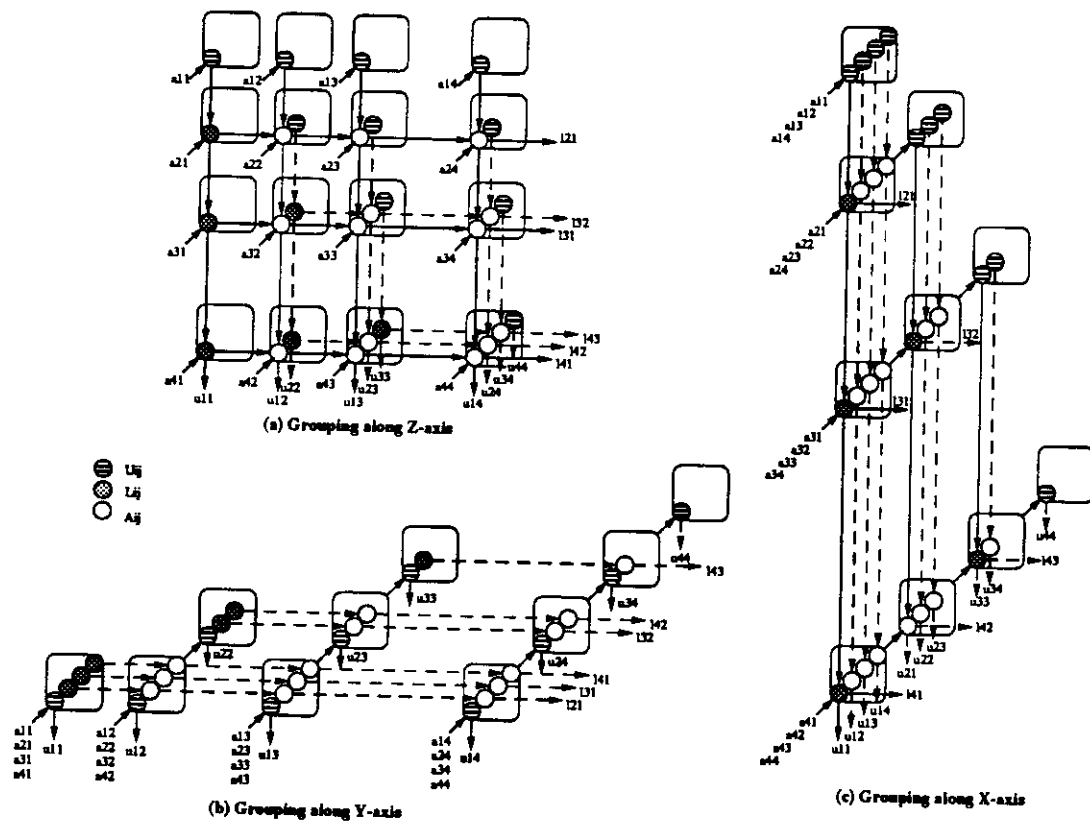


Figure 5.34: G-graphs for the LU-decomposition algorithm

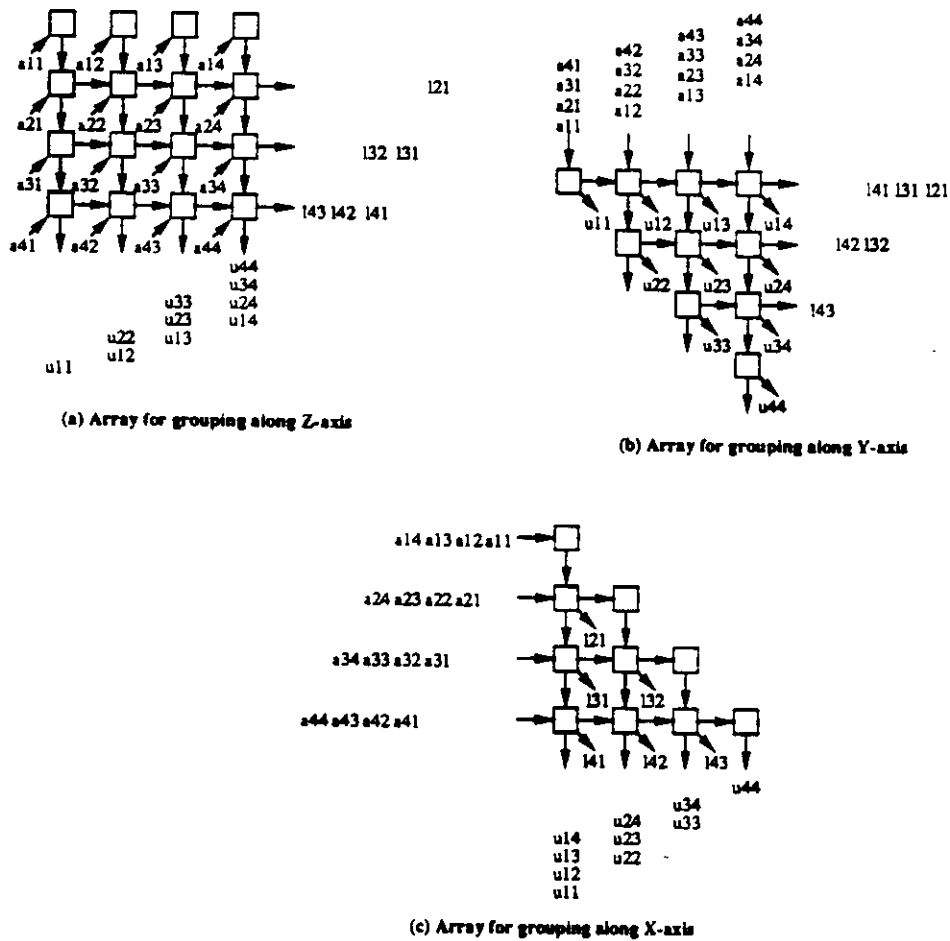


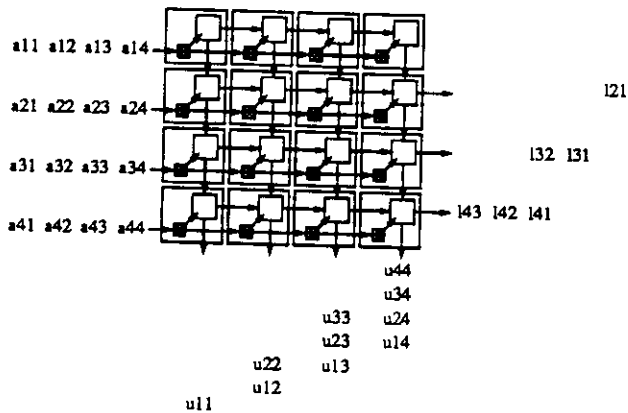
Figure 5.35: Arrays for computing the LU-decomposition with fixed-size data

5.11.4 Realizing the G-graphs as two-dimensional arrays

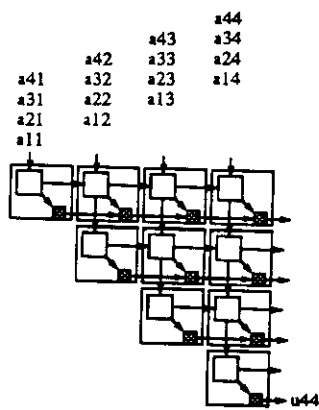
For problems with fixed-size data, G-graphs as those obtained in Figure 5.34 are directly realized as two-dimensional arrays³ (Lemma 1). The resulting arrays are shown in Figure 5.35, which correspond to square and triangular structures.

Note that inputs and outputs to the three arrays exhibit different characteristics. For example, in the square array obtained from grouping along the Z -axis all cells require external inputs, while results are available at the lower and rightmost boundaries. In contrast, the triangular array obtained from grouping along the Y -axis has data input only at topmost cells, while matrix U is left inside the array

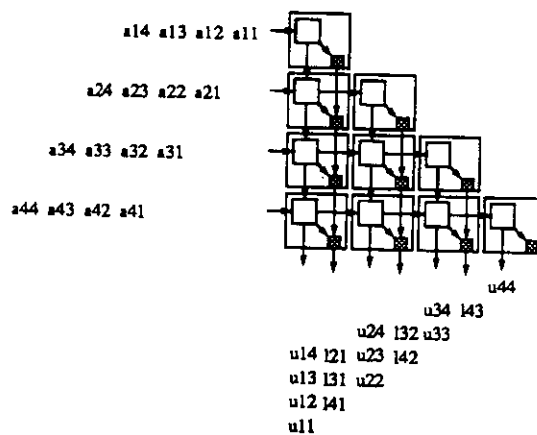
³Recall that the realization of linear arrays is treated as a case of partitioning an algorithm, so that is not discussed in this Chapter.



(a) Array for grouping along Z-axis



(b) Array for grouping along Y-axis



(c) Array for grouping along X-axis

Figure 5.36: Decoupling I/O from computation in the LU-decomposition

and matrix L appears at the right boundary. Finally, the array derived by grouping along the X -axis exhibits external data input only to leftmost cells, matrix L is left inside the array, and matrix U appears at the bottom cells.

Delivering data to inner cells or reading results from inner cells of an array can be implemented as discussed in Section 4.11, that is, by introducing an additional path between cells that is used only for transferring data. This is also similar to the I/O structures for partitioned implementations described in Section 4.6. These I/O structures are depicted in Figure 5.36.

5.11.5 Evaluation of the arrays

We discuss now the performance that is obtained in the arrays for computing the LU-decomposition. For such purposes, we use number of cells, throughput and utilization as performance and cost measures, and obtain expressions for such measures in terms of the prisms sizes (i.e., p, q and prisms length). Measures related to cell characteristics (such as local storage and cell bandwidth) depend on p and q only, so that they are identical for groupings along the different axes.

5.11.5.1 Computational load

The computational load imposed by the LU-decomposition algorithm is obtained from the multi-mesh dependency graph shown in Figure 5.33. This graph is composed of n dependent meshes. The number of operations in the algorithm is given by

$$N = \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

because there are i^2 operations in mesh i (when traversing meshes from the innermost to outermost along axis Z).

5.11.5.2 Grouping along the Z -axis

Grouping along the Z -axis leads to the two-dimensional G-graph shown in Figure 5.34a, and consequently to the square array shown in Figure 5.35a. The number of cells in such an array is

$$K_f^Z = \left\lceil \frac{n}{p} \right\rceil \left\lceil \frac{n}{q} \right\rceil = \frac{n^2}{pq}$$

if $(n/p), (n/q)$ are integer values.

Throughput of this array is determined by the bottom rightmost cell, because such a cell computes more primitive operations than other cells, as is inferred from the G-graph. The corresponding prism, shown in Figure 5.37, includes p by q paths along the Z -axis that do not have the same length. In fact, the length of

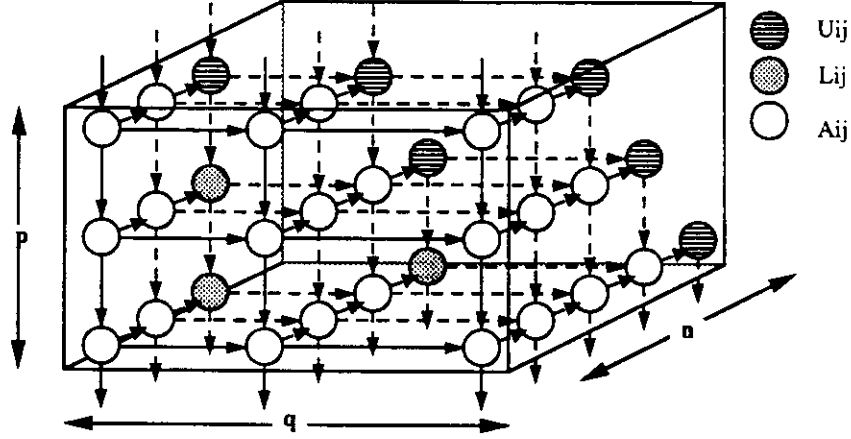


Figure 5.37: Prism of primitive nodes along the Z -axis

paths at the bottom of the prism increases from $(n - q + 1)$ up to n , as depicted in Figure 5.37. In contrast, the length of paths next to the bottom of the prism increases from $(n - q + 1)$ up to $(n - 1)$, and there are two paths with such a final length. A similar pattern appears in other parts of the prism, up to the top of the prism where paths length varies from $(n - q + 1)$ up to $(n - p + 1)$ and there are p paths with the final length. Consequently, throughput is given by

$$\begin{aligned}
 T_Z^{-1} = t_Z^{max} &= \sum_{i=1}^p \sum_{j=1}^q (\text{nodes along } Z) \\
 &= [(n - q + 1) + (n - q + 2) + \dots + (n - 1) + n] \\
 &\quad + [(n - q + 1) + (n - q + 2) + \dots + 2(n - 1)] \\
 &\quad + [(n - q + 1) + (n - q + 2) + \dots + 3(n - 2)] \\
 &\quad \vdots \\
 &\quad + [(n - q + 1) + (n - q + 2) + \dots + p(n - p + 1)] \\
 &= \sum_{i=1}^p \left[\sum_{j=n-q+1}^n j - \sum_{j=1}^{i-1} j \right] \\
 &= \sum_{i=1}^p \left[\frac{n(n+1)}{2} - \frac{(n-q)(n-q+1)}{2} - \frac{(i-1)i}{2} \right] \\
 &= \frac{p}{2} \left[n(n+1) - (n-q)(n-q+1) - \frac{1}{3}(p+1)(p-1) \right] \\
 &= \frac{p}{2} \left[2nq - q^2 + q - \frac{1}{3}(p+1)(p-1) \right] \\
 &= \frac{p}{6} [3q(2n - q + 1) - (p+1)(p-1)]
 \end{aligned}$$

For large n , we obtain

$$T_Z^{-1} = t_Z^{max} \rightarrow \frac{p}{6}(3q)(2n) = pqn$$

Utilization is

$$\begin{aligned} U_Z &= \frac{N}{K_f^Z T_Z^{-1}} = \frac{N}{K_f^Z t_Z^{max}} \\ &= \frac{\frac{1}{6}n(n+1)(2n+1)}{\frac{n}{p} \frac{n}{q} \frac{p}{6} [3q(2n-q+1) - (p+1)(p-1)]} \\ &= \frac{q(n+1)(2n+1)}{3nq(2n-q+1) - n(p+1)(p-1)} \end{aligned}$$

For large n , this becomes

$$U_Z \rightarrow \frac{qn(2n)}{3nq(2n)} = \frac{1}{3}$$

5.11.5.3 Grouping along Y -axis

The number of G-nodes (and consequently number of cells) obtained by grouping along the Y -axis in Figure 5.33 is computed as

$$K_f^Y = \sum_{i=1}^{\lceil n/p \rceil} \lceil ip/q \rceil$$

because the triangular G-graph in Figure 5.34b has $\lceil n/p \rceil$ horizontal paths and each path has $\lceil ip/q \rceil$ G-nodes (where i is the path number from inner to outer path in the figure).

Assuming that p/q and n/p are integers, the expression for number of cells above is written as

$$K_f^Y = \sum_{i=1}^{n/p} (p/q)i = \frac{p}{q} \frac{n}{p} \left(\frac{n}{p} + 1 \right) / 2 = \frac{n(n+p)}{2pq}$$

Throughput of the resulting array, shown in Figure 5.35b, is determined by the computation time of the top-rightmost cell. Assuming that $(n-p) \geq n/2$, $(n-q) \geq n/2$, this computation time is

$$t_Y^{max} = pqn - \sum_{i=1}^p q(i-1) = pq(2n - p + 1)/2$$

because vertical paths of primitive nodes enclosed in the corresponding prism have length n for the outermost q paths, but the remaining paths have decreasing length, as inferred from Figure 5.33.

Utilization is

$$\begin{aligned} U_Y &= \frac{N}{K_f^Y T_Y^{-1}} = \frac{N}{K_f^Y t_Y^{max}} \\ &= \frac{n(n+1)(2n+1)/6}{(n/2pq)(n+p)(pq/2)(2n-p+1)} \\ &= \frac{2(n+1)(2n+1)}{3(n+p)(2n-p+1)} \end{aligned}$$

For large n ,

$$U_Y \rightarrow \frac{4n^2}{6n^2} = \frac{2}{3}$$

5.11.5.4 Grouping along X -axis

The number of G-nodes obtained by grouping along the X -axis is given by

$$K_f^X = \sum_{i=1}^{\lceil n/q \rceil} \lceil iq/p \rceil$$

because the triangular G-graph in Figure 5.34c has $\lceil n/q \rceil$ vertical paths and each path has $\lceil iq/p \rceil$ G-nodes (where i is the path number from right to left in the figure).

Assuming that q/p and n/q are integers, the expression for number of cells above is written as

$$K_f^X = \sum_{i=1}^{n/q} (q/p)i = \frac{q}{p} \frac{n}{q} \left(\frac{n}{q} + 1 \right) / 2 = \frac{n(n+q)}{2pq}$$

Throughput of the resulting array, shown in Figure 5.35c, is determined by the computation time of the bottom-leftmost cell. Assuming that $(n-p) \geq n/2$, $(n-q) \geq n/2$, this computation time is

$$t_X^{max} = pqn - \sum_{i=1}^p q(i-1) = pq(2n - p + 1)/2$$

Utilization is

$$\begin{aligned} U_X &= \frac{N}{K_f^X T_X^{-1}} = \frac{N}{K_f^X t_X^{max}} \\ &= \frac{n(n+1)(2n+1)/6}{(n/2pq)(n+q)(pq/2)(2n-p+1)} \\ &= \frac{2(n+1)(2n+1)}{3(n+q)(2n-p+1)} \end{aligned}$$

For large n ,

$$U_X \rightarrow \frac{4n^2}{6n^2} = \frac{2}{3}$$

5.11.5.5 Summary of evaluation measures

Table 5.1 summarizes the expressions obtained in evaluating the arrays for computing the LU-decomposition algorithm. This table indicates that groupings along the X or Y axes are more advantageous than grouping along the Z -axis, because they provide better utilization and the same throughput. Such a conclusion could also be inferred directly from the MMG, due to the incompleteness of the graph.

Table 5.1: Summary of evaluation measures for LU-decomposition

Measures	Assumptions	
Z-axis:		
Number of cells	n^2/pq	
(Throughput) ⁻¹	$\frac{3pq(2n-q+1)-p(p+1)(p-1)}{6}$	→ pqn
Utilization	$\frac{q(n+1)(2n+1)}{3nq(2n-q+1)-n(p+1)(p-1)}$	→ $\frac{1}{3}$
Y-axis:		
Number of cells	$n(n+p)/2pq$	$n/p, p/q$ integers
(Throughput) ⁻¹	$\frac{1}{2}pq(2n-p+1)$	→ pqn $n-p \geq n/2,$ $n-q \geq n/2$
Utilization	$\frac{2(n+1)(2n+1)}{3(n+p)(2n-p+1)}$	→ $\frac{2}{3}$
X-axis:		
Number of cells	$n(n+q)/2pq$	$n/q, q/p$ integers
(Throughput) ⁻¹	$\frac{1}{2}pq(2n-p+1)$	→ pqn $n-p \geq n/2,$ $n-q \geq n/2$
Utilization	$\frac{2(n+1)(2n+1)}{3(n+q)(2n-p+1)}$	→ $\frac{2}{3}$

CHAPTER 6

Mapping algorithms onto class-specific arrays

The method described in Chapters 4 and 5 can also be used to map algorithms onto class-specific arrays. Several of these arrays have been built or published [Anna87, Groo87, Niel88, Drak87, Syma88] and in [More89a] we propose one that is well suited for the use of our method.

Since the number of processing elements in a class-specific array is fixed, and such a structure is used in a relatively general purpose mode, the size of the array does not correspond to the sizes of matrices; consequently, mapping requires to use the partitioning capabilities of our method.

The regularization stage of the method remains unchanged; that is, for a particular algorithm the same MMG is obtained as for the application-specific case. The only additional consideration is that primitive nodes in the graph may correspond to operations more complex than the capabilities of functional units in the array, as long as these operations are executed in one cell and fulfill the requirement of at most three inputs and three outputs. This issue is discussed in detail later.

On the other hand, the second stage of the method is influenced by the restrictions imposed by the architecture of the target array. Since these restrictions may have a large number of alternatives, the method has to be adapted to each particular case. We cannot cover all possible situations here; instead, we discuss the issues to consider in this adaptation.

The aspects involved are illustrated using the LU-decomposition algorithm, which is mapped onto a particular class-specific linear array.

6.1 The regularization stage

In the design of an algorithm-specific array one uses a description of the algorithm at the finest level of granularity desirable for an implementation, because such an information is used to determine the characteristics of cells that com-

pose the array. For example, to realize an algorithm which includes division with cells not capable of executing that operation, one may choose to describe division in terms of basic operations (i.e., a sequence of multiplies); those multiplies are realized in the array through the application of the method.

In contrast, granularity in mapping onto a class-specific array is related to the execution of operations in a cell. That is, the algorithm only needs to be described in terms of operations that are indivisible for a cell, even when the cell has to execute such operations as sequences of basic operations. For example, mapping triangularization by Givens' rotations onto an array whose cells are only capable of performing multiplication and addition requires determining how rotation angles and rotations are computed. However, each rotation angle or rotation is computed entirely in one cell (there is no advantage in spreading such a computation throughout cells), so the mapping process only needs to consider a description in terms of the complex operations that are allocated to each cell.

Dividing complex operations across cells might seem a suitable approach to achieve load balancing. That is, complex operations that are composed of several primitive operations (and therefore take several time-steps to complete) could be distributed across different cells that share such a load. However, load balancing can also be achieved by allocating a varying number of complex operations per cell, taking into account the computation time of each operation. As will be discussed in this chapter, this latter approach is simpler and effective in achieving load balancing.

The regularization stage requires determining the multi-mesh dependency graph (MMG). This graph is derived as discussed in Sections 5.5 and 5.7, and was illustrated in Section 5.11 for the LU-decomposition algorithm. The algorithm and resulting MMG are repeated in Figure 6.1 for later use in this chapter.

6.2 The mapping stage and the specific target architecture

The second stage of the method when partitioning an algorithm for a class-specific array consists of:

- Collapsing the MMG onto a mesh graph. This step involves the selection of a collapsing direction and of a prism size. Moreover, it implies determining a schedule for the operations included in a prism.

```

For k = 1 to n
  ukk = 1/akk
  For j = (k + 1) to n
    ukj = akj
  For i = (k + 1) to n
    lik = aik * ukk
  For i = (k + 1) to n
    For j = (k + 1) to n
      aij = aij - lik * ukj

```

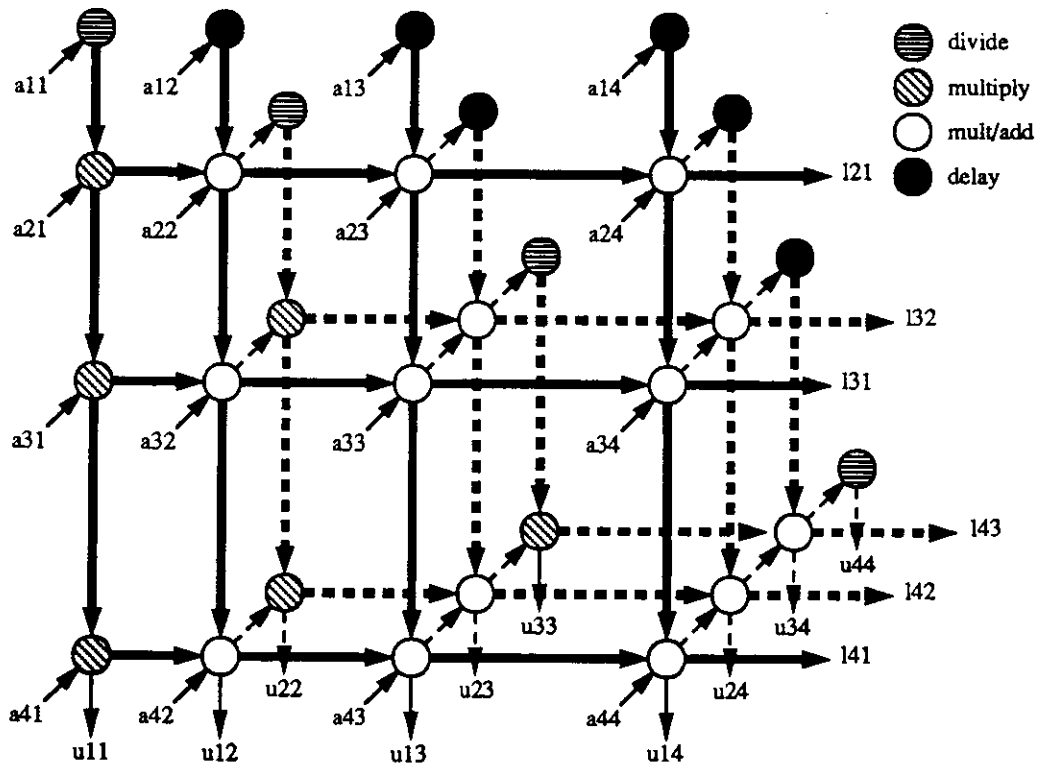


Figure 6.1: The LU-decomposition algorithm and its MMG

- Partitioning the mesh graph so that it can be executed in the array. The strategies discussed in Section 2.6 can be used. This partitioning determines the schedule of nodes and the I/O.

We now discuss how the basic architectural characteristics influence each of these steps. Note that several characteristics might influence the same aspect of the process, many times in conflicting ways; in such cases, different alternatives should be analyzed to select the most appropriate one. This selection is aided by the representation of the algorithm in the regularized form (the MMG).

Type of cell

As discussed in Section 2.2, there are three types of cells: systolic, pseudo-systolic and local-access. The type of cell determines the size of prisms in the grouping process. For systolic cells the size of the prism's base is 1 by 1, for pseudo-systolic cells is p by q , while for local-access cells it depends on the size of the array (i.e., (n/K) by n in a linear array with K cells, (n/K) by (n/K) in a two-dimensional array also with K cells). Moreover, the values of p and q for a pseudo-systolic cell are determined by the size of the cell's internal storage, the cell bandwidth, and the number of stages in the pipeline, as discussed in the previous chapters.

Functional units

The type and number of functional units per cell, and their organization, determines the schedule of operations within a prism. To obtain maximal utilization of arithmetic resources, it is necessary to schedule in every time-step as many operations as functional units exist in the array. For example, maximal utilization in cells composed of a multiplier and an ALU requires scheduling one multiply and one ALU operation in each time-step. However, dependencies in an algorithm might not allow doing so.

Datapaths

The datapaths to access operands from memory, from neighboring cells, and from the cell's internal storage determine the allocation of data and the schedule of operations.

To schedule several operations per time-step in a cell (as may be required by the number of functional units), it is necessary that many operands be available and several results be stored in every time-step. Since the number of memory accesses that can be initiated per time-step is usually limited, the maximal utilization is achieved only while executing operations that access several sources/destinations other than memory. For example, if cells are composed of a multiplier and an ALU that are accessed independently and memory allows initiating only two accesses per time-step, maximal utilization requires that at least four sources/destinations of data are other than memory.

Cells may share storage for operands and results. Consequently, scheduling has to avoid conflicts in using that shared resource(s), limiting the flexibility of such a scheduling.

I/O paths

I/O paths to internal cells influences the direction of collapsing. For example, external I/O is available only at the ends of a linear array. Consequently, transferring input data to inner cells requires to use outer cells. However, cells may not be capable of performing useful computations and data transfers simultaneously. Using an architecture with such characteristics requires to devise, if possible, mappings that allow allocating operations with input data on cells at the ends of the array, while inner cells operate on intermediate results generated by neighbor cells.

6.3 Mapping onto local-access arrays

Mapping onto class-specific systolic or pseudo-systolic arrays follows the computational model and method described in Chapters 4 and 5, as outlined in the previous section. In contrast, mapping onto local-access arrays exhibits several important differences as a consequence of having large memory per cell, as discussed below.

6.3.1 The impact of large storage per cell

1. Local-access arrays are suitable to perform partitioning using *coalescing*, because large memories allow the allocation of a portion of the problem to each cell. Moreover, cell bandwidth is low because many operations are performed

with data local to each cell, thus reducing the need to transfer data between cells.

2. Since communication between cells is low, as described in (1), cells may operate in a *loosely-coupled* mode instead of the tightly-coupled that characterizes systolic and pseudo-systolic arrays. Consequently, the computational model suitable for local-access cells is *asynchronous*. That is, cells operate independently on data that is stored locally, transfer results to neighbor cells as determined by the mapping process, and synchronize through queues or flags attached to ports.

In addition to the differences listed above, an important similarity between local-access and pseudo-systolic/systolic cells arises from the existence (or lack) of some small storage close to the functional units. A local-access cell may or may not have a *register file* which may be used to perform a function similar to the buffers in pseudo-systolic cells. Such a register file impacts the mapping process, as discussed later. In the remaining of this chapter, we assume that local-access cells have a register file.

6.3.2 Coalescing the MMG

There are some properties of an MMG such as the one in Figure 6.1 that are necessary to consider when *coalescing* an algorithm onto a local-access array:

1. The MMG may have uneven distribution of primitive nodes throughout meshes. For example, Figure 6.1 shows that there are fewer nodes at the left of the graph than at its right. The same is true from top to bottom. Such a distribution of nodes varies monotonically across the graph.
2. The MMG may have uneven distribution of computational requirements throughout nodes. For example, Figure 6.1 shows that many nodes compute multiply/adds (i.e., update the matrix) which are “light-weight” nodes in the sense that they require a basic operation, while there are fewer “heavy-weight” nodes which require division. This difference in computational requirements of nodes is even larger in algorithms such as triangularization by Givens’ rotations, where heavy-weight nodes correspond to computation of rotation angles (including divides and square-root) while light-weight nodes are rotations (multiplies and adds only).

Partitioning by coalescing doesn't explicitly derive a mesh-graph, as done for systolic and pseudo-systolic arrays. Although one could derive a mesh-graph and map that graph onto a local-access array, coalescing allows going directly from the MMG to the array without the need to derive the mesh-graph. As it will be apparent from this chapter, the mesh-graph is implicit in the mapping process.

Coalescing requires dividing the MMG into subgraphs that are allocated to each cell. Consequently, there must be as many partitions as cells in the array. A suitable way to obtain these partitions consists of dividing the MMG into sections by performing cuts along one of the axes in the three-dimensional space. There are three options for this task:

Uniform partitions, that is, divide the MMG into *partitions of the same size*, and allocate each partition to a cell.

This approach has the advantage of being simple, but it might lead to unbalanced distribution of load in the array and consequently bad utilization of cells, as a result of the uneven distribution of nodes throughout the MMG and the different computation time of nodes.

Interleaved uniform partitions, that is, divide the MMG also into partitions of the same size but such *partitions are composed of nodes from non-neighbor meshes in the MMG*. For a linear array, for example, traversing meshes along an axis of the three-dimensional space leads to allocating mesh i to cell $(i \bmod K)$, where K is the number of cells in the array.

Since the distribution of nodes throughout the MMG is uneven and the size of meshes varies monotonically along dimensions of the graph, this approach attempts to compensate such unevenness by spreading meshes of similar size throughout cells. For example, the K smallest (or largest) meshes in the MMG are first distributed across the K cells in a linear array, then the next K meshes, and so on. Since meshes of similar size are distributed to cells in each case, good load balancing is expected. This technique requires bidirectional communication between cells and a more complex scheduling policy than uniform partitioning, as will be described later.

Non-uniform partitions, that is, divide the MMG into partitions of variable size in such a way that *the total computational load assigned to each partition is roughly the same*, and allocate each of such partitions to a cell.

This approach requires additional work to determine an adequate size for each partition, but potentially leads to better utilization of cells than uniform

```

Compute  $n_i =$  load in mesh  $i$  in MMG, for  $i = 1$  to  $n$ 
Compute  $N_t =$  total load in execution of algorithm
Compute  $N_{avg} = \frac{N_t}{\#cells} =$  average load per cell
 $i = 0$ 
For each cell do
     $i = i + 1$ 
    allocate mesh  $i$  of the MMG to  $PE_k$  until load in
     $PE_k$  is nearest possible to average load per cell

```

Figure 6.2: A heuristic approach to perform non-uniform coalescing

partitions because it provides better load balancing. In the next section, we propose a simple heuristic procedure that allows determining the size of partitions.

Later in this chapter, we will illustrate these different partitioning strategies using LU-decomposition and a linear array.

6.3.3 A heuristic approach to non-uniform coalescing

To perform non-uniform coalescing, we need a technique that allows selecting the size of each partition. We propose a heuristic approach, described by the algorithm in Figure 6.2 for a linear array. As indicated in the figure, this technique allocates meshes from the MMG to a cell until the load accumulated on such a cell is close to an ideal average per cell.

To choose the break point in allocating meshes to cells, the heuristic technique uses a greedy strategy. That is, before allocating a mesh to a cell we compute the total load placed on such a cell and compare it with the ideal average. If the load is higher than the average, the difference from that average is computed assuming that the mesh is both allocated and not allocated to the cell. The final selection is determined by the smallest difference among the two.

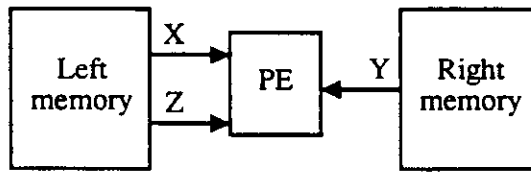


Figure 6.3: The allocation of data to memory modules

6.3.4 The allocation of data and the schedule of primitive operations

Mapping onto local-access arrays requires determining where data is stored, how data flows through the array, and how operations are scheduled for execution. We can assume that *memory attached to a cell is divided (physically or logically) into two modules* and that two memory accesses (one to each module) may be initiated every time-step; we refer to those modules as *left-memory* and *right-memory*. (If memory allows initiating only one access per cycle, then memory bandwidth is lower; in such a case, allocation and flow of data are determined in the same manner as the case of insufficient memory bandwidth described later.) Under these conditions, and assuming that transmittent data flows along the X -axis in the MMG, we propose the following allocation of data to memory modules as shown in Figure 6.3:

- Data flowing along the X -axis in the multi-mesh graph is allocated to the left memory module of a cell. That is, a value that is transferred along the X -axis of the MMG is accessed by a cell from the left memory module.
- Data that flows along the Y -axis in the MMG is allocated to the right memory module.
- Data that flows along the Z -axis in the MMG is also allocated to the left memory module.

With the allocation of data to memory modules given above, transmittent data may be read from memory into the register file, and operations may be initiated by reading one element from each memory module without conflicts. This allocation works well when memory reads and memory writes may be interleaved without conflicts. For example, an operation that takes more than one time-step allows reading data from memory in one cycle and storing results in the following cycle(s). That is the case of Givens' rotations for example, where data elements

$$\begin{aligned}
a_{jk}^* &= a_{jk} - a_{ji}a_{ik} \\
a_{ik}^* &= ca_{ik} + sa_{jk} \\
a_{jk} &= a_{jk}^* \\
a_{ik} &= a_{ik}^*
\end{aligned}$$

Figure 6.4: Rotation in the square-root free algorithm

for one rotation are used to perform three multiplies (or four, depending on the algorithm) and two adds before sending results to memory, as shown in Figure 6.4. Consequently, while one rotation is being computed results from the previous one may be sent to memory without conflicts.

In contrast, if every read operation implies writing a result in memory, then there is not enough memory bandwidth to support the data allocation described above. In such a case, one must rely on a more complex scheduling of operations that uses the register file to save intermediate results and re-uses data from there. For example, in LU-decomposition every multiply/add operation reads two operands (in addition to a common value for several operations), and produces one result. If a multiply/add is initiated every time-step, two memory reads and one memory write are initiated in every cycle and there is not enough memory bandwidth to support such a scheme.

This problem of memory bandwidth and cell computation rate is equivalent to that found in the design of algorithm-specific arrays described earlier in this dissertation. Consequently, a similar solution to the one devised there may also be applied in this case. Such a solution is based on the concept of prisms in the MMG. That is, *data allocation and scheduling primitive operations is done by prisms, and the prisms base size is determined by the size of the register file within a cell.*

To illustrate this issue further, let's look into scheduling primitive operations for LU-decomposition in more detail. In this algorithm, it is convenient to optimize the computation of multiply/adds because these are the most frequent operations. Ignoring for the moment restrictions arising from conflicts in accessing memory, a cell could obtain from memory the value corresponding to an element l_{ik} , save it in the register file, perform all operations associated to that element while reading from memory the corresponding a_{ij} and u_{kj} pairs, and then repeat the process by obtaining a new l_{ik} . This schedule allows using a cell pipeline for long intervals and should be the driving force in a mapping procedure. Consequently, *primitive*

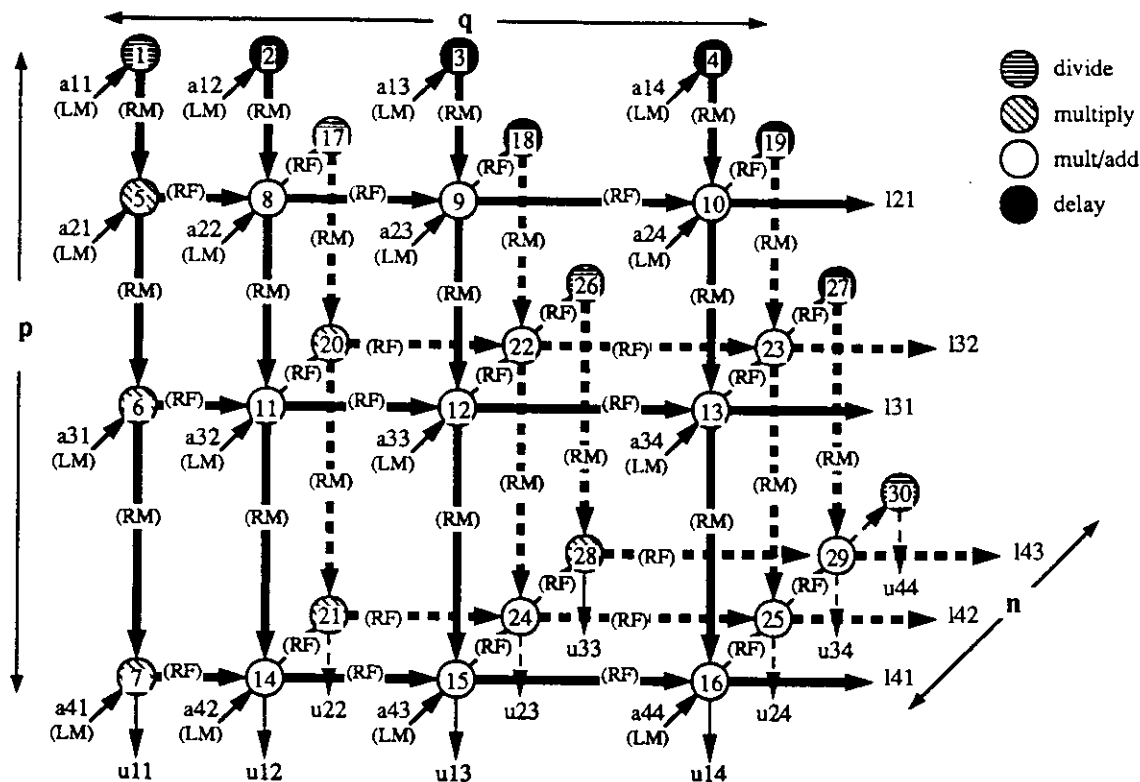


Figure 6.5: Schedule of primitive operations in the LU-decomposition

operations are scheduled following the flow of transmittent data in the MMG (i.e., following the flow of l_{ik}). Moreover, paths of transmittent data allocated to a cell are long, so that the pipeline may be kept full for long intervals.

