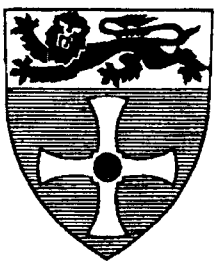
THE UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

UNIVERSITY OF
NEWCASTLE UPON TYNE



Automatic Parallelisation for a Class of URE Problems

by

Xian Chen

PhD Thesis

November 1995

Abstract

This thesis deals with the methodology and software of automatic parallelisation for numerical supercomputing and supercomputers. Basically, we focus on the problem of Uniform Recurrence Equations (URE) which exists widely in numerical computations.

We propose a complete methodology of automatic generation of parallel programs for regular array designs. The methodology starts with an introduction of a set of canonical dependencies which generates a general modelling of the various URE problems. Based on these canonical dependencies, partitioning and mapping methods are developed which gives the foundation of the universal design process. Using the theoretical results we propose the structures of parallel programs and eventually generate automatically parallel codes which run correctly and efficiently on transputer array.

The achievements presented in this thesis can be regarded as a significant progress in the area of automatic generation of parallel codes and regular (systolic) array design. This methodology is integrated and self-contained, and may be the only practical working package in this area.

Acknowledgements

I sincerely thank my supervisor, Dr. Graham Megson, for suggesting this area of research and guidance over the course of my work.

I would also like to express my appreciation to Prof. P. Lee and Dr. J. K. Wright, both members of my thesis committee. My thanks also go to the members of the Algorithm Engineering Research Group, their efforts were greatly appreciated.

Financial support for this research was provided by the Research Committee of University of Newcastle upon Tyne grant No. F406/RC/01 with additional funds from CVCP Overseas Research Students Awards Scheme ORS/92029017.

Finally, I thank mostly my mother, my sister and my wife for their support and understanding during my studies, which is the essential condition for me to complete the work.

Declaration

I certify that no part of the material offered here in this thesis has been previously submitted by me for a degree or other qualification in this or any other university.

Xian Chen

Dedicated to My Father

Terminology

Below is a list of common abbreviations and assumptions. Local variables are specified as necessary in the text.

Abbreviations

EVPA: Enlarged Virtual Processor Array.
SNF: Smith Normal Form of matrix.
HNF: Hermite Normal Form of matrix.
HSNF: Half SNF of matrix.
DF: Diagonalisation Form of matrix.
DC: Data Flow Cube.
DV: Data Flow Vector.
LSGP: Local Sequential and Global Parallel.
LPGS: Local Parallel and Global Sequential.
N-D: N dimensional.

General Assumptions

Unless otherwise stated you can assume the following

GCD stands for the greatest common divisor, and $gcd(\cdot)$ for the GCD of a vector.
$det(\cdot)$ stands for the determinant of a square matrix.
An integral matrix $\mathbf{M}$ is said to be an unimodular matrix if $\mid det(\mathbf{M}) \mid = 1$
$\mathbf{M} = diag(x_0, \cdots, x_n)$ means that $\mathbf{M}$ is a diagonal matrix with $x_0, \cdots, x_n$ as its diagonal entries.

A blackbold small letter, say $\mathbf{a}$, stands for a column vector.
A overbar vector indicates a row vector. For instance, $\overline{\mathbf{a}} = [a_0, a_1, a_3]$.
A bold capital letter, say $\mathbf{M}$, stands for a matrix.

Generally $\mathbf{b}_i \in \mathbf{B}$ means that $\mathbf{b}_i$ is the i-th column vector of matrix $\mathbf{B}$.
Generally $b_i \in \mathbf{b}$ means that b_i is the i-th entry of vector $\mathbf{b}$.
Generally $b_{i,j} \in \mathbf{B}$ means that $b_{i,j}$ is the (i,j)-th entry of vector $\mathbf{B}$.

Superscript "^{j}" means in processor-time domain.
Superscript "^{K}" means in the K-D time domain.
Superscript "$^{(p_0 \cdots p_i \cdots p_M)}$", e.g., $^{(2310)}$, means a permutation.
Superscript "^{q}" means in S^q domain.
Superscript "$^{q'}$" means in $S^{q'}$ domain under basis $\mathbf{E}$ (see below).
Superscript "^{s}" means in supernode domain.

Superscript "u" and "l" stands for the upper bound and the lower bound, respectively.

Subscript "SUC" means simple uni-directional connected mesh.
Subscript "SBC" means simple bi-directional connected mesh.

Terms for Global usage

$\mathbf{A}$ is the activity matrix of LSGP.

$\mathbf{A}_{m,N}$ is the coefficient matrix of the system of the m inequalities of the N-dimension polyhedron.

$\mathbf{A}^p$ is the positive coefficient matrix expressing $\mathbf{D}$ (see below) under the basis $\mathbf{E}$ (see below).

$\mathbf{A}^d$ is a positive coefficient matrix expressing $\mathbf{D}$, its entries are between 0 and 1.

A^M is M-D hypercube processor domain.

$\mathbf{a}$ is the index vector of a processor.

$\overline{\mathbf{a}}^{max}$ indicates the minimum sizes of the partitioning parallelepiped required to achieve the canonical dependencies.

$\mathbf{B}$ is obtained from $\mathbf{B}_s$ and defines the partitioning parallelepiped and is also the transformation from the original space to the supernode space.

$\mathbf{B}_d$ is the basis by which $\mathbf{D}$ can be expressed by $\mathbf{A}^d$.

$\mathbf{B}_s$ is the partitioning parallelepiped by which a computation polyhedron can be mapped within the processor array while satisfying the canonical dependencies.

C_j stands for a condition $g_j - d_j^{q'} \leq q''$ to define an area for data flows in a supernode, see g_j , q'' later, $d_j^{q'}$ is the dependency in $S^{q'}$ space.

$\mathcal{CH}^K$ stands for a convex hull.

$\mathbf{c}$ is the constant vector in the system of inequalities representing a polyhedron.

$\mathbf{c}^e = [c_0^e, \cdots, c_{N-1}^e]^T$ indicates the extra sizes of a local supernode memory space.

c is is the determinant of $\mathbf{T}$ (see below), and also the whole compression factor.

$\mathbf{D}$ is the original dependency matrix.

$DC_{\mathbf{d}^s}^{\mathbf{d}^p,d^t}$, or $DC_{\mathbf{d}^s}^{\mathbf{d}^j}$, is a data flow cube with a processor dependency $\mathbf{d}^p$, time delay d^t and supernode dependency $\mathbf{d}^s$. Note that for brevity, sometimes, d^t and $\mathbf{d}^p$ can be omitted so that DC has forms of $DC_{\mathbf{d}^s}^{\mathbf{d}^p}$ or $DC_{\mathbf{d}^s}$ where the meaning is clear.

$\mathbf{d}$ stands for a dependency vector.

$\mathbf{D}^p$ is the dependency matrix in the processor domain.

$\mathbf{d}^p$ stands for a dependency vector in the processor domain.

$\mathbf{D}^A$ is the dependency matrix in a LSGP processor domain.

$\mathbf{d}^A$ stands for a dependency vector in the LSGP processor domain.

$\mathbf{d}_{110}^s$ stands for supernode dependency vector $[1,1,0]^T$.

d^t stands for a time delay.

$\mathbf{E}$ is a Positive Expressing Basis (PEB).

$\mathbf{F}$ is the scaling matrix to scale $\mathbf{E}$ to $\mathbf{B}_s$.
$\mathbf{F}^a$ is a scaling matrix to scale $\mathbf{E}$ such that the polyhedron can be mapped into a processor array.
$\mathbf{F}_d$ is a scaling matrix to scale $\mathbf{E}$ to $\mathbf{B}_d$.
$\mathbf{f}^{K-r}$ and $\mathbf{l}^{K-r}$ are the first and the last computation nodes of polyhedron $\mathcal{P}^{K-r}$.

$\mathbf{G}$ is a diagonal matrix with the elements of $\mathbf{g}$ as its diagonal entries.
$\mathbf{g} = [\cdots, g_i, \cdots]^T$ indicates the sizes in each direction of a supernode.

$\mathbf{H}_k$ is the HNF, of $\mathbf{A}$, with only k as its diagonal entries.

$\mathbf{I}$ is the identity matrix.
$\mathbf{i}$ stands for a computation node, i.e. an iteration in a set of nested loops.
IB^{w_i} and OB^{w_i} are the input and the output buffers in a processor, respectively, carried out at the ordinal number w_i (see below).
$IP^{w_i}_{\mathbf{p}}$ and $OP^{w_i}_{\mathbf{p}}$ are the input and the output packs of communications of a processor, respectively, carried out at the ordinal number w_i (see below) along the interconnection primitive $\mathbf{p}$ (see below).

$\mathbf{J}$ is the antidiagonal identity matrix.
$\mathbf{j}$ is the index vector in processor-time domain.

$\mathbf{l} = [l_0, \cdots, l_{M-1}]^T$ is the length vector of a processor array.

$\mathbf{k} = [k_0, \cdots, k_{M-1}]^T$ is the LSGP compression vector. For (N-1)-D SBC case, $k = k_0 = \cdots = k_{M-1}$.

M is the number of dimensions of processor array.
$m_0, \cdots, m_{N-1}$ indicate the sizes of local supernode memory space.
m is the number of inequalities of the polyhedron.

N is the number of dimensions of original nested loops.
N_{links} is the number of communication links in a processor.
N_{DC} the number of DC's.
n_d is the number of dependencies.
n_v is the number of the vertices of the computational polyhedron.

K = N - M is the dimension in a lower dimensional mapping.

$\mathbf{O}$ stands for a matrix filled with "0".
$\mathbf{o}$ stands for a vector filled with "0".

$\mathbf{P}$ is a matrix representing the interconnection primitives of the architecture.

$\mathbf{P}_p$ is the permuting matrix.
p is an interconnection primitive, also a column vector of $\mathbf{P}$.
$\mathcal{P}$ is a polyhedron.

$\mathbf{Q}$ is the LSGP same-time matrix.
$Q^{\mathbf{d}^p}$ is a collection of data associated with processor dependency $\mathbf{d}^p$.
$Q_{i,j}$ is a collection of nodes associated with $\mathbf{d}_i^p$ and $\mathbf{d}_j$.
q stands for the quasi-supernode.
$\mathbf{q}'$ is the index vector in the space $S^{q'}$.
$\mathbf{q}''[\cdots, q_i'', \cdots]^T = \mathbf{q}' \bmod \mathbf{g}$.

$RB_{\mathbf{p}}$ stands for the relay buffer along the interconnection primitive p.

$S^{q'}$ is a space under basis $\mathbf{E}$.

$\mathbf{S}$ is the space projection matrix of $K \times N$.
S is the N-D original space.
$\mathcal{SP}$ is a set of polyhedra.
$\mathbf{s}^p$ is the N-D space projection vector.
s stands for a supernode.

$\mathbf{T}$ is the space-time transformation matrix.
$\mathbf{T}_b$ is the mapping matrix from the N-D time domain to the supernode domain in LSGP case.
$\mathbf{T}'$ is the lower-dimensional space-time transformation matrix.
$\overline{T}$ is a set of $\bar{\mathbf{t}}$.
$\bar{\mathbf{t}}$ is a timing vector.
t is the index vector in the N-D time domain for the LSGP case.

$\mathbf{U}$ is the image of the vertices in processor array domain mapped by $\mathbf{S}$.
$\mathbf{U}^S$ and $\mathbf{r}^S$ are the mapping matrix and the model vector for the SNF independent partitioning.
$\mathbf{U}^H$ and $\mathbf{r}^H$ are the mapping matrix and the model vector for the HSNF independent partitioning.
$\mathbf{U}^D$ and $\mathbf{r}^D$ are the mapping matrix and the model vector for the DF independent partitioning.

$\mathbf{V}$ the vertices of the computational polyhedron.
$\mathbf{V}^e$ the vertices of the enlarged supernode polyhedron.
v stands for a vertex.

$\mathbf{W}$ stands for the vertices of the polyhedron under basis $\mathbf{E}$.
$\mathbf{w}'$ indicates the location of a supernode in a processor after LSGP partitioning.
w_i indicates the ordinal number of computing each $\mathbf{w}'$.

$\mathbf{\Pi}$ is the timing projection matrix.
$\mathbf{1}$ is a vector filled only with "1".
$\mathbf{1}_i$ is a sample (or choosing) vector such that its i-th element is 1, and zero otherwise.
$\overline{\mathbf{0}}_i = 0 \ldots 0$, total 2^i "0"s in a line, $\overline{\mathbf{1}}_i = 1 \ldots 1$, total 2^i "1"s in a line.
$[x_0, y_0) \times \cdots \times [x_{N-1}, y_{N-1})$ indicates a cubic area in the supernode of dimension N.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Automatic Parallelisation

This thesis deals with the methodology and software of automatic parallelisation for numerical supercomputing and supercomputers. Many scientists and engineers have tasks which require several millions or billions of floating point operations such as computer vision [13], ocean circulation study [41], electromagnetic field study [38], VLSI circuits simulation [102], fluid and reservoir simulation [79] and various other numerical computations.

To solve computational problems quickly, these scientists and engineers use the fastest computing equipment they can find or afford. Over the last forty years, hardware technology has undergone rapid transformation. For a single processor, the processing speed has increased dramatically. It is said that the processing speed has doubled every 3-4 years in the last two or three decades. Unfortunately this increase cannot continue forever. Hardware development is achieving the physical limits of current micro-electronic technology (Unless fundamentally changing the technology, the width of lines of the latest VLSI chip is about half micro-meter, which is near the wavelengths, about a few thousand angstroms, of visible light being used for photoetching techniques. In addition, placing wires closer together also cause mutual inductance and you cannot alter the length of wires by micronization. This is why further speed-up is not possible with current technology). Thus there is a limit after which computational tasks become so heavy that no single processor can carry them out in an acceptable time. As a result, today's supercomputers have been designed to cope with enormous computing tasks. Most of supercomputers, no

matter what kind of architecture they use, typically have some kinds of array facilities. Therefore, the problem of exploiting parallelism for a particular problem and automatically generating parallel codes which can run on a supercomputer arise in the front of the users of such machines.

A basic question is, as a scientist or engineer, not specialising in parallel computing, how easy is it to write parallel codes for computing tasks in your area? Usually, the task can be achieved, but only after quite a long period, maybe a few months for a beginner, to study the theory of parallelism, the parallel language and a structure of the supercomputer. And then, to test and debug parallel codes is often an extremely difficult task, much more difficult than for corresponding sequential codes. To some extent, we can say that the difficulty and length of time required to write an efficient parallel program inhibits the actual application of supercomputing.

As a result, exploitation of automatic parallelization has been of concern for a long time before. An obvious solution is to augment compilers to generate parallelism. Indeed since the early 1970's, many researchers have attempted to exploit the possibility of automatically generating parallel codes from a sequential algorithm. A great amount of work has been done and a huge number of papers have appeared in the literature. In the early stages, people focussed on numerical problems which were oriented towards For-loops structures and discovered some basic ideas [56] [4]. From the middle of the 1980's, attention focussed primarily on the problem of data dependency and the description of recurrence equations, as well as the problem of partitioning and mapping methods [84] [88] [87] [73]. During the 1990's, more aspects of the problem have been explored and theories have matured to a usable state, especially in the case of Uniform Recurrence Equations (UREs) [93] [24] [94]. We may divide the research into the following topics for which we list only the literature which make significant contributions.

In the early days, Lamport's work [56] is the landmark. The concept of *space-time mapping* (hyperplane transformation) is still in use today. This concept is also referred to as a wavefront transformation [55]. The *data dependency* that exists in a loop nest is also important in this area. The general concepts were introduced in the 1970's (such as [4]). Quinton [84] gave a comprehensive description of the so-called URE problem, though

the concept of URE was suggested very early [50]. Rajopadhye [88] [87] discussed both URE and ARE (Affine Recurrence Equation) problems and discovered that some kinds of ARE problems can be transformed into equivalent URE problem (see the definitions of URE and ARE later). A commonly used notation for representing general dependences are "direction vectors" which are proposed in [4], [5] and [109] while [15] presented the concept of reduced dependency. [107] gave an interesting description of the wavefront transformation and data dependency from another viewpoint.

At the end of the 1970's. Padua [78] raised interest in *Independent partitioning* , which divides the computation of a loop nest into a number of independent blocks when possible. Some researchers contributed to this problem in the 1980's, and the work of Shang and Fortes [93] at the beginning of the 1990's made the latest fundamental progress for this topic.

Partitioning an algorithm to fit into a fixed-size array is essential to many applications, Moldovan and Fortes in 1986 were the early researchers to study the general partitioning problem, and proposed [73] a well-known technique, Local Parallel and Global Sequential (LPGS) partitioning. Although the research for partitioning some specific algorithms started earlier [43]. Since the 1980's, this problem has received considerable attention, and many methods have been proposed. The supernode partitioning introduced in [46] is an important idea for partitioning, while Darte's work [24] in the beginning of the 1990's marks another fundamental contribution to the theory of partitioning. Some partitioning methods come from more specific and limited view-points, eg, [81] [91]. A common case is that an array with lower-dimensions is given to carry out the computation of a set of nested multiple loops, which requires a so-called *Lower-dimensional mapping* which differs from Lamport's ordinary space-time mapping. In spite of some early methods for specific algorithms, the first important steps toward a formal solution were made in [58] and [59] in the late 1980's and the early 1990's. The theory was raised to a higher level in [94] although there still remain serious problems.

From the middle of the 1980's, based on some of the theoretical achievements, people began to try to design software packages and tools for the task of automatic parallel code generation [74] [33], although they were not very successful, further progress has made

and the prospect of building a practical software tool and to achieve real applications now exists.

1.2 Structures of Processor Array

Before continuing it is essential to give a brief review of the resources of parallel computing. So far, there are different classifications for the structures of parallel machines [28] [29] [95]. The widest-used category [29] is based on instruction and data streams, that is, Single-Instruction & Single-Data (SISD), Single-Instruction & Multiple-Data (SIMD), Multiple-Instruction & Single-Data (MISD) and Multiple-Instruction & Multiple-Data (MIMD). Obviously, SISD is a conventional sequential machine while examples of MISD are difficult to find. In SIMD, Multiple processors run under the control of a single instruction stream, while in MIMD, every processor is controlled by its own instruction streams.

In the SIMD class, a sub-classification can be given. Array Processors use a control unit to instruct a number of independent processors; Vector Processors manipulate vectors as operands instead of scalars; and Pipelined Processors operate on data as it flows through a pipeline of processing elements [97]. The MIMD class can be divided further into Multicomputer Systems and Multiprocessor Systems. The former consists of a number of autonomous computers which may or may not communicate with each other, while the latter is characterised by interaction between processors at the process, data set and data elements level.

Another classification is concerned with memory, i.e., shared-memory machines and distributed-memory machines, although strictly they fall into the Multiprocessor System class [97]. In the shared-memory machine, all processors share a global memory from which they can read and write data. The advantages are significant. Firstly there is no need of data flow between processors and a simple write or read to memory is sufficient. An obvious disadvantage is the bottleneck effect of controlling consistent access to the global memory. Many expensive hardware components, such as buses and cache memories, have been used to improve the memory bandwidth, but it is difficult to overcome the inherent shortcoming. In contrast, the distributed-memory scheme allows each processor to have its

own local memory, forming a node of an array, which allows cheap, flexible and expansible hardware, but also suffers from the disadvantage of overheads when explicit data flows between processors are required which make programming difficult.

The concept of a systolic array was introduced in [53]. Generally speaking, a pure systolic array is a regular set of interconnected cells each capable of performing some simple fixed operation upon data which flows through the array at regular beats. The array acts something like a multi-dimensional complex pipelined array. The principle is to alleviate the shared memory bottleneck by bring data from the global memory once and re-using it many times as it is pumped around the processors with low overhead communication. The concept of the systolic array has been widely applied. In general we may term a distributed-memory processor array each of whose processors performs identical operation at regular beats a "software systolic array". The attractive point of a systolic array is the identity of operations of each processors and the regularity of the operations of the whole array. A full description of structures and types of parallel machines can be found in [44].

Among all the types of parallel computing machines, we will focus on the "software systolic array". The main reasons are

1 Its hardware is cheap and easily available, less restrictions on applications.

2 Programming for it is difficult, so it needs automatic parallelisation more urgently than more general purpose systems.

3 The same technology of automatic parallelisation for the "software systolic array" can be applied directly to automatic design of a hardware systolic arrays.

Without specific requirements, we assume that the multiprocessor array takes the shape of a rectangular parallelepiped, termed a regular array. Each processor is a node of the array and is physically linked with its nearest neighbours. The processor is able to access quickly its local memory, but needs relative long time to open a port for data transference. The direction of a link from one processor to another is indicated by a so-called interconnection primitive **p**, giving rise to the following definition.

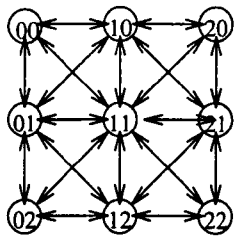


Figure 1.1: Full Bi-directional Connected Regular Array

Definition 1.2.1 *A mesh connected regular array is a tuple* $(A^M, \mathbf{P})$ *where* A^M *is a M-D rectangular parallelepiped (hypercube) of size* $l_0 \times \ldots \times l_{M-1}$ *(let* $\mathbf{l} = [l_0, \ldots, l_{M-1}]^T$*) , in which every integral point is a processor.* A^M *is referred to as the processor array.* $\mathbf{P}$ *is an integral matrix of interconnection primitives whose columns indicate the directed-links from one processor to another.*

For example, the processor array in Figure 1.1 is a 3×3 ($\mathbf{l} = [3, 3]$) regular array with interconnection primitives

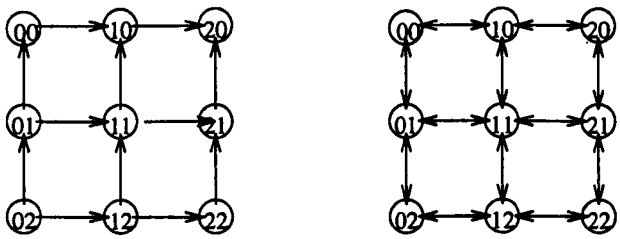
$$\mathbf{P} = [\mathbf{p}_0, \cdots, \mathbf{p}_7] = \begin{bmatrix} 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 \end{bmatrix}.$$

For instance, considering processor 11, $\mathbf{p}_0 = [1, 1]^T$ stands for the link from processor 11 to processor 20 and $\mathbf{p}_7 = [-1, 0]^T$ for the link from processor 11 to processor 12.

Two other easily available interconnection models are shown in Figure 1.2 (for example, in a 2-D array) , which have the interconnection primitives for Figure 1.2.(a), and (b) are

$$\mathbf{P}_{SUC} = [\mathbf{p}_0, \mathbf{p}_1] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{P}_{SBC} = [\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3] = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

and are termed as SUC and SBC model, respectively. Because they are the most available meshes for parallel computing machines, we build our methodology based on them in the



(a) Simple Uni-direction Connected. (SUC).

(b) Simple Bi-direction Connected. (SBC)

Figure 1.2: The Patterns of Interconnections

following chapters.

1.3 Nested Loops and Data Dependencies

Nested loop structures are the most time-consuming part in most scientific computational algorithms. It is hard to write a program which runs for a significantly long time without using loops at some point. Consequently a great deal of research has focused on exploiting the parallelism in such loop-structures. The computations of N nested loops can be considered as the body of a polyhedron in a N-D Euclidean space, usually called the computation polyhedron. Each point, or a node, in the computation polyhedron is an iteration of the loop body, and is referred to by its iteration vector, integral N-tuples, $\mathrm{i} = [i_0, \ldots, i_{N-1}]^T$. In the process of computation, each node executes the operations associated with it when its required data is available,

Iterations are not generally isolated from each other. For example, to compute one iteration, we may need some data whose results are computed by other iterations, that is, some nodes are dependent on others. Generally, the nested loops have the form [107]

Loops 1.3.1 *General For-Loops*

FOR $i_0 := \lceil max(\mathrm{l}_0^0, \mathrm{l}_0^1, \ldots) \rceil$ TO $\lfloor min(\mathrm{u}_0^0, \mathrm{u}_0^1, \ldots) \rfloor$

.....

FOR $i_{N-1} := \lceil max(\mathrm{l}_{N-1}^0, \mathrm{l}_{N-1}^1, \ldots) \rceil$ TO $\lfloor min(\mathrm{u}_{N-1}^0, \mathrm{u}_{N-1}^1, \ldots) \rfloor$

$f(\mathrm{i}) = g(f(\mathrm{i}_0), \ldots, f(\mathrm{i}_{n_d}))$

where $\mathrm{l}_i^j = \bar{\mathrm{l}}_i^j \mathrm{i} + c_i^{l,j}$ and $\bar{\mathrm{l}}_i^j = [l_{i,0}^j, \ldots, l_{i,i-1}^j, 0, \ldots, 0]$; $\mathrm{u}_i^j = \bar{\mathrm{u}}_i^j \mathrm{i} + c_i^{u,j}$ and $\bar{\mathrm{u}}_i^j = [u_{i,0}^j, \ldots, u_{i,i-1}^j, 0, \ldots, 0]$ (apparently both l_i^j and u_i^j are a linear functions, defined by $\bar{\mathrm{l}}_i^j$ or $\bar{\mathrm{u}}_i^j$, of the indices of the outer level loops). The "max" operation means that for each loop there may be a number of nominees of the lower bound and only the largest one is chosen from all the possible ones as the lower bound of the loop. The "min" has the opposite meaning.

Collecting expressions in all max(...) and min(...) in the nested loops above yield a system of inequalities, having a form

$$a_{j,0} i_0 +, \ldots, + a_{j,N-1} i_{N-1} \leq + c_j \tag{1.1}$$

for $j = 0, \ldots, m-1$, or in matrix form

$$\mathbf{A}_{m,N} \mathrm{i} \leq \mathbf{c} \tag{1.2}$$

where m is the number of the inequalities. The system of inequalities in eqn(1.2) defines an N-D polyhedron $\mathcal{P}$. A polyhedron can be also confined by a set of finite vertices. If $\overline{\mathbf{l}}_i^j$'s and $\overline{\mathbf{u}}_i^j$ are integral vectors, all vertices are integral.

Notice that $f(\mathbf{i}) = g(f(\mathbf{i}_0), \ldots, f(\mathbf{i}_{n_d-1}))$ is defined as a Recurrence Equation [87] over $\mathcal{P}$, where $\mathbf{i} \in \mathcal{P}$; $\mathbf{i}_j \in \mathcal{P}$ for $j = 0, \ldots, n_d - 1$, and g is a single valued function of $\mathbf{i}_j$'s. In general, the computational body may consist of a system of recurrence equations instead of just one, but this makes no basic changes in the concept or application of our technique. Thus for brief, we restrict ourselves to the case of a single recurrence equation.

A recurrence equation is called a Uniform Recurrence Equation (URE) iff $\forall\ j = 0, \ldots, n_d - 1$, $\mathbf{i}_j = \mathbf{i} - \mathbf{d}_j$, where $\mathbf{d}_j$'s are N-D constant vector. In a URE problem, all dependencies are straightforwardly defined by a constant distance in each dimension. Such dependencies are expressed by a dependency measure, termed the dependency vector $\mathbf{d}$, pointing from one node to another. The matrix $\mathbf{D}$ (with each column a dependency vector) describes all dependency relations among the nodes in a given computation polyhedron.

A recurrence equation is called an Affine Recurrence Equation (ARE) iff $\forall\ j = 0, \ldots, n_d - 1$, $\mathbf{i}_j = \mathbf{A}_j^f \mathbf{i} - \mathbf{d}_j$, where the $\mathbf{A}_j^f$'s are $N \times N$ constant matrices.

In ARE problems, the dependencies are implicitly expressed by linear transformations of the variable indices. If the $\mathbf{A}^f$'s are all identity matrices, the problem degenerates naturally into a URE. Alternatively, if $\det(\mathbf{A}^f) = 0$ for some dependency, we can transform it to a URE by using pipelining methods [30]. Data is pipelined along a constant vector which is the non-null solution of equation $\mathbf{A^f x} = \mathbf{o}$ ($\mathbf{o}$ is a null vector). If neither of the above cases apply, the dependency can be rewritten as $\mathbf{i}_2 = \mathbf{i}_1 + \mathbf{q}(\mathbf{i}_1)$, where $\mathbf{q}(\mathbf{i}_1)$ means a transformation. But because $\mathbf{q}(\mathbf{i}_1)$ is not a constant vector, routing [33] must be used to produce UREs instead, which needs additional mechanisms to control the flow of data in the array.

If the relation from $\mathbf{i}$ to $\mathbf{i}_j$'s does not fall into the URE or ARE category, it is defined quite generally as $\mathbf{i}_j = m_j(\mathbf{i})$, where m_j stands for a general mapping from an N-D integral space to another N-D integral space. Sometimes, the dependency relations can not be expressed by vectors, but by some symbols representing dependency directions [109]. The distances are not well-defined.

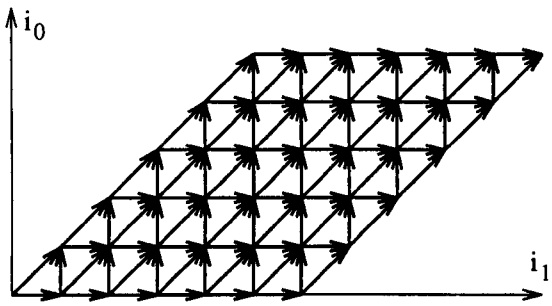


Figure 1.3: A Computational Graph

We restrict ourselves to the UREs and AREs which can be transformed into UREs systematically. This is not only because the URE is the simplest case to study in this field, but in fact, URE covers a wide range of numerical computational problems, from the simple cases of matrix multiplications, AR and ARMA processes to the more complex Knapsack Problem [71]. Indeed, many complex numerical problems are expressed as iterational processes where the computation of an iteration relies on the previous iterations with fixed access distances. This mathematical background gives a sound nature to URE for applications. Furthermore, the research on the case of URE will pave a road to more complex situations. Some members of the Algorithm Engineering Research Group (AERG) in Newcastle University are working on transforming more general recurrences to URE as mentioned above which can make a direct use of the techniques being derived for URE and on uniform mappings and routings for general ARE problems.

As regards the more complex dependency representation which may be involved more in logic reasoning problems, unfortunately, the theory of space-time scheduling is currently far beyond the application stage and is not be considered in our project. For URE, we define the computational graph as follows

Definition 1.3.1 *A Computational Graph is defined as a tuple(*$\mathbf{V}$*,*$\mathbf{D}$*), where* $\mathbf{V}$ *is a set of finite vertices* $\in Z^N$*, which defines a computation polyhedron and* $\mathbf{D}$ *is the dependency matrix. The computation graph is acyclic.*

In fact, here $\mathbf{V}$ is a matrix of $n \times n_v$, n_v is the number of the vertices. $\mathbf{D}$ is a $N \times n_d$ matrix, n_d is the number of dependency vectors.

For example, the computational graph of the following simple For loop program.

FOR $i_0 := 0$ TO 5
  FOR $i_1 := i_0$ TO $i_0 + 6$
    $A(i_0, i_1) := 3A(i_0, i_1 - 1) + A(i_0 - 1, i_1 - 1) \times A(i_0 - 1, i_1)$[1]

is illustrated in Figure 1.3. From the figure, we can collect the vertices of the graph and the dependencies as (henceforth assume that the outmost loops are listed first in a column dependency vector)

$$\mathbf{V} = \begin{bmatrix} 0 & 0 & 5 & 5 \\ 0 & 6 & 5 & 10 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Usually, extracting the dependencies of an URE is easy. However, it is sometimes quite difficult to determine all the vertices of a polyhedron defined by a set of inequalities, a classic problem in computational geometry. It is not a difficult problem in principle. In fact, a vertex $\mathbf{v}$ is the solution of N linearly independent equations from the system $\mathbf{A}_{m,N}\mathbf{x} = \mathbf{c}$ and satisfies the condition $\mathbf{A}_{m,N}\mathbf{v} \leq \mathbf{c}$ [90]. However, because $\mathbf{A}_{m,N}$ contains potentially C_m^N sets of linearly independent equations, we have to solve the C_m^N sets of linear equations; and then test the potential vertices by $\mathbf{A}_{m,N}\mathbf{v} \leq \mathbf{c}$ to obtain the final results. When $m \gg N$, this is a time-comsuming work. We do not consider this problem explicitly in the thesis although the reader is referred to [90] for necessary mathematics.

1.4 Space-Time Mapping

As early as 1974, Lamport [56] introduced the idea of a linear mapping of a computational graph onto a (N-1)-D space (or processor array) domain and 1-D time domain by a space-time transformation defined by an integral matrix $\mathbf{T}$

$$\mathbf{T} = \begin{bmatrix} \mathbf{S} \\ \bar{\mathbf{t}} \end{bmatrix} \tag{1.3}$$

where $\bar{\mathbf{t}} = [t_0, \ldots, t_{N-1}]$ is the timing, or scheduling, vector, and

$$\mathbf{S} = \begin{bmatrix} \bar{\mathbf{s}}_0 \\ \vdots \\ \bar{\mathbf{s}}_{N-1} \end{bmatrix} = \begin{bmatrix} s_{0,0} & \cdots & s_{0,N-1} \\ \vdots & \vdots & \vdots \\ s_{N-2,0} & \cdots & s_{N-2,N-1} \end{bmatrix} \tag{1.4}$$

[1] If the addressed element of the array A is out of the index polyhedron, it is zero

is the space mapping, a $(N-1) \times N$ matrix. Since then this concept has being adopted by many researchers, e.g., [84], [73], [109], [81].

In this method, $\bar{\mathbf{t}}$, is used to form a number of parallel hyperplanes in the Euclidean space. All the nodes on one hyperplane are independent of each other and, can, therefore, be computed concurrently. **S** is used to map each node to a processor in an (N-1)-D array. The mapping of a polyhedron onto a N-1 space by pre-multiplying **S** can be thought of as a projection of the polyhedron along a space projection vector $\mathbf{s}^p \in Z^N$ which is perpendicular to every row vector of the space mapping, i.e., $\mathbf{S}\mathbf{s}^p = \mathbf{o}$.

It is well-known that $\bar{\mathbf{t}}$ and $\mathbf{s}^p$ must satisfy the two conditions:

$$\bar{\mathbf{t}}\mathbf{d}_i > 0, \quad \forall i, 0 \leq i < n_d \quad \text{or} \quad \bar{\mathbf{t}}\mathbf{D} > \bar{\mathbf{o}} \quad \text{(dependency constraint)} \tag{1.5}$$

$$c = \bar{\mathbf{t}}\mathbf{s}^p > 0 \quad \text{(no conflict in a processor)} \tag{1.6}$$

where $1/c$ is the computational efficiency of the mapping; $\mathbf{d}_i$ are the dependency vectors in **D**. For the first condition, notice that if, for instance, $\mathbf{i}_2 = \mathbf{i}_1 + \mathbf{d}_i$, we have $\bar{\mathbf{t}}\mathbf{i}_2 = \bar{\mathbf{t}}\mathbf{i}_1 + \bar{\mathbf{t}}\mathbf{d}_i$. This indicates that the execution time of $\mathbf{i}_2$ is behind that of $\mathbf{i}_1$ if $\bar{\mathbf{t}}\mathbf{d}_i > 0$, which satisfies the neccesary conditions of $\mathbf{i}_2$ depending on $\mathbf{i}_1$. For the second condition, notice that $\mathbf{i}_k = \mathbf{i}_1 + k\mathbf{s}^p$ for all $k \in Z$ will be mapped to the same processor by **S** due to $\mathbf{S}\mathbf{s}^p = \mathbf{o}$, so any two of them must not be executed at the same time. Because $\mathbf{i}_k$ is executed at time $\bar{\mathbf{t}}\mathbf{i}_k = \bar{\mathbf{t}}\mathbf{i}_1 + k\bar{\mathbf{t}}\mathbf{s}^p = \bar{\mathbf{t}}\mathbf{i}_1 + kc$, all the $\mathbf{i}_k$'s are guaranteed to be executed at different times by eqn(1.6).

More generally, we can use

$$\mathbf{T} = \begin{bmatrix} \mathbf{S} \\ \mathbf{\Pi} \end{bmatrix} \tag{1.7}$$

to map the N-D polyhedron onto a M-D space domain and a K-D time domain [104]. Here, **S** is a $M \times N$ matrix and $\mathbf{\Pi}$ is a $K \times N$ timing projecting matrix (K = N - M).

It should be pointed that there is a constraint for dependency vectors. In fact in a sequential loop nest, the nodes are executed in lexicographical order; thus dependences extracted from such nested loops nest must be "lexicographical positive" [107].

Definition 1.4.1 *A vector* **d** *is lexicographically positive, written* $\mathbf{d} \succ 0$, *if* $\exists i$: *(*$d_i > 0$ *and* $\forall j < i$: $d_j \geq 0$*), where* d_i *is the i-th component of* **d**.

That is to say there is at least one positive component before any negative component of the vector. There must also be at least one positive component.

1.5 About the Thesis

A research project Systolic And Regular Array Computation Environments (SARACEN), a high level CAD tool for parallel code generation, began in University of Newcastle during 1990, see Figure 1.4. The main procedure is summarised as:

Step 1 Semantic analysis of a high-level descriptions of a computational algorithm and extraction of the computational graph.

Step 2 Partitioning and mapping the computational graph onto a given regular array.

Step 3 Determination of the structure, boundaries and data flows of the parallel program, and generation of parallel codes

Work on the SARACEN system is divided into self-contained themes allowing work to progress in each area simultaneously . Data is exchanged between sections of the software via a common intermediate format. Consequently in this thesis we can assume that our input will be a system of uniform recurrence equations. The problem of transforming non-uniform systems of equations to uniform ones is not addressed in this work. Indeed, it is nontrival and is the subject of a joint project with Manchester University (EPSRC REFLEX) and the interested reader is referenced to the thesis [89] by L.Rapanotti which discusses transformational issues in detail (Step 1 above).

This text is based on my own work as part of AERG which works on the area of automatic parallelization. In particular, and for a limited class of recurrences, this thesis proposes a theory of mapping algorithms to parallel architectures and develops software which automatically generates parallel programs.

Step 2 of the project is concerned with the major theoretical work of this thesis. Although the concept of space-time mapping was introduced very early, it is still quite an open topic especially in the area of mapping onto a given regular processor array, particularly where constraints on connections are concerned. Traditionally the algorithm mapping is performed with the aid of a design tool (e.g., a visualization system).

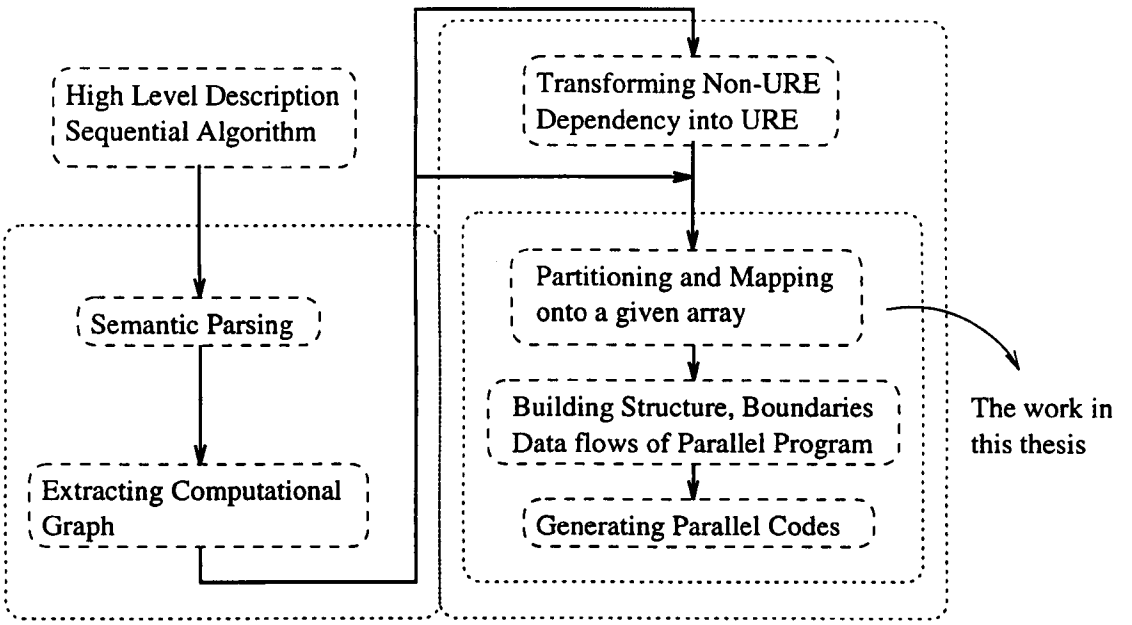


Figure 1.4: Chart of the SARACEN Project

The rest of the thesis is organizied as follows. Chapter 2, Chapter 3, Chapter 4 and Chapter 5 consider the mapping problem. Chapter 2 gives a survey and analysis to the partitioning and mapping problem; Chapter 3 presents the improvements for some existing partitioning techniques; Chapter 4 proposes our basic methodology of partitioning and the mapping onto fixed-sized (N-1)-D processor arrays; Chapter 5 deals with the lower-dimensional mapping problem, i.e., mapping onto M-D array while M < N-1. Step 3 is quite a new research area, since, so far, few researchers have managed to convert theory into practice effectively and some aspects have hardly been addressed at all. Chapter 6 considers the problems of determining the structure, boundaries and data flows of the parallel program, given the theory of previous chapters. In Chapter 7, we present automatically generated parallel codes which can run correctly and efficiently on given arrays. The experimental results and some discussions can be found in Chapter 8, while Chapter 9 provides a summary, evaluation, and comparison for our methodology. Finally, some samples of the automatically generated parallel codes are presented in Appendix G. Some of the algorithms are also collected into other appendices.

Chapter 2

Survey and Analysis of Partitioning and Mapping

In this chapter, a survey is made of the current methods of partitioning and mapping for parallelisation. A number of algorithms of independent partitioning are analysed and evaluated. For the general partitioning and mapping problem, a classification is proposed. Under the classification, analysis is made to exploit the advantages of two classes of partitioning and mapping methods, as well as to expose their weaknesses. Such criteria are helpful in developing better methodologies for partitioning and mapping. A brief survey and analysis is also given for the lower-dimensional mapping problem.

2.1 The Problems

To speed up program computation, it is hoped to compute all nodes in the computation polyhedron simultaneously. Unfortunately such a requirement is generally impossible, because firstly the results of some nodes must be calculated before others, and secondly, we usually do not have enough processors to match the number of nodes in the computation polyhedron. In partitioning, and as we will see later, in some situations the computation polyhedron can be separated into a number of independent subsets, where no data transference is needed between the subsets. Consequently the computations associated with the subsets can be assigned to individual processors, and carried out in parallel.

In many cases, such an independent partitioning does not arise naturally, and the well-known technique of space-time mapping is applied to expose and exploit the parallelism

in the computation graph. By means of both the time mapping and space mapping, each node in the polyhedron is assigned to a distinct processor at a distinct moment. That is, *a linear injective transformation* from the original computational space onto a processor-time space. However, such a transformation only works well for applications involving regular computations and where the number of processors is dependent on the loop-bounds of the original problem.

There are three problems to be concerned with: (1) whether the transformed polyhedron in the processor-time space can be fitted into a given-size actual processor array; (2) whether the resulting dependencies can be implemented by the given interconnection primitives of the actual array; (3) whether the data transference of any communication is carried out efficiently. A partitioning procedure is necessary to solve these three problems. A naive and simple approach is to group a number of nodes together and assign them to one processor, the volume of the transformed polyhedron is obviously reduced to fit the size of the physical array and much of the inter-processor data transference is removed, as is the accompanying overheads for communication. However a good strategy of partitioning should also be helpful in the efficiency of mapping data dependencies onto a given interconnection pattern.

Finally, in addition to the three problems above, there is one key point left: whether the N-D computational task can be carried out with a given lower-dimension processor array. In the well-known space-time scheduling concept, the time domain is a 1-D space, leaving (N-1)-dimensions to the space domain. In many cases, especially when N is large, the available processor array has only M-dimensions where M is less than N-1. Squashing the polyhedron into the correct number of dimensions poses serious difficulties that we address later.

2.2 Independent Partitioning

The best and simplest way of parallelizing nested loops is to partition them into several independent partitions, each of which can be completed in one processor. In this way, apart from the task loading, there is no need for inter-processor communications.

Independent partitioning is possible under two circumstances. Firstly, if r = rank(**D**)

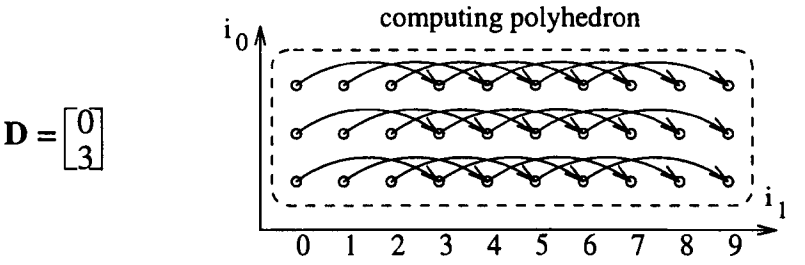


Figure 2.1: Two Natures of Independent Partitioning

is less than N , because r is the number of a linearly independent vectors in $\mathbf{D}$, there are not enough linearly-independent dependency vectors in the available directions to connect all nodes in the space together as a whole, therefore the space spanned by these vectors is only a subspace of the original one. Each of the subspace can be grouped into an independent partitioning. Secondly, if the distances associated with the dependency vectors are so "long" that they jump over the neighbouring nodes to some node far away, thus nodes have no dependent relationship with their neighbours. Therefore we can further make jumping partitionings each of which consists of only the nodes connected by "long" dependency vectors.

Consider Figure 2.1. At first, because $\det(\mathbf{D}) = 1$, we can see that there is no dependency in the i_0 dimension, therefore each row of the polyhedron can be divided into an independent partitioning. Secondly, it can be seen that in each row the nodes 0, 3, 6, 9 have a dependent relationship but are independent to other nodes, thus they can be further assigned into a group. The same fact holds also for the nodes 1, 4, 7 and the nodes 2, 5, 8. Thus we have 3 independent partitionings in each row.

Several methods for independent partitioning have been established. The method of coarsest granularity, proposed in [107], is available only for the first case, i.e., $r = rank(\mathbf{D}) < N$. First we find N-r vectors $\overline{\mathrm{v}}$ such that $\overline{\mathrm{v}}\mathbf{D} = \overline{\mathrm{o}}$, then span a unimodular matrix $\mathbf{T}$ with the $\overline{\mathrm{v}}$'s as its first N-r rows, see [107]. $\mathbf{T}$ is the transformation matrix. It is proved [107] that the entries in the first N-r rows of the transformed dependency matrix, $\mathbf{TD}$, are zero. This means that there is no dependency in the transformed N-r outmost loops, so they can be carried out as DOALL loops. For instance, for a given set of nested loops with one statement $\mathrm{A}(i,j) = \mathrm{A}(i-2, j+3) - \mathrm{A}(i-4, j+6)$, and dependency

matrix $\mathbf{D} = \begin{bmatrix} 2 & 4 \\ -3 & -6 \end{bmatrix}$, where $\det(\mathbf{D}) = 0$. We can define the unimodular matrix $\mathbf{T} = \begin{bmatrix} 3 & 2 \\ -1 & -1 \end{bmatrix}$ and let $\mathbf{i}' = \mathbf{Ti}$, then $\mathbf{D}' = \mathbf{TD} = \begin{bmatrix} 0 & 0 \\ 2 & 1 \end{bmatrix}$. The original statement can be transformed to $\mathbf{A}$(i',j') = $\mathbf{A}$(i',j'-2) - $\mathbf{A}$(i',j'-1), where there is no dependency with respect to the first index, allowing it to be carried out with DOALL.