However, the scheme above is not feasible because, for every pair (a_{ij}, u_{kj}) read from memory, an updated value of a_{ij} must also be stored in memory. Instead, scheduling multiply/adds may be done by prisms as depicted in Figure 6.5 for a linear array. In such a figure, each edge of the MMG has been tagged with the name of the associated storage module: RF for register file, LM and RM for left and right memory modules respectively. This prism correspond to executing the first operations in the leftmost cell of the array. Scheduling inner prisms and in inner cells follows the same pattern described here. The schedule is as follows:

- a_{11} is read from LM, u_{11} is computed and stored in RM. This process is repeated with a_{12}, a_{13} and so on until a_{1q} (q is determined by the size of the register file, as discussed later).

- a_{21} is read from LM, u_{11} is read from RM, l_{21} is computed and stored in RF. This process is repeated with a_{31}, a_{41} and so on until a_{p1} (p is also determined by the size of the register file). Elements l_{i1} are also transferred to the next cell to the right.

These operations are pipelined without lost cycles, as long as q is larger than the number of stages in the pipeline. At the end of these two steps, all values of l_{i1}, u_{1j} contained in the prism are computed and stored in RF and RM respectively. Now, multiply/add operations are scheduled as follows:

- a_{22} is read from LM, the corresponding u_{12} is read from RM, l_{21} is read from RF, a multiply/add operation is executed and the result is stored in RF. This process is repeated with a_{23}, a_{24} and so on until a_{2q} .
- The step above is repeated for other horizontal paths of the MMG until reaching path p .

At the end of the steps above, the portion of the matrix that has been updated is stored in the register file, values l_{i1} are also stored in RF, while values u_{1j} are stored in RM. The entire schedule is repeated, but this time reading elements a_{ij} from RF as follows:

- a_{22} is read from RF, u_{22} is computed and stored in RM. This process is repeated with a_{23}, a_{24} and so on until a_{2q}
- a_{32} is read from RF, u_{22} is read from RM, l_{32} is computed, stored in RF and transferred to the next cell. This process is repeated with a_{42}, a_{52} and so on until a_{p2} .
- a_{33} is read from RF, the corresponding u_{23} is read from RM, l_{32} is read from RF, a multiply/add operation is executed and the result is stored in RF. This process is repeated with a_{34}, a_{35} and so on until a_{3q} .
- The last step above is repeated for other horizontal paths of the MMG until reaching path p .

With the schedule above, there are no cycles lost as long as paths (vertical and horizontal) inside the prism are longer than the number of stages in the pipeline. Moreover, the schedule above determines the relationship between the size of the

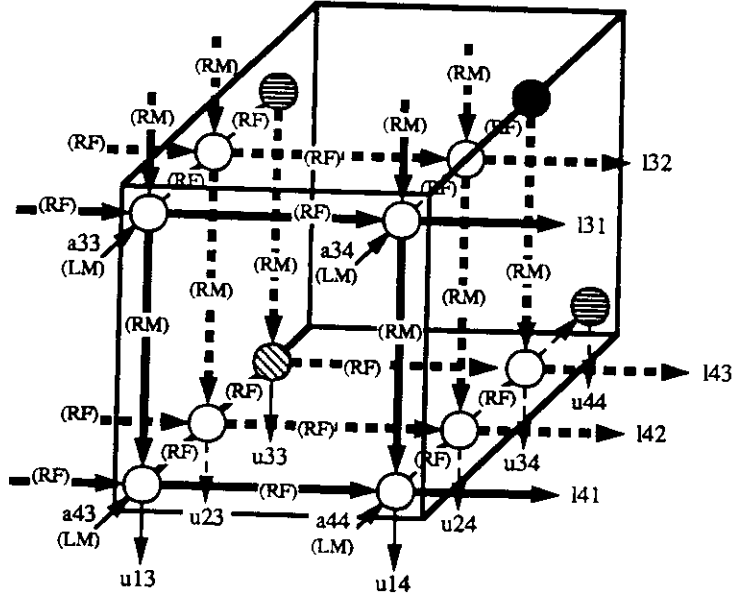


Figure 6.6: Scheduling primitive operations in an inner prism of the MMG

register file and prism size, which is identical to the one found for storage per cell in Chapter 5. Representing the size of the register file as RF_w , this relation is

$$RF_w = p + pq = p(q + 1)$$

In the remaining of this chapter, we assume that RF_w is large enough so that time-steps lost due to pipeline latency while executing inner meshes of an incomplete prism are negligible.

Scheduling prisms as discussed above is characterized by using the arithmetic pipeline to compute every variable used. Instead, a prism that appears later in the MMG doesn't need to compute values l_{ik} and u_{kj} because those are transmittent data, as depicted in Figure 6.6. However, the schedule expects to compute and store l_{ik} in the register file. Consequently, inner cells require to transfer those values into RF without performing computations, introducing some overhead due to data transfers.

Notice that the overhead mentioned above exists only for the leftmost prism allocated to a cell. By scheduling prisms in horizontal order, other prisms in the same cell may use values l_{ik} already in RF so that no further overhead exists.

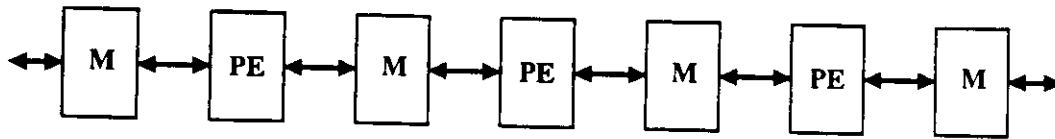


Figure 6.7: An hypothetical memory-linked array

6.4 Example: mapping onto a memory-linked array

We now illustrate mapping for a particular target array, using LU-decomposition as application example.

6.4.1 The target architecture

The architecture used in this example consists of a linear array of processing cells linked by large dual-port memory modules, as shown in Figure 6.7. Consequently, each cell has access to large storage which is physically divided into two modules, one attached to each side of a cell. Communications between cells occurs through the memories, and memory modules at the ends of the array communicate with a host for I/O. This target architecture has been previously proposed (i.e., MWAP [Kung87b] (pp.31-32)), and a commercial memory-linked array is QUEN [Niel88].

We assume that cells in the target array contain a 4-stage pipelined arithmetic unit consisting of a multiplier and an ALU connected in cascade. In addition, each cell contains a 3-port (2-out, 1-in) register file, is capable of parallel access to the two memory modules to which it is attached to, has independent addressing logic for each memory module, and has internal program memory. Such a cell is depicted in Figure 6.8. Data memory is accessed as a 2-stage pipeline. Consequently, an operation that accesses data from/to memory sees the cell as an 8-stage pipeline: 2-stage memory (to read operands), 4-stage arithmetic pipeline, 2-stage memory (to write results).

Each cell may initiate one read or write one word per time-step from/to each memory module. In addition to reading/writing operands from/to memory, cells use the register file as source and destination of operands, as also depicted in Figure 6.8.

The arithmetic pipeline allows initiating one multiply and one ALU opera-

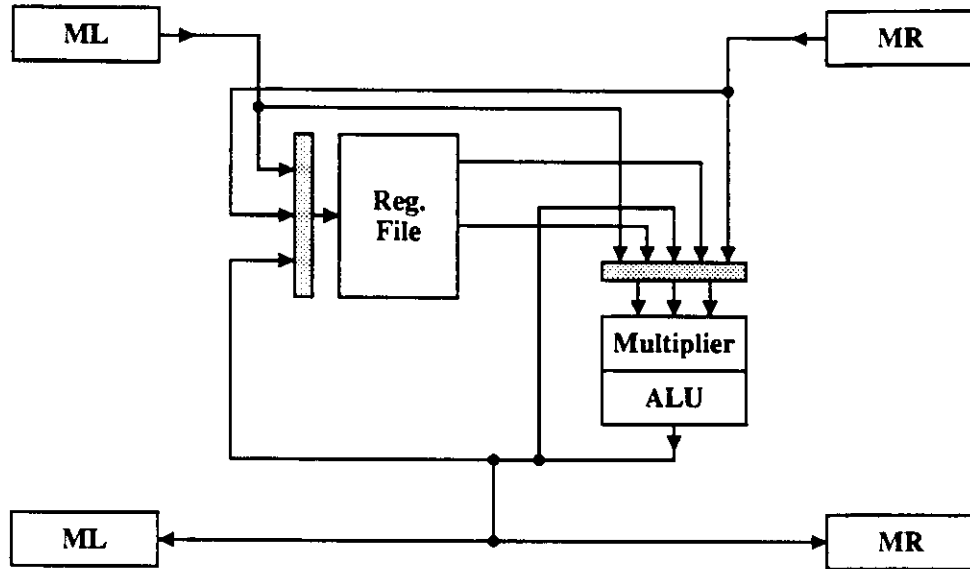


Figure 6.8: A cell in the target class-specific array

tion per time-step. Complex operations (such as divide, square-root, rotation) are implemented as sequences of basic operations and iterative procedures. For example, we assume that a divide is implemented in the equivalent of eight multiply operations. Consequently, primitive operations in an algorithm have different computation time in the target array. All operations are performed on single-precision floating-point data, and all data transfers are done on floating-point data as well.

We assume that synchronization of cells is achieved using flags in the memory modules (i.e., as MWAP does [Kung87b]).

6.4.2 The performance evaluation measures

The main performance measures used to evaluate different mappings of an algorithm onto a class-specific array are computation time (t) and utilization of processing elements (U). Such measures are computed as follows:

$$t = t_b$$

$$U = \frac{N_f}{KFt_b}$$

where

- t_b is the computation time of the busiest cell.

- N_f is the total number of [flops] required by the algorithm.
- K is the number of cells.
- F is the cell computation rate (i.e., number of [flops] per cell per time-step).

The total number of operations in the LU-decomposition algorithm is inferred from the MMG shown in Figure 6.1 as:

$$\begin{aligned}
 N_{\text{div}} &= n \\
 N_{\text{mult}} &= \sum_{i=1}^n (i-1) = n(n-1)/2 \\
 N_{\text{mult/add}} &= \sum_{i=1}^n (i-1)^2 = n(n-1)(2n-1)/6
 \end{aligned}$$

These expressions do not include delay nodes in the MMG. Such nodes are data transfers in the algorithm (i.e., $u_{kj} = a_{kj}$), so that they do not contribute computing load. Since we assume that a divide requires 8 [flops] in the target cells, and since a multiply/add needs 2 [flops], the total number of [flops] N_f required by LU-decomposition is

$$\begin{aligned}
 N_f &= 8n + \frac{n(n-1)}{2} + \frac{2n(n-1)(2n-1)}{6} \\
 &= \frac{1}{6}[4n^3 - 3n^2 + 47n] \text{ [flops]}
 \end{aligned}$$

For a matrix of size $n = 200$, the expression above leads to

$$N_f(200) = 53149200 \text{ [flops]}$$

In the remaining of this chapter, we use this value of N_f for evaluations of estimated performance.

Computing LU-decomposition in a single cell

Before mapping the LU-decomposition algorithm onto the target array, let's look into the performance achievable by computing the algorithm in a single cell. This result gives a measure of the cells suitability for the type of algorithm at hand. The schedule of primitive operations is as described in Section 6.3.4. We assume that the register file is large enough so that computing cycles lost to pipeline latency are negligible, and that a new operation may be initiated every cycle without

conflicts. Moreover, since overhead in the execution of inner prisms (according to the chosen schedule) arise from transferring values of l_{ik} into RF, we schedule prisms in horizontal order so that each value l_{ik} is computed, stored in RF, and used for every operation that requires it. Consequently, no overhead in transferring data exists.

From the expressions given earlier for number of operations in the algorithm, and since a multiply/add is initiated in a single cycle, the total number of time-steps required to execute LU-decomposition in the target array is

$$t_1(n) = 8n + \frac{n(n-1)}{2} + \frac{n(n-1)(2n-1)}{6} = \frac{n}{3}(n^2 + 23)$$

For a matrix of size $n = 200$, this becomes

$$t_1(200) = 2668200 \text{ [time steps]}$$

Consequently, utilization achievable in an implementation with a single cell is

$$U_1(200) = \frac{N_f}{K * F * t_1} = \frac{5314900}{1 * 2 * 2668200} = 0.996$$

That is, executing the LU-decomposition algorithm in one cell of the target array allows obtaining almost unitary utilization. The only losses of utilization arise when computing simple multiply operations and divides, because no other operation is executed in the ALU at the same time. However, as determined from the MMG, such operations are few compared with the number of multiply/adds that may be pipelined in the single cell.

6.4.3 The mapping process

We discuss coalescing LU-decomposition onto the memory-linked array using the different techniques to determine the partitions discussed earlier.

6.4.4 Uniform coalescing

Let's consider first performing uniform coalescing and the performance that is achievable with such a strategy. As indicated before, we consider three possible

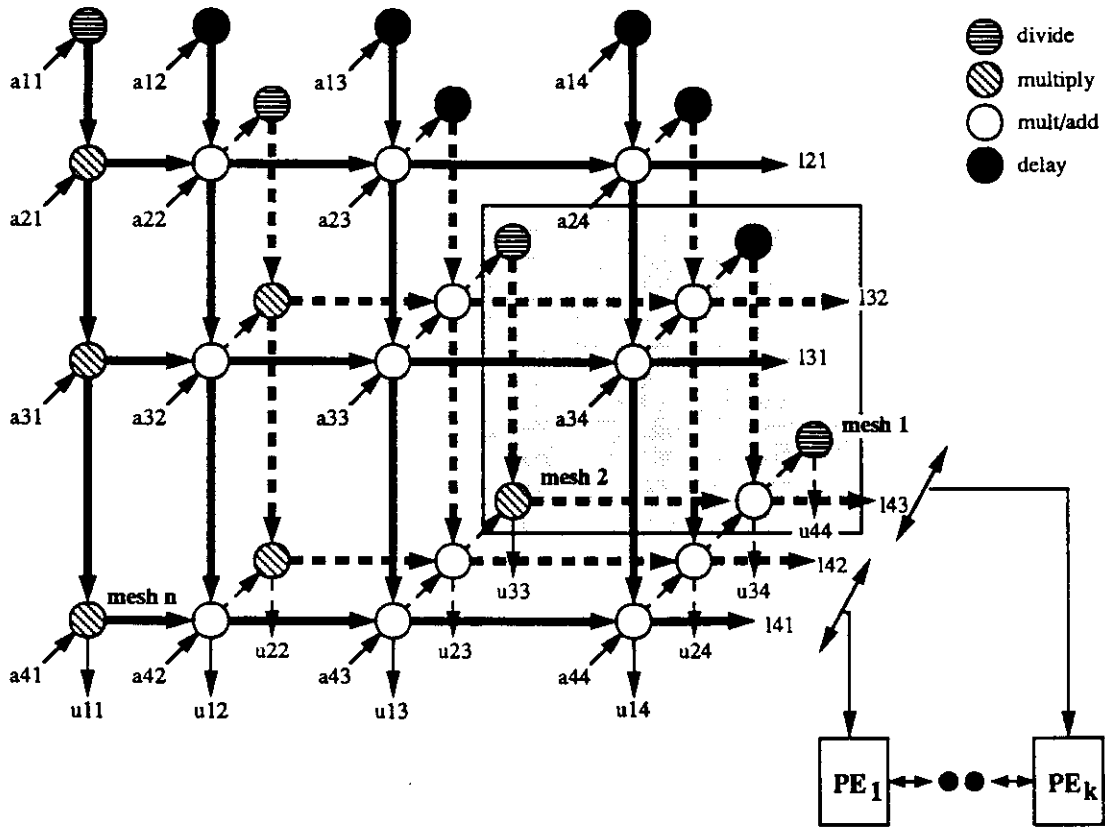


Figure 6.9: Coalescing the LU-decomposition along the Z -axis

ways to coalesce the graph, namely by cutting the MMG along each axis in the three-dimensional space. These alternatives are reviewed now.

6.4.4.1 Along the Z -axis

Coalescing the MMG in Figure 6.1 by partitioning the graph along the Z -axis and assigning each partition to one cell is depicted in Figure 6.9. This technique corresponds to allocating to each cell a number of iterations of the outermost loop of the algorithm in Figure 6.1. Data is input only to the leftmost cell, so that *no pre-loading of data to inner cells is required*. Moreover, data transfers to cells at the right occur as results of computations, so that *no time-steps are required to perform data transfers*.

Cells to the right in the array receive as input a smaller matrix than the one processed by a cell to the left; such smaller matrix is generated (i.e., updated) in

the left cell. Moreover, each cell produces part of the resulting matrix, in row-wise manner; since the output is a triangular matrix, more results are produced in leftmost cells than in rightmost ones.

Let's look into the performance of this mapping. Traversing meshes along the Z -axis as shown in Figure 6.9, the number of operations in mesh i and the total number of time-steps N_Z^i required to execute such a mesh are

$$\begin{aligned} N_{div}^i &= 1 \\ N_{mult}^i &= (i - 1) \\ N_{mult/add}^i &= (i - 1)^2 \\ N_Z^i &= 8 + (i - 1) + (i - 1)^2 \end{aligned}$$

Since we consider that the register file is large enough, these expressions do not include cycles lost due to dependencies and pipeline latency. Moreover, since each element l_{ik} is computed and used only in once cell, no overhead due to data transfers exist.

With uniform coalescing, mapping by cutting along the Z -axis requires that each cell computes n/K meshes of the graph. If n/K is not an integer value, and since meshes of the MMG along the Z -axis have a decreasing number of nodes, we allocate $\lfloor n/K \rfloor$ meshes of the graph to half of the PEs (the leftmost ones), and $\lceil n/K \rceil$ meshes to the remaining ones.

For $K = 10$, the number of time-steps required by each cell when solving a problem of size $n = 200$ is given in Table 6.1, which shows that uniform coalescing along the Z -axis leads to bad load balancing as a consequence of the uneven distribution of nodes and nodes' computation time throughout the MMG. The total time required by the target array to compute the LU-decomposition is given by the busiest cell, namely PE_1 . Considering the number of [flops] in the algorithm for a problem of size $n = 200$, the performance of the array with this mapping is

$$\begin{aligned} t_Z &= t_{PE_1} = 722820 \text{ [time steps]} \\ U_Z(200) &= \frac{N_f}{K * F * t_{PE_1}} = \frac{5314900}{10 * 2 * 722820} = 0.37 \end{aligned}$$

In addition to low utilization, an (apparent) drawback of the approach described above is the latency introduced between cells. That is, cells to the right start computing the algorithm delayed with respect to cells at the left. However, due to the load unbalance, such a latency doesn't affect performance of the array (i.e., rightmost cells have fewer operations to perform).

Table 6.1: Load per cell in uniform coalescing along Z -axis

Cell	Cycles	Cell	Cycles
1	722820	6	162820
2	578820	7	98820
3	450820	8	50820
4	378820	9	18820
5	242820	10	2820

The expressions given above are a simplified view of the results obtained with the mapping performed here. In particular, such expressions assume that data is always available, and there are no time-steps lost to dependent operations and latency in the arithmetic pipeline. As we discussed before, some time-steps are lost when the size of meshes within a prism is smaller than the number of stages in the pipeline, as it is the case when executing innermost meshes along the Z -axis (i.e., in the last cell). However, since the busiest cell is PE_1 , this issue doesn't impact the performance of the implementation.

6.4.4.2 Along the X -axis

Let's consider now the case of coalescing the MMG in Figure 6.1 along the X -axis and assigning each partition to one cell, as shown in Figure 6.10. This approach corresponds to allocating to each cell a number of columns of the input matrix, so that data must be pre-loaded in cells before starting execution of the algorithm. Moreover, each cell produces part of the resulting matrix, in column-wise manner, so that cells to the right generate more results than cells to the left. (In case n/K is not an integer value, the allocation of meshes to cells is done as described for mapping along the Z -axis.)

We discuss now the performance achievable with this uniform coalescing. Numbering meshes from right to left in Figure 6.10, the number of operations in mesh i along the X -axis is

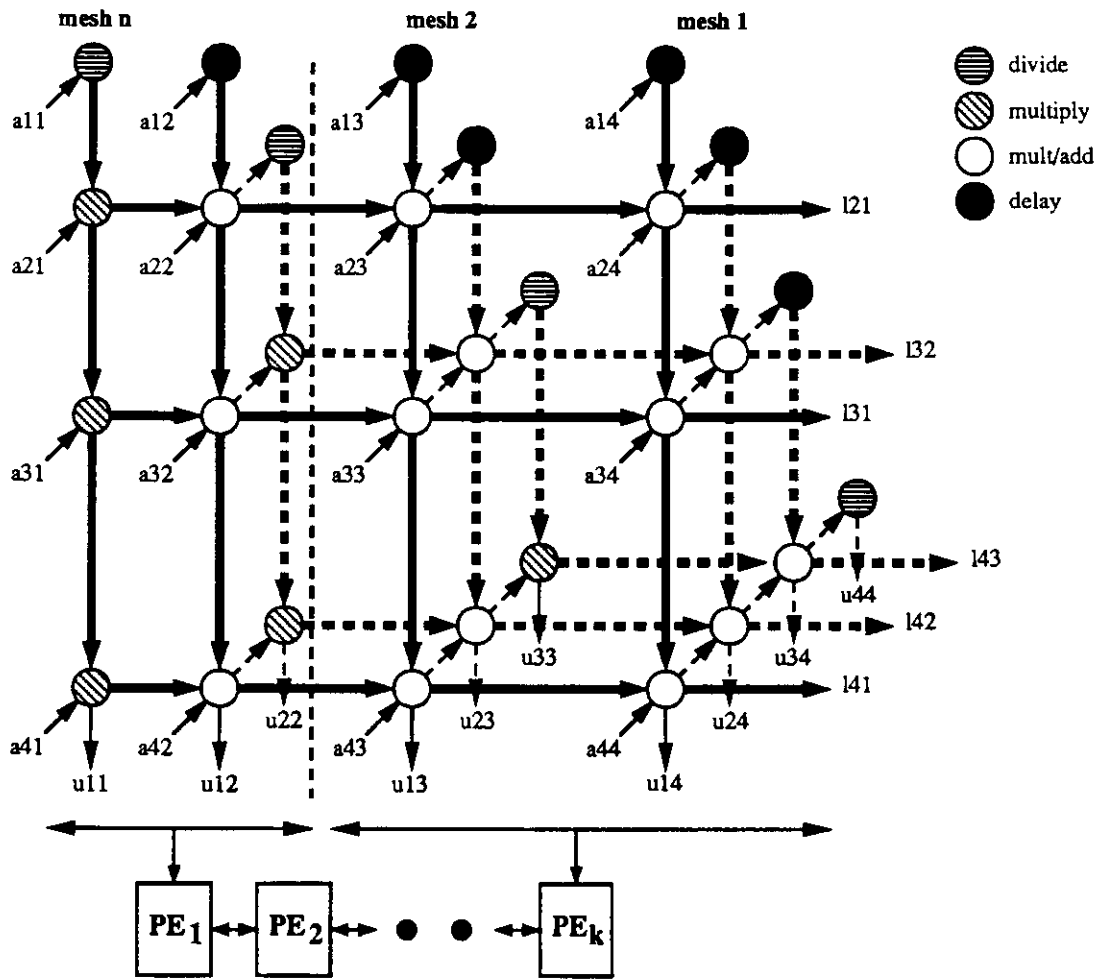


Figure 6.10: Coalescing the LU-decomposition along the X-axis

	Number of ops.
divides	1
multiplies	$(i - 1)$
multiply/adds	$\sum_{j=i}^{n-1} j = [n(n - 1) - i(i - 1)]/2$

Consequently, the total number of cycles N_X^i required to execute mesh i is

$$\begin{aligned} N_X^i &= 8 + (i - 1) + \frac{1}{2}[n(n - 1) - i(i - 1)] \\ &= 8 + \frac{1}{2}n(n - 1) - \frac{1}{2}(i - 1)(i - 2) \end{aligned}$$

This mapping requires the transfer of values l_{ik} through cells. As stated earlier, these data transfers are a source of overhead in the algorithm. This overhead consists of values l_{ik} that a cell receives from the left, which corresponds to the number of multiply/add operations in the leftmost mesh allocated to a cell. That is,

$$\begin{aligned} L_k &= (n/K)(K - k - 1) && \text{leftmost mesh allocated to PE}_k \\ O_k &= \sum_{j=L_k}^{n-1} j && \text{overhead in PE}_k \\ &= n(n - 1)/2 - L_k(L_k - 1)/2 \end{aligned}$$

For $K = 10$, the number of cycles required by each cell when operating on a matrix of size $n = 200$ is given in Table 6.2, which indicates that this approach exhibits better load balancing than mapping along the Z -axis. This is a consequence of distributing triangular meshes across cells rather than square meshes as it was the case before. The total time required to compute the algorithm is given by the busiest cell, namely PE_{10} , which is also the cell with the largest overhead due to data transfers. A similar analysis to mapping along the Z -axis leads to the following performance estimates:

$$\begin{aligned} t_X &= 416730[\text{time steps}] \\ U_X(200) &= \frac{N}{K * F * t_{\text{PE}_{10}}} = \frac{5314900}{10 * 2 * 416730} = 0.64 \end{aligned}$$

Same as in mapping along the Z -axis, the expressions given above are a simplified view of the results obtained with this scheme. In particular, the expressions above are valid only for the outermost meshes. Upon traversing a few meshes along the Z -axis, the size of meshes allocated to leftmost PEs becomes smaller than the pipeline latency, leading to the introduction of idle cycles. However, the performance of the array is determined by PE_{10} , where this overhead is small compared to the number of operations allocated to this cell.

Table 6.2: Load per cell in uniform coalescing along X -axis

PE	Cycles	PE	Cycles
1	40620	6	333570
2	116010	7	366960
3	183000	8	391950
4	241590	9	408540
5	291780	10	416730

6.4.4.3 Along the Y -axis

The third uniform coalescing alternative consists of grouping the MMG in Figure 6.1 along the Y -axis, as depicted in Figure 6.11. This technique corresponds to allocating data to each cell in row-wise manner, that is, each PE has access to several rows of the input matrix. Consequently, this partitioning strategy requires to preload data in cells. However, all results are obtained in the rightmost cell.

We discuss now the performance achievable with uniform coalescing in this case. Numbering meshes from bottom to top in Figure 6.11, the operations in mesh i along the Y -axis are

	Number of ops.
divides	1
multiplies	$(n - i)$
multiply/adds	$\sum_{j=i}^{n-1} j = [n(n - 1) - i(i - 1)]/2$

Consequently, the total number of cycles N_Y^i required to execute mesh i is

$$N_Y^i = 8 + (n - i) + \frac{1}{2}n(n - 1) - \frac{1}{2}i(i - 1)$$

This mapping computes and uses each l_{ik} value within the same cell, so that no overhead due to data transfers arises from this source. However, values u_{kj} are transferred across cells. Since the schedule in prisms expects to find values u_{kj} in RM, such values have to be passed from LM to RM, introducing overhead that corresponds to the number of u_{kj} values that arrive to a cell, or equivalently, that corresponds to the number of multiply/add operations in the topmost mesh allocated to a cell. That is,

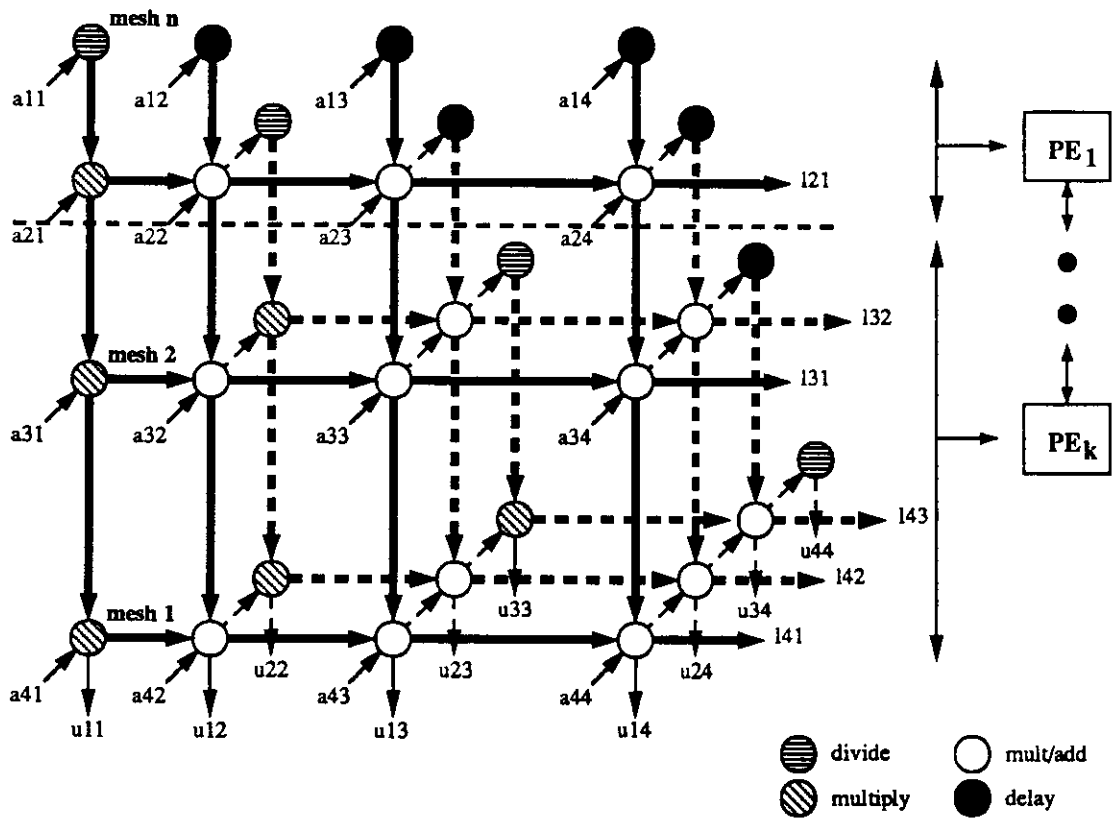


Figure 6.11: Coalescing the LU-decomposition along the Y-axis

Table 6.3: Load per cell in uniform coalescing along Y -axis

PE	Cycles	PE	Cycles
1	37020	6	333970
2	113210	7	368160
3	181000	8	393950
4	240390	9	411340
5	291380	10	420330

$$\begin{aligned}
 T_k &= (n/K)(K - k + 1) && \text{topmost mesh allocated to PE}_k \\
 O_k &= \sum_{j=T_k}^{n-1} j && \text{overhead in PE}_k \\
 &= n(n-1)/2 - T_k(T_k - 1)/2
 \end{aligned}$$

For $K = 10$, the number of time-steps required by each cell with a 200 by 200 matrix is given in Table 6.3. The total time required to compute the algorithm is given by the busiest cell, namely PE_{10} , which is also the cell with largest overhead due to data transfers. Similarly to the previous cases, performance is computed as

$$\begin{aligned}
 t_Y &= 420330[\text{time steps}] \\
 U_Y(200) &= \frac{N}{K * F * t_{\text{PE}_{10}}} = \frac{5314900}{10 * 2 * 420330} = 0.63
 \end{aligned}$$

This scheme has a performance almost identical to that derived by mapping along the X -axis, due to the symmetry of the MMG along axes X and Y . Once again, the expressions given above are a simplified version of the results obtained with the mapping performed. These expressions are not valid for innermost meshes of the graph along the Z -axis, but this doesn't affect the resulting performance.

6.4.5 Interleaved uniform coalescing

The results obtained in Section 6.4.4 are one way to map the LU-decomposition algorithm onto the target array. An alternative consists of interleaved non-uniform coalescing, as discussed earlier.

Let's consider interleaved non-uniform coalescing along the X -axis. For $K = 10$ and $n = 200$, and considering that $N_X^i = 8 + n(n-1)/2 - (i-1)(i-2)/2$ as determined earlier, the distribution of load across cells is given in Table 6.4 (ignoring overhead due to data transfers).

Table 6.4: Load per cell in interleaved uniform coalescing along X -axis

PE	Cycles	PE	Cycles
1	275610	6	265910
2	273710	7	263910
3	271790	8	261890
4	269850	9	259850
5	267890	10	257790

Results in Table 6.4 are encouraging. However, as we stated earlier, interleaved coalescing leads to complex scheduling of primitive nodes. Such complexities are depicted in Figure 6.12, where we consider $n = 6$, $K = 2$. Highlighted nodes and edges are mapped onto one cell, while dimmed nodes and edges are mapped onto the second cell. Using the schedule described in Section 6.3.4 and depicted in Figure 6.12, we find that predecessor nodes in the MMG appear scheduled *after* successor nodes (this is a consequence of the MMG incompleteness). For example, node scheduled at time $t = 23$ in the second cell (the one executing dimmed nodes) is a predecessor of node scheduled at time $t = 21$ in the first cell. Similar examples appear throughout the MMG. Consequently, the simple and efficient schedule devised earlier can't be applied in this case and has to be replaced by a more complex one. In addition, this scheme requires bidirectional communications between cells.

In the next section, we show that non-uniform coalescing also allows achieving load balancing but without the scheduling and bidirectional communications problems found in this case. Consequently, we don't discuss this mapping approach further. It should be noted that non-uniform coalescing seems to be the technique applied in Warp to achieve load balancing while using the domain parallel model of computation [Kung88a].

6.4.6 Non-uniform coalescing

The results obtained with uniform coalescing can be improved by allocating to cells portions of the MMG of variable size. We use only coalescing along the X -axis for this approach, because it produced the best results with uniform coalescing.

Using non-uniform coalescing and the heuristic procedure to determine the size

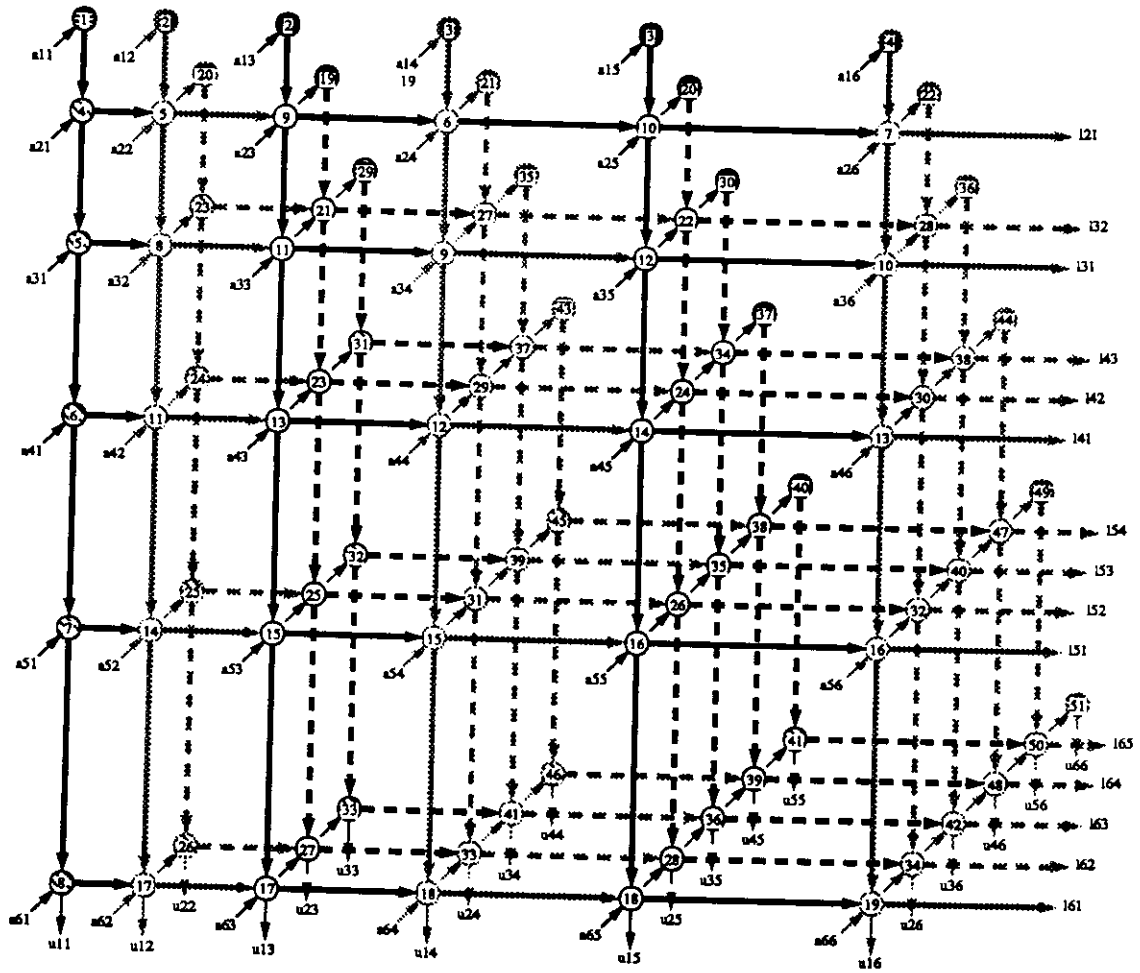


Figure 6.12: Scheduling primitive nodes in interleaved uniform coalescing

Table 6.5: Load per cell in non-uniform coalescing along X -axis

PE	Cycles	Meshes	PE	Cycles	Meshes
1	251212	200 - 149	6	287600	70 - 56
2	279947	148 - 124	7	281552	55 - 42
3	284787	123 - 104	8	290057	41 - 28
4	276025	103 - 87	9	295622	27 - 14
5	287173	86 - 71	10	278340	13 - 1

of partitions described in Section 6.3.3, we obtain the allocation of meshes to cells shown in Table 6.5; entries of this table include the overhead due to transferring values l_{ik} across cells. From this table, we infer that load balancing has been improved with respect to the best case in uniform coalescing, and is similar to that possible with interleaved uniform coalescing but without the complexities in scheduling neither bidirectional communication. Total time required to compute the LU-decomposition algorithm is given by the busiest cell, namely PE₉, with 295622 time-steps. Consequently, performance of this mapping is given by

$$t_Y = 295622[\text{time steps}]$$

$$U_Y(200) = \frac{N}{K * F * t_{PE_9}} = \frac{5314900}{10 * 2 * 295622} = 0.90$$

This mapping produces utilization that is close to the maximum possible for the LU-decomposition algorithm in the target array, as discussed when mapping the algorithm onto a single cell.

6.5 Conclusions

In this chapter, we have described the suitability of the method proposed in this dissertation for mapping algorithms onto class-specific arrays, in particular linear local-access arrays. Differences in mapping onto arrays with pseudo-systolic/systolic and local-access cells were discussed. Local-access arrays have large storage per cell, so that partitioning by coalescing has been applied. Three strategies have been discussed, which we refer to as uniform, interleaved uniform, and non-uniform coalescing.

A mapping technique that uses an explicit representation of the algorithm,

as the MMG in our method, allows analyzing the impact that different issues have on the mapping process. In this case, we have shown that the allocation of operations to cells is determined in straight-forward manner from the MMG, paying attention to load balancing. Moreover, scheduling such operations is also accomplished without difficulty from the MMG, exploiting the arithmetic pipeline within cells and taking into account overheads due to data transfers.

The analysis has been illustrated using an hypothetic memory-linked linear array, such as MWAP [Kung87b] (a commercial version of MWAP is known as the QUEN processor [Niel88]), and mapping the LU-decomposition algorithm. Results obtained here indicate that uniform coalescing is simple, but produces bad load balancing. In contrast, non-uniform coalescing allows achieving very good utilization (for example, $U = 0.90$ for an LU-decomposition of size $n = 200$ in an array with ten cells). Non-uniform coalescing requires a mechanism to determine the size of partitions allocated to cells. We have proposed a simple heuristic procedure that maps a varying number of meshes from the MMG onto cells, based on load balancing criteria.

This chapter has also shown that interleaved uniform coalescing requires bidirectional communications and cannot use a simple schedule of primitive operations, as in the other two cases. Although such an approach has the potential to reach good utilization, these drawbacks make it less attractive than non-uniform coalescing which produces similar utilization without the drawbacks mentioned.

CHAPTER 7

A comparison with other methods based on dependencies

Now that we have presented our method for the design of mesh arrays, it is of interest to analyze how this technique compares with other methods proposed in the literature, in particular with those that also use dependencies as the basis for the transformational process.

Specific dependency-based methods differ significantly in applicability, ease of use and in the resulting array characteristics. The topic of this chapter is to compare *data-dependency* graph-based techniques with methods in which dependencies are presented as index relations (i.e., *index-dependencies*). Basic properties of both classes of dependencies were given in Chapter 3. We use Rao's method [Rao85, Rao88] as a representative example of index-dependency based techniques; other methods based on index-dependencies include those described in [Quin84, Capp83, Yaac88b]. On the other hand, we use our Multi-Mesh Graph (MMG) method and S.Y. Kung Signal Flow Graph (SFG) technique [Kung88c] as data-dependency based approaches.

In Chapter 3, we proposed a framework to compare design techniques which identifies two stages in the application of any method: *algorithm regularization* and *derivation of arrays*. This framework is depicted in Figure 7.1. Moreover, we stated criteria to evaluate the suitability of design methods based on such a framework, which are listed in Table 7.1. We use the design framework and associated criteria to compare the methods indicated above, and illustrate the comparison using the transitive closure algorithm.

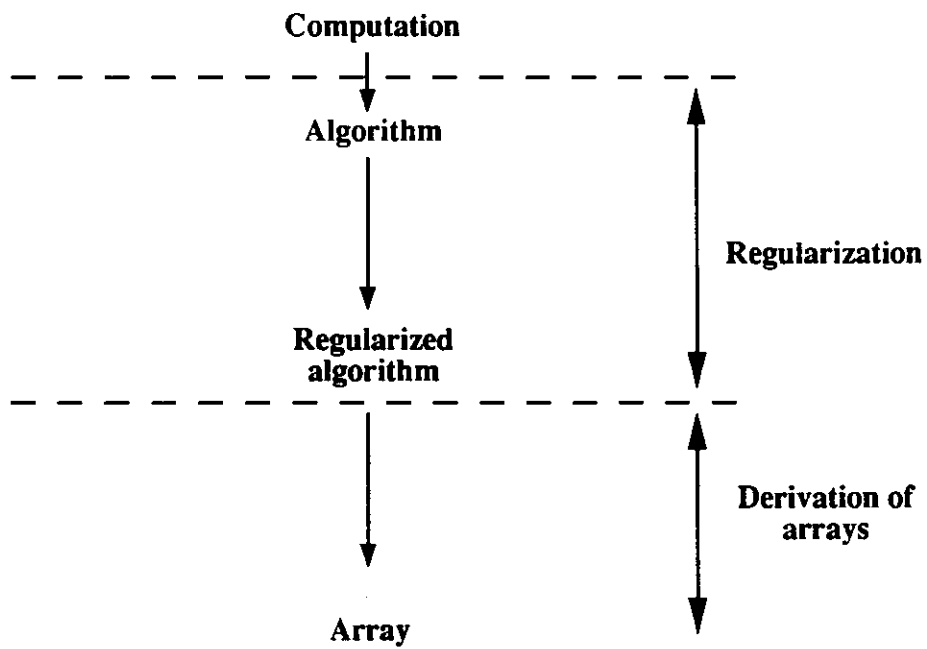


Figure 7.1: A framework to compare design methods

Table 7.1: Criteria to evaluate the suitability of design methods

Regularization stage <ul style="list-style-type: none">• General class of admissible algorithms• Capabilities to derive regularized representation• Effective regularized representation
Derivation of arrays <p>The capabilities of the method to:</p> <ul style="list-style-type: none">• Perform transformations• Incorporate implementation constraints and restrictions<ul style="list-style-type: none">• memory per cell• cell and I/O bandwidth.• Incorporate different cell attributes<ul style="list-style-type: none">• pipeline• non-conventional arithmetic• specialized functional units• Optimize specific performance/cost measures• Design arrays for fixed-size data and partitioned problems• Produce algorithm-specific and class-specific arrays
Overall <ul style="list-style-type: none">• Easy to use• Suitable for automation

7.1 The regularization of algorithms

In this section, we center our attention on the process of regularizing an algorithm so that it can be used in a design method.

7.1.1 Index-dependencies

As stated in Section 3.5, it is possible to associate the computation of each instance of a variable in an algorithm with a point in a multi-dimensional space defined by the indices in the algorithm. For this purpose, all variables must have the same number of indices. Dependencies among variables are related to the distance between those variables in the index-space, and are represented as expressions with those indices. We refer to these as *index-dependencies*.

For an arbitrary algorithm, dependencies such as those above might be complex to handle due to characteristics such as broadcasting, diverse fan-in/fan-out, or non-regularities. However, for certain classes of algorithms these dependencies exhibit a regular structure. Examples are Regular Iterative Algorithms (RIAs) [Rao88], Uniform Recurrent Equations (UREs) [Quin84], and Affine Recurrent Equations (AREs) [Yaac88b]. As stated earlier, we use RIAs as representative of index-dependency based descriptions.

A Regular Iterative Algorithm [Rao88] is defined by the triple $\{I, X, F\}$, where

- I is the index space.
- X is the set of variables defined at every point in the index space. $x_a(\vec{k})$ denotes variable x_a defined at index point \vec{k} , and takes on a unique value throughout the entire evaluation of the algorithm.
- F is the set of functional relations among the variables, restricted to be such that if $x_a(\vec{k})$ is computed using $x_b(\vec{k} - \vec{d}_{ba})$ then \vec{d}_{ba} is a constant vector independent of \vec{k} and of the extent of the index space. Moreover, for every \vec{l} contained in the index space, $x_a(\vec{l})$ is computed using $x_b(\vec{l} - \vec{d}_{ba})$.

An additional restriction not stated above is that, since the values of \vec{d}_{ba} determine the communication links in an array, nearest-neighbor connections between cells require $\vec{d}_{ba} = 1 \forall a, b$.

```

For  $i := 1$  to  $N_1$  do
  For  $j := 1$  to  $N_2$  do
    For  $k := 1$  to  $N_3$  do

```

$$c_{i,j} = c_{i,j} + a_{i,k}b_{k,j}$$

(a) Original algorithm

```

For  $i := 1$  to  $N_1$  do
  For  $j := 1$  to  $N_2$  do
    For  $k := 1$  to  $N_3$  do

```

$$a(i, j + 1, k) = a(i, j, k)$$

$$b(i + 1, j, k) = b(i, j, k)$$

$$c(i, j, k) = c(i, j, k - 1) + a(i, j, k)b(i, j, k)$$

(b) Regular Iterative Algorithm (RIA)

Figure 7.2: Regular iterative algorithm for matrix multiplication

Examples of algorithms and their respective RIAs, taken from [Rao85], are given in Figure 7.2 and Figure 7.3. These examples correspond to matrix multiplication and a two-dimensional filtering problem.

The regularization stage in Rao's method is summarized in Figure 7.4. Since the RIA is the form of the algorithm used to derive arrays, the RIA corresponds to the regularized description. For a particular computation, the question is how to derive an RIA. A few algorithms are expressed as RIAs in straight-forward manner; that is the case of matrix multiplication for example, as illustrated in Figure 7.2. However, obtaining RIAs for most other algorithms is a rather complex task. Two approaches were proposed by Rao to solve this problem, as shown in Figure 7.4: (1) synthesis by reformulating existing algorithms, and (2) synthesis by first principles [Rao85]. In the first case, an algorithm expressed as a sequence of FOR loops *might* be converted into an RIA using a three-step procedure: (a) converting the algorithm to single assignment form, (b) index-matching, and (c) localization of dependencies. Performing the last of these steps may not be easy or feasible, and the solution to this problem has been regarded as a heuristic one [Rao85, Royc88a]; some research has specifically addressed this issue [Royc88b],

$$y_{ij} = \sum_{k=1}^n a_k y_{i-k,j-k} + \sum_{k=0}^n b_k u_{i-k,j-k}$$

(a) Original algorithm

For $i := 1$ to n **do**

For $j, k := 1$ to N **do**

$$x(i, j + 1, k + 1) = f_{x,i}(x(i, j, k), y(i, j, k), w(i, j, k))$$

$$y(i + 1, j, k) = f_{y,i}(x(i, j, k), y(i, j, k), w(i, j, k))$$

$$w(i - 1, j, k) = f_{w,i}(x(i, j, k), w(i, j, k))$$

$f_{x,i}, f_{y,i}, f_{w,i}$ are linear functions determined by a synthesis procedure

(b) Regular Iterative Algorithm (RIA)

Figure 7.3: Regular iterative algorithm for a two-dimensional filtering problem

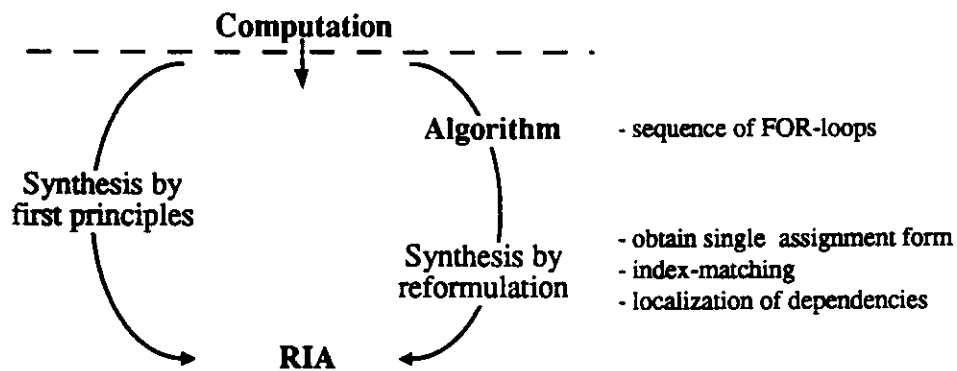


Figure 7.4: The regularization stage in Rao's method

For k **from** 1 **to** n
 For i **from** 1 **to** n
 For j **from** 1 **to** n
 $x_{ij}^k \leftarrow x_{ij}^{k-1} \oplus x_{ik}^{k-1} \otimes x_{kj}^{k-1}$
 (a) Warshall's algorithm

For i **from** 1 **to** n
 For j **from** $[\max(i, n) - n + 1]$ **to** $[\min(i, n) + n]$
 For k **from** 1 **to** $\min(i, j, n)$

$$a(i, j + 1, k) = \begin{cases} a(i, j, k) & \text{if } j \neq k \\ c(i, j, k) \oplus [a(i, j, k) \otimes b(i, j, k)] & \text{if } j = k \end{cases}$$

$$b(i + 1, j, k) = \begin{cases} b(i, j, k) & \text{if } i \neq k \\ c(i, j, k) \oplus [a(i, j, k) \otimes b(i, j, k)] & \text{if } i = k \end{cases}$$

$$c(i, j, k + 1) = c(i, j, k) \oplus [a(i, j, k) \otimes b(i, j, k)]$$

 (b) Regular iterative algorithm (RIA)

Figure 7.5: Regular iterative algorithm for transitive closure

but the effectiveness of the solutions is not yet clear. On the other hand, synthesis by first principles seeks to find an RIA by considering a problem afresh.

The latter of the two approaches above was used in [Rao85] to derive an RIA for the transitive closure computation; the corresponding RIA is shown in Figure 7.5. Rao formulated transitive closure by relating it to “a certain smoothing problem in image processing.” In such a problem, pixels in a black-and-white image composed of (N by N) pixels have intensity $a(i, j)$, and the output image is required to be such that the intensity at pixel (i, j) is equivalent to the transitive closure of element $a(i, j)$.

Since the dependency structure in an RIA is identical at every point in the index space, these dependencies may be represented by a Reduced Dependency Graph (RDG). This is a directed graph, where nodes correspond to variables in

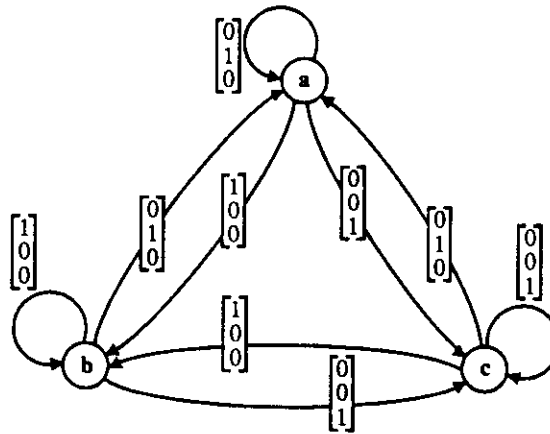


Figure 7.6: Reduced dependency graph for transitive closure

the algorithm and edges represent dependencies between pairs of variables. As an example, Figure 7.6 depicts the RDG for the transitive closure algorithm given in Figure 7.5. Edges in the RDG are tagged with a data-dependency vector, which corresponds to the difference between indices of the corresponding pair of variables.

RIAs seem an attractive regular description of algorithms, due to their compactness and suitability for manipulation. However, an analysis of the process of obtaining RIAs indicates the following limitations of such a description:

- Transforming an algorithm into an RIA might add computing load, in terms of additional variables and operations. For example, the RIA shown in Figure 7.5 has introduced two additional computed variables that do not exist in Warshall's algorithm for transitive closure, plus evaluation of conditionals and operations on the extra variables. Consequently, implementing the RIA implies performing more operations than those present in Warshall's algorithm. More dramatic cases of added computing load are found for example in Gaussian elimination, with and without pivoting, as obtained in [Rao85] and [Royc88a], respectively. These two RIAs were obtained by reformulating existing algorithms. Figure 7.7 depicts the original Gaussian elimination algorithm with partial pivoting and the resulting RIA derived in [Royc88a].¹
- Currently, there is no systematic technique to obtain an RIA for a given algorithm, in spite of the procedure indicated earlier. Attempts to solve specific issues have been reported in [Royc88a, Dong88b, Royc88b].

¹The original algorithm shown here is modified from the one given in [Royc88a], because that one appears to have some errors.


```

For  $k := 1$  to  $(m - 1)$  do
  Determine  $p \in \{k, k + 1, \dots, n\}$  so  $|a_{pk}| = \max_{k \leq i \leq n} |a_{ik}|$ ;
  For  $i := k$  to  $n$  do
    Swap  $a_{ki}$  and  $a_{pi}$ 
  For  $i := (k + 1)$  to  $m$  do
     $\eta := (a_{ik}/a_{kk})$ ;
    For  $j := (k + 1)$  to  $n$  do
       $a_{ij} = a_{ij} - \eta a_{kj}$ 

```

a) Sequential representation

For all triples $(i, j, k), 1 \leq i \leq M; k \leq j \leq N$ and $1 \leq k \leq M - 1$ **do**

$$\begin{aligned}
 s(i, j, k) &= \overline{t(i, j, k)} \times s(i - 1, j, k) + t(i, j, k) \times a(i, j, k) \\
 t(i, j, k) &= \begin{cases} p(i - 1, j, k) < |a(i, j, k) \times \overline{c(i, j, k)}| & \text{if } j = k \\ t(i, j - 1, k) & \text{if } j > k \end{cases} \\
 r(i, j, k) &= \begin{cases} r(i - 1, j, k) \times \overline{t(i, j, k)} + i \times t(i, j, k) & \text{if } j = k \\ \text{null} & \text{otherwise} \end{cases} \\
 w(i, j, k) &= \begin{cases} s(i, j, k) & \text{if } i = m \\ w(i + 1, j, k) & \text{if } i \leq n \end{cases} \\
 x(i, j, k) &= \begin{cases} r(i, j, k) & \text{if } i = m \text{ and } j = k \\ x(i + 1, j, k) & \text{if } j = k \\ x(i, j - 1, k) & \text{if } j > k \end{cases} \\
 \rho(i, j, k) &= \begin{cases} \frac{a(i, j, k)}{w(i, j, k)} \times \overline{c(i, j, k)} \vee (x(i, j, k) = i) & \text{if } j = k \\ \rho(i, j - 1, k) & \text{if } j > k \end{cases} \\
 c(i, j, k + 1) &= c(i, j, k) \vee (i = x(i, j, k)) \\
 a(i, j, k + 1) &= a(i, j, k) - \rho(i, j, k) \times w(i, j, k)
 \end{aligned}$$

b) Regular iterative algorithm (RIA) [Royc88a]

Figure 7.7: Algorithms for Gaussian elimination with partial pivoting

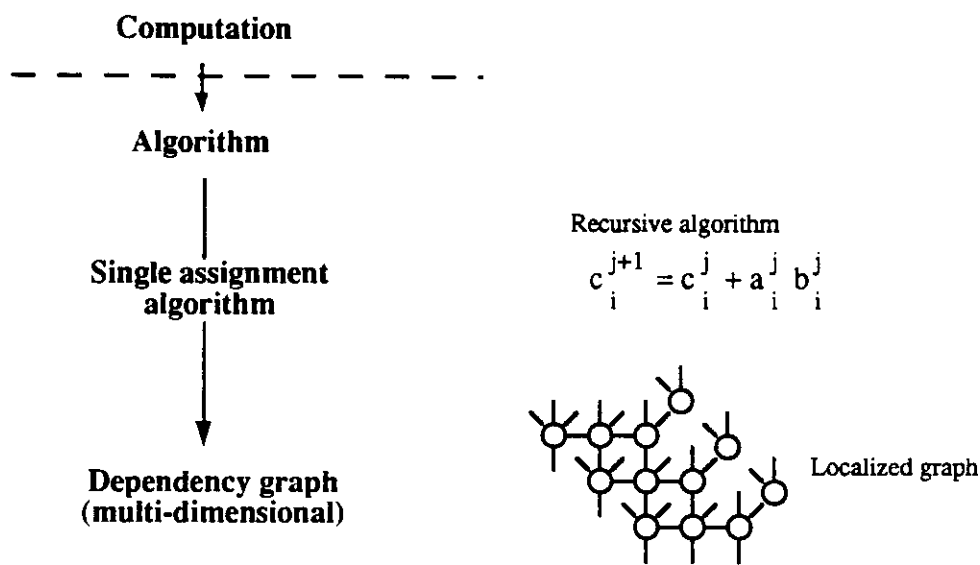


Figure 7.8: The regularization stage in the Signal Flow Graph method

Because of these limitations, the representation of algorithms as RIAs has restricted applicability.

7.1.2 S.Y. Kung's method

The design method proposed by S.Y. Kung [Kung87a, Kung88c] uses data-dependencies in an algorithm, although this fact has not been recognized explicitly and some of the steps in the process seem to indicate otherwise. The regularization stage in this method, summarized in Figure 7.8, consists of identifying a suitable algorithm expression (i.e., a single assignment representation) and generating a dependency graph. The resulting graph is used to derive arrays, so that it corresponds to the regularized representation.

This method does not state what are the characteristics of the dependency graph (DG), implying that it may have almost any structure. In fact, Kung states that “the structure of a DG greatly affects the final array design” so that “further modifications on the DG are often desirable in order to achieve a better design” [Kung88c] (pp.119). However, the general objectives of those further modifications are not indicated and, as a consequence, the process to derive the regularized form is carried-out with ad-hoc transformations.

Since DGs may exhibit diverse characteristics, Kung identifies two different

$$\begin{aligned}
& \text{For } i, j, k \text{ from } 1 \text{ to } n \\
& c(i, j, k) \leftarrow \begin{cases} x(i, j, k-1) & \text{if } j = k \\ c(i, j+1, k) & \text{if } j < k \\ c(i, j-1, k) & \text{if } j > k \end{cases} \\
& r(i, j, k) \leftarrow \begin{cases} x(i, j, k-1) & \text{if } i = k \\ r(i+1, j, k) & \text{if } i < k \\ r(i-1, j, k) & \text{if } i > k \end{cases} \\
& x(i, j, k) \leftarrow x(i, j, k-1) \oplus r(i, j, k) \otimes c(i, j, k)
\end{aligned}$$

Figure 7.9: Single assignment algorithm for transitive closure

design strategies: a *canonical* method for homogeneous DGs (i.e., shift-invariant), and a *generalized* method for heterogeneous DGs. The claim is that a large number of algorithms may be expressed in terms of a regular and localized DG, and the canonical method exploits this regularity to obtain simple and regular structures.

Kung regards the dependency graph of an algorithm as a multi-dimensional one (the method addresses multi-dimensional projections in the second stage). This concept has arisen from interpreting the DG as a representation of index-dependencies rather than data-dependencies (recall that an index-dependency graph has one dimension for each index in the algorithm, potentially leading to graphs with many dimensions).

Let us review the regularization stage in this method for the transitive closure algorithm, as done in [Kung87c, Kung88c]. Starting from the sequential Warshall's algorithm, the method obtains the single assignment form shown in Figure 7.9. This localized representation is derived by adding propagating variables r for rows and c for columns at each iteration, and using these new variables in the computation of the expression that composes the algorithm.

The dependency graph for the single assignment algorithm is shown in Figure 7.10, where nodes compute three variables. Since this graph exhibits bidirectional flow of data, Kung et al. state that there is no possible systolic schedule for it. To get around this problem, they reindex the nodes in the DG until the graph becomes a more regular one with possible systolic schedules. For this purpose, they introduce the following reindexing:

$$\text{node}(i, j, k) \rightarrow \text{node}(((i - k) \bmod N) + 1, ((j - k) \bmod N) + 1, k)$$

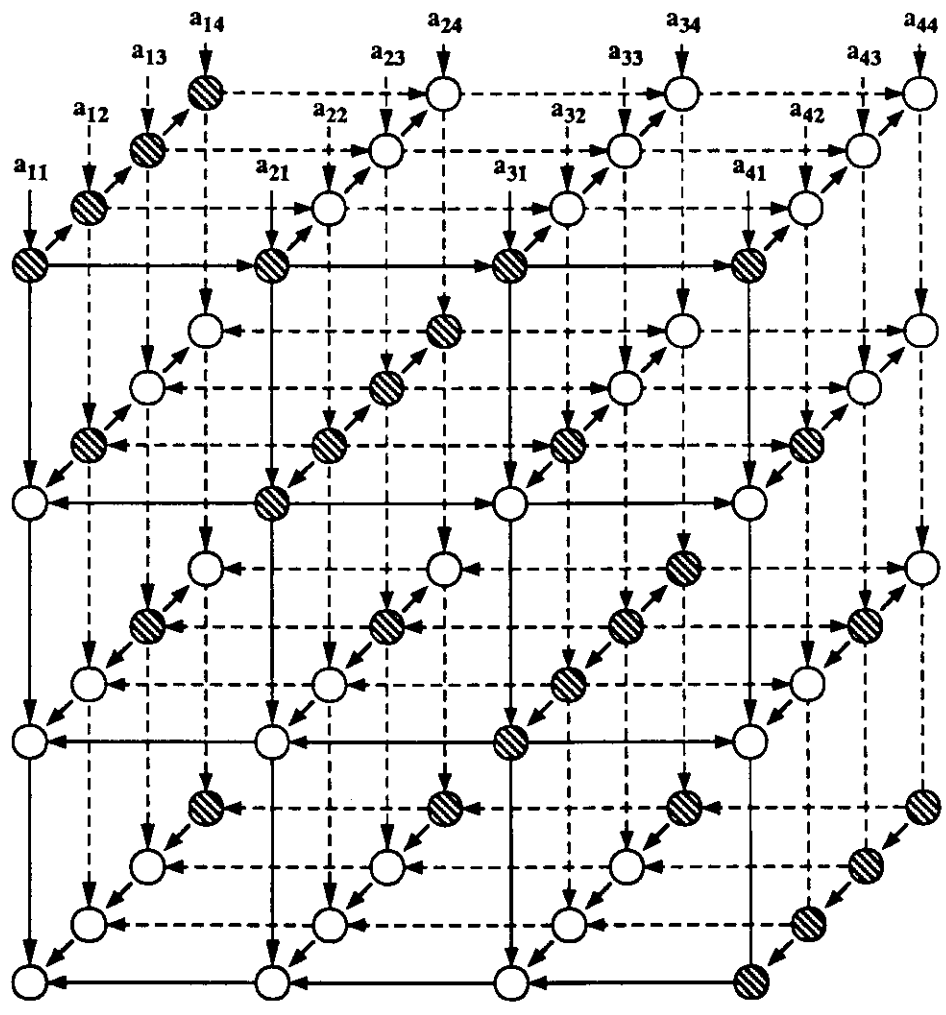


Figure 7.10: The dependency graph for transitive closure in the SFG method

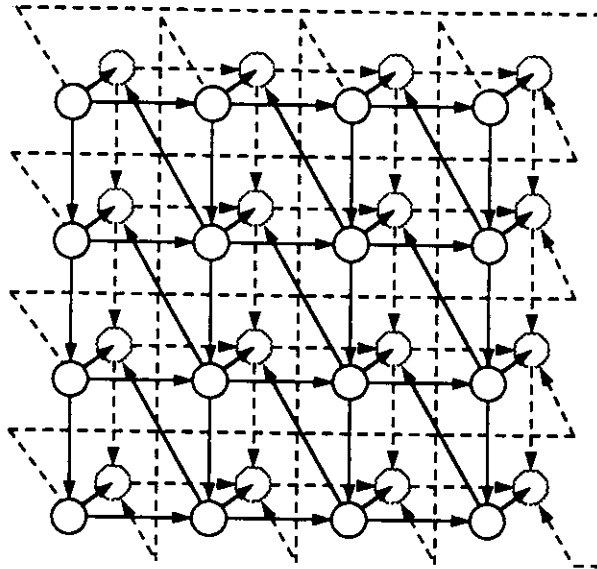


Figure 7.11: Reindexed dependency graph

The reindexing arranges the ij -planes in such a way that the nodes that are sources of transmittent data appear at the left and innermost part of each plane. Edges within the planes remain among nearest-neighbors. On the other hand, edges between planes become less regular; these edges are either diagonal ones (direction $[-1,-1,1]$) or spiral ones. Two planes of the resulting graph are depicted in Figure 7.11.

The reindexed graph is used to derive arrays in the second stage of the method, but it leads to complex and not efficient structures (spiral links and low cell utilization). To overcome these limitations, Kung et al. perform modifications to the graph to obtain another one that is more convenient. This modification takes advantage of properties in the algorithm (operations that are not needed and transmittent data) to remove the spiral edges. The new graph is shown in Figure 7.12. The fact that the graph in Figure 7.11 as well as the graph in Figure 7.12 are used for the second stage of the method is a clear indication that this technique has no precise form for the regularized representation.

From the discussion above, we infer that the SFG method has the following drawbacks in terms of algorithm regularization:

- It does not state what is the form of the regularized description.
- It does not provide a systematic mechanism to derive the DG.
- The intent of modifications on a DG to obtain a better one are not stated, so

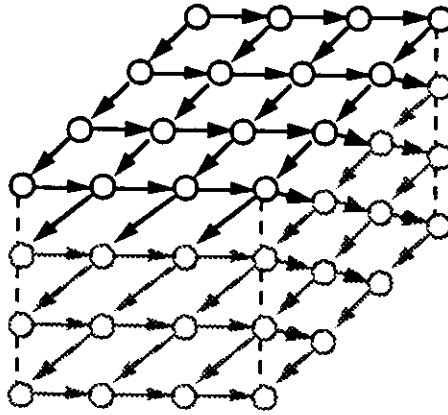


Figure 7.12: Modified DG for the transitive closure algorithm

that devising them is not clear. From the examples given, these are ad-hoc transformations.

7.1.3 MMG method

The design method proposed in this dissertation represents dependencies in an algorithm using an Explicit Dependency Graph (EDG), that is, a graph where each node represents a primitive operation and each edge corresponds to a dependency among two operations. We refer to these as *data-dependencies*, and to the graph as the *fully-parallel data-dependency graph* (FPG), which corresponds to a single assignment representation of an algorithm.

As stated in Chapter 5, the FPG of an arbitrary algorithm might exhibit no structure so that edges are long and intersecting in many places. However, matrix and vector operators in a matrix algorithm allow obtaining a regular graph, which we refer to as a *multi-mesh dependency graph* (MMG). Moreover, we have provided transformations that allow obtaining such an MMG from an FPG, exploiting the graphical representation of the algorithm.

The regularization stage in our method is summarized in Figure 7.13. The MMG is the description used to derive arrays, so that it corresponds to the regularized form. This representation is obtained through a systematic procedure, as discussed in Chapters 4 and 5.

Let us illustrate the regularization of the transitive closure algorithm in our method. Figure 7.14 depicts the FPG for a problem of size $n = 4$, which is obtained

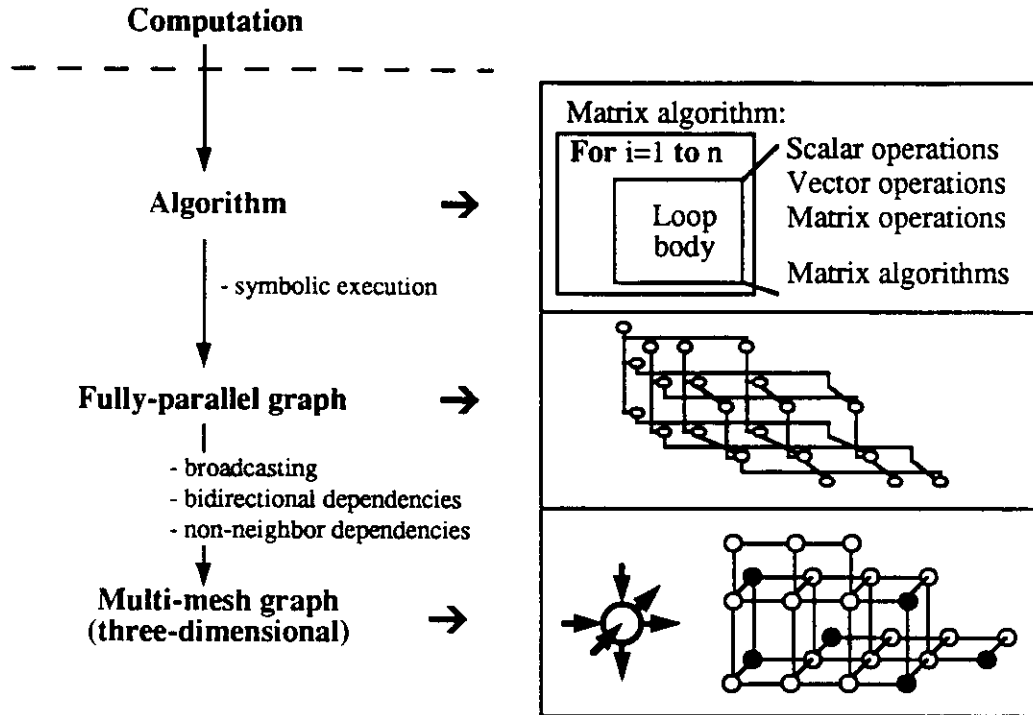


Figure 7.13: The regularization stage in the MMG method

from the symbolic execution of Warshall's algorithm. This FPG is characterized by many broadcasted elements. In addition, one can identify several operations that are superfluous (which have been highlighted in the figure), because the result is equal to one of the input operands. This property, a consequence of primitive operations AND/OR, is dependent on the specific algorithm but it serves to illustrate the capabilities of an explicit description. Superfluous operations may be removed if that is advantageous for an implementation (i.e., if it simplifies the resulting array).

The FPG shown in Figure 7.14 consists of n levels, where each level corresponds to one iteration of the outer-most loop in the original Warshall's algorithm. At each level, there is global and local broadcasting. Global broadcasting corresponds to data that is broadcasted throughout the entire level, while locally broadcasted data reaches only a portion of the level. Sources of broadcasting change from level to level; at the k -th level of the graph, the k -th row of matrix data, as well as the k -th element of each row, are broadcasted.

Regularizing the graph in Figure 7.14 consists of replacing data broadcasting

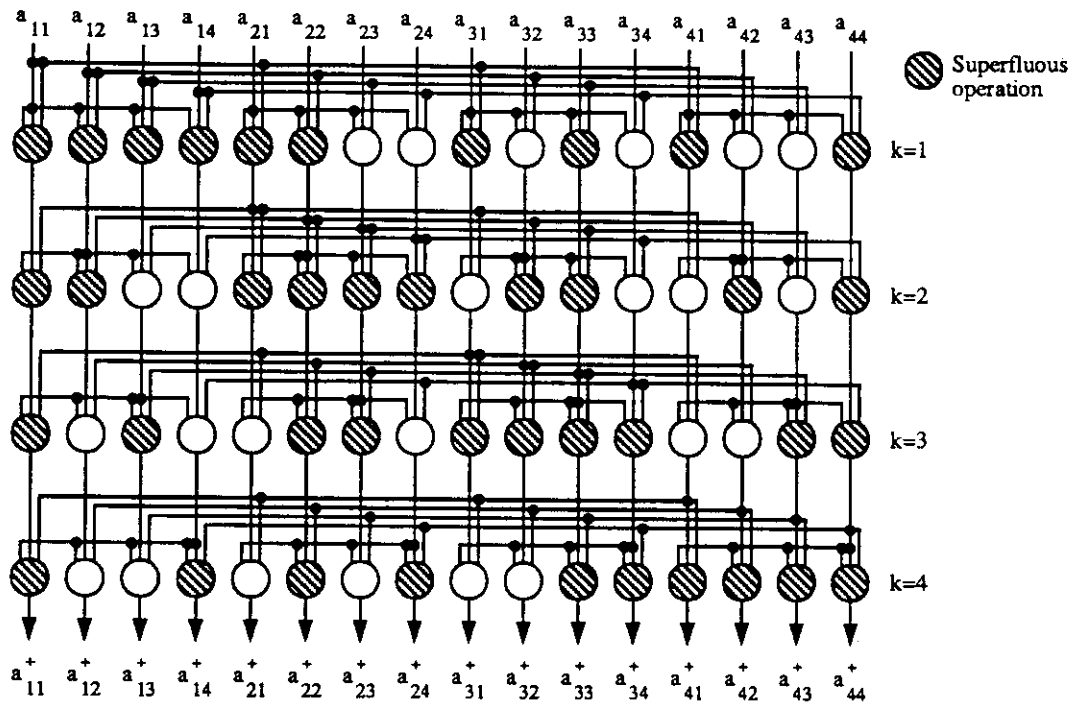


Figure 7.14: The FPG for the transitive closure algorithm

by transmittent data, drawing the graph as a three-dimensional structure, removing bidirectional flow of transmittent data, and adding delay nodes to make all dependencies among nearest-neighbor nodes. The specific transformations used to remove these characteristics from the FPG have been presented in Chapter 4 and formalized in Chapter 5. We describe this process in detail now.