The Greatest Common Divisor method, [78], employs the greatest common divisor or GCD of every row of $\mathbf{D}$ to build a diagonal matrix as the partitioning matrix $\mathbf{D}'$. In many cases this method does not achieve the maximal partitioning (but we should not be over-critical as it is now quite an old method). For example, for a given set of nested loops with one statement $\mathbf{A}$(i,j) = $\mathbf{A}$(i-3,j-8) - $\mathbf{A}$(i-5,j-18), and dependency matrix $\mathbf{D}$, the corresponding partitioning matrix $\mathbf{D}'$ is

$$\mathbf{D} = \begin{bmatrix} 3 & 5 \\ 8 & 18 \end{bmatrix} \longrightarrow \mathbf{B}' = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

indicating that only two independent sets can be partitioned with $\mathbf{B}'$. In fact, notice that there is "1" in the 0-th row of $\mathbf{B}'$, this means that the independent partitioning cannot be achieved along the 0-th dimension. But since the only non-zero element in the 1st row of $\mathbf{B}'$ is "2", we can divide the whole space into two independent sets: $S_0 = \{\mathbf{i} : i_0 \in Z$ and $i_1 \bmod 2 = 0\}$ and $S_1 = \{\mathbf{i} : i_0 \in Z$ and $i_1 \bmod 2 = 1\}$. It can be checked that S_0 and S_1 are self-contained with respect to any additions of the columns of $\mathbf{D}$. This means that S_0 and S_1 are independent of each other.

The Minimum Distance method introduced in [81] came from a simple thought that if there is a dependent relationship between two nodes $\mathbf{i}^1$ and $\mathbf{i}^2$ there must be a integral solution $\mathbf{a}$ for the equation $\mathbf{i}^1 = \mathbf{i}^2 + \mathbf{Da}$. However, usually it is quite difficult to find an integral solution for such an equation, so the $\mathbf{D}$ is transformed to an upper triangular matrix $\mathbf{D}^t$ by means of a linear mapping $\mathbf{K}$ which is an integer matrix, that is, $\mathbf{D}^t = \mathbf{DK}$. The independent partitioning begins from finding a set of initial points $\mathbf{i}_{i0}$'s, where $i = 0, \cdots, det(\mathbf{D}^t) - 1$. Then for node $\mathbf{i}$ if there is an integral solution $\mathbf{a}$ for equation $\mathbf{i} = \mathbf{i}_{i0} + \mathbf{D}^t\mathbf{a}$, the node $\mathbf{i}$ is grouped into the partitioning i. It was claimed that this method can achieve the maximal independent partitioning. Unfortunately the disadvantages of this method cannot be ignored. It is not very clear how to determine the set of initial points. Furthermore, to find an integral solution of $\mathbf{i} = \mathbf{i}_{i0} + \mathbf{D}^t\mathbf{a}$ for every node is still

very costly procedure, even if $\mathbf{D}^t$ is triangular matrix.

In [93] the Partitioning Vector Method (PVM) is proposed. Unfortunately, this method can not guarantee the maximal partitioning. In the same paper, the Smith Normal Form (SNF) method was also derived. In the SNF method the maximal partitioning is guaranteed. Any matrix can be transformed to its unique SNF, that is, $\mathbf{D} = \mathbf{U}^u\mathbf{D}^d\mathbf{V}^u$, where $\mathbf{U}^u$ and $\mathbf{V}^u$ are unimodular matrices, and $\mathbf{D}^d$ is a diagonal matrix $diag(d_0, d_1, \ldots, d_{r-1}, 0, ...)$ such that $d_0 \mid d_1 \mid .. \mid d_{r-1}$. Proceeding similarly to PVM, first make the N-r outmost loops DOALL loops. Second, partition the inner blocks with r pairs of $\overline{\mathbf{p}}_i$ and a_i, where $\overline{\mathbf{p}}_i$ is the i-th row vector of $\mathbf{U}$, and a_i is d_i. That is, group the nodes $\mathbf{i}_1$ and $\mathbf{i}_2$ ($\mathbf{i}_1$ and $\mathbf{i}_2 \in \mathbf{J}$) together if $\forall i\ \overline{\mathbf{p}}_i \times \mathbf{i}_1 \mod a_i = \overline{\mathbf{p}}_i \times \mathbf{i}_2 \mod a_i$. See reference for proof. More intuitively, we point out that because this method makes the partitioning in r different directions (the maximum number), there is no further partitioning possible. Consequently very little room is left for improvement.

However, the SNF is an overly conservative condition for the maximal partitioning problem, because the condition $d_0 \mid d_1 \mid .. \mid d_{r-1}$ is unnecessary. In fact, a diagonalising matrix is sufficient, at least in principle. In addition, it is inconvenient to collect nodes one by one using $\overline{\mathbf{p}} \times \mathbf{i} \pmod{a}$, because a computation for each node in the whole polyhedron is required. Also the sequential relations among the nodes grouped into a block may become unclear. A method is required to perform the transformation of the nested loops such that all parallelism exploited can be expressed by a loop nest consisting of a number of DOALL loops.

2.3 General Partitioning and Projecting For Space-Time Mapping

If the computation polyhedron can not to be divided into several independent blocks, or the number of such blocks is much less than the available processors, a space-time mapping has to be used to exploit parallelism in the computation. However, partitioning still plays a key role.

Usually it is hoped that, by use of partitioning, the whole polyhedron can be allocated within a physical array, and the inter-processor communications can be reduced. The

time consumed by communication consists of two parts. The first part is the time for actual data flows, which is reduced by changing a remote communication to a local one inside one partition allocated to one processor. The second part is the time for overhead operations to prepare a communication, such as setting parameters of the data flows to the interface of a link. In some current processors (such as the transputer [45]), this overhead is significant. By partitioning, the data output from a processor can be sent out in one lump, so only one overhead operation is needed for the data transference.

Unfortunately, so far, there is no clear classification with regard to the methods of general partitioning. Partitioning is closely related to projection ,where "projection" means mapping a N-D space onto M-D space along a projection direction ($M < N$). We avoid using the term "mapping" deliberately, because it is often used to express the allocation of nodes onto a processor array. Sometimes, projecting is equivalent to mapping if the M-D space is the processor space. However the relation between partitioning and projecting can be taken as a criterion for classification. One category is projecting ahead of partitioning, such as LPGS (Locally Parallel Globally Sequential) and LSGP (Locally Sequential Globally Parallel). The second is the reverse – partitioning ahead of projecting, for example, tiling [108].

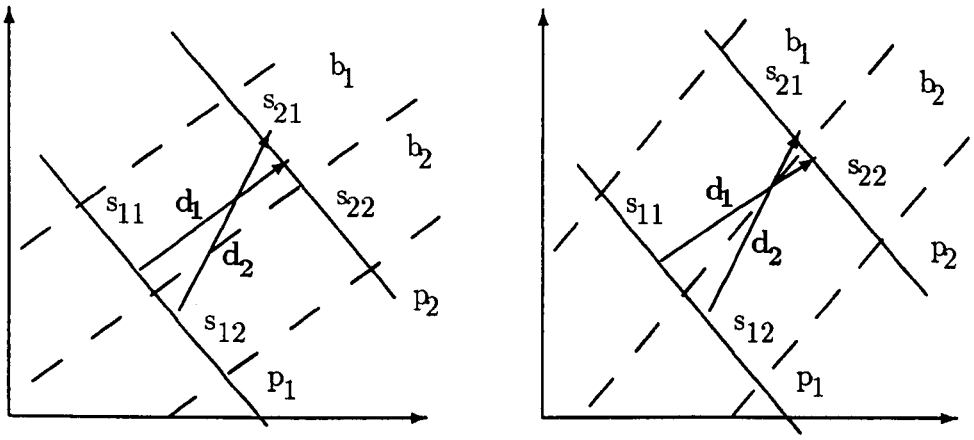
2.3.1 Projecting-Partitioning Methods

The first class of partitioning-projecting methods takes mainly two forms: LPGS and LSGP, so we discuss them in detail.

LPGS Method

The method proposed in [73] is the typical form of LPGS. Here the objective is to perform partitioning and mapping so that the computation polyhedron can be allocated into a fixed-sized array. For simplicity, we take a 3-D computation polyhedron (i.e. a nest of 3 loops) as example henceforth. In this method,

$$\mathbf{T} = \begin{bmatrix} \bar{s}_0 \\ \bar{s}_1 \\ \bar{t} \end{bmatrix}$$



(a) Bands Parallel to One Dependency Vector.

(b) Counter Data Flow between Bands.

Figure 2.2: Partitioning of Moldovan's Method. p_1 and p_2 are the hyperplanes, b_1 and b_2 are the bands, s_{11},...,s_{22} are the segments formed by the hyperplanes and the bands. $\mathbf{d}_1$ and $\mathbf{d}_2$ are dependency vectors.

is a transformation matrix, where $\bar{t}$ is the timing vector, $\bar{s}_0$ and $\bar{s}_1$ are space mapping functions. For any node $\mathbf{i} \in \mathbf{J}$, if

$$n_0 = \bar{s}_0 \times \mathbf{i} \pmod{N_0}, n_1 = \bar{s}_1 \times \mathbf{i} \pmod{N_1},$$

node $\mathbf{i}$ will be computed at processor (n_0, n_1) at time $\bar{t} \times \mathbf{i}$. Obviously N_0 and N_1 can be taken as the sizes of the given regular processor array. However, by this kind of projecting and partitioning, a single time hyperplane can have more than one node mapped to the same processor. In other words, conflicts take place and nodes compete for the right to evaluate themselves! To avoid this problem, each time hyperplane is broken up into a number of segments. The computation polyhedron is divided into a number of bands which cross a time hyperplane to form the segments such that the nodes in one processor belong to different segments of different time hyperplanes. On execution of the computation, we complete each segment, one by one (hence the name globally sequential), on one time hyperplane, then do the same to the next hyperplane, so conflicts are avoided, see Figure 2.2.(a).

An important contribution of [73] was a criterion for selecting the time vector and the space mapping function according to the dependencies and the available physical links $\mathbf{P}$.

The criterion can be summarised by the equation.

$$\mathbf{SD} = \mathbf{PK} \tag{2.1}$$

where $\mathbf{D}$ is a dependency matrix, $\mathbf{K}$ is an integral matrix such that $\forall k_{ji} \in \mathbf{K}$, $k_{ji} \geq 0$ and $\Sigma_j k_{ji} \leq \bar{\mathbf{t}} \times \mathbf{d}_i$. Notice that the $\mathbf{SD}$ represents the projected dependencies which must be implemented by a positive combination of the column vectors of $\mathbf{P}$, and that the time for implementing a projected dependency $\mathbf{S} \times \mathbf{d}_j$ by such a combination must be within the time when it is needed, that is $\bar{\mathbf{t}} \times \mathbf{d}_i$, $\mathbf{d}_i \in \mathbf{D}$.

The method has two advantages. Firstly it makes great use of the processor resource, because on one time hyperplane, every processor is overloaded, and hence busy most of the time. Secondly, on one time hyperplane there is no need for communications. So only after a processor completes all of its assigned nodes on one hyperplane, does it need to output some intermediate results to its neighbours. As a result, the output data can be sent out as one large package.

However, the method also suffers from a number of disadvantages. Because the computational polyhedron is divided into bands and these bands are mapped onto a processor array in a way that ensures the left-hand sides of all the bands are mapped to one side of the array while the right-hand sides to the other side, the problem of data transference from the right-hand side of a band to the left-hand side of the band on its right arises. Consequently, a set of First-In-First-Out buffers and long-distance links from one side to the other side are needed along each edge of the processor array. Obviously, such additional hardware is not always available and restricts the applications of the method.

In addition, when the given regular processor array offers only the simple interconnection primitives $\mathbf{P}$ which have no negative entries, we may face serious difficulties in choosing $\mathbf{S}$. Considering eqn (2.1), because there are no negative elements in $\mathbf{PK}$, we have to look for a $\mathbf{S}$ such that all the elements of $\mathbf{SD}$ must be non-negative for an arbitrary $\mathbf{D}$ which may consists of negative entries. Since the searching takes place in an $N \times (N-1)$-D integral space, it involves a computationally expensive procedure. Furthermore, even if such an $\mathbf{S}$ is obtained, the timing vector $\bar{\mathbf{t}}$ selected according to $\mathbf{SD} = \mathbf{PK}$ cannot guarantee $\det(\mathbf{T}) = c = 1$. Usually, we may have $c > 1$, even $\gg 1$, i.e., very poor efficiency, because efficiency is less than or equal to $\frac{1}{c}$. In fact, only one of c processors is active at

a moment, leaving the remaining $c-1$ processors idle. These idle times correspond to "invalid holes" in the processor-time domain. Trivially, one may suggest looking directly for a $\mathbf{T}$ under condition of $\mathbf{SD} = \mathbf{PK}$ and $\det(\mathbf{T}) = 1$. Then, the search space is enlarged to $N \times N$ dimensions. There is no way to show the existence of such a solution for general situations. Therefore, there is no guarantee for the performance of the LPGS method.

Finally we remark that in [25] this method was said to be limited to the case when there is no counter data flow between bands. This limitation may not be true for Moldovan's method itself, because it is the segments, not the bands, which are computed sequentially, there is no restriction on data flow between the bands. It can be seen from Figure.1.(b) that there is counter data flow, because of $\mathbf{d}_1$ pointing form b_1 to b_2, and $\mathbf{d}_2$ pointing inversely. However, the computations of segments are carried out in the order s_{11}, s_{12}, s_{21} and s_{22} there is no conflict with respect to dependency relations. In fact, s_{11} and s_{12} are on the same timing hyperplane p_1, so there is no dependency relation between them; and the same holds for s_{21} and s_{22}. Also when evaluating s_{21} and s_{22}, the data needed by them are ready because p_2 is performed after p_1. However, if the condition of no counter data flows between bands is imposed on Moldovan's method, every dependency vector must be parallel to the bands. For this purpose, $\mathbf{d}_2$ must not exist, and the bands have to be modified so that they are parallel to $\mathbf{d}_1$, as in Figure.1.(a). The positive result of this is that there is no longer the need of additional hardware. Therefore, only in this sense, the restriction of no counter data flows is sensible. However, this case exists only if $\det(\mathbf{D}) = 0$, which means that the partitioning degenerates to an independent one.

Other methods also exist, in particular, [76], [40] and [48] can be considered as the extensions of the above method.

LSGP Method

Darte's method [24] is a good example of the LSGP subclass, which marks an important theoretical advance. Two important concepts introduced in this method are a "same-time" matrix $\mathbf{Q}$ and an activity matrix $\mathbf{A}$. $\mathbf{Q}$, which is perpendicular to $\bar{\mathbf{t}}$, spans a (N-1)-D subspace where all nodes are executed simultaneously, while $\mathbf{A} = \mathbf{SQ}$ spans a subspace, in the cell space, where all cell are active simultaneously.

The method can be presented as a procedure of partitioning and mapping onto fixed-size arrays. Since the LSGP approach will be used later in our method, we briefly describe it as follows:

Step 1 Given $\mathbf{s}^p$ and c, where $\mathbf{s}^p$ is perpendicular to the $\mathbf{S}$, and c is an overall compression factor (from computational polyhedron to processor array)

Step 2 Determine $\bar{\mathbf{t}}$ by $c = \bar{\mathbf{t}}\mathbf{s}^p$

Step 3 Derive $\mathbf{S}$ from $\mathbf{s}^p$, $\mathbf{Q}$ from $\bar{\mathbf{t}}$.

Step 4 Compute the Hermite Normal Form (HNF) of $\mathbf{A} = \mathbf{SQ}$ and form a partitioning box of size $k_0, \ldots, k_{N-2}$, where $k_0, \ldots, k_{N-2}$ are the diagonal elements of the HNF , and c $= \prod_i k_i$. The quantities k_i indicate the compression ratio in the i-th dimension. We refer to $\mathbf{k} = [k_0, \ldots, k_{N-2}]^T$ as the compression vector indicating the compression factors in each dimension.

Step 5 Project the computation polyhedron as a virtual array by $\mathbf{S}$ along projecting vector s.

Step 6 Draw boxes on the virtual array with edges parallel to the axis and of size $k_0 \times \cdots \times k_{N-2}$. The nodes in one box are active at the different times. Hence, they can be clustered and mapped into one processor, so that no conflicts occur.

The process is illustrated as follows:

$$c \begin{array}{l} \mathbf{s}^p \Longrightarrow \mathbf{S} \\ \Downarrow \\ \bar{\mathbf{t}} \Longrightarrow \mathbf{Q} \end{array} \Longrightarrow \mathbf{A} \overset{HNF}{\Longrightarrow} k_0, \ldots, k_{N-2}$$

This diagram shows simply that the compression factor c is decomposited to $\mathbf{s}^p$ and $\bar{\mathbf{t}}$ which generate $\mathbf{S}$ and $\mathbf{Q}$, respectively; $\mathbf{A} = \mathbf{SQ}$ is transformed to its HNF whose diagonal entries determine the sizes of conflict-free partitioning box.

Darte's method is very clever, but it still has some disadvantages. The first difficulty is with the selection of $\mathbf{S}$ such that the computation polyhedron can be mapped onto an actual array of not only fixed-size but fixed-size in each dimensions. For instance, if we have an processor array of size $N_0 \times N_1$. We may be able to find an $\mathbf{S}$ such that

the polyhedron can be projected as a virtual array of size $(n_0 \times N_0) \times (n_1 \times N_1)$, so the compression factor c should be $n_0 \times n_1$. However, the activity matrix **A** resulting from **S** and **Q** can be transformed to its HNF with 1 and $n_0 \times n_1$, or n_0 and n_1, as the diagonals. If n_0 and n_1 are the diagonal elements, partitioning with a parallelepiped of $n_0 \times n_1$ can map the virtual array onto the actual array. If this is not possible, partitioning with a parallelepiped of $1 \times (n_0 \times n_1)$ fails to give a suitable mapping, i.e., compression is too strong in one dimension, but too weak in the second dimension. We can check that modification of **Q** has no effect on the HNF of **A**, but that modifying **S** does. However, if **S** has to be changed for a different HNF while keeping c unchanged, the virtual array itself is changed. As a result, the previous compression factor c is no longer valid, so everything has to be re-evaluated. Essentially the technique is a high-dimensional searching procedure in which multiple-target functions are indirectly modified to fit their N-1 targets, which is a difficult job.

Furthermore, the LSGP method does not consider the problem of redefined data communications. Because both **S** and $\bar{t}$ are determined according to the requirement of mapping onto a given regular array, no flexibility is left for satisfying $\mathbf{SD} = \mathbf{PK}$. No proof can be predicted for the existence of a satisfactory pair of **S** and $\bar{t}$ with so many conditions for arbitrary dependencies and simple interconnection primitives.

The final shortcoming of the method lies in a fact that the timing layers (hyperplanes) are too thin and no further segmenting can be made on a layer so that data produced in one layer may be needed immediately by the nodes on the next layer and in another box. Therefore, there is no time left to collect the data belonging to a number of sequential nodes in one processor as a package communication. This is similar to the case of no partitioning at all.

In [111] almost the same result as Darte's is obtained from a different aim and by a different mathematical framework. They began with the efficiency problem of a space-time mapping. As before, let the transformation matrix $\mathbf{T} = [\bar{t}^T, \mathbf{S}^T]^T$. It was found that if $\det(\mathbf{T}) = c' > 1$, on projecting a polyhedron as a virtual array, only one in every c' cells of the virtual array is active at a moment (one time hyperplane), and the $c' - 1$ others are idle. A great waste of processor resources. To improve efficiency, c' cells can be clustered

into one processor so that the processor keeps busy all time. This is a well-known fact. However, the contribution is to give a clustering strategy such that in a clustered block there is no conflict. It was found that no more than one cell is active at a moment if the partitioning is made by a parallelepiped with edges whose lengths are the factors of c'. In fact, this c' is exactly the compression factor c in Darte's method. Thus the method is the same as Darte's one in nature, and shares the same advantages, as well as disadvantages.

In these two methods, LSGP and LPGS, projecting is done before partitioning. That is, firstly, map the polyhedron on a (N-1)-D hyperplane, then partition the virtual array, or projected nodes, to blocks to be allocated to processors. The main difference in their partitionings lies in the fact that the LSGP method collects the projected nodes in a non-continuous way, "jumping" with a constant step; while in Darte's approach partitions of the virtual array occur in a continuous piecewise manner.

Other similar methds

A method was proposed in [96], similar to the ones above in the sense of partitioning the projected nodes on a hyperplane. At first, take the timing vector also as the projection vector s. Project the computation polyhedron onto a hyperplane perpendicular to $\overline{\mathbf{t}}$, and do the same for the dependencies. Among the projected dependencies, select the shortest one as a grouping vector g and another (for the case of 3-D polyhedron) as auxiliary grouping vector a. Let k be an integer such that $k \times \mathbf{g}$ is the shortest integral vector. In partitioning, k projected nodes along g are grouped into one block, and the first projected node, say $\mathbf{i}_s$, in the block is taken as a "seed", to determine other seeds for other blocks. That is, $l \times k \times \mathbf{g} + \mathbf{i}_s$ and $m \times \mathbf{a} + \mathbf{i}_s$ (l and m are integers) are seeds for other blocks. These blocks are clustered again, and are allocated to a hypercube structure by dividing one block into two halves and proceeding recursively.

The main advantage of this method is the reduction in the amount of data flow. Because the grouping vector is chosen as the shortest projected dependency vector, many dependencies are limited inside a group. Because two projected dependency vectors are used to create other blocks, they are also the interconnections between blocks, so physical links can be assigned to them, reducing the need to relay data. The method has three

drawbacks, two of which are significant. Firstly, the way of allocating nodes was given only for a fixed-size hypercube structure. If the hypercube structure is not used, it is not clear how to map these blocks onto an actual array. The hypercube is one kind of architecture, but not always available. The second problem is more problematical. In one block, there can be more than one node which lies on the same time hyperplane. To avoid conflicts, extra control or further segmentation is needed, but not discussed. Finally, like Darte's method, data flows cannot be made into a single package easily.

Because Moldovan's method was thought to be invalid when there are counter data flows between the bands, a modification of choosing the bands to be parallel to the timing hyperplanes was proposed in [25]. This change is so significant that it is no longer LPGS. The computation polyhedron is sliced into bands, each of which consist of c timing hyperplanes ($c = \det(\mathbf{T})$, the compressing factor as above). (c-1) vectors $\mathbf{v}_1, ..., \mathbf{v}_{c-1}$ are chosen such that nodes $\mathbf{i}, \mathbf{i} + \mathbf{v}_1, ..., \mathbf{i} + \mathbf{v}_{c-1}$, are in the different timing hyperplanes of a band. These nodes are active at different instances, therefore they can be assigned to one processor. Hence within a band, the method is similar to LSGP. However, these bands will be carried out in sequence, so the method can be thought to be "locally sequential" and "partially parallel" and "globally sequential". Unfortunately how to select the (c-1) vectors remains a serious problem. Because there is no systematic way to select them and no rules to determine their positions, it is possible to merge remote nodes to one processor, resulting in non-local data flows.

The problem of mapping piecewise regular algorithms onto piecewise regular arrays with a given number of processors by means of applying the above LPGS and LSGP methods is considered in [99].

2.3.2 Partitioning-Projecting Methods

Tiling, [108] [46], is another class of partitioning, which is usually carried out on the original computation polyhedron. First, cluster adjacent nodes as a block (referred to as a supernode or a tile) according to some criteria. The original computation space is transformed to a supernode space, and the original polyhedron, to a supernode polyhedron. Then the supernode polyhedron is mapped onto a processor array. An obvious advantage

of tiling over the previous methods is that because a supernode sends its output data just once after all computations in it are finished, the data can be easily transferred as a single package.

As an example of tiling, consider the method in [46]. Here a set of hyperplanes $\mathbf{h}_i \times x = k$ is introduced, where $i = 1, ..., N$; $x \in R^N$; $k \in Z$, which slices the computation polyhedron in N directions. The nodes enclosed in one section formed by these hyperplanes is assigned to one supernode. The method is relatively straightforward, the major contribution being the derivation of a number of conditions for the selection of the $\mathbf{h}_i$. Unfortunately the problem of how to map the supernodes onto a fixed-connection and fixed-size processor array is not considered.

King, Chou and Ni [52] considered the problem of partitioning according to dependency vectors in the 2-D case. They found that among 2-D dependency vectors it is always possible to find two, say $\mathbf{d}_1$ and $\mathbf{d}_2$, such that every other dependency can be expressed as a positive combination of the two vectors, i.e., $c_i \times \mathbf{d}_i = a_i \times \mathbf{d}_1 + b_i \times \mathbf{d}_2$, where $\mathbf{d}_i \in \mathbf{D}$; $c_i > 0$; a_i and $b_i \geq 0$. Let $l_1 = max(a_i/c_i)$ and $l_2 = max(b_i/c_i)$ $\forall i$. If the nodes in a 2-D computation space are clustered into supernodes by parallelograms with edges along $\mathbf{d}_1$ with size l_1 and along $\mathbf{d}_2$ with size l_2, no matter how many and what the original dependency vectors are, there exists three dependency vectors for the supernodes, that is [1,0], [0,1] and [1,1]. This idea is beneficial for the simplification and unification of parallelisation procedures, because canonical (or unchanged) dependency matrix can be used. However, due to computational difficulties, they did not extend the idea to the 3-D case, did not deal with the problem of space-time mapping, and so failed to exploit the benefits which may result from the canonical form of the dependency matrices.

2.3.3 Comparison

Comparing the two main classes of methods, projecting-partitioning or partitioning-projecting, we can make a summary of their main advantages and disadvantages:
If space mapping is performed before partitioning, i.e., projecting-partitioning, it is relatively easy to map a polyhedron onto a given size array. Because after the space projecting, we have the information about the virtual array, so it would be relatively easy to par-

tition the virtual array to fit the actual array. In addition, and quite importantly, high efficiency is easily achieved. The two weaknesses of this class are, firstly, in LSGP-like methods the time layer is too thin, so that output data can not stay in a partition long enough to be packed with others. Secondly, the shape and size of clustered blocks are determined with the requirement of no conflicts. As a result, there is no guarantee that the resulting interconnections will be fitted to the given interconnection primitives, or that local communications will be possible. In contrast, the partitioning-projecting class possesses the opposite strong and weak points.

In the tiling method where partitioning is performed ahead of projection, during the partitioning process we do not have the necessary information to guide the method to scale the size of a supernode such that the supernode polyhedron can then be mapped onto an given array. This information is closely related to the problem of space projection. Furthermore, unless the space-time matrix for the mapping of the supernode space onto the actual computational space is unimodular, the efficiency tends to be quite low. Of course, tiling also has its advantages. As mentioned above, because the supernode can be treated just like nodes, data flows can be easily transferred as a package. The requirement of supernode interconnections being fitted to given primitives and the locality of data flows can be obtained by means of determining a suitable shape and size for the supernodes.

As both classes have their own fundamental weaknesses, it is difficult to apply them to practical applications. As a result, the development of software packages for practical use in automatic parallelization has been restricted.

2.4 Lower-Dimensional Mapping

In practice, it is desirable to consider lower dimensional arrays especially when $N > 3$ so as to produce physically realistic architectures.

The problem of mapping an algorithm onto a lower dimensional processor array has received relatively less attentions to date. Some work concentrates on a specific problem [51], such as mapping four or five dimensional bit level algorithms onto a two-dimension processor array. There are also a number of papers dealing with more general cases [104], [58], [59], [94], [110].

The method in [104] was an early attempt to deal with the issue of lower dimensional mapping and is essentially simple to understand. A non-singular $N \times N$ matrix $\mathbf{T} = \begin{bmatrix} \mathbf{S} \\ \mathbf{\Pi} \end{bmatrix}$ is used to map the original computational N-D polyhedron to form a K-D time polyhedron by $\mathbf{\Pi}$ which is a $K \times N$ timing matrix, and a (N-K)-D processor array by $\mathbf{S}$ which is a $(N-K) \times N$ space mapping matrix. A K-D minimum hypercube is built to contain the K-D polyhedron. Finally, a K-D projecting vector is used to map the hypercube into a 1-D time domain without producing computational conflict. The shortcoming of the technique is that when an irregular K-D polyhedron is regularised to a hypercube, the total volume of the computational region can be increased significantly, thus reducing efficiency. An altenative, but very similar, method involving a mapping onto lower-dimensional array by means of a multiprojection is given in [99]. Unfortunately there is no evaluation to show that the approach achieves an optimum multiprojection for the cases of irregular polyhedra.

An alternative approach to using two time projections (first from N-D to K-D then from K-D to 1-D time) is a single timing vector $\bar{\mathbf{t}}$ to map the original N-D polyhedron onto 1-D time domain directly (see [58], [59]). However, the resulting $\mathbf{T} = \begin{bmatrix} \mathbf{S} \\ \bar{\mathbf{t}} \end{bmatrix}$ is singular matrix, because $\mathbf{S}$ is still $(N-K) \times N$. It is well-known that a singular transformation matrix will result in a conflict mapping, i.e., more than one node in the polyhedron is mapped to the same processor and executed at the same instance, and thus strictly forbidden. To avoid this problem, a number of conditions are usually introduced .

The same concept is expressed in a better way by [94] where the original computational polyhedron is assumed to be a hypercube. Because $rank(\mathbf{T}) < N$, the null-space of $\mathbf{T}$ consists of the vectors $\mathbf{c}'$ by which two nodes $\mathbf{j}$ and $\mathbf{j}+\mathbf{c}'$ are mapped to the same space-time point. Once $\mathbf{S}$ is given, the problem is focused on looking for the minimum $\bar{\mathbf{t}}$, under some other conditions, such that no integral members of the null-space of $\mathbf{T}$ can be contained within the hypercube. Such a $\mathbf{T}$ is termed conflict-free. In principle, it is always possible to find a conflict-free $\mathbf{T}$.

Unfortunately, it is very difficult to find the minimum $\bar{\mathbf{t}}$ associated with a conflict-free $\mathbf{T}$. The method for seeking $\bar{\mathbf{t}}$ is: (1) from a small norm, produce all of the vectors $\bar{\mathbf{t}}$ having the same norm; (2) for each of the $\bar{\mathbf{t}}$'s, check whether the hypercube contains any

integral member of the kernel of $\mathbf{T}$. If some integral members are in the kernal, increase the norm of $\bar{\mathbf{t}}$, repeat (1) and (2). Clearly this searching procedure is extremely time consuming. For example, step (2) may have to be repeated as many times as the volume of the computational polyhedron, a tremendous number of computations when the size of the polyhedron is large. In addition, step (2) also poses a difficult problem in the theory of linear programming [110].

Furthermore, the performance of the method is often not that good. For example, if the original polyhedron is a hypercube, the method is good, because it enumerates all the possible $\bar{\mathbf{t}}$ one by one. However, when the shape of the original polyhedron is irregular, which is very common (e.g., banded sparse matrix problems), the method still uses a hypercube to contain the polyhedron, and is unable to exploit the benefit arising from the irregularity. This is because it is more difficult to perform step (2) on an irregular shaped polyhedron, and also because the method only uses a conflict-free criterion to select $\bar{\mathbf{t}}$ (this is not sufficient). Besides the conflict-free condition, we must take the order of executions into consideration. Otherwise the executions of a computation may violate data dependency requirements. For example, if the original polyhedron is defined by vertices $[0,0,0]^T$, $[0,0,10]^T$, $[10,0,10]^T$ and $[10,10,10]^T$, it can be verified that no more than one node can be mapped to the same instance by a $\bar{\mathbf{t}}$ = [64,10,1], i.e., conflict-free. But the $\bar{\mathbf{t}}$ cannot be used, because a node $[9,9,10]^T$ is mapped to 676, while $[10,0,10]^T$ is mapped to 641, which violates a lexicographic order of execution which is usually assumed. To avoid this problem with the conflict-free mapping method, we have to embed the irregular polyhedron into the smallest possible hypercube.

The General Parameter Method (GPM) was proposed in [34]. The GPM uses $(M+1)\times N$ independent parameters to represent the matrix $\mathbf{T}_{(M+1)\times N}$ and searches for optimal solutions under some constraints for these parameters. It was shown that this method is equivalent to [94], so the method suffers similar shortcomings although searching complexity is improved.

In general the methods of [58], [59], [94] and [34] which all attempt to construct $\mathbf{T}_{(M+1)\times N}$ encounter serious obstacles in synthesising an actual array. We must determine which node is addressed by a particular processor at a particular time. It is easy to map

an N-D domain to an (M+1)-D domain by $\mathbf{T}_{(M+1)\times N}$, but an obvious difficulty is how to do the reverse: the mapping from (M+1)-D domain to N-D domain. No papers deal with the problem so far but it is essential for code generation.

2.5 Summary

In this survey, the two main classes of partitioning and mapping, independent partitioning and general partitioning and mapping, are discussed. From the discussion, we can say that the theory on independent partitioning for UREs is essentially mature.

The general partitioning and mapping for UREs is still an open area for further study. The existing methods, such as LSGP, LPGS and Tiling, all suffer from various kinds of disadvantages ranging from hardware limitation, and low computation or communication efficiency to design difficulties. We will address and solve some of these problems in the following chapters by both improving existing methods and proposing a new integrated methodology.

Chapter 3

Improvements of Some Existing Partitioning and Mapping Methods

In this chapter we consider some improvements to independent partitioning, LSGP and LPGS methods. In Chapter 2 we discussed these partitioning approaches [93], [25] and [73] and the important roles they have played in the development of field. For the independent partitioning of [93], a simplified SNF partitioning method is suggested and the generation of parallel codes is presented. For the LSGP of [25], we propose a synthesis method which guarantees the production of a given-regular-shapemapping easily. For the LPGS of [73], one improved approache is presented which remove the long-distance links for data flow from one side of the array to the other and is helpful in reducing the communication and the complexity of design.

3.1 On the Maximal Independent Partitioning

3.1.1 Introduction

As mentioned in Section 2.2, the best and simplest way of parallelising nested loops is to partition them into several independent partitions if possible. Each of the partitions can be completed in one processor, so there is no need for inter-processor communications. The SNF method of [93] is successful for this objective.

To place the improvements in context it is necessary to give a brief description of the SNF method. In algebra, if a matrix $\mathbf{M}$ can be transformed into $\mathbf{M}'$ by only elementary row and column operations, $\mathbf{M}$ is said to be equivalent to $\mathbf{M}'$, noted as $\mathbf{M} \sim \mathbf{M}'$. The

SNF of $\mathbf{D}$ is such that

$$\mathbf{U}^S\mathbf{D}\mathbf{V}^S = \mathbf{R}^S = \begin{bmatrix} \mathbf{R}_s^S & \mathbf{O} \\ \mathbf{O} & \mathbf{O} \end{bmatrix}$$

where $\mathbf{U}^S$ and $\mathbf{V}^S$ are unimodular matrices, $\mathbf{R}_s^S = diag(r_0^S, \cdots, r_{s-1}^S)$ is an integral $s \times s$ diagonal matrix and $r_0^S \mid r_1^S \mid \cdots r_{s-1}^S$. "|" means "divide".

The SNF approach of producing independent partitioning ($\mathbf{U}^S$, $\mathbf{r}^S$) is described as follows:

Let partitioning matrix $\mathbf{U}^S$ be such that $\mathbf{U}^S\mathbf{D}\mathbf{V}^S = \mathbf{R}^S$. Define $\mathbf{r}^S = [r_0^S, \cdots, r_{s-1}^S, \infty, \cdots, \infty]^T \in Z^N$. The partitions $\{I_{\mathbf{y}(1)}, \cdots, I_{\mathbf{y}(\mu)}\}$ of the original computational domain S is such that $I_{\mathbf{y}(j)} := \{\mathbf{i} : \mathbf{U}^S\mathbf{i}(mod\ \mathbf{r}^S) = \mathbf{y}(j), \mathbf{i} \in S\}$, where $\mathbf{y}(j) \in Z^N$ is referred to as the index vector of the partition $I_{\mathbf{y}(j)}$, $j = 1, \cdots, \mu$ (the notation $\mathbf{U}\mathbf{i}(mod\ \mathbf{r})$ means that $\forall i$ and do $\overline{\mathbf{u}}_i\mathbf{i}(mod\ r_i)$, where $r_i \in \mathbf{r}$ and $\overline{\mathbf{u}}_i$ is the ith row of $\mathbf{U}$).

$\mathbf{i}^1$ and $\mathbf{i}^2$ are said to be pseudo-connected if there exists a vector λ such that $\mathbf{i}^2 = \mathbf{i}^1 + \mathbf{D}\lambda$, that is, they are dependent on each other if they are pseudo-connected, and vice versa. So the independent partitioning is such that the pseudo-connected and only pseudo-connected nodes are grouped into one partition. It can be proved [93] that $\mathbf{i}^1$ and $\mathbf{i}^2$ are pseudo-connected iff $\mathbf{U}^S\mathbf{i}^1(mod\ \mathbf{r}^S) = \mathbf{U}^S\mathbf{i}^2(mod\ \mathbf{r}^S)$. It is not easy to give an intuitive explanation for the SNF method, but we will explain its nature later in Subsection 3.1.4.

However, the SNF is not the only way to achieve the maximal partitioning. We propose two methods, Diagonalisation Form (DF) and a simplified version of SNF (HSNF), which can do the same task but HSNF is preferable. An algorithm to produce HSNF, as well as SNF, via DF is presented, which shows a significant computational advantage. Furthermore, the problem of algorithm transformation, which was left open in [93], is also discussed.

3.1.2 Improved Methods of Independent Partitioning

Inspecting carefully the proof of Lemma 5.1 of [93], one can find that the condition $\mathbf{U}^S\mathbf{D}\mathbf{V}^S = \mathbf{R}^S$ is a sufficient condition for the maximal partitioning but not the necessary one. The condition $r_0^S \mid r_1^S \mid \cdots r_{s-1}^S$ is unnecessary. In fact, it was not mentioned in the

proof of the lemma. Therefore, $\mathbf{U}^S$ and $\mathbf{r}^S$ can be replaced by $\mathbf{U}^D$ and $\mathbf{r}^D$ such that

$$\mathbf{U}^D\mathbf{D}\mathbf{V}^D = \mathbf{R}^D = \begin{bmatrix} \mathbf{R}_s^D & \mathbf{O} \\ \mathbf{O} & \mathbf{O} \end{bmatrix}$$

$\mathbf{R}_s^D = diag(r_0^D, \cdots, r_{s-1}^D)$ is an integral diagonal matrix. $\mathbf{R}^D$ is a diagonal matrix but not SNF. Let $\mathbf{r}^D = [r_0^D, \cdots, r_{s-1}^D, \infty, \cdots, \infty]^T \in Z^N$. The partitioning $(\mathbf{U}^D, \mathbf{r}^D)$ is called the Diagonalisation Form partitioning (DF), and it can be proved by the proof of Lemma 5.1 of [93] that $\mathbf{i}^1$ and $\mathbf{i}^2$ are pseudo-connected iff $\mathbf{U}^D\mathbf{i}^1(mod\ \mathbf{r}^D) = \mathbf{U}^D\mathbf{i}^2(mod\ \mathbf{r}^D)$.

The first matter we are concerned with is whether the SNF partitioning of $(\mathbf{U}^S, \mathbf{r}^S)$ and the DF of $(\mathbf{U}^D, \mathbf{r}^D)$ have the same result (that is they obtain the maximal partitioning). We will see later that $det(\mathbf{R}_s^S) = det(\mathbf{R}_s^D)$, i.e., their upper bounds of partition number are the same.

The advantage of DF partitioning $(\mathbf{U}^D, \mathbf{r}^D)$ lies in the fact that much less computation is needed to diagonalise a matrix than to produce the SNF of the matrix. However, as shown later, from the view-point of producing parallel codes, $(\mathbf{U}^S, \mathbf{r}^S)$ sometimes has an advantage over $(\mathbf{U}^D, \mathbf{r}^D)$, because $\mathbf{r}^S$ may have more "1" elements than $\mathbf{r}^D$. The more "1" elements it has, the simpler the parallel codes corresponding to it will be, which is explained on page 39 after we introduce the structure of parallel programs.

However, in the procedure of producing the SNF, once all 1's have been found, we can stop the procedure because the actual SNF itself is unnecessary. Therefore, a "half" SNF (HSNF) is suggested, which is a diagonal matrix which is not SNF but which has as many "1" as possible on its diagonal. Then the corresponding partitioning is $(\mathbf{U}^H, \mathbf{r}^H)$, where $\mathbf{U}^H$ is such that $\mathbf{U}^H\mathbf{D}\mathbf{V}^H = \mathbf{R}^H$, and $\mathbf{r}^H$ is the diagonal of $\mathbf{R}^H$ with "0" replacing the ∞; $\mathbf{R}^H$ is a diagonal matrix with as many as "1" elements as possible.

3.1.3 Producing HSNF

A new algorithm to produce the HSNF of the matrix $\mathbf{D}_{N\times n_d}$ is developed via a simple algorithm for diagonalising a matrix. The basic idea is as follows

Step 1 Diagonalise $\mathbf{D}$.

Step 2 Calculate the gcd of the entries in the diagonal from the top-left one, say d_{00}, to the right-bottom. If gcd $\neq 1$, stop.

Step 3 Check whether the top-left entry, say d_{00}, can divide all the entries in the diagonal. If any one, say d_{jj} can not be divided by d_{00}, add d_{jj} to d_{j0}.

Step 4 Diagonalise **D** again. All entries in the second diagonalised matrix can be divided by its top-left entry.

Step 5 Go to Step 2 to repeat this procedure for $d_{11}, d_{22}, \cdots$ as the top-left entry, in turn.

Algorithm 3.1.1 *Diagonalise Matrix*-**D**(**Y**, **D**, **Z**, *top-left)*

FOR i := top-left TO min(N,n_d)-1
  DO {
    FOR $j := i+1$ TO N-1 *operation 1*
      $g := gcd(d_{ii}, d_{ji})$ and calculate p and q such that $g = p \times d_{ii} + q \times d_{ji}$
      $\mathbf{X} := \begin{bmatrix} p & -d_{ji}/g \\ q & d_{ii}/g \end{bmatrix}$, $\begin{bmatrix} \overline{\mathbf{d}}_i \\ \overline{\mathbf{d}}_j \end{bmatrix} := \mathbf{X}^T \begin{bmatrix} \overline{\mathbf{d}}_i \\ \overline{\mathbf{d}}_j \end{bmatrix}$, $\begin{bmatrix} \overline{\mathbf{y}}_i \\ \overline{\mathbf{y}}_j \end{bmatrix} := \mathbf{X}^T \begin{bmatrix} \overline{\mathbf{y}}_i \\ \overline{\mathbf{y}}_j \end{bmatrix}$
    FOR $j := i+1$ TO $n_d - 1$ *operation 2*
      $g := gcd(d_{ii}, d_{ij})$ and calculate p and q such that $g = p \times d_{ii} + q \times d_{ij}$
      $\mathbf{X} := \begin{bmatrix} p & -d_{ij}/g \\ q & d_{ii}/g \end{bmatrix}$, $[\mathbf{d}_i : \mathbf{d}_j] := \mathbf{X}[\mathbf{d}_i : \mathbf{d}_j]$, $[\mathbf{z}_i : \mathbf{z}_j] := \mathbf{X}[\mathbf{z}_i : \mathbf{z}_j]$
  }WHILE(d_{ii} is not the only non-zero entry in $\mathbf{d}_i$)
End of Algorithm

where $\mathbf{d}_i$ and $\overline{\mathbf{d}}_i$ are the column and row vectors of **D**, respectively, and $d_{ij} \in \mathbf{D}$. This algorithm appears straightforward, *operation 1* and *operation 2* are designed to annihilate the entries, except d_{ii}, on $\overline{\mathbf{d}}_i$ and $\mathbf{d}_i$, respectively. It is worthwhile mentioning that after the *operation 1*, d_{ii} becomes the gcd of $\mathbf{d}_i$. In fact, in the operation $\begin{bmatrix} \overline{\mathbf{d}}_i \\ \overline{\mathbf{d}}_j \end{bmatrix} := \mathbf{X}^T \begin{bmatrix} \overline{\mathbf{d}}_i \\ \overline{\mathbf{d}}_j \end{bmatrix}$, because $[p, q][d_{ii}, d_{ji}]^T = gcd(d_{ii}, d_{ji})$, $d_{ii} := gcd(d_{ii}, d_{ji})$; so $d_{ii} = gcd(d_{ii}, d_{i+1,i}, \cdots, d_{N-1,i})$. This also holds for *operation 2*. The following algorithm produces the HSNF of **D**.

Algorithm 3.1.2 *HSNF*-**D**(**Y**, **D**, **Z**)

Make identity matrices **Y** and **Z**
$i := 0$
Diagonalise Matrix-**D**(**Y**, **D**, **Z**, i) *operation 1*
WHILE($gcd(d_{ii}, \cdots, d_{min,min}) = 1$)
  FOR $j := i$ TO min(N,n_d)-1 *operation 2*
    Find the smallest entry, say d_{kk}, along the diagonal of **D** from d_{jj}
    Swap $\overline{\mathbf{d}_j}$ with $\overline{\mathbf{d}_k}$, do the same for **Y**, Swap $\mathbf{d}_j$ with $\mathbf{d}_k$, do the same for **Z**
    FOR $l := j+1$ TO min(N,n_d)-1 *operation 3*
      IF d_{ii} cannot divide d_{ll}
        $d_{jl} := d_{jl} + d_{ll}$ and $\mathbf{z}_j := \mathbf{z}_j + \mathbf{z}_l$

Diagonalise Matrix-$\mathbf{D}(\mathbf{Y}, \mathbf{D}, \mathbf{Z}, i)$ *operation 4*

$i := i + 1$

$\mathbf{U}^H := \mathbf{Y}, \quad \mathbf{V}^H := \mathbf{Z}$

End of Algorithm

We should prove that for any diagonal matrix $\mathbf{D} = diag(d_{00}, d_{11}, \cdots, d_{N-1,N-1})$, after *operation 2* and *operation 4*, its new d_{00} can divide any entries in the diagonal. In fact, without loss of generality, suppose d_{00} is the smallest in the diagonal and it can not divide any of $d_{11}, \cdots, d_{N-1,N-1}$. After *operation 3*, $\mathbf{d}_0$ of $\mathbf{D}$ becomes $[d_{00}, d_{11}, \cdots, d_{N-1,N-1}]^T$. In *Diagonalise Matrix*-$\mathbf{D}$, after annihilating $\mathbf{d}_0$, $d_{00} = gcd(d_{11}, \cdots, d_{N-1,N-1})$, and any other entries of $\mathbf{D}$ are a linear combination of $d_{11}, \cdots, d_{N-1,N-1}$. As a consequence of this, the new d_{00} can divide any other entries in the modified $\mathbf{D}$. Therefore, at most, N-1 iterations are needed to produce the SNF of $\mathbf{D}$. It remains to explain the WHILE loop test. Less computations are required to produce HSNF because the process stops midway when all "1" elements in the diagonal have been found.

Comparing with the method [49], our algorithm is faster. In step i, there is (N-i-1)-D HNF at the bottom-left of the operating matrix in the method of [49], instead of a (N-i-1)-D diagonal submatrix in our method. To ensure the property of $d_{ii} \mid d_{(i+1),(i+1)}$, we have to check if any entries of the (N-i-1)-D submatrix at the bottom-left of the operating matrix cannot be divided by d_{ii}. If there is, the operating matrix must be modified and the same operation (computiong HNF or diagonal matrix) must be repeated. Therefore in step i, there are $(N-i-1)^2/2$ operation of computing HNF at most in the method of [49], while there are $(N-i-1)$ operation of computing diagonal matrix at most in our method. Even if the computation of diagonalising a matrix is roughly twice that of producing the HNF, our algorithm is still much faster.