Eliminate data broadcasting

We first transform the FPG by replacing data broadcasting by transmittent data. Globally broadcasted data is drawn orthogonal to the flow of locally broadcasted elements, in a three-dimensional structure. The resulting graph, shown in Figure 7.15, is a three-dimensional graph that does not fulfill the requirements of an MMG. This graph has the same dependency structure as the one used in the SFG method, shown in Figure 7.10; however, nodes in the graph derived with the MMG method compute only one variable (instead of three in the SFG method).

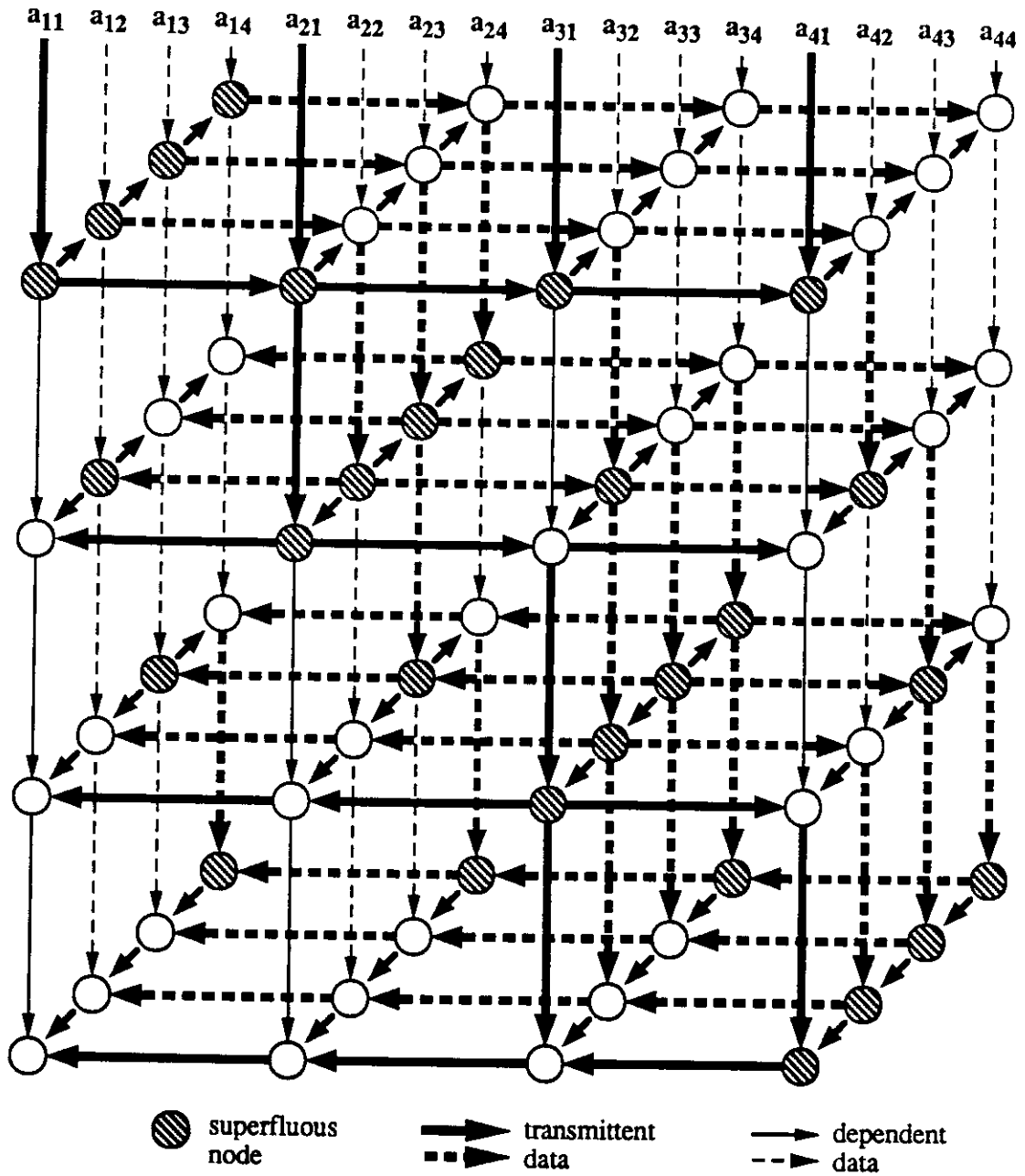
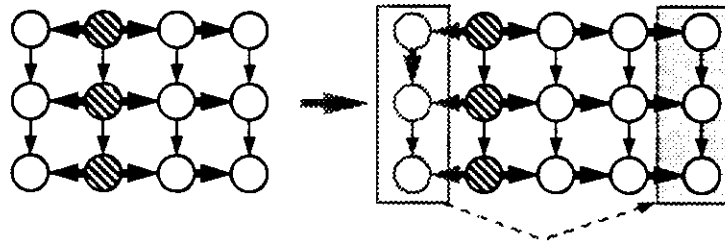
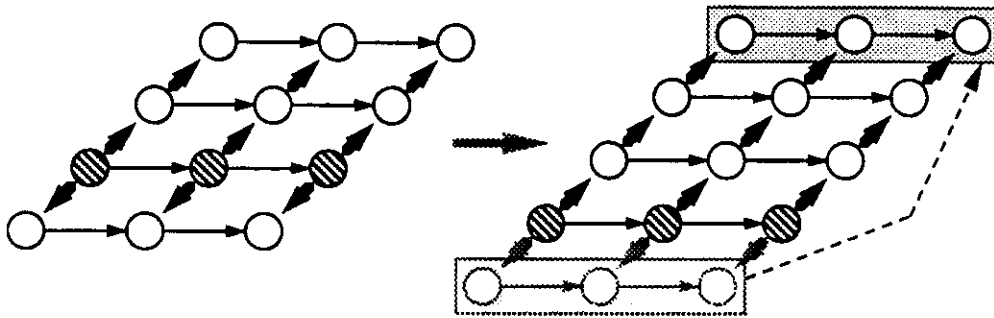


Figure 7.15: Replacing broadcasting by transmittent data



(a) Removing bi-directional transmittent data along X-axis



(b) Removing bi-directional transmittent data along Z-axis

Figure 7.16: Removing bidirectional transmittent data

Eliminate bidirectional transmittent data

Due to the varying source of broadcasting, the graph in Figure 7.15 exhibits bidirectional flow of transmittent data. As indicated in Section 5.5, this undesirable property can be eliminated if all nodes at one side of the source of transmittent data are part of a movable subgraph. The graph in Figure 7.15 fulfills this requirement, so that nodes may be moved to one side of the source of transmittent data. This transformation is applied in two steps: first, nodes to the left of sources of horizontal transmittent data are flipped to the right end of each level of the graph, as shown in Figure 7.16a. The application of this transformation leads to the graph in Figure 7.17. Then, nodes in front of sources of transmittent data along the Z -axis are flipped to the end of this direction, as shown in Figure 7.16b. The graph obtained as a result of this last transformation is shown in Figure 7.18.

Remove non-nearest neighbor dependencies

Figure 7.18 still exhibits one characteristic that is not allowed in a multi-mesh dependency graph: dependencies between nodes at the boundaries of the

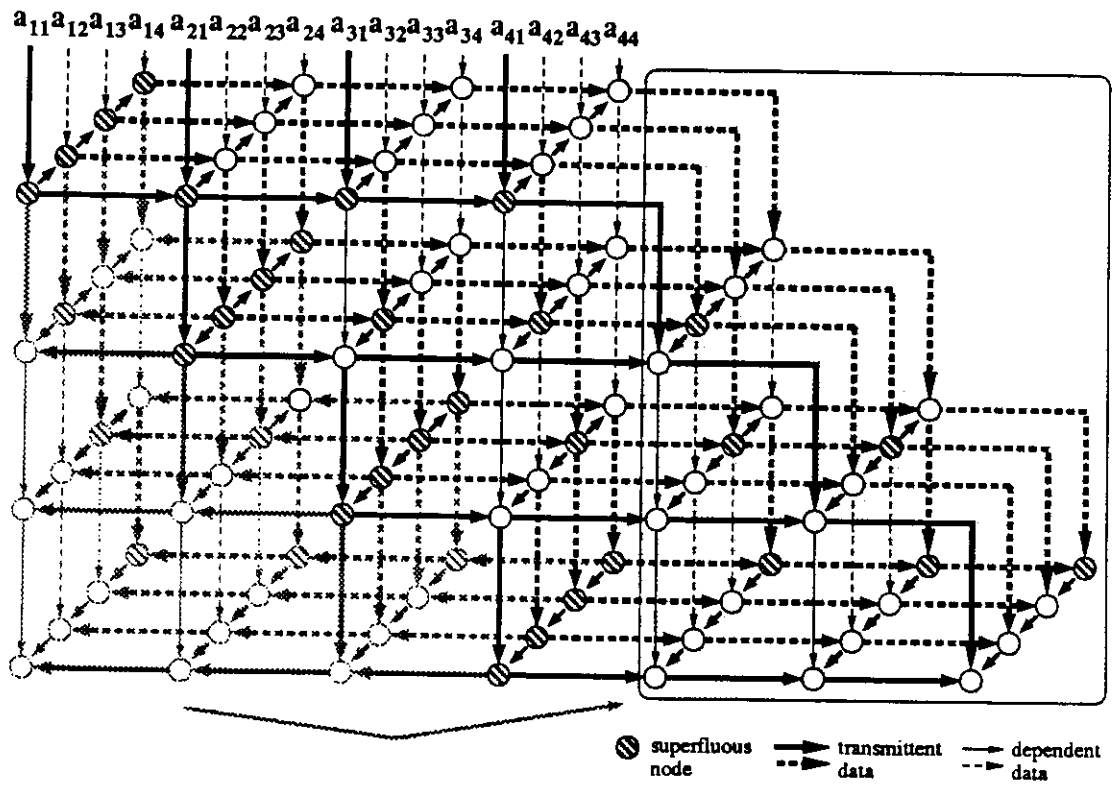


Figure 7.17: Removing bidirectional transmittent data along X-axis

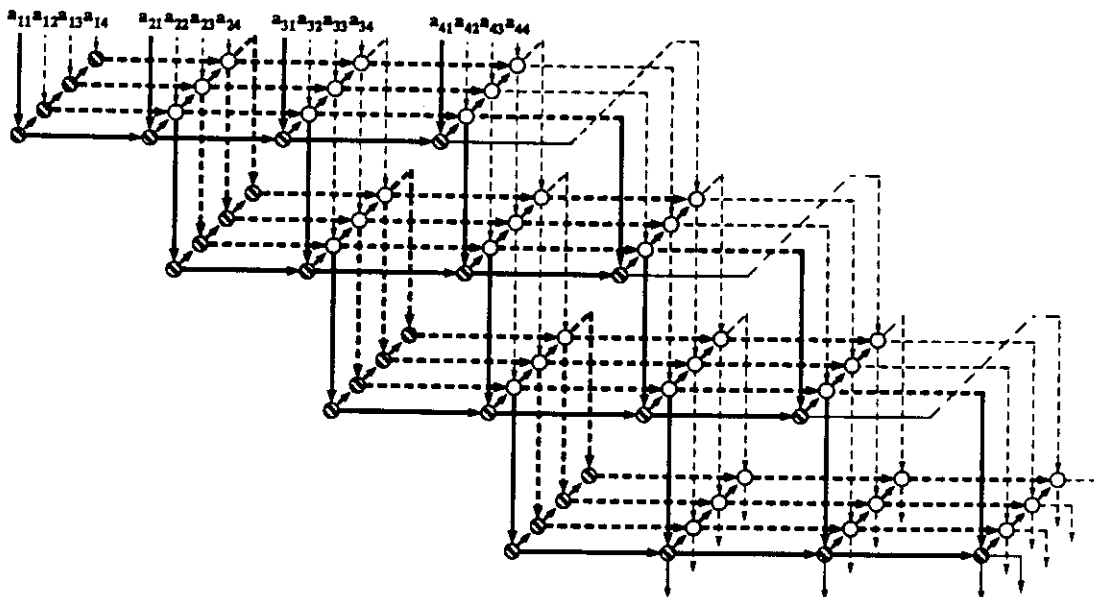


Figure 7.18: Unidirectional dependency graph

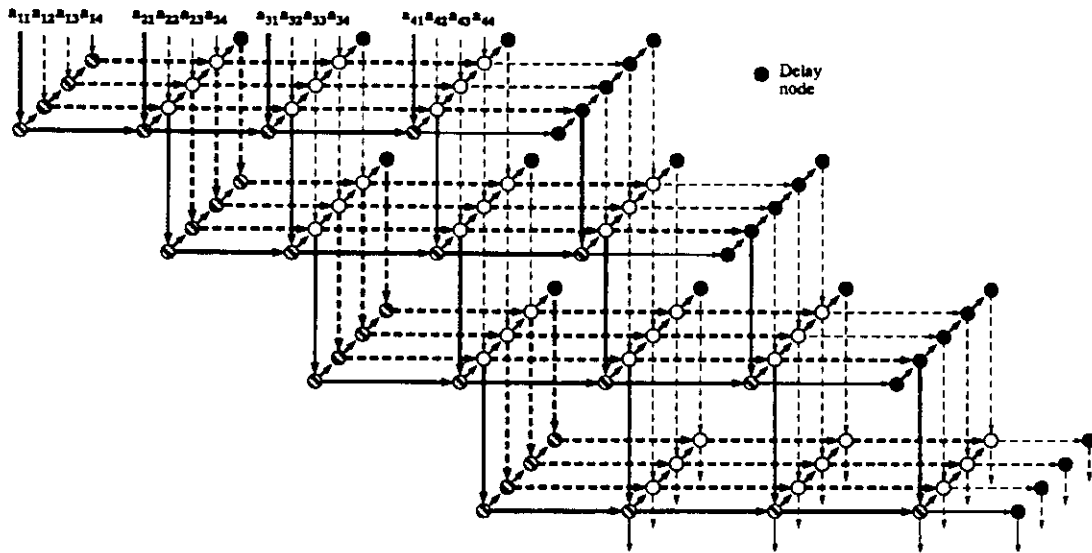


Figure 7.19: Multi-mesh dependency graph

three-dimensional structure are not between nearest-neighbors. As indicated in Section 5.5, this irregularity is eliminated by adding delay nodes in the vacant positions, leading to the multi-mesh dependency-graph shown in Figure 7.19. In this figure, we have also replaced superfluous nodes by delay nodes.

From the discussion above, one can infer that the MMG is advantageous in describing an algorithm, because the MMG provides information on all operations and dependencies without imposing constraints on the form of the algorithm. Moreover, transforming an algorithm described by an FPG into a regular MMG is performed in a simple and systematic manner, by using the transformations described in Chapter 5 and taking advantage of the graphic capabilities offered by the data-dependencies.

7.2 The derivation of arrays

Once an algorithm has been regularized, the next step is to derive arrays for it using a transformational paradigm. Such transformations should be applied considering the criteria indicated in Table 7.1. In this section, we analyze how this stage of the design is performed by the methods being compared.

7.2.1 Rao's method

Rao's method implicitly considers systolic arrays with identical cells as the target for an implementation. At each time step, each cell computes all expressions appearing in the RIA [Rao88]. Consequently, the method does not need to describe features of the architecture.

The design process in this method attempts to exploit the parallelism available in the RIA, while at the same time minimizing the resources used in the implementation. To achieve these objectives and render the problem tractable, Rao restricts his attention to *linear* partitions of the set of computations which produce the *processor space*. Once the processor space is decided upon, a *linear schedule* with respect to the index vectors is used to determine the schedule of operations assigned to each cell. The process can be described as follows [Rao88]:

1. A set of parallel lines is drawn through the index space, so that all computations corresponding to index points that lie on the same line are executed in the same cell. An array (including communication links) is obtained by projecting the embedded dependency graph of the RIA along these lines onto a lower dimensional lattice of points known as the processor space. The direction along which this projection is made is represented by an integer vector \vec{u} , and is defined as the *iteration vector*.
2. Once the processor space is decided upon, computations mapped onto a given cell are scheduled. That is, a "time slot" is assigned to each variable (with respect to a global reference time) during which its computation is performed by the cell.

The choice of schedule is constrained both by dependencies in the algorithm and by the choice of processor space.

Formally, the two steps described in the preceding paragraphs are performed

as transcribed below [Rao88]:

1. The S -dimensional iteration vector \vec{u} defines the topology of the array. Two index points \vec{k}_1 and \vec{k}_2 are mapped onto the same cell if and only if $\vec{k}_1 - \vec{k}_2 = \alpha\vec{u}$, where α is some scalar integer. This means that \vec{k}_1 and \vec{k}_2 map on the same point when the index space is projected along the direction defined by \vec{u} .

Let P be any $(S - 1) \times S$ -dimensional integer matrix of rank $(S - 1)$ that is orthogonal to \vec{u} (i.e., $P\vec{u} = 0$). Then, the array is defined by the lattice of points obtained by mapping the index space according to $\vec{p} = P\vec{k}$, where $\vec{k} \in$ index space.

In the last expression above, \vec{p} defines the location of the processor that carries out the computation at index point \vec{k} . The necessary communication links are defined by the vector weights on the edges in the RDG. If $y(\vec{k})$ is dependent on $x(\vec{k} - \vec{d})$, then there must be a directed link $P(\vec{k} - \vec{d}) \rightarrow P\vec{k}$ in the array, for all \vec{k} .

2. Once the array is obtained as described above, computations have to be scheduled. To avoid having to know the capabilities of cells, a schedule that partitions computations into global steps is devised, with the following restrictions:

- If variable $y(\vec{k})$ is computed using variable $x(\vec{k} - \vec{d})$, then step $s_y(\vec{k})$ at which $y(\vec{k})$ is computed must be strictly larger than step $s_x(\vec{k} - \vec{d})$ to which $x(\vec{k} - \vec{d})$ is assigned. That is, $s_y(\vec{k}) \geq s_x(\vec{k} - \vec{d}) + 1$.
- All computations at step τ are completed by every cell in the array before step $(\tau + 1)$ is begun.
- At each step, each cell must be assigned a “small” number of computations.

A *uniform affine* schedule is chosen, where $s_x(\vec{k}) = (\vec{\lambda}^T \vec{k} + \gamma_x)$, $\vec{\lambda}$ is a constant vector, independent of x , whereas γ_x is a scalar that is specific to x . Then, $(\gamma_y - \gamma_x + \vec{\lambda}^T \vec{d}) \geq 1$, and this must be true for all such dependencies, that is, for every directed edge in the RDG. Expressed in matrix form, these constraints can be written as $(\vec{\gamma}^T C + \vec{\lambda}^T D) \geq [11 \cdots 1]$, where

- C is the *edge-vertex incidence matrix* or the *connection matrix* commonly found in circuit analysis.
- D is the $(S \times E)$ -dimensional *index displacement matrix*.

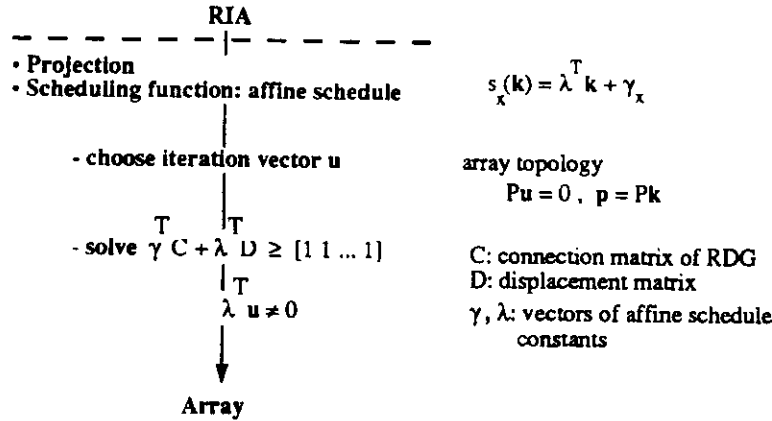


Figure 7.20: The derivation of arrays in Rao's method

- $\bar{\gamma}$ is the vector obtained by stacking $\{\gamma_x\}$ in the appropriate order, consistent with the arrangement of the rows in the connection matrix.
3. If $x(\vec{k}_1)$ and $x(\vec{k}_2)$ are computed by the same cell, then they must not be assigned to the same step in the schedule. This implies that $(\gamma_x + \bar{\lambda}^T \vec{k}_1 \neq \gamma_x + \bar{\lambda}^T \vec{k}_2)$, for all $(\vec{k}_1 - \vec{k}_2) = \alpha \vec{u}$, which simplifies to $\bar{\lambda}^T \vec{u} \neq 0$. This constraint is dependent upon the choice of the iteration vector \vec{u} .

From linear programming considerations, it can be shown that if there exists a feasible solution to the set of constraints $(\bar{\gamma}^T C + \bar{\lambda}^T D) \geq [1 \ 1 \ \dots \ 1]$, then there always exists a feasible solution that meets the additional constraint $\bar{\lambda}^T \vec{u} \neq 0$.

The procedure to derive arrays reproduced above from [Rao88] is summarized in Figure 7.20.

Let us review how an array for the transitive closure algorithm is derived in Rao's method, as described in [Rao85]. The RIA for this algorithm was shown in Figure 7.5. First, the iteration vector $[0 \ 0 \ 0]^T$ is chosen (though it is not indicated why), and then a linear scheduling function is obtained by solving $(\bar{\gamma}^T C + \bar{\lambda}^T D) \geq [1 \ 1 \ \dots \ 1]$ subject to the condition $\bar{\lambda}^T \vec{u} \neq 0$ (these steps are not shown in detail in [Rao85]). The resulting array is shown in Figure 7.21, which has approximately $3n^2$ cells; each cell computes all the operations that appear in the RIA of Figure 7.5.

The procedure and example described above show that Rao's method relies on algebraic descriptions for algorithm, transformations (recall that the architecture is implicit in this technique). The algebraic nature of transformations leads to

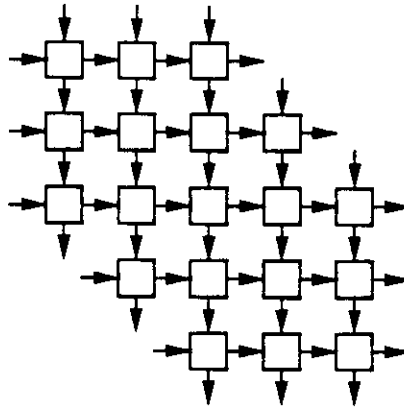


Figure 7.21: The array for transitive closure in Rao's method [Rao85]

wards procedures that could be automated. This is an advantage of the approach, a consequence of its highly specialized area of application. Moreover, these transformations potentially allow finding many solutions.

However, the restricted scope of Rao's method is a major drawback, because transformations have limited capabilities. That is, the strict requirements on the architecture imply that the method can deal with only a limited set of design issues. For example, this technique assumes that the implementation is a systolic-type architecture, and no considerations or tradeoffs are possible among implementation parameters such as size of local storage and limited cell bandwidth. Moreover, transformations and optimality criteria are very distant from an implementation, so that they are difficult to use by a designer that needs to devise an array with actual requirements and constraints.

In addition, approaches such as Rao's are not suitable to map algorithms onto class-specific arrays, due to the inability to incorporate properties of the array as part of the method. In such a case, the architectural model is not implicit and has to be taken into account. Moreover, this method does not have built-in capabilities for partitioning problems.

7.2.2 S.Y. Kung's method

Let us review now the process of deriving arrays in Kung's method. The procedure, which is summarized in Figure 7.22, has the following two steps:

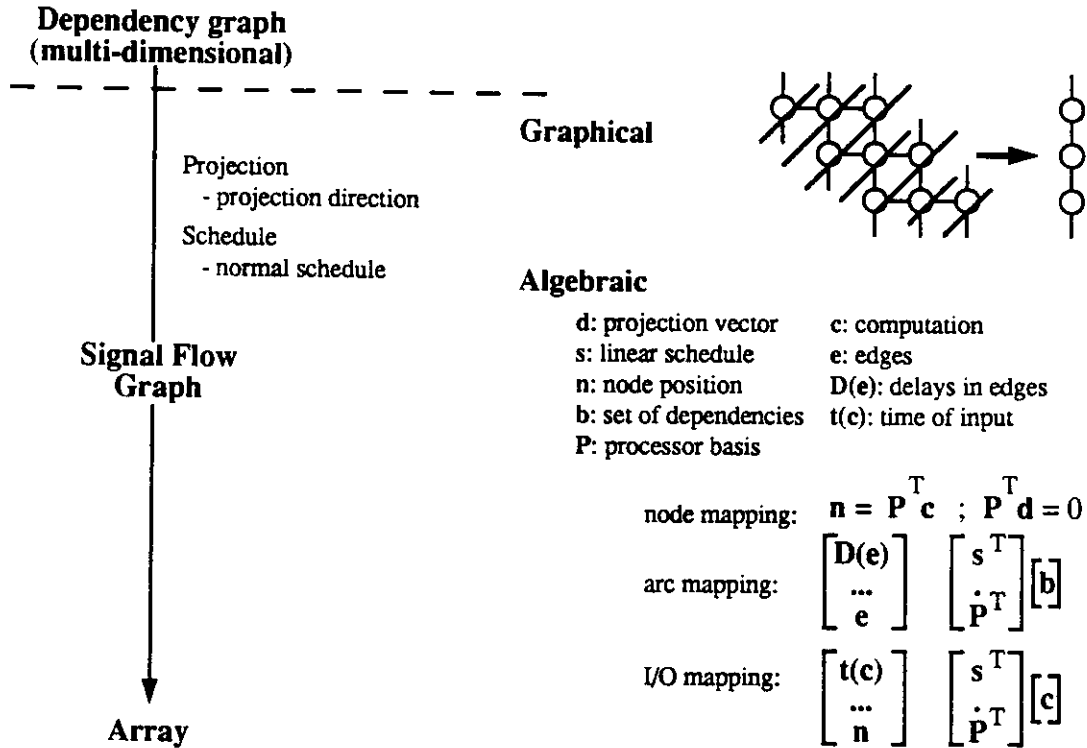


Figure 7.22: The derivation of arrays in the SFG method

- Derive a Signal Flow Graph (SFG) from the dependency graph (DG) by projecting the DG.
- Realize the SFG as an array.

In addition to using the graphical representation of dependency graphs and projections, this method also includes an algebraic approach to derive arrays. Such an approach follows the same principles that appear in Rao's technique, as discussed earlier, and combines the two steps above into a single one. The algorithm is represented by the dependency graph, and dependency vectors extracted from that graph are used in the algebraic manipulations.

Deriving a SFG requires to choose a suitable projection direction. However, the method does not provide tools to aid in such a selection. In [Kung88c] (pp. 157), while discussing the derivation of arrays for LU-decomposition, it is stated: "the SFGs derived by different projection directions may have substantially different properties. In order to find an optimal projection, we may try several directions and see how the results are." In other words, exhaustive search is being suggested. It should be pointed out that, conceptually at least, there is a large number of possible

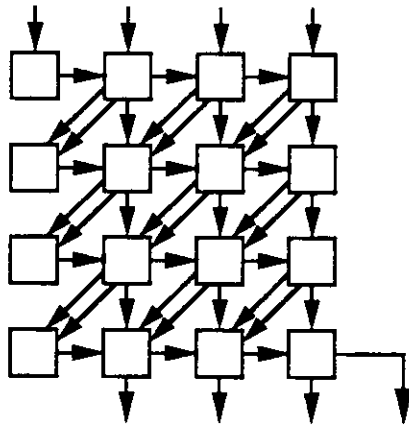


Figure 7.23: Array for transitive closure from S.Y. Kung's method [Kung87c]

projection directions. In practice, most of the examples given use projections along axes of a three-dimensional space.

Derivation of arrays for the transitive closure algorithm was reported by Kung et al. in [Kung87c]. They used the algebraic rather than the graphical capabilities of the method, because they found it difficult to determine a permissible schedule vector from the graph in Figure 7.12 by simple observation. Consequently, the array is derived directly from the dependency graph without obtaining an SFG. The array depicted in Figure 7.23 is obtained after choosing projection vector $d = [1 \ 0 \ 0]^T$ and solving the expressions indicated in Figure 7.22. This array has n^2 cells, two diagonal links between cells (with one and two time-steps of delay in each one, respectively), and requires some control to load data in cells and reuse that data for n time-steps.

7.2.3 MMG method

The transformational process to derive arrays in our data-dependency based method is summarized in Figure 7.24. Since the method was described in Chapters 4 and 5, we just summarize it here and immediately apply it to the transitive closure. This method consists of the following steps:

- *Collapsing* the MMG onto a two-dimensional G-graph by grouping *prisms* of primitive nodes onto different *G-nodes*. As described in Chapter 5, grouping along axes of the three-dimensional space leads to simpler and more efficient

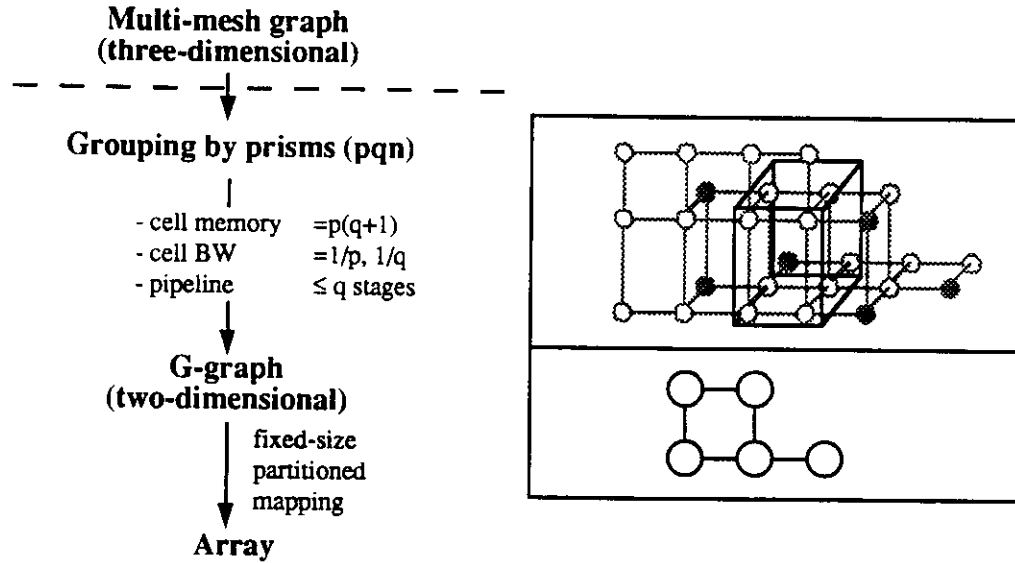


Figure 7.24: The derivation of arrays in the MMG method

implementations, so that selecting the direction of collapsing is limited to three alternatives, one along each of the axes.

The size of prisms determines properties of cells, such as *local storage*, *communication bandwidth*, and *cell pipelining*. Consequently, tradeoffs are performed that allow selecting a prism size, and therefore cell properties, according to specific requirements for an implementation. The type of tradeoffs possible and the corresponding results were described in Chapter 5. In order to compare the results obtained from this method with those derived with the other techniques discussed in this chapter, we consider grouping by prisms of base size 1 by 1 (i.e., grouping for systolic arrays). Projecting the MMG of transitive closure along the three axes is shown in Figure 7.25.

As is already known, projecting is just a particular case in the MMG method (i.e., prisms of base size 1 by 1). The general case consists of grouping prisms of base size p by q , as discussed in Chapters 4 and 5, which leads to pseudo-systolic arrays.

- **Scheduling** the order of execution of primitive operations that compose a G-node. In the case of prisms of base size 1 by 1, the only schedule possible is determined by the dependencies. For larger base size, the schedule is done by meshes of primitive nodes, as described in the previous chapters.
- **Realizing** the G-graph as an array. For problems with fixed-size data, a G-graph is directly realized as an array. Figure 7.26 depicts the arrays for

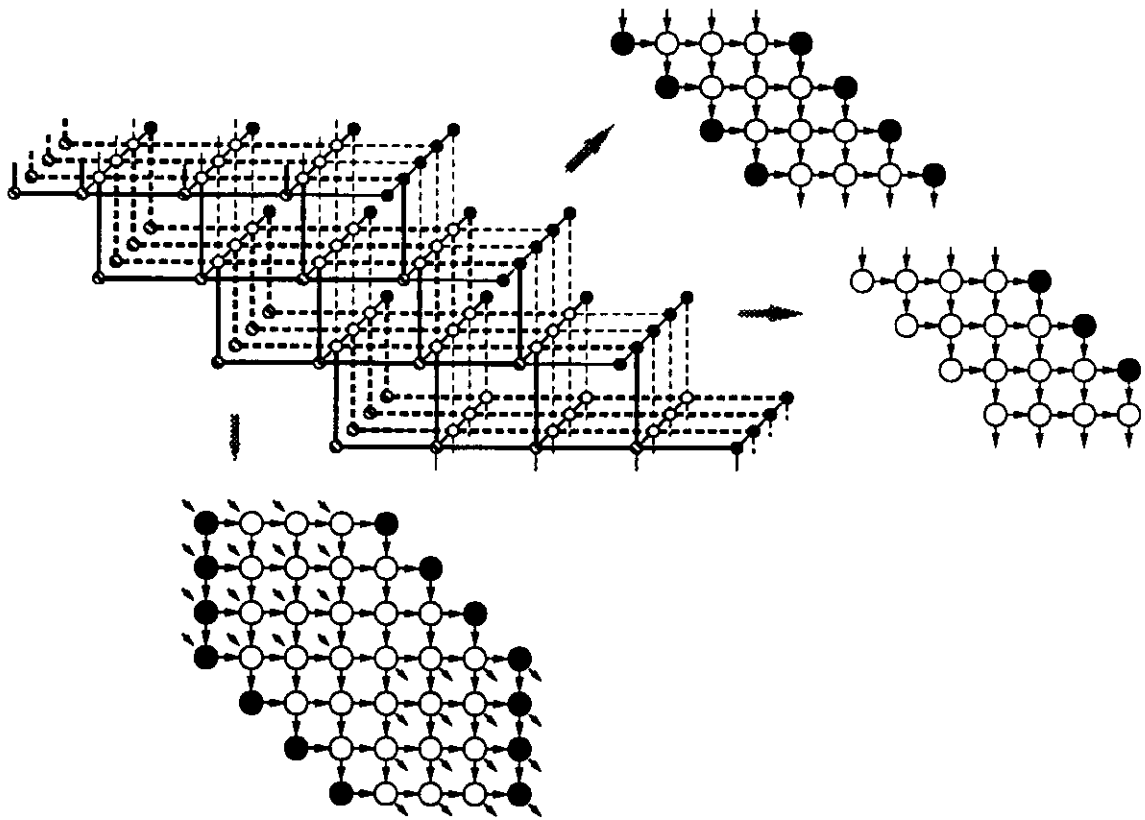


Figure 7.25: Projecting the MMG onto G-graphs

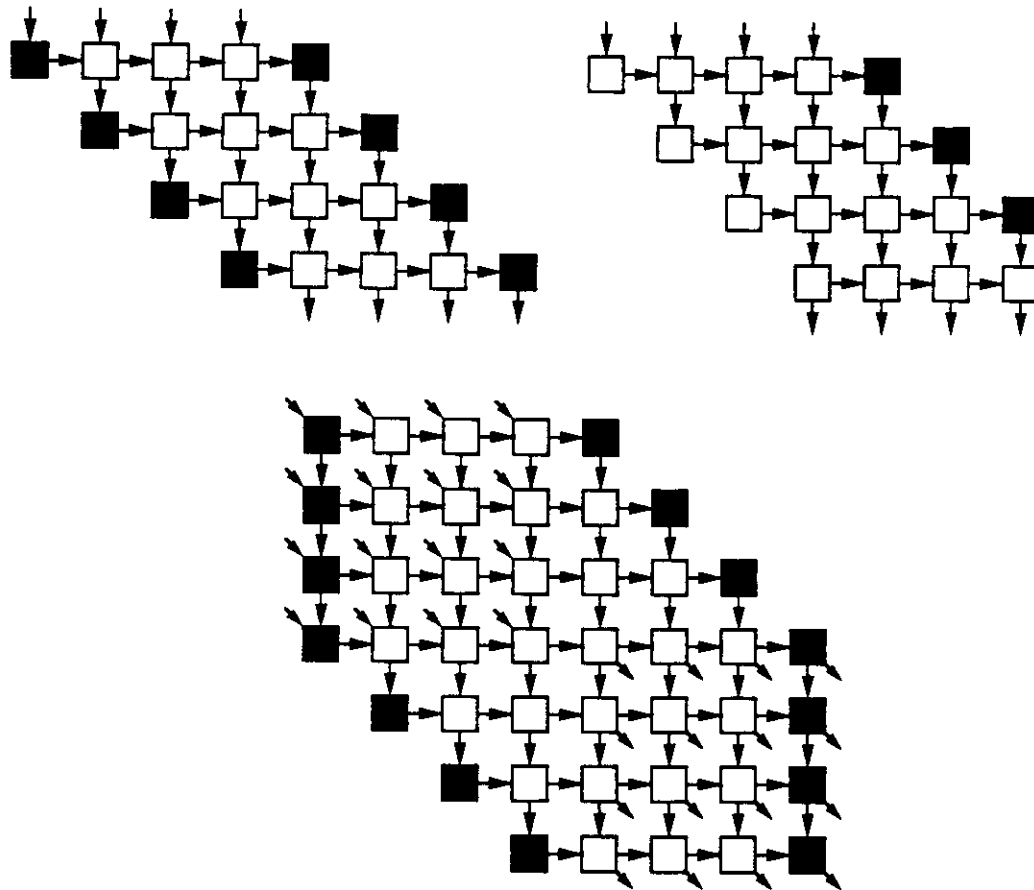


Figure 7.26: Systolic arrays for transitive closure

transitive closure obtained from the G-graphs in Figure 7.25.

For partitioned implementations, the realization is performed by selecting sets of G-nodes (i.e., G-sets) and mapping such G-sets sequentially onto an array (i.e., using cut-and-pile), as described in Chapter 4

The cost and performance of the three systolic arrays derived above can be inferred directly from the MMG. For example, grouping along the Y -axis is not convenient, because paths have different lengths so utilization of cells is not optimal. On the other hand, grouping along axes X or Z collapses paths of the same length, with the associated benefits in performance. Note that the array derived by grouping along the Y -axis has a similar structure to the one obtained with Rao's method.

Table 7.2: Comparison of systolic arrays for transitive closure

Method	Throughput	Ops. per cell	# cells	Links per cell
RIA	$1/n$	3	$\approx 3n^2$	4
SFG	$1/n$	3	n^2	8
MMG	$1/n$	1	$n(n-1)s$ $2n$ delay regs.	4

7.2.4 Comparison of derivation of arrays

The discussion in this section allows us to conclude that the derivation of arrays in our MMG method is systematic, flexible and suited to obtain these structures, taking into account architectural features, implementation constraints and performance and cost measures. In contrast, the SFG and Rao's methods are less capable because they do not consider properties of cells in the transformational process, there is little (if any) support to select certain parameters that are part of the process (i.e., projection direction), and the process is too distant from the implementations.

Given that Rao's approach is specifically oriented towards the design of systolic arrays and Kung's method has been used mainly in that context, an important comparison is how effective the three techniques compared here are in devising such arrays. Table 7.2 summarizes the salient features of the different structures described in this chapter.

7.3 Conclusions regarding the comparison

In terms of the evaluation criteria for a design method indicated in Table 7.1, we conclude the following characteristics of our data-dependency based method:

- Applicable to matrix algorithms, as defined in Chapter 5.
- Able to incorporate implementation restrictions such as limited storage and limited bandwidth per cell.
- Able to perform tradeoffs between local storage and cell bandwidth.
- Able to use pipelined cells and devise specialized cells.

- Considers performance and cost measures while applying the transformations.
- Exhibits strong capabilities to regularize algorithm and to describe transformations and architecture.
- Suitable for interactive use.

In contrast, Rao's method is applicable only to a class of algorithms (RIAs), it does not incorporate implementations restrictions or features of an architecture, and exhibits fewer capabilities to regularize the algorithm and to describe the transformations. Moreover, although Rao's method seems more suitable for automation, it produces arrays that are less efficient than those derived with our technique.

On the other hand, Kung's method is also less effective than our approach because it is less systematic in the process of regularizing an algorithm, and it does not have a precise definition of the regularized form. Moreover, the SFG method has many options (such as selecting the direction of projection, using the graphical or algebraic approach), and few facilities to help in selecting among those options.

We have not discussed the suitability of the three methods to derive arrays for partitioned execution of algorithms. Although partitioning has not been explicitly considered in Rao's technique, his method could use an indirect strategy as described in Chapter 2: first derive a large virtual array, and then partition that array for execution in a smaller one (i.e., applying coalescing *or* cut-and-pile). The same is true for Kung's method, with the only difference that these capabilities have been proposed as part of the technique.

On the other hand, our method is suitable to devise partitioned implementations using a direct strategy and a *combination* of coalescing and cut-and-pile. In Chapter 2, we stated that the direct strategy is more advantageous than the indirect one. Consequently, our method can exploit algorithm properties that are suitable for partitioned execution, which is not the case with the other methods discussed here. In addition, we have not discussed the possibilities of performing tradeoffs between linear and two-dimensional arrays for partitioned implementations, an additional feature of our method.

Moreover, the MMG method can also be used to perform mapping onto class-specific structures, which has not been discussed in this chapter but was illustrated in Chapter 6.

Table 7.3: Summary of evaluation criteria in methods compared

Regularization stage

Class of admissible algorithms	
RIA	only RIAs
SFG	single assignment algorithm
MMG	canonical form of matrix algorithms
Procedure to regularize algorithm	
RIA	no systematic technique
SFG	no systematic technique; ad-hoc transformations
MMG	systematic, transformational
Effectiveness of regularized form	
RIA	regular algorithm representation; adds computing load
SFG	depends on ad-hoc transformations
MMG	effective

All the aspects discussed above are summarized in Tables 7.3 and 7.4.

Consequently, we conclude that our data-dependency based approach meets the evaluation criteria for a design method. In contrast, Rao's method and Kung's technique fail for several of those criteria. The advantages are even more significant when one considers that the type of transformations used in our approach are suitable for a CAD tool, because the graph representations are amenable for the user-interface and the procedures implemented within such a tool.

Table 7.4: Summary of evaluation criteria in methods compared

Derivation of arrays

Architecture	
RIA	no need to describe architecture; systolic array
SFG	some concern for architecture
MMG	considers properties of architecture
Transformational process	
RIA	single-step transformation
SFG	rather ad-hoc transformational process
MMG	systematic transformational process
Restrictions and cell attributes	
RIA	systolic cells
SFG	predefined cells
MMG	memory per cell, bandwidth, tradeoffs
Partitioning	
RIA	indirect partitioning
SFG	indirect partitioning; coalescing and cut-and-pile (LSGP, LPGS)
MMG	direct partitioning; combination of coalescing and cut-and-pile
Arrays produced	
RIA	algorithm-specific only
SFG	algorithm-specific and class-specific
MMG	algorithm-specific and class-specific
Ease of use and automation	
RIA	hard to use manually, amenable for automation
SFG	too many options, amenable for interactive use
MMG	simple, graphical, amenable for interactive use

CHAPTER 8

Summary and further research

This dissertation has addressed the systematic realization of matrix computations on mesh arrays, which are two-dimensional structures with nearest-neighbor connected cells (i.e., systolic-type structures, although the architectures considered are more general than the systolic model.) The research described here deals with the design of *algorithm-specific* arrays, as well as mapping algorithms onto *class-specific* arrays.

We first introduced an extension to the concept of systolic cell to include a small local memory. The new type of cell operates in such a way that cell bandwidth is a fraction of computation rate. This is an attractive property for VLSI/WSI implementation, especially for cells that have a pipelined operation unit. We called this a *pseudo-systolic cell*. We then analyzed aspects of the design and implementation of arrays for matrix algorithms, such as tradeoffs in throughput, tradeoffs in cell storage and cell bandwidth, range of application of arrays, and partitioning approaches. A classification of design issues was proposed which distinguishes among *restrictions* for a particular implementation (i.e., aspects that are fixed before a design starts), *controllable* and *uncontrolled parameters*.

Since designing arrays requires suitable techniques and tools, several methods for the design of systolic arrays were reviewed. A framework was proposed to compare design approaches. This framework identifies two stages in the application of any technique: *algorithm regularization* and *derivation of arrays*. Criteria to evaluate the suitability of methods under this framework were indicated; such criteria also constitute a set of guidelines for the development of a powerful design technique. From the review of existing methods, we concluded that the previously proposed approaches are not general enough to accommodate a large variety of matrix algorithms, that they are difficult to use, and that they are not able to take into account varying requirements or to incorporate flexible optimization criteria as part of the design. Moreover, most methods are oriented towards the design of arrays for fixed-size matrices and are only indirectly applicable to the case of large matrices.

To overcome the problems mentioned above, we proposed a data-dependency graph-based design method that has been the main topic of this dissertation. This method is a general design technique that follows a transformational approach. We stated a canonical representation of matrix algorithms: a recursive definition in terms of a loop statement, and a loop body composed of matrix/vector operators and matrix algorithms. We used such a description to derive a *fully-parallel data-dependency graph* (FPG) by symbolic execution of the description. A regularization process then transforms the FPG into a *multi-mesh dependency graph* (MMG), which consists of a three-dimensional graph with unidirectional nearest-neighbor dependencies. It was shown that this regularization process is always possible, due to the characteristics of the operations that compose a matrix algorithm. In a second stage of the method, arrays are derived from the MMG by collapsing the three-dimensional graph onto a two-dimensional one (a *G-graph*), and realizing this G-graph as an algorithm-specific array or mapping it onto a class-specific array. The second stage allows the incorporation of implementation restrictions and the evaluation of tradeoffs in properties of cells. Moreover, this stage allows deriving arrays for fixed-size data problems and partitioned problems, as well as mapping onto class-specific arrays, while performing optimization of specific performance and cost measures.

The method developed is applicable to pseudo-systolic cells. Since systolic cells are a particular case of pseudo-systolic ones, the method is applicable to them as well. Moreover, our approach is such that local memory in cells is organized as two FIFO buffers, so that local address generation is not an issue. Tradeoffs between memory size and cell bandwidth are possible. Moreover, efficient use of pipelined cells is allowed.

We have applied the method to a variety of algorithms, including matrix multiplication, convolution, LU-decomposition, triangularization by Givens' rotations, Cholesky decomposition, transitive closure, the Faddeev algorithm, and computation of BA^{-1} ; some of these are given as appendices. Through these examples, we determined that the proposed method is easy to apply, incorporates implementation requirements, and optimizes selected performance measures.

The transformations that compose the technique derived in this dissertation were formalized. A canonical form to represent matrix algorithms was proposed, and the equivalence of graphs derived through the transformations was proved. This formalization demonstrated how the FPG of a matrix algorithm is transformed into an MMG suitable for realization as an array, and conditions for achieving such a representation were determined.

The method proposed in this dissertation was compared with other design techniques within the framework discussed earlier. In particular, we evaluated our data-dependency based approach with respect to methods based on index-dependencies, using the technique for Regular Iterative Algorithms (RIAs) [Rao85] as a representative example of index-dependencies. This comparison also included the Signal Flow Graph method [Kung88c]. We concluded that our method is advantageous, because it meets the evaluation criteria and produces arrays that are more efficient than those obtained with other approaches.

The method devised here also allows comparisons between linear and two-dimensional arrays. We showed that, for partitioned execution of matrix algorithms, linear arrays are advantageous over two-dimensional arrays with the same number of cells, for while they have the same I/O bandwidth, linear structures potentially provide a higher throughput and are more suitable for including fault-tolerance features.

Based on the properties of matrix algorithms that are reflected in the MMG, a *linear canonical array* for partitioned execution of matrix algorithms has been proposed in this dissertation. Such an array achieves high utilization, uses pipelined cells, and has an off-cells communication rate lower than the computation rate. Cells of the proposed architecture consist of a pipelined functional unit, internal storage in the form of FIFO buffers, and queues attached to ports. The array is a linear set of cells with support for external I/O and memory modules attached to those cells. An example of such a canonical structure, where cells have a multiplier and an ALU, is described in [More89a]. Estimates of performance of that array have indicated utilization of about 90% for problems of size 200 by 200 in an array with 10 cells.

Mapping problems onto linear local-access arrays was also considered. We discussed issues arising from the algorithms, the architecture and the mapping process. The capabilities of the method in this context were illustrated using a hypothetical memory-linked architecture, where cells are connected by memory modules of large capacity so that partitioning by coalescing is the suitable mapping technique. A heuristic approach to achieve load balancing during partitioning by coalescing was devised; this produced a high estimated utilization (on the order of 90%) while solving problems of size 200 by 200 in an array with 10 cells. These results are significantly better than those obtained in other arrays that have comparable characteristics [Tsen88].

Further research in this area may be divided into the following fields:

1. Development of a CAD tool that implements the method proposed here. Most steps in this technique are suitable for complete automation, and the remaining ones are suitable for an interactive environment. Moreover, the visual (i.e., graphical) properties of the method make it very attractive for a CAD tool that effectively uses the currently available graphics capabilities of computing workstations.

Some work in this area is already under development [Erce88]. Moreover, this task may benefit from experience accumulated by others in the implementation of graphics-oriented design tools [Omtz88, Kung88d, Kung89].

2. Application of the method to other algorithms and class-specific architectures. Some experience has already been gained in the latter case, because our technique has been used to study mapping algorithms such as QR-decomposition onto the QUENTM processor [Niel88, More89b], with estimated utilization higher than 60%. This value is about twice that obtained earlier on the same machine. Further work includes using other machines as target implementations, as well as other algorithms for class-specific mappings and algorithm-specific realizations.
3. Study of the suitability of the regularized representation as a tool for mapping algorithms onto general-purpose parallel computers, such as multiprocessors, hypercubes, or shared-memory machines. The description of a matrix algorithm through the MMG makes it possible to study in detail the characteristics of computational load, load balancing, and communication load imposed by the algorithm. The type of analysis feasible is similar to that used in this dissertation to map matrix algorithms onto a hypothetical memory-linked array. Results from such an analysis are very promising and indicate that it is possible to obtain higher performance than that usually achieved in implementations of matrix algorithms. This benefit arises from the structured and static form of dependencies in matrix algorithms (the MMG), which allows properties of the computation to be taken into account at compile time.

Bibliography

- [Ager82] T. Agerwala and E. Arvind, editors, *Special Issue on Data Flow Systems*, IEEE Computer (February 1982).
- [Ahme82] H. Ahmed, J. Delosme, and M. Morf, Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing, *IEEE Computer*, 15(1):65–82 (January 1982).
- [Andr76] H.C. Andrews and C.L. Patterson, Singular Value Decomposition and Digital Image Processing, *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-24(1):26–53 (February 1976).
- [Anna87] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb, The Warp Computer: Architecture, Implementation and Performance, *IEEE Transactions on Computers*, C-36(12):1523–1538 (December 1987).
- [Anne88] J. Annevelink and P. Dewilde, HIFI: A Functional Design System for VLSI Processing Arrays, pp. 413–452, in *International Conference on Systolic Arrays* (May 1988).
- [Atha88] A. Athavale and J. JáJá, Compiling Programs for Systolic Arrays, pp. 509–519, in *VLSI Signal Processing, III* (November 1988).
- [Avil83] J.H. Avila and P.J. Kuekes, A One Gigaflop VLSI Systolic Processor, pp. 159–165, in *SPIE Real-Time Signal Processing VI* (August 1983).
- [Barn83] T.P. Barnwell and D.A. Schwartz, Optimal Implementation of Flow Graphs on Synchronous Multiprocessors, pp. 188–193, in *Asilomar Conference on Circuits and Systems* (November 1983).
- [Blac81] J. Blackmer, G. Frank, and P. Kuekes, A 200 Million Operations per Second (MOPS) Systolic Processor, pp. 10–18, in *SPIE Real-Time Signal Processing IV* (August 1981).
- [Boja84] A. Bojanczyk, R.P. Brent, and H.T. Kung, Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-Connected Processors, *SIAM Journal on Scientific and Statistical Computing*, 5(1):95–104 (March 1984).

- [Boja86] A.W. Bojanczyk, R.P. Brent, and F.R. de Hoog, Parallel QR decomposition of Toeplitz Matrices, pp. 39–44, in *SPIE Advanced Algorithms and Architectures for Signal Processing* (August 1986).
- [Bouk72] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick, The ILLIAC IV System, *Proceedings of the IEEE*, 60(4):369–388 (April 1972).
- [Bren85a] R.P. Brent and F.T. Luk, A Systolic Array for the Linear-Time Solution of Toeplitz Systems of Equations, *Journal of VLSI and Computer Systems*, 1(1):1–22 (1985).
- [Bren85b] R.P. Brent, F.T. Luk, and C. van Loan, Computation of the Singular Value Decomposition Using Mesh-Connected Processors, *Journal of VLSI and Computer Systems*, 1(3):242–270 (1985).
- [Brom81] K. Bromley and H.J. Whitehouse, Signal Processing Technology Overview, pp. 102–106, in *SPIE Real-Time Signal Processing IV* (August 1981).
- [Brom88] K. Bromley, S.Y. Kung, and E. Swartzlander, editors, *International Conference on Systolic Arrays*, IEEE Computer Society Press (May 1988).
- [Capp83] P.R. Cappello and K. Steiglitz, Unifying VLSI Array Designs with Geometric Transformations, pp. 448–457, in *International Conference on Parallel Processing* (August 1983).
- [Capp84] P.R. Cappello and K. Steiglitz, Unifying VLSI Array Design with Linear Transformations of Space-Time, in *Advances in Computing Research*, vol. 2, pp. 23–65, JAI Press Inc. (1984).
- [Cava87] J.R. Cavallaro and F.T. Luk, CORDIC Arithmetic for an SVD processor, pp. 215–222, in *8th Symposium on Computer Arithmetic* (May 1987).
- [Chap85] R. Chapman, T.S. Durrani, and T. Willey, Design Strategies for Implementing Systolic and Wavefront Arrays using OCCAM, pp. 292–295, in *International Conference on Acoustics, Speech and Signal Processing* (March 1985).
- [Chen86] M. Chen, Synthesizing VLSI Architectures: Dynamic Programming Solver, pp. 776–784, in *International Conference on Parallel Processing* (August 1986).

- [Chou88] S.I. Chou and C.M. Rader, Algorithm-based Error Detection of a Cholesky Factor Updating Systolic Array Using CORDIC Processors, pp. 104–111, in *SPIE Real-Time Signal Processing XI* (August 1988).
- [Chua84] H.Y.H. Chuang and G. He, Design of Problem-Size Independent Systolic Array Systems, pp. 152–157, in *International Conference on Computer Design* (October 1984).
- [Chua85] H.Y.H. Chuang and G. He, A Versatile Systolic Array for Matrix Computations, pp. 315–322, in *12th Annual Symposium on Computer Architecture* (June 1985).
- [Como87] P. Comon and Y. Robert, A Systolic Array for Computing BA^{-1} , *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-35(6):717–723 (June 1987).
- [Davi88] G.A. Davidson, P.R. Cappello, and A. Gersho, Systolic Architectures for Vector Quantization, *IEEE Transactions on Acoustics, Speech and Signal Processing*, 36(10):1651–1664 (October 1988).
- [Delo86] J.M. Delosme and I.C.F. Ipsen, Design Methodology for Systolic Arrays, pp. 245–259, in *SPIE Advanced Algorithms and Architectures for Signal Processing* (August 1986).
- [Dong87a] J.J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*, Technical Memorandum 23, Mathematics and Computer Science Division, Argonne National Laboratory (October 1987).
- [Dong87b] J.J. Dongarra, J.L. Martin, and J. Worlton, Computer Benchmarking: Paths and Pitfalls, *IEEE Spectrum*, 38–43 (July 1987).
- [Dong88a] V. van Dongen, PRESAGE, a Tool for the Design of Low-Cost Systolic Circuits, pp. 2765–2768, in *International Symposium on Circuits and Systems* (1988).
- [Dong88b] V. van Dongen and P. Quinton, Uniformization of Linear Recurrence Equations: A Step Towards the Automatic Synthesis of Systolic Arrays, pp. 473–482, in *International Conference on Systolic Arrays* (May 1988).
- [Drak87] B.L. Drake, F.T. Luk, J.M. Speiser, and J.J. Symanski, SLAPP: A Systolic Linear Algebra Parallel Processor, *IEEE Computer*, 20(7):45–50 (July 1987).

- [Engs87] B.R. Engstrom and P.R. Cappello, *The SDEF Systolic Programming System*, Technical Report TRCS87-15, Department of Computer Science, University of California Santa Barbara (August 1987).
- [Erce87a] M. Ercegovac and T. Lang, On-line Scheme for Computing Rotation Factors, pp. 196–203, in *8th Symposium on Computer Arithmetic* (May 1987).
- [Erce87b] M. Ercegovac and T. Lang, *Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD*, Technical Report CSD-870046, Computer Science Department, University of California Los Angeles (1987).
- [Erce88] M.D. Ercegovac and T. Lang, *Graph-based method for the design of arrays for matrix computations*, Proposal for Research, Computer Science Department, University of California Los Angeles (November 1988).
- [Fadd63] D.K. Faddeev and V.N. Faddeeva, *Computational Methods of Linear Algebra*, pp. 150–158, W.H. Freeman and Co. (1963).
- [Fish83] A.L. Fisher, H.T. Kung, and L.M. Monier, Architecture of the PSC: A Programmable Systolic Chip, pp. 48–53, in *10th Annual Symposium on Computer Architecture* (1983).
- [Fort85] J.A.B. Fortes and D.I. Moldovan, Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms, *Journal of Parallel and Distributed Computing*, 2:277–301 (1985).
- [Fort87a] J.A.B. Fortes and B.W. Wah, editors, *Special Issue on Systolic Arrays*, IEEE Computer Society (July 1987).
- [Fort87b] J.A.B. Fortes and B.W. Wah, Systolic Arrays - From Concept to Implementation, *IEEE Computer*, 20(7):12–17 (July 1987).
- [Fort88] J.A.B. Fortes, K. Fu, and B.W. Wah, Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays, in Veljko Mirlutinović, editor, *Computer Architecture*, pp. 454–494, North-Holland (1988).
- [Foul87] D.E. Foulser and R. Schreiber, The Saxpy Matrix-1: A General Purpose Systolic Computer, *IEEE Computer*, 20(7):35–44 (July 1987).
- [Fox88] G. Fox, editor, *Third Conference on Hypercube Concurrent Computers and Applications. Banded and Full Matrix Algorithms*, ACM Press (January 1988).

- [Fris86] P. Frison, P. Gachet, and P. Quinton, Designing Systolic Arrays with DIASTOL, pp. 93-105, in *VLSI Signal Processing, II* (November 1986).
- [FXS87] *FX/Series Product Summary*, Alliant Computer Systems Corporation (June 1987).
- [Gann82] D. Gannon, Pipelining Array Computations for MIMD Parallelism: A Functional Specification, pp. 284-286, in *International Conference on Parallel Processing* (August 1982).
- [Gent81] W.M. Gentleman and H.T. Kung, Matrix Triangularization by Systolic Arrays, pp. 19-26, in *SPIE Real-Time Signal Processing IV* (August 1981).
- [Golu85] G.H. Golub and C.F. van Loan, *Matrix Computations*, The John Hopkins University Press (1985).
- [Groo87] A.J. De Groot, E.M. Johansson, and S.R. Parker, Systolic Array for Efficient Execution of the Faddeev Algorithm, pp. 86-93, in *SPIE Real-Time Signal Processing X* (August 1987).
- [Guer86] C. Guerra and R. Melhem, Synthesizing Non-Uniform Systolic Designs, pp. 765-771, in *International Conference on Parallel Processing* (August 1986).
- [Hayn82] L.S. Haynes, R.L. Lau, D.P. Siewiorek, and D.W. Mizell, A Survey of Highly Parallel Computing, *IEEE Computer*, 15(1):9-24 (January 1982).
- [Hein87] C.E. Hein, R.M. Zieger, and J.A. Urbano, The Design of a GaAs Systolic Array for an Adaptive Null Steering Beamforming Controller, *IEEE Computer*, 20(7):92-93 (July 1987).
- [Hell83] D.E. Heller and I.C.F. Ipsen, Systolic Networks for Orthogonal Decompositions, *SIAM Journal on Scientific and Statistical Computing*, 4(2): (June 1983).
- [Hell84] D. Heller, Partitioning Big Matrices for Small Systolic Arrays, in S.Y. Kung, H.J. Whitehouse, and T. Kailath, editors, *Concurrent Array Processors*, pp. 185-199, Prentice Hall (1984).
- [Hill85] W.D. Hillis, *The Connection Machine*, MIT Press (1985).
- [Hwan82] K. Hwang and Y.H. Cheng, Partitioned Matrix Algorithms for VLSI Arithmetic Systems, *IEEE Transactions on Computers*, C-31(12):1215-1224 (December 1982).

- [Ibar87] O.H. Ibarra and M.A. Palis, VLSI Algorithms for Solving Recurrence Equations and Applications, *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-35(7):1046–1063 (July 1987).
- [Jain87] K. Jainandunsing, H. Nelis, and E.F. Deprettere, Systematic Design of Fixed Size Systolic Arrays, Applied to the Orthogonal Faddeev Equations Solver, pp. 471–477, in *European Conference on Circuit Theory and Design (ECCTD)* (September 1987).
- [John84] L. Johnsson, Highly Concurrent Algorithms for Solving Linear Systems of Equations, pp. 105–126, in *Conference on Elliptic Problem Solvers II* (1984).
- [Jove84] J.M. Jover and T. Kailath, Design Framework for Systolic-Type Arrays, pp. 8.5.1–8.5.4, in *International Conference on Acoustics, Speech and Signal Processing* (March 1984).
- [Kand88] D.A. Kandle, A Systolic Signal Processor for Signal-Processing Applications, *IEEE Computer*, 20(7):94–95 (July 1988).
- [Karp67] R.M. Karp, R.E. Miller, and S. Winograd, The Organization of Computations for Uniform Recurrence Equations, *Journal of the Association for Computing Machinery*, 14(3):563–590 (July 1967).
- [Klem80] V.C. Klema and A.J. Laub, The Singular Value Decomposition: Its Computation and Some Applications, *IEEE Transactions on Automatic Control*, AC-25(2):164–176 (April 1980).
- [Ko88] C.K. Ko and O. Wing, Mapping Strategy for Automatic Design of Systolic Arrays, pp. 285–294, in *International Conference on Systolic Arrays* (May 1988).
- [Kogg81] P. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill (1981).
- [Kore83] I. Koren and G.M. Silberman, A Direct Mapping of Algorithms onto VLSI Processing Arrays Based on the Data Flow Approach, pp. 335–337, in *International Conference on Parallel Processing* (August 1983).
- [Kuck81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, Dependence Graphs and Compiler Optimizations, pp. 207–218, in *ACM* (1981).
- [Kung78] H.T. Kung and C.E. Leiserson, Systolic Arrays (for VLSI), in *Symposium on Sparse Matrix Computations and their Applications* (November 1978).

- [Kung79] H.T. Kung, Let's Design Algorithms for VLSI Systems, pp. 65–90, in *CALTECH Conference on VLSI* (January 1979).
- [Kung82] H.T. Kung, Why Systolic Architectures?, *IEEE Computer*, 15(1):37–46 (January 1982).
- [Kung83a] H.T. Kung and W.T. Lin, An Algebra for Systolic Computation, pp. 141–160, in *Conference on Elliptic Problem Solvers* (1983).
- [Kung83b] S.Y. Kung and Y.H. Hu, A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems, *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-31(1):66–75 (February 1983).
- [Kung87a] S.Y. Kung, P.S. Lewis, and S.N. Jean, Canonic and Generalized Mapping from Algorithms to Arrays - A Graph Based Methodology, pp. 124–133, in *20th Annual Hawaii International Conference on System Sciences* (1987).
- [Kung87b] S.Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang, Wavefront Array Processors - Concept to Implementation, *IEEE Computer*, 20(7):18–33 (July 1987).
- [Kung87c] S.Y. Kung, S.C. Lo, and P.S. Lewis, Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems, *IEEE Transactions on Computers*, C-36(5):603–614 (May 1987).
- [Kung88a] H.T. Kung, presentation, in *SPIE Real-Time Signal Processing XI* (August 1988).
- [Kung88b] H.T. Kung, Warp Experience: We Can Map Computations onto a Parallel Computer Efficiently, pp. 668–675, in *International Conference on Supercomputing* (July 1988).
- [Kung88c] S.Y. Kung, *VLSI Array Processors*, Prentice Hall (1988).
- [Kung88d] S.Y. Kung and S.N. Jean, A VLSI Array Compiler System (VACS) for Array Design, pp. 495–508, in *VLSI Signal Processing, III* (November 1988).
- [Kung89] S.Y. Kung and J.S.N. Jean, Array Compiler Design for VLSI/WSI Systems, in *International Conference on Systolic Arrays* (May 1989).
- [Lack88] R.J. Lackey, H.F. Baurle, and J. Barile, Application-specific Super Computer, pp. 187–195, in *SPIE Real-Time Signal Processing XI* (August 1988).

- [Lam85] M.S. Lam and J. Mostow, A Transformational Model of VLSI Systolic Design, *IEEE Computer*, 42-52 (February 1985).
- [Lee88] P. Lee and Z.M. Kedem, Synthesizing Linear Array Algorithms from Nested For Loop Algorithms, *IEEE Transactions on Computers*, 37(12):1578-1598 (December 1988).
- [Lev-88] H. Lev-Ari and B. Friedlander, On the Systematic Design of Fault-Tolerant Processor Arrays with Application to Digital Filtering, pp. 483-494, in *VLSI Signal Processing, III* (November 1988).
- [Lewi88] P.S. Lewis, Algorithms and Architectures for Multichannel Enhancement of Magnetoencephalographic Signals, pp. 741-745, in *Asilomar Conference on Signals, Systems and Computations* (1988).
- [Li84] G.J. Li and B.W. Wah, The Design of Optimal Systolic Arrays, *IEEE Transactions on Computers*, C-34(1):66-77 (October 1984).
- [Linc88] R.A. Lincoln and K. Yao, Efficient Systolic Kalman Filtering Design by Dependence Graph Mapping, pp. 396-407, in *VLSI Signal Processing, III* (November 1988).
- [Lopr88] D.P. Lopresti, P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences, *IEEE Computer*, 20(7):98-99 (July 1988).
- [Luk86] F.T. Luk, Architectures for Computing Eigenvalues and SVD's, pp. 24-33, in *SPIE Highly Parallel Signal Processing Architectures* (January 1986).
- [Luk87] F.T. Luk, editor, *Advanced Algorithms and Architectures for Real-Time Signal Processing II*, SPIE-The International Society for Optical Engineering (August 1987).
- [Luk88] F.T. Luk, editor, *Advanced Algorithms and Architectures for Real-Time Signal Processing III*, SPIE-The International Society for Optical Engineering (August 1988).
- [McWh83] J.G. McWhirter, Recursive Least-Squares Minimization Using a Systolic Array, pp. 105-112, in *SPIE Real-Time Signal Processing VI* (August 1983).
- [Meie87] U. Meier and A. Sameh, Numerical Linear Algebra on the CEDAR Multiprocessor, pp. 1-9, in *SPIE Advanced Algorithms and Architectures for Signal Processing II* (August 1987).
- [Mira84] W.L. Miranker and A. Winkler, Space-Time Representations of Computational Structures, *Computing*, 32:93-114 (1984).

- [Mold82] D.I. Moldovan, On the Analysis and Synthesis of VLSI Algorithms, *IEEE Transactions on Computers*, C-31(11):1121–1126 (November 1982).
- [Mold83] D.I. Moldovan, On the Design of Algorithms for VLSI Systolic Arrays, *Proceedings of the IEEE*, 71(1):113–120 (January 1983).
- [Mold86] D.I. Moldovan and J.A.B. Fortes, Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays, *IEEE Transactions on Computers*, C-35(1):1–12 (January 1986).
- [Mold87] D.I. Moldovan, ADVIS: A Software Package for the Design of Systolic Arrays, *IEEE Transactions on Computer-Aided Design*, CAD-6(1):33–40 (January 1987).
- [More87] J.H. Moreno and T. Lang, Design of Special-Purpose Arrays for Matrix Computations: Preliminary Results, pp. 53–65, in *SPIE Real-Time Signal Processing X* (August 1987).
- [More88a] J.H. Moreno and T. Lang, Arrays for Partitioned Matrix Algorithms: Tradeoffs Between Cell Storage and Cell Bandwidth, in *SPIE Real-Time Signal Processing XI* (August 1988).
- [More88b] J.H. Moreno and T. Lang, *Designing Arrays for the Faddeev Algorithm*, Technical Report CSD–880013, Computer Science Department, University of California Los Angeles (March 1988).
- [More88c] J.H. Moreno and T. Lang, Graph-based Partitioning of Matrix Algorithms for Systolic Arrays: Application to Transitive Closure, pp. 28–31, in *International Conference on Parallel Processing* (August 1988).
- [More89a] J.H. Moreno and T. Lang, Linear Array for Partitioned Execution of Matrix Algorithms with High Utilization, in *SPIE Real-Time Signal Processing XII* (August 1989).
- [More89b] J.H. Moreno and R. Nielsen, *Mapping the QR-decomposition algorithm onto the QUEN processor*, Technical Report, Interstate Electronics Corporation, 1001 E. Ball Rd, Anaheim, CA 92803 (March 1989).
- [Most84] J. Mostow and B. Balzer, Application of a Transformational Software Development Methodology to VLSI Design, *Journal of Systems and Software*, 4:51–61 (1984).
- [Núñez84] F.J. Núñez and N. Torralba, Transitive Closure Partitioning and its Mapping to a Systolic Array, pp. 564–566, in *International Conference on Parallel Processing* (August 1987).

- [Núñez88] F.J. Núñez and M. Valero, A Block Algorithm for the Algebraic Path Problem and its Execution on a Systolic Array, pp. 265–274, in *International Conference on Systolic Arrays* (May 1988).
- [Nash84] J.G. Nash and S. Hansen, Modified Faddeev Algorithm for Matrix Manipulation, pp. 39–46, in *SPIE Real-Time Signal Processing VII* (August 1984).
- [Nash86a] J.G. Nash, S. Hansen, and K.W. Przytula, Systolic Partitioned and Banded Linear Algebraic Computations, pp. 10–16, in *SPIE Real-Time Signal Processing IX* (August 1986).
- [Nash86b] J.G. Nash, K.W. Przytula, and S. Hansen, Systolic/Cellular Processor for Linear Algebraic Operations, pp. 306–315, in *VLSI Signal Processing, II* (1986).
- [Nash88] J.G. Nash, K.W. Przytula, and S. Hansen, The Systolic/Cellular System for Signal Processing, *IEEE Computer*, 20(7):96–97 (July 1988).
- [Nava86a] J.J. Navarro, J.M. Llaberia, and M. Valero, Computing Size-Independent Matrix Problems on Systolic Array Processors, pp. 271–278, in *19th Annual Symposium on Computer Architecture* (June 1986).
- [Nava86b] J.J. Navarro, J.M. Llaberia, and M. Valero, Solving Matrix Problems with No Size Restriction on a Systolic Array Processor, pp. 676–683, in *International Conference on Parallel Processing* (August 1986).
- [Nava87] J.J. Navarro, J.M. Llaberia, and M. Valero, Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors, *IEEE Computer*, 20(7):77–89 (July 1987).
- [Niel88] R. Nielsen, High-performance Sonar System, *Defense Science* (August 1988).
- [Omtz88] E.T.L. Omtzigt, SYSTARS: A CAD Tool for the Synthesis and Analysis of VLSI Systolic/Wavefront Arrays, pp. 383–389, in *International Conference on Systolic Arrays* (May 1988).
- [Przy88] K.W. Przytula and J.G. Nash, A Special Purpose Coprocessor for Signal Processing, pp. 736–740, in *Asilomar Conference on Signals, Systems and Computations* (1988).
- [Quin84] P. Quinton, Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations, pp. 208–214, in *11th Annual Symposium on Computer Architecture* (June 1984).

- [Rajo86] S.V. Rajopadhye, S. Purushothaman, and R.M. Fujimoto, On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies, in *6th Conference on Foundations of Software Technology and Theoretical Computer Science* (December 1986).
- [Rajo88a] S.V. Rajopadhye, I/O Behavior of Systolic Arrays, pp. 423–434, in *VLSI Signal Processing, III* (November 1988).
- [Rajo88b] S.V. Rajopadhye, Systolic Arrays for LU decomposition, pp. 2513–2516, in *International Symposium on Circuits and Systems* (1988).
- [Rama83] I.V. Ramakrishnan, D.A. Fussell, and A. Silberschatz, On Mapping Homogeneous Graphs on A Linear Array-Processor Model, pp. 440–447, in *International Conference on Parallel Processing* (August 1983).
- [Rao85] S.K. Rao, *Regular Iterative Algorithms and their Implementation on Processor Arrays*, PhD dissertation, Information Systems Laboratory, Stanford University, Stanford, California (October 1985).
- [Rao86] S.K. Rao and T. Kailath, What is a Systolic Algorithm, pp. 34–48, in *SPIE Highly Parallel Signal Processing Architectures* (January 1986).
- [Rao88] S.K. Rao and T. Kailath, Regular Iterative Algorithms and their Implementation on Processor Arrays, *Proceedings of the IEEE*, 76(3):259–269 (March 1988).
- [Robe86] I. Robert, *Algorithmes et Architectures Systoliques*, Institut National Polytechnique de Grenoble (1986).
- [Royc88a] V.P. Roychowdhury and T. Kailath, Regular Processor Arrays for Matrix Algorithms with Pivoting, pp. 237–246, in *International Conference on Systolic Arrays* (May 1988).
- [Royc88b] V.P. Roychowdhury, S.K. Rao, L. Thiele, and T. Kailath, On the Localization of Algorithms for VLSI Processor Arrays, pp. 459–470, in *VLSI Signal Processing, III* (November 1988).
- [Same85a] A. Sameh, Algorithms and Experiments for Parallel Linear Systems Solvers, in *2nd SIAM Conference on Parallel Processing for Scientific Computing* (November 1985).
- [Same85b] A. Sameh, Parallel Linear Systems Solvers, in *Conference on Vector and Parallel Processors for Scientific Computation* (May 1985).
- [Schi86] D.E. Schimmel and F.T. Luk, A Practical Real Time SVD Machine with Multi-level Fault Tolerance, pp. 142–148, in *SPIE Real-Time Signal Processing IX* (August 1986).