It is easy to show that $det(\mathbf{R}_s^H) = det(\mathbf{R}_s^S)$. In fact, after being diagonalised for the first time, $\mathbf{D}$ has the form of $\begin{bmatrix} \mathbf{R}_s^D & \mathbf{O} \\ \mathbf{O} & \mathbf{O} \end{bmatrix}$. Any following elementary row and column operation for HSNF or SNF are limited to only the first s rows and the first s columns. No nonzero entries are created outside the square area of $\mathbf{R}_s^D$. So because only elementary row and column operations are applied, there must be $\mid det(\mathbf{R}_s^H) \mid = \mid det(\mathbf{R}_s^S) \mid$. An

example may be helpful to illustrate the HSNF algorithm (but going on to SNF):

$$\underset{Original}{\begin{bmatrix} 8 & -55 & 0 \\ 4 & -22 & 0 \\ 6 & 0 & 2 \end{bmatrix}} \overset{operation1}{\Longrightarrow} \underset{DF}{\begin{bmatrix} 11 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}} \overset{operation2}{\Longrightarrow} \begin{bmatrix} 2 & 0 & 0 \\ 11 & 11 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

$$\overset{operation4}{\Longrightarrow} \underset{HSNF}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 22 & 0 \\ 0 & 0 & 4 \end{bmatrix}} \overset{operation2}{\Longrightarrow} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 22 & 22 \end{bmatrix} \overset{operation4}{\Longrightarrow} \underset{SNF}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 44 \end{bmatrix}}$$

3.1.4 Algorithm Generation

The SNF independent partitioning approach is an important advance in theory. However, the suggested partitioning procedure of assigning any $\mathbf{i} : \mathbf{i} \in S$ *and* $\mathbf{U}^S\mathbf{i}(mod\ \mathbf{r}^S) = \mathbf{y}(j)$ into a partition $\mathbf{y}(j)$ is impractical. If the S is large, the computation of $\mathbf{U}^S\mathbf{i}$ will be very time-consuming. Moreover, no parallel codes are produced in this way. We hope to transform a serial algorithm into parallel one as the result of the independent partitioning procedure. Fortunately, this is not too difficult with HSNF.

Having obtained the HSNF of $\mathbf{D}$ using Algorithm 3.1.2, in order to move the potential parallel loops so they become the outermost loops, reverse $\mathbf{R}^H$, that is

$$\mathbf{J}\mathbf{U}^H\mathbf{D}\mathbf{V}^H\mathbf{J} = \mathbf{J}\mathbf{R}^H\mathbf{J}$$

For simplicity, let $\mathbf{R} = \mathbf{J}\mathbf{R}^H\mathbf{J}$. We know that $\mathbf{R} = diag(r_0, \cdots, r_{N-1})$, where

$$r_i \begin{cases} = 0 & i = 0, \cdots, n_0 - 1 \\ > 1 & i = n_0, \cdots, n_0 + n_l - 1 \\ = 1 & i = n_0 + n_l, \cdots, N - 1 \end{cases}$$

where $n_0 = N - r$ and n_l is the number of diagonal entries larger than 1.

Let the transformation matrix be defined as $\mathbf{T} = \mathbf{J}\mathbf{U}^H$. Because $\mathbf{T}$ is unimodular, the original sequential FOR-Loops, indexed by $\mathbf{i} = [i_0, \cdots, i_{N-1}]^T$, can be transformed directly to another set of FOR-loops, indexed by $\mathbf{j} = [j_0, \cdots, j_{N-1}]^T$, such that $\mathbf{j} := \mathbf{T}\mathbf{i}$. that is

$$\begin{array}{l} \text{FOR } i_0 := l_0^i \text{ TO } u_0^i \\ \quad \cdots \\ \quad\quad \text{FOR } i_{N-1} := l_{N-1}^i \text{ TO } u_{N-1}^i \end{array} \overset{\mathbf{j} := \mathbf{T}\mathbf{i}}{\Longrightarrow} \begin{array}{l} \text{FOR } j_0 := l_0^j \text{ TO } u_0^j \\ \quad \cdots \\ \quad\quad \text{FOR } j_{N-1} := l_{N-1}^j \text{ TO } u_{N-1}^j \end{array}$$

where l_k^i and u_k^i may be the functions of $i_0, \cdots, i_{k-1}$; l_k^j and u_k^j may be the functions of $j_0, \cdots, j_{k-1}$.

It is interesting to see the data dependencies in the transformed space. Let $\mathbf{V}^J = (\mathbf{V}^H\mathbf{J})^{-1}$. Note that $\mathbf{V}^J$ is a unimodular matrix. In fact, the new dependencies, $\mathbf{D}^j$, are

$$\mathbf{D}^j = \mathbf{TD} = \mathbf{R}(\mathbf{V}^H\mathbf{J})^{-1} = \mathbf{RV}^J = \begin{bmatrix} \mathbf{O}_{n_0} \\ r_{n_0} \times \overline{\mathbf{v}^J}_{n_0} \\ \cdots \\ r_{n_0+n_l-1} \times \overline{\mathbf{v}^J}_{n_0+n_l-1} \\ \overline{\mathbf{v}^J}_{n_0+n_l} \\ \cdots \\ \overline{\mathbf{v}^J}_{N-1} \end{bmatrix}$$

where $\mathbf{O}_{n_0}$ is a n_0 row matrix and $\overline{\mathbf{v}^J}_i \in \mathbf{V}^J$ and $gcd(\overline{\mathbf{v}^J}_i) = 1$.

Because the first n_0 rows of $\mathbf{D}^j$ are zero, the first n_0 FOR-loop's with respect to $j_0, \cdots, j_{n_0-1}$ can be carried out concurrently, so they become DOALL-loop's.

Consider each of the middle n_l rows of $\mathbf{D}^j$. Because of $gcd(\overline{\mathbf{d}}_i^j) = gcd(r_i\overline{\mathbf{v}^J}_i) = r_i$, where $\overline{\mathbf{d}}_i^j \in \mathbf{D}^j$, data dependencies jump r_i nodes in this dimension, so nodes can be grouped into r_i independent partitions along this direction. Recalling the analysis of the nature of the independent partitioning in Section 2.2, this is exactly the Greatest Common Divisor method. Therefore the SNF method can be said equivalent to the GCD method after a linear transformation $\mathbf{T}$.

The FOR-loop with respect to j_i corresponding to the $r_i > 1$ can be split into two loops

DOALL $j_i^p = 0$ TO $r_i' - 1$

FOR $j_i := j_i^p + r_i\lceil(l_i^j - j_i^p)/r_i\rceil$ TO u_i^j STEP r_i

Because $r'_{n_0+n_l} = \cdots = r'_{N-1} = 1$, to outer-most levels the last n_l FOR-loops with respect to $j_{n_0+n_l}, \cdots, j_{N-1}$ cannot be split any more, so they remain unchanged. Let $n_p = n_0 + n_l$. All "DOALL $j_i^p = 0$ TO $r_i' - 1$" can be moved outwards. In fact the outward movements change nothing with the lexicographical order of the execution of the loops, because it is the j_i's (not j_i^p's) that are the real indices to access an node and they

stay in their original order. Therefore, the parallel algorithm has a form:

Loops 3.1.1 *The parallel algorithm of Independent Partitioning*

$$\text{DOALL } j_0 := l_0^j \text{ TO } u_0^j$$
$$\cdots$$
$$\text{DOALL } j_{n_0-1} := l_{n_0-1}^j \text{ TO } u_{n_0-1}^j$$
$$\text{DOALL } j_{n_0}^p = 0 \text{ TO } r'_{n_0} - 1$$
$$\cdots$$
$$\text{DOALL } j_{n_p-1}^p = 0 \text{ TO } r'_{n_p-1} - 1$$
$$\text{FOR } j_{n_0} := j_{n_0}^p + r'_{n_0}\left\lceil \frac{l_{n_0}^j - j_{n_0}^p}{r'_{n_0}} \right\rceil \text{ TO } u_{n_0}^j \text{ STEP } r'_{n_0}$$
$$\cdots$$
$$\text{FOR } j_{n_p-1} := j_{n_p-1}^p + r'_{n_p-1}\left\lceil \frac{l_{n_p-1}^j - j_{n_p-1}^p}{r'_{n_p-1}} \right\rceil \text{ TO } u_{n_p-1}^j \text{ STEP } r'_{n_p-1}$$
$$\text{FOR } j_{n_p} := l_{n_p}^j \text{ TO } u_{n_p}^j$$
$$\cdots$$
$$\text{FOR } j_{N-1} := l_{N-1}^j \text{ TO } u_{N-1}^j$$

The FOR-loop's are executed in one processor.

It has been mentioned that in many cases HSNF partitioning ($\mathbf{U}^H$, $\mathbf{r}^H$) is preferable to DF partitioning ($\mathbf{U}^D$, $\mathbf{r}^D$). This statement relies on the fact that although the DF and HSNF share the same product of the diagonal elements, the latter may consist of more "1" elements than the former. For instance, in the example of the last subsection, DF has the diagonal elements of 11, 2, 4 and HSNF has 1, 22, 4. If the DF partitioning is adopted, two loops have to be split, whereas for HSNF, only one loop needs splitting. The less the loops are split, the simpler the parallel algorithm will be. There is essentially no difference between HSNF and SNF. So HSNF may be more useful because less computation is required to produce the HSNF .

3.2 A Synthesis Method using LSGP Partitioning for Given-Shape Regular Arrays

3.2.1 Introduction

We present a method to partition and map a computational polyhedron onto processor arrays. Based on the theoretical framework of an existing LSGP method, a systematic design procedure is proposed which is able to map the polyhedron onto a regular array with given regular shape, i.e., given sizes in each dimension.

3.2.2 A New LSGP Synthesis Method

The method proposed in [24] (see Subsection 2.3.1) is an important theoretical advance. We recall the major points of [24]. There are four essential relationships: $c = \bar{\mathbf{t}}\mathbf{s}^p$; $\mathbf{S}\mathbf{s}^p = \mathbf{o}$; $\bar{\mathbf{t}}\mathbf{Q} = \bar{\mathbf{o}}$; $\mathbf{A} = \mathbf{S}\mathbf{Q}$. And finally, $\mathbf{A}$ can be made equivalent to an triangular matrix (HNF) whose diagonal entries can be used to define a conflict-free partitioning box.

However, it has also be pointed that this partitioning method suffers a significant drawback when mapping a polyhedron onto a given regular processor array. In view of the neat formalism in [24], we wish to focus on modifying the method to overcome this difficulty. To achieve this goal, we consider the problem from a different perspective.

From a $\mathbf{s}^p$, we can construct a space mapping $\mathbf{S}$, by which a projected virtual array can be determined provided that the vertices of the computational polyhedron are given [1]. Thus, the required compression factors in each dimension can be found very easily. We can construct an upper triangular matrix $\mathbf{A}$ with the compression factors as its diagonals. Then, find the $\mathbf{Q}$ by solving $\mathbf{S}\mathbf{Q} = \mathbf{A}$. Finally, we determine $\bar{\mathbf{t}}$ from $\mathbf{Q}$ under some additional constraints, such as $\mathbf{S}\mathbf{D} = \mathbf{P}\mathbf{K}$ [73]. Obviously, it is ensured that the original computational polyhedron can be partitioned and mapped onto and within the given regular processor array. Our method is illustrated as

$$\mathbf{s}^p \Longrightarrow \mathbf{S} \overset{k_0, \ldots, k_{N-2}}{\Longrightarrow} \mathbf{A} \Longrightarrow \mathbf{Q} \Longrightarrow \bar{\mathbf{t}}$$

Compared with the method on page 23, because the compression factors are set as the parameters of the procedure, the given-regular-shape mapping is achieved in principle and the derivation of the HNF is no longer needed.

This new method is described in detail as follows:

Step 1 Choose a $\mathbf{s}^p = [s_0^p, \ldots, s_{N-1}^p]^T$ with $s_{N-1}^p = 1$. Construct the space mapping which has a form of

$$\mathbf{S} = [\mathbf{S}' : \mathbf{s}'] \tag{3.1}$$

where $\mathbf{S}'$ is a $(N-1) \times (N-1)$ unimodular matrix.

[1] This is the usual case. However, as will been seen, there are some cases where the computational body is infinite, like a semi-infinite cylinder

Let $\mathbf{S}' = \mathbf{I}$ for simplicity (not necessary, but beneficial to the following operations). By the definition of $\mathbf{S}$, we know that $\mathbf{S}\mathbf{s}^p = \mathbf{o}$, where $\mathbf{o}$ is a null-filled column vector. $\mathbf{s}'$ is obtained as $\mathbf{s}' = -\mathbf{S}'[s_0^p, \ldots, s_{N-2}^p]^T$, because $s_{N-1}^p = 1$.

Step 2 Determine the compression factors in each dimension. At first, project all the possible vertices to a set of projected vertices $\mathbf{W} = \mathbf{S}\mathbf{V}$.

The compression factors in each dimension are formulated as, $i = 0, \ldots, N-2$

$$k_i = \lceil \frac{\max_{0 \le j < n_v} w_{i,j} - \min_{0 \le j < n_v} w_{i,j}}{l_i} \rceil \tag{3.2}$$

where $w_{i,j} \in \mathbf{W}$, n_v is the number of vertices and l_i is the length of the processor array in the i-th dimension. The desired $c = \prod_{i=0}^{N-2} k_i$.

Step 3 Construct an $(N-1) \times (N-1)$ uppertriangular $\mathbf{A}$ with $k_0, \ldots, k_{N-2}$ as the diagonal elements.

May fill (not necessary, but beneficial to reducing the design complexity later) the entries above the diagonal: $\forall i$ and j, $0 \le i, < N-1$ and $i < j < N-1$,

$$a_{i,j} = \begin{cases} f & g = gcd(k_i, k_j) > 1 \\ 0 & otherwise \end{cases} \tag{3.3}$$

where $f = \frac{k_j}{g^r}$, $\mathrm{GCD}(f, g) = 1$ and $r \ge 1$. That is, f is a factor of k_j and without g as its factor. For example, for $k_j = 3 \times 5^2$ and $g = 5$, we have $r = 2$ and $f = 3$.

Step 4 Derive integral solutions for $\mathbf{Q}$, an $N \times (N-1)$ matrix, where

$$\mathbf{S}\mathbf{Q} = \mathbf{A} \tag{3.4}$$

Because rank(S) = N-1, we know that the general solution of $\mathbf{Q}$ consists of two parts,

$$\mathbf{Q} = \mathbf{Q}_p + \mathbf{q}^n \mathbf{b}^T \tag{3.5}$$

where $\mathbf{Q}_p$ is a particular solution, $\mathbf{q}^n$, the null-solution, and $\mathbf{b}$ is a combination vector. $\mathbf{Q}_p$ and $\mathbf{q}^n$ are given by the following proposition.

Proposition 3.2.1 $\mathbf{q}^n = \mathbf{s}^p$ *and* $\mathbf{Q}_p = \begin{bmatrix} \mathbf{A} \\ \overline{\mathbf{o}} \end{bmatrix}$

where $\overline{\mathbf{o}}$ stands for a null-filled row vector.

Proof: Obviously, since $\mathbf{Sq}^n = \mathbf{o}$ and $\mathbf{S}' = \mathbf{I}$, the first N-1 elements of $\mathbf{q}^n$ are $-\mathbf{S}'^{-1}\mathbf{s}' = -\mathbf{s}'$ if we let $t^n_{N-1} = 1$. Thus $\mathbf{q}^n = \mathbf{s}^p$. Fortunately, it is not difficult to find a particular solution under our arrangement for $\mathbf{S}$. Letting the last row of $\mathbf{Q}_p$ be a null vector, we have

$$\mathbf{Q}_p = \begin{bmatrix} \mathbf{S}'^{-1}\mathbf{A} \\ \overline{\mathbf{o}} \end{bmatrix} = \begin{bmatrix} \mathbf{A} \\ \overline{\mathbf{o}} \end{bmatrix} \tag{3.6}$$

□

Step 5 Select a $\mathbf{Q}$ from its general solutions obtained by changing $\mathbf{b}$, according to the following condition:

Condition 1. $\forall \mathbf{q}_i \in \mathbf{Q}$, $\mathrm{GCD}(\mathbf{q}_i) = 1$, where $\mathrm{GCD}(\mathbf{v})$ means the gcd of the elements of the vector $\mathbf{v}$.

Step 6 Derive the timing vector $\overline{\mathbf{t}}$ by solving

$$\overline{\mathbf{t}}\mathbf{Q} = \overline{\mathbf{o}} \tag{3.7}$$

and normalise by its gcd = $\mathrm{GCD}(\overline{\mathbf{t}})$. Test two conditions:

Condition 2. $\overline{\mathbf{t}}\mathbf{s}^p = c$ (the test for this condition is unnecessary if $\mathbf{S}' = \mathbf{I}$ and the $\mathbf{A}$ is filled as eqn (3.3)).

Condition 3. $\overline{\mathbf{t}}\mathbf{D} \geq \overline{\mathbf{k}}_p$. $\overline{\mathbf{k}}_p$ will be defined later (to allow implementation with a set of given interconnection primitives).

If conditions do not hold, go to **Step 5**.

3.2.3 The Conditions for Valid Q and $\overline{\mathbf{t}}$

In this section, we explain the three conditions and the synthesis of $\mathbf{A}$.

Valid Q

In the proof of Theorem 1 of [24], Matrix $\mathbf{Q}$ is a basis for a subspace in which all the nodes are executed at the same time but must be allocated to different processors in order to

avoid conflicts. To avoid any conflicts, the subspace spanned by $\mathbf{Q}$ must include all the nodes which are executed at the same time. In other words, it is necessary that $\mathbf{Q}$ can be extended to a basis for the full space, by which no nodes may be missed. Such a $\mathbf{Q}$ is termed valid. Let $\mathbf{Q}$ be extended to a square matrix $\mathbf{R} = [\mathbf{r}_0^S : \mathbf{Q}]$, which is the basis to access the full integer space. $\mathbf{R}$ must be unimodular (if not, there must be "holes" in the integer space spanned by $\mathbf{R}$). Obviously, $\exists \mathbf{r}_i^S \in \mathbf{Q}$ such that $g_i = GCD(\mathbf{r}_i^S) > 1$, we have $\mid \det\mathbf{R} \mid \geq g_i > 1$. So we give Condition 1 to avoid this case. Condition 1 is necessary but not sufficient.

The necessary and sufficient condition for an unimodular $\mathbf{R}$ is that

$$GCD(R_0, \ldots, R_{N-1}) = 1 \tag{3.8}$$

where R_i is the subdeterminant of $\mathbf{Q}$ obtained by deleting the i-th row and prefixing $(-1)^i$. In fact, if $\mathbf{R}$ is unimodular, we have

$$det\mathbf{R} = \sum_{i=0}^{N-1} r_{i,0} R_i = 1 \tag{3.9}$$

where $r_{i,0} \in \mathbf{r}_0^S$ and R_i is also the cofactor of $r_{i,0}$. It is well-known that the necessary and sufficient condition for an integral solution of $\mathbf{r}_0^S$ with eqn (3.9) is GCD($R_0, \ldots, R_{N-1}$) = 1. However, eqn (3.8) is not easily tested because we have to calculate N (N-1)-D determinants. Fortunately, we have another method to provide the sufficient condition

Theorem 3.2.1 $\mathbf{Q}$ *is valid, i.e.,* $GCD(R_0, \ldots, R_{N-1}) = 1$, *if* $\bar{\mathbf{t}}\mathbf{s}^p = c$.

This is Condition 2 where we use the $\bar{\mathbf{t}}$ which is produced from $\mathbf{Q}$, instead of $\mathbf{Q}$ itself, to check the validity of the $\mathbf{Q}$. Since the c is known and $\bar{\mathbf{t}}$ also is the result of the design, the test is easy to carry out.

Proof: First a little preparation. It is claimed that a row vector $\bar{\mathbf{t}}$ is called a valid timing vector if GCD($\bar{\mathbf{t}}$) = 1 and $\bar{\mathbf{t}}\mathbf{Q} = \bar{\mathbf{o}}$. Second $\mathbf{S}$ can be extended to an unimodular matrix

$$\mathbf{S}_u = \begin{bmatrix} 0 \ldots 0\,1 \\ \mathbf{S} \end{bmatrix}$$

It is claimed that the first row of $\mathbf{R}^{-1}$ is a valid $\bar{\mathbf{t}}$ and the first column of $\mathbf{S}_u^{-1}$ is $\mathbf{s}^p$.

In fact, let $g = GCD(R_0, \ldots, R_{N-1})$. Derive $\mathbf{r}_0^S$ by solving $\sum_{i=0}^{N-1} r_{i,0} R_i = g$. Obviously $det\mathbf{R} = g$. Let the first row of $\mathbf{R}^{-1}$ be $\overline{\mathbf{b}}$ which is nothing else but just $[R_0, \ldots, R_{N-1}]/g$. It is clear that the $\overline{\mathbf{b}}$ possesses the two properties of $\overline{\mathbf{t}}$.

$$\mathbf{S}_u = \begin{bmatrix} 0 \ldots 0\, 1 \\ \mathbf{S} \end{bmatrix} = \begin{bmatrix} 0 \ldots 0 & 1 \\ \mathbf{S}' & \mathbf{s}' \end{bmatrix}.$$

Letting the first column of $\mathbf{S}_u^{-1}$ be $\mathbf{x}$, we know $\mathbf{S}'\mathbf{x} = \mathbf{o}$ and $[0, \ldots, 0, 1]\mathbf{x} = 1$, which means $x_{N-1} = 1$, by definition. So the $\mathbf{x}$ must be $\mathbf{s}^p$.

Finally, it is known that the cofactor of the first element of $\mathbf{S}_u\mathbf{R}$ is c and $det(\mathbf{S}_u\mathbf{R}) = g$, because $\mathbf{S}_u$ is unimodular. $(\mathbf{S}_u\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{S}_u^{-1}$ whose first element is $\overline{\mathbf{t}}\mathbf{s}^p$. Thus $\overline{\mathbf{t}}\mathbf{s}^p = c/g$.

So we have $g = 1$ if $\overline{\mathbf{t}}\mathbf{s}^p = c$. □

Because all the null solutions of a system of N-1 homogeneous N-D equations are different only by a scalar factor, **Step 6** of the method produces the timing vector, and requires the solution of one (N-1)-D diophantine equation. According to Theorem 3.2.1 Condition 2 removes the possibility of conflicts. Condition 2 is necessary and sufficient, and includes Condition 1. However, we keep Condition 1 in **Step 5**, because it is easy to test without solving diophantine equations and will also be sufficient if **A** is chosen as in eqn (3.3).

The Design of A for valid Q

However, if we do not design the activity matrix **A** carefully, it is possible that only a small portion, or even none, of the **Q**'s produced by Step 5 can pass the test of Condition 2. A great waste of time. It would be best that **Step 5** produces only valid **Q**. To discuss the problem, let us consider the case of 3 dimensions. Our question is this, can **A** be just a simple diagonal matrix? If it cannot, then how do we fill the entries above the diagonal?

At first, consider the case when **A** is a diagonal matrix. We have

Theorem 3.2.2 *Let the* $\mathbf{A} = diag(k_0, k_1)$. *If* $GCD(k_0, k_1) = g > 1$, *no* **Q** *is valid; otherwise all are valid provided that Condition 1 is true.*

Proof: In fact, from eqn (3.6) and (3.5), we have, remembering $\mathbf{q}^n = \mathbf{s}^p$ and $s_{N-1}^p = 1$,

$$\mathbf{Q}^T = \begin{bmatrix} k_0 + b_0 s_0^p & b_0 s_1^p & b_0 \\ b_1 s_0^p & k_1 + b_1 s_1^p & b_1 \end{bmatrix}$$

then

$$\mid R_0 \mid = k_1 b_0, \quad \mid R_1 \mid = k_0 b_1 \quad and \quad \mid R_2 \mid = k_0 k_1 + k_0 b_1 s_1^p + k_1 b_0 s_0^p$$

It can be seen that R_0, R_1 and R_2 have at least one common divisor g, if $GCD(k_0, k_1) = g > 1$.

Otherwise, we have $GCD(k_0, k_1) = 1$. Suppose $k_0 b_1$ and $k_1 b_0$ share a common divisor $d > 1$. Then $k_0 k_1 + k_0 b_1 s_0^p + k_1 b_0 s_0^p = k_0 k_1 + dZ$, where Z is an integer. If $k_0 k_1 + k_0 b_1 s_0^p + k_1 b_0 s_0^p$ has the same factor d, $k_0 k_1$ must contain d, i.e., either k_0 or k_1 contains d. (if not, d cannot divide R_2). Because $GCD(k_0, k_1) = 1$, d must be a factor of either b_1 or b_0, or of both. Assume k_0 contains d. b_0 must contain the d, that is $GCD(k_0, b_0) = d$, because k_1 cannot contain d. This means that the elements of the first column of $\mathbf{Q}$ share a common divisor d. But this is impossible, because it violates Condition 1. The same argument holds assuming k_1 contains d. Therefore, d is not a factor of $k_0 k_1 + k_0 b_1 s_0^p + k_1 b_0 s_0^p$, because neither k_0 nor k_1 contain it. So we can claim that R_0, R_1 and R_2 do not share any common factors.□

Obviously, this is not a good choice because there is no guarantee of $GCD(k_0, k_1) = 1$. Now we have to fill the entries above the diagonal. There are two situations to consider, $GCD(k_0, k_1) = 1$ or > 1. First, consider the $\mathbf{A}$ such that $GCD(k_0, k_1) = 1$. We find some of the $\mathbf{Q}$'s are valid, some are not. For example, let

$$\mathbf{A} = \begin{bmatrix} 151 & 1 \\ 0 & 50 \end{bmatrix}$$

and $\mathbf{s}^p = [5, 2, 1]^T$. We may have a $\mathbf{Q}^T$,

$$\begin{bmatrix} 146 & -2 & -1 \\ -9 & 46 & -2 \end{bmatrix} \qquad when \ \ \mathbf{b} = [-1, -2]^T$$

Then R_0, R_1 and R_2 are 50, 301 and 6698, respectively. They have no common divisors.

However we can also have another $\mathbf{Q}^T$

$$\begin{bmatrix} 146 & -2 & -1 \\ -4 & 48 & -1 \end{bmatrix} \qquad when \ \ \mathbf{b} = [-1, -1]^T$$

R_0, R_1 and R_2 are 50, 150 and 7000, respectively, sharing the gcd 50. The $\bar{t} = [1, 3, 140]$. $\bar{t}s^p = 151$, but c = 7550. So the $\mathbf{Q}$ is invalid. This means that $a_{i,j}$ should be zero if $\mathrm{GCD}(k_i, k_j) = 1$.

As regards the case of $GCD(k_0, k_1) > 1$, we have

Theorem 3.2.3 *If* $\mathbf{A}$ *is an uppertriangular matrix in which* $GCD(k_0, k_1) = g > 1$ *and* $a_{i,j}$ $(i < j)$ *is chosen as* $f = k_j/g^r$, *all* $\mathbf{Q}$ *are valid if Condition 1 is true.*

Proof: $a_{0,1}$ is assigned as f and $k_1 = fg^r$. Then, we have

$$\mathbf{Q}^T = \begin{bmatrix} k_0 + b_0 s_0^p & b_0 s_1^p & b_0 \\ f + b_1 s_0^p & k_1 + b_1 s_1^p & b_1 \end{bmatrix}$$

and

$$\begin{aligned} \mid R_0 \mid &= k_1 b_0, \quad \mid R_1 \mid = k_0 b_1 - f b_0 \\ \mid R_2 \mid &= k_0 k_1 + k_0 b_1 s_0^p + k_1 b_0 s_0^p - f b_0 s_1^p \end{aligned} \tag{3.10}$$

Assume b_0, R_1 and R_2 share a common divisor d. Considering R_0, d must be a factor of either k_1 or b_0. At first, Assume k_1 contains d.

If d is g, considering R_1, it must be a factor of b_0 (if not, d (=g) cannot divide $R_1 = gZ - fb_0$, since no g in f). But, this violates Condition 1 with respect to the first row of $\mathbf{Q}^T$.

If d is a factor of f, R_1 can be rewritten as $k_0 b_1 - dZ$. To make d be a factor of $k_0 b_1 - dZ$, it must be a factor of either k_0 or b_1. It cannot be a factor of k_0, because it belongs to the part of k_1 which is not shared by k_0. Then d should be a factor of b_1. But, this means that d is also a common divisor of the elements of the second row of $\mathbf{Q}^T$, violating Condition 1.

Therefore, d cannot be a factor of k_1, so b_0 should contain it. To make it divide $k_0 b_1 - f b_0$, d must be a factor of either k_0 or b_1. It cannot be a factor of k_0. If it is, it will be a common divisor of the first row of $\mathbf{Q}^T$, not allowed by Condition 1.

So it should be a factor of b_1. Thus $R_2 = k_0 k_1 + dZ$. If d divides R_2, it must be a factor of k_1 and must belong to f. But, this is impossible, because if it is, d is also a common divisor of the second row of $\mathbf{Q}^T$, violating Condition 1.

In summary, we cannot find a common divisor for R_0, R_1 and R_2 without violating Condition 1. □

To avoid any invalid **Q**, consider the two theorems. We take the form of a diagonal matrix for **A**, adding non-zero element f at the position of $a_{i,j}$ if GCD$(k_i, k_j) > 1$. That is eqn (3.3). Observe that these results can be extended to n-D dimension problem. but the discussion is rather lengthy and is omitted.

Valid $\bar{\mathbf{t}}$

Condition 3 is required for physical implementation of intercommunication according to the criterion **SD** = **PK**. Let $\mathbf{D}^p = \mathbf{SD}$ be the dependencies in the virtual projected array. Now, we should consider the data flows among processors after partitioning. Usually, the partitioning box is large enough to enclose the projected dependencies in all dimensions, i.e., $\forall i, 1 \leq i < n$,

$$k_i \geq \max_{j=0}^{n_d} \mid d^p_{i,j} \mid$$

where $d^p_{i,j} \in \mathbf{D}^p$ and n_d is the number of columns of **D**. Thus, in this case, the data flows among processors only one step in any direction. We can construct the dependency matrix $\mathbf{D}^A$ of the processor array by just assigning $d^A_{i,j} = \frac{d^p_{i,j}}{|d^p_{i,j}|}$, where $d^A_{i,j} \in \mathbf{D}^A$ (note that $\mathbf{D}^p$ is different from $\mathbf{D}^A$: the former is the dependency in the mapped quasi-processor domain, while the latter is that in the real processor domain). So, **SD** = **PK** becomes $\mathbf{D}^A = \mathbf{PK}$. $\forall \mathbf{d}^A_i \in \mathbf{D}^A$, enumerate the number k_i of $\mathbf{p}$'s $\in \mathbf{P}$ to implement $\mathbf{d}^A_i$. Let $\bar{\mathbf{k}} = [k_0, \ldots, k_{n_d-1}]$, which is the lower boundary for the time to carry out the intercommunication among processors.

3.2.4 An Example

Suppose that we map a computational graph

$$\mathbf{V} = \begin{bmatrix} 0 & 20 & 0 & 0 & 0 & 20 & 20 & 20 \\ 0 & 0 & 20 & 0 & 20 & 0 & 20 & 20 \\ 0 & 0 & 0 & 10 & 10 & 10 & 0 & 10 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$$

onto an processor array of size 4×4 with interconnection primitives $\mathbf{P} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$.

Partitioning and mapping

For simplicity, we choose $\mathbf{s}^p = [1,1,1]^T$. Then the $\mathbf{S}$ should be

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \quad \mathbf{W} = \mathbf{SV} = \begin{bmatrix} 0 & 20 & 0 & -10 & -10 & 10 & 20 & 10 \\ 0 & 0 & 20 & -10 & 10 & -10 & 20 & 10 \end{bmatrix}$$

The matrix on the right-hand side gives the vertices of the projected virtual array. The desired compression factors are 8, 8. This means that the nodes of the virtual array in a 8×8 partitioning box will be allocated into one processor. The dependencies in the virtual array are

$$\mathbf{D}^p = \mathbf{SD} = \begin{bmatrix} 4 & -1 & 1 & 0 \\ 2 & 0 & 2 & 1 \end{bmatrix}$$

It can be understood that the 8×8 blocking box is large enough so that no dependency vectors in the projected virtual array (i.e., the column vectors of $\mathbf{SD}$) can penetrate through the partitioning box. Therefore

$$\mathbf{D}^A = [\mathbf{d}_0^A, \mathbf{d}_1^A, \mathbf{d}_2^A, \mathbf{d}_3^A] = \begin{bmatrix} 1 & -1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

For each column of $\mathbf{D}^A$, each non-zero entries must be implemented by a $\mathbf{p} \in \mathbf{P}$. For instance, $\mathbf{d}_0^A$ are implemented by $\mathbf{p}_0 \oplus \mathbf{p}_1$, where $\oplus$ means "direct sum", and $\mathbf{p}_0 = [1,0]^T$ and $\mathbf{p}_1 = [0,1]^T$ are interconnection primitives. Thus $\overline{\mathbf{k}}_p = [2,1,2,1]$.

According to eqn (3.3) and (3.6), the $\mathbf{A}$ is constructed, as well as $\mathbf{Q}_p$ and $\mathbf{q}^n$

$$\mathbf{A} = \begin{bmatrix} 8 & 1 \\ 0 & 8 \end{bmatrix} \quad \mathbf{Q}_p = \begin{bmatrix} 8 & 1 \\ 0 & 8 \\ 0 & 0 \end{bmatrix} \quad \mathbf{q}^n = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

It is known that Condition 1 is necessary and sufficient to obtain the valid $\mathbf{Q}$. In practice, Condition 1 is easily satisfied. However, Condition 3 for a valid $\overline{\mathbf{t}}$ is non-trivial. In the example, 16 valid $\mathbf{Q}$ are tested before a valid $\overline{\mathbf{t}} = [40, 19, 5]$, with a $\mathbf{b}^T = [5,3]$, is found. The corresponding $\mathbf{Q}$ is

$$\mathbf{Q} = \begin{bmatrix} 3 & -2 \\ -5 & 5 \\ -5 & -3 \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 40 & 19 & 5 \end{bmatrix}$$

The matrix on the right-hand side is a space-time mapping matrix, for which it is easy to verify that $det(\mathbf{T}) = c = 64$.

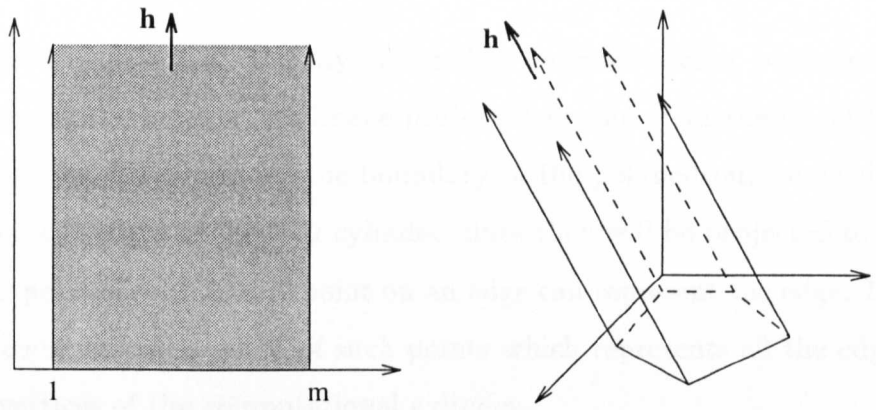


Figure 3.1: The Endless Cylinder for Computational Domain with One Infinite Index. **h** is the axis of the cylinder.

It is also easy to verify that $\mathbf{A} = \mathbf{SQ}$, so the 8×8 blocking box is ensured. Let diagonal matrix $\mathbf{F} = diag(\frac{1}{8}, \frac{1}{8})$. The LSGP compression stands for a transformation

$$\mathbf{W}' = \mathbf{FW} = \begin{bmatrix} 0 & 20/8 & 0 & -10/8 & -10/8 & 10/8 & 20/8 & 10/8 \\ 0 & 0 & 20/8 & -10/8 & 10/8 & -10/8 & 20/8 & 10/8 \end{bmatrix}$$

The $\mathbf{W}'$ has a coverage of 3.875×3.875, just within the given regular processor array. In addition, $\overline{\mathbf{t}}\mathbf{D} = [6, 24, 14, 19]$. Compared with $\overline{\mathbf{k}}_p = [2, 1, 2, 1]$, as expected, it gives enough time to carry out data communication relays.

3.2.5 A Particular Area of Application

An important aspect of application of so-called systolic arrays is signal processing, which is characterised by one semi-infinite index, say t, of iteration. For example, a recursive filter is expressed by

$$y(t) = x(t) + \sum_{j=1}^{m} a_j y(t-j), \qquad t = 0, 1, \ldots$$

see Figure 3.1.(a).

Generally, such kinds of computational domain take the shape of multi-facet N-D cylinder. Because the cylinder is straight, it is always possible to find an axis, **h** parallel to the sides. Our task is to partition and map the infinite computational polyhedron onto a (N-1)-D processor array.

One point is obvious that the space mapping vector, $\mathbf{s}^p$, must be parallel to **h**. If not, the infinite polyhedron will be projected to an infinite virtual array which cannot

be realised in practice. Usually, an endless cylinder cannot be expressed directly by a set of vertices. However, its image projected by the $\mathbf{s}^p$ in the (N-1)-D space domain is a polyhedron. To determine the boundary of the polyhedron, we need consider only the points on the edges of the N-D cylinder, since they will be projected to the vertices of the (N-1)-D polyhedron. So any point on an edge can represent the edge. In this sense, when $\mathbf{h}$ is known, we call a set $\mathbf{V}$ of such points which represents all the edges of the cylinder as the vertices of the computational cylinder.

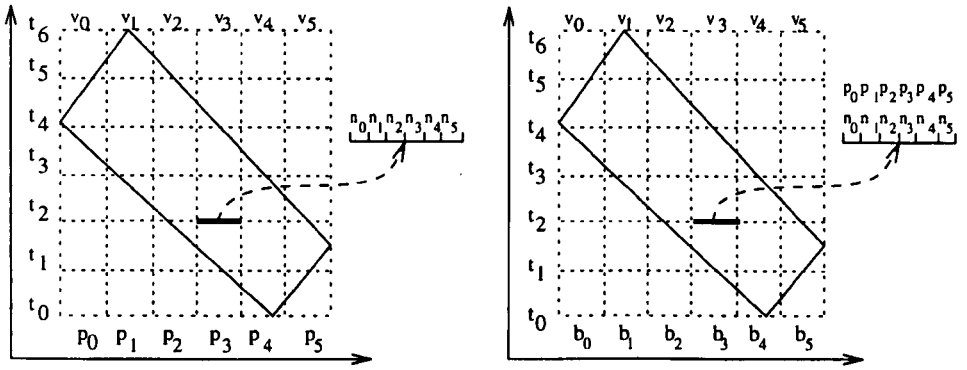
This constraint of $\mathbf{s}^p \parallel \mathbf{h}$ presents a difficulty in applying the supernode partitioning. Because in the supernode partitioning, the space mapping vector is either one of the edges or the diagonal of the supernode parallelepiped which is determined by other requirements, it usually diverges from the axis of the polyhedron. Furthermore, in signal processing, sometimes, it is demanded that the processing system produces an output corresponding to an input with certain delay. This means that the processor array must produce outputs once after finishing each "thin" time hyperplane. However, by any supernode partitioning, the time hyperplane becomes "thick", which implies block-data processing.

Our LSGP method is applicable to this problem, because it does not involve any partitioning similar to supernode partitioning. The only additional operation is to introduce a linear transformation $\mathbf{B}$ which is unimodular such that $s^b_{N-1} = 1$, where $s^b_{N-1} \in \mathbf{s}^b = \mathbf{B}^{-1}\mathbf{s}^p$, where $\mathbf{s}^b$ is the projection vector in the space with $\mathbf{B}$ as the basis. It is easy to find $\mathbf{B}$. In fact, since $\mathrm{GCD}(\mathbf{s}^p) = 1$, it is always possible to find $\overline{\mathbf{b}}_0$ such that $\overline{\mathbf{b}}_0\mathbf{s}^p = 1$. Then $\overline{\mathbf{b}}_0$ can be extended to an unimodular matrix by the method proposed in [107].

Apply the transformation on the given computation graph $(\mathbf{V}, \mathbf{D})$,

$$\mathbf{V}^b = \mathbf{B}^{-1}\mathbf{V}, \quad \text{and} \quad \mathbf{D}^b = \mathbf{B}^{-1}\mathbf{D}.$$

As for $\mathbf{s}^b$, the superscript "b" means in the space with $\mathbf{B}$ as the basis. The projection and partitioning of **Step 1** to **Step 6** is performed in the space spanned by $\mathbf{B}$.



(a) LSGP. (b) LPGS.

Figure 3.2: Demonstration of LSGP and LPGS. Each segment consists of 6 nodes.

3.3 Bouncing LPGS Method

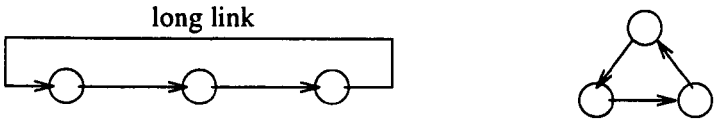
3.3.1 Introduction

The disadvantage of the LSGP methods in general is that when an irregular computational polyhedron is mapped within a regular processor array, there will be a great waste of the computational resource at the edges. The higher the dimensions, the larger the overall waste will be. The LPGS method in [73] is superior with regard to this problem.

For example, suppose a 2-D problem is implemented by a linear array with 6 processors, see Figure 3.2. In LSGP, the 6 nodes in a segment are carried out sequentially by one processor and one processor is responsible for one column (for instance, processor p_0 is for column v_0), while in LPGS the 6 nodes are calculated in parallel by 6 processors, one segment after another from left to right, from bottom to top. In the case of LSGP, at time t_5, processors p_0, p_1 and p_2 are busy, while p_3, p_4 and p_5 are idle, but we cannot enter the next time hyperplane t_6 until p_0, p_1 and p_2 finish their jobs on t_5. In contrast, in LPGS, there is no computation in b_3, b_4 and b_5, so we can jump from point (t_5, b_2) directly to point (t_6, b_1), which is more efficient.

However, as mentioned before, one of the main disadvantages is the need for additional long distance buffering hardware which seriously restricts its application. It can be seen from Figure 3.3.(a) and (c) that the data flow from band b_0 to band b_1 requires a long link from processor p_2 to processor p_0. This kind of long link is not always available in a

given regular processor array, or covers large area when manufacturing VLSI chips. One might try naively to use a clever re-arrangement of processors in the architecture,



but this is not always possible in higher dimensional structures.

3.3.2 Bouncing LPGS

We can address the problem above by introducing another kind of allocation strategy, the so-called Bouncing LPGS. The method is shown in Figure 3.3.(b) and (d). When mapping band b_1, the allocation of nodes is just the reverse-order of band b_0, so that the transition from one band to its next results in a "data flow" inside a processor, instead of a long distance transfer. Data flows from left to right in even-labeled bands, then does the reverse in the odd-labeled bands. In effect this is like a ball bouncing between two ends of a sealed tube. The precondition for implementing the Bouncing LPGS is a bi-connected interconnection mesh.

The allocation function of the Bouncing LPGS in the example is expressed as

$$A(\mathbf{i}^p) = \begin{cases} i' & i' < l \\ 2l - i' & l \leq i' \end{cases} \tag{3.11}$$

where $\mathbf{i}^p = [i^p] = \mathbf{Si}$ is a node in the projected space determined by the transformation T, l is the number of processors in the linear array and $i' = \mathbf{i}^p \bmod 2l$.

This method can be extended straightforwardly to n dimensions. $A(\mathbf{i}^p) = [i^a_1, \ldots, i^a_{N-1}]^T$, $\mathbf{i}^p = [i^p_1, \ldots, i^p_{N-1}]^T$, where

$$i^a_i = \begin{cases} i'_i & i'_i < l \\ 2l - i'_i & l \leq i'_i \end{cases} \qquad \text{and } i'_i = i^p_i \bmod 2l_i \tag{3.12}$$

Although LPGS (including the improved LPGS's) gives the potential to be more efficient in the cases of irregular computational polyhedron, more run-time controls are required to achieve the goal. As mentioned, in figure 3.2.(b), for instance we can jump from (t_5, b_2) directly to (t_6, b_1). But the jump will result in a serious confusion for data dependencies in time domain since such jumps happen without regularity, depending only

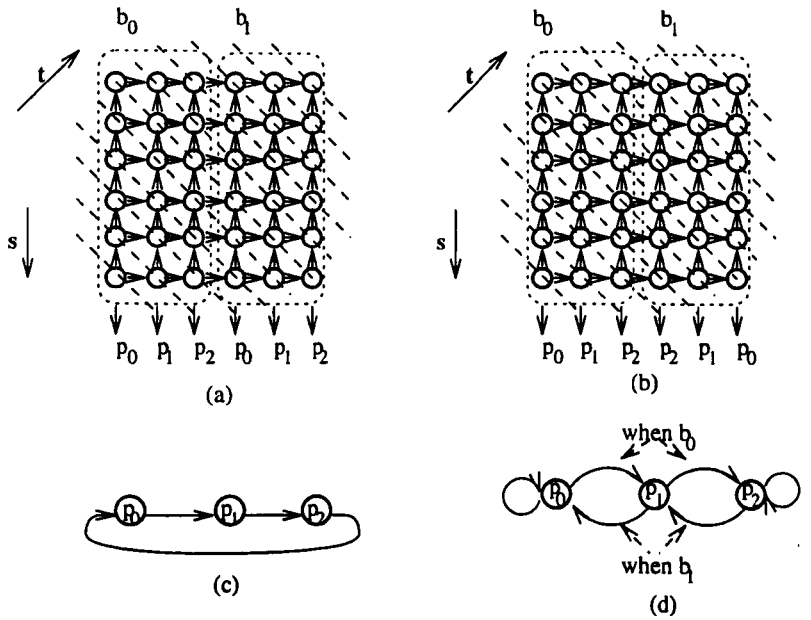


Figure 3.3: LPGS and Bouncing LPGS. Vector t is the normal of time hyperplane and s^p, the vector of space mapping. (a) LPGS allocation of nodes to processors; (b) Bouncing LPGS allocation of nodes to processors; data flow of LPGS; (d) data flow of Bouncing LPGS

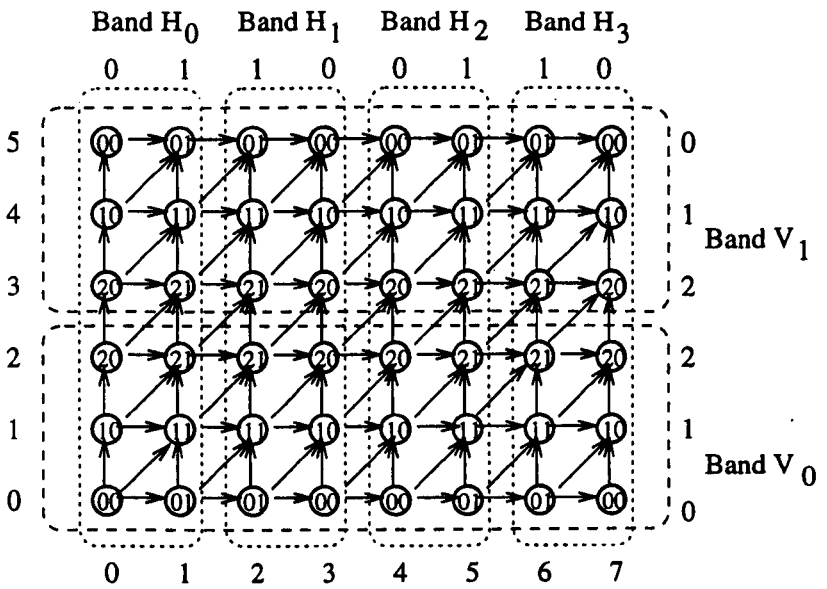


Figure 3.4: One time hyperplane of 2-D Bouncing LPGS. The intersections of Band H's and Band V's are regions which the processor array computes at one moment. The number in each node indicates its allocation in a 3×2 array.

on the boundary of the mapped polyhedron. For an ordinary LPGS, for example, we can determine a fixed time interval when a datum produced in a processor will be used in another processor . It will be very difficult to determine the time interval with the unpredictable jumps. Thus some control mechanisms have to be added and this remains an open problem for actual application.

3.4 Summary

In the chapter, we address improvements of three existing methods of partitioning. Firstly, based on the theoretical framework of [94] a new algorithm for HSNF and SNF is developed which reduces the computation complexity significantly for the independent partitioning problem. More importantly, we propose a method of performing algorithm transformation so that the real-time partitioning is systematically carried out instead of collecting computations node by node, finally parallel codes are generated systematically.

Secondly a method is proposed to modify the LSGP partitioning in [24], which guarantees to obtain the given-regular-shape mapping easily in principle. Furthermore, due to the proper design of two key matrices, the computation process is easily carried out. The method exhibits fundamental advantages over Darte's method, such as no need to compute HNF's and the production of a given regular array.

Finally a Bouncing LPGS is proposed to improve the widely known LPGS partitioning method [73]. The Bouncing LPGS is characterised by mapping each consecutive band in two opposite directions so that the data flow will go forwards and backwards in turn in the processor array like a bouncing ball, therefore the long distance links from one side of the array to the other are no longer needed.

Chapter 4

A Methodology of Partitioning and Mapping for Given (N-1)-D Regular Arrays

In this chapter a methodology is proposed to partition and map an arbitrary computation graph onto a given regular M-D array under the URE condition, where M = N - 1. We start with a positive expressing basis to obtain a set of canonical dependencies. Based on these canonical dependencies, two basic models of space mappings, as well as, timing vectors are derived. The partitioning parallelepipeds are scaled to map the original polyhedron into a given array, then an LSGP method is used to improve efficiency. This methodology has significant advantages in mapping an irregular computation graph onto a given regular processor array while having high efficiency in both communication and computation.

4.1 A New Methodology

We found that mainly due to the difficulty of fixed-size mapping, the Partitioning-Projecting method has not received enough attention compared with Projecting–Partitioning. But if we have some kind of information about the space mapping before the space projection actually takes place, it is possible to scale firstly the partitioning so that the partitioned space can be mapped within a given-size array. This suggests that we may begin with a set of standard dependencies and a set of fixed interconnections to exploit the possible space mappings and find a way of scaling the partitioning.

We extend C.T.King's concept [52] to form a supernode with N partitioning vectors

such that the entries of a supernode dependency matrix are only 0 or 1, that is, the dependency of a supernode is very local and is orientated along the positive direction in every dimension. This allows us to produce a canonical dependency matrix, independent of the dependencies in the problem being considered. With the canonical dependency matrix and two basic interconnection meshes, we are able to consider two models of space mapping. They are the simplest mappings possible but possess the properties we require.

With the two space mapping models, we can predict the projected image of a computational polyhedron. The lengths of the edges (the N partitioning vectors) of the parallelepiped are determined such that the resulting supernode polyhedron can be allocated within a given size array, and the canonical dependencies can be ensured. Thus, we obtain an available partitioning strategy to map the original computation polyhedron onto a given-regular-shape and fixed-interconnection array. Such a mapping will usually have a low computational efficiency. To improve efficiency, the LSGP method is employed to partition the supernode space further. By exploiting the knowledge of the first supernode partitioning, the procedure for implementing the second (LSGP) becomes relatively easy, and can even be embedded into the first partitioning. The two basic space mapping matrices are not unique and their permuted versions can be utilised for optimisation purposes. See Figure 4.1.

In the figure, the original polyhedron is partitioned to a supernode polyhedron with canonical dependencies. Because the supernode polyhedron may be larger than to be required, it has to be partitioned further (actually the two partitionings are carried out together) so that the resulting supernode polyhedron can be mapped within the processor array (or for the case of SBC the resulting supernode polyhedron is mapped within an EVPA and then the EVPA is LSGP partitioned into the processor array).

The methodology in this chapter focuses on the derivation of two integral transformation matrices, **B** and **T**, and an integer vector **k**. **B** is a transformation from the original computation space to a supernode space. **T** is a transformation from the supernode space to a 1-D time domain and a M-D Enlarged Virtual Processing Array (EVPA) domain. The quantity **k** is an LSGP compression vector containing compression factors for all dimensions and allows the LSGP partitioning to compress the EVPA onto a given regular

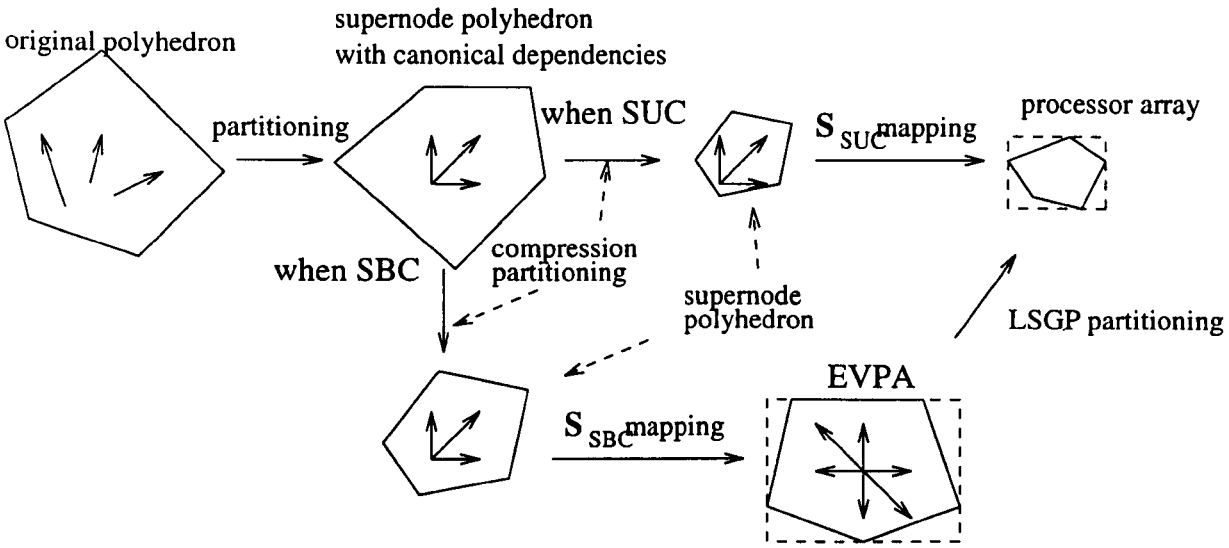


Figure 4.1: The Basic Idea of Our Partitioning and Mapping Method. The arrows within each polyhedron indicate the pattern of dependency vectors

processor array.

The rest of the chapter is organised as follows: first the construction of a canonical dependency matrix is discussed, second the selection of the two space mappings and their timing vectors. Next LSGP partitioning, scaling the supernode parallelepiped and integralization of the partitioning matrix are discussed. Finally we consider some examples to illustrate the method.

4.2 A Transformation for Canonical Dependencies

4.2.1 A Cone Including Dependencies

It is shown [52] that in a 2-D computation space, any vectors in a set of dependency vectors can be expressed by a positive linear combination of two vectors from the set. This can be seen clearly by a geometrical explanation. All of the dependency vectors form a 2-D cone whose two extreme edges are two of the dependency vectors, see Figure 4.2, where $\mathbf{d}_0$ and $\mathbf{d}_3$ form a two-edge cone including all dependency vectors. As a result, any vector within the cone can be expressed by a positive linear combination of the two edges. If example, let $\mathbf{d}_0 = [2, -3]^T$, $\mathbf{d}_2 = [3, 2]^T$ and $\mathbf{d}_3 = [1, 3]^T$. It can be checked that $\mathbf{d}_2 = \frac{7}{9}\mathbf{d}_0 + \frac{13}{9}\mathbf{d}_3$. The two edges which are linearly independent, can be used as a basis of

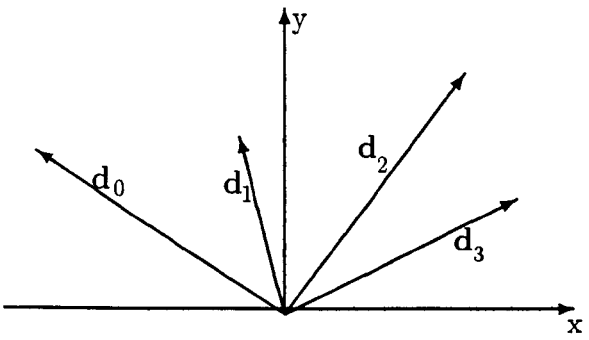


Figure 4.2: The Cone of Dependency Vectors.

a linear transformation, so that in the resulting space any dependency vectors have only positive components, an important feature that we will make use of later.

Generally, in N-D space we cannot always form a N-edge cone from the set of dependency vectors to include all other dependency vectors. For example, in 3-D space, if there are four dependency vectors which form a 4-edge convex cone, no three of them can form a 3-edge cone which includes the fourth.

However, as we prove later, it is always possible to find N lexicographically positive vectors in N-D space such that any of dependency vectors can be expressed by a positive linear combination of them.

Definition 4.2.1 *A vector* $\mathbf{v} = [v_0, \cdots, v_n]$ *is termed to be i-th unit lexicographically positive (simply i-th ULP) if* $v_i = 1$ *and* $\forall j < i$: $d_j = 0$. *For example* $[0, \cdots, 0, 1, v_{i+1}, \cdots, v_n]^T$, *where* v_j, j = *i+1,...,n, is any integer.*

Theorem 4.2.1 *For a set of any N-D dependency vectors, there exists at least one set of N lexicographically positive vectors, called a Positive Expressing Basis (PEB)* **E**, *such that any of the dependency vectors can be expressed by a positive linear combination of them.*

Proof: The proof of the theorem is also the procedure to find the positive expressing basis. Without loss of generality, suppose the first positive component of any dependency vector $d_i = 1$. If this is not the case, the vector can be normalised (that is, divide the entries of the vector with d_i. Some of entries of the normalized vectors may become non-integer, this does not affect the proof of the theorem). A N-D dependency matrix can

be decomposed into N submatrices, each of which is i-th ULP, called submatrix i, i = 0,$\cdots$,M.

The proof proceeds by induction. The first k vectors of **E** are supposed to be k column vectors in i-th ULP form, i = 0,$\cdots$,k-1, such that the submatrices 0,...,k-1 are positively expressed by them. Now, It is sufficient to show that the submatrices 0,$\cdots$,k can be positively expressed by k+1 ULP vectors. This involves constructing the k-th ULP vector, merged to the existing k vectors, so as to express the submatrix k. For the sake of brevity, in the derivation of the k-th ULP vector below, we omit the first N-k zero elements in each column vector, eg, $[0,\cdots,0,d_{N-k-1},\cdots,d_{N-1}]^T$ is expressed by $[d_{N-k-1},\cdots,d_{N-1}]^T$. So the submatrix k will be expressed as

$$\begin{bmatrix} 1 & \cdots & 1 \\ d_{k-1,0} & \cdots & d_{k-1,l-1} \\ \vdots & \cdots & \vdots \\ d_{0,0} & \cdots & d_{0,l-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ e_{k,k-1} & 1 & 0 & \cdots & 0 \\ \vdots & e_{k-1,k-2} & 1 & \cdots & 0 \\ \vdots & \vdots & e_{k-2,k-3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & 0 \\ e_{k,0} & e_{k-1,0} & e_{k-2,0} & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} a_{k,0} & \cdots & a_{k,l-1} \\ a_{k-1,0} & \cdots & a_{k-1,l-1} \\ a_{k-2,0} & \cdots & a_{k-2,l-1} \\ \vdots & \cdots & \vdots \\ a_{0,0} & \cdots & a_{0,l-1} \end{bmatrix} \tag{4.1}$$

where the matrix on the left-hand side is the submatrix k with l vectors. The first matrix on the right-hand side contains the first k+1 vectors of the **E** in which the first column (the k-th ULP vector) is to be determined, and the second matrix $[a_{i,j}]$ on the right-hand side is the coefficient matrix for the positive combination, which is also to be determined.

All entries in the coefficient matrix must be positive as required by the definition, and all elements of the first row, $a_{k,0},\cdots,a_{k,l-1}$, must be 1. In fact, considering eqn(4.1), it can be seen that the first row of the left-hand side matrix is the product of the first row of the middle matrix and the right-hand side matrix. But since the first row of the left-hand side matrix is [1 0 $\cdots$], we have $[1\ \cdots\ 1] = [a_{k,0}\cdots a_{k,l-1}]$.

Next, choose $e_{k,k-1}$ and determine $a_{k-1,0},...,a_{k-1,l-1}$. Let

$$e_{k,k-1} = \min_{0 \le i \le l-1} (\lfloor d_{k-1,i} \rfloor, 0) \tag{4.2}$$

then

$$a_{k-1,i} = d_{k-1,i} - e_{k,k-1} \qquad i = 0,...,l-1 \tag{4.3}$$

because $e_{k,k-1} \le d_{k-1,i}$, $a_{k-1,i} \ge 0$.

Now, the coefficients with respect to the (k-1)-th ULP vector in the subset are known, and we can modify the submatrix k such that the contribution of the (k-1)-th ULP vector is eliminated. The modified submatrix k can be expressed as

$$\begin{bmatrix} 1 & \cdots & 1 \\ e_{k,k-1} & \cdots & e_{k,k-1} \\ d^1_{k-2,0} & \cdots & d^1_{k-2,l-1} \\ \vdots & \cdots & \vdots \\ d^1_{0,0} & \cdots & d^1_{0,l-1} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ d_{k-1,0} & \cdots & d_{k-1,l-1} \\ \vdots & \cdots & \vdots \\ d_{0,0} & \cdots & d_{0,l-1} \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ e_{k-1,k-2} \\ \vdots \\ e_{k-1,0} \end{bmatrix} \begin{bmatrix} a_{k-1,0} \\ \vdots \\ a_{k-1,l-1} \end{bmatrix}^T$$

$$= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ e_{k,k-1} & 0 & 0 & \cdots & 0 \\ e_{k,k-2} & 1 & 0 & \cdots & 0 \\ \vdots & e_{k-2,k-3} & 1 & \cdots & 0 \\ \vdots & \vdots & e_{k-3,4} & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & 0 \\ e_{k,0} & e_{k-2,0} & e_{k-3,0} & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} a_{k,0} & \cdots & a_{k,l-1} \\ a_{k-2,0} & \cdots & a_{k-2,l-1} \\ \vdots & \cdots & \vdots \\ a_{0,0} & \cdots & a_{0,l-1} \end{bmatrix}$$

and we proceed similarly to determine $e_{k,k-2}$ and $a_{k-2,0} \cdots a_{k-2,l-1}$. Repeat the process until $e_{k,0}$ is produced.

If the submatrix k is empty, let $e_{k,k-1} = \cdots = e_{k,0} = 0$. □

Let the final coefficient matrix be $\mathbf{A}^p$. Obviously, $\mathbf{D} = \mathbf{E}\mathbf{A}^p$. Theorem 4.2.1 implies that by a linear transformation with E the dependencies in the new space consist of only positive components, and, as will be seen later, this turns out to be quite and important property.

Example 4.2.1 *Derive* **B** *from a* **D**

Suppose a given dependency matrix **D** and decompose it as follows:

$$\underset{\mathbf{D}}{\begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ 0 & -2 & 1 & -3 \end{bmatrix}} \rightarrow \underset{\text{submatrix 2}}{\begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 0 & -3 \end{bmatrix}} \oplus \underset{\text{submatrix 1}}{\begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix}} \oplus \underset{\text{submatrix 0}}{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}$$

Obviously, the 0-th column of **E** should be $[0,0,1]^T$. An equation can be formed to determine the 1st vector of **E**

$$\begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ e_{1,0} & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ a_{0,0} \end{bmatrix}$$

Let $e_{1,0}$ = -2, then $a_{0,0}$ = 0.

In the similar way, the 2nd vector of **E** can be determined according to the submatrix 2.

$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 0 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ e_{2,1} & 1 & 0 \\ e_{2,0} & -2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ a_{1,0} & a_{1,1} \\ a_{0,0} & a_{0,1} \end{bmatrix}$$

Let $e_{2,1}$ = -1, then $[a_{1,0},a_{1,1}]$ = [0,2]. So the submatrix 2 is modified as

$$\begin{bmatrix} 1 & 1 \\ -1 & -1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ 0 & -3 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ e_{2,0} & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ a_{0,0} & a_{0,1} \end{bmatrix}$$

Let $e_{2,0} = 0$, then $[a_{0,0}, a_{0,1}] = [0,1]$. The procedure of producing the **E** is finished. As a result, we obtain

$$\mathbf{E} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{A}_p = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

It can be checked that $\mathbf{D} = \mathbf{E}\mathbf{A}_p$.

4.2.2 Forming Canonical Dependencies in Supernode Space

The vectors in **E** indicate the directions of the edges of a clustering parallelepiped for grouping the computation graph nodes into supernodes.

Definition 4.2.2 *A quasi-supernode space $S^q \in R^N$, indexed by* $\mathbf{q}$, *is an image of an integral space by a mapping of* $\mathbf{M}^{-1}$ *and* $det(\mathbf{M}) > 1$. *All integral points in the quasi-supernode space form a subspace of S^q, termed a supernode space S^s and indexed by* $\mathbf{s}$.

The quasi-supernode space is a compressed version of the original computational space which allows partitioning along its coordinate axes. It remains to show that the dependencies in the supernode space can have a canonical form.

We are interested in a pair consisting of a PEB matrix and its corresponding coefficient matrix, $\mathbf{B}_d$ and $\mathbf{A}^d$, such that

$$\mathbf{D} = \mathbf{B}_d \mathbf{A}^d \tag{4.4}$$

and $0 \leq a^d_{i,j} \leq 1, \ \forall a^d_{i,j} \in \mathbf{A}^d$.

It is easy to determine a positive diagonal matrix $\mathbf{F}_d = diag(f^d_0, \cdots, f^d_M)$ ($\mathbf{F}_d^{-1} = diag(1/f^d_0, \cdots, 1/f^d_M)$) such that

$$\mathbf{D} = \mathbf{E}\mathbf{F}_d^{-1}\mathbf{F}_d\mathbf{A}_p = (\mathbf{E}\mathbf{F}_d^{-1})(\mathbf{F}_d\mathbf{A}_p) = \mathbf{B}_d\mathbf{A} \tag{4.5}$$

where $\mathbf{B}_d = \mathbf{E}\mathbf{F}_d^{-1}$ and $\mathbf{A} = \mathbf{F}_d\mathbf{A}_p$. In fact, $\forall i, 0 \leq i < N$

$$a^d_{i,j} = f^d_i a^p_{i,j}, \forall j, \ \ 0 \leq j < n_d$$

where $a^d_{i,j} \in \mathbf{A}^d$, $a^p_{i,j} \in \mathbf{A}^p$. Let $a^{max}_i = \max_{0 \leq j < n_d} a^p_{i,j}$. Then,

$$f^d_i \leq \frac{1}{a^{max}_i} \tag{4.6}$$

Let $\overline{\mathbf{a}}^{max} = [a^{max}_0, \cdots, a^{max}_M]$.

We make parallelepipeds with the columns of $\mathbf{B}_d$ as their edges to partition the computation polyhedron. To cluster more than one node into a partition, $\det(\mathbf{B}_d) > 1$. This is equivalent to a linear transformation with $\mathbf{B}_d$ as the basis from the original computational space S^c to a quasi-supernode space S^q, so $\mathbf{B}_d$ can be called the basis of the quasi-supernode space. For any node $\mathbf{i} \in S^c$

$$\mathbf{q} = \mathbf{B}_d^{-1}\mathbf{i} \tag{4.7}$$

where $\mathbf{q} \in S^q$ is the image of $\mathbf{i}$. In S^q, the mapped dependency matrix $\mathbf{D}^q$ is exactly $\mathbf{A}^d$, i.e.,

$$\mathbf{D}^q = \mathbf{A}^d = \mathbf{B}_d^{-1}\mathbf{D} \tag{4.8}$$

So we have

$$0 \leq d^q_{i,j} \leq 1; \tag{4.9}$$

where $d^q_{i,j} \in \mathbf{D}^q$. Because $\det(\mathbf{B}_d)$ is larger than 1, the nodes of the original space may be mapped to non-integral points of S^q. In S^q, collect all projected nodes $\mathbf{q} = [q_0, \cdots, q_M]^T$ in a hypercube

$$s_j \leq q_j < s_j + 1; \qquad s_j \in Z; j = 0, \cdots, M \tag{4.10}$$

to the integral point $\mathbf{s} = [s_0, \cdots, s_M]^T$, $\mathbf{s}$ will be a supernode, including all the projected nodes in the hypercube. Any projected node can be re-expressed as

$$\begin{aligned} \mathbf{q} &= \mathbf{s} + \mathbf{\Delta}; \\ 0 &\leq \Delta_j < 1; \qquad j = 0, \cdots, M \end{aligned} \tag{4.11}$$

where $\mathbf{\Delta} \in R^N$ and $\Delta_j \in \mathbf{\Delta}$.

Now, we can derive the dependency relations among the supernodes. In the original computational space, any dependency relation between two nodes can be expressed as

$$\mathbf{i}^2 = \mathbf{i}^1 + \mathbf{d}_i \tag{4.12}$$

where $\mathbf{d}_i \in \mathbf{D}$. Applying the linear transformation of eqn(4.7) to eqn (4.12) produces

$$\mathbf{B}_d^{-1}\mathbf{i}^2 = \mathbf{B}_d^{-1}\mathbf{i}^1 + \mathbf{B}_d^{-1}\mathbf{d}_i, \qquad \mathbf{q}^2 = \mathbf{q}^1 + \mathbf{d}^q_i \tag{4.13}$$

where $\mathbf{q}^2$ and $\mathbf{q}^1 \in S^q$; $\mathbf{d}^q_i \in \mathbf{D}^q$. Consider eqn (4.9) and eqn (4.11), we have

$$\begin{aligned} q^2_j &= s^2_j + \Delta^2_j \\ &= s^1_j + \Delta^1_j + d^q_{ij}; \qquad j = 0, \cdots, M \end{aligned} \tag{4.14}$$

where $q^2_j \in \mathbf{q}^2$, $s^1_j \in \mathbf{s}^1$, $s^2_j \in \mathbf{s}^2$, and $d^q_{ij} \in \mathbf{D}^q$. $\mathbf{s}^1$ and $\mathbf{s}^2$ are two supernodes.

Let $x_j = \Delta^1_j + d^q_{ij}$. Because $0 \leq x_j < 2$, $s^2_j = s^1_j$ if $x_j < 1$ and $s^2_j = s^1_j + 1$ if $x_j \geq 1$. Therefore, we can list all possible dependency vectors $\mathbf{d}^s_j$, $j = 1, \cdots, 2^M$ in S^s. For example, N = 3,

$\mathbf{d}^s_1 = [0,0,1]^T$; $\quad x_2 < 1, \ x_1 < 1, \ x_0 \geq 1$
$\mathbf{d}^s_2 = [0,1,0]^T$; $\quad x_2 < 1, \ x_1 \geq 1, \ x_0 < 1$
$\mathbf{d}^s_3 = [0,1,1]^T$; $\quad x_2 < 1, \ x_1 \geq 1, \ x_0 \geq 1$
$\mathbf{d}^s_4 = [1,0,0]^T$; $\quad x_2 \geq 1, \ x_1 < 1, \ x_0 < 1$
$\mathbf{d}^s_5 = [1,0,1]^T$; $\quad x_2 \geq 1, \ x_1 < 1, \ x_0 \geq 1$
$\mathbf{d}^s_6 = [1,1,0]^T$; $\quad x_2 \geq 1, \ x_1 \geq 1, \ x_0 < 1$
$\mathbf{d}^s_7 = [1,1,1]^T$; $\quad x_2 \geq 1, \ x_1 \geq 1, \ x_0 \geq 1$

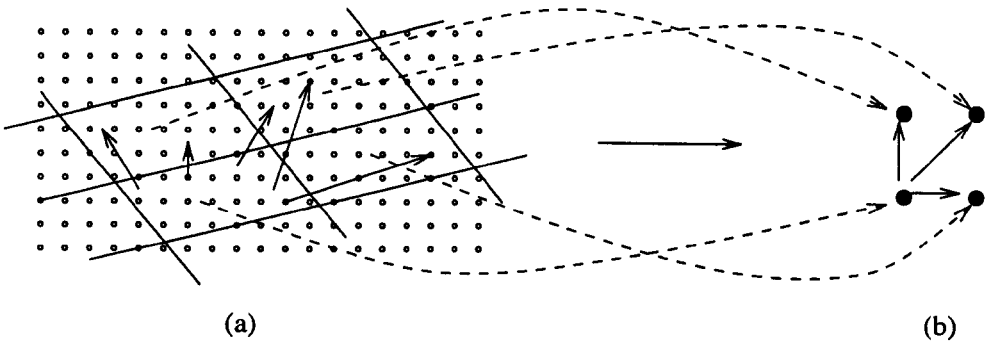


Figure 4.3: Supernode Partitioning and Canonical Dependencies. (a) the original domain with irregular dependencies is partitioned by partitioning parallelepipeds, (b) the supernode domain with canonical dependencies

Collecting all $\mathbf{d}^s$'s into a matrix, we obtain the 3-D canonical dependency matrix

$$\mathbf{D}^s = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{4.15}$$

in supernode space. Note that there is no difference when permuting the columns of a dependency matrix. We have the following general theorem

Theorem 4.2.2 *By the transformation* $\mathbf{B}_d$, *all these dependency vectors in* S^s *form a canonical dependency matrix* $\mathbf{D}^s$ *which looks like a table of binary numbers without* $\mathbf{0}$.

This theorem can be explained intuitively. By using $\mathbf{E}$ as the edges of the supernode parallelepiped, any components of dependencies in the quasi-supernode domain become positive. By scaling the size of the parallelepiped so that no components of the dependencies penetrate an adjacent parallelepiped, we guarantee that the components of dependencies between supernodes (parallelepipeds) can be only 0 or 1. Hence the theorem means that by partitioning with $\mathbf{B}_d$ any original arbitrary dependencies can be transformed to an universal set of canonical dependencies. See Figure 4.3.

The number of supernode dependencies is fixed as $2^N - 1$, independent of the original ones. Sometimes the number of supernode dependencies may be more than that of the original problem. This does not add to the complexity in the dependency problem. In

fact, the supernode dependencies rearrange the data dependencies more efficiently so that data flow paths are simplified for some data. For example, an original dependency vector $[2,4]^T$ becomes a combination of $[1,0]^T$, $[0,1]^T$ and $[1,1]^T$, so some data is routed locally and by point-to-point connections.

4.3 Selection of Space Projection and Timing Vector

4.3.1 S-T Transformation and Interconnection Primitives

The supernode space S^s behaves like an ordinary integral space, therefore we can use the space-timing mapping concept to transform it onto a 1-D time domain and a M-D space domain.

We assume that our computational resource consists of a mesh connected network of processing cells which form a regular array. In many applications, only simple patterns of interconnection primitives such as these are available. Therefore, we work with the two simple ones, SUC and SBC, mentioned in Chapter 1. It is obvious that if a computation can be implemented with the SUC and SBC patterns, it must be able to be carried out by using more complex patterns of interconnection primitive with less data relays provided that the complex pattern includes the SUC and SBC as a subgraph. Of course the SBC includes the SUC as a subgraph, but, because, there is a significant difference between the methods dealing with them, it is worthwhile discussing them individually.

This selection of networks is not arbitrary. In fact, they are just the uni-directional and bi-directional connections of a hypercube structure. Our task is not to invent new structures but to develop strategies based on the most widely available interconnection meshes.

The matrices for SUC and SBC can be expressed as a unit matrix and a directive sum of a unit matrix and a minus unit matrix, respectively, that is,

$$\mathbf{P}_{SUC} = \mathbf{I}_{M\times M} \tag{4.16}$$

$$\mathbf{P}_{SBC} = [\mathbf{I}_{M\times M} : -\mathbf{I}_{M\times M}] \tag{4.17}$$

where ":" indicates a partitioning of the matrix.

4.3.2 Choosing S and $\bar{t}$

The canonical dependency matrix is the base from which we can derive the timing vectors according to some criteria and conditions together with a space mapping which has a certain kind of canonical form.

At first, it is easy to see that for the canonical dependencies, the condition of eqn (1.5) is equivalent to

$$t_i > 0, \qquad \forall i, 0 \leq i < N \tag{4.18}$$

In fact, considering $\mathbf{D}^s$ in eqn(4.15) whose first N column vectors compose an identity matrix, we have $\bar{\mathbf{t}}\mathbf{D}^s = [t_0 \cdots t_{N-1}, d_i^t, \cdots]$, where the d_i^t's are a positive linear combination of $t_0 \cdots t_{N-1}$, which indicates that eqn (1.5) is equivalent to eqn (4.18). Another obvious fact is that for the $\mathbf{P}_{SUC}$ the corresponding space mapping $\mathbf{S}_{SUC}$ must be non-negative to satisfy the criterion $\mathbf{S_{SUC}D^S = P_{SUC}K}$. In fact, $\mathbf{S_{SUC}D^S = P_{SUC}K}$ is equivalent to $[\mathbf{S}_{SUC} : \mathbf{M}] = \mathbf{K}$, where $\mathbf{M}$ is a matrix. $\mathbf{S}_{SUC}$ must be non-negative because $\mathbf{K}$ is non-negative as required in [73], see page 21.

With the canonical Dependency matrix and the simple patterns of interconnections, the criterion of $\mathbf{SD^s = PK}$ can be simplified to the following theorem.

Theorem 4.3.1 *With the canonical Dependency matrix, the data flows can be implemented with the simple patterns of interconnections and arrive at the correct place at the correct time if and only if* $\forall j,\ j = 0, \cdots, M$

$$t_j \geq \sum_{i=0}^{M-1} |s_{i,j}| \qquad \textit{where } s_{i,j} \in \mathbf{S} \tag{4.19}$$

Proof: note that any $\mathbf{d}^s \in \mathbf{D}^s$ can be written as $\mathbf{d}^s = [d_0^s, \cdots, d_i^s, \cdots, d_M^s]^T$, where $d_i^s = 0, 1,\ \forall i,\ 0 \leq i < n$. The image of the transformation $\mathbf{T}$ in the time-space domain is

$$\mathbf{Td}^s = \begin{bmatrix} \Sigma_{i=0}^{M} s_{0,i} d_i^s \\ \vdots \\ \Sigma_{i=0}^{M} s_{M-1,i} d_i^s \\ \Sigma_{i=0}^{M} t_i d_i^s \end{bmatrix} = \sum_{j=0}^{M-1} (\Sigma_{i=0}^{M} s_{j,i} d_i^s)\mathbf{1}_j + (\Sigma_{i=0}^{M} t_i d_i^s)\mathbf{1}_M \tag{4.20}$$

where $\mathbf{1}_j$ is a vector such that its j-th element is 1, and zero otherwise. The first term in the right-hand side of eqn (4.20) is $\mathbf{Sd}^s$ and will be implemented by the column vectors

of $\mathbf{P}_{SUC}$ or $\mathbf{P}_{SBC}$. Because the columns of $\mathbf{P}_{SUC}$ and $\mathbf{P}_{SBC}$ are given in terms of $\mathbf{1}_j$ or -$\mathbf{1}_j$, m "vector-times" of the primitive vectors are used to implement the $\mathbf{Sd}^s$, where

$$m = \sum_{j=0}^{M-1} |\sum_{i=0}^{M} s_{j,i} d_i^s| \tag{4.21}$$

and one vector-time means to use a primitive vector once (for instance, if $\mathbf{Sd}^s = [1, -1]$, it will be implemented by $\mathbf{p}_0$ and $\mathbf{p}_3$ of $\mathbf{P}_{SBC}$ and the vector-times are two).

This means that a data flow indicated by $\mathbf{d}^s$ can be relayed k times to arrive where it will be used, via the available interconnection primitives of $\mathbf{P}_{SUC}$ or $\mathbf{P}_{SBC}$. So, to have enough time to perform the relays

$$\sum_{i=0}^{M} t_i d_i^s \geq m = \sum_{j=0}^{M-1} |\sum_{i=0}^{M} s_{j,i} d_i^s| \tag{4.22}$$

If eqn (4.19) holds, it can be deduced that

$$\begin{aligned} \sum_{i=0}^{M} t_i d_i^s &\geq \sum_{i=0}^{M} \sum_{j=0}^{M-1} |s_{j,i} d_i^s| = \sum_{j=0}^{M-1} \sum_{i=0}^{M} |s_{j,i} d_i^s| \\ &\geq \sum_{j=0}^{M-1} |\sum_{i=0}^{M} s_{j,i} d_i^s| \end{aligned}$$

Therefore the condition of eqn (4.19) is sufficient. Furthermore, it can be seen that the condition eqn (4.19) is also necessary. In fact, For all the dependencies $\mathbf{d}^s = \mathbf{1}_j, \forall j,\ 0 \leq j < n$, eqn (4.22) is equivalent to eqn (4.19) because

$$d_i^s = \begin{cases} 1 & \text{if i = j} \\ 0 & \text{otherwise} \end{cases}$$

, therefore any violation to eqn (4.19) will result in a violation to an eqn (4.22). □

Theorem 4.3.1 simplifies the criterion for selecting the space mapping $\mathbf{S}$ in our context, but there are still many choices. We need further criteria to limit the scope of the choices for the mapping. It is pointed out in [94] that for a fixed-size domain of computation, the total execution time is a monotonically increasing function of the absolute values of the entries of $\bar{\mathbf{t}}$. This is true when partitioning is not considered. However, in some circumstances, as will be seen later, we may enlarge the entries of $\bar{\mathbf{t}}$ to improve efficiency, i.e., reducing the total execution time.

However the principle can be used here. It is hoped that the norm of $\bar{\mathbf{t}}$ can be made as small as possible but with a lower bound given by eqn (4.18) and eqn (4.19), since a large positive $\bar{\mathbf{t}}$ will enlarge the extension of the mapping along the time domain. Furthermore it is clear that we need to construct $\mathbf{S}$ with as few non-zero entries as possible and with the smallest absolute value possible for the non-zero entries. Otherwise $\mathbf{S}$ may produce a large mapping extension in the space domain, as a result of which more nodes have to be squeezed into a supernode so that the supernode domain can be mapped into a given regular processor domain.

We choose $\mathbf{S}_{SUC}$ as

$$\mathbf{S}_{SUC} = [\mathbf{o}, \mathbf{I}_M] = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix} \tag{4.23}$$

The corresponding projection vector $\mathbf{s}^p_{SUC} = [1, 0, \cdots, 0]^T$, and the timing vector $\bar{\mathbf{t}}_{SUC}$ is chosen as

$$\bar{\mathbf{t}}_{SUC} = [1, \cdots, 1] \tag{4.24}$$

Clearly the pair of $\bar{\mathbf{t}}$ and $\mathbf{S}$ satisfies eqn (4.19), and possess the agreeable property that $e = \bar{\mathbf{t}}_{SUC}\mathbf{s}^p_{SUC} = 1$. The image of $\mathbf{D}^s$ in the domain mapped by the space mapping is (permuted),

$$\mathbf{S}_{SUC}\mathbf{D}^s = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{4.25}$$

when N = 3. This means that both the dependency vectors of $\mathbf{D}^s$, $[0, d_1, \cdots, d_M]^T$ and $[1, d_1, \cdots, d_M]^T$, must be implemented by one combination of the columns of $\mathbf{P}_{SUC}$.

We propose that $\mathbf{S}_{SBC}$ takes a form of

$$\mathbf{S}_{SBC} = [\mathbf{o}, \mathbf{I}_M] + [-\mathbf{I}_M, \mathbf{o}] = \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots \\ 0 & -1 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & -1 & 1 \end{bmatrix} \tag{4.26}$$

The image of $\mathbf{D}^s$ mapped by $\mathbf{S}_{SBC}$ is,

$$\mathbf{S}_{SBC}\mathbf{D}^s = [\mathbf{o}, \mathbf{I}_M]\mathbf{D}^s + [-\mathbf{I}_M, \mathbf{o}]\mathbf{D}^s$$

$$= \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -1 & 1 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{4.27}$$

when N = 3.

Comparing eqn (4.25) and eqn (4.27), it is found that in $\mathbf{S}_{SBC}\mathbf{D}^s$, data flows are dispersed by using the interconnection primitive $[1,0]^T$ and -$[1,0]^T$ (and $[0,1]^T$ and -$[0,1]^T$) once instead of $[1,0]^T$ (and $[0,1]^T$) twice, i.e., a balanced usage of the bi-direction interconnection primitives (this is why we choose $\mathbf{S}_{SBC}$ in this way). Furthermore this property is true for N dimensions, due to the following proposition.

Proposition 4.3.1 $\mathbf{S}_{SBC}\mathbf{D}^s$ *has the same number of non-zero entries as* $\mathbf{S}_{SUC}\mathbf{D}^s$, *but half of them are positive, half are negative.*

Proof: As mentioned before, $\mathbf{D}^s$ is like a transposed table of binary numbers from 1 to $2^M - 1$. It is trivial to add 0 at the beginning to make it a full table of binary numbers. The i-th row of the full binary table can be expressed as a repeating pattern $\overline{\mathbf{0}}^i\overline{\mathbf{1}}^i\overline{\mathbf{0}}^i\overline{\mathbf{1}}^i$ 2^{N-i-2} times, where $\overline{\mathbf{0}}^i = [0\cdots0]$, total 2^i "0"s in a line, and $\overline{\mathbf{1}}^i = [1\cdots1]$, total 2^i "1"s in a line. Similarly, the (i+1)-th row can be written as a repeating pattern $\overline{\mathbf{0}}^i\overline{\mathbf{0}}^i\overline{\mathbf{1}}^i\overline{\mathbf{1}}^i$ 2^{N-i-2} times.

In eqn (4.26), the item $[\mathbf{o}, \mathbf{I}_M]\mathbf{D}^s$ is the table with the first row deleted, and the item $[-\mathbf{I}_M, \mathbf{o}]\mathbf{D}^s$, the table with the last row deleted. Thus, the i-th row of $\mathbf{S}_{SBC}\mathbf{D}^s$ is the difference of the (i+1)-th and the i-th rows of the table, i.e.,

$$\overline{\mathbf{0}}^i\overline{\mathbf{0}}^i\overline{\mathbf{1}}^i\overline{\mathbf{1}}^i - \overline{\mathbf{0}}^i\overline{\mathbf{1}}^i\overline{\mathbf{0}}^i\overline{\mathbf{1}}^i = \overline{\mathbf{0}}^i\overline{-\mathbf{1}}^i\overline{\mathbf{1}}^i\overline{\mathbf{0}}^i$$

repeated 2^{N-i-2} times, where $\overline{-\mathbf{1}}^i = -1\cdots-1$, total 2^i "-1"s in a line. It is clear, therefore, that there are 2^{M-1} "1"s and the same number of "-1"s in every row of $\mathbf{S}_{SBC}\mathbf{D}^s$. Obviously, $\mathbf{S}_{SUC}\mathbf{D}^s$ has 2^M non-zero entries, but all are positive. □

It can be proved that the corresponding projection vector $\mathbf{s}^p_{SBC} = [1,\cdots,1]^T$. In fact considering eqn (4.26), there is one "1" and one "-1" in each line, thus $\mathbf{S}_{SBC}\mathbf{s}^p_{SBC} = \mathbf{o}$, as required by definition. The $\bar{\mathbf{t}}$ which satisfies eqn (4.18) and eqn (4.19) with the smallest

norm has the form $[1, 2, \cdots, 2, 1]$. So $e = \sum_{i=0}^{M} t_i$. Unfortunately, $e \gg 1$ can occur – implying poor efficiency.

4.3.3 Permuting the Space Projection Matrix

Obviously the matrices $[\mathbf{o}, \mathbf{I}_M]$ and $[\mathbf{o}, \mathbf{I}_M] + [\mathbf{I}_M, \mathbf{o}]$ are not the only matrices for $\mathbf{P}_{SUC}$ and $\mathbf{P}_{SBC}$, respectively, satisfying the criteria. In fact, by permuting the columns of $\mathbf{S}_{SUC}$ and $\mathbf{S}_{SBC}$, we obtain N! versions. We term eqn (4.23) the normal form of $\mathbf{S}_{SUC}$, and eqn (4.26) the normal form of $\mathbf{S}_{SBC}$.

Any other version of $\mathbf{S}_{SUC}$ (and $\mathbf{S}_{SBC}$) is expressed by $\mathbf{S}_{SUC}^{(p_0 \cdots p_i \cdots p_M)}$ (and $\mathbf{S}_{SBC}^{(p_0 \cdots p_i \cdots p_M)}$), where p_i means that the ith column of $\mathbf{S}_{SUC}$ is permuted to the p_i-th column of $\mathbf{S}_{SUC}^{(p_0 \cdots p_i \cdots p_M)}$. For instance, with N = 4

$$\mathbf{S}_{SUC}^{(1302)} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \text{ and } \mathbf{S}_{SBC}^{(1302)} = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \end{bmatrix}$$

$\mathbf{S}_{SUC}^{(p_0 \cdots p_i \cdots p_M)}$ can also be expressed by $\mathbf{S}_{SUC}^{(p_0 \cdots p_i \cdots p_M)} = \mathbf{S}_{SUC}\mathbf{P}_p^{(p_0 \cdots p_i \cdots p_M)}$, where $\mathbf{P}_p^{(p_0 \cdots p_i \cdots p_M)}$ is a permuting matrix and obtained by permuting the i-th column of $\mathbf{I}$ to the position of the p_i-th column. For example,

$$\mathbf{P}_p^{(1302)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In many places, for simplicity, the superscript of $\mathbf{P}_p^{(p_0 \cdots p_i \cdots p_M)}$ is omitted when the meaning is clear. By permuting the columns of $\mathbf{S}_{SUC}$, its projection vector $\mathbf{s}^p$ may be changed. For instance, the projection vector of $\mathbf{S}_{SUC}^{(1302)}$ is $[0, 1, 0, 0]^T$. However, the column permutation of $\mathbf{S}_{SBC}$ does not change the projection vector. It can be seen from $\mathbf{S}_{SBC}^{(1302)}$ that, for any permutation there is one "1" and one "-1" in every row. Thus $\mathbf{S}_{SUC}^{(1302)}\mathbf{s}_{SBC}^p = \mathbf{o}$. For general cases, we deduce that

$$\mathbf{o} = \mathbf{S}_{SBC}\mathbf{s}_{SBC}^p = \mathbf{S}_{SBC}\mathbf{P}_p\mathbf{P}_p^{-1}\mathbf{s}_{SBC}^p = \mathbf{S}_{SBC}^{(p_0 \cdots p_M)}\mathbf{s}_{SBC}^p$$

where $\mathbf{P}_p^{-1}$ is a permuting matrix, while $\mathbf{P}_p^{-1}\mathbf{s}_{SBC}^p$ changes nothing.

When N is large, flexibility from the N! versions of $\mathbf{S}_{SUC}$ and $\mathbf{S}_{SBC}$ apparently complicates matters. However, it does give the opportunity to optimise the partitioning and

mapping. So it is quite worthwhile keeping all of them as candidates for S in the follow-up procedures of design.

4.4 Further LSGP Partitioning for SBC

If $e > 1$, only one processing cell in every e cells is active at a moment. A great waste of computational resources. We perform a second partitioning (the first is the supernode partitioning) to improve efficiency using a LSGP method. A number of LSGP methods are proposed to deal with the problem. We choose the method in [24] here due to its brief mathematical nature (see Chapter 2 page 22-24).

We proceed by defining an enlarged virtual processor array (EVPA) of size $(l_0 \times k_0) \times \cdots \times (l_{M-1} \times k_{M-1})$. At first, the supernode parallelepiped is scaled so that the resulting supernode polyhedron can be mapped within the EVPA using the space mapping, then the supernode polyhedron is compressed by the LSGP partitioning onto the original processor array.

We propose a method to perform the LSGP partitioning, which is expressed as the following strategy.

Strategy 4.4.1 *Use an "enlarged"* $\bar{\mathbf{t}}$ *such that* $e = \sum_{i=0}^{M} t_i = g^M$, *where g is a small integer such as 2 (when N >3) or 3. "enlarged" means that the entries* t_i *of* $\bar{\mathbf{t}}$ *are greater than their lower bounds. For all* $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$, *changing* $\bar{\mathbf{t}}$ *under conditions of eqn(4.19) and with* $\sum_{i=0}^{M} t_i = g^M$, *evaluate* $\mathbf{Q} : \bar{\mathbf{t}}\mathbf{Q} = o$, $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}\mathbf{Q}$, $\mathbf{A}$ *and M! k's. If there exists a* $\mathbf{k}$ *such that* $k_0 = \cdots = k_{M-1} = g$, *put the* $\bar{\mathbf{t}}$ *into a set* $\overline{T}^{(p_0 \cdots p_M)}$, *otherwise ignore it. In this way, each permutation* $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$ *is accompanied by a set of* $\bar{\mathbf{t}}$*'s,* $\overline{T}^{(p_0 \cdots p_M)}$. *In fact by employing any pair of* $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$ *and* $\bar{\mathbf{t}} \in \overline{T}^{(p_0 \cdots p_M)}$, *we are guaranteed to make an LSGP partitioning of even compressions with a factor g on every dimension.*

Clearly it is a very time-consuming task to work out all the available $\bar{\mathbf{t}}$'s when N is large. Fortunately the $\overline{T}^{(p_0 \cdots p_M)}$'s can be pre-computed. In addition, we introduce the following proposition to reduce the required computation by half.

Proposition 4.4.1 $\mathbf{S}_{SBC}\mathbf{P}_p$ *and* $\mathbf{S}_{SBC}(\mathbf{J}_N\mathbf{P}_p)$ *have the same* $\overline{T}^{(p_0 \cdots p_M)}$, *where* $\mathbf{J}_N$ *is an antidiagonal identity matrix.*

Proof. Because

$$\begin{aligned}
\mathbf{J}_M\mathbf{S}_{SBC}\mathbf{J}_N &= \mathbf{J}_M([\mathbf{o},\mathbf{I}_M]+[-\mathbf{I}_M,\mathbf{o}])\mathbf{J}_N \\
&= [\mathbf{o},\mathbf{J}_M]\begin{bmatrix} \overline{\mathbf{o}} & 1 \\ \mathbf{J}_M & \mathbf{o} \end{bmatrix}+[-\mathbf{J}_M,\mathbf{o}]\begin{bmatrix} \mathbf{o} & \mathbf{J}_M \\ 1 & \overline{\mathbf{o}} \end{bmatrix} \\
&= [\mathbf{I}_M,\mathbf{o}]+[\mathbf{o},-\mathbf{I}_M] \\
&= -\mathbf{S}_{SBC}
\end{aligned}$$

we have

$$\begin{aligned}
\text{HNF} &= (\mathbf{S}_{SBC}\mathbf{P}_p\mathbf{Q})\mathbf{U} \\
&= (\mathbf{J}_M\mathbf{S}_{SBC}\mathbf{J}_N\mathbf{P}_p\mathbf{Q})(-\mathbf{U})
\end{aligned}$$

where $\mathbf{U}$ is a unimodular matrix. So, $\mathbf{S}_{SBC}\mathbf{P}_p \mapsto$ HNF and $\mathbf{S}_{SBC}\mathbf{J}_N\mathbf{P}_p \mapsto \mathbf{J}_M$HNF, but $\mathbf{J}_M$HNF does not change the diagonal elements of the HNF. □

Let $\mathbf{P}'_p = \mathbf{J}_N\mathbf{P}_p$. $\mathbf{P}'_p$ is also a member of the set of the permutation matrices. If the $\overline{T}^{(p_0\cdots p_M)}$ for $\mathbf{S}_{SBC}\mathbf{P}_p$ has been computed, we need not do it for $\mathbf{S}_{SBC}\mathbf{P}'_p$, because they share the same one as proved in Proposition 4.4.1.