- [Schr82] R. Schreiber, Systolic Arrays for Eigenvalue Computation, pp. 27–34, in *SPIE Real-Time Signal Processing V* (August 1982).
- [Schw84] D.A. Schwartz and T.P. Barnwell, A Graph Theoretic Technique for the Generation of Systolic Implementations for Shift-Invariant Flow Graphs, pp. 8.3.1–8.3.4, in *International Conference on Acoustics, Speech and Signal Processing* (March 1984).
- [SIAM87] SIAM, editor, *Third SIAM Conference on Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics (December 1987).
- [Snyd82] L. Snyder, Introduction to the Configurable Highly Parallel Machine, *IEEE Computer*, 15(1):47–64 (January 1982).
- [Snyd84] L. Snyder, Parallel Programming and the Poker Programming Environment, *IEEE Computer*, 17(7):27–36 (July 1984).
- [Snyd86] L. Snyder, Programming Environments for Systolic Arrays, pp. 134–144, in *SPIE Highly Parallel Signal Processing Architectures* (January 1986).
- [Spei81] J.M. Speiser and H. Whitehouse, Parallel Processing Algorithms and Architectures for Real-Time Signal Processing, pp. 2–9, in *SPIE Real-Time Signal Processing IV* (August 1981).
- [Spei83] J.M. Speiser and H. Whitehouse, A Review of Signal Processing with Systolic Arrays, pp. 2–6, in *SPIE Real-Time Signal Processing VI* (August 1983).
- [Spei88] J.M. Speiser, An Overview of Matrix-Based Signal Processing, pp. 284–289, in *Asilomar Conference on Signals, Systems and Computations* (1988).
- [Syma83] J.J. Symanski, Implementation of Matrix Operations on the Two-Dimensional Systolic Array Testbed, pp. 136–142, in *SPIE Real-Time Signal Processing VI* (August 1983).
- [Syma86] J.J. Symanski, Architecture of the Systolic Linear Algebra Parallel Processor (SLAPP), pp. 17–21, in *SPIE Real-Time Signal Processing IX* (August 1986).
- [Syma88] J.J. Symanski and K. Bromley, Video Analysis Transputer Array (VATA) Processor, in *SPIE Real-Time Signal Processing XI* (August 1988).

- [Torr88] N. Torralba and J.J. Navarro, A One Dimensional Systolic Array for Solving Arbitrarily Large Least Mean Square Problems, pp. 103–112, in *International Conference on Systolic Arrays* (May 1988).
- [Torr89] N. Torralba and J.J. Navarro, Size-independent Systolic Algorithms for QR Iteration and Hessenberg Reduction, pp. 166–175, in *International Conference on Systolic Arrays* (May 1989).
- [Tsen88] P.S. Tseng, M. Lam, and H.T. Kung, The Domain Parallel Computation Model on Warp, pp. 130–137, in *SPIE Real-Time Signal Processing XI* (August 1988).
- [Weis81] V. Weiser and A. Davis, A Wavefront Notation Tool for VLSI Array Design, in H.T. Kung et al., editor, *VLSI Systems and Computations*, pp. 226–234, Computer Science Press (October 1981).
- [Yaac88a] Y. Yaacoby and P. Cappello, *Converting Affine Recurrence Equations to Quasi-Uniform Recurrence Equations*, Technical Report TRCS87-18, Department of Computer Science, University of California Santa Barbara (February 1988).
- [Yaac88b] Y. Yaacoby and P.R. Cappello, Scheduling a System of Affine Recurrence Equations onto a Systolic Array, pp. 373–382, in *International Conference on Systolic Arrays* (May 1988).

APPENDIX A

Arrays for the Faddeev algorithm

A.1 Introduction

The traditional approach to offer flexibility in an array (so that different algorithms may be computed in the same structure) consists of providing programmable features that are activated via software. An alternative is using an algorithm for a class of problems, such as the Faddeev algorithm [Fadd63], which has the capability of performing a variety of matrix computations without the need for programmable features in the array. Some overhead or cost is involved of course, which consists of performing additional operations. Several arrays to compute the Faddeev algorithm have been discussed in the literature [Nash84, Nash86a, Nash86b, Chua85]. Nash and Hansen [Nash84] proposed a trapezoidal array for problems with fixed-size data and an implementation of their scheme was presented in [Nash86b]. The same structure is used in [Nash86a] to partition the algorithm. The Faddeev algorithm is also implemented in [Chua85] for both fixed-size and variable-size problems, and in [Groo87] for partitioned implementation in a two-dimensional array of transputers.

In this chapter, we apply our design method to the Faddeev algorithm. We discuss the design of arrays for fixed-size problems as well as partitioning the algorithm, and evaluate the structures obtained. We show that for matrices of size n by n it is possible to achieve throughput $[n]^{-1}$ or $[2n]^{-1}$ in two-dimensional arrays with $O(n^2)$ cells. The utilization of these arrays tends to $7/9$.¹ One of the two-dimensional schemes derived here corresponds to that proposed by Nash and Hansen in [Nash84], though their paper did not include an evaluation of the array in terms of throughput and utilization as it is possible with our graph-based technique.

In partitioned mode, we show that for large matrices the throughput of linear and two-dimensional arrays tends to $(3K)/(7n^3)$ (where K is the number of PEs) and utilization tends to 1. The two-dimensional partitioned scheme devised here is

¹Linear arrays with $O(n)$ cells and throughput $2/(3n^2 - n + 2)$ are derived in [More88b].

more efficient than the one proposed in [Chua85]. In addition, it does not need the complex loading and un-loading of data required in [Nash86a]. Moreover, we show that the linear array is simpler, has slightly better throughput and utilization with the same number of units than a square array, and exhibits better characteristics for fault-tolerant implementations than a two-dimensional structure.

A.2 The modified Faddeev algorithm

The matrix version of the Faddeev algorithm evaluates the expression $CX + D$ subject to the condition $AX = B$, where A, B, C, D are given matrices, X is a column vector, and A is of full rank. The algorithm can be expressed by representing the data as the extended matrix

$$\begin{array}{c|c} A & B \\ \hline -C & D \end{array}$$

and performing linear combinations on this extended matrix with the objective of transforming matrix C into a matrix of zeroes. If we represent such linear combinations as W , the operations performed are $(-C + WA)$ and $(D + WB)$. The annulment of C requires that $W = CA^{-1}$, so that $D + WB = D + CA^{-1}B$. Since $X = A^{-1}B$, the final result $D + WB = D + CX$ replaces the values of matrix D in the expression above [Nash84].

Several matrix operations are possible by selecting specific entries for matrices A, B, C and D . Figure A.1 depicts some alternatives, including matrix multiplication/addition, matrix inversion and solution of linear systems of equations. Therefore, the Faddeev algorithm allows a degree of “programmability” by selecting the values of input data. The cost of this flexibility consists of additional operations, because the algorithm operates on four matrices.

In addition to its capability to perform different matrix operations, the Faddeev algorithm has other advantages:

- The algorithm does not need to compute the elements of W , because it must only annul the elements of C . Such annulment is done by ordinary Gaussian elimination.
- When solving linear systems of equations, the algorithm does not need the back-substitution step usually found in triangularization methods. Instead, results are obtained directly at the end of the annulment of C .

$$\begin{array}{ccc}
\frac{A}{-I} \left| \begin{array}{c} I \\ 0 \end{array} \right. & \rightarrow & A^{-1} \\
\frac{I}{-C} \left| \begin{array}{c} B \\ D \end{array} \right. & \rightarrow & D + CB \\
\frac{A}{C} \left| \begin{array}{c} B \\ 0 \end{array} \right. & \rightarrow & CA^{-1}B + D
\end{array}$$

Figure A.1: Matrix operations with the Faddeev algorithm

Since Gaussian elimination does not guarantee numerical stability and fails for zero pivot elements, Nash and Hansen have proposed a modification to the original Faddeev algorithm [Nash84]. This modification consists of “adding an orthogonal factorization capability for added numerical stability and to allow the coefficient matrix to be non-square for over- and under-determined systems of equations”. Nash and Hansen’s scheme uses Givens rotations to annul matrix C . The utilization of Givens rotations requires to divide the process of annulling matrix C into the following two-step procedure [Nash84]:

- Triangularization of matrix A through Givens rotations and application of such rotations to matrix B .
- Gaussian elimination of the elements of C using the rotated matrix A to compute pivots, and application of the same transformations to D using the rotated matrix B .

Figure A.2 shows the dependency graph of the modified Faddeev algorithm for 4 by 4 matrices, after replacing data broadcasting by transmittent data. Moreover, delay nodes have been added to enhance communications regularity between nodes of the graph and to obtain nodes with at most one external input. Operation nodes correspond to the computation of multiply/add, division, rotation angle and rotation.

The graph in Figure A.2 is divided into sections that correspond to different iterations in the algorithm. We can distinguish four parts in each section of the graph in Figure A.2, namely those used to operate on the four different matrices. In the top-left part, leftmost elements of matrix A are used to compute rotation

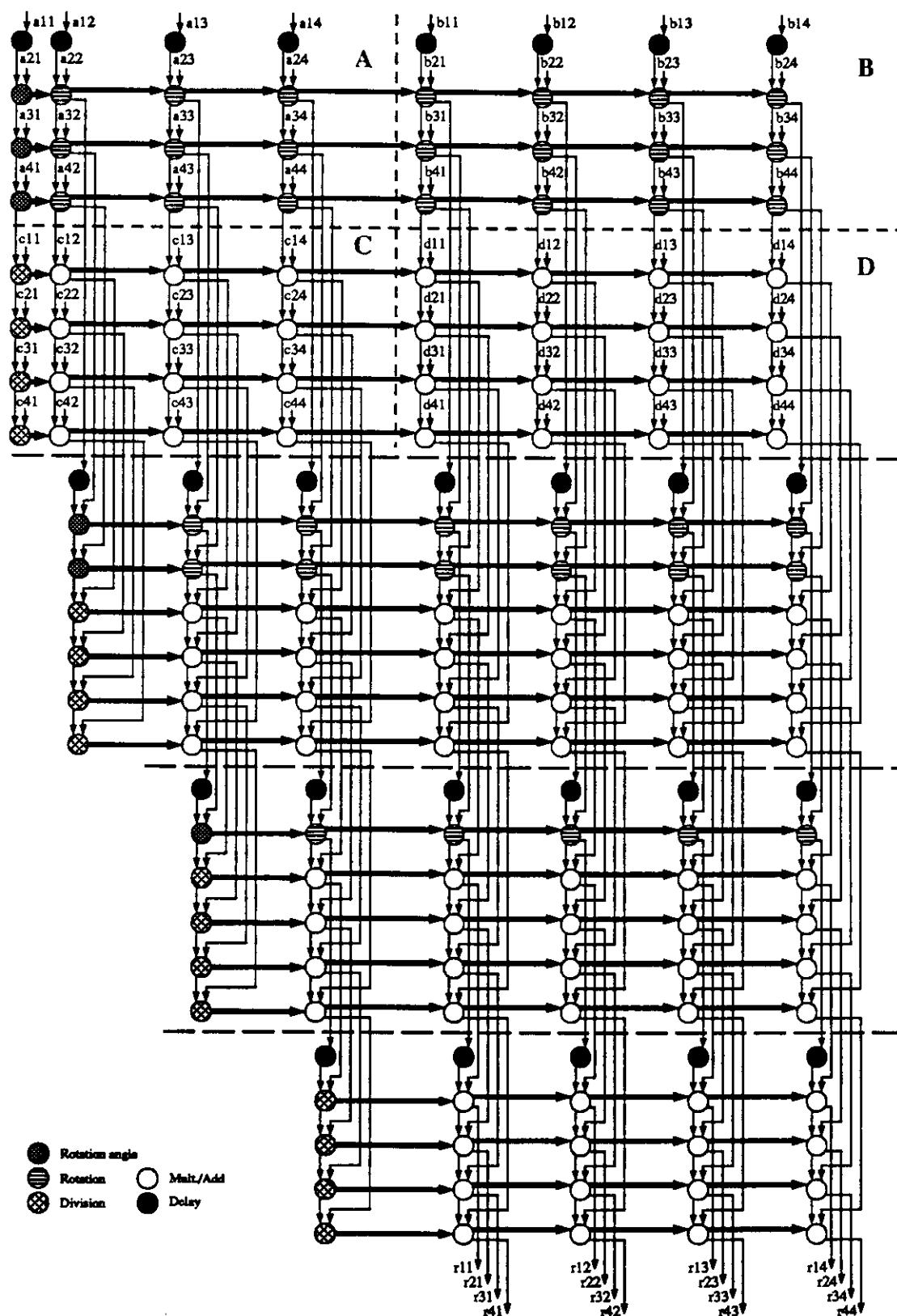


Figure A.2: The fully-parallel graph without broadcasting

angles. These angles are broadcasted horizontally to the remaining elements of A on the same row and to the elements of B also on the same row. All these elements are rotated according to such angles. Elements of the transformed matrix A (say A') and the rotated matrix B (say B') flow towards the lower parts of the section. In the lower-left part, leftmost elements of A' are used to compute pivots for performing Gaussian elimination on matrix C . Pivots are broadcasted horizontally and used together with elements of B' to perform the same transformation on matrix D .

The graph in Figure A.2 is now transformed into an MMG by allocating each section of the graph to a different plane of a three-dimensional graph, as shown in Figure A.3. In what follows, we use this MMG to derive arrays for the Faddeev algorithm.

A.3 Arrays for problems with fixed-size data

Let us consider first deriving G-graphs for problems with fixed size data, and realizing the G-graphs as two-dimensional arrays. As an example, the G-graph obtained by coalescing prisms of size 1 by 1 along the Z -axis is depicted in Figure A.4. The G-graph is directly realized as the systolic array shown in Figure A.5. This array is characterized by a region of maximal utilization of cells (the bottom rightmost one, corresponding to the computation of matrix D in the algorithm); utilization of cells decreases for cell towards the top and left of the array. Moreover, this array requires external input in every cell, while results are left inside the n^2 lower rightmost cells.

Deriving G-graphs by grouping along axes X and Y and realizing the resulting graphs as arrays leads to the trapezoidal structures depicted in Figures A.6 and A.7. The array in this latter figure corresponds to the structure proposed in [Nash84]. These two arrays receive external input only at one side (i.e., left or top) and produce results also only at one side (right or bottom, respectively).

From the MMG one may search for two-dimensional arrays with higher throughput than what is achievable in the arrays above. While grouping along axes Y or Z , nodes in the different sections of the graph may be grouped separately. This approach leads to pairs of trapezoidal graphs as the ones used to derive the trapezoidal arrays described above. Nodes at the same position in these graphs are from the same path in the MMG. Since there is a connection along such a path, the two graphs have their corresponding nodes interconnected. Figure A.8 shows the array

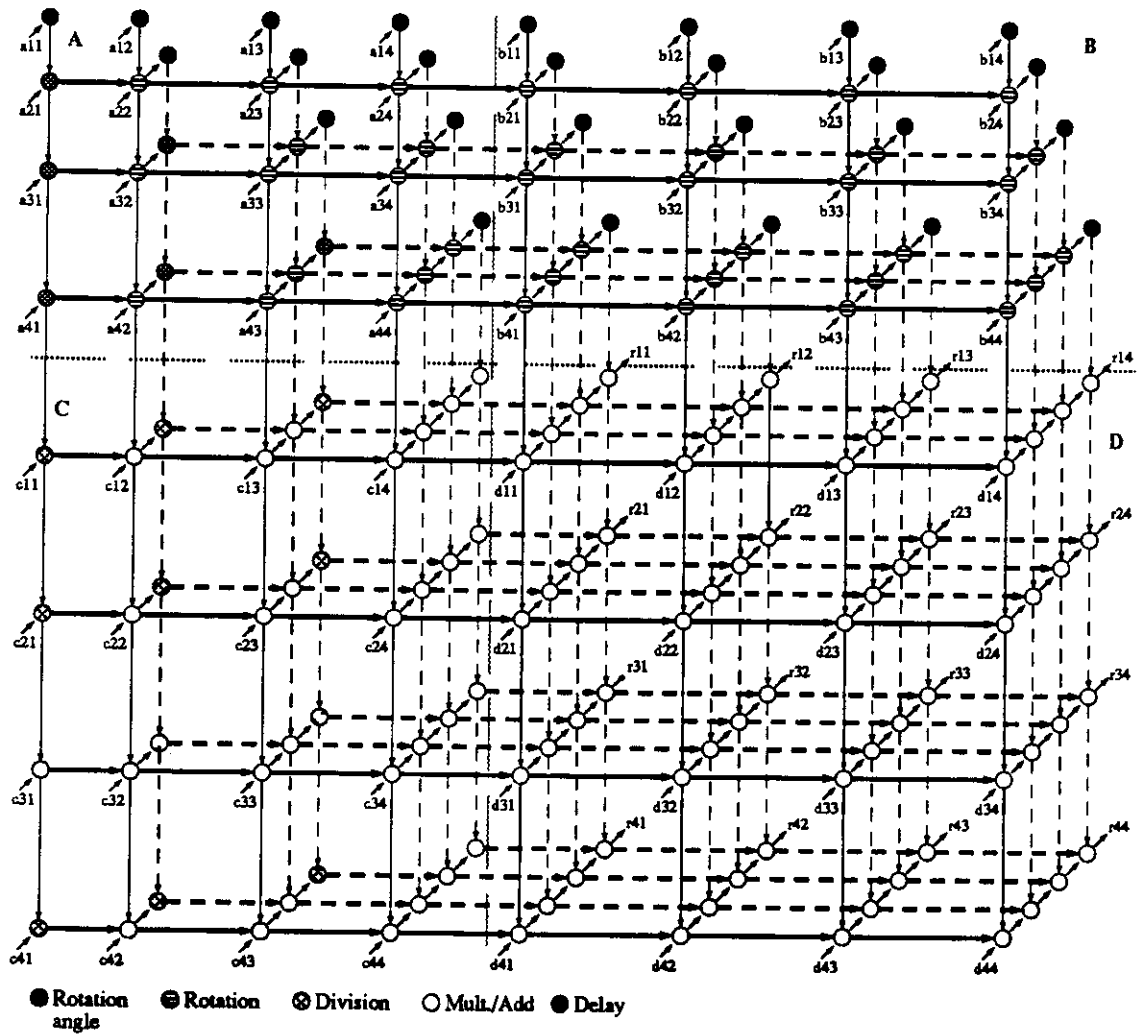


Figure A.3: The multi-mesh dependency graph

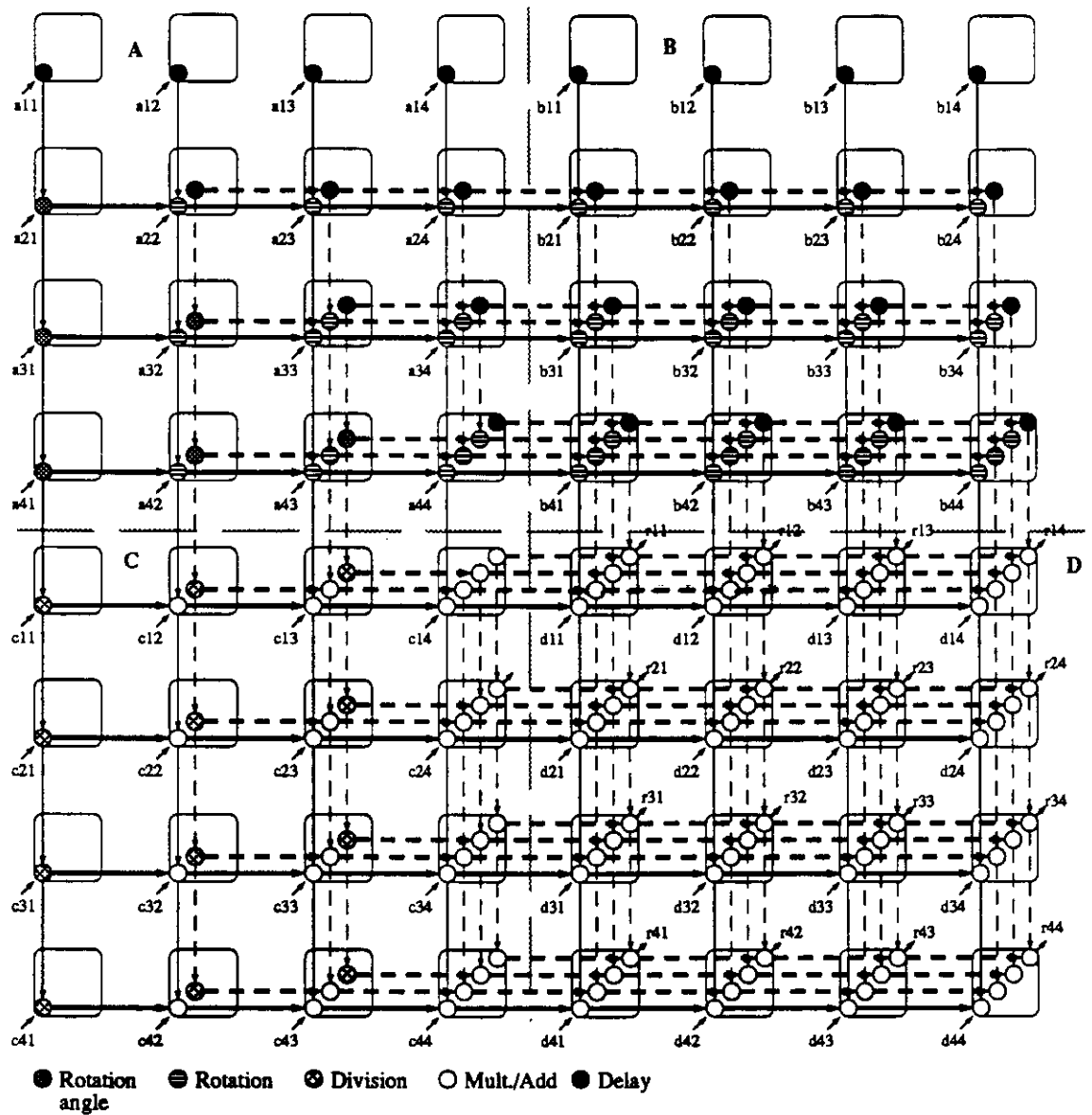


Figure A.4: The G-graph from grouping along the Z-axis

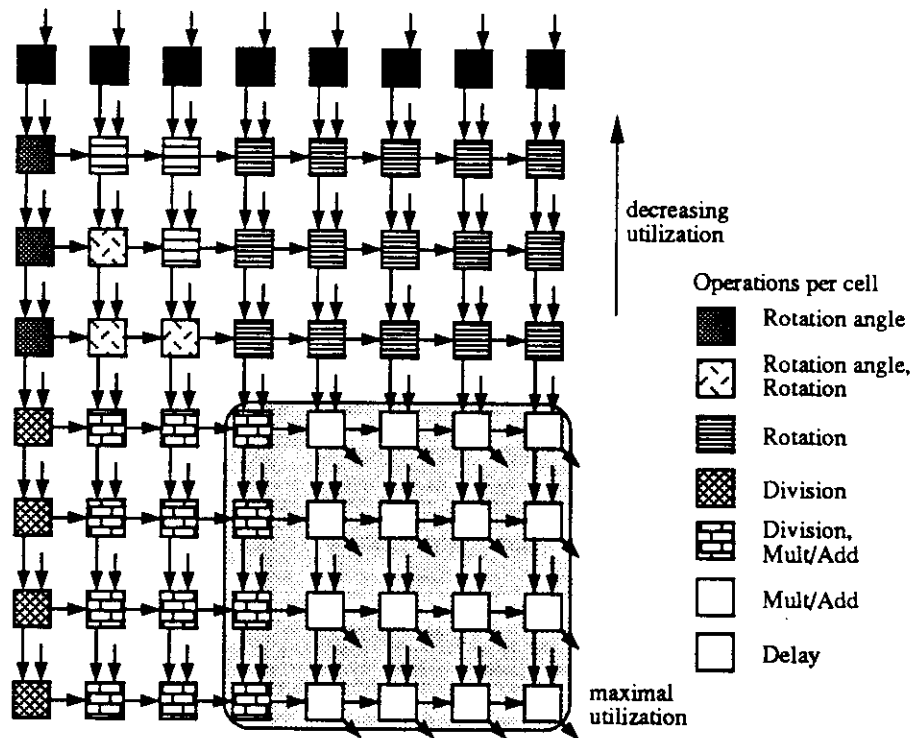


Figure A.5: The systolic array from grouping along the Z-axis

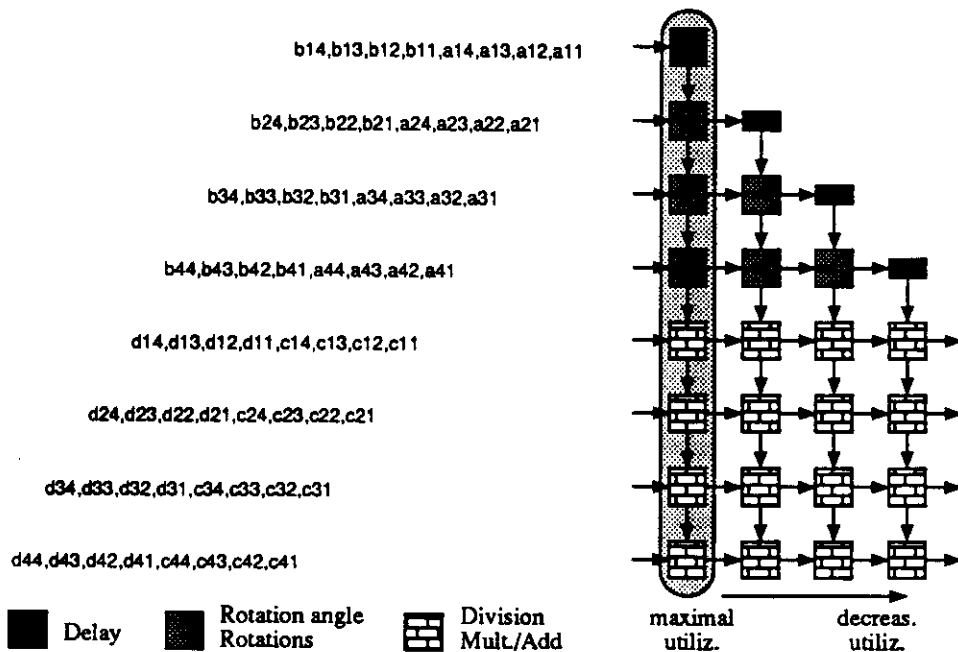


Figure A.6: The systolic array from grouping along the X-axis

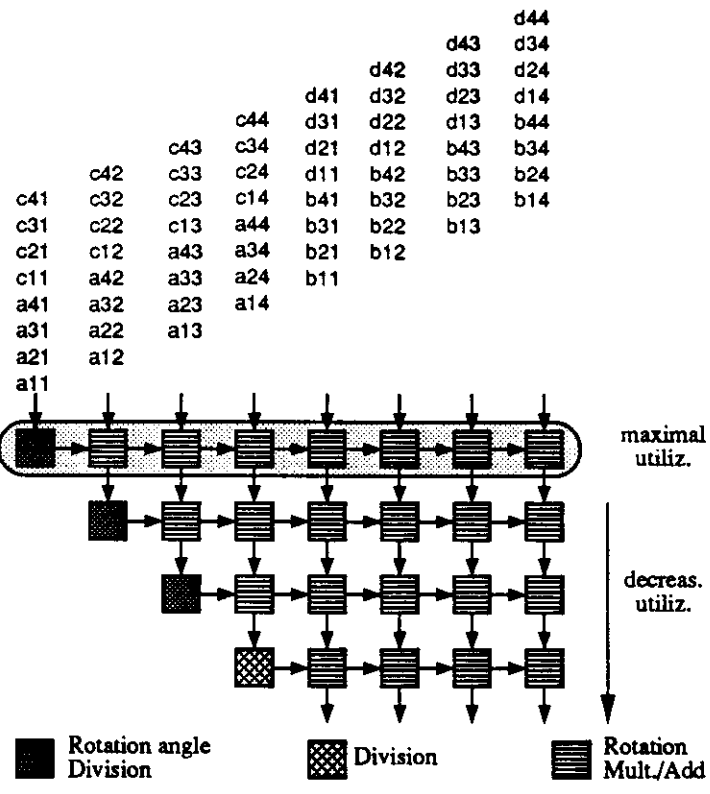


Figure A.7: The systolic array from grouping along the Y-axis

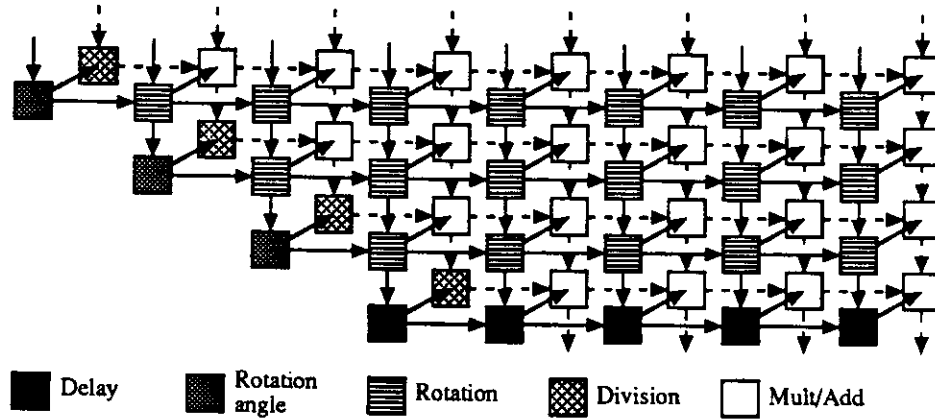


Figure A.8: A bi-trapezoidal array

obtained after realizing the pair of graphs derived by grouping along the Y -axis. The cell with longest computation time in this structure performs n operations, so that throughput is twice that of the single trapezoidal arrays.

A.4 Evaluation of the arrays for problems with fixed-size data

We compare now the performance of the different arrays to compute the Faddeev algorithm for fixed-size data devised in the previous section.

The number of computing nodes in Figure A.2 (i.e., the number of operations in the algorithm) is given by

$$\begin{aligned}
 N &= \sum_{i=0}^{n-1} (2n - i)(2n - i - 1) \\
 &= \frac{7}{3}n^3 - \frac{n}{3}
 \end{aligned}$$

This expression is different from the one given by DeGroot et al. in [Groo87], because they count as operations cycles when a cell is waiting to collect the first two operands before performing the first operation. This delay is due to the single-input capacity of cells. Consequently, their measure of complexity of the algorithm (i.e., $3n^3 + n^2$ operations) is greater than the actual value.

Performance and cost measures are shown in Table A.1 (the \pm sign in the entries for trapezoidal structures is due to the difference in number of cells between both trapezoidal schemes). This table includes a relative measure of complexity

Table A.1: Performance measures for arrays for fixed size problems

Array	# Cells	Throughput [1/ops]	Input pads	Utilization
Z-axis	$4n^2$	n^{-1}	$2n^2$	$\frac{7n^2-1}{12n^2} \rightarrow \frac{7}{12}$
X/Y-axes	$\frac{3}{2}n^2 \pm \frac{n}{2}$	$[2n]^{-1}$	$2n$	$\frac{7n^2-1}{9n^2 \pm 3n} \rightarrow \frac{7}{9}$
Bi-trapezoidal	$3n^2 \pm n$	n^{-1}	$4n$	$\frac{7n^2-1}{9n^2 \pm 3n} \rightarrow \frac{7}{9}$

Array	Cells complexity
Z-axis	$\frac{n}{2}(3n - 1)$ complex, $\frac{n}{2}(5n - 3)$ simple; $2n$ delays
X/Y-axes	n complex, $\frac{3n^2}{2} \pm \frac{n}{2}$ simple
Bi-trapezoidal	$2n$ complex, $3n^2 \pm n$ simple

of cells: a “simple” cell corresponds to a cell which does not perform division or computation of rotation angles, while a “complex” cell performs these two operations. Note that such a classification is not a rigorous one, because cell complexity is highly implementation-dependent.

From the expressions in Table A.1, utilization of the trapezoidal arrays is better than that of the square array, as it could be inferred from the MMG. The structure with two-planes offers twice the throughput of the other trapezoidal arrays with the same utilization. Thus, such an alternative is more attractive if one can afford the larger number of PEs and input connections.

A.5 The partitioning problem in the Faddeev algorithm

In the previous sections we devised two-dimensional arrays for the Faddeev algorithm for fixed-size matrices. We center our attention now to the problem of partitioning this algorithm for problems with large size data. We first review some structures previously proposed to execute the Faddeev algorithm in partitioned mode and later apply our method.

A.5.1 Partitioned structures previously proposed

An array to compute the Faddeev algorithm in partitioned mode was proposed by Nash et al. in [Nash86a]. This structure is based on the one proposed in [Nash84] for fixed-size matrices. It consists of a square array which is used to process both triangular and rectangular portions of a trapezoidal model of the algorithm such as the one shown in Figure A.7. The scheme partitions matrices A and B into vertical strips as wide as the size of the array and feeds such strips sequentially to the array (i.e., applies cut-and-pile as partitioning strategy). After both A and B have been processed completely, matrices C and D are partitioned and processed in the same manner.

Putting Nash et al. procedure in terms of transformations as those used in our method, they separate the nodes in the different sections of the dependency graph shown in Figure A.2 and group them independently, leading to a pair of trapezoidal subgraphs (as the one that leads to the bi-trapezoidal array shown in Figure A.8). Moreover, they map each trapezoidal subgraph independently onto the target square array. As we described in Section A.3, the two subgraphs have their corresponding nodes interconnected. Such interconnections represent intermediate results from processing A and B that are needed to process C and D . Mapping the trapezoidal subgraphs independently implies that intermediate results are left stored inside the array. As a consequence, Nash et al. scheme requires an unloading/loading step every time a new part of a strip (i.e., a new cut) is brought into the array for processing. In addition, a skewing/de-skewing procedure is used for maximal utilization of the array when computing the square portions of the algorithm. The authors claim that this overhead is not significant, although they do not provide figures for such a claim.

A different partitioning approach was used by Chuang and He in [Chua84], based on I/O constraints. They do not use Givens rotations for the annulment of matrix C in the Faddeev algorithm, but use neighbor pivoting instead [Gent81]. Consequently, the dependency graph for their version of the Faddeev algorithm is not the one shown in Figure A.2. They describe the algorithm using a trapezoidal model as the G-graph that leads to the array shown in Figure A.7. Different partitioning schemes are proposed, leading to different structures. The most adequate of those structures consist of an array composed of a triangular and a square section of the same width. To use such a structure, they partition the trapezoidal model into horizontal strips as wide as the size of the arrays. The triangular portion of the horizontal strips is executed in the triangular array, while the remaining of the

strips is executed in the square array (i.e., using cut-and-pile). Since the size of the rectangular part of a strip is larger than the triangular part, the square array is used several times in each horizontal strip while the triangular array is used only once, leading to low utilization of the triangular array. The other partitioning schemes proposed by Chuang and He exhibit similar characteristics.

Another work on partitioning the Faddeev algorithm was presented by De Groot et al. in [Groo87]. They describe the implementation of a partitioned scheme that corresponds to one of those proposed by Nash et al. in [Nash86a]. It consists of visualizing the algorithm as implemented by a large virtual array and mapping multiple contiguous cells from the virtual array into each cell of the target array (i.e., coalescing the virtual array). Nash et al. discarded this scheme in spite of its simplicity because it requires $O(n^2)$ memory locations per cell. De Groot et al. implementation uses an array of transputers with 128K memory per processor, so that memory is not a major limitation. In fact, their main concern is to increase the processor utilization due to the low communication bandwidth of their array. Consequently, they pay the cost of increased memory requirements with the objective of reducing communications.

We apply now our method to derive arrays for partitioned execution of the Faddeev algorithm.

A.5.2 Partitioning for linear arrays

Let us assume that we want to partition the Faddeev algorithm for n by n matrices so that it fits in a linear structure with only K cells, where $K \ll n$. For these purposes, we use the trapezoidal G-graph obtained by grouping the MMG along the Y-axis shown in Figure A.9. In this G-graph, horizontal paths of G-nodes have the same computation time and such time decreases for lower horizontal paths.

We realize the G-graph as a linear array by selecting G-sets of K G-nodes, as shown in Figure A.10a. The schedule of the G-sets is discussed later. Intermediate results from G-sets are saved in external memories. Those intermediate results include rotation angles or pivots flowing horizontally, and rotated or pivoted rows of the matrices flowing vertically. Such data is available at the boundary of the set, so that saving it in external memories is straight-forward.