A further reduction by half is achieved by the following proposition.

Proposition 4.4.2 *Suppose* $\mathbf{S}$ *is a member of the set of permuted* $\mathbf{S}_{SBC}$ *and is accompanied by* $\overline{T}$*; if* $\mathbf{S}' = \mathbf{SJ}$ *(i.e.,* $\mathbf{P}'_p = \mathbf{P}_p\mathbf{J}$*), then* $\mathbf{S}'$ *will be accompanied by* $\overline{T}\mathbf{J}$.

Proof. In fact, suppose $\mathbf{A} = \mathbf{SQ}$ and $\bar{\mathbf{t}}\mathbf{Q} = \mathbf{o}$. Since $\mathbf{SQ} = (\mathbf{SJ})(\mathbf{JQ})$ and $(\bar{\mathbf{t}}\mathbf{J})(\mathbf{JQ}) = \mathbf{o}$, $\bar{\mathbf{t}}' = \bar{\mathbf{t}}\mathbf{J}$ with $\mathbf{S}'$ will result in the same $\mathbf{A}$ as $\bar{\mathbf{t}}$ with $\mathbf{S}$. □

Notice that $\mathbf{S}'$ is also a member of the N! permuted versions of $\mathbf{S}_{SBC}$, therefore if $\overline{T}$ for $\mathbf{S}$ has been computed, we can simply obtain the $\overline{T}'$ for $\mathbf{S}'$ by just doing $\overline{T}\mathbf{J}$ without complex computing.

4.5 Scaling the Supernode Parallelepiped and Optimisation

Once a space mapping is known, we can determine the scale of the parallelepiped for partitioning nodes to supernodes. In subsection 4.2.2, the cone basis $\mathbf{E}$ has been scaled to $\mathbf{B}_d$ which defines the partitioning parallelepipeds for forming the canonical supernode dependency. However, to map the supernode domain within a given regular processor array, we must to rescale the $\mathbf{E}$ to $\mathbf{B}_s$ which defines the partitioning parallelepiped for fixed-size mapping. Usually, the parallelepiped of $\mathbf{B}_s$ is larger that of $\mathbf{B}_d$ so that the former can take over the latter, otherwise we should combine them together as to be discussed later.

Similar to subsection 4.2.2, the scaling of $\mathbf{E}$ can be expressed by introducing a positive diagonal matrix $\mathbf{F} = diag(f_0, \cdots, f_M)$, forming a scaled cone basis $\mathbf{B}_s = \mathbf{E}\mathbf{F}^{-1}$.

As also known in subsection 4.2.2, partitioning the computation polyhedron with the parallelepiped defined by $\mathbf{B}_s$ is equivalent to a linear transformation with $\mathbf{B}_s$ as the basis from the original space to a quasi-supernode space S^q. Let $\mathbf{V}^q$ be the transformed polyhedron vertices in S^q, i.e.

$$\mathbf{V}^q = \mathbf{B}_s^{-1}\mathbf{V} = \mathbf{F}\mathbf{E}^{-1}\mathbf{V} = \mathbf{F}\mathbf{W} \tag{4.28}$$

where $\mathbf{W} = \mathbf{E}^{-1}\mathbf{V}$ contains the vertices in a space with $\mathbf{E}$ as the basis. The space mapping $\mathbf{S}$ maps $\mathbf{V}^q$ onto the processor array space

$$\mathbf{U} = \mathbf{S}\mathbf{V}^q = \mathbf{S}\mathbf{F}\mathbf{W} = \mathbf{S}_F\mathbf{W} \tag{4.29}$$

where $\mathbf{S}_F = \mathbf{S}\mathbf{F}$. Eqn (4.29) shows that the scaling of $\mathbf{E}$ is equivalent to a scaling of $\mathbf{S}$ (i.e. scaling each column of $\mathbf{S}$) associated with the unchanged vertices $\mathbf{W}$ and simplifies the scaling procedure greatly.

It is known that $\mathbf{S}$ has N! permuted versions. We have to compute the $\mathbf{F}^{(p_0 \cdots p_M)}$ corresponding to each of them, one way to do this follows from the fact that for any permuted version $\mathbf{S}^{(p_0 \cdots p_M)}$ of $\mathbf{S}$, eqn (4.29) is modified to

$$\begin{aligned} \mathbf{U} &= \mathbf{S}^{(p_0 \cdots p_M)}\mathbf{V}^q = (\mathbf{S}\mathbf{P}_p)\mathbf{F}^{(p_0 \cdots p_M)}\mathbf{W}) \\ &= \mathbf{S}\mathbf{F}'(\mathbf{P}_p\mathbf{W}) = \mathbf{S}_{F'}\mathbf{W}^{(p_0 \cdots p_M)} \end{aligned} \tag{4.30}$$

where $\mathbf{F}' = \mathbf{P}_p \mathbf{F}^{(p_0 \cdots p_M)} \mathbf{P}_p^{-1}$ [1]. These formulas are applicable to a permuted vertex matrix $\mathbf{W}^{(p_0 \cdots p_M)}$ in order to find a scaling diagonal matrix $\mathbf{F}'$, and the scaling diagonal matrix $\mathbf{F}^{(p_0 \cdots p_M)}$ which corresponds to $\mathbf{S}^{(p_0 \cdots p_M)}$ will be

$$\mathbf{F}^{(p_0 \cdots p_M)} = \mathbf{P}_p^{-1} \mathbf{F}' \mathbf{P}_p \tag{4.31}$$

Definition 4.5.1 *The upper bound and lower bound of* $\mathbf{U}$ *in the i-th dimension are defined as*

$$u_i^u = \max_{0 \le j < n} u_{i,j} \qquad u_i^l = \min_{0 \le j < n_v} u_{i,j}$$

respectively, where $u_{i,j} \in \mathbf{U}$. *And* $\mathbf{w}_i^u$ *is a vertex associated with* u_i^u, *and* $\mathbf{w}_i^l$ *a vertex associated with* u_i^l, *i.e.,*

$$\begin{aligned} \mathbf{w}_i^u &= \{\mathbf{w}_j : u_i^u = \mathbf{1}_i^T \mathbf{S}_F \mathbf{w}_j, \ \ 0 \le j < n_v\} \\ \mathbf{w}_i^l &= \{\mathbf{w}_j : u_i^l = \mathbf{1}_i^T \mathbf{S}_F \mathbf{w}_j, \ \ 0 \le j < n_v\} \end{aligned} \tag{4.32}$$

where $\mathbf{w}_i^u$, $\mathbf{w}_i^l$ *and* $\mathbf{w}_j \in \mathbf{W}$, $\mathbf{1}_i$ *is a sampling (or choosing) vector.*

Definition 4.5.2 *An accurate scaling of the* $\mathbf{S}$ *is such that* $\mathbf{U}$ *is just within the processor array* [2], *i.e.,* $\forall i, \ \ 0 \le i < M, \ \ u_i^u - u_i^l = l_i$. *If* $\forall i, \ \ 0 \le i < M, \ \ u_i^u - u_i^l \le l_i$, $\mathbf{U}$ *is said to be acceptable.* $u_i^u - u_i^l$ *is called the projected size in the i-th dimension.*

It would seem that an accurate $\mathbf{U}$ is the best result of re-scaling $\mathbf{E}$. However, this is not what we actually want in practice. In fact, an accurate $\mathbf{U}$ means that, in the case N = 1, u_0^l is assigned to processor 0 and u_0^u, to processor l_0, so $l_0 + 1$ processors are needed in total instead of l_0 processors.

4.5.1 SUC Cases

Scaling $\mathbf{S}_{SUC}$

The scaling of $\mathbf{S}_{SUC}$ is straightforward. Because there is one and only one "1" in each row of $\mathbf{S}_{SUC}$, $\forall i, \ \ 0 \le i < M$, we have,

$$u_{i,j} = f_{i+1} w_{i+1,j}, \ \ \forall j \ \ , 0 \le j < n_v \tag{4.33}$$

[1] $\mathbf{F}'$ must be and is a diagonal matrix whose diagonal entries are those of $\mathbf{F}$ but permuted by $\mathbf{P}_p$. In fact, $\mathbf{P}_p$ can be decomposed as $\mathbf{P}_p = \mathbf{P}_k \mathbf{P}_{k-1} \cdots \mathbf{P}_1$, where $\mathbf{P}_i$ is a basic permuting matrix: permuting two rows. $\mathbf{F}' = \mathbf{P}_k(\mathbf{P}_{k-1}(\cdots(\mathbf{P}_1 \mathbf{F} \mathbf{P}_1^{-1})\cdots)\mathbf{P}_{k-1}^{-1})\mathbf{P}_k^{-1}$. Each term in $(\cdots)$ from the innermost to the outmost is a diagonal matrix produced by permuting two diagonal entries of its predecessor.

[2] If without special indication the processor array also indicates the EVPA.

where $w_{i,j} \in \mathbf{w}_j$. Now we cannot know u_i^u and u_i^l, because f_{i+1} is unknown. However, due to the requirement that $f_{i+1} > 0$, it is possible to find expressions for them, viz

$$u_i^u = f_{i+1} w_i^u \qquad u_i^l = f_{i+1} w_i^l \tag{4.34}$$

where

$$w_i^u = \max_{0 \le j < n_v} w_{i,j} \qquad w_i^l = \min_{0 \le j < n_v} w_{i,j} \tag{4.35}$$

An accurate scaling of the $\mathbf{S}_{SUC}$ demands that

$$f_i^a = \frac{l_{i-1}}{w_i^u - w_i^l}, \quad \forall i, \quad 1 \le i < N \tag{4.36}$$

where superscript "a" means that the variable is for accurate scaling. Note that f_0^a is not determined. Let $f_0^a = +\infty$ which will disappear later.

Eqn (4.36) can also be expressed in a form of matrices. Letting $\mathbf{L} = diag(+\infty, l_0, \cdots, l_M)$ and $\mathbf{W}^{u-l} = diag(\frac{1}{w_0^u - w_0^l}, \cdots, \frac{1}{w_M^u - w_M^l})$, Eqn (4.36) is equivalent to

$$\mathbf{F}^a = \mathbf{L}\mathbf{W}^{u-l} \tag{4.37}$$

where $\mathbf{F}^a = diag(+\infty, f_1^a, \cdots, f_M^a)$.

Eqn (4.36) or eqn (4.37) are not the only formulas for determining the f_i. Recall that one purpose of the supernode partitioning is to include all the dependencies in the parallelepiped, in eqn (4.5) a diagonal matrix, $\mathbf{F}_d$, was introduced for this purpose, and its elements determined by eqn (4.6). Therefore, the f_i could be formulated as

$$f_i = min(f_i^d, f_i^a) \quad \forall i, \quad 0 \le i < N \tag{4.38}$$

Obviously, if $\exists i$, $1 < i < M$, $f_i^d < f_i^a$, we cannot obtain the accurate scaling. Consequently to create the canonical dependencies we have to sacrifice some efficiency.

Scaling permuted versions of $\mathbf{S}_{SUC}$

In order to produce $\mathbf{F}^{(p_0 \cdots p_M)a} = diag(f_0^a, \cdots, f_M^a)$ for any permuted version $\mathbf{S}_{SUC}\mathbf{P}_p^{(p_0 \cdots p_M)}$ of $\mathbf{S}_{SUC}$ (note $\mathbf{P}_p^{(p_0 \cdots p_M)}$ simply as $\mathbf{P}_p$), it would be better to make a use of eqn (4.30) and

eqn (4.31). Comparing eqn (4.30) with eqn (4.29), we know that $\mathbf{F}'$ can be derived in the same way as $\mathbf{F}$, by just permuting the rows of $\mathbf{W}$. For the case of SUC, this is equivalent to permuting the diagonal elements of $\mathbf{W}^{u-l}$ in eqn (4.37). Therefore [3]

$$\mathbf{F}' = \mathbf{L}(\mathbf{P}_p \mathbf{W}^{u-l} \mathbf{P}_p^{-1}) \tag{4.39}$$

Substituting eqn (4.39) into eqn (4.31) produces

$$\mathbf{F}^{(p_0 \cdots p_M)a} = \mathbf{P}_p^{-1}(\mathbf{L}(\mathbf{P}_p \mathbf{W}^{u-l}))\mathbf{P}_p^{-1})\mathbf{P}_p) = (\mathbf{P}_p^{-1}\mathbf{L}\mathbf{P}_p)\mathbf{W}^{u-l} \tag{4.40}$$

Then every element of $\mathbf{F}^{(p_0 \cdots p_M)a}$ should be checked and modified with eqn (4.38).

4.5.2 SBC Case

The scaling of $\mathbf{S}_{SBC}$ is a much more difficult problem than that of $\mathbf{S}_{SUC}$ because there are two non-zero elements in each row of $\mathbf{S}_{SBC}$. Note that

$$\mathbf{S}_F = \begin{bmatrix} -f_0 & f_1 & 0 & 0 & \cdots \\ 0 & -f_1 & f_2 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & -f_{M-1} & f_M \end{bmatrix}. \tag{4.41}$$

then, $\forall i,\ 0 \le i < M$

$$u_{i,j} = \bar{\mathbf{s}}_i^F \mathbf{w}_j = -f_i w_{i,j} + f_{i+1} w_{i+1,j},\ \ \forall j\ \ , 0 \le j < n_v \tag{4.42}$$

where $\bar{\mathbf{s}}_i^F = \bar{\mathbf{1}}_i \mathbf{S}_F = [0, \cdots, 0, -f_i, f_{i+1}, 0, \cdots, 0]$. According to the definition of accurate scaling, a system of equations is obtained, $\forall i,\ 0 \le i < M$ such that

$$\begin{aligned} l_i &= u_i^u - u_i^l = \bar{\mathbf{1}}_i[\mathbf{S}_F \mathbf{w}_i^u - \mathbf{S}_F \mathbf{w}_i^l] \\ &= -f_i w_{i,0}^u + f_{i+1} w_{i,1}^u + f_i w_{i,0}^l - f_{i+1} w_{i,1}^l \\ &= (w_{i,0}^l - w_{i,0}^u) f_i + (w_{i,1}^u - w_{i,1}^l) f_{i+1} \end{aligned} \tag{4.43}$$

where

$$\begin{aligned} w_{i,0}^u &= \bar{\mathbf{1}}_i \mathbf{w}_i^u \qquad & w_{i,1}^u &= \bar{\mathbf{1}}_{i+1} \mathbf{w}_i^u \\ w_{i,0}^l &= \bar{\mathbf{1}}_i \mathbf{w}_i^l \qquad & w_{i,1}^l &= \bar{\mathbf{1}}_{i+1} \mathbf{w}_i^l \end{aligned} \tag{4.44}$$

[3] if $\mathbf{M}$ is a diagonal matrix, $\mathbf{P}_p \mathbf{M} \mathbf{P}_p^{-1}$ performs simply the permutation of the diagonals of $\mathbf{M}$

There are (N - 1) equations but N variables, so one of $f_0, \cdots, f_M$ can be taken as an argument, say f_k, and the rest can be determined as a piece-wise linear function of f_k. Two criteria that $f_0, \cdots, f_M > 0$ and $\frac{1}{f_0}, \cdots, \frac{1}{f_M}$ must be large enough to enclose the dependencies $\mathbf{D}$, are used to delimit the valid interval $[f_k^{lb}, f_k^{ub}]$ of f_k. The actual procedure is complex (see Appendix A). We describe this procedure briefly as follows.

Step 1 Take f_k as the argument and let $f_k = 0$ be an initial point, then compute a set of initial $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ corresponding to the initial f_k.

Step 2 Derive the linear functions for $\mathbf{F}^a$ with these $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$.

Step 3 Increase f_k from the initial point. At some points of f_k, $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ are overstepped by other $\mathbf{w}_i$'s. Replace the old $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ with the new ones.

Step 4 Repeat Step 2 and Step 3 until any of $f_0, \cdots, f_M$ is less than zero.

Step 5 Redetermine the valid interval of f_k such that $f_i \leq f_i^d$, $\forall\, i$, where f_i^d is defined by eqn (4.6).

In Appendix A, only the situation where $\mathbf{S}_{SBC}$ are the space mappings is discussed. The algorithms derived there can be extended to any permuted versions of $\mathbf{S}_{SBC}$. Fortunately this extension is not very difficult, considering eqn (4.30).

It can be concluded that a complex model of $\mathbf{S}$ also complicates the procedure of fixed-size partitioning dramatically.

4.5.3 Optimising

Optimising for the SUC Case

The permuted versions give N! choices for optimisation. The procedure of optimisation is given in Algorithm B.2.2. We should work out all the N! $\mathbf{F}^{(p_0 \cdots p_M)}$ and compute the transformed polyhedron vertices $\mathbf{V}^q$ by eqn (4.28) for each of them. Note that $\bar{\mathbf{t}}_{SUC}\mathbf{V}^q$ gives the time moments for executing each of the vertices and the difference between the longest and the shortest time moments indicates the total computation time. We have to compute the total computation times for all the cases of N! $\mathbf{F}^{(p_0 \cdots p_M)}$'s and choose the one which minimises the total computation time.

Optimising for the SBC Case

Once the problem is scaled we need to choose the best mapping. It can be seen that up to now, we have many possible choices and must

1. determine f_k within its interval $[f_k^{lb}, f_k^{ub}]$,
2. select which of $f_1, \cdots, f_{M-1}$ is used as the argument f_k,
3. select timing vector $\bar{\mathbf{t}}$ from the set $\overline{T}^{(p_0 \cdots p_M)}$ and its accompanying $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$
4. select a $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$ from all permuted versions of $\mathbf{S}_{SBC}$.

Optimising each of these tasks is considered below.

1. The criteria to determine f_k is to directly minimise the volume of supernode, i.e., to maximise

$$v_k = \prod_{i=0}^{M} \frac{1}{f_i} \tag{4.45}$$

which gives the minimal total computation time. But some of the edges of the resulted parallelepiped may be quite long, meanwhile some others may be quite short, so the shape of the parallelepiped may be strange. However, for the sake of minimising computation time, let us take it as the optimum criterion for determining f_k, that is, determining the volume and shape of the parallelepiped. Let the minimal v_k be v_k^{min} (we may change the criterion in practice).

2. For completeness, we should let argument $f_k = f_i$, and evaluate $v_k^{min}, \forall i, 1 \leq i < M$. Let $v^{min} = \max_{0 \leq k < n} v_k^{min}$. Choose the f_k, as well as the corresponding $f_0, \cdots f_M$, such that $v_k^{min} = v^{min}$.

3. Here we make a transformation with $\mathbf{EF}^{-1}$ as the basis to determine the vertices $\mathbf{V}^q$ in quasi-supernode space by eqn (4.28). Pre-multiply $\mathbf{V}^q$ by each $\bar{\mathbf{t}}_i \in \overline{T}^{(p_0 \cdots p_M)}$ accompanying the current $\mathbf{S}$, $0 \leq i < n_t$, n_t is the number of the timing vectors in the $\overline{T}^{(p_0 \cdots p_M)}$. The shortest computation time t^{min} is

$$t^{min} = \min_{i=0}^{n_t - 1} \{ \max_{j=0}^{n_v - 1} (\bar{\mathbf{t}}_i \mathbf{v}_j^q) - \min_{j=0}^{n_v - 1} (\bar{\mathbf{t}}_i \mathbf{v}_j^q) \} \tag{4.46}$$

where $\mathbf{v}_j^q \in \mathbf{V}^q$, this is equivalent to a matrix $\overline{T}^{(p_0 \cdots p_M)} \mathbf{F} \mathbf{E}^{-1} \mathbf{V}$. Next among all rows find the minimal difference of the maximal element and the minimal element of a row. The

optimal timing vector $\overline{\mathbf{t}}_i^{min}$ will be

$$\overline{\mathbf{t}}^{min} = \{\overline{\mathbf{t}}_i : \max_{j=0}^{m-1}(\overline{\mathbf{t}}_i \mathbf{v}_j^q) - \min_{j=0}^{m-1}(\overline{\mathbf{t}}_i \mathbf{v}_j^q) = \ t^{min}, \ 0 \le i < n_t\}. \tag{4.47}$$

4. For each $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$, compute $t_{(p_0 \cdots p_M)}^{min} v_k^{min}$ and select the minimal one and take its corresponding $\mathbf{S}_{SBC}^{(p_0 \cdots p_M)}$ as $\mathbf{S}$. By the following proposition, the searching space, N! permuted versions, can be cut down to half.

Proposition 4.5.1 *For the processor array, if* $\mathbf{l} = \mathbf{J}_M \mathbf{l}$ *(e.g.,* $\mathbf{l} = [5, 6, 8, 6, 5]^T$*), then the two permuted versions of* $\mathbf{S}_{SBC}$ *produced by* $\mathbf{P}_p$ *and* $\mathbf{J}\mathbf{P}_p$ *will behave the same.*

Proof Let $\mathbf{U} = \mathbf{S}_{SBC}\mathbf{P}_p\mathbf{F}\mathbf{V}$ and $\mathbf{U}' = \mathbf{S}_{SBC}\mathbf{J}_N\mathbf{P}_p\mathbf{F}'\mathbf{V}$, and $\mathbf{U}$ is accurately mapped. Suppose $\mathbf{F} = \mathbf{F}'$. Then we have $\mathbf{U} = -\mathbf{J}_M\mathbf{U}'$, because $\mathbf{J}_M\mathbf{S}_{SBC}\mathbf{J}_N = -\mathbf{S}_{SBC}$. Thus $\mathbf{U}'$ is also accurately mapped, because $\mathbf{l} = \mathbf{J}_M\mathbf{l}$. This means that the assumption of $\mathbf{F} = \mathbf{F}'$ is correct.

Meanwhile, we know from Proposition 4.4.1 that the two versions have the same accompanied $\overline{T}^{(p_0 \cdots p_M)}$, therefore they result in the same t^{min}. □

Because $\mathbf{S}_{SBC}\mathbf{P}_p$ and $\mathbf{S}_{SBC}\mathbf{J}_N\mathbf{P}_p$ both belong to the set of permuted versions of $\mathbf{S}_{SBC}$ and they achieve the same t^{min} according to Proposition 4.5.1 if $\mathbf{l} = \mathbf{J}_M\mathbf{l}$, only one needs to be tested.

4.5.4 Integralization of the Quasi-supernode Transformation Matrix

Up to now, the matrix of the quasi-supernode transformation has been obtained, but in the real number domain. The matrix $\mathbf{B}_s$ has to be approximated to an integral matrix $\mathbf{B}$ ($\forall i$, $0 \le i < N$, approximate $\mathbf{b}_i^s \in \mathbf{B}_s$ with $\mathbf{b}_i \in Z^N$, the $\mathbf{b}_i$ form $\mathbf{B}$) to guarantee that supernodes are equal up to a translation.

The integralization process makes the elements of $\mathbf{b}_i$ deviate from $\mathbf{b}_i^s$, so there exists the possibility that the quasi-supernode polyhedron transformed by $\mathbf{B}$ may no longer fit within the boundaries of the processor array. For the case of SUC, if the length of $\mathbf{b}_i$ is shorter than that of $\mathbf{b}_i^s$, the size of supernodes will decrease and the size of the supernode

domain will increase so that the supernode domain can no longer be mapped into the given array by the $\mathbf{S}$. For the case of SBC, the situation is complex because the span of $\mathbf{U}$ in the processor domain along any one dimension is affected by the sizes of supernode domain along two dimensions. When the $\mathbf{b}_i$ deviate from $\mathbf{b}_i^s$, if the sizes of supernode domain along the two dimensions do not change together properly, it is possible for the spans of $\mathbf{U}$ to be enlarged over the sizes of processor domain.

Therefore, we must choose the suitable $\mathbf{b}_i$, from a number of candidates, which force the transformed quasi-supernode polyhedron inside the domain of the processor array. Unfortunately, the accurate match of the size of the quasi-supernode polyhedron to that of the processor array cannot be guaranteed in general. The only thing we can hope for is to find an integral version $\mathbf{B}$ of $\mathbf{B}_s$ such that the size of the resulted quasi-supernode polyhedron is smaller than but approaches to the size of the processor array in every dimension.

We produce the candidates for $\mathbf{b}_i$ in a simple way. Suppose

$$g_i = \lceil \frac{1}{f_i} \rceil + j \tag{4.48}$$

where $j = 0, \cdots, n_c - 1$ and n_c is the number of candidates of the $\mathbf{b}_i$. Then let

$$\mathbf{b}_i = g_i \mathbf{e}_i \tag{4.49}$$

where $\mathbf{e}_i \in \mathbf{E}$. These integers, form a vector $\mathbf{g} = [g_0, \cdots, g_{N-1}]^T$ indicating the size of a supernode, and a diagonal matrix $\mathbf{G} = diag(g_0, \cdots, g_{N-1})$. Obviously $\mathbf{B} = \mathbf{EG}$.

For the case of SUC, we can just let $n_c = 1$, i.e., $\mathbf{b}_i = \lceil \frac{1}{f_i} \rceil \mathbf{e}_i$. This formula always ensures an acceptable mapping, because $\mathbf{b}_i$ is longer than $\mathbf{b}_i^s$. For the case of SBC, for all $\mathbf{b}_i$, we begin by taking a vector from the bottom of the list for candidates of $\mathbf{b}_i$ as the actual $\mathbf{b}_i$, forming a $\mathbf{B}$. Compute $\mathbf{U} = \mathbf{SB}^{-1}\mathbf{V}$. Check whether the size of $\mathbf{U}$ is beyond the processor array. If this is the case, take another set of vectors from the lists and repeat the process, until a $\mathbf{B}$ for which $\mathbf{U}$ is acceptable is found. Generally speaking, if we make n_c big enough, it is always possible to find an acceptable $\mathbf{B}$. In fact, more choices for $\mathbf{b}_i$ from eqn (4.48) allow the lengths of $\mathbf{b}_i$ to increase evenly in every direction, so that the sizes of the supernode polyhedron will decrease correspondingly without change in shape. As a result, all the sizes of mapped $\mathbf{U}$ in every dimensions decrease instead of increasing.

4.6 Examples and Discussions

Examples are given here to aid understanding of the methodology and illustrate its properties.

4.6.1 A 3-D Example

Example 4.6.1 *3-D computation Problem and 2-D SBC Array*

The computational graph is arbitrarily given by

$$\mathbf{V} = \begin{bmatrix} 0 & 100 & 0 & 0 & 0 & 100 & 100 & 100 \\ 0 & 0 & 100 & 0 & 100 & 0 & 100 & 100 \\ 0 & 0 & 0 & 100 & 100 & 100 & 0 & 100 \end{bmatrix} \tag{4.50}$$

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ 0 & -2 & 1 & -3 \end{bmatrix} \tag{4.51}$$

which are extracted from a nested loop program of the form.

FOR $i_0 := 0$ TO 100
  FOR $i_1 := 0$ TO 100
    FOR $i_2 := 0$ TO 100
      $A(i_0, i_1, i_2) := A(i_0 - 1, i_1 + 1, i_2)$
        $+A(i_0, i_1 - 1, i_2 + 2) + A(i_0, i_1, i_2 - 1) + A(i_0 - 1, i_1 - 1, i_2 + 3)$

Let a mesh connected regular array be defined by the interconnection primitives $\mathbf{P} = \mathbf{P}_{SBC}$, and processor array $\mathbf{l} = [4, 4]^T$. Let $\mathbf{k} = [3, 3]^T$, and pre-compute the $\frac{3!}{2}$ possible $\overline{T}^{(p_0 \cdots p_2)}$ shown in the table below.

$(p_0 \cdots p_2)$	2 1 0	1 2 0	2 0 1
	0 1 2	0 2 1	1 0 2
$\overline{T}^{(p_0 \cdots p_2)}$	1 2 6	2 1 6	1 3 5
	1 5 3	2 3 4	1 6 2
	2 4 3	2 4 3	2 3 4
	3 2 4	2 6 1	3 1 5
	3 4 2	4 2 3	3 2 4
	3 5 1	4 3 2	3 4 2
	4 2 3	5 1 3	4 3 2
	6 2 1	5 3 1	6 1 2

(210) and (012) share the same $\bar{t}$ set, as do (120) and (021), and (201) and (102). For example

$$\mathbf{P}_p^{(120)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad and \qquad \mathbf{P}_p^{(021)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Obviously, $\mathbf{P}_p^{(120)} = \mathbf{J}\mathbf{P}_p^{(021)}$. Further more, the columns of (120)+ (021) and (201)+(102) are just the reverse of each other. It is easy to check that $\mathbf{P}_p^{(120)} = \mathbf{P}_p^{(102)}\mathbf{J}$ and $\mathbf{P}_p^{(201)} = \mathbf{P}_p^{(021)}\mathbf{J}$

Next derive $\mathbf{E}$. Having the aid of Example 4.2.1. $\mathbf{D}$ can be re-expressed as

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ 0 & -2 & 1 & -3 \end{bmatrix} = \mathbf{E}\mathbf{A}_p = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

where the first matrix of the product is the $\mathbf{E}$, and the second is the coefficient matrix. This shows that $\mathbf{D}$ is positively expressed by $\mathbf{E}$. And we have $\bar{\mathbf{a}}^{max} = [1, 2, 1]$.

Next the processor array is enlarged to a EVPA defined by $l_0 = l_1 = 4 \times 3 = 12$, and because the EVPA is square, there are a total of three distinct kinds of permutations (210), (120) and (201). Let's take one, (120), to show the procedure of scaling a supernode parallelepiped (read Appendix A first to understand the procedure).

$$\mathbf{P}_p^{(120)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad \mathbf{P}_p^{(201)} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Note that $\mathbf{P}_p^{(120)}\mathbf{P}_p^{(201)} = \mathbf{I}$. Thus

$$\mathbf{W} = \mathbf{E}^{-1}\mathbf{V} = \begin{bmatrix} 0 & 100 & 100 & 0 & 100 & 100 & 200 & 200 \\ 0 & 200 & 200 & 100 & 300 & 300 & 400 & 500 \\ 0 & 100 & 0 & 0 & 0 & 100 & 100 & 100 \end{bmatrix} \tag{4.52}$$

When N = 3, only f_1 is used as the argument to express f_0 and f_2, so a piecewise function is derived by means of Appendix A with

$$f_0^a = \begin{cases} 0.2f_1 + 0.024 & 0 < f_1 \leq 0.08 \\ -f_1 + 0.12 & 0.08 < f_1 \leq 0.12 \end{cases}$$
$$f_2^a = 0.5f_1 + 0.06 \qquad 0 < f_1 \leq 0.12$$

$\prod_{i=0}^{M} f_i$ is a N-order piecewise polynomial of f_k, we use the optimum searching method to maximise it. The optimum f_0, f_1 and f_2 are 0.036, 0.084, 0.102, respectively. It is known that $\mathbf{F}'$ should be re-permuted as $\mathbf{F}^{(120)} = \text{diag}(0.084, 0.102, 0.036)$.

To compute the executing times of the quasi-supernode vertices $\mathbf{V}^q$ associated with all possible timing vectors $\bar{t} \in \overline{T}^{(120)}$, we evaluate $\overline{T}^{(120)}\mathbf{F}^{(120)}\mathbf{W}$, where the vector set $\overline{T}^{(120)}$ behaves like a matrix. Evaluating the differences of the maximum and minimum values of each row gives the executing times [145 150 152 157 128 130 116 121] for each of the eight timing vectors in $\overline{T}^{(120)}$, respectively. The shortest is 116, thus the timing vector, [5, 1, 3] gives the minimal executing time $t^{min}_{(120)} = 116$. However, this is not the only thing that we are concerned with, the size of the parallelepiped is important too. Thus, the total computation time is proportional to $t^{min:total}_{(120)} := t^{min}_{(120)} v^{min}_3 = \frac{t^{min}_{(120)}}{f_0 f_1 f_2} = 214272$ steps.

In the same way, the total computation times associated with the other permutations are computed,

$(p_0 \cdots p_M)$	(210)	(120)	(201)
$f_0 \times f_1 \times f_2$	$9.5\ 10^{-4}$	$3.1\ 10^{-4}$	$3.6\ 10^{-5}$
$t^{min}_{(p_0 \cdots p_M)}$	152	116	110
$t^{min:total}_{(p_0 \cdots p_M)}$	160062	377373	303369
$\bar{t}^{min}_{(p_0 \cdots p_M)}$	[6,2,1]	[5,1,3]	[6,1,2]

Obviously, the permutation (210) gives the minimal total computation time. Its corresponding $\mathbf{F}^{(210)} = \mathrm{diag}(0.12, 0.114, 0.07)$, $\bar{t} = [6, 2, 1]$,

$$\mathbf{S} = \begin{bmatrix} 0 & 1 & -1 \\ 1 & -1 & 0 \end{bmatrix} \text{ and } \mathbf{B}_s = \mathbf{E}(\mathbf{F}^{(210)})^{-1} = \begin{bmatrix} 8.3 & 0. & 0 \\ -8.3 & 8.7 & 0 \\ 0 & -17.5 & 14.3 \end{bmatrix}$$

Now, $\mathbf{B}_s$ has to be integralised to $\mathbf{B}$. Let $\mathbf{b}_0$, $\mathbf{b}_1$ and $\mathbf{b}_2$ have candidates as follows:

$$\mathbf{b}_0 = \begin{bmatrix} 9 \\ -9 \\ 0 \end{bmatrix}, \begin{bmatrix} 10 \\ -10 \\ 0 \end{bmatrix}, \cdots; \quad \mathbf{b}_1 = \begin{bmatrix} 0 \\ 9 \\ -18 \end{bmatrix}, \begin{bmatrix} 0 \\ 10 \\ -20 \end{bmatrix}, \cdots; \quad \mathbf{b}_2 = \begin{bmatrix} 0 \\ 0 \\ 15 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 16 \end{bmatrix}, \cdots.$$

Selecting $\mathbf{b}_0 = [9, -9, 0]^T$, $\mathbf{b}_1 = [0, 10, -20]^T$ and $\mathbf{b}_2 = [0, 0, 16]^T$ from their own candidate lists, produces a candidate $\mathbf{B}$, which is checked for validity using $\mathbf{U} = \mathbf{S}^{(210)}_{SBC}\mathbf{BV}$ and is valid. Everything is okay, so the projected size $u^u_0 - u^l_0 = 11.5$ and $u^u_1 - u^l_1 = 11.6$, just inside the 12×12 EVPA.

In summary, we obtain two matrices

$$\mathbf{B} = \begin{bmatrix} 9 & 0 & 0 \\ -9 & 10 & 0 \\ 0 & -20 & 16 \end{bmatrix} \quad \textit{and} \quad \mathbf{T} = \begin{bmatrix} 0 & 1 & -1 \\ 1 & -1 & 0 \\ 6 & 2 & 1 \end{bmatrix}$$

Table 4.1: 4-D examples.

	small polyhedron				large polyhedron			
	SUC		SBC		SUC		SBC	
	EVPA	proj.	EVPA	proj.	EVPA	proj.	EVPA	proj.
0-th d	4	3.97	8	7.8	4	3.94	8	7.96
1st d	4	3.93	8	7.9	4	3.97	8	7.86
2nd d	4	3.75	8	7.2	4	3.97	8	7.7
time		2.4s		0.6s		0.3s		0.7s

and a LSGP compression vector $\mathbf{k} = [3,3]^T$. Observe that this example is relatively simple, because when N = 3, the scaling job is easy. There are still a lot of details we cannot cover here, so the example gives only a indication how this methodology works. Consult the appendices for more details□

4.6.2 More Results and Discussions

More results are given below to show the properties of the methodology. In the following examples, the size of the processor array is 4 in all dimensions. Let $\mathbf{k} = [2,2]^T$ for an SBC.

Table 4.1 gives the results of partitioning and mapping of 4-D graphs, where "proj." means the projected array and "time" indicates the time consumed for executing the algorithm in seconds. The "small polyhedron" is a computational polyhedron of the form

$$\mathbf{V} = \begin{bmatrix} 0 & 10 & 0 & 10 & -10 & 0 & 10 & 0 & 20 & 10 \\ 0 & 0 & 20 & 20 & 0 & 40 & 20 & 20 & 40 & 40 \\ 0 & 0 & 0 & 10 & 0 & 10 & 10 & 0 & 10 & 10 \\ 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 0 & 10 \end{bmatrix}$$

the "large polyhedron" is one with the same shape but enlarged 10 times in every dimension.

Table 4.2 is similar, but the "small polyhedron" is

$$\mathbf{V} = \begin{bmatrix} 0 & 10 & 0 & 10 & -10 & 0 & 0 & 10 & 0 & 20 \\ 0 & 20 & 20 & 0 & 0 & 0 & 20 & 20 & 40 & 40 \\ 0 & 0 & 0 & 10 & 0 & 0 & 10 & 10 & 0 & 10 \\ 0 & 0 & 0 & 0 & 10 & 0 & 10 & 10 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 & 10 & 10 & 10 & 10 \end{bmatrix}$$

Table 4.2: 5-D examples.

	small polyhedron				large polyhedron			
	SUC		SBC		SUC		SBC	
	EVPA	proj.	EVPA	proj.	EVPA	proj.	EVPA	proj.
0-th d	4	3.94	8	7.9	4	3.94	8	7.8
1st d	4	3.87	8	7.78	4	3.95	8	7.9
2nd d	4	4.	8	6.7	4	3.96	8	7.9
3rd d	4	3.75	8	7.9	4	3.84	8	7.9
time		2.6s		296s		1.s		3.4s

and the "large polyhedron" also is the same one but enlarged 10 times in every dimension. From the results of above, it can be seen that the projected arrays usually match the EVPA's well, especially for the "large polyhedron".

4.7 Summary

In this chapter, a methodology is proposed for general partitioning and mapping problem under the URE condition. The algorithm can be briefly summarised as follows and is illustrated in Figure 4.4:

Pre-compiling

Find the timing vector sets for all permuted versions of one basic space mapping model.

Compiling

1. Derive a positive expression basis from a dependency matrix.
2. Using the positive expression basis as the directions of supernode partitioning parallelepipeds, scale the parallelepiped according to the basic space mapping model.
3. Optimise the parallelepiped from the permuted versions of the basic model according to their timing vector sets.
4. Approximate the edges of the parallelepiped to integral vectors.

The methodology shows significant advantages over the previous methods [73] and

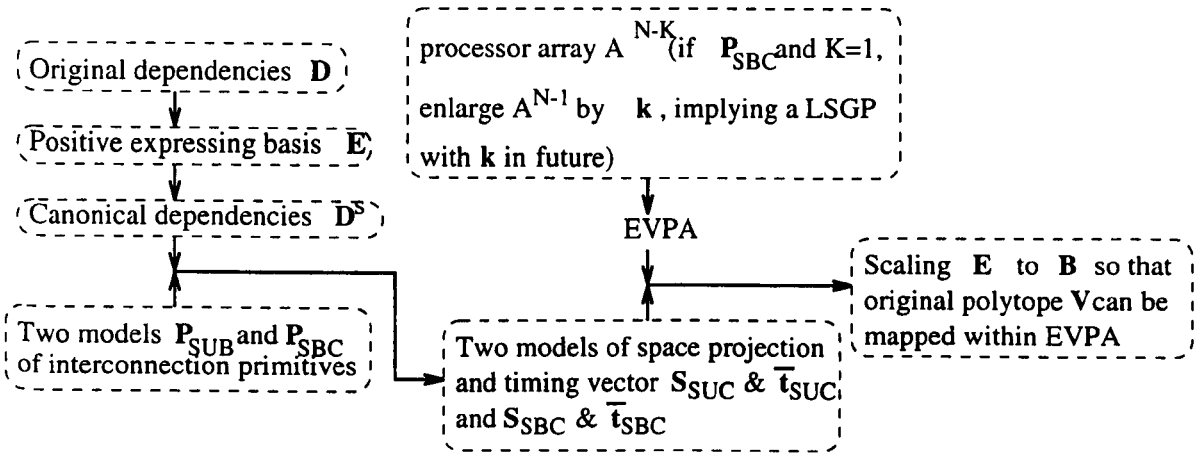


Figure 4.4: A Conceptual Chart of the Partitioning and Mapping Method

[24]. In particular, it considers all the properties with which we are concerned in this thesis: mapping onto a given re processor array, implementing dependencies with given interconnection primitives, transferring data efficiently (locally and packaged), and high computational efficiency (no "holes" in the computation space). There is no further restriction upon it except the URE condition, so it is a practical working model.

Chapter 5

Optimal Mapping Onto Lower-Dimensional Regular Arrays

In this chapter a new methodology of partitioning and mapping an N-D computation graph onto a given regular and fixed-mesh M-D array is proposed (M = N - K and K>1). The partitioning method of Chapter 4 is also applied here as an initial step to obtain a supernode polyhedron with the canonical dependencies. The resulting N-D supernode polyhedron is then, projected into a K-D time polyhedron and an M-D space polyhedron. The latter can be allocated within a given M-D array by scaling supernode partitioning parallelepipeds properly. The former is re-projected along a 1-D time domain by a valid minimum projection vector $\overline{p}$ which is derived to achieve high efficiency.

5.1 A Methodology for Partitioning and Mapping onto Lower Dimensional Array

A significant challenge to researchers is to partition and map a computational polyhedron onto a given regular processor array of lower dimension with fixed-shape and fixed-interconnections. We propose an integrated method for this purpose by building on the methodology developed in Chapter 4. We extend this method to the cases of given M-D processor arrays, where $M = N - K$ and $K > 1$. This extension is non-trivial when we consider a valid and efficient mapping onto a lower-dimensional array. We abandon the method of searching directly for a conflict-free $\overline{t}$ as in [58], [59], and [94], because of the obvious difficulties outlined in Chapter 2. Instead, we employ the method given briefly

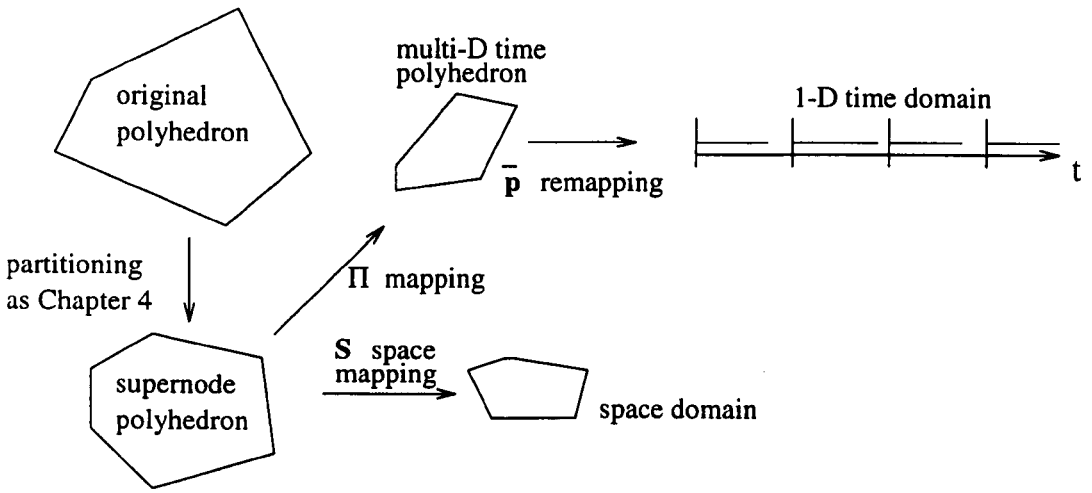


Figure 5.1: The Basic Idea of Lower dimensional Partitioning and Mapping Methods

below:

Step 1 Derive a positive expression basis from a dependency matrix.

Step 2 Find a unimodular space-time transformation $\mathbf{T}$ by which the supernode computational polyhedron is mapped into a K-D time domain and M-D space domain.

Step 3 Scale the parallelepiped in some dimensions, according to $\mathbf{S}$, such that the supernode polyhedron can be mapped within a given regular processor array.

Step 4 Find a valid and minimum projecting vector $\overline{\mathbf{p}}$ by which the K-D time domain of the supernode polyhedron can be mapped to a 1-D time domain.

Figure 5.1 shows that the supernode polyhedron is mapped within the processor array and a multi-D time domain. The key point is that the multi-D time polyhedron should be remapped by $\overline{\mathbf{p}}$ along an 1-D time domain as compact (short) as possible.

Since Step 1 and Step 3 have been discussed in the previous chapter. We deal with Step 2 and focus especially on Step 4 under the assumption that the supernode partitioning has been carried out so that a supernode space has been established and canonical dependencies have been obtained.

Some researchers have pointed out the necessity of avoiding data link conflict in synthesis of lower-dimensional array. In fact, this conflict can occur anywhere if the number

of links is less than that of dependency vectors. To come over the obstacle, all the outgoing data, no matter what variables and what sources, must and can be transferred out as a packet along a link as long as they go in the same direction, which will be discussed in Chapter 6 and 7. Nevertheless, the canonical dependencies can prevent the kind of data link conflict described in [58], [59] if links are provided in the same way as in [58], [59], because the greatest common divisor of the entries of each canonical dependency vector is one.

The rest of the chapter is organized as follows. Firstly, we select two families of the space-time transformation matrices for SUC and SBC. Next, we focus on mapping the K-D time domain to a 1-D domain efficiently by the derivation of a valid minimum projection vector. Finally optimisation is discussed and a number of examples are given to illustrate the operation and performance of the method.

5.2 The Transformation into K-D Time Domain and M-D Processor Array

5.2.1 Selecting a Family of T

Let a non-singular matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{S} \\ \mathbf{\Pi} \end{bmatrix}$$

be used to map the original N-D computational polyhedron to form a K-D time polyhedron (where $\mathbf{\Pi}$ is a $K \times N$ timing matrix) and a M-D processor array (where $\mathbf{S}$ is a $M \times N$ space mapping matrix).

For the SUC and SBC interconnection patterns, we choose $\mathbf{S}_{SUC}$ and $\mathbf{S}_{SBC}$ on the understanding that they should have as few non-zero entries as possible, and that $\mathbf{S}_{SUC}\mathbf{D}^s$ and $\mathbf{S}_{SBC}\mathbf{D}^s$ can be implemented with $\mathbf{P}_{SUC}$ and $\mathbf{P}_{SBC}$, respectively. In addition, it is desired that $\mathbf{S}_{SBC}\mathbf{D}^s$ has bi-directional data flows. Let

$$\begin{aligned} \mathbf{S}_{SUC} &= [\mathbf{O}_{M\times K}, \mathbf{I}_M] \\ \mathbf{S}_{SBC} &= [\mathbf{O}_{M\times K}, \mathbf{I}_M] + [\mathbf{O}_{M\times(K-1)}, -\mathbf{I}_M, \mathbf{o}] \end{aligned} \tag{5.1}$$

As before it can be verified that the $\mathbf{S}_{SUC}$ and $\mathbf{S}_{SBC}$ satisfy the requirements above, and that in each row of $\mathbf{S}_{SBC}\mathbf{D}^s$, there are the same number of "1" and "-1" values.

When choosing Π, a number of criteria must be taken into account. Firstly, the resulting $\mathbf{T}$ must be non-singular to produce a conflict-free mapping, and there must be enough time for data flow between processors. If these conditions are satisfied, we say that the Π is valid. To maintain high efficiency, the resulting $\mathbf{T}$ must be unimodular, which means there are no "holes" in the resulting computational polyhedron. Furthermore Π has the least non-zero entries and the non-zero entries are as small as possible, in order for the transformed supernode polyhedron to be as compact as possible. Such a Π is termed *priori-optimum.* The matrices Π_{SUC} and Π_{SBC} are selected as

$$\begin{aligned} \Pi_{SUC} &= [\mathbf{I}_K, \mathbf{L}_{K\times M}] \\ \Pi_{SBC} &= \begin{bmatrix} \mathbf{I}_K & \mathbf{L}_{(K-1)\times M} \\ & \overline{\mathbf{o}} \end{bmatrix} \end{aligned} \tag{5.2}$$

where $\mathbf{L}$ is a matrix such that there is one and only one "1" in each column. $\mathbf{T}_{SUC} = \begin{bmatrix} \mathbf{S}_{SUC} \\ \Pi_{SUC} \end{bmatrix}$ and $\mathbf{T}_{SBC} = \begin{bmatrix} \mathbf{S}_{SBC} \\ \Pi_{SBC} \end{bmatrix}$. For example, when N = 5, K = 3, we may write

$$\mathbf{T}_{SUC} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \mathbf{T}_{SBC} = \begin{bmatrix} 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Obviously $det(\mathbf{T}_{SUC}) = 1$ because $det(\begin{bmatrix} \mathbf{S}_{SUC} \\ \mathbf{\Pi}_{SUC} \end{bmatrix}) = det(\begin{bmatrix} \mathbf{\Pi}_{SUC} \\ \mathbf{S}_{SUC} \end{bmatrix})$ and the latter is a upper matrix with only "1"s on the diagonal. It can be proved that $det(\mathbf{T}_{SBC}) = 1$. In fact, adding the i-th column to the (i-1)-th column, $i = N-1, \ldots, K$ and swapping the first M rows with the left K rows, we observe that

$$\mathbf{T}_{SBC} \Longrightarrow \begin{bmatrix} \mathbf{O}_{M\times K} & \mathbf{I}_M \\ \mathbf{U}_K & \mathbf{M}_{K\times M} \end{bmatrix} \Longrightarrow \begin{bmatrix} \mathbf{U}_K & \mathbf{M}_{K\times M} \\ \mathbf{O}_{M\times K} & \mathbf{I}_M \end{bmatrix}$$

where $\mathbf{U}_K$ is a upper-triangular matrix. The key point to observe is that in the last row of Π_{SBC}, there is only one "1" in the (K-1)-th position.

As will be discussed in the next section, the K-D time domain will be projected into a 1-D time domain by a projecting vector $\overline{\mathbf{p}} = [p_0, \ldots, p_{K-2}, p_{K-1}]$. (usually, $p_{K-1} = 1$ and $p_0, \ldots, p_{K-2} > 1$). Thus, eventually, we still attain a timing vector $\overline{\mathbf{t}}$ such that

$$\overline{\mathbf{t}} = \overline{\mathbf{p}}\mathbf{\Pi} \quad \text{and} \quad \mathbf{T}' = \begin{bmatrix} \mathbf{S} \\ \overline{\mathbf{t}} \end{bmatrix}. \tag{5.3}$$

The transformed dependency matrix is $\mathbf{T'D^s} = \begin{bmatrix} \mathbf{SD^s} \\ \bar{\mathbf{t}}\mathbf{D^s} \end{bmatrix}$ and the last row $\bar{\mathbf{t}}\mathbf{D^s}$ indicates the time for data flow (or delay) among processors to implement the data dependency $\mathbf{SD^s}$.

In Chapter 4 theorem 4.3.1 was given to guarantee enough time for data delay, it can be seen that the condition is satisfied by $\bar{\mathbf{p}}\mathbf{\Pi}_{SUC}$ and $\bar{\mathbf{p}}\mathbf{\Pi}_{SBC}$. From eqn(5.2), we obtain $\bar{\mathbf{t}} = [p_0, \ldots, p_{K-1}, x_0, \ldots, x_{M-1}]$, where $x_i \in \{p_0, \ldots, p_{K-1}\}$ for SUC, and $x_i \in \{p_0, \ldots, p_{K-2}\}$ for SBC. In the case of SBC, for $j = K, \ldots, N-2$, $\sum_{i=0}^{M-1} |s_{i,j}| = 2$, while the corresponding t_j is larger than 1, since $t_j \in \{p_0, \ldots, p_{K-2}\}$.

Furthermore, it can be checked that the $\mathbf{\Pi}_{SUC}$ and $\mathbf{\Pi}_{SBC}$ contain the least and smallest non-zero entries (i.e., removal of any element will result in them being invalid). Therefore they are priori-optimum. It is pointed out that because $det(\mathbf{T}_{SBC}) = 1$, the further LSGP partitioning is no longer needed unlike the case of (N-1)-D array in Chapter 4.

5.2.2 Scaling the Supernode Parallelepiped

As proposed in Chapter 4, the positive expressing basis $\mathbf{E}$ is already derived and then should be scaled and approximated to an integral matrix $\mathbf{B}$ which defines the partitioning parallelpipes. The procedure of scaling is quite similar to that of Chapter 4. However, since $\mathbf{S}$ is lower-dimensional, there is more freedom in determining the partitioning parallelepiped, (i.e., the diagonal matrix $\mathbf{F} = diag(f_0, \cdots, f_{N-1})$ in eqn (4.29).). In fact, because $\mathbf{U} = \mathbf{S}_F\mathbf{W}$ and $\mathbf{S}_F = \mathbf{SF}$, for a $\mathbf{s}_i \in \mathbf{S}$ such that $\mathbf{s}_i = \mathbf{o}$, f_i has no contribution to $\mathbf{S}_F$ and $\mathbf{U}$, so the f_i will not be determined by the requirement of mapping within a given array. Instead the f_i is delimited only by an upper bound $\frac{1}{a_i^{max}}$. In other words, the size of the parallelepiped in the i-th dimension has a lower bound of a_i^{max}, which is required by the canonical dependencies in supernode space.

Observe that in the (N-1)-D SUC case, there is only one free dimension caused by the 1-D time vector. But for the lower-D case, there are K free dimensions for SUC and K-1 free dimensions for SBC. The problem is how to determine the actual sizes of the supernode parallelepiped with respect to the relevant dimensions with these degrees of freedom. From the point of view of theory, the sizes should be as small as possible to achieve the minimum total executing time. But in practice, the smaller granularity means

more supernodes, and hence more overhead operations. A tradeoff has to be made.

5.3 Maximum Local K-D Time Domain

The supernode polyhedron, $\mathcal{P}^s$, produced by the partitioning transformation can be defined by its set of vertices, $\mathbf{V}^s$. However, it can be understood that this supernode polyhedron is not really the polyhedron which will be computed in one processor. In fact, a particular processor $\mathbf{p}$ will be assigned a part of $\mathcal{P}^s$, called local supernode polyhedron $\mathcal{P}^s_{\mathbf{p}}$. With $\mathbf{S}$, we can split $\mathcal{P}^s$ into $\mathcal{P}^s_{\mathbf{p}}$'s for each processors. However, before doing so, some preliminary work should be presented.

5.3.1 Intersecting a Polyhedron with a Hyperplane

For any face of a n-D polyhedron, the facet is a more precise description. A facet is plotted by a number of edges. Each of the edges is defined by a pair of vertices. So the facet is determined by a number, say n_v, of vertices $\mathbf{v}_j$, $n_v \geq n$. When a facet is determined by $\mathbf{v}_0, \cdots, \mathbf{v}_{n_v-1}$, we define the facet space range $\mathbf{g}^u_i$ and $\mathbf{g}^l_i$ in the i-th dimension as

$$\mathbf{g}^u_i = \max_{j=0}^{n_v-1} v_{i,j}, \text{ and } \mathbf{g}^l_i = \min_{j=0}^{n_v-1} v_{i,j}, \qquad i = 0, \cdots, n-1 \tag{5.4}$$

where $v_{i,j} \in \mathbf{v}_j$.

In the example above, we saw that a n-D polyhedron is intersected by a hyperplane, which produces a (n-1)-D sub-polyhedron. The vertices and faces of the sub-polyhedron must be derived. Fortunately, this job is not too difficult, since in our case, the intersecting hyperplane is perpendicular to a coordinate axis. A hyperplane $i_r = h$ will intersect all facets whose space range is such that $\mathbf{g}^l_r \leq h \leq \mathbf{g}^u_r$. Now substituting h as i_r into the faces corresponding to these facets, we obtain the faces of the sub-polyhedron.

It can be understood that the vertices of the sub-polyhedron can only be the intersection of the hyperplane with the edges of the original convex hull (when $n > 2$), not with the faces. That is, any point on a facet but not on an edge of the facet cannot be a vertex of a sub-polyhedron of the original, see Fig 5.2 where a 3-D polyhedron "ABCD" is intersected by plane p, creating 2-D polyhedron "abc". It can be seen that points a, b and c are the intersections of edges AD, BD and CD with plane p, respectively. More strictly, if point

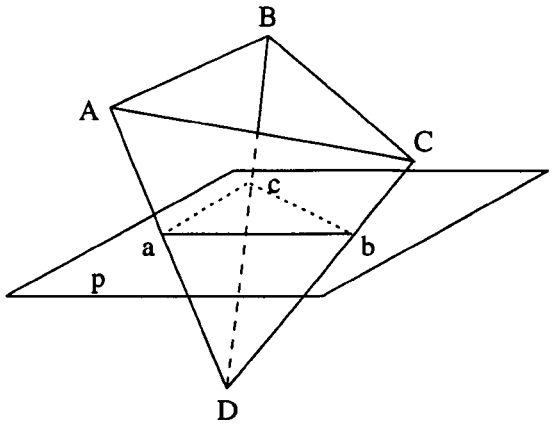


Figure 5.2: The Intersection of Polyhedron with a Hyperplane.

$\mathbf{j} = [j_0, \ldots, j_{r-1}, h, j_{r+1}, \ldots, j_n]^T$ is such a point on a facet, it is always possible to find a short enough vector $\Delta\mathbf{j} = [\Delta j_0, \ldots, \Delta j_{r-1}, 0, \Delta j_{r+1}, \ldots, \Delta j_n]^T$ such that $\mathbf{j} + \Delta\mathbf{j}$ and $\mathbf{j} - \Delta\mathbf{j}$ are still on the facet. After intersection, all $[j_0, \ldots, j_{r-1}, j_{r+1}, \ldots, j_n]^T$, $[j_0 + \Delta j_0, \ldots, j_{r-1} + \Delta j_{r-1}, j_{r+1} + \Delta j_{r+1}, \ldots, j_n + \Delta j_n]^T$ and $[j_0 - \Delta j_0, \ldots, j_{r-1} - \Delta j_{r-1}, j_{r+1} - \Delta j_{r+1}, \ldots, j_n - \Delta j_n]^T$ are also on the surface of the sub-polyhedron. Therefore, $[j_0, \ldots, j_{r-1}, j_{r+1}, \ldots, j_n]^T$ cannot be a vertex, because it lies between two points. On the other hand, only the intersection of a hyperplane with an edge determines a point. Then, for all edges, if the pair of vertices $\mathbf{v}^1$ and $\mathbf{v}^2$ is such that $v_r^1 \leq h \leq v_r^2$ or $v_r^2 \leq h \leq v_r^1$, a vertex $\mathbf{w}$ of the sub-polyhedron is determined as

$$w_i = v_i^1 + \frac{h - v_r^1}{v_r^2 - v_r^1}(v_i^2 - v_i^1), \quad i = 0, \cdots, r-1, r+1, \cdots, n-1 \tag{5.5}$$

Obviously, the vertices of the sub-polyhedron are not necessarily integral vectors.

5.3.2 Maximum Local Supernode Domain

Notice that the computation task splitting is easily done in the processor-time domain. Therefore, the process of splitting consists of a domain transformation from supernode to processor-time, then splitting, and finally domain transformation from processor-time to supernode.

Step 1 Map $\mathcal{P}^s$ into the processor-time domain by $\mathbf{U}$ to form a N-D processor-time polyhedron $\mathcal{P}^q = \mathbf{U}\mathcal{P}^s$, defined by its vertices $\mathbf{V}^q = \mathbf{U}\mathbf{V}^s$. The $\mathbf{U}$ can be any unimodular matrix with $\mathbf{S}$ as its first M row vectors. Thus any of the $\mathbf{T}$ in the above section can be used. Suppose the processor-time domain is indexed by $\mathbf{j}$.

Step 2 Use M hyperplanes $j_0 = p_0, \cdots, j_{M-1} = p_{M-1}$ to intersect the N-D $\mathcal{P}^q_{\mathbf{p}}$ in series, determining a $\mathcal{P}^q_{\mathbf{p}}$ assigned onto a particular processor $\mathbf{p} = [p_0, \cdots, p_{M-1}]^T$. That is, firstly, intersecting the N-D $\mathcal{P}^q_{\mathbf{p}}$ with $j_0 = p_0$ results in a (N-1)-D sub-polyhedron; then, intersecting the (N-1)-D sub-polyhedron with $j_1 = p_1$ results in a (N-2)-D sub-polyhedron, and so on. At the end, we obtain a (N-M)-D sub-polyhedron indexed by $j_M, \cdots, j_{N-1}$, as well as its vertices $\mathbf{v}^j_{\mathbf{p}}$'s which have a form of $[v^j_M, \cdots, v^j_{N-1}]^T$. Collect all $\mathbf{v}^j_{\mathbf{p}}$'s into $\mathbf{V}^j_{\mathbf{p}}$ which is the vertices of (N-M)-D sub-polyhedron with respect to processor $\mathbf{p}$.

Step 3 Remap the (N-M)-D sub-polyhedron back to the supernode space. Attaching $\mathbf{p}$ to the front of $\mathbf{v}^j_{\mathbf{p}}$'s, we obtain the vertices, $\mathbf{v}^p_{\mathbf{p}}$'s, of the sub-polyhedron in the N-D processor-time domain, i.e., $\mathbf{v}^p_{\mathbf{p}} = [p_0, \cdots, p_{M-1}, v^j_M, \cdots, v^j_{N-1}]^T$. Now, compute $\mathbf{v}^s_{\mathbf{p}} = \mathbf{U}^{-1}\mathbf{v}^p_{\mathbf{p}}$ for all $\mathbf{v}^p_{\mathbf{p}}$'s, which confine the local supernode polyhedron assigned to the processor $\mathbf{p}$. Collect all $\mathbf{v}^s_{\mathbf{p}}$ forming $\mathbf{V}^s_{\mathbf{p}}$ which is the vertices of $\mathcal{P}^s_{\mathbf{p}}$.

Step 4 Repeat this procedure for every processor.

Note that $\mathcal{P}^s_{\mathbf{p}}$ is independent of $\mathbf{U}$ as long as the condition in Step 1 for contructing U holds. Unfortunately, for a large processor array, there are many $\mathcal{P}^s_{\mathbf{p}}$'s. In some cases, the $\mathcal{P}^s_{\mathbf{p}}$'s share the same shape subject to a translation and we can take only one (only the shape of $\mathcal{P}^s_{\mathbf{p}}$ affects the following operations). Otherwise we can merge them into a maximum local supernode polyhedron.

At first, translate $\mathcal{P}^s_{\mathbf{p}}$'s to the origln point by substracting the average value of each row vector of $\mathbf{V}^s_{\mathbf{p}}$, and then merge all the central-translated $\mathbf{V}^s_{\mathbf{p}}$ into a large set of points. Finally, feed the set of points to an algorithm which can produced the convex hull of the set (A convex-hull algorithm [65] [98] can be used which gives the vertex set $\mathbf{V}^K$ and all faces, characterized by their normal equations, of the convex hull). The maximum local polyhedron and its vertices are indicated with $\mathcal{P}^l$ and $\mathbf{V}^l$, respectively.

5.3.3 Mapping to K-D Time Domain

Because the number of the integer points (supernodes) closed within $\mathcal{P}^l$ is finite, the $\mathbf{V}^l$ is also the polytope of $\mathcal{P}^l$, and, by definition, any non-vertex point $\mathbf{s}$ (integer) of $\mathcal{P}^l$ can

be expressed by convex combinations of $\mathbf{V}^l$, that is,

$$\mathrm{s} = \sum_{j=0}^{n_v-1} \lambda_j \mathrm{v}_j^l \tag{5.6}$$

where $\mathrm{v}_j^l \in \mathbf{V}^l$, and $\sum_{j=0}^{n_v-1} \lambda_j = 1$ and $0 \leq \lambda_j < 1$.

By applying $\mathbf{\Pi}^K$ upon $\mathcal{P}^l$, we obtain a K-D polyhedron $\mathbf{\Pi}^K\mathcal{P}^l \longrightarrow \mathcal{P}^K$. Let $\mathbf{V}^K$ indicate the vertex set of $\mathcal{P}^K$ (henceforth, "polyhedron" simply indicates such a polyhedron in the K-D time domain). Note that $\mathbf{V}^K \subseteq \mathbf{\Pi}^K\mathbf{V}^l$ and this is true for any arbitrary $\mathbf{\Pi}^K$. In fact, applying $\mathbf{\Pi}^K$ on eqn (5.6), we have $\mathbf{\Pi}^K\mathrm{s} = \sum_{j=0}^{n_v-1} \lambda_j \mathbf{\Pi}^K \mathrm{v}_j^l$ where λ_j's are the same as those of eqn (5.3.3). This means that $\mathbf{\Pi}^K\mathrm{s}$ cannot be a vertex of $\mathcal{P}^K$, so we can only figure out the vertices of $\mathcal{P}^K$ from the $\mathbf{\Pi}^K\mathrm{v}_j^l$'s. A convex-hull algorithm is applied to the set of points $\mathbf{\Pi}^K\mathbf{V}^l$ to find its convex hull $\mathcal{CH}^K$.

5.4 Valid Minimum Projecting Vector

It is easy to determine a hypercube with bounds $b_0, \ldots, b_{K-1}$ to contain the polyhedron, i.e.

$$b_j = \max_i^{n_K-1} v_{i,j}^K - \min_i^{n_K-1} v_{i,j}^K + 1; \qquad j = 0, \ldots, K-1 \tag{5.7}$$

where $v_{i,j}^K \in \mathrm{v}_j^K \in \mathbf{V}^K$ and n_K is the cardinality of $\mathbf{V}^K$.

As mentioned before, the straightforward method (see [104]) to map the K-D polyhedron to a 1-D domain is to construct a projecting vector $\overline{\mathrm{p}} = [p_0, \ldots, p_{K-1}]$, where

$$p_{K-1} = 1, \text{ and } p_i = b_{i+1}p_{i+1}. \qquad i = K-2, \ldots, 0 \tag{5.8}$$

Obviously, when the shape of the polyhedron is not a regular hypercube, the efficiency can be very low, because too much of the surrounding hypercube is identified with null computations. The problem is that the elements of $\overline{\mathrm{p}}$ defined in eqn (5.8) rely on a sufficient condition for avoiding conflict, but not a necessary one.

An example in Figure 5.3.(a), which is rather extreme, illustrates the possibility of decreasing the elements of $\overline{\mathrm{p}}$ while keeping the conflict-free property. The method in eqn (5.8) yields $\overline{\mathrm{p}}$ = [6,1]. However, it is easy to verify that a vector [2,1] is sufficient and necessary to achieve a conflict-free mapping. Intuitively this is because the band width

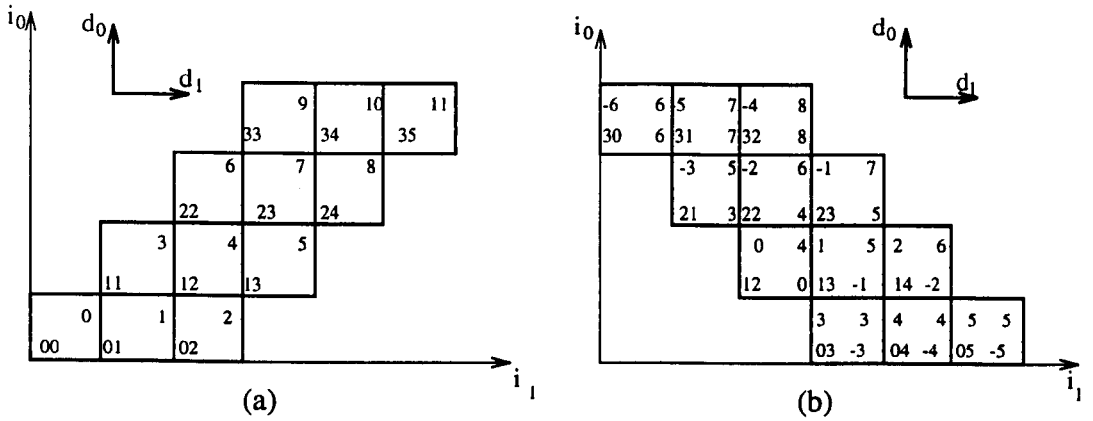


Figure 5.3: 2-D Time Polyhedron. For both (a) and (b), the numbers at the bottom-left corner of a square is the index of the node and the number at the top-right corner, the allocation in the 1-D domain projected by $\overline{p}$ = [2,1]. In (b), the numbers at the top-left and bottom-right corners are the allocations projected by $\overline{p}$ = [-2,1] and [2,-1], respectively.

of the polyhedron is 3. This observation is not true in general as demonstrated by the polyhedron of Figure 5.3.(b) where the polyhedron is simply the reverse of Figure 5.3.(a), the same $\overline{p}$ causes a conflicting mapping.

From Figure 5.3.(b), it is also clear that a small $\overline{p}$ (e.g., [-2,1]) is possible to avoid conflicts. But that the $\overline{p}$ is unacceptable, because it violates the dependency relation in the i_0 direction. A naive attempt to fix the problem is to reverse the polyhedron along the i_1 axis (i.e., a transformation $i_1' = -i_1$, such that Figure 5.3.(b) is the same as (a) in shape). This transformation is equivalent to using $\overline{p}$ = [2,-1]. However, since the execution sequence is from left to right, this projection, or the transformation, violates the dependency relation in the i_1 direction.

Definition 5.4.1 *A projecting vector $\overline{p}$ is said to be valid* $\forall\ \mathbf{j}^1, \mathbf{j}^2 \in \mathcal{P}^K$, *if* $\forall\ \mathbf{j}^1 \succ \mathbf{j}^2$, $\overline{p}\mathbf{j}^1 > \overline{p}\mathbf{j}^2$; *and to be minimum if any decrease of its elements will cause the projection to be invalid.*

Now it can be understood that "valid" guarantees no violation of data dependency by such a mapping. Henceforth, $\overline{p}$ indicates the valid minimum projecting vector. We use an example to illustrate the idea of obtaining a valid minimum projecting vector.

Example 5.4.1 *The Projection of 3-D Time Domain to 1-D Time Domain*

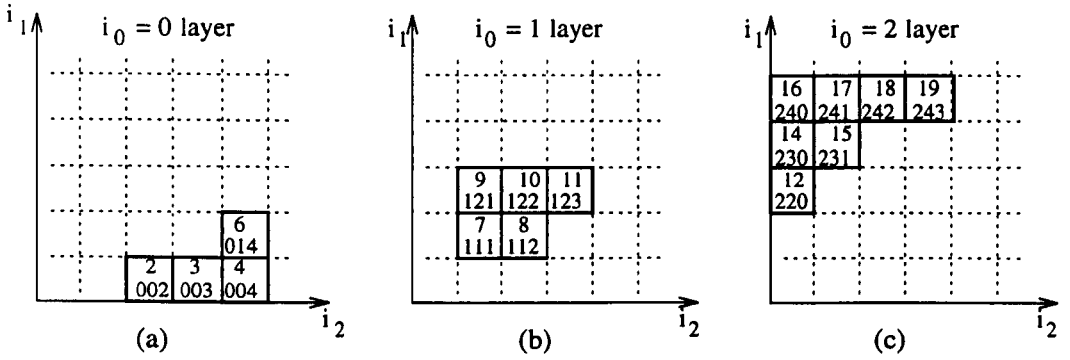


Figure 5.4: The layouts of a 3-D Polyhedron. The number at the bottom of each box is the index of the node and the number at the top, the allocation in the 1-D domain projected by $\overline{\mathbf{p}} = [4,2,1]$.

A polyhedron is defined by a set of 6 vertices

$$\begin{bmatrix} 0 & 0 & 0 & 2 & 2 & 2 \\ 0 & 0 & 1 & 2 & 4 & 4 \\ 2 & 4 & 4 & 0 & 0 & 3 \end{bmatrix}$$

The layouts of the 3-D polyhedron intersected by three planes $i_0 = 0$, 1, 2 are shown in Figure 5.4.

Now consider the determination of the minimum $\overline{\mathbf{p}} = [p_0, p_1, p_2]$, in the i_2 direction, the series of adjacent nodes implies that $p_2 = 1$ is required to avoid conflict. We need to determine the minimum p_1. Since p_1 is used to project the rows of different i_1, it is desired that the first node, say $\mathbf{j}^f = [i_0, i_1 + 1, f]^T$, of the $(i_1 + 1)$ row is just behind the last node of the i_1 row, say $\mathbf{j}^l = [i_0, i_1, l]^T$, that is $[p_0, p_1, p_2]\mathbf{j}^f = [p_0, p_1, p_2]\mathbf{j}^l + 1$. Therefore

$$p_1 = l - f + 1;$$

However, there are a number of adjacent rows in each layer of the polyhedron, we must choose the maximum p_1. Both the pair of rows $i_1 = 1$ and $i_1 = 2$ on the layer $i_0 = 1$ and the pair $i_1 = 3$ and $i_1 = 4$ on the layer $i_0 = 2$ produce a $p_1 = 2$.

Finally, we derive the minimum p_0. Because p_0 is used to project the layers of different i_0, we hope that p_0 is chosen such that each layer can be projected one by one. That is, if the last node of a layer is $\mathbf{j}^l = [i_0, l_1, l_2]^T$ and the first node of the layer above it is

$\mathbf{j}^f = [i_0 + 1, f_1, f_2]^T$, then $[p_0, p_1, p_2]\mathbf{j}^f = [p_0, p_1, p_2]\mathbf{j}^l + 1$. So, similar to p_1

$$p_0 = ([l_1, l_2] - [f_1, f_2]) \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} + 1$$

The last node of the layer $i_0 = 0$ is $[0, 1, 4]^T$, the first and the last nodes of the layer $i_0 = 1$ are $[1, 1, 1]^T$ and $[1, 2, 3]^T$, respectively. Similarly the first in layer $i_0 = 2$ is $[2, 2, 0]^T$, so the pair $[0, 1, 4]^T$ and $[1, 1, 1]^T$ produces a $p_0 = 4$. The pair of nodes $[1, 2, 3]^T$ and $[2, 2, 0]^T$ do the same. Thus we obtain a valid projection vector $\overline{\mathbf{p}} = [4, 2, 1]$. And it is easy to verify that a decrease of any elements of the $\overline{\mathbf{p}}$ will cause conflicts. The instance of execution for each node projected by $\overline{\mathbf{p}}$ is also shown in Figure 5.4. The polyhedron has 16 nodes and the executing time is 18 steps. In contrast, by means of the conventional method of eqn (5.8), the projecting vector will be $\overline{\mathbf{p}} = [25, 5, 1]$ and the corresponding executing time is 72 steps. In this example, the efficiency is improved significantly, from 0.22 to 0.89. □

It must be pointed out that the time mapping efficiency is not the same as the overall efficiency of the whole processor array. The former indicates the internal efficiency of a processor during its active period, while the later is not only proportional to the internal efficiency of a processor but also depends on the distribution of the active periods of every processor. The distribution of the active periods of processors is determined by data dependency, that is, some processor must be inactive until other processors provide necessary data.

5.5 General Methodology for Valid Minimum Projecting Vector

We will develop a general methodology for deriving $\overline{\mathbf{p}}$. Unfortunately, this is not an easy algorithm to describe and requires careful reading.

5.5.1 The First and Last Nodes of Polyhedrons

Before giving the method of deriving $\overline{\mathbf{p}}$, more information about an arbitrary polyhedron is required.

Definition 5.5.1 *A (K-r)-D subpolyhedron, noted as $\mathcal{P}^{K-r}$, is a polyhedron created by intersecting a K-D polyhedron with r hyperplanes $i_0 = h_0$, $i_1 = h_1$, ..., $i_{r-1} = h_{r-1}$ step by step. For uniformity, the indices of $\mathcal{P}^{K-r}$ are marked $i_r, \ldots, i_{K-1}$.*

A pair of two adjacent (K-r-1)-D subpolyhedra are such that the lower, $\mathcal{P}_0^{K-r-1}$, is produced with hyperplanes $i_0 = h_0$, ..., $i_{r-1} = h_{r-1}$, $i_r = h_r$, the upper, $\mathcal{P}_1^{K-r-1}$, with $i_0 = h_0$, ..., $i_{r-1} = h_{r-1}$, $i_r = h_r + 1$.

Definition 5.5.2 *An integral point in a K-D polyhedron $\mathcal{P}$ (or subpolyhedron) is referred to as the first node, $\mathbf{f}^K$, if $\forall$ integral points $\mathbf{j} \in \mathcal{P}$, $\mathbf{j} \succ \mathbf{f}^K$. Similarly, an integral point is referred to as the last node, $\mathbf{l}^K$, if $\forall$ integral point $\mathbf{j} \in \mathcal{P}$, $\mathbf{l}^K \succ \mathbf{j}$.*

If the polyhedron represents a nested loop computation, $\mathbf{f}^K$ is the first iteration to be executed, and $\mathbf{l}^K$, the last one. If the polyhedron is the original convex hull, so that all the vertices are integers, $\mathbf{f}^K$ and $\mathbf{l}^K$ must belong to the vertex set. By the definition it is easy to look for $\mathbf{f}^K$ and $\mathbf{l}^K$ in the vertex set . However, if the polyhedron is a (K-r)-D subpolyhedron, we require a different method to find $\mathbf{f}^{K-r}$ and $\mathbf{l}^{K-r}$.

Consider $\mathbf{f}^{K-r} = [f_r^{K-r}, \ldots, f_{K-1}^{K-r}]^T$ for $\mathcal{P}^{K-r}$. Start from the index i_r. First, find the lower bound b_r of $\mathcal{P}^{K-r}$ in the i_r direction according to $b_r = \min_j w_{r,j}$, where $w_{r,j} \in \mathbf{w}_j$ and $\mathbf{w}_j$ is a vertex of $\mathcal{P}^{K-r}$. Let $f_r^{K-r} = \lceil b_r \rceil$. Then intersect $\mathcal{P}^{K-r}$ with the hyperplane $i_r = f_r^{K-r}$. We obtain further a (K-r-1)- D subpolyhedron $\mathcal{P}^{K-r-1}$. Next, find the lower bound b_{r+1} of the $\mathcal{P}^{K-r-1}$ and let $f_{r+1}^{K-r} = \lceil b_{r+1} \rceil$. Continue the recursive procedure until f_{K-1}^{K-r} is found. If $\lceil b_{r+1} \rceil$ is outside $\mathcal{P}^{K-r-1}$, this means that the intersection by $i_r = \lceil b_r \rceil$ is invalid and we must reassign $f_r^{K-r} = \lceil b_r \rceil + 1$, and intersect $\mathcal{P}^{K-r}$ by $i_r = \lceil b_r \rceil + 1$. The process is repeated until $\lceil b_{r+1} \rceil$ is inside $\mathcal{P}^{K-r-1}$. The process is illustrated in Figure 5.5. The intersection made by $i_r = \lceil b_r \rceil$ is invalid, because $\lceil b_{r+1} \rceil$ is outside the $\mathcal{P}^{K-r-1}$ of Figure 5.5.(b). Letting $f_r^{K-r} = \lceil b_r \rceil + 1$, we try again. The $\mathcal{P}^{K-r-1}$ of Figure 5.5.(c) is acceptable. Thus, finally, we obtain $\mathbf{f}^{K-r} = [2, 2]^T$.

The same procedure is applicable for determining $\mathbf{l}^K$, but we replace "lower bound" with "upper bound" and replace $\lceil\ \rceil$ with $\lfloor\ \rfloor$.

In a (K-r)-D subpolyhedron, for a pair of adjacent (K-r-1)-D subpolyhedron, a *Cross-layer distance* $\mathbf{d}^{K-r}$ indicates the difference of the first node of the upper subpolyhedron

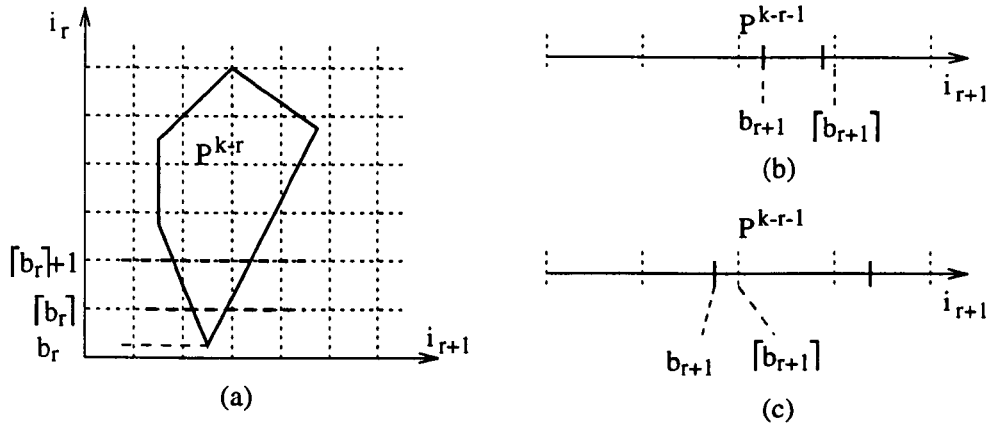


Figure 5.5: Finding $\mathbf{f}^{K-r}$ for $\mathcal{P}^{K-r}$. (a) is the $\mathcal{P}^{K-r}$ being intersected by two planes. (b) and (c) are the resulting subpolyhedra $\mathcal{P}^{K-r-1}$ produced by intersecting with $i_r = \lceil b_r \rceil$ and $i_r = \lceil b_r \rceil + 1$, respectively. The selection of the hyperplane which determines $\mathbf{f}^{K-r}$ is effected by the resulting subpolyhedra, that is, begin with $i_r = \lceil b_r \rceil$, if $\lceil b_{r+1} \rceil$ is outside the resulting $\mathcal{P}^{K-r-1}$, modify $i_r = \lceil b_r \rceil + 1$.

and the last node of the lower one, i.e., $\mathbf{d}^{K-r} = \begin{bmatrix} 1 \\ \mathbf{f}_1^{K-r-1} \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{l}_0^{K-r-1} \end{bmatrix}$.

5.5.2 Deriving $\overline{\mathbf{p}}$

From the Example 5.4.1, we know that the process to derive $\overline{\mathbf{p}}$ is a recursive procedure from p_{K-1} to p_0. In a (K-r)-D subpolyhedron, for a pair of adjacent (K-r-1)-D subpolyhedron, p_r should be determined such that no overlapped-projection is made, that is

$$\overline{\mathbf{p}}^{K-r}\mathbf{d}^{K-r} \geq 1 \implies p_r \geq \overline{\mathbf{p}}^{K-r-1}(\mathbf{l}_0^{K-r-1} - \mathbf{f}_1^{K-r-1}) + 1 \tag{5.9}$$

where $\overline{\mathbf{p}}^{K-r} = [p_r, \overline{\mathbf{p}}^{K-r-1}]$. Furthermore this should hold for all possible (K-r)-D subpolyhedra and for all possible pairs of two adjacent (K-r-1)-D subpolyhedra. We must find the lexicographically greatest $\mathbf{d}^{K-r}$. In principle, this can be achieved by trying all the intersecting hyperplanes $i_0 = h_0$, ..., $i_r = h_r$, but obviously, this approach is not practical when the size of the convex hull is large.

Fortunately, it is sufficient to search for the potential p_r by considering only the intersecting hyperplanes associated with the vertices of all possible $\mathcal{P}^{K-s}$, $s = 0, \ldots, r+1$. We explain the method in real space for simplicity.

Lemma 5.5.1 *p_r is a piecewise linear function of $h_0, \ldots, h_r$.*

Proof In fact, $\mathbf{f}^{K-r-1}$ and $\mathbf{l}^{K-r-1}$ belong to the vertices of the subpolyhedra $\mathcal{P}^{K-r-1}$. Thus each of them is the intersection of K-r-1 faces of $\mathcal{P}^{K-r-1}$. The K-r-1 faces of $\mathcal{P}^{K-r-1}$ are produced from the K-r-1 faces of the original $\mathcal{CH}^K$, intersected by r+1 hyperplanes $i_0 = h_0, \ldots, i_r = h_r$. Let $\mathbf{h} = [h_0, \ldots, h_r]^T$. For the $\mathcal{P}_0^{K-r-1}$, from the K-r-1 equations related to $\mathbf{l}_0^{K-r-1}$, $i = 0, \ldots, K-r-1$

$$[a_{i,0}, \ldots, a_{i,r}]\mathbf{h} + [a_{i,r+1}, \ldots, a_{i,K-1}]\mathbf{l}_0^{K-r-1} + a_{i,K} = 0$$

we see that $\mathbf{l}_0^{K-r-1}$ can be expressed in a form of $\mathbf{l}_0^{K-r-1} = \mathbf{b}^l + \mathbf{A}^l\mathbf{h}$. Similarly, for the $\mathcal{P}_1^{K-r-1}$ which is produced by using $i_0 = h_0, \ldots, i_{r-1} = h_{r-1}$ and $i_r = h_r + 1$, the $\mathbf{f}_1^{K-r-1}$ has the same form, $\mathbf{f}_1^{K-r-1} = \mathbf{b}^f + \mathbf{A}^f\mathbf{h}$. Therefore p_r is a linear function of $h_0, \ldots, h_r$ as $p_r = a_0h_0, \ldots, a_rh_r + a_{r+1}$. But, when $h_0, \ldots, h_r$ vary, they may go outside the boundaries of some facets which are associated with the $\mathbf{l}_0^{K-r-1}$ or those with the $\mathbf{f}_1^{K-r-1}$. Thus some of the faces become unrelated, while other faces will join in. When the membership of the related faces change, the linear function for p_r will change, too, that is, $a_0, \ldots, a_{r+1}$ are also piecewise linear functions of $h_0, \ldots, h_r$. □

If $h_0, \ldots, h_{i-1}$ are fixed, p_r can be rewritten as $p_r = a_ih_i+, \ldots, +a_rh_r + a'_{r+1}$. For a linear function, there is no critical point for local maximum or minimum value within its area of definition. However for a piecewise linear function, the critical points appear only at those points where changes of any of $h_0, \ldots, h_r$ will cause a change of the function to another set of $a_0, \ldots, a_{r+1}$. If this were not the case, when h_0 varies while a_0 keeps constant, p_r is still a linear function with respect to h_0, so there is no critical point in an small interval around the h_0.

Theorem 5.5.1 *If* $\mathbf{h}^c = [h_0^c, \ldots, h_r^c]^T$ *is a critical point, then* $\forall i$, h_i^c *(or* $h_i + 1$ *when* $i = r$*) must be at a vertex of the* $\mathcal{P}^{K-i}$*, where "at a vertex" means* h_i^c *is the* v_i *of the vertex* $[v_i, \ldots, v_{K-1}]^T$.

Proof: The proof proceeds by induction. At first, fix $h_0, \ldots, h_{r-1}$ to produce a $\mathcal{P}^{K-r}$, then $p_r = a_rh_r + a'_{r+1}$. The p_r cannot achieve a local maximum value unless h_r^c is at a vertex of $\mathcal{P}^{K-r}$ where any movement of h_r will cause a change of a_r. If not, when h_r

varies in an interval small enough, the set of facets intersected by h_r remains unchanged, because h_r does not move outside these bounds in the i_r direction (where the bound is the component of a vertex in the given direction), a_r is unchanged.

Suppose $\forall j, i+1 \leq j \leq r$, h_j^c is at a vertex of $\mathcal{P}^{K-j}$. When h_i varies, the $h_{i+1}^c, \ldots, h_r^c$ must change as well to keep the property, since their corresponding vertices are changed. This implies that if the h_i does not move over the bounds of the facets it currently intersects, there is no change of associated facets of $\mathcal{P}^{K-i}$, so h_{i+1}^c is still at the vertex of the same set of facets, and so on for $h_{i+2}^c, \ldots, h_r^c$. Thus, $a_i, \ldots, a_r$ remain unchanged. In contrast, $h_{i+1}^c, \ldots, h_r^c$ determined in this way are linear functions of h_i (because a vertex of the $\mathcal{P}^{K-i-1}$ is a linear function of the intersecting hyperplane $i_i = h_i$, and so on. So there is a recursive linear relation between h_i and $h_{i+1}^c, \ldots, h_r^c$). Then $p_r = a_i h_i +, \ldots, + a_r h_r + a'_{r+1}$ can be rewritten as $p_r = a'_i h_i + a''_{i+1}$. Therefore, when $i = r$, to achieve a local maximum value of p_r, h_i^c must be allocated at a vertex of the $\mathcal{P}^{K-i}$. □

By the above theorem, unlike [94] and [34] whose searching spaces are related to the size (volume) of the original polyhedron, the computing complexity of our method is related only to the number of the vertices of a series of polyhedra. For any sensible computational polyhedron, the number of vertices is quite limited, no matter how large the polyhedron becomes. This is the major computing advantage of our method over others.

For a subpolyhedron $\mathcal{P}^{K-i}$ with a vertex set $\mathbf{V}^{K-i}$, we form a set $H^{K-i} = \{v_0, \ldots, v_{n_h}\}$ of candidates of intersecting hyperplanes by collecting all different $v_{i,j} \in \mathbf{v}_j \in \mathbf{V}^{K-i}$, where n_h is the cardinality of H^{K-i}. Now, the operation to derive $\overline{\mathbf{p}}$ can be described as follows.

Step 1 Create a set of polyhedra $\mathcal{SP}_i$ which consists of all subpolyhedra $\mathcal{P}^{K-i}$ ($\mathcal{SP}_0 = \{\mathcal{P}^K\}$). When $i > 0$, $\mathcal{SP}_i$ is produced from $\mathcal{SP}_{i-1}$. For each of the subpolyhedra of $\mathcal{SP}_{i-1}$, say $\mathcal{P}^{K-i+1}$, find intersecting hyperplanes $i_{i-1} = v_j$ for all $v_j \in H^{K-i+1}$, producing n_h $\mathcal{P}^{K-i}$'s; collect all such created $\mathcal{P}^{K-i}$'s into $\mathcal{SP}_i$. K-1 $\mathcal{SP}$'s are created, including $\mathcal{SP}_0$.

Suppose that $\overline{\mathbf{p}}^{(K-r-1)} = [p_{r+1}, \ldots, p_{K-1}]$ have been determined. Now, derive the p_r. For brevity, let $r' = K - r$.

Step 2 Take one subpolyhedron of $\mathcal{SP}_r$, say $\mathcal{P}^{r'}$. For each $v_j \in H^{r'}$, make three intersecting hyperplanes, $i_r = v_j - 1$, $i_r = v_j$ and $i_r = v_j + 1$. Then intersect the $\mathcal{P}^{r'}$ with the three parallel hyperplanes, obtaining $\mathcal{P}_{-1}^{(r'-1)}$, $\mathcal{P}_0^{(r'-1)}$ and $\mathcal{P}_1^{(r'-1)}$.

Step 3 Find the first and the last nodes for each of the three subpolyhedra, $\mathbf{f}_{-1}^{(r'-1)}$ and $\mathbf{l}_{-1}^{(r'-1)}$, $\mathbf{f}_0^{(r'-1)}$ and $\mathbf{l}_0^{(r'-1)}$, and $\mathbf{f}_1^{(r'-1)}$ and $\mathbf{l}_1^{(r'-1)}$. Calculate a possible p_r

$$p_r = max\{\overline{\mathbf{p}}^{(r'-1)}(\mathbf{l}_{-1}^{(r'-1)} - \mathbf{f}_0^{(r'-1)}), \overline{\mathbf{p}}^{(r'-1)}(\mathbf{l}_0^{(r'-1)} - \mathbf{f}_1^{(r'-1)})\} + 1 \tag{5.10}$$

Step 4 Repeat Step 2 and Step 3 for all subpolyhedra of $\mathcal{SP}_r$. Let the p_r be the greatest value evaluated by eqn (5.10).

Example 5.5.1 *The Projection of 4-D Time Domain to 1-D Domain*

An arbitrary example is given to show the method. The original $\mathcal{P}^4$ is

$$\begin{bmatrix} 0 & 5 & 0 & 5 & -5 & 0 & 5 & 0 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 & 10 \\ 0 & 20 & 10 & 10 & 20 & 20 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -5 & 0 & -5 & -5 & 0 \end{bmatrix}$$

The expressions of the faces are omitted, for brevity. At first we create the $\mathcal{SP}$'s. $\mathcal{SP}_0 = \{\mathcal{P}^4\}$. The H^4 of the $\mathcal{P}^4$ is $\{-5, 0, 5, 10\}$. So four hyperplanes $i_0 = -5, 0, 5, 10$ are used to intersect $\mathcal{P}^4$. However, since the intersections with -5 and 10 produce only a single point polyhedron, they are omitted. Two polyhedra $\mathcal{P}_0^3$ and $\mathcal{P}_1^3$are produced

$$\begin{bmatrix} 0 & 15 & 15 & 10 & 10 & 20 & 20 \\ 0 & -2.5 & -2.5 & -5 & -5 & -5 & -5 \\ 0 & 5 & 7.5 & 5 & 0 & 7.5 & 10 \end{bmatrix} \begin{bmatrix} 10 & 10 & 5 & 0 & 20 & 5 & 0 \\ 0 & 0 & -2.5 & 0 & -5 & -2.5 & 0 \\ 5 & 10 & 5 & 2.5 & 10 & 2.5 & 0 \end{bmatrix}$$

forming $\mathcal{SP}_1$. Then we create $\mathcal{SP}_2$ from $\mathcal{SP}_1$. The H_0^3 which corresponds to $\mathcal{P}_0^3$ is $\{0, 10, 15, 20\}$. The intersection with hyperplanes $i_1 = 10, 15, 20$ produce polytones $\mathcal{P}_0^2$, $\mathcal{P}_1^2$ and $\mathcal{P}_2^2$,

$$\begin{bmatrix} -1.6 & -1.6 & -5 & -5 \\ 3.3 & 5 & 5 & 0 \end{bmatrix} \begin{bmatrix} -2.5 & -2.5 & -5 & -5 \\ 5 & 7.5 & 3.75 & 7.5 \end{bmatrix} \begin{bmatrix} -5 & -5 \\ 7.5 & 10 \end{bmatrix}$$

while the hyperplane $i_1 = 0$ yields a single point polyhedron, omitted. For $\mathcal{P}_1^3$, similarly, two 2-D subpolyhedra $\mathcal{P}_3^2$ and $\mathcal{P}_4^2$ are produced. All the five polyhedra together form the $\mathcal{SP}_2$.