The structure resulting from the approach outlined above is shown in Fig-

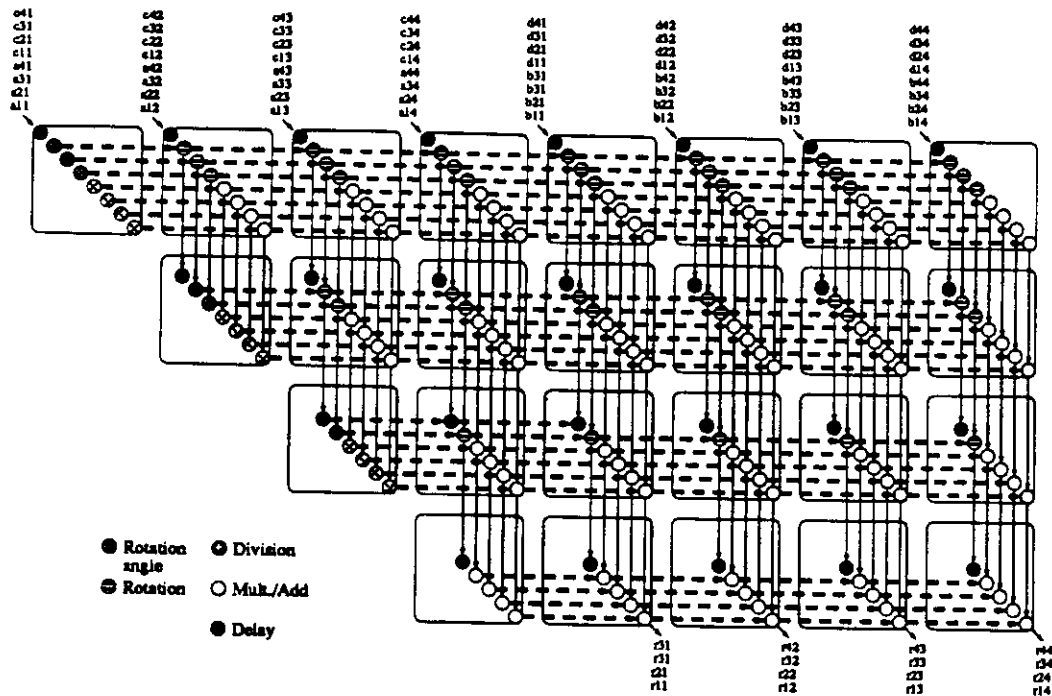


Figure A.9: The G-graph from grouping along the Y-axis

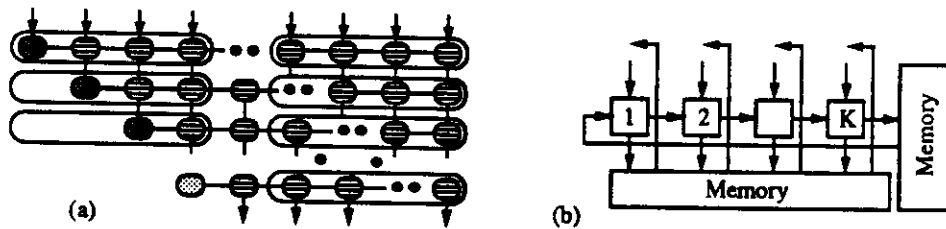


Figure A.10: Partitioned linear array for the Faddeev algorithm

ure A.10b. All G-nodes executed concurrently have the same computation time, except when executing boundary sets in some horizontal paths which might not use all cells in the array.

A.5.3 Partitioning for two-dimensional arrays

Mapping the trapezoidal G-graph shown in Figure A.9 for execution in a two-dimensional structure with K cells requires to simulate a triangular array and a square array, because those are the major components of the G-graph. Both requirements are fulfilled in a square array, with the proper control signals. G-

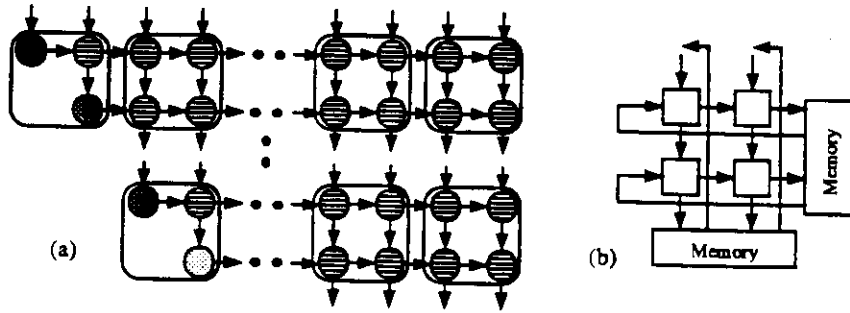


Figure A.11: Two-dimensional partitioning of the Faddeev algorithm

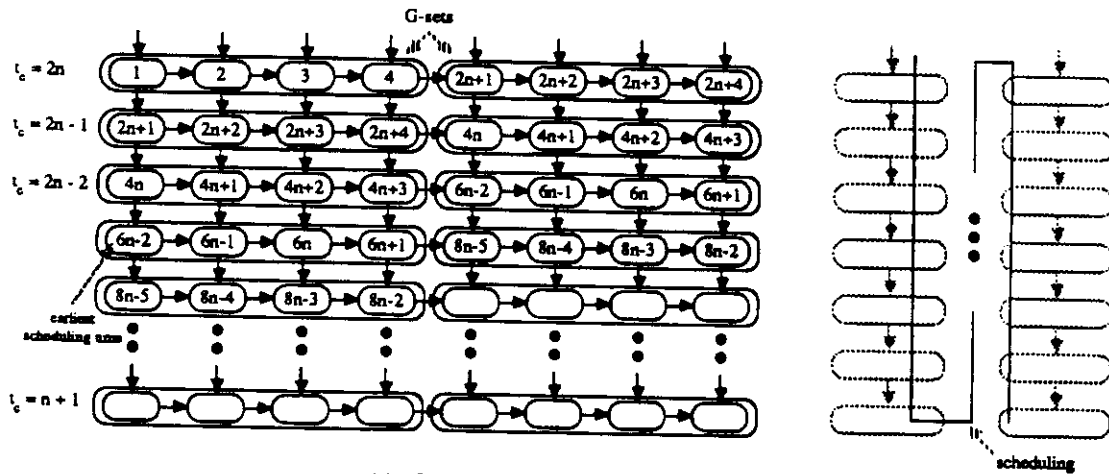
sets are mapped onto the array as square blocks of \sqrt{K} by \sqrt{K} nodes, as shown in Figure A.11a. Intermediate results are saved in external memories. Those intermediate results consist of rotation angles or pivots flowing horizontally, and rotated or pivoted rows of the matrices flowing vertically. The structure resulting from this approach is shown in Figure A.11b. Note that computation time of G-nodes is not the same for all nodes in a G-set.

A.5.4 Scheduling and I/O in partitioned Faddeev algorithm

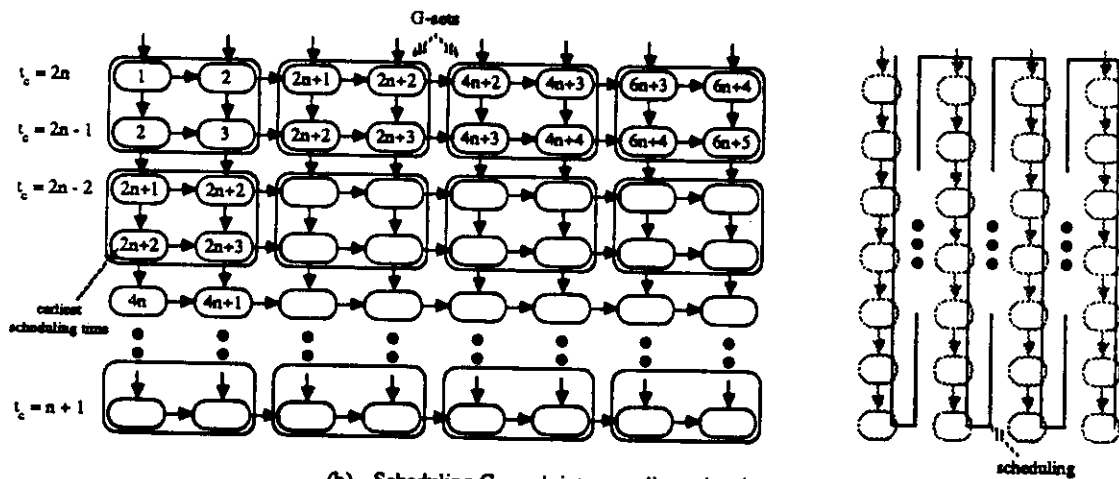
We discuss now the schedule of G-sets mapped onto linear and two-dimensional arrays. To illustrate such schedule, we use the G-graph shown in Figure A.12 (this graph corresponds to the internal portion of a large G-graph), where G-nodes in horizontal subpaths have identical computation time. Nodes in Figure A.12 have been tagged with their earliest scheduling time (i.e., at what time they could start execution) relative to a reference time.

Mapping onto a linear array was performed by composing a G-set with nodes in horizontal subpaths, because such nodes have the same computation time. Scheduling G-sets can be done by horizontal or vertical subpaths. Due to I/O bandwidth, we choose to schedule G-sets by vertical subpaths as depicted in Figure A.12a. Scheduling G-sets for execution in a two-dimensional array is similar to the linear array discussed above. Due to I/O bandwidth, we also choose to schedule G-sets by vertical subpaths, as illustrated in Figure A.12b. With these schedules of G-sets, suitable arrays for partitioned execution of the Faddeev algorithm are shown in Figure A.13. In these cases, I/O bandwidth is given by

$$A_{I/O} = \frac{2nK}{\sum_{i=1}^n t_{c_i}}$$



(a) - Scheduling G-graph into linear array



(b) - Scheduling G-graph into two-dimensional array

Figure A.12: Scheduling G-graph into linear and two-dimensional arrays

$$\begin{aligned}
&= \frac{2nK}{(2n+1)n - \frac{1}{2}n(n+1)} \\
&= \frac{4K}{3n+1}
\end{aligned}$$

where t_{c_i} is computation time of G-nodes in the i -th horizontal path of the G-graph. Under the conditions described above, *linear and two-dimensional arrays have the same I/O bandwidth from the host.*

A.5.5 Comparison of arrays for partitioned execution

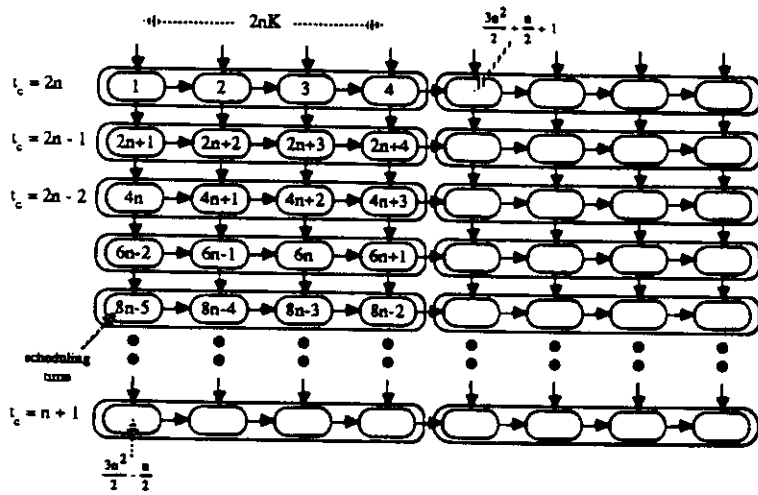
We compare now the characteristics of the arrays for the Faddeev algorithm in partitioned mode presented in previous subsections. We use the same performance measures to compare these arrays as the ones used for fixed-size cases, plus overhead due to partitioning.

As stated in Section A.4, the number of nodes in the dependency graph is $(\frac{7}{3}n^3 - \frac{n}{3})$. Throughput is determined by the computation time of the busiest cell in the array. Such information is obtained from the G-graph. In the following subsections, we present the derivation of the corresponding expressions for the different arrays.

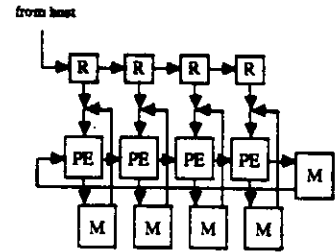
Linear array

When scheduling the G-graph shown in Figure A.7 for execution in a linear array with K cells, the first horizontal subpath of the graph is mapped in $2n/K$ sets. Each G-node in this subpath consists of $2n$ operations. The second horizontal subpath is mapped in $\lceil (2n-1)/K \rceil$ sets, because the length of the subpath is shorter and each G-node consists of $2n-1$ operations. This pattern repeats for all horizontal subpaths and the last one is mapped in $\lceil (n+1)/K \rceil$ sets. Therefore, the array is used for

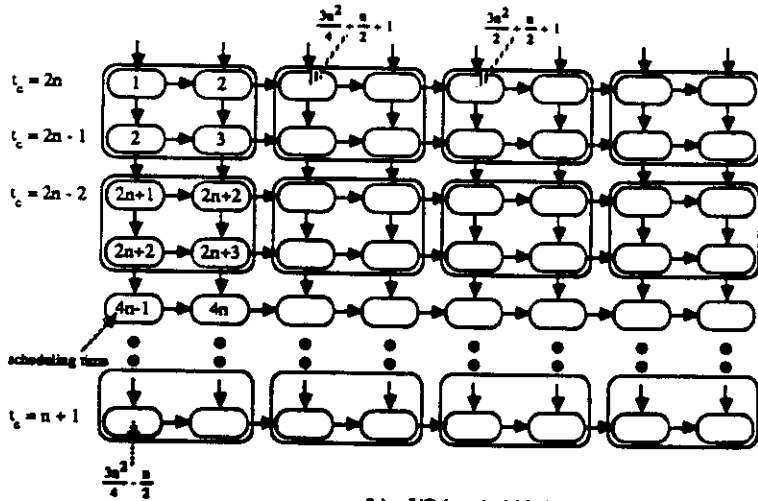
$$\begin{aligned}
\sum_{i=0}^{n-1} \left\lceil \frac{2n-i}{K} \right\rceil (2n-i) &= \sum_{j=0}^{\frac{n}{K}-1} \left[\left(\frac{2n}{K} - j \right) \sum_{i=0}^{K-1} (2n-i-jK) \right] \\
&= \sum_{j=0}^{\frac{n}{K}-1} \left(\frac{2n}{K} - j \right) \left[(2n-jK)K - \sum_{i=0}^{K-1} i \right] \\
&= \sum_{z=\frac{n}{K}+1}^{\frac{2n}{K}} \left[K^2 z^2 - \frac{K(K-1)}{2} z \right] \quad z = \frac{2n}{K} - j
\end{aligned}$$



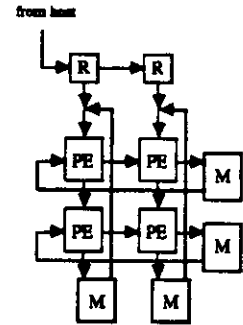
(a) - I/O bandwidth in linear array



$$I/O\ BW = (4m)/(3n+1)$$



(b) - I/O bandwidth in two-dimensional array



$$I/O\ BW = (4K)/(3n+1)$$

Figure A.13: I/O bandwidth in partitioning the Faddeev algorithm

$$\approx \frac{28n^3 - 9n^2(K-1)}{12K} \quad [\text{ops}]$$

and throughput is

$$T_{\text{linear}} = \frac{12K}{28n^3 - 9n^2(K-1)} \quad [\text{ops}]^{-1}$$

Utilization is given by

$$\begin{aligned} U_{\text{linear}} &= \frac{\sum \text{nodes}}{K/T} = \frac{\frac{1}{3}n(7n^2 - 1)}{K \left(\frac{1}{12K} [28n^3 - 9n^2(K-1)] \right)} \\ &= \frac{28n^2 - 4}{28n^2 - 9n(K-1)} \end{aligned}$$

Therefore, for large n , utilization tends to 1 and throughput tends to $\frac{3}{7} \frac{K}{n^3}$.

Square array

In the square array, G-sets are scheduled as square blocks of \sqrt{K} by \sqrt{K} nodes. Therefore, the first \sqrt{K} horizontal subpaths of the G-graph are mapped to the cells in $2n/\sqrt{K}$ sets. Computation time of these sets is given by the computation time of nodes in the first horizontal subpath, which consist of $2n$ operations. The next \sqrt{K} horizontal subpaths are mapped to the array in $(2n - \sqrt{K})/\sqrt{K}$ sets, because the length of these subpaths is shorter than previous ones. Computing these sets requires $(2n - \sqrt{K})$ operations. The remaining horizontal subpaths follow a similar pattern, so that the array is used for

$$\begin{aligned} \sum_{i=0}^{p-1} \left(\frac{2n - i\sqrt{K}}{\sqrt{K}} \right) (2n - i\sqrt{K}) &= \frac{1}{\sqrt{K}} \sum_{i=0}^{p-1} (2n - i\sqrt{K})^2 \\ &= \sqrt{K} \sum_{i=0}^{p-1} \left(\frac{2n}{\sqrt{K}} - i \right)^2 \\ &= \sqrt{K} \sum_{z=\frac{2n}{\sqrt{K}}-1}^{\frac{2n}{\sqrt{K}}} z^2 \quad z = \left(\frac{2n}{\sqrt{K}} - i \right) \\ &\approx \frac{7}{3} \frac{n^3}{K} \quad [\text{ops}] \end{aligned}$$

where $p = n/\sqrt{K}$ is the number of rows of \sqrt{K} by \sqrt{K} sets of G-nodes needed to cover the entire G-graph. Throughput is

$$T_{square} = \frac{3K}{7n^3} \quad [\text{ops}]^{-1}$$

and utilization is given by

$$\begin{aligned} U_{square} &= \frac{\sum \text{nodes}}{K/T} = \frac{\frac{1}{3}n(7n^2 - 1)}{K \left(\frac{1}{3K}(7n^3) \right)} \\ &= \frac{7n^3 - n}{7n^3} \end{aligned}$$

Therefore, for large n , utilization tends to 1 and throughput tends to $\frac{3}{7} \frac{K}{n^3}$

Nash et al. array

Nash et al. [Nash86a] use a square array to map their bi-trapezoidal model. They map each of the trapezoidal subgraphs independently, so that they require certain overhead in unloading/loading and skewing/de-skewing data. Since this overhead has not been reported quantitatively, we compute the throughput of their scheme ignoring the overhead. Consequently, this is an upper bound of what is achievable with their implementation.

Computing how long their array is used can be decomposed into computing how long each of the trapezoidal subgraphs uses the array, as follows (where $p = n/\sqrt{K}$):

- Triangularizing matrix A and transforming matrix B (i.e., executing the first trapezoidal subgraph). In this subgraph, there are as many nodes as in the trapezoidal graph mapped onto the square array discussed above, but the computation time of each node is smaller. Consequently, execution of these G-sets takes

$$\sum_{i=0}^{p-1} \frac{2n - i\sqrt{K}}{\sqrt{K}} (n - i\sqrt{K}) = \frac{5n^3 + 6n^2\sqrt{K} + nK}{6K}$$

- Annulling matrix C (i.e., executing the triangular portion of the second trapezoidal sub-graph). Execution of these G-sets takes

$$\sum_{i=0}^{p-1} \left(\frac{n - i\sqrt{K}}{\sqrt{K}} \right) n = \frac{n^3 + n^2\sqrt{K}}{2K}$$

- Updating matrix D (i.e., executing the rectangular portion of the second trapezoidal sub-graph). This operation takes

$$\sum_{i=0}^{p-1} \left(\frac{n}{\sqrt{K}} \right) n = \frac{n^3}{K}$$

These three terms, together with an extra term accounting for overhead in data transfers, give the throughput of this implementation as

$$T_{\text{Nash}} = \frac{6K}{14n^3 + 9n^2\sqrt{K} + nK + 6K(\text{OVHD})} \quad [\text{ops}]^{-1}$$

Utilization is given by

$$U_{\text{Nash}} = \frac{\sum \text{nodes}}{K/T} = \frac{14n^2 - 2}{14n^3 + 9n^2\sqrt{K} + nK + 6K(\text{OVHD})}$$

Consequently, Nash et al. implementation has the same throughput as the square array proposed above if there is no overhead in data transfers. In practice, the throughput and utilization are lower. In addition, their scheme exhibits complexity in the control required to perform those data transfers into and out of the array. I/O bandwidth of Nash et al. scheme is higher than the square array above, because of the loading/unloading of data.

De Groot et al. partitioned scheme

De Groot et al. [Groo87] evaluate their partitioning scheme and array considering that data communication is ten times slower than performing a single operation in a cell. This is a consequence of their implementation, an hypercube with transputers as nodes. In addition, they use completion time as performance measure. Although completion time is an important parameter for certain applications, we believe that throughput is more important so we use the latter for our evaluation.

De Groot's scheme consists of a trapezoidal structure with $(3P^2 + P)/2$ cells, where P is the dimension of the square and triangular portions of their array. To compare this array with the ones derived here, we express P in terms of K , that is in terms of the number of cells in our arrays. Thus $(3P^2 + P)/2 = K$ leads us to $P = \frac{1}{6}[\sqrt{24K + 1} - 1]$. The trapezoidal model describing the algorithm is partitioned into blocks of adjacent nodes and each block is mapped onto a single cell. Consequently, in the square portion of the trapezoidal model such blocks have (n/P) vertical subpaths with (n/P) nodes in each subpath. Throughput of their

implementation is given by the computation time of the first row of cells in the array, since those cells have the most operations to compute. Such cells perform $2n + (2n - 1) + \dots + (2n - \frac{n}{P} + 1)$ operations for each of the (n/P) vertical sub-paths. Thus, computation time of the cells is

$$\begin{aligned}
t_{cell} &= \frac{n}{P} \left[2n + (2n - 1) + \dots + (2n - \frac{n}{P} + 1) \right] \\
&= \frac{n}{P} \sum_{i=1}^{n/P} (2n - \frac{n}{P} + i) \\
&= \frac{n}{P} \left[\sum_{z=2n-\frac{n}{P}+1}^{2n} z \right] \quad z = 2n - \frac{n}{P} + i \\
&\approx 2n \left(\frac{n}{P} \right)^2 - \frac{1}{2} \left(\frac{n}{P} \right)^3 \quad [\text{ops}]
\end{aligned}$$

Replacing the value of P computed above, we obtain

$$\begin{aligned}
t_{cell} &= \frac{36n^3}{12K - \sqrt{24K + 1} + 1} - \frac{54n^3}{(12K - \sqrt{24K + 1} + 1)(\sqrt{24K + 1} - 1)} \\
&= \frac{18n^3}{12K - \sqrt{24K + 1} + 1} \left[2 - \frac{3}{\sqrt{24K + 1} - 1} \right] \\
&\approx \frac{36n^3}{12K - \sqrt{24K + 1} + 1} \quad [\text{ops}]
\end{aligned}$$

so that throughput is

$$T_{DeGroot} = \frac{12K - \sqrt{24K + 1} + 1}{36n^3} \quad [\text{ops}]^{-1}$$

and utilization becomes

$$\begin{aligned}
U_{DeGroot} &= \frac{\sum \text{nodes}}{N/T} = \frac{\frac{1}{3}n(7n^2 - 1)}{K \left(\frac{36n^3}{12K - \sqrt{24K + 1} + 1} \right)} \\
&= \frac{(7n^2 - 1)(12K - \sqrt{24K + 1} + 1)}{108n^2K} \\
&= \frac{84n^2K - 12K + (7n^2 - 1)(1 - \sqrt{24K + 1})}{108n^2K}
\end{aligned}$$

Therefore, for large n , utilization tends only to $7/9$ and throughput tends to $\frac{1}{3} \frac{K}{n^3}$.

The results above are summarized in Table A.2. We have not completed the entries in the table for Nash et al. array, because the overhead in loading/skewing data has not been reported quantitatively. Furthermore, we have not included

Table A.2: Performance measures for partitioned implementations

Array	Throughput [1/ops]	I/O BW
Linear	$\frac{12K}{28n^3 - 9n^2(K-1)}$	$\frac{4K}{3n+1}$
Square	$\frac{3K}{7n^3}$	$\frac{4K}{3n+1}$
De Groot	$\frac{12K - \sqrt{24K+1} + 1}{36n^3}$	—
Nash	$\frac{6K}{14n^3 + 9n^2\sqrt{K+nK+6K} + 6K(\text{ovhd})}$	$\frac{4K}{3n+1} + \text{ovhd}$

Array	Utilization	Overhead
Linear	$\frac{28n^2 - 4}{28n^2 - 9n(K-1)} \rightarrow 1$	none
Square	$\frac{7n^3 - n}{7n^3} \rightarrow 1$	none
De Groot	$\frac{(7n^2 - 1)(12K - \sqrt{24K+1} + 1)}{108n^2K} \rightarrow 7/9$	$O(n^2)$ storage
Nash	$\frac{14n^2 - 2}{14n^3 + 9n^2\sqrt{K+nK+6K} + 6K(\text{ovhd})}$	loading, skewing

I/O bandwidth for De Groot et al. scheme, because of their approach towards data transfer. From this table we infer that, for large n , both our linear and square arrays tend to the same throughput (i.e., $\frac{3K}{7n^3}$) and optimal utilization. In addition, both exhibit the same I/O bandwidth from the host. These linear and square arrays have better performance measures than the array proposed by De Groot et al. and do not exhibit the overhead required in the scheme proposed by Nash et al.

In addition to the performance measures described above, the linear array is more advantageous than the two-dimensional one because:

- it is simpler to implement
- for a finite value of n it has slightly higher utilization than the two-dimensional structure
- it is better suited to incorporate fault-tolerance capabilities (i.e., it is easier to skip a faulty cell in a linear array than to reconfigure a two-dimensional

structure)

Consequently, we conclude that *for partitioned execution of the Faddeev algorithm, a linear array offers better performance and implementation than a two-dimensional array.*

A.6 Conclusions

We have presented the application of our graph-based method to derive arrays for computing the Faddeev algorithm, for fixed-size and partitioned problems. We have derived two-dimensional implementations for such an algorithm, and we have compared these arrays with other schemes previously proposed in the literature. The evaluation used performance measures such as number of processing elements (PEs), throughput, I/O bandwidth, utilization of PEs and overhead due to partitioning.

Our results show that, for fixed-size problems, throughput reaches $(2n)^{-1}$ or n^{-1} with $2n$ and $4n$ cells respectively. Utilization of all these arrays tends to $7/9$. One of the arrays presented here corresponds to the one proposed in [Nash84].

Two-dimensional and linear structures for partitioned problems were also derived. We obtained a two-dimensional array that is more efficient and has less overhead than other structures previously proposed. We have shown that throughput of our partitioned implementation, both linear and two-dimensional, tends to $(3K)/(7n^3)$, where K is the number of cells. Moreover, we have shown that both linear and two-dimensional structures have the same I/O bandwidth from the host, namely $(4K)/(3n + 1)$ [words/ops].

APPENDIX B

Arrays to compute BA^{-1}

B.1 Introduction

In [Como87], Comon and Robert introduced a systolic array of $n(n+1)$ elementary processors that computes BA^{-1} in $(4n+p-2)$ time steps, where A is a dense (non-singular) n by n matrix, and B is a dense p by n matrix. Moreover, such an array can be directly extended to compute the vector $Y = BA^{-1}R$, where R is a vector with n components. These computations arise frequently in signal processing applications, as described by Comon and Robert and other researchers [Kung82, Ahme82, Spei81]. The algorithm used in [Como87] is shown in Figure B.1.

In this appendix, we apply our method to the algorithm in Figure B.1. We derive the array in [Como87] and another array that computes the algorithm in the same time but using only $[\frac{1}{2}n(n+1) + pn]$ units. Moreover, this new array exhibits throughput $(n+1)^{-1}$ and requires $(n+2p)$ I/O ports, while these measures are $(n+p)^{-1}$ and $2n$ respectively for the the array in [Como87].

B.2 Systolic arrays for computing BA^{-1}

Figure B.2 depicts the fully-parallel dependency graph for the algorithm in Figure B.1, where A is a 4 by 4 matrix and B is a 2 by 2 matrix. Such a graph is obtained by symbolic execution of the sequential algorithm in Figure B.1. This graph exhibits broadcasting of data, bi-directional data flow, and $O(n^2)$ I/O bandwidth.

We first transform the fully-parallel dependency graph in Figure B.2 into a multi-mesh graph. To achieve this transformation, we remove from the FPG those characteristics that are not allowed in the MMG, in the following order: replace broadcasting by transmittent data [Kung88c], remove bi-directional data flow by moving nodes to one side of the sources of broadcasting, and add delay nodes so

```

For k = 1 to n
begin
   $c_{kk}^{(k)} = 1/c_{kk}^{(k-1)}$ 
  For j = 1 to n, j ≠ k
     $c_{kj}^{(k)} = -c_{kk}^{(k)} c_{kj}^{(k-1)}$ 
  For i = (k + 1) to (n + p)
    begin
      For j = 1 to n, j ≠ k
         $c_{ij}^{(k)} = c_{ij}^{(k-1)} + c_{ik}^{(k-1)} c_{kj}^{(k)}$ ;
         $c_{ik}^{(k)} = c_{ik}^{(k-1)} c_{kk}^{(k)}$ ;
      end
    end
end

```

$$C^0 = \begin{bmatrix} A \\ B \end{bmatrix}, \quad C^n = \begin{bmatrix} F \\ D \end{bmatrix}, \quad D = BA^{-1}$$

Figure B.1: The algorithm to compute BA^{-1}

that dependencies are strictly between neighbor nodes. The resulting graph, shown in Figure B.3, consists of parallel meshes of nodes that are dependent, although the meshes do not have the same number of nodes.

According to our method, we now collapse the MMG onto a two-dimensional G-graph by grouping primitive nodes onto different nodes. We present two alternative groupings, along axes X and Y . Grouping along the Z -axis is significantly less efficient, as it can be inferred from the MMG, so that it does not merit discussing it here.

Grouping along the Y -axis

The first alternative that we consider consists of grouping nodes in the MMG by vertical paths, that is, each path along the Y -axis in the graph is collapsed onto a different G-node. The resulting G-graph, shown in Figure B.4a, leads to the array shown in Figure B.4b. This array corresponds to the one proposed by Comon and Robert. The performance of such an array is discussed later.

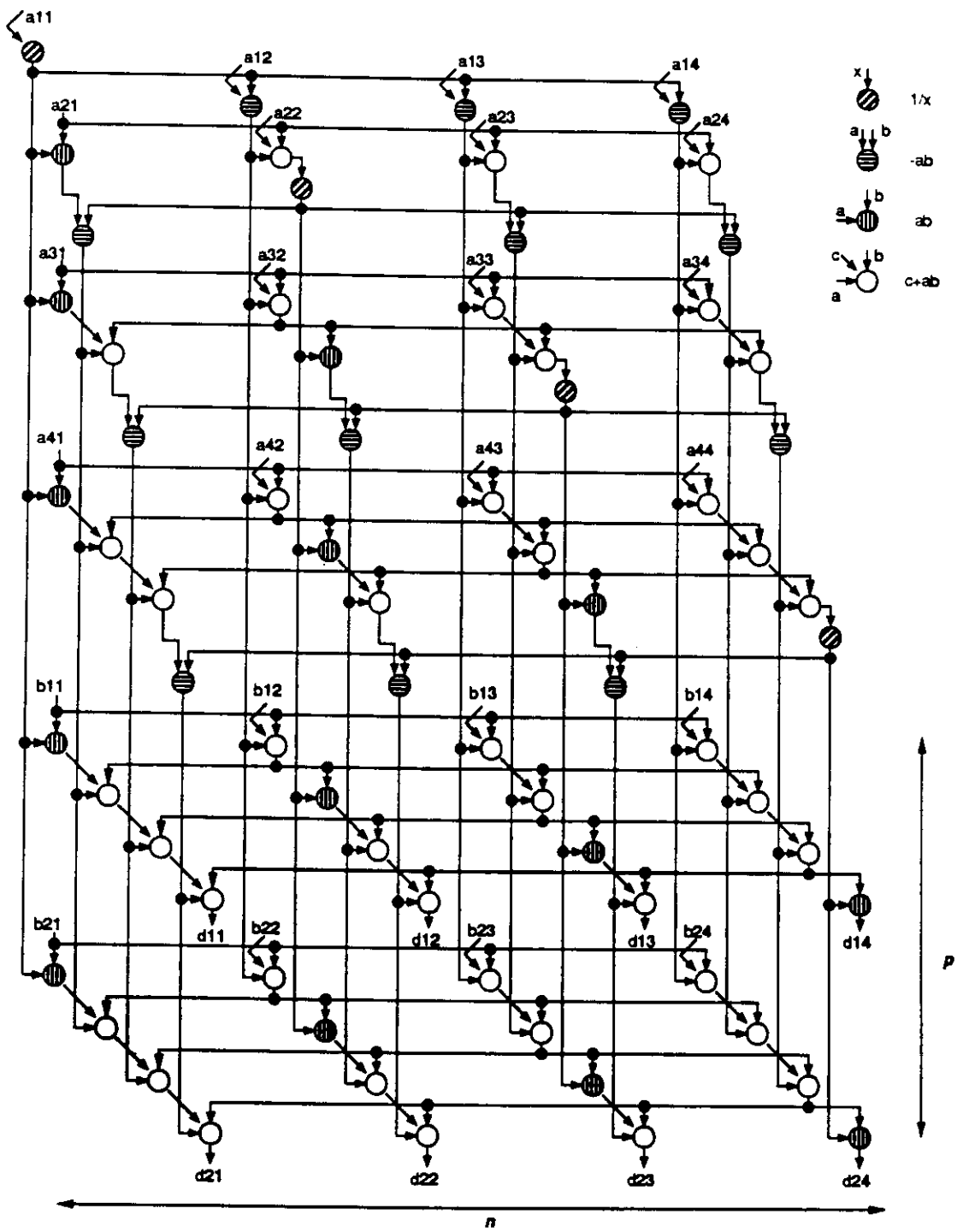


Figure B.2: The fully-parallel dependency graph

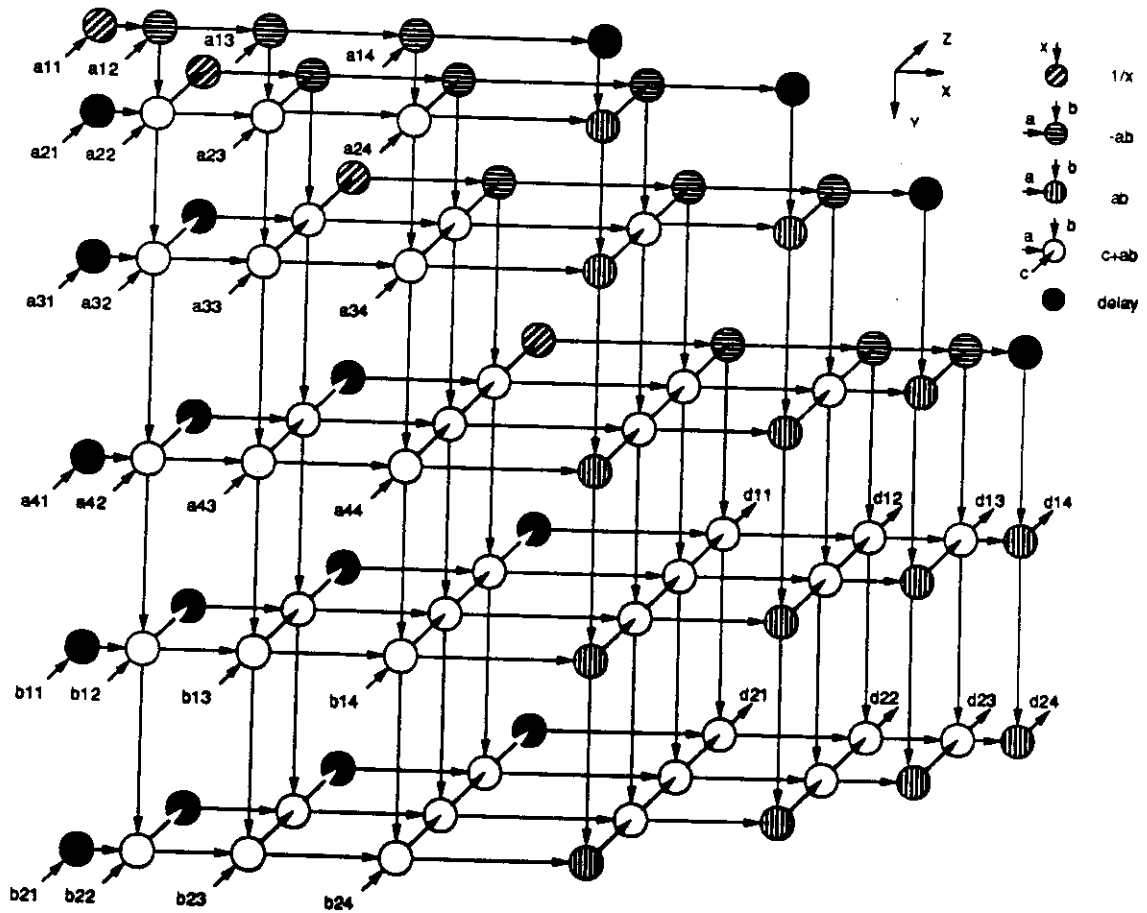


Figure B.3: The multi-mesh dependency graph

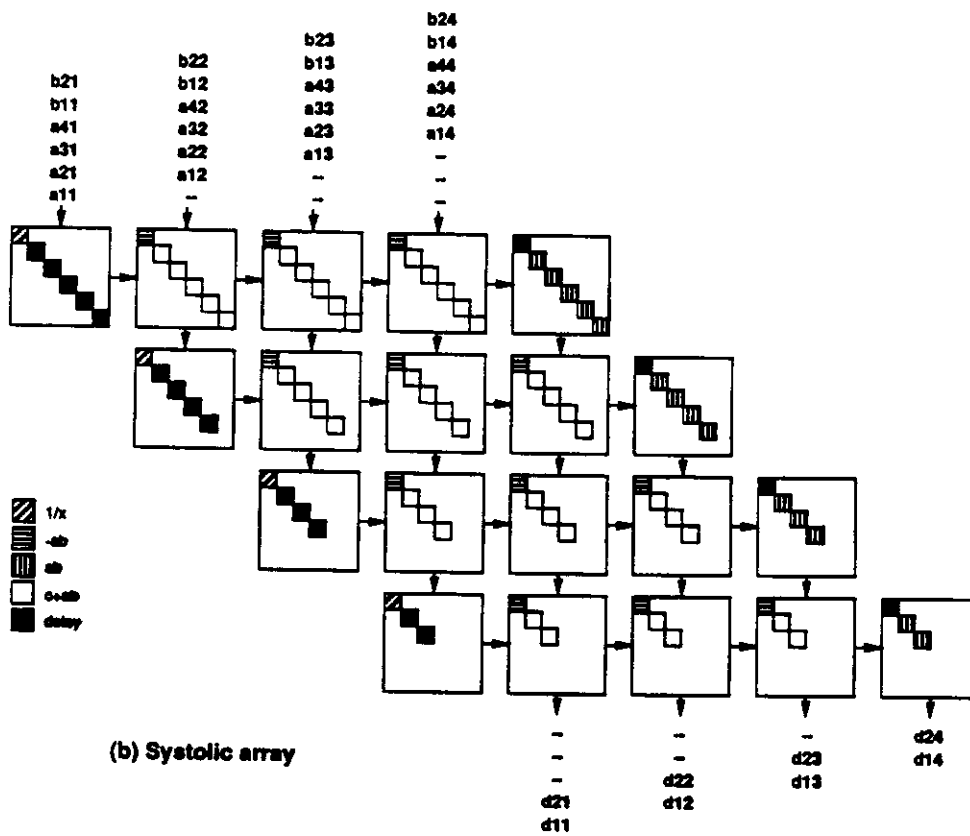
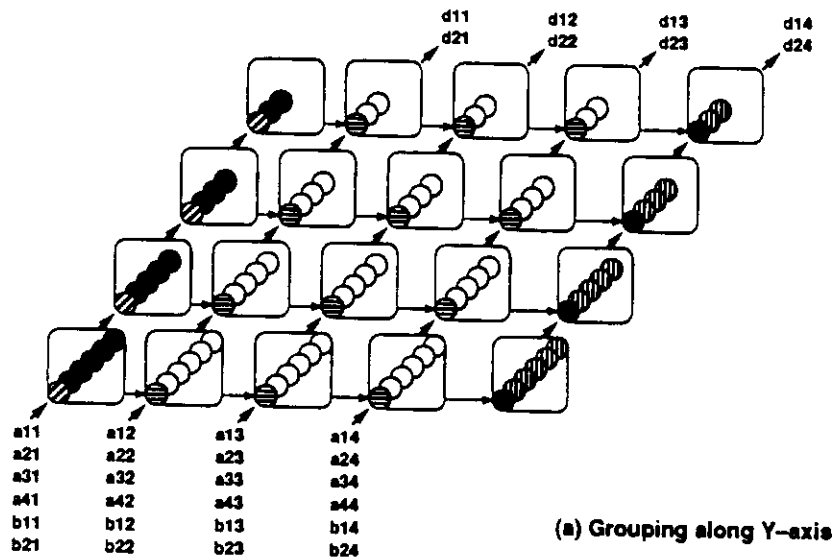