We know $p_3 = 1$, and suppose that $p_2 = 5$ has been determined from $\mathcal{SP}_2$. Let us derive p_1 from $\mathcal{SP}_1$. Take $\mathcal{P}_0^3$ as an example. In H_0^3, at first, take 10 to form three hyperplanes 11, 10 and 9 intersecting the $\mathcal{P}_0^3$. The 2-D subpolyhedra produced by the hyperplanes 11 and 9 are

$$\begin{bmatrix} -1.8 & -1.8 & -4.5 & -5 & -5 \\ 3.6 & 5.5 & 1 & 0.75 & 5.5 \end{bmatrix} \begin{bmatrix} -1.5 & -1.5 & -4.5 & -4.5 \\ 3 & 4.5 & 4.5 & 0 \end{bmatrix}$$

Note that the polyhedra $\mathcal{P}_1^2$,$\mathcal{P}_0^2$ and $\mathcal{P}_{-1}^2$ are the 2-D subpolyhedra produced by $i_1 = 11, 10, 9$, respectively. We can find the $\mathbf{f}_1^2 = [-5, 1]^T$ and $\mathbf{l}_0^2 = [-2, 5]^T$, and $\mathbf{f}_0^2 = [-5, 0]^T$ and $\mathbf{l}_{-1}^2 = [-2, 4]^T$. Due to $\overline{\mathbf{p}}^2 = [5, 1]$, by eqn (5.10), p_1 is evaluated as 20. Repeat the procedure for the remaining elements of H_0^3, we find no new p_1 greater than 20. And repeat the procedure for $\mathcal{P}_1^3$, $p_1 = 20$ is still the greatest. So, at last, $\overline{\mathbf{p}}^3 = [20, 5, 1]$.

Repeating the procedure for p_0, the valid minimum projecting vector $\overline{\mathbf{p}}$ is derived to be [386,20,5,1]. The $\mathbf{l}^4$ and $\mathbf{f}^4$ are $[10, 10, 0, -5]^T$ and $[-5, 0, 20, 0]^T$. The actual executing time is 6166 time units. The conventional projecting vector by eqn (5.8) is [1386,66,11,1], and the corresponding executing time is 22056 units.

□

Again a great improvement (3.57 times) to the executing time is observed. Of course, everything depends on the shape of the polyhedron. However, for many irregular polyhedra, the improvement is quite significant. A testing program was used to check for all the examples used so far that every node in a given polyhedron is mapped by the $\overline{\mathbf{p}}$'s into their proper position. Algorithms C.3 and C.4 collected in Appendix C describe the process for intersecting and find first and last nodes of a polyhedron.

5.6 Optimisation

5.6.1 Method

Optimisation of the mapping can be made with both $\mathbf{S}$ and $\mathbf{\Pi}$, where the columns of $\mathbf{S}$ can be permuted. However, since there are K empty columns in $\mathbf{S}_{SUC}$ and K-1 empty columns

in $\mathbf{S}_{SBC}$, which make no difference, there are $\frac{N!}{K!}$ permuted versions of $\mathbf{S}_{SUC}$, and $\frac{N!}{(K-1)!}$ permuted versions of $\mathbf{S}_{SBC}$. It is obvious that if the columns of $\mathbf{\Pi}$ are permuted with $\mathbf{S}$, the $\mathbf{T}$ keeps the desired properties, such as unimodularity and satisfying eqn (4.19). The criterion for selecting $\mathbf{S}$ is that the $\mathbf{F}$ evaluated using $\mathbf{S}$ should yield the largest supernode, that is, $\prod_{j=0}^{N-1} f_j$ should be as small as possible. This criterion expresses the practical choice of granularity for the supernodes, and will be discussed later.

As for $\mathbf{\Pi}$, we can change the position of the only non-zero element "1" of each column of $\mathbf{L}_{K\times M}$ and $\mathbf{L}_{(K-1)\times M}$, resulting in K^M versions for SUC and $(K-1)^M$ versions for SBC. Furthermore, the rows of $\mathbf{\Pi}$ can be permuted to produce K! versions, which causes no violation of eqn (4.19) and has no effect on the unimodularity of $\mathbf{T}$. Therefore, the total optimisation space is $K!K^M$ and $K!(K-1)^M$ for SUC and SBC, respectively.

The optimisation procedure is briefly described as follows:

Step 1 Find all permuted versions of $\mathbf{S}$.

Step2 For each $\mathbf{S}$, evaluate the $\mathbf{F}$'s. Choose the $\mathbf{F}$ which results in the maximum volume of supernode. From the $\mathbf{F}$, determine $\mathbf{B}$ and then, compute the supernode polyhedron.

Step3 Produce all $\mathbf{\Pi}$'s containing all the possible $\mathbf{L}$'s. Then for each of them generate all permuted versions.

Step4 Produce the K-D time domain polyhedra by multiplying each of $\mathbf{\Pi}$'s by the supernode polyhedron. Then, for each of the K-D polyhedra, find the valid minimum $\overline{\mathbf{p}}$, as well as the corresponding executing time. Choose the $\mathbf{\Pi}$ and $\overline{\mathbf{p}}$ which results in the minimum executing time.

To compare with the method in [94], we fix $\mathbf{S}$. Because $\overline{\mathbf{t}} = \overline{\mathbf{p}}\mathbf{\Pi}$ we divide the optimisation procedure into two steps, instead of one step as in [94]. From the construction of $\mathbf{\Pi}$, we know that the $\mathbf{\Pi}$ is priori-optimum. Because the shape of the original polyhedron is changeable, we have to exploit all possible priori-optimum $\mathbf{\Pi}$ to achieve the posteriori-optimum, i.e., to find which of them match the shape of the polyhedron best, when combined with its corresponding $\overline{\mathbf{p}}$. Of course, actually, we want to obtain an optimum $\overline{\mathbf{t}}$.

5.6.2 An Example

Example 5.6.1 *Example of an Irregular Problem*

FOR $i_0 = 0$ TO 9
  FOR $i_1 = i_0$ TO $i_0 + 9$
    FOR $i_2 = i_1$ TO $i_1 + 9$
      $A(i_0, i_1, i_2) = A(i_0 - 1, i_1 - 1, i_2 - 1) + A(i_0, i_1 - 1, i_2 - 1) + A(i_0, i_1, i_2 - 1)$

Suppose that a linear array is used to implement the computation. Thus, $N = 3$ and $K = 2$. The computational graph can be found out:

$$\mathbf{V} = \begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 9 & 9 & 9 \\ 0 & 0 & 9 & 9 & 9 & 9 & 18 & 18 \\ 0 & 9 & 9 & 18 & 9 & 18 & 18 & 27 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

The computational polyhedron is a skewed long prism which is quite irregular, stretching from point (0, 0, 0) to point (9, 18, 27) and including 1000 nodes. Since this chapter focuses on lower-dimensional mapping only, we do not pay attention to the partitioning which is fully discussed in Chapter 4 and do not make it here, that is, let $\mathbf{B}$ be an identity matrix. However, since $\mathbf{D}$ is a subset of the canonical dependencies $\mathbf{D}^s$, the design of $\mathbf{T}$ proposed in Subsection 5.2.1 can be applied directly to this example.

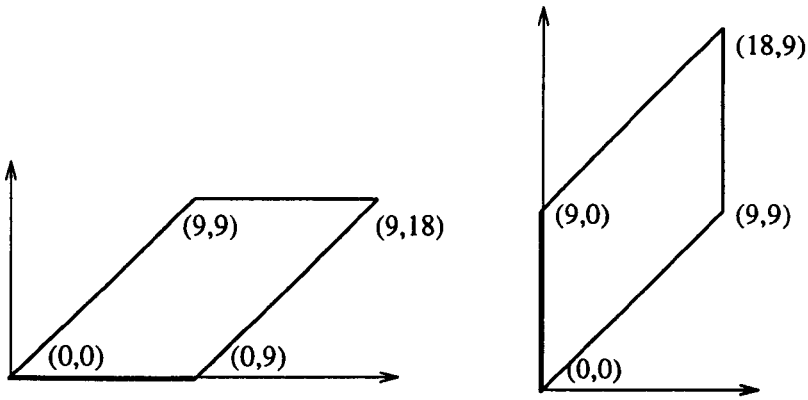
Letting $\mathbf{S} = [1, 0, 0]$, the computational polyhedron is mapped onto 10 processors. Then we can have the model of $\mathbf{\Pi}^2$, $\begin{bmatrix} 1 & & \\ 0 & \mathbf{l}_0 & \mathbf{l}_1 \end{bmatrix}$ where only one element of $\mathbf{l}_0$ is "1", and the same holds for $\mathbf{l}_1$. There are 2 candidates produced from the model, $\mathbf{\Pi}_0^2 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ and $\mathbf{\Pi}_1^2 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. We also row-permute the 2 candidates and obtain $\mathbf{\Pi}_2^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ and $\mathbf{\Pi}_3^2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$.

Let $\mathbf{U} = \begin{bmatrix} \mathbf{S} \\ \mathbf{\Pi}_2^2 \end{bmatrix}$. It can easily be checked that

$$\mathbf{V}^q = \mathbf{U}\mathbf{V}^s = [\mathbf{v}_0^q, \mathbf{v}_1^q, \mathbf{v}_2^q, \mathbf{v}_3^q, \mathbf{v}_4^q, \mathbf{v}_5^q, \mathbf{v}_6^q, \mathbf{v}_7^q] = \begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 9 & 9 & 9 \\ 0 & 0 & 9 & 9 & 9 & 9 & 18 & 18 \\ 0 & 9 & 9 & 18 & 18 & 27 & 27 & 36 \end{bmatrix}$$

There are 4 relevant edges which can be expressed in parametric form

$\text{Edge}\overline{\mathbf{v}_0^q\mathbf{v}_4^q}$: $[j_0, j_0, 2j_0]^T$, $\text{Edge}\overline{\mathbf{v}_1^q\mathbf{v}_5^q}$: $[j_0, j_0, 2j_0 + 9]^T$
$\text{Edge}\overline{\mathbf{v}_2^q\mathbf{v}_6^q}$: $[j_0, j_0 + 9, 2j_0 + 9]^T$, $\text{Edge}\overline{\mathbf{v}_3^q\mathbf{v}_7^q}$: $[j_0, j_0 + 9, 2j_0 + 18]^T$



(a) (b)

Figure 5.6: Two Maximum Local 2-D Time Polyhedra. (a) is $\mathcal{P}_1^2$ defined by $\mathbf{V}_1^2$ and (b) $\mathcal{P}_2^2$ defined by $\mathbf{V}_2^2$

Using $j_0 = p$ to intersect the 4 edges, we obtain $\mathbf{V}_p^j = \begin{bmatrix} p & p & p+9 & p+9 \\ 2p & 2p+9 & 2p+9 & 2p+18 \end{bmatrix}$ for any processor p. Adding "p"s to be as the first row of $\mathbf{V}_p^j$ and then pre-multiplying $\mathbf{U}^{-1}$ on it, we just obtain

$$\mathbf{V}_p^s = \begin{bmatrix} p & p & p & p \\ p & p & p+9 & p+9 \\ p & p+9 & p+9 & p+18 \end{bmatrix}, \text{ for instance } \mathbf{V}_0^s = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 9 \\ 0 & 9 & 9 & 18 \end{bmatrix}$$

Notice that in this particular example all $\mathbf{V}_p^s$ are the same subject to a translation $[p,p,p]^T$, thus we just take $\mathbf{V}_0^s$ to be as $\mathbf{V}^l$ defining the maximum local supernode polyhedron. Furthermore we should map it to 2-D time domain for all $\mathbf{\Pi}^2$'s. That is,

$$\mathbf{V}_1^2 = \mathbf{\Pi}_1^2\mathbf{V}^l = \mathbf{\Pi}_2^2\mathbf{V}^l = \begin{bmatrix} 0 & 0 & 9 & 9 \\ 0 & 9 & 9 & 18 \end{bmatrix} \text{ and } \mathbf{V}_2^2 = \mathbf{\Pi}_0^2\mathbf{V}^l = \mathbf{\Pi}_3^2\mathbf{V}^l = \begin{bmatrix} 0 & 9 & 9 & 18 \\ 0 & 0 & 9 & 9 \end{bmatrix}$$

all of which are parallelograms and are already the 2-D time convex hulls. They define two maximum local 2-D time polyhedra $\mathcal{P}_1^2$ and $\mathcal{P}_2^2$, respectively, see Fig 5.6. Both of them contain 100 points.

Note that for $\mathbf{V}$, $\mathbf{s}^l = [9,18,27]^T$ and $\mathbf{s}^f = [0,0,0]^T$. It is easy to compute the 4 timing vectors and their corresponding executing times: $\bar{t}_0 = \bar{p}\mathbf{\Pi}_0^k = [9,1,9]$ and 343 units, $\bar{t}_1 = \bar{p}\mathbf{\Pi}_1^k = [9,9,1]$ and 271 units, $\bar{t}_2 = \bar{p}\mathbf{\Pi}_2^k = [1,9,1]$ and 199 units and $\bar{t}_3 = \bar{p}\mathbf{\Pi}_3^k = [1,1,9]$ and 271 units. As a result, we take $\mathbf{\Pi}_2^2$ and $\bar{p} = [9,1]$ as the optimum $\mathbf{\Pi}$ and the optimum $\bar{p}$. As a result, we have

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \text{ and } \bar{p} = [9,1], \quad \text{or } \mathbf{T}_{2\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 9 & 1 \end{bmatrix}$$

The speed-up is 5.02 = 1000/199 and the overall efficiency is 50.2%. The major reason for the 50.2% overall efficiency is that there is a delay, 11 units, between adjacent processors along the array. The accumulated delays over 9 processors result in 99 time units. Since [34], [58], [59], [94] and [104] did not give a clue of how to deal with such an irregular problem, it is difficult to make an evaluation of these methods on the problem. If there are no extra measures added to deal with irregular problems, the straight approach we can consider is to embed the irregular polyhedron into a large cube of size $10 \times 19 \times 28$ and then apply these methods on the cube. However, it can be expected that the result cannot be any good, because the cube is only 18.7% filled. Suppose that these methods can achieve $xx\%$ efficiency for a full-filled cubic problem, the efficiency for this particular problem can be only $xx\% \times 18.7\%$, that will be very poor. □

5.7 Special Example: Partitioning and Mapping a Knapsack Problem onto a Linear Array

Generally speaking, the design process of partitioning and mapping are based on numerical computations, since the computational problem is described by much data. Unfortunately we have as yet no general technique for parameterizing the design process. For the following example which is defined by a small number of parameters, we can parameterise the design procedure.

5.7.1 Description of Computational Structure

The problem of minimizing or maximizing some linear function subject to a set of linear constraints is called a linear program, or an integer problem if their unknowns are restricted to be integers and non-negative. A particular integer program is the so-called Knapsack problem which is described by

$$\begin{array}{l} F_k(y) = max \sum_{j=1}^{k} v_j x_j \quad k = 0, \cdots, n \\ subject\ to \quad \sum_{j=1}^{k} w_j x_j \leq y \quad y = 0, \cdots, b \end{array} \tag{5.11}$$

where w_j and v_j are the weight and value of the j th item and x_j is the number of items of type j to be included in the Knapsack of capacity b. They all are integers. In general

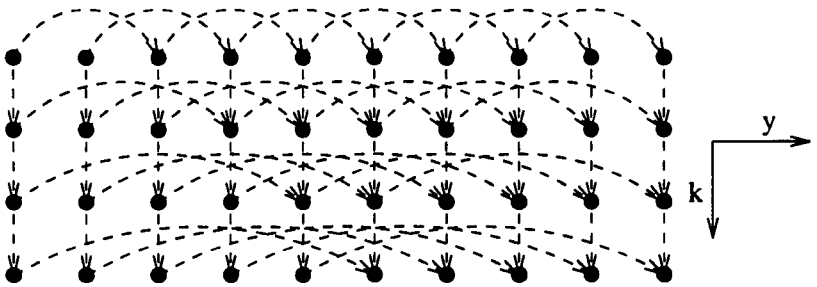


Figure 5.7: Knapsack Data dependency Graph $n = 4$ and $b = 10$

the Knapsack problem is NP hard. A dynamic programming method was proposed for solving Knapsack problem. The basic method consists of two passes [42].

Forward Pass: $F_k(y) = max\{F_{k-1}(y),\ F_k(y - w_k) + v_k\}$

Backward Pass: find the solution vector $x^* \in N^n$ such that $\sum_{i=1}^{n} v_i x_i^* = F_n(b)$

where $F_0(y) = 0,\ F_k(0) = 0,\ F_k(i) = -\infty$ for $k = 0, \cdots, n$ and $y = 0, \cdots, b$, and $i < 0$. This is a 2-D recursive procedure, but not an URE problem, see Figure 5.7 which shows the dependency graph of an example of $n = 4$ and $b = 10$. Note that the dependency vectors are changed while n and b are changed. [71] developed a run-time dependency algorithm which extends the computational polyhedron to 3-dimensions, but, with a set of URE dependencies, a significant improvement in computability. This algorithm has a form of set of recurrence equations

$$
\begin{aligned}
F(\mathbf{i}) &= \begin{cases} f(F(\mathbf{i}-\mathbf{e}_1), D_3(\mathbf{i}), V(\mathbf{i})) & \text{if } \mathbf{i} \in \mathcal{P} \\ 0 & \text{if } \mathbf{i} \ni \mathcal{P} \end{cases} \\
V(\mathbf{i}) &= \begin{cases} V(\mathbf{i}-\mathbf{r}) & \text{if } \mathbf{i}-\mathbf{r} \in \mathcal{P} \\ v(J(\mathbf{i})) & \text{if } \mathbf{i}-\mathbf{r} \ni \mathcal{P} \end{cases} \\
A(\mathbf{i}) &= \begin{cases} A(\mathbf{i}-\mathbf{r}) & \text{if } \mathbf{i}-\mathbf{r} \in \mathcal{P} \\ a(J(\mathbf{i})) & \text{if } \mathbf{i}-\mathbf{r} \ni \mathcal{P} \end{cases} \\
B(\mathbf{i}) &= \begin{cases} B(\mathbf{i}-\mathbf{r}) & \text{if } \mathbf{i}-\mathbf{r} \in \mathcal{P} \\ b(J(\mathbf{i})) & \text{if } \mathbf{i}-\mathbf{r} \ni \mathcal{P} \end{cases} \\
C(\mathbf{i}) &= \begin{cases} A(\mathbf{i}) & \text{if } \mathbf{i} \in \mathcal{P} \\ C(\mathbf{i}-\mathbf{d}_1) - 1 & \text{if } \mathbf{i} \in \overline{\mathcal{P}} - \mathcal{P} \end{cases} \\
S(\mathbf{i}) &= \begin{cases} B(\mathbf{i}) & \text{if } \mathbf{i} \in \mathcal{P} \\ S(\mathbf{i}-\mathbf{d}_1) & \text{if } \mathbf{i} \in \overline{\mathcal{P}} - \mathcal{P} \end{cases} \\
D_3(\mathbf{i}) &= \begin{cases} D_3(\mathbf{i}-\mathbf{d}_3) & \text{if } C(\mathbf{i}) \neq 0 \\ D_1(\mathbf{i}-\mathbf{d}_2) & \text{if } C(\mathbf{i}) = 0 \text{ and } S(\mathbf{i}) = 1 \\ D_1(\mathbf{i}) & \text{if } C(\mathbf{i}) = 0 \text{ and } S(\mathbf{i}) = 0 \end{cases} \\
C(\mathbf{i}) &= \begin{cases} F(\mathbf{i}) & \text{if } \mathbf{i} \in \mathcal{P} \\ D_1(\mathbf{i}-\mathbf{d}_1) & \text{if } \mathbf{i} \in \overline{\mathcal{P}} - \mathcal{P} \end{cases}
\end{aligned}
\tag{5.12}
$$

where $\mathbf{r} = [0,1,0]^T$, $\mathbf{e}_1 = [1,0,0]^T$, $\mathbf{d}_1 = [0,1,1]^T$, $\mathbf{d}_2 = [0,1,0]^T$ and $\mathbf{d}_3 = [0,1,-1]^T$; $\mathcal{P}$, the computation polyhedron, is a rectangular cube $n \times b \times c$, where $c = \lceil \frac{b}{2} \rceil$. We do not give a full explanation of the set of recurrence equations and the meaning of their variables, which is beyond the scope of the thesis, see [71] for detail. The matter we are concerned with is the computation graph produced by eqn (5.12). Because $\mathcal{P}$ is a rectangular cube, the computational polyhedron is also defined by the set of vertices, $\mathbf{V}^o$

$$\mathbf{V}^o = \begin{bmatrix} 0 & n & 0 & 0 & 0 & n & n & n \\ 0 & 0 & b & 0 & b & 0 & b & b \\ 0 & 0 & 0 & c & c & c & 0 & c \end{bmatrix} \tag{5.13}$$

By comparing the left-hand sides and righ-hand sides of eqn (5.12), the dependencies are

$$\mathbf{D} = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \tag{5.14}$$

This is an URE problem with conditional statements to control computation.

5.7.2 Supernode Polyhedron

Suppose that a linear array with m processors and unidirectional interconnections is given to implement the computation of the Knapsack problem.

We derive a positive expressing basis from $\mathbf{D}$

$$\mathbf{E} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \tag{5.15}$$

It is easy to verify that all the column vectors of $\mathbf{D}$ can be positively expressed by the $\mathbf{E}$.

The supernode parallelepiped $\mathbf{B}$ is obtained by scaling each column of $\mathbf{E}$. In this special case, since the processor array is 1-D, only the last column is required to be scaled. Thus

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & k \end{bmatrix} \qquad \mathbf{B}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{k} & \frac{1}{k} \end{bmatrix} \tag{5.16}$$

is generated to transform the original polyhedron to a supernode polyhedron.

We need to know the boundary of the resulted supernode polyhedron. It is easy to find the images $\mathbf{V}^s$ of the original vertices $\mathbf{V}^o$ in the supernode domain. That is

$$\mathbf{V}^s = \lfloor \mathbf{B}^{-1}\mathbf{V}^o \rfloor = \begin{bmatrix} 0 & n & 0 & 0 & 0 & n & n & n \\ 0 & 0 & b & 0 & b & 0 & b & b \\ 0 & 0 & \lfloor \frac{b}{k} \rfloor & \lfloor \frac{c}{k} \rfloor & \lfloor \frac{b+c}{k} \rfloor & \lfloor \frac{c}{k} \rfloor & \lfloor \frac{b}{k} \rfloor & \lfloor \frac{b+c}{k} \rfloor \end{bmatrix} \tag{5.17}$$

As seen later, since the space mapping matrix takes a form of [0, 0, 1] (liner array), the last row of $\mathbf{V}^s$ will determine the projected coverage of supernode domain along the 1-D array. Thus

$$\frac{b+c}{k} < m \quad \text{or} \quad k = \begin{cases} \frac{b+c}{m} - 1 & \text{if } \frac{b+c}{m} \text{ is integral} \\ \lceil \frac{b+c}{m} \rceil & \text{otherwise} \end{cases} \tag{5.18}$$

5.7.3 Transformation onto a Time-Processor Domain

A unimodular space-time projection matrix $\mathbf{T}_{3\times3} = \begin{bmatrix} \mathbf{S}_{1\times3} \\ \mathbf{\Pi}_{2\times3} \end{bmatrix}$ is used to project the supernode polyhedron into a 1-D Processor domain by $\mathbf{S}$ and a 2-D time domain by $\mathbf{\Pi}$. The 2-D time domain is, then, re-projected along a 1-D time domain by a valid minimum projection row vector $\overline{\mathbf{p}}$. Letting $\mathbf{S} = [0, 0, 1]$, the supernode polyhedron can be allocated onto to the m processor linear array as follows.

There are four candidates for $\mathbf{\Pi}$ which have the least and smallest non-zero entries (this is the criterion for priori-optimisation) and makes $\mathbf{T}^u$ unimodular. Among them, we find that $\mathbf{\Pi} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ is optimal. Therefore, we have

$$\mathbf{T} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \qquad \mathbf{T}^{-1} = \begin{bmatrix} -1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{5.19}$$

Suppose that the time-Processor domain is indexed by $\mathbf{j} = [j_0, j_1, j_2]^T$.

Then the supernode polyhedron is mapped into a 2-D domain

$$\mathbf{\Pi V}^s = \begin{bmatrix} 0 & 0 & b & 0 & b & 0 & b & b \\ 0 & n & \lfloor \frac{b}{k} \rfloor & \lfloor \frac{c}{k} \rfloor & \lfloor \frac{b+c}{k} \rfloor & \lfloor \frac{c}{k} \rfloor + n & \lfloor \frac{b}{k} \rfloor + n & \lfloor \frac{b+c}{k} \rfloor + n \end{bmatrix}$$

$$\Longrightarrow \mathbf{V}^t = \begin{bmatrix} 0 & 0 & b & b \\ 0 & \lfloor \frac{c}{k} \rfloor + n & \lfloor \frac{b}{k} \rfloor & \lfloor \frac{b+c}{k} \rfloor + n \end{bmatrix}$$

where $\mathbf{V}^t$ defines a trapezium in the 2-D time domain. The upper and bottom edges of the trapezium are $\lfloor \frac{b+c}{k} \rfloor - \lfloor \frac{b}{k} \rfloor + n$ and $\lfloor \frac{c}{k} \rfloor + n$, respectively. Because $\lfloor \frac{b+c}{k} \rfloor \geq \lfloor \frac{b}{k} \rfloor + \lfloor \frac{c}{k} \rfloor$, the upper edge is equal to or larger than the bottom.

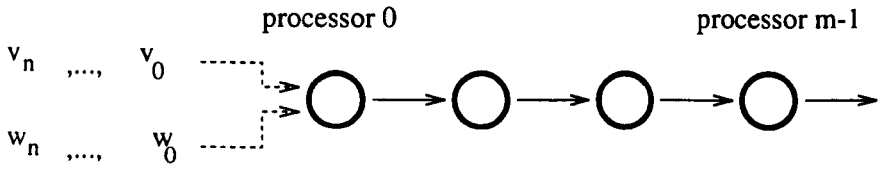


Figure 5.8: The m processor linear array implementing the Knapsack problem

Let

$$\overline{\mathbf{p}} = [p_0, p_1] \quad \text{where} \quad p_0 = \lfloor \frac{b+c}{k} \rfloor - \lfloor \frac{b}{k} \rfloor + n \text{ and } p_1 = 1 \tag{5.20}$$

It can be verified that the $\overline{\mathbf{p}}$ maps the trapezium $\mathbf{V}^t$ to a 1-D time domain without mapping two points on one location, and gives the minimum executing time $t^c = b(\lfloor \frac{b+c}{k} \rfloor - \lfloor \frac{b}{k} \rfloor + n) + \lfloor \frac{b+c}{k} \rfloor + n + 1$.

We can also give a space-time projection matrix $\mathbf{T}' = \begin{bmatrix} \mathbf{S} \\ \overline{\mathbf{t}} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & p_0 & 1 \end{bmatrix}$ which projects the supernode polyhedron directly into the 1-D processor domain and the 1-D time domain.

Finally, the input of the Knapsack problem is the series of v_i's and w_i's on eqn (5.11). According to the boundary conditions in [71], v_i and w_i should be sent to the node (i,0,0) of the original polyhedron, for $i = 0, \ldots, n$. When considering how to assign v_i's and w_i's to the linear array, we must find which processor the node (i, 0 ,0) is allocated to. It is easy to see that the node (i, 0 ,0) is allocated to processor j, where $j = \lfloor \mathbf{S}\mathbf{B}^{-1}[i,0,0]^T \rfloor$. Obviously $j = 0$, $\forall i$, $i = 0, \ldots, n$. Thus all v_i's and w_i's are enter the array at processor 0, see Figure 5.8. Similarly output is from the processor $m-1$.

5.8 Summary

A methodology of partitioning and mapping an arbitrary computation graph onto a given array with lower dimension is proposed. The original computational problem is transformed to a computational polyhedron with canonical dependencies. Two models, for SUC and SBC, of unimodular space-time mapping matrix $\mathbf{T}$ consisting of $\mathbf{S}$ and prioi-

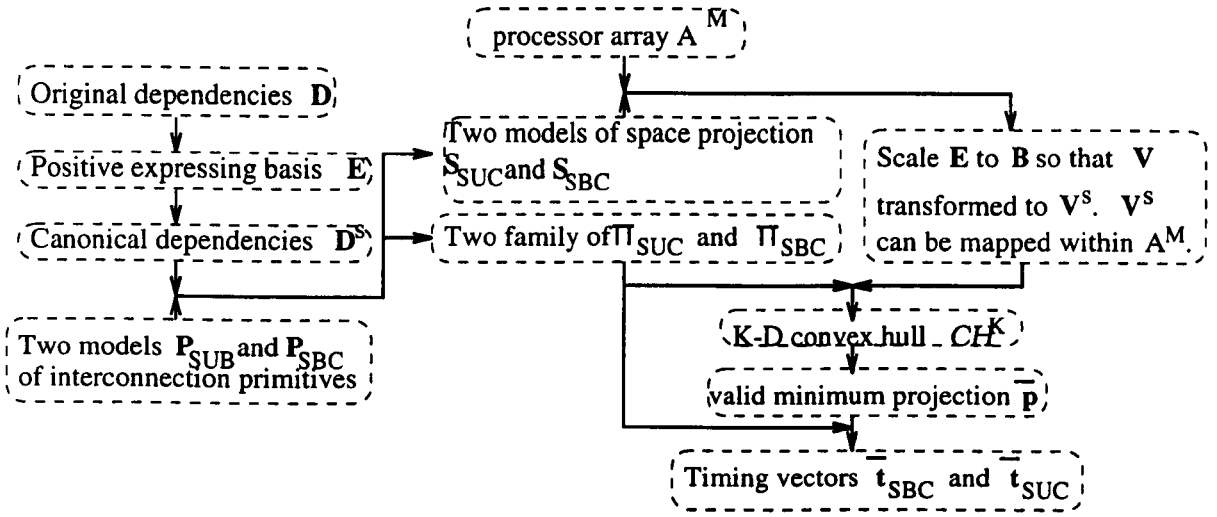


Figure 5.9: A Conceptual Chart of Lower dimensional Partitioning and Mapping Methods

optimum Π are proposed. The resulting N-D polyhedron is then projected into a K-D time polyhedron by Π and an M-D space polyhedron by S. The K-D time polyhedron is re-projected along a 1-D time domain by a valid minimum projection vector $\bar{p}$. For clarity the whole process is illustrated by the conceptual chart given in Figure 5.9

The main effort is to develop an optimal lower-dimensional mapping method which can cope with irregular polyhedra and gives improved efficiency while at the same time delivering fundamentally reduced computing complexity (polynomial with respect to the number of vertices of the K-D time polyhedron).

The methodology shows many advantages: implementing dependencies with given interconnection primitives, transferring data efficiently and mapping onto a given-shape processor array with, more importantly, lower dimension. The lower dimension mapping is highly efficient, especially for an irregular computational body. The method proposed here can handle irregular polyhedra effectively to improve the efficiency in time domain to an acceptable level.

Chapter 6

The Structure of Parallel programs

In this chapter the emphasis moves away from the mapping process proposed in chapters 3 and 4 and concentrates more on the practicalities of generating the associated parallel programs. A method of algorithm generation is developed for the situations where M-D processor arrays are given to implement a N-D nested loop structure. The resulting parallel algorithm is characterised by M DOALL loops in the space domain and a FOR loop in the time domain. Let K = N - M. There are two cases to consider: K> 1 and K = 1, which are treated separately.

6.1 Introduction

The task of this chapter is to transform sequential algorithms to parallel algorithms, which have the following forms:

$$\begin{array}{lcl} \textit{SequentialAlgorithm} & \Longrightarrow & \textit{parallelAlgorithm} \\ \textit{FOR} \cdots\cdots\cdots & & \textit{DOALL} \cdots\cdots\cdots \\ \quad \vdots & & \quad \vdots \\ \quad \textit{FOR} \cdots\cdots\cdots & \Longrightarrow & \quad \textit{DOALL} \cdots\cdots\cdots \\ \quad\quad \textit{statements} & & \quad\quad \textit{FOR} \cdots\cdots\cdots \\ & & \quad\quad\quad \textit{statements} \end{array}$$

using the partitioning and mapping techniques developed in the two previous chapters and, where "statements" may involve some sequential loops. Clearly the DOALL then corresponds to spatial position on a processor array and the sequential loop, to the algorithm for individual processors.

As shown above, partitioning is necessary to ensure that the computational polyhedron will be mapped within a given regular shape processor array, but it also presents many difficulties to automatic algorithm generation. A major problem of an algorithm generation method is to determine the structure of the parallel algorithm and the structure of the "statements" under the DOALL loops. In our case, the statements will be a set of sequential FOR-loops for a single supernode.

Obviously, we must know the bounds of both the DOALLs and FOR-loops. These bounds result from the supernode partitioning, so, at first, in this chapter we must exploit the features of supernodes and supernode space to determine the boundary of the supernode domain and the boundary of a particular supernode. The bounds of the DOALLs and FOR-loops are obtained by applying transformations on these boundaries.

The method of algorithm generation is heavily dependent on the techniques of partitioning and mapping. When $K > 1$, a lower-dimension mapping exists and, we have a problem of addressing N-D space from a (M+1)-D processor-time space. To solve this problem, an interesting mechanism is invented to make a K-D time domain behave as a 1-D time domain (this is termed the Lower-D Case). In addition, if the Locally-Sequential-Globally-Parallel (LSGP) method is involved in the partitioning, we have to cope with another difficult situation where a special time basis is introduced to access nodes. A fast algorithm is developed here to produce the index of the required nodes (and is termed the LSGP Case). For each of the two cases we wish to derive an algorithm generation technique. For brevity the case of an (N-1)-D processor array with SUC mesh, which involves neither lower-dimensional mapping nor LSGP partitioning, is taken as special case of the LSGP Case. Furthermore, we target a distributed-memory machine and so have to consider the problem of communications. Thus rules to create data flow packets among processors have to be established.

There have been many papers on the problem of partitioning and mapping from a theoretical viewpoint. However, so far, the actual generation of parallel algorithm remains quite an open topic for which little work has been done. It is "real obstacle to application", as was pointed out by an anonymous referee of our paper [18], because of its difficulty. All the methods presented in the chapter are original, and pave a practical way to applications.

In the following discussions, $\mathbf{B}$ and $\mathbf{T}$, for the different cases, $\overline{\mathbf{p}}$ and $k_0, \cdots, k_{N-2}$ are all given without derivation (refer to chapters 3 and 4 if necessary). Now in order to clarify concepts and methods we will make reference to the following example throughout the chapter.

Example 6.1.1 *A 3-D URE Problem*

Loops 6.1.1 *Original Sequential Nested Loops*

FOR $i_0 := 0$ TO 20
  FOR $i_1 := 0$ TO 20
    FOR $i_2 := 0$ TO 10
      $A(i_0, i_1, i_2) := A(i_0 - 1, i_1 + 1, i_2 + 3) + A(i_0, i_1 - 1, i_2 - 1) +$
      $+ A(i_0, i_1 - 1, i_2 + 1) + A(i_0, i_1 - 1, i_2)$

The computational polyhedron can be determined by the set of inequalities:

$$\begin{array}{rcl} i_0 & \geq & 0 \\ i_1 & \geq & 0 \\ i_2 & \geq & 0 \\ i_0 & \leq & 20 \\ i_1 & \leq & 20 \\ i_2 & \leq & 10 \end{array} \quad or \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{i} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 20 \\ 20 \\ 10 \end{bmatrix} \tag{6.1}$$

which contains $4851 = 21 \times 21 \times 11$ nodes, and whose dependency matrix is

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix} \tag{6.2}$$

In the example, the cubic computational polyhedron is selected only because it is easy to know how many nodes it contains. We point out that our methods can also cope with irregular polyhedra. For Example 6.1.1, we also suppose that a 1-D array with 4 processors and SUC mesh is given to implement the computation of the problem. This choice demonstrates the technique for the lower-dimensional case. 2-D arrays with 4×4 processors and SUC or SBC mesh are used, for the other cases.

The rest of the chapter is organised as follows: firstly we exploit the features of supernodes and supernode space which are the basis we work on. In Section 6.3, we determine the out-going data from a supernode. Section 6.4 transforms the algorithm onto a lower-dimensional array, while section 6.5 considers the case of LSGP. In Section 6.6, the derivation of the bounds of nested loops from a set of inequalities, which is essential for forming a loop program is discussed

6.2 On Supernodes

Once the partitioning matrix $\mathbf{B}$ is obtained, a transformation from the original computational domain to the quasi-supernode domain can be made by $\mathbf{i} = \mathbf{Bq}$, where $\mathbf{q}$, which may not be integral, is the index of the quasi-supernode space. The volume of the computational polyhedron is compressed by this transformation because $det(\mathbf{B}) > 1$.

Collecting all the projected nodes $\mathbf{q} = [q_0, \cdots, q_{N-1}]^T$ in a hypercube

$$\begin{aligned} s_j &\leq q_j < s_j + 1; \qquad s_j \in Z; j = 0, \cdots, N-1 \text{ or} \\ \mathbf{s} &\leq \mathbf{q} < \mathbf{s} + \mathbf{1} \end{aligned} \tag{6.3}$$

to the integral point $\mathbf{s} = [s_0, \cdots, s_{N-1}]^T$, $\mathbf{s}$ will be a supernode, including all the projected nodes in the hypercube, where $\mathbf{1} = [1, \cdots 1]^T$. All such supernodes form a supernode polyhedron. We say that $\mathbf{s}$ is the floor integration of the $\mathbf{q}$, noted as $\mathbf{s} = \lfloor \mathbf{q} \rfloor$, where $\lfloor matrix \rfloor$ means to take the floor function for all elements of the matrix.

For such a supernode domain, we are concerned with two problems. What is the boundary of the supernode polyhedron? and which nodes are involved in a single supernode?

6.2.1 Boundary of the Supernode Polyhedron

For the original set of nested loops, a system of inequalities expresses the original computational polyhedron

$$\mathbf{A}_{m,N}\mathbf{i} \leq \mathbf{c} \tag{6.4}$$

where m is the number of the inequalities, see eqn(6.1).

It is easy to obtain the quasi-supernode polyhedron by substituting $\mathbf{i} = \mathbf{Bq}$ into eqn (6.4)

$$\mathbf{A}^q_{m,N}\mathbf{q} \leq \mathbf{c} \tag{6.5}$$

where $\mathbf{A}^q_{m,N} = \mathbf{A}_{m,N}\mathbf{B}$.

For instance in Example (6.1.1), here we use the $\mathbf{B}$ which is derived by means of the method of Chapter 5,

$$\mathbf{E} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & -1 & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 6 & 0 & 0 \\ -6 & 1 & 0 \\ -18 & -1 & 2 \end{bmatrix} \quad \mathbf{B}^{-1} = \frac{1}{6}\begin{bmatrix} 1 & 0 & 0 \\ 6 & 6 & 0 \\ 12 & 3 & 3 \end{bmatrix} \tag{6.6}$$

Obviously $\mathbf{g} = [6, 1, 2]$ and $\det(\mathbf{B}) = 12$, which means that there will be 12 nodes enclosed in each partition. Eqn (6.1) is transformed to

$$\begin{array}{ll} -q_0 & \leq 0 \\ 6q_0 - q_1 & \leq 0 \\ 18q_0 + q_1 - 2q_2 & \leq 0 \\ 3q_0 & \leq 10 \\ -6q_0 + q_1 & \leq 20 \\ -18q_0 - q_1 + 2q_2 & \leq 10 \end{array} \quad \text{or} \quad \begin{bmatrix} -1 & 0 & 0 \\ 6 & -1 & 0 \\ 18 & 1 & -2 \\ 3 & 0 & 0 \\ -6 & 1 & 0 \\ -18 & -1 & 2 \end{bmatrix} \mathbf{q} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 10 \\ 20 \\ 10 \end{bmatrix} \tag{6.7}$$

However, to find a set of hyperplanes confining the supernode polyhedron is nontrivial, because applying $\lfloor\ \rfloor$ operations to all $\mathbf{q} = \mathbf{B}^{-1}\mathbf{i}$ so as to condense them to integral points produces some supernodes outside the quasi-supernode polyhedron confined by eqn (6.7). For instance, in Example (6.1.1), point $\mathbf{i} = [1, 20, 1]^T$, which is within the original polyhedron eqn(6.1), is transformed to $\mathbf{q} = [0.166, 21, 12.5]^T$ and, then, condensed to $\mathbf{s} = [0, 21, 12]^T$. However, substituting s into the left-hand side of the fifth inequality of eqn (6.7), we see that $[-6, 1, 0][0, 21, 12]^T = 21 > 20$, so s is outside the quasi-supernode polyhedron confined by eqn (6.7) (when substituting a point into the set of inequalities confining a polyhedron, it is said to be inside the polyhedron if all the inequalities hold, or said to be outside if any of the inequalities fail to hold). This phenomena is shown more clearly in a 2-D case.

In Fig 6.1, $\overline{ABCD}$ is the transformed quasi-supernode polyhedron. The "o"s indicate the images of the original vertices. The dots "•" are the supernodes which are outside the quasi-supernode polyhedron (the point j, marked by "×", will be explained later). We must construct an enlarged polyhedron in Z^N which includes all and only the supernodes. The polyhedron $\overline{abcda}$ is the required supernode polyhedron.

Enclosing all supernodes

The method of constructing the supernode polyhedron is to translate some of hyperplanes of the quasi-supernode polyhedron. Any hyperplane consisting of coefficients smaller than

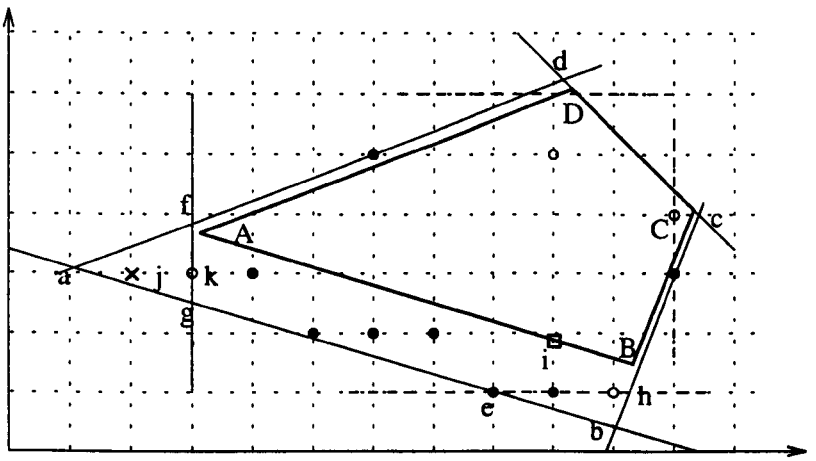


Figure 6.1: 2-D quasi-supernode polyhedron and the corresponding supernode polyhedron.

-1 is translated along its outward normal so that all supernodes are in the halfspace confined by the corresponding inequality. It can be understood that we must find the outermost supernode along the normal direction of such a hyperplane (termed furthest-supernode of the hyperplane), and that a furthest-supernode is made by floor integration of a point on the hyperplane. In Figure 6.1, the integral point e is the furthest-supernode associated with the plane (line) $\overline{AB}$. The line $\overline{ab}$ results from the translation of $\overline{AB}$ so that it passes through e.

We restrict ourselves to the situation where all nodes produced by a hyperplane of the original set of inequalities are integral [1], so the nodes are on the hyperplane. It is possible to produce all nodes on the hyperplane. A basis consisting of N-1 integral vectors can be found which generates all the integral nodes on the hyperplane.

Let a hyperplane $h : \bar{\mathbf{a}}\mathbf{i} + c = 0$. An algorithm is given to search for the furthest-supernode for a hyperplane and translate the hyperplane to pass through the furthest-supernode.

[1]This is quite a common case, especially in polyhedra generated from loop programs. In fact, this holds if $\mid a_l \mid = 1$, where a_l is the last non-zero coefficient of the hyperplane. For, instance, $2i_0+3i_1-i_2 = 3$

Algorithm 6.2.1 *The translation of a hyperplane h*

Define $Dist(\mathbf{q}, h)$: $\frac{|\bar{\mathbf{a}}\mathbf{q}+c|}{\sqrt{\bar{\mathbf{a}}\cdot\bar{\mathbf{a}}^T}}$

Generate a unimodular matrix $\mathbf{U}_a$ with $\bar{\mathbf{a}}$ as its first row
Derive $\mathbf{U}_a^{-1} = [\mathbf{z}_0, \mathbf{z}_1, \cdots, \mathbf{z}_{N-1}]$
FOR i = 1, N-1
    compute $\frac{\mathbf{y}_i}{y_i} = \mathbf{B}^{-1}\mathbf{z}_i$ and normalize such that $gcd(\mathbf{y}_i, y_i) = 1$
$\mathbf{q}_f := \{\mathbf{q}(\mathbf{x}) : Dist(\mathbf{q}(\mathbf{x}), h) = max_{x_1=1,\cdots,x_{N-1}=1}^{y_1-1,\cdots,y_{N-1}-1} Dist(\mathbf{q}(\mathbf{x}), h)\}$
where $\mathbf{x} = [x_1, \cdots, x_{N-1}]^T$ and $\mathbf{q}(\mathbf{x}) = \lfloor \mathbf{q}_s + \sum_{i=1}^{N-1} \frac{\mathbf{y}_i}{y_i} x_i \rfloor$
$c := -\overline{\mathbf{a}'}\mathbf{q}_f$
End of Algorithm

The algorithm is explained as follows. In fact, for a hyperplane with $\bar{\mathbf{a}} = [a_0, \cdots, a_j, 1, 0, \cdots]$, and because $gcd(\bar{\mathbf{a}}) = 1$, we can build an unimodular matrix $\mathbf{U}_a$ with $\bar{\mathbf{a}}$ as its first row [107]. Make $\mathbf{U}_a^{-1}$, the inverse matrix of $\mathbf{U}_a$. Obviously, the last N-1 column vectors, $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_{N-1}]$, of $\mathbf{U}_a^{-1}$ are perpendicular to $\bar{\mathbf{a}}$ and span the null-space of $\bar{\mathbf{a}}$. Any integral node on $\bar{\mathbf{a}}\mathbf{i} + c = 0$ can be expressed as a linear combination of the $\mathbf{z}_1, \cdots, \mathbf{z}_{N-1}$ plus a special point which may be a vertex on the hyperplane.

After projection to the quasi-supernode space, the hyperplane becomes $\bar{\mathbf{a}}\mathbf{B}\mathbf{q} + c = 0$, and any transformed nodes on the hyperplane can be expressed in a form of

$$\mathbf{q} = \mathbf{q}_s + (\mathbf{B}^{-1}\mathbf{Z})\mathbf{x} = \mathbf{q}_s + \sum_{i=1}^{N-1} \frac{\mathbf{y}_i}{y_i} x_i \tag{6.8}$$

where $\mathbf{q}_s$ is an initial point on the hyperplane, the elements of the vector $\mathbf{x}$ are the combinational coefficients; $\frac{\mathbf{y}_i}{y_i} = \mathbf{B}^{-1}\mathbf{z}_i$, $\mathbf{y}_i$ is an integral vector and y_i is an integer, and $gcd(\mathbf{y}_i, y_i) = 1$. Compute $\lfloor \mathbf{q} \rfloor$ for all $\mathbf{q}$ which are expressed by eqn (6.8) to find a point which is of the maximum perpendicular distance from the hyperplane, that is, the furthest-supernode. We need to check only the transformed nodes expressed by eqn (6.8) and covered by $x_i = 0, \cdots, y_i - 1$ for $i = 0, \cdots, N-1$, because the distances of $\lfloor \mathbf{q} \rfloor$ are a periodic function of the x_i's. The hyperplane for a supernode polyhedron is created by translating $\bar{\mathbf{a}}\mathbf{B}\mathbf{q} + c = 0$ to pass through the furthest-supernode.