Figure B.4: Grouping nodes along the Y-axis

Table B.1: Performance measures of systolic arrays to compute BA^{-1}

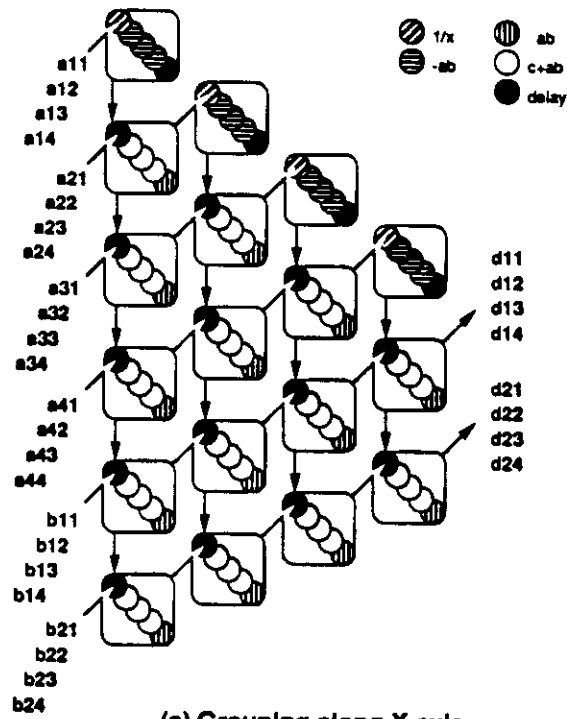
	Y-axis (Comon - Robert)	X-axis
Computation time	$4n + p - 2$	$4n + p - 2$
Throughput	$(n + p)^{-1}$	$(n + 1)^{-1}$
Number of cells	$n(n + 1)$	$\frac{1}{2}n(n + 1) + pn$
I/O ports	$2n$	$n + 2p$
Utilization	$\frac{n(n+2p+1)}{2(n+1)(n+p)} \rightarrow 1/2$	$\frac{n^2}{n(n+1)} \rightarrow 1$
Cells' complexity	3 types $1/x, [-ab, c + ab], ab$	2 types $[1/x, -ab], [c + ab, ab]$

Grouping along the X-axis

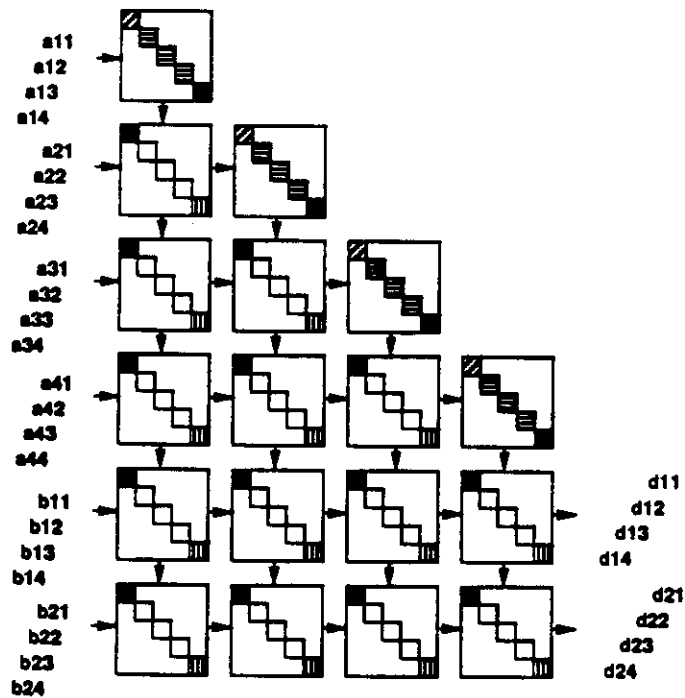
A second alternative consists of grouping nodes in the MMG along the X-axis, that is, each horizontal path in the graph is collapsed onto a different node. The graph obtained from such grouping is shown in Figure B.5a, which can be mapped onto the array shown in Figure B.5b. Performance of this array is described below.

B.3 Evaluation of the arrays

Table B.1 summarizes the characteristics and performance of the two arrays derived here. From such a table, we conclude that the array derived from grouping nodes along the X-axis, shown in Figure B.5, has better performance measures than Comon and Robert's scheme (obtained by grouping along the Y-axis) due to the following reasons:



(a) Grouping along X-axis



(b) Systolic array

Figure B.5: Grouping nodes along the X-axis

- It computes the algorithm in the same number of steps but using fewer units (if $p < n/2$). This improvement is achieved by having good utilization of cells, while this is not the case in Comon and Robert's scheme.
- Throughput of the array while computing successive instances of the algorithm (i.e., computation of the algorithm for different sets of data), is independent of p .
- Lower number of I/O ports.

However, Comon and Robert's scheme has the advantage that it can compute BA^{-1} for successive matrices B_1, B_2, \dots on the same array, without modifications. This case corresponds to extending the multi-mesh graph shown in Figure B.3 along the Y -axis, so that added nodes become part of existing groups when grouping along the Y -axis. This is in contrast to grouping along the X -axis, where the added nodes lead to more cells (in this case, successive matrices B should be handled as a partitioning problem).

The G-graphs in Figures B.4a and B.5a are also suitable for partitioning the algorithm (i.e., computing a large problem on a small size array), using the facilities for partitioning available in our method. For these purposes, the graph derived from grouping along the X -axis is more advantageous, due to the identical computation time of all G-nodes.

B.4 Conclusions

We have applied our data-dependency graph-based method for the design of systolic arrays to an algorithm that computes BA^{-1} . We have systematically derived two systolic structures for such algorithm, among them one previously proposed by Comon and Robert. We concluded that the array shown in Figure B.5 has better performance than Comon and Robert's scheme.

This exercise has shown that the application of our graph-based design method is powerful, producing results that are new and more efficient than others devised in ad-hoc manners.

APPENDIX C

Arrays for LU-decomposition with neighbor pivoting

C.1 Introduction

The LU-decomposition algorithm requires division by the diagonal elements of the matrix. Unless the matrix is well conditioned, it is necessary to use pivoting for numerical stability. The strategies suggested for this task are complete or partial pivoting. However, neither of these two schemes is amenable for parallel computation because they require global communication. Gentleman and Kung [Gent81] proposed another approach, called *neighbor pivoting*, where the pivot is selected as the largest element between two neighbors. They used this approach to devise a systolic array for matrix triangularization. They claim that neighbor pivoting is stable and that numerical experiments have confirmed so. We use this pivoting scheme for LU-decomposition to illustrate the capabilities of our method. We make no specific statements regarding the suitability neighbor pivoting from the numeric point of view.

C.2 The fully-parallel graph

The LU-decomposition algorithm with neighbor pivoting is described by the fully-parallel dependency graph shown in Figure C.1, for a problem of size 4 by 4. In this algorithm, pivots are selected as the largest of a diagonal element and the element in the next row and same column. That is, at iteration k the pivot is chosen as $\max(a_{k,k}, a_{k+1,k})$. If the chosen pivot is element $a_{k+1,k}$, rows k and $(k + 1)$ of L and A must be exchanged.

C.3 Deriving the MMG

The graph in Figure C.1 exhibits data broadcasting and bidirectional dependencies. Broadcasting is replaced by transmittent data, as depicted in Figure C.2.

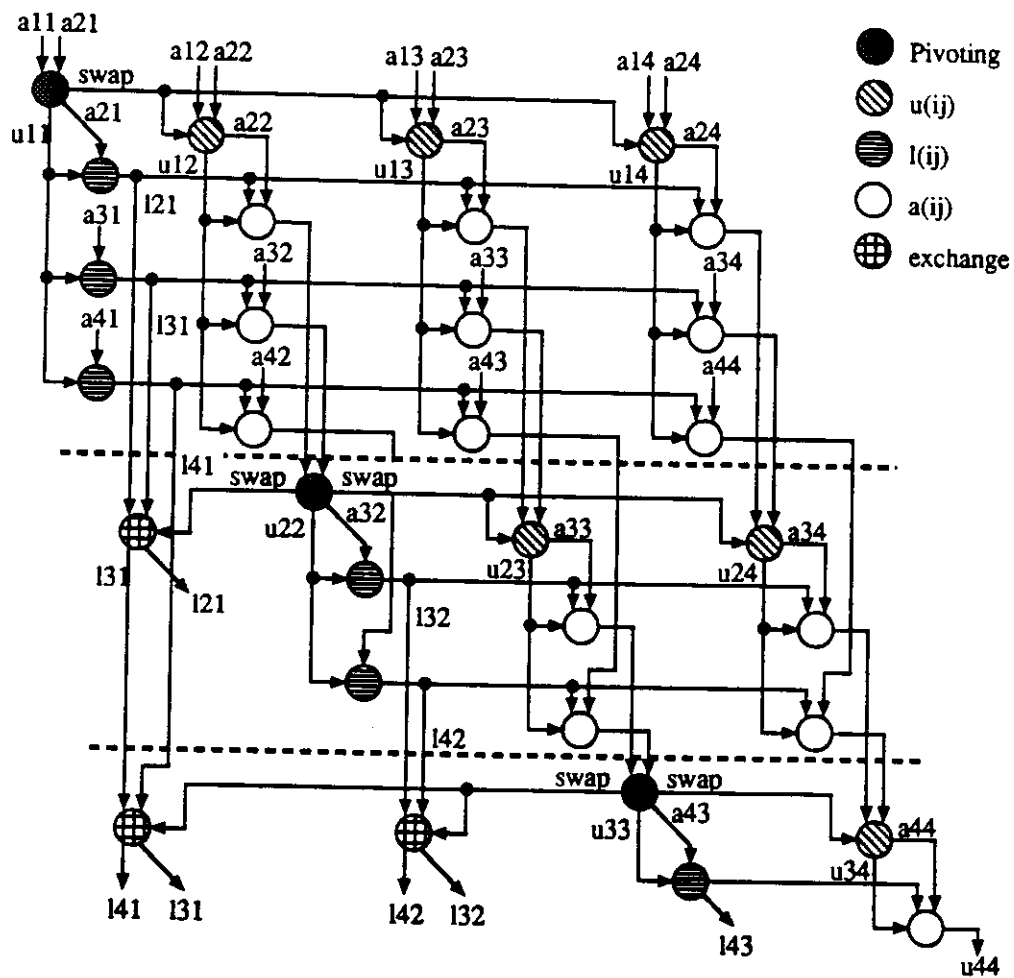


Figure C.1: LU-decomposition with neighbor pivoting

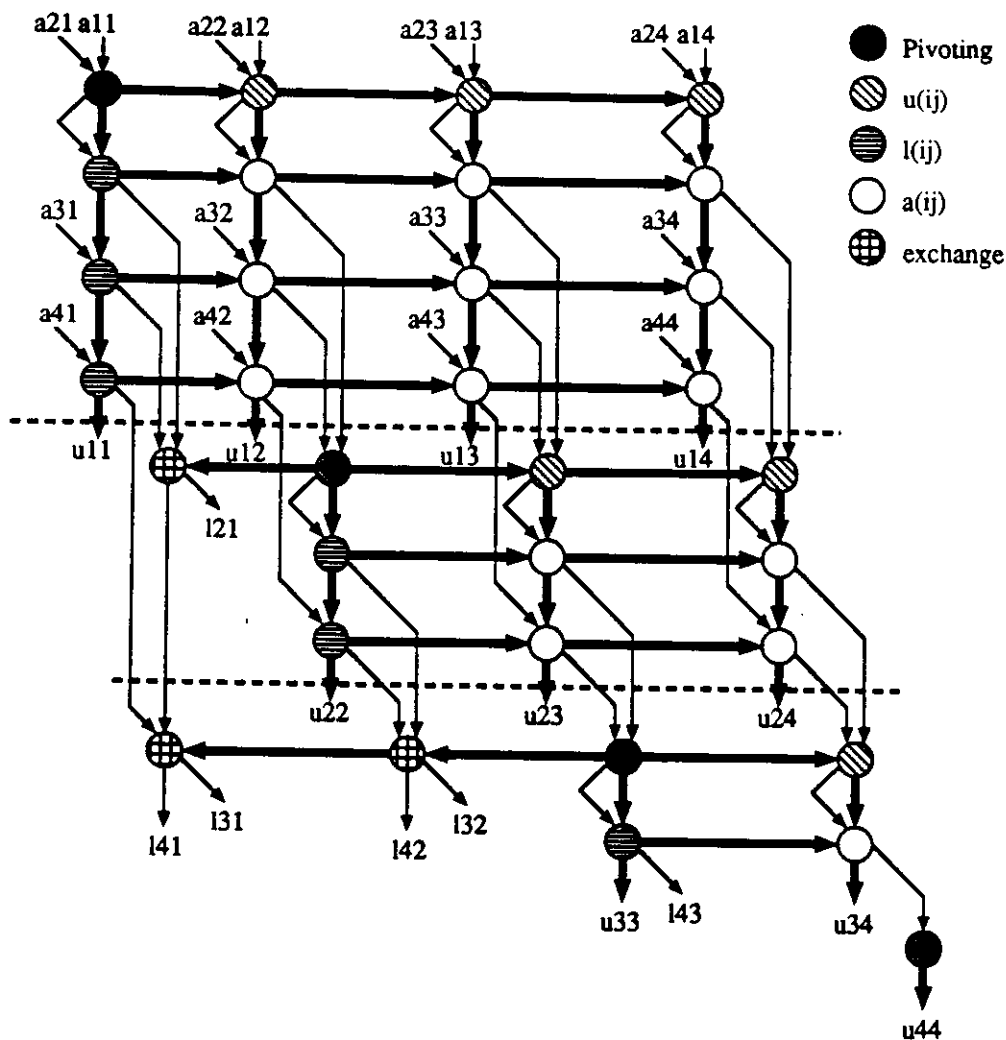


Figure C.2: Removing broadcasting

Bidirectional dependencies arise from the need to use the information about the selection of a pivot to exchange previously computed values $l_{i,j}$ ($i < j$) as well as to compute values $u_{i,j}$ ($j > i$). This undesirable feature is removed by flipping nodes at the left of the sources of bidirectional broadcasting to the right of such sources, as shown in Figure C.3.

The resulting graph is drawn in a three-dimensional space, as depicted in Figure C.4. Delay nodes have been added to the graph to make all dependencies among nearest neighbors. Note that nodes at the top of each mesh deliver two values to nodes immediately below them, while the remaining nodes deliver only one value in that direction. This characteristic is due to the need to transfer the

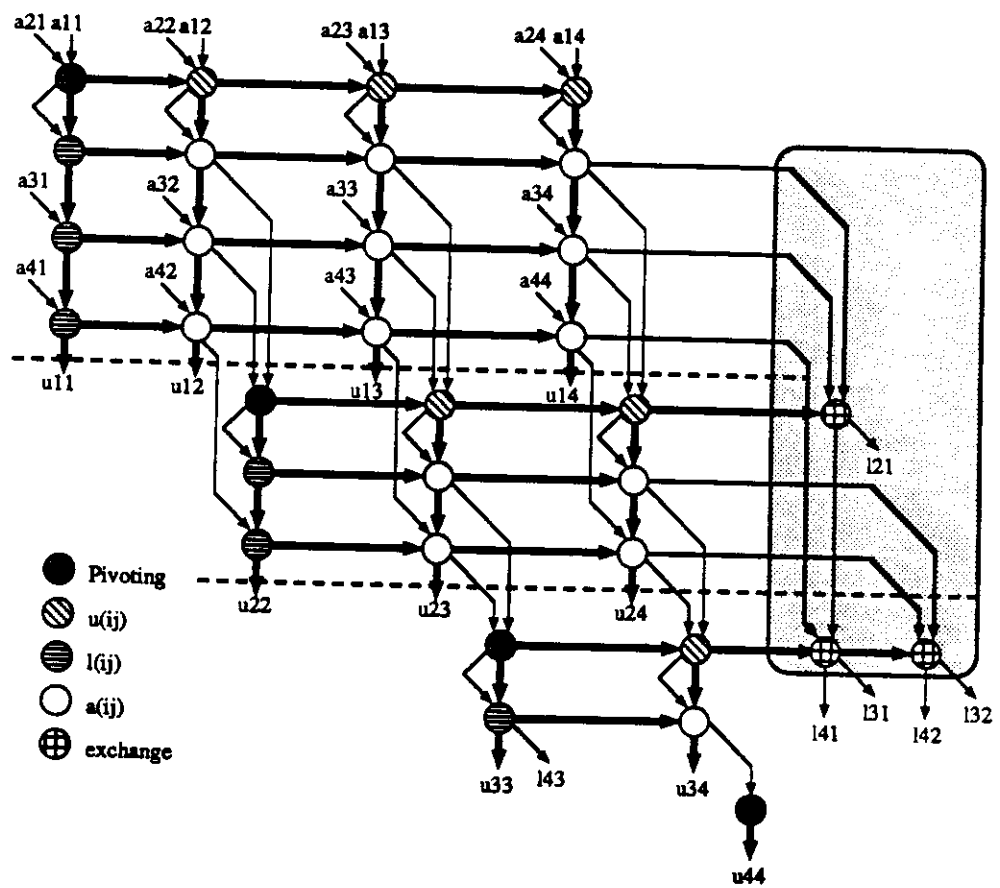


Figure C.3: Removing bidirectional dependencies

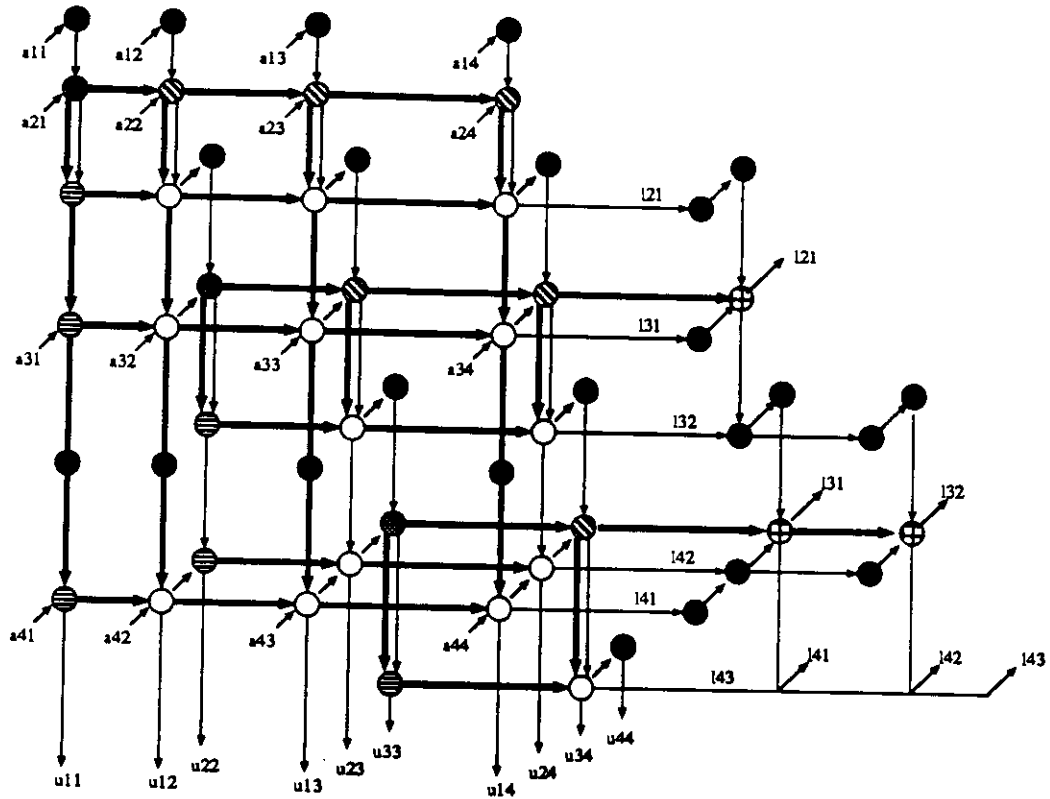


Figure C.4: The multi-mesh graph

values $u_{i,j}$ and $a_{i+1,j}$ to compute the next row of the updated matrix A . Since these two values are transferred and used together, they may be regarded as single value. However, we have chosen to draw two edges to emphasize the nature of the problem.

C.4 Deriving arrays for problems with fixed-size data

The graph in Figure C.4 is suitable for deriving arrays by grouping along any axis in the three-dimensional space. In particular, grouping along the X -axis is advantageous because it leads to fewer cells. The G-graph obtained from such a grouping is depicted in Figure C.5 and leads to a triangular array.

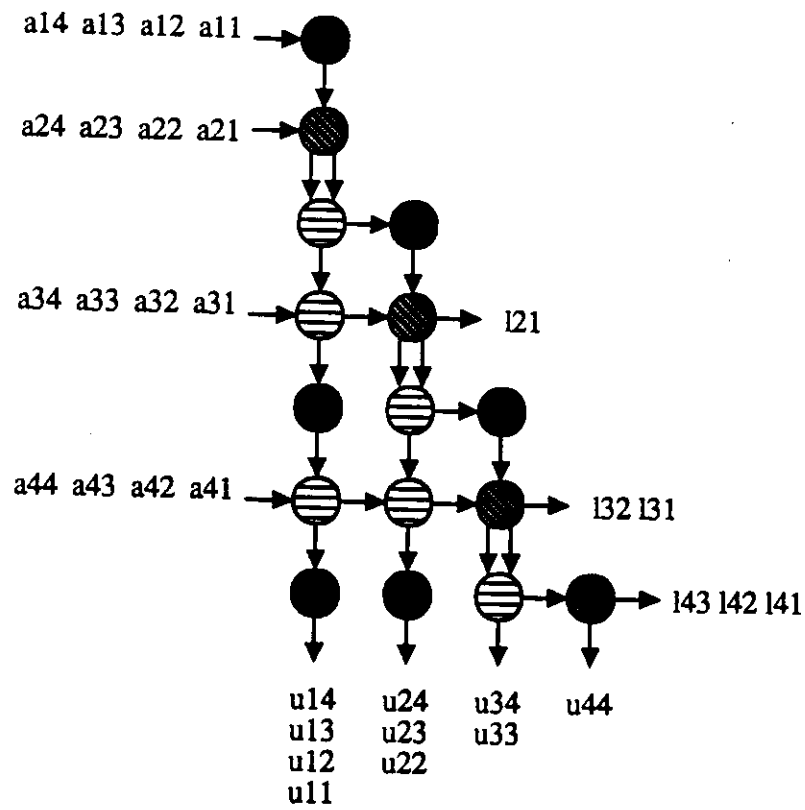


Figure C.5: A G-graph for LU-decomposition with neighbor pivoting

APPENDIX D

Algorithms with affine dependencies

The canonical form of matrix algorithms given in Chapter 5 does not have any restriction on how loop indices are used to access elements of matrices and vectors. As indicated there, the two common types of references usually considered are: (1) *uniform expressions* and (2) *affine expressions*. Uniform expressions are the most frequent of the two, and appear in all the algorithms used as examples throughout this dissertation. However, there are some algorithms that access variables using affine expressions. In this chapter, we illustrate the capabilities of our method using one example of such a class, namely an algorithm to perform convolution.

Affine expressions with loop indices have the general form $(i + j - k_0)$ (i.e., a linear combination of indices and a constant). In contrast, uniform expressions are of the form $(i - i_0)$ (i.e., an index plus/minus a constant). As will be shown here, the method is able to deal with affine dependencies without difficulty. It should be noted that using uniform or affine expressions to access variables does not imply that the algorithm must be a uniform or an affine system of equations, as required by other methods.

D.1 The convolution algorithm

As an example of affine dependencies, we consider the convolution algorithm depicted in Figure D.1. Note that the single statement in the body of the loop contains the term x_{i+j-1} , which originates the affine dependency.

```
For  $i = 1$  to  $n$ 
  For  $j = 1$  to  $m$ 
     $y_i = y_i + w_j x_{i+j-1}$ 
```

Figure D.1: Convolution algorithm with affine dependency

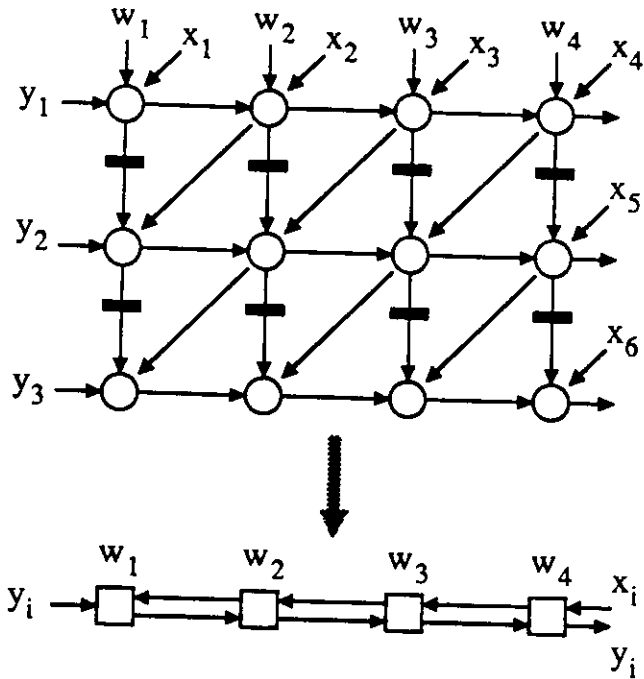


Figure D.2: Typical linear array for convolution algorithm

A typical realization of an algorithm as the one in Figure D.1 is shown in Figure D.2. In this figure, the algorithm has been represented first by a dependency graph, and the graph has been realized as a linear array with bidirectional flow of data. Values of x and y enter the array from opposite ends, weights are stored within cells, and results appear at the rightmost end. As a consequence of the bidirectional flow each cell is idle half of the time, waiting for data to be delivered from a neighbor cell. Utilization of the array is only $U = 0.5$.

Fully-parallel graph

The fully-parallel graph of the algorithm in Figure D.1 is obtained from the symbolic execution of that algorithm. This graph is shown in Figure D.3.

D.2 Deriving the MMG

The FPG in Figure D.3 consists of a sequence of vector operators with a single vector result. However, there is broadcasting across levels of the FPG. In fact,

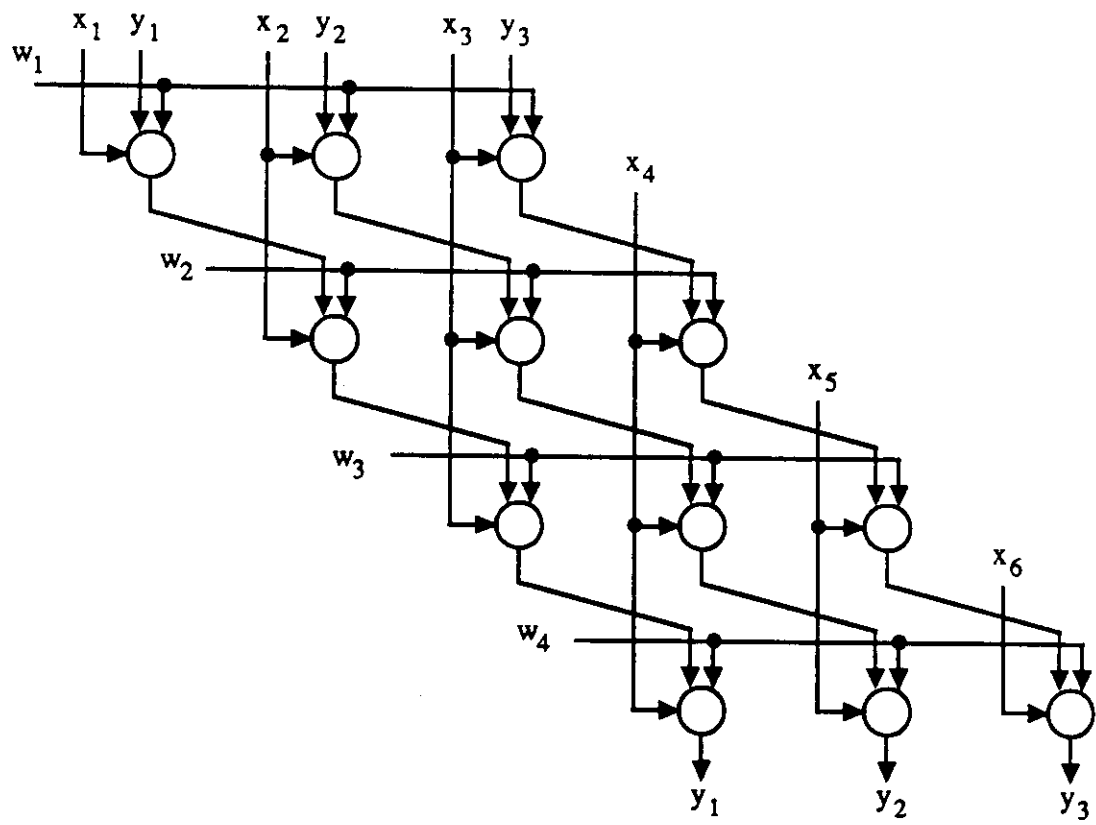


Figure D.3: The fully-parallel graph of convolution

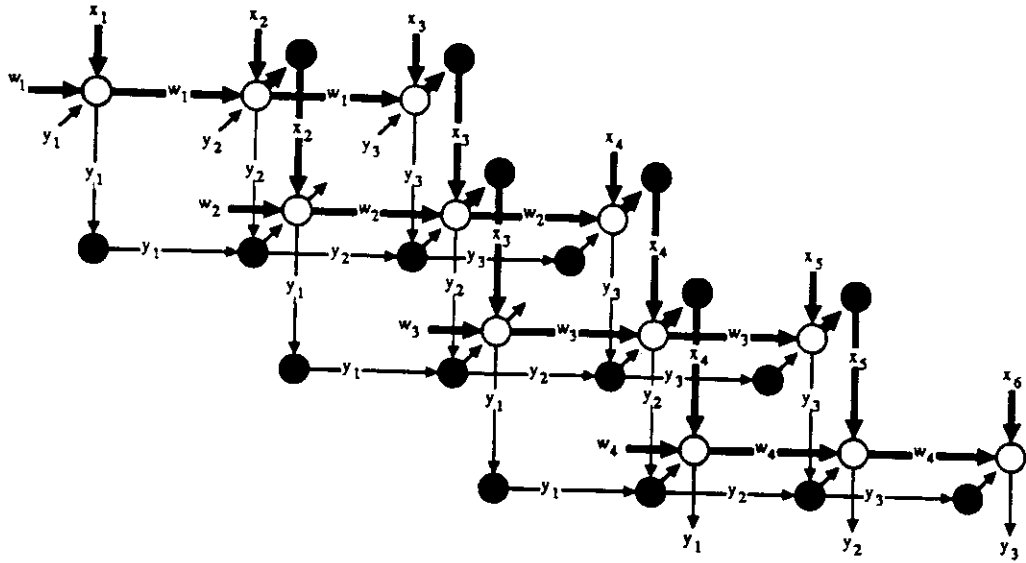


Figure D.4: The multi-mesh dependency graph

values of x_i are used in several levels (which are dependent). Consequently, after replacing broadcasting by transmittent data, each vector operator produces two outputs that are used as input at the next level of the graph: the result from the operation performed, and the transmittent data. Due to this property, drawing the MMG requires to use the construct described in Lemma 24; that is, primitive nodes of the vector operators must be allocated to adjacent planes in the three-dimensional space instead of the same plane.

The resulting MMG is shown in Figure D.4. For clarity in the description, edges of this graph have been tagged with the name of the corresponding data element. Note that there is one vector operand per plane, with operation nodes surrounded by delay nodes; these delay nodes are needed to obtain nearest-neighbor dependencies.

D.3 Deriving arrays for problems with fixed-size data

Let us consider the derivation of systolic arrays from the MMG in Figure D.4. We illustrate here only the case of grouping along the Z -axis (prisms of base size 1 by 1). Since operation nodes in Figure D.4 are flanked by delay nodes, grouping along the Z -axis leads to a two-dimensional G -graph where G -nodes have computation time $t_G = 3$ but they perform only one operation. This drawback arises as a consequence of introducing delay nodes to obtain nearest-neighbor dependencies.

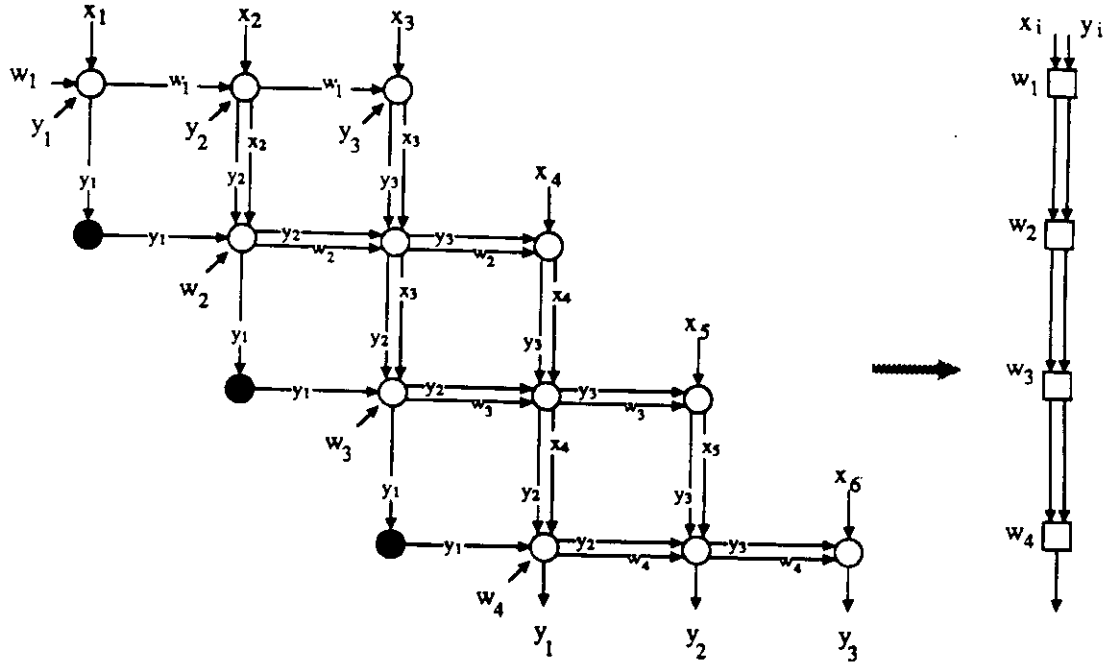


Figure D.5: Collapsing MMG along the Z -axis

Since in this algorithm there is only one vector operation per plane, the delay nodes introduce a significant overhead.

However, as we indicated earlier, vector operators in this algorithm produce two outputs that are used at the next level of the graph. These two outputs may be regarded as a single one, so that there is no need for delay nodes except at the boundary of the graph. This situation is depicted in Figure D.5, where we have collapsed the MMG into a G-graph along the Z -axis. We have explicitly shown two edges between nodes to indicate the transfer of two values.

The G-graph in Figure D.5 is realized as a linear array, as also depicted in the figure. This array has unidirectional flow of data, so that utilization of cells is optimal. Values of x and y enter from the same end of the array, weights are stored in the cells, and output values appear at the other end of the structure. (The last few values of x_i - x_4 through x_6 in the figure - are also input at the end of the array and transferred through the cells. That is, delay nodes are added to the G-graph so that those inputs may appear at the first cell.)

In the same manner described above, the method allows deriving other arrays by collapsing the MMG along the remaining axes.