By applying the procedure, the quasi-supernode polyhedron of eqn (6.5) is expanded to

$$\mathbf{A}_{m,N}^{q}\mathbf{q} \leq \mathbf{c}^{p} \tag{6.9}$$

Eqn (6.9) is different from eqn (6.5) only with respect to the constant vector.

In Example 6.1.1, consider the last inequality of eqn (6.7), $-18q_0 - q_1 + 2q_2 - 10 \leq 0$, which corresponds the original inequality $i_2 - 10 \leq 0$. It can be verified that

$$\mathbf{U}_a = \mathbf{U}_a^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{B}^{-1}\mathbf{Z} = \frac{1}{6}\begin{bmatrix} 0 & 1 \\ -6 & 6 \\ -3 & 12 \end{bmatrix}$$

So all supernodes can be expressed by

$$\lfloor \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix} + \frac{x_1}{2}\begin{bmatrix} 0 \\ -2 \\ -1 \end{bmatrix} + \frac{x_2}{6}\begin{bmatrix} 1 \\ 6 \\ 12 \end{bmatrix} \rfloor$$

where $[0,0,5]^T$ is a vertex on the plane. The furthest-supernode is $[0,5,15]^T$, Therefore the plane is translated to $-18q_0 - q_1 + 2q_2 - 25 = 0$.

Similarly, eqn (6.7) is modified to

$$\begin{array}{llll} -q_0 & \leq 0; & 3q_0 & \leq 10 \\ 6q_0 - q_1 & \leq 0; & -6q_0 + q_1 & \leq 25 \\ 18q_0 + q_1 - 2q_2 & \leq 1; & -18q_0 - q_1 + 2q_2 & \leq 25 \end{array} \tag{6.10}$$

to enclose all supernodes. Obviously the supernode [0,21,12]^T previously outside the polyhedron is now within the enlarged supernode polyhedron.

Enclosing only the supernodes

When enlarging the quasi-supernode polyhedron to enclose all the supernodes, it is quite possible to enclose some integral points which are not valid, i.e., false supernodes, into the new polyhedron, for instance, the point j marked "×" in Figure 6.1. Because point j is included within the enlarged supernode polyhedron, so it will be treated as a supernode, but it contains no nodes at all. These false supernodes should be removed because extra time is required to compute them in real algorithms.

To remove the false supernode, let us consider the 2-D problem shown in Figure 6.1. The following facts can be observed.

1. It can be seen that the false supernodes appear only around the vertices associated with any translated lines, for instance, point A in Figure 6.1. This makes it possible to dismiss the false supernodes by adding some extra cutting lines passing the vertices, such

as the line $\overline{fg}$, which dismisses the false supernode j.

2. It can be seen that for the vertex A which is not a integer, such a cutting line must pass the supernode k which is the floor integration of point A and is not allowed to pass through the original polyhedron. This is because the cutting line must not force any valid supernodes outside the domain.

3. Some of the extra cutting lines, i.e. the dashed lines, are not necessary, because there are no false supernodes to be dismissed around the relevant vertex. For example, $\overline{eh}$ passing the supernode e cuts nothing, because there are no supernodes (integral points) in the triangular area $\overline{ehbe}$.

As a result, Figure 6.1 requires only one cutting line $\overline{fg}$, and the supernode polyhedron is confined by 5 lines $\overline{fdcbgf}$.

These results can be extended to N-D cases. The false supernodes may appear only near the margins of facets of the polyhedron, that is, along and about the edges associated with any translated hyperplanes, rather than the vertices. So all such edges should be found initially. The edges can be found as follows. Suppose that there are n_v vertices lying on a hyperplane. It is easy to find the $\frac{n_v(n_v-1)}{2}$ connecting lines between any two vertices. For each of the connecting lines, generate an auxiliary hyperplane which contains the connecting line, and then check with the other n_v-2 vertices which are off the connecting line. If the n_v-2 vertices are only on one side of the auxiliary hyperplane, take the line as an edge; otherwise, ignore the line. The main computation lies in the checking which needs about $\frac{Nn_v(n_v-1)(n_v-2)}{2}$ arithmetic operations. If n_v = N, the checking is no longer needed, because all the connecting lines are edges.

Recall that all the edges are intersections of N-1 hyperplanes of eqn (6.9). It is easy to find which N-1 hyperplanes are associated with an edge. Suppose an edge exists between two vertices $\mathbf{w}^1 = [w_0^1, \cdots, w_{N-1}^1]^T$ and $\mathbf{w}^2 = [w_0^2, \cdots, w_{N-1}^2]^T$, corresponding to the vertices $\mathbf{v}^1$ and $\mathbf{v}^2$ in the original space. We give an algorithm to generate the necessary cutting hyperplanes to remove false supernodes along the edge.

Algorithm 6.2.2 *Generate Hyperplanes to Remove False Supernodes*

$\mathbf{d} := [d_0, \cdots, d_{N-1}]^T = \mathbf{w}^2 - \mathbf{w}^1$, $\mathbf{d}$ is the integral direction of the edge
$\mathbf{d}^v := \mathbf{v}^2 - \mathbf{v}^1$ and normalise such that $gcd(\mathbf{d}^v) = 1$
$\mathbf{d}^w := \mathbf{B}^{-1}\mathbf{d}^v$
$j = \{r : d_r^w = max_{i=0}^{N-1} d_i^w\}$
Generate two planes $p^1 : \ q_j = \lfloor w_j^1 \rfloor$ and $p^2 : \ q_j = \lceil w_j^2 \rceil$
Generate C_N^2 hyperplanes $h_i : d_{i_2} q_{i_1} - d_{i_1} q_{i_2} - (d_{i_2} w_{i_1} - d_{i_1} w_{i_2}) = 0$, where C_N^2 is a binomial coefficients, i_1 and $i_2 \in [0, N-1]$, and $i_1 \neq i_2$
FOR i = 1, C_N^2
  IF h_i not cutting through the quasi-supernode polyhedron
    $d_{i_1} := m d_{i_1}^w$ and $d_{i_2} := m d_{i_2}^w$ such that d_{i_1}, d_{i_2} and m are integers,
      and $gcd(d_{i_1}, d_{i_2}, m) = 1$, where i_1 and i_2 indicate the subspace h_i lie on
    $\mathbf{q}_f := \{\mathbf{q}(r) : Dist(\mathbf{q}(r), h_i) = max_{j=0}^{m-1} Dist(\mathbf{q}(j), h_i)\}$,
      where $\mathbf{q}(j) = \lfloor [w_{i_1}^1 + \frac{j d_{i_1}}{m}, w_{i_2}^1 + \frac{j d_{i_2}}{m}] \rfloor$
    Translate h_i to pass through $\mathbf{q}_f$
    Form a prism with the translated h_i and the N-1 hyperplanes of eqn (6.9)
    Close it by p^1 and p^2
    IF there are integral points inside the closed prism
      Append the translated h_i into the set of inequalities of eqn (6.9)
End of Algorithm

The basic idea of the algorithm is:

Step 1 Generate C_N^2 potential cutting hyperplanes containing the edge, each of which is perpendicular to a coordinate plane. If a cutting hyperplane penetrates the domain, remove it.

Step 2 Translate a cutting hyperplane so that it passes the related furthest-supernode of the edge.

Step 3 Form a prism with the translated hyperplane and the N-1 hyperplanes which are associated with the edge.

Step 4 Check if there are integral points, false supernodes, in the prism. If yes, add the hyperplane to eqn (6.9) to remove the false supernodes.

We still use Fig 6.1 to explain. Point A is equivalent to an edge; line $\overline{fg}$ corresponds to the cutting hyperplane which contains point A originally but, then, is translated to pass point k which is the furthest-supernode related to point A. Lines $\overline{ad}$ and $\overline{ab}$ are equivalent to the N-1 hyperplanes associated with the edge, they together with $\overline{fg}$ form a prism (a

triangle, here). Point j is an integral point in the prism. It is a false supernode which has to be removed by adding $\overline{fg}$.

The supernode polyhedron has the form of

$$\mathbf{A}^{s}_{m',N}\mathbf{s} \leq \mathbf{c}^{s} \tag{6.11}$$

where $m' \geq m$. Eqn (6.11) is the same as eqn (6.9) except that $m'-m$ cutting hyperplanes are appended.

For Example 6.1.1, we, omit the complex operations and find that 4 cutting planes are needed to remove all possible false supernodes. Finally, the supernode polyhedron, indexed with an integral tuple $\mathbf{s} = [s_0, \cdots, s_{N-1}]^T$, is confined by ten inequalities

$$\begin{bmatrix} -1 & 0 & 0 \\ 6 & -1 & 0 \\ 18 & 1 & -2 \\ 3 & 0 & 0 \\ -6 & 1 & 0 \\ -18 & -1 & 2 \\ 0 & -2 & 1 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \\ 0 & 1 & 0 \end{bmatrix} \mathbf{s} \leq \begin{bmatrix} 0 \\ 0 \\ 1 \\ 10 \\ 25 \\ 25 \\ 5 \\ 30 \\ 70 \\ 40 \end{bmatrix} \tag{6.12}$$

6.2.2 Vertices of the Enlarged Quasi-Supernode Polyhedron

The next problem we are concerned with is to find the vertices, $\mathbf{V}^e$, for the enlarged quasi-supernode polyhedron. In Subsection 6.5.1, the vertices will be projected by $\mathbf{T}$ to find a bounding box in the processor-time space. The straightforward way of solving the problem is to find the intersections of the new set of hyperplanes, i.e., to solve $C^N_{n_q}$ systems of N-D equations selected from the new n_q hyperplanes confining the enlarged quasi-supernode polyhedron, and then form the resulting intersecting points and remove those which are outside the polyhedron. This process is very time-consuming, so we use a faster approach given below.

Step 1 For a translated hyperplane, find all edges which pass through the vertices on its previous hyperplane and penetrate the hyperplane.

Step 2 Extend each of the edges to penetrate the translated hyperplane. The intersection points are new temporary vertices, take them in place of the corresponding old vertices. Then renew all the relevant edges according to the new vertices.

Step 3 Repeat Step 1 and 2 for all translated hyperplanes.

6.2.3 Boundary of a Single Supernode

A single supernode containing a collection of the original nodes is a basic unit of computation in a processor for a given time step. To do the computing, for any supernode $\mathbf{s}$, we must know which actual nodes are contained inside. There are two ways to solve this problem.

1. Substituting $\mathbf{q} = \mathbf{B}^{-1}\mathbf{i}$ into eqn (6.3) yields $b\mathbf{s} \leq \mathbf{B}'\mathbf{i} < b\mathbf{s} + b\mathbf{1}$, where $\mathbf{B}'$ is the adjoin matrix of $\mathbf{B}$, $b = det(\mathbf{B})$. However, we must remember that no nodes can be outside the original polyhedron. Therefore, the polyhedron for a supernode addressed by $\mathbf{s}$ is

$$\begin{bmatrix} -\mathbf{B}' \\ \mathbf{B}' \\ \mathbf{A}_{m,N} \end{bmatrix} \mathbf{i} < b \begin{bmatrix} -\mathbf{I} \\ \mathbf{I} \\ \mathbf{O} \end{bmatrix} \mathbf{s} + \begin{bmatrix} \mathbf{1} \\ b\mathbf{1} \\ \mathbf{c}+\mathbf{1} \end{bmatrix}$$

2. Because $\mathbf{E}$ is integral and unimodular, we build a transformation $\mathbf{q}' = \mathbf{E}^{-1}\mathbf{i}$ which maps $\mathbf{i}$ to another integral space $S^{q'}$ indexed by $\mathbf{q}'$. Obviously, $\mathbf{q}' = \mathbf{G}\mathbf{q}$. Substituting $\mathbf{q} = \mathbf{G}^{-1}\mathbf{q}'$ into eqn (6.3), we have $\mathbf{G}\mathbf{s} \leq \mathbf{q}' < \mathbf{G}\mathbf{s} + \mathbf{g}$. Therefore, in Q' space, the nodes in a supernode form a hypercube with size of $\mathbf{g}$. In this space, the dependencies become $\mathbf{D}^{q'} = \mathbf{E}^{-1}\mathbf{D}$. Furthermore, let $\mathbf{q}'' = \mathbf{q}' - \mathbf{G}\mathbf{s}$, where $\mathbf{0} \leq \mathbf{q}'' < \mathbf{g}$. The dependencies with $\mathbf{q}''$'s are the same as $\mathbf{D}^{q'}$, because $\mathbf{q}''$ is equivalent to $\mathbf{q}'$ except for a shift.

However, the original polyhedron boundary still has to be imposed to keep out invalid nodes. Substituting $\mathbf{i} = \mathbf{E}(\mathbf{q}'' + \mathbf{G}\mathbf{s})$ into eqn(6.4) produces $\mathbf{A}_{m,N}\mathbf{E}\mathbf{q}'' \leq \mathbf{c} - \mathbf{A}_{m,N}\mathbf{B}\mathbf{s}$. In addition, as will be seen later, in most cases, a supernode is addressed by a mapping $\mathbf{s} = \mathbf{T}^{-1}\mathbf{j}$, where $\mathbf{j}$ is the index in the processor-time domain. Thus, the nodes indexed by $\mathbf{q}''$ in the supernode addressed by $\mathbf{j}$ are confined by

$$\begin{bmatrix} -\mathbf{I} \\ \mathbf{I} \\ \mathbf{A}_{m,N}\mathbf{E} \end{bmatrix} \mathbf{q}'' < \begin{bmatrix} \mathbf{1} \\ \mathbf{g} \\ \mathbf{c} - \mathbf{A}_{m,N}\mathbf{B}\mathbf{T}^{-1}\mathbf{j} + \mathbf{1} \end{bmatrix} \tag{6.13}$$

where $\begin{bmatrix} -\mathbf{I} \\ \mathbf{I} \end{bmatrix} \mathbf{q}'' < \begin{bmatrix} \mathbf{1} \\ \mathbf{g} \end{bmatrix}$ is termed the hypercube-boundary-subset, and the remaining part is referred to as the non-hypercube-boundary-subset because they are associated with the boundary of the original polyhedron instead of with the hypercube-boundary of a supernode. We also say the supernode is confined in an area $[0, g_0) \times \cdots [0, g_{N-1})$, where $[x, y)$ indicates an integral interval from x to y-1. Since in the second method, manipulations in a supernode become much simpler, we use this approach for the following work.

6.3 Data Flows

An important problem is to find out what data will be transmitted outside the supernode and in which direction of the processor array they flow. It is not necessary to discuss the problem of the in-coming data, because the outgoing data of a supernode is just the in-coming data of another supernode at a correct location and correct time determined by the dependencies in the processor-time domain.

6.3.1 Outgoing Data of a Single Supernode

Let us consider one of the original dependency vectors $\mathbf{d} \in \mathbf{D}$. The data dependency projected in the quasi-supernode domain is produced, $\mathbf{d}^q = \mathbf{B}^{-1}\mathbf{d} = [d_0^q, \cdots, d_{N-1}^q]^T$.

For a transformed node $\mathbf{q}$ where $\mathbf{s} \leq \mathbf{q} < \mathbf{s} + \mathbf{1}$, if $q_i + d_i^q \geq s_i + 1$, the variables in $\mathbf{q}$ involved with $\mathbf{d}^q$ will flow over the i-th boundary of the supernode. Such data flows can be thought of as a data transfer "1" in the i-th dimension of the supernode domain. We denote the inequality $s_j + 1 - d_j^q \leq q_j$ as C_j, and $s_j + 1 - d_j^q > q_j$, as $\overline{C_j}$. C_j and $\overline{C_j}$ can be also expressed as $g_j - d_j^{q'} \leq q''$ and $g_j - d_j^{q'} > q''$, respectively. For simplicity, consider the 2-D case of Figure 6.2.

If there is a quasi-supernode dependency vector $\mathbf{d}^q = [d_0^q, d_1^q]^T$, a supernode is divided into four parts. The transformed nodes in A_2 are confined by $C_0\overline{C_1}$ and transfer data

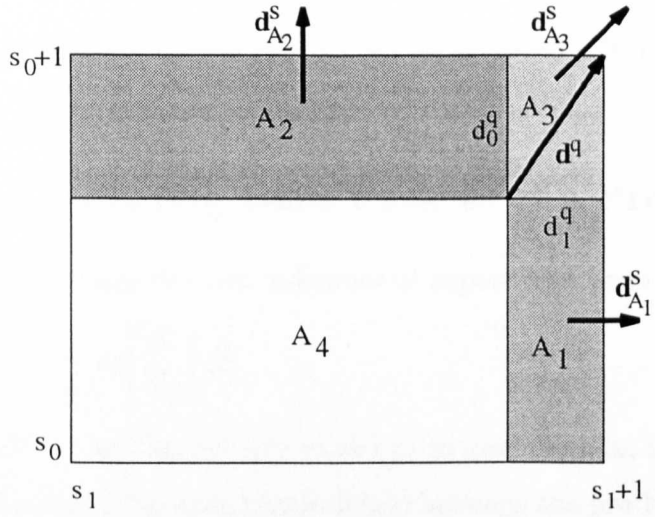


Figure 6.2: Dependencies in the quasi-supernode domain and in the supernode domain. .

out in the 0-th dimension, forming a supernode dependency vector $\mathbf{d}^s_{A_2} = [1,0]^T$; the transformed nodes in A_1 are confined by $\overline{C_0}C_1$ and transfer data out in the dimension 1, forming a supernode dependency vector $\mathbf{d}^s_{A_1} = [0,1]^T$; the transformed nodes in A_3 are confined by C_0C_1 transfer data out in both the 0 and 1 dimensions, forming a supernode dependency vector $\mathbf{d}^s_{A_3} = [1,1]^T$. Notice that there is no data flow out from A_4 where the nodes plus the $\mathbf{d}^q$ are still in the supernode.

Generally, a N-D supernode can be divided into 2^N areas confined by $\cap_{j=0}^{N-1} C_j^d$, where $C_j^d = C_j$ or $\overline{C_j}$. For a given area, if $C_j^d = C_j$, the area will form a data flow just to the next supernode along the j-th direction. Therefore, we have a set of $2^N - 1$ canonical dependencies in the supernode space, which form a N-bit binary table without $[0 \cdots 0]$, i.e., $\mathbf{D}^s$, as introduced in Subsection 4.2.2. For instance, from the 3-bit binary table, a supernode dependency vector $[1,0,1]^T$, noted as $\mathbf{d}^s_{101}$, can be expected to contribute from the area $C_0\overline{C_1}C_2$.

With the expression, such as $\cap_{j=0}^{N-1} C_j^d$, describing the confining area, we can draw the part of a supernode where variables flow outside in the direction $\mathbf{d}^s$. This is achieved by replacing the inequality $s_i \leq q_i$ in $\mathbf{s} \leq \mathbf{q} < \mathbf{s}+\mathbf{1}$ with $s_i + 1 - d_i^q \leq q_i$ if the term is C_i, or replacing the inequality $s_i < q_i$ with $s_i + 1 - d_i^q > q_i$ if the term is $\overline{C_i}$, for every term of the expression. It is now straightforward to obtain a hypercube similar to the hypercube-boundary-subset of eqn (6.13) by replacing the i-th entry of right-hand side constants with $-(g_i - d_i^{q'}) + 1$ or the (N+i)-th entry, with $(g_i - d_i^{q'})$, if the term is C_i or

$\overline{C_i}$, respectively. We call this a Data-Flow-Cube (DC). There are $2^N - 1$ DCs at most for a given N-D dependency vector.

6.3.2 Outgoing Data Packets of a Processor

It is easy to map the dependencies of supernodes onto the processor-time space. That is,

$$\begin{bmatrix} \mathbf{d}^p \\ d^t \end{bmatrix} = \begin{bmatrix} \mathbf{S} \\ \overline{\mathbf{t}} \end{bmatrix} \mathbf{d}^s \tag{6.14}$$

where $\mathbf{d}^p$ is the dependency in the processor domain, indicating the direction of the data flow; d^t is the delay step (time-delay) between the producing time and the consuming time of the data. Therefore, each DC has three attributes, $\mathbf{d}^s$, $\mathbf{d}^p$ and d^t, written as $DC_{\mathbf{d}^s}^{\mathbf{d}^p, d^t}$.

All the $\mathbf{d}^p$'s contribute to a dependency matrix $\mathbf{D}^p$ in the processor domain. For instance, for 3-D computational problem with 2-D processor array, for the two cases of $\mathbf{S} = \mathbf{S}_{SUC}$ and $\mathbf{S} = \mathbf{S}_{SBC}$, we have

$$\mathbf{D}_3^p = \begin{cases} \begin{bmatrix} \mathbf{d}_{01}^p & \mathbf{d}_{10}^p & \mathbf{d}_{11}^p \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} & \text{for SUC} \\ \begin{bmatrix} \mathbf{d}_{0,-1}^p & \mathbf{d}_{0,-1}^p & \mathbf{d}_{1,0}^p & \mathbf{d}_{-1,0}^p & \mathbf{d}_{-1,1}^p & \mathbf{d}_{1,-1}^p \\ 0 & 0 & 1 & -1 & -1 & 1 \\ 1 & -1 & 0 & 0 & 1 & -1 \end{bmatrix} & \text{for SBC} \end{cases}$$

The discussion above considers only one original dependency vector. The same operation must be repeated for all $\mathbf{d} \in \mathbf{D}$. Note that each dependency produces the same kinds of dependency vectors like $\mathbf{S}_{SUC}\mathbf{D}^s$ or $\mathbf{S}_{SBC}\mathbf{D}^s$ in the processor array domain, but defined in different DCs. Considering all the $\mathbf{d}^p$'s and $\mathbf{d}$'s, we can have $n_{D^p} \times n_d$ such values. That is $Q_{i,j} = \{\mathbf{q} :$ along $\mathbf{d}_i^p \in \mathbf{D}^p$ and associated with $\mathbf{d}_j \in \mathbf{D}\}$, where n_{D^p} and n_d are the numbers of vectors of $\mathbf{D}^p$ and $\mathbf{D}$, respectively. A $Q_{i,j}$ may consist of more than one DC. For efficient data transference, all DC's which flow in the same direction, i.e. the same $\mathbf{d}^p$, can be collected into a single data packet.

Note that in Example 6.1.1, $A(i_0, i_1, i_2) = A(i_0 - 1, i_1 + 1, i_2 + 3) + A(i_0, i_1 - 1, i_2 - 1) + A(i_0, i_1 - 1, i_2 + 1) + A(i_0, i_1 - 1, i_2)$, so that all the dependencies are associated with a single variable $A(i, j)$. This means that it is quite possible to transfer the same members of $A(i, j)$ along one direction several times. To avoid this case, if the original dependencies

$\mathbf{d}_{j_0}, \cdots, \mathbf{d}_{j_{r-1}}$ are associated with one variable, say A, we may transfer the members of A in the area $\bigcup_{k=0}^{r-1} Q_{i,j_k}$ along $\mathbf{d}_i^p$ only once. In addition, it is desired to modify the logical sum to the form of direct logical sum, that is, any two terms in the sum do not share any elements. Therefore, Q_{i,j_k} should be modified to $Q_{i,j_k} - Q_{i,j_0} -, \cdots, -Q_{i,j_{k-1}}$.

6.4 Algorithm Generation For A Lower Dimensional Processor Array

Given the above techniques we can proceed to the problem of algorithm generation for a given array. For the Lower-D Case, with a K-D time domain, we analyse the drawback of the K-D time parallel algorithm, and develop the 1-D time parallel algorithm which are similar to the K-D ones in form.

6.4.1 K-D Parallel Algorithms

As known, for the Lower-D Case, the unimodular space-time projection matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{S}_{M\times N} \\ \mathbf{\Pi}_{K\times N} \end{bmatrix}$$

is used to project the supernode polyhedron into a M-D processor domain by $\mathbf{S}$ and a K-D time domain by $\mathbf{\Pi}$. That is, a supernode $\mathbf{s}$ will be computed at the processor $\mathbf{Ss}$ at the moment $\mathbf{\Pi s}$. The $M+K$ $(M+K=N)$ dimensional processor-time space is indexed with an integral tuple $\mathbf{j} = [j_0, \cdots, j_{N-1}]^T$.

However, our interest in this chapter is not the mapping from the supernode domain to the time-processor domain, $\mathbf{s} \Longrightarrow \mathbf{j}$, but the reverse mapping, $\mathbf{s} \Longleftarrow \mathbf{j}$, since for an actual parallel computation, we must know which supernode should be executed at a particular processor at a particular moment. We use $\mathbf{s} = \mathbf{T}^{-1}\mathbf{j}$ to reference a supernode to be computed at a certain processor at a certain K-D moment. Substituting $\mathbf{s} = \mathbf{T}^{-1}\mathbf{j}$ into eqn (6.11), we can obtain a N-D processor-time polyhedron

$$\mathbf{A}^j_{m',N}\mathbf{j} \leq \mathbf{c}^j \tag{6.15}$$

where $\mathbf{A}^j_{m',N} = \mathbf{A}^s_{m',N}\mathbf{T}^{-1}$.

From eqn (6.15) and by means of the algorithm proposed in Section 6.6, we derive loops with the following form

Loops 6.4.1 *N-D Processor-time Loops*

DOALL $j_0 := l_0$ TO u_0

$\vdots$

DOALL $j_{M-1} := l_{M-1}(j_0, \cdots, j_{M-2})$ TO $u_{M-1}(j_0, \cdots, j_{M-2})$

FOR $j_M := l_M(j_0, \cdots, j_{M-1})$ TO $u_M(j_0, \cdots, j_{M-1})$

FOR $j_{M+1} := l_{M+1}(j_0, \cdots, j_M)$ TO $u_{M+1}(j_0, \cdots, j_M)$

$\vdots$

FOR $j_{N-1} := l_{N-1}(j_0, \cdots, j_{N-2})$ TO $u_{N-1}(j_0, \cdots, j_{N-2})$

computing a supernode

where $l_i(j_0, \cdots, j_{i-1})$ involves $\lceil\ \rceil$ and *max* operations, but, here, we pay attention only to the fact that it is a function of $j_0, \cdots, j_{i-1}$. Similar comments are true for $u_i(j_0, \cdots, j_{i-1})$ which involve $\lfloor\ \rfloor$ and *min* operations

6.4.2 K-D Time Domain to 1-D Time Domain

There are K FOR-loops for a processor in algorithm 6.4.1, corresponding to the local K-D time domain which is usually not regular and different for each processor. If we let these K FOR-loops run individually in each processor without some kind of control, the time delay of data transferences between processors will become variable. Therefore, we should reorganize the K FOR-loops into a single loop and let the single loop run synchronously in each processor.

If the K-D time domain is re-mapped with $\overline{\mathbf{p}}$ along a 1-D time domain indexed by t, for any dependency $\mathbf{d}^s \in \mathbf{D}^s$, the dependency in this domain is $\overline{\mathbf{p}}\mathbf{\Pi}\mathbf{d}^s$, which is a constant and can be implemented with a uniform pipeline mechanism. Therefore, we modify the K nested FOR loops with respect to $j_M, \cdots, j_{N-1}$ by introducing a single increasing argument t and decompose the t into $j_M, \cdots, j_{N-1}$ such that $t = \Sigma_{i=M}^{N-1} p_{i-N+K} j_i$. The decomposition must be unique, because it is used to reference supernodes. If the decomposition is not unique, more than one supernode may be referenced at a processor at a particular time t, but only one can be executed. Fortunately, it turns out that by employing the boundaries of $j_M, \cdots, j_{N-1}$, it is possible to find such a decomposition.

Let the first executing supernodes be $\mathbf{s}^f$. The value $\mathbf{s}^f$ is computed at moment $t^l = \overline{\mathbf{p}}\mathbf{\Pi}\mathbf{s}^f$. The Loops 6.4.1 are changed as follows

Loops 6.4.2 *Processor-time Loops with 1-D Time domain*

DOALL $j_0 := l_0$ TO u_0

$\vdots$

DOALL $j_{M-1} := l_{M-1}(j_0, \cdots, j_{M-2})$ TO $u_{M-1}(j_0, \cdots, j_{M-2})$

$t = t^l$

FOR $j_M := l_M(j_0, \cdots, j_{M-1})$ TO $u_M(j_0, \cdots, j_{M-1})$

$\vdots$

FOR $j_{N-2} := l_{N-2}(j_0, \cdots, j_{N-3})$ TO $u_{N-1}(j_0, \cdots, j_{N-3})$

FOR $j_{N-1} := t - \Sigma_{i=M}^{N-2} p_{i-M} j_i$ TO $u_{N-1}(j_0, \cdots, j_{N-2})$

IF $j_{N-1} \geq l_{N-1}(j_0, \cdots, j_{N-2})$

compute a supernode

$t := t + 1$

Let us prove the correctness. For the inner-most loop, because the lower bound of j_{N-1} is $t - \Sigma_{i=M}^{N-2} p_{i-M} j_i$, the relation

$$t = \Sigma_{i=M}^{N-1} p_{i-N+K} j_i$$

holds at the lower end of the inner-most loop, since $p_{K-1} = 1$ which is the coefficient of j_{N-1}. As j_{N-1} and t increase in step, the equation always holds. This is the decomposition we expect. However, only when $j_{N-1} \geq l_{N-1}(j_0, \cdots, j_{N-2})$, is such a decomposition valid and a supernode can be referenced; otherwise the decomposition is outside the valid processor-time domain defined in eqn (6.15), so no supernode will be computed.

It is interesting to compare the two loops. The loops in Loops 6.4.2 look quite similar to the those in Loops6.4.1. However in Loops 6.4.2, the real count variable is the t which increases one by one from t^l. Although the K nested FOR loops remain, their purpose is to decompose t to $j_M, \cdots, j_{N-1}$ subject to the boundary condition.

In Example 6.1.1, we find that

$$\mathbf{T} = \begin{bmatrix} \mathbf{S} \\ \mathbf{\Pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \qquad \mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{6.16}$$

The corresponding $\overline{\mathbf{p}} = [18, 1]$ gives the minimum executing time of 1034 time units. And so $\mathbf{T}' = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 18 \end{bmatrix}$.

Substituting $\mathbf{s} = \mathbf{T}^{-1}\mathbf{j}$ into eqn (6.12), the processor-time polyhedron is produced

$$\begin{array}{llll} -j_0 & \leq 0 & -17j_0 + 2j_1 - j_2 & \leq 25 \\ 7j_0 - j_2 & \leq 0 & 2j_0 + j_1 - 2j_2 & \leq 5 \\ 17j_0 - 2j_1 + j_2 & \leq 1 & -2j_0 - j_1 + 2j_2 & \leq 30 \\ 3j_0 & \leq 10 & j_0 + 2j_1 - j_2 & \leq 70 \\ -7j_0 + j_2 & \leq 25 & -j_0 + j_2 & \leq 40 \end{array} \tag{6.17}$$

Then, we can obtain the boundaries of the processor-time domain:

$$\begin{array}{l} l_0 = 0 \\ u_0 = 3 \\ l_1(j_0) = \lceil \frac{24j_0 - 1}{2} \rceil \\ u_1(j_0) = \lfloor min(55, 12j_0 + 25, \frac{6j_0 + 95}{2}, \frac{18j_0 + 65}{2}) \rfloor \\ l_2(j_0, j_1) = \lceil max(7j_0, -17j_0 + 2j_1 - 25, j_0 + 2j_1 - 70, \frac{2j_0 + j_1 - 5}{2}) \rceil \\ u_2(j_0, j_1) = \lfloor min(-17j_0 + 2j_1 + 1, 7j_0 + 25, j_0 + 40, \frac{2j_0 + j_1 + 30}{2}) \rfloor \end{array} \tag{6.18}$$

where the explicit derivation follows from Section 6.6.

Now consider what should be done in a processor at a given moment. According to eqn (6.13), in Example 6.1.1, the non-hypercube-boundary-subset of a supernode is

$$\begin{array}{lll} -q_0'' & < & -6j_0 + 1; \\ q_0'' - q_1'' & < & 7j_0 - j_2 + 1; \\ 3q_0'' + q_1'' - 1q_2'' & < & 17j_0 - 2j_1 + j_2 + 1; \\ q_0'' & < & 6j_0 + 21 \\ -q_0'' + 1q_1'' & < & -7j_0 + j_2 + 21; \\ -3q_0'' - q_1'' + q_2'' & < & -17j_0 + 2j_1 - j_2 + 11; \end{array}$$

and the nested loops for a supernode are

Loops 6.4.3 *Single Supernode Loops*

FOR $q_0'' = \max(6j_0, 0)$ TO $\min(6j_0 + 20, 5)$
  FOR $q_1'' = \max(q_0'' + 7j_0 - j_2, 0)$ TO $\min(q_0'' + 7j_0 - j_2 + 20, 0)$
    FOR $q_2'' = \max(3q_0'' + q_1'' + 17j_0 - 2j_1 + j_2, 0)$ TO $\min(3q_0'' + q_1'' + 17j_0 - 2j_1 + j_2 + 10, 1)$

Because $\mathbf{g} = [6, 1, 2]^T$, the supernode is also said to be confined in $[0, 6) \times [0, 1) \times [0, 2)$. As regards outgoing data, in Example 6.1.1, $\mathbf{SD}^q = [\frac{1}{6}, 0, 0, 0]$. Fortunately, since there is only one nonzero element in $\mathbf{SD}^q$ and we are given a linear array with a SUC mesh, this is the simplest case to find the outgoing data. That is, there is only one condition: C_2 : $s_2 + 1 - d_2^q \leq q_2$ and $d_2^q = \frac{1}{6}$. Note that

$$\mathbf{D}^{q'} = \mathbf{E}^{-1}\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \tag{6.19}$$

Replacing the constant of the first inequality of eqn (6.13) with -4 (i.e. $-4 = -(g_0 - d_0^{q'} = -(6-1)+1$), this condition results in $q_0'' \geq 5$. Therefore, there is only one data flow cube $DC_{100}^{1,1}$ which is confined by $[5,6) \times [0,1) \times [0,2)$ in a supernode.

In summary, the transformed parallel algorithm becomes

DOALL $j_0 := 0$ TO 3
  $t = 0$
  FOR $j_1 := \lceil \frac{24j_0-1}{2} \rceil$ TO $\lfloor min(55, 12j_0+25, \frac{6j_0+95}{2}, \frac{18j_0+65}{2}) \rfloor$
    FOR $j_2 := t - 18j_1$ TO $\lfloor min(-17j_0+2j_1+1, 7j_0+25, j_0+40, \frac{2j_0+j_1+30}{2}) \rfloor$
      $t := t+1$
      IF $j_2 \geq \lceil max(7j_0, -17j_0+2j_1-25, j_0+2j_1-70, \frac{2j_0+j_1-5}{2}) \rceil$
        FOR $q_0'' = \max(6j_0, 0)$ to $\min(6j_0+20, 5)$
          $q_1'' := 0$ if $q_0'' + 7j_0 - j_2 \in [0,20]$
          $q_{20}'' = 3q_0'' + q_1'' + 17j_0 - 2j_1 + j_2$
          FOR $q_2'' = \max(q_{20}'', 0)$ to $\min(q_{20}''+10, 1)$
            $A(q_0'', q_1'', q_2'') := A(q_0''-1, q_1'', q_2'') + A(q_0'', q_1''-1, q_2''-2) +$
                $+ A(q_0'', q_1''-1, q_2'') + A(q_0'', q_1'', q_2''-1)$
        Data transfer operations

The Data transfer operations will be discussed in Chapter 7 as a special case of (N-1)-D SBC case. Now the correctness of the methods of building the supernode polyhedron and a supernode can be verified. We know that there are $4851 = 21 \times 21 \times 11$ nodes in the original polyhedron of Example 6.1.1. Without expanding the quasi-supernode polyhedron, only 464 supernodes are enclosed in the polyhedron of eqn (6.7) and only 2119 nodes are accessible. With the expansion to eqn (6.10) (without dismissing false supernodes) 1404 supernodes are enclosed, 288 of which are false, although all the 4851 nodes are accessible. With expansion and dismissal, there are 1116 supernodes enclosed in the supernode polyhedron confined by eqn (6.12), all the 4851 nodes are accessible, none of the 1116 supernodes are empty, i.e., no false supernodes. This is exactly as expected.

6.5 Algorithm Generation For LSGP Case

Now let us consider the LSGP Case where an (N-1)-D processor array with an SBC mesh is given. In addition to the supernode partitioning, a LSGP partitioning (compression) is used to improve efficiency. For a uniform compression, we select the set of equal

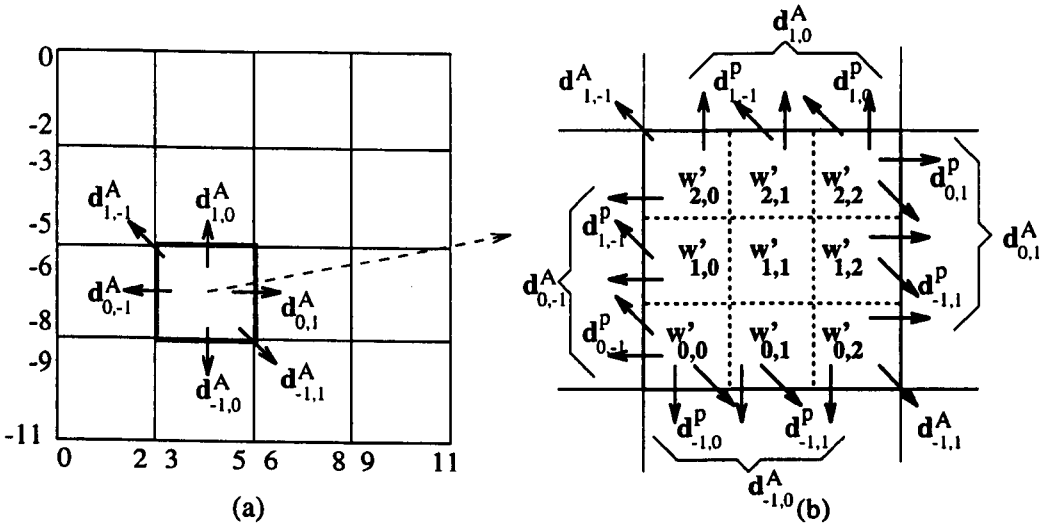


Figure 6.3: LSGP Partitioning layout and dependencies. (a) shows the layout of a 12×12 EVPA located in a 4×4 processor array. (b) the dependencies of a processor.

compression factors $k = k_0 = \cdots = k_{N-2}$ in advance. The size of the processor array is also enlarged to $(k \times l_0) \times, \cdots, \times (k \times l_{N-2})$, i.e., creating an EVPA. See Figure 6.3.

For generating the algorithm, we have to deal with three problems:

1. We need only to define a rectangular boundary for computing in the processor-time domain instead of an accurate boundary as in the Lower-D Case. This implies that all processors are assumed working in all the valid time. The actual work of each processor is defined by the supernodes assigned to it.

2. In LSGP partitioning, a new coordinate basis is introduced to access supernode space. We must find a quick method of accessing the new space from the processor-time space.

3. Since only the supernodes which are on the boundary of hypercubes of LSGP partitioning can contribute outgoing data, we have to establish rules to produce only the necessary data flows.

In the following, we focus on the case of (N-1)-D array with SBC mesh, while the case of (N-1)-D array with SUC mesh and the case of pure LSGP are presented in Appendix

6.5.5 without detail.

6.5.1 Rectangular Boundary in Virtual Processor-Time Domain

$\mathbf{W} = \mathbf{TV}^e$ indicates the vertices of the mapped virtual polyhedron in the virtual processor-time domain. The maximum and minimum values, noted as w_i^u and w_i^l, respectively, of the i-th row of $\mathbf{W}$ indicate the coverage of the mapped virtual polyhedron in the i-th dimension. The w_i^u's and w_i^l's may not be integral and may have to be reduced to integers within the given array. The virtual processor-time domain is an integral domain and no supernodes can be mapped outside the array. That is, w_i^u is approximated to $\lfloor w_i^u \rfloor$, and w_i^l, to $\lceil w_i^l \rceil$. For the first N-1 rows of $\mathbf{W}$, which are associated with space allocation, the relation $\lfloor w_i^u \rfloor - \lceil w_i^l \rceil + 1 \leq k_i l_i$ must hold (this has been guaranteed by the $\mathbf{B}$ and $\mathbf{T}$ derived by the method proposed in Chapter 4). For the last row of $\mathbf{W}$, the $\lfloor w_{N-1}^u \rfloor - \lceil w_{N-1}^l \rceil + 1$ indicates the computation time.

Let $t^l = \lceil w_{N-1}^l \rceil$ and $t^u = \lfloor w_{N-1}^u \rfloor$, and $\mathbf{w}^l = [\lceil w_0^l \rceil, \cdots, \lceil w_{N-2}^l \rceil]^T$ and $\mathbf{w}^u = [\lfloor w_0^u \rfloor, \cdots, \lfloor w_{N-2}^u \rfloor]^T$ which indicate the boundary of the mapped virtual array. The mapped virtual polyhedron is indexed by $\mathbf{w} = [w_0, \cdots, w_{N-2}]^T$.

Note that since the boundary of the supernode polyhedron is not used to confine the computation, some of processors may not compute useful work all the time. The actual computation is produced by including the boundary of the original computational polyhedron together with the bounds of supernodes.

6.5.2 Algorithm Generation Involving LSGP

In Loops 6.4.1, the coordinate $\mathbf{j}$ of the processor-time space references a supernode $\mathbf{s}$ directly, that is

$$\textit{Loops}6.4.1: \quad \mathbf{j} \overset{\mathbf{T}^{-1}}{\Longrightarrow} \mathbf{s}$$

However, when LSGP partitioning is involved, The $\mathbf{j}$ cannot directly reference $\mathbf{s}$, because $\mathbf{T}$ is not unimodular. We have to generate a special N-D time basis, indexed by $\mathbf{t}$,

such that j can easily reference s, and establish a mapping from t to s (see later)

$$Loops6.5.1: \quad \underset{j}{\cdot} \overset{solve\ equations\ of\ \mathbf{H}_k}{\Longrightarrow} \underset{t}{\mathbf{T}_b} \Longrightarrow s$$

where $\mathbf{H}_k$ is a lower-triangular matrix, $\mathbf{T}_b$ is the N-D time basis, both of which are to be discussed below. For the convenience of presentation, we generate the N-D time basis $\mathbf{T}_b$ first, then find a quick transformation from j to t.

A N-D time basis

In the following discussions, we generate a N-D time basis according to Darte's definition and, then, modify the N-D time basis to one which can be easily accessed from j.

The last row of $\mathbf{T}$ is the timing vector $\bar{t}$. A unimodular matrix $\mathbf{T}_b'^{-1}$ can be constructed with $\bar{t}$ as its last row. Then $\mathbf{T}_b' = [\mathbf{Q}', t_{N-1}]$, where $\mathbf{Q}'$ is the $N \times (N-1)$ matrix, has the significant feature that $\bar{t}\mathbf{T}_b' = [0, \cdots, 0, 1]$. Therefore, $\mathbf{T}_b'$ is a N-D time basis, and $\mathbf{Q}'$ spans a sub-space such that all the nodes are executed at one instance. $\mathbf{Q}'$ is not unique, but t_{N-1} is, and because $det(\mathbf{T}_b') = 1$, $\mathbf{T}_b'$ can be a basis to access all points.

For any supernode which is expressed as $s = \mathbf{T}_b' t'$, where $t' = [t_0', \cdots, t_{N-1}']^T$ is the index in the space spanned by $\mathbf{T}_b'$, it will be executed at the moment t_0' at cell

$$\mathbf{S}\mathbf{T}_b' t' = \mathbf{A}[t_0', \cdots, t_{N-2}']^T + \mathbf{r}' t_{N-1}'$$

in the EVPA, where $\mathbf{A}_{(N-1)\times(N-1)} = \mathbf{S}\mathbf{Q}'$ is the activity matrix, and $\mathbf{r}' = \mathbf{S}t_{N-1}$. We can produce (N-1)! forms of $\mathbf{A}$ by permuting the rows of $\mathbf{S}$. Let $\mathbf{P}_p^{(p_0 \cdots p_i \cdots p_{N-2})}$ be a permuting matrix.

For the (N-1)! forms of $\mathbf{A}$, we compute their HNF's, (Hermite Normal Form), i.e., HNF $= \mathbf{A}^{(p_0 \cdots p_i \cdots p_{N-2})}\mathbf{U}^u = \mathbf{P}_p^{(p_0 \cdots p_i \cdots p_{N-2})}\mathbf{S}\mathbf{Q}'\mathbf{U}^u$, where $\mathbf{U}^u$ is an unimodular matrix. From the method in Chapter 4, $\mathbf{T}$ guarantees that in the (N-1)! HNF's a HNF exists such that all the diagonal elements of the HNF are k, noted as $\mathbf{H}_k$. Suppose $\mathbf{P}_k$ is the permuting matrix and $\mathbf{U}_k$, the unimodular matrix to produce the $\mathbf{H}_k$.

Now we can modify the space-mapping matrix and time basis. Let $\mathbf{S}_P = \mathbf{P}_k\mathbf{S}$, $\mathbf{Q} = \mathbf{Q}'\mathbf{U}_k^u$ and $\mathbf{T}_b = [\mathbf{Q}, t_{N-1}]$. The matrix $\mathbf{T}_b$ is the new N-D time basis which accesses

all supernodes [2] and keeps the feature we are concerned with, $\bar{\mathbf{t}}\mathbf{T}_b = [0, \cdots, 0, 1]$. The corresponding activity matrix is a lower-triangular matrix with equal diagonal elements, i.e., $\mathbf{S}_P\mathbf{Q} = \mathbf{H}_k$.

Note that since $\mathbf{P}_k$ permutes the rows of $\mathbf{S}$, the boundary of the mapped virtual array should be permuted correspondingly, that is, $\mathbf{w}^u := \mathbf{P}_k\mathbf{w}^u$ and $\mathbf{w}^l := \mathbf{P}_k\mathbf{w}^l$.

Quick accessing from processor-time space to supernodes

Now let us access a supernode in the N-D time space from a given processor. At first, the N-D time space is mapped onto the EVPA. The EVPA, then, is divided and assigned to real processors. Finally, in reverse, we build a transformation from the real processors to supernodes with respect to time.

A supernode $\mathbf{s}$ is expressed by $\mathbf{s} = \mathbf{T}_b\mathbf{t}$ in the N-D time space spanned by $\mathbf{T}_b$, where $\mathbf{t} = [t_0, \cdots, t_{N-1}]^T$ is its index in the N-D time space. The $\mathbf{s}$ is executed at moment t_{N-1} and at a location

$$\mathbf{w} = \mathbf{S}_P\mathbf{s} = \mathbf{S}_P\mathbf{T}_b\mathbf{t} = \mathbf{H}_k[t_0, \cdots, t_{N-2}]^T + \mathbf{r}t_{N-1} \tag{6.20}$$

in the EVPA permuted by $\mathbf{P}_k$, where $\mathbf{r} = \mathbf{P}_k\mathbf{r}'$.

Now, the mapped virtual array can be allocated to the actual processor array. We let the processor $\mathbf{j}'$, indexed by $j_0, \ldots, j_{N-2}$, contain all the $\mathbf{w}$ such that

$$\begin{aligned} \mathbf{w}^l + k\mathbf{j}' &\leq \mathbf{w} < \mathbf{w}^l + k\mathbf{j}' + k\mathbf{1}, \text{ or} \\ w_i^l + j_i k &\leq w_i < w_i^l + (j_i + 1)k, \; i = 0, \cdots, N-2 \end{aligned} \tag{6.21}$$

Thus, the supernode, indexed by $\mathbf{t}$, to be executed at the processor $\mathbf{j}'$ at the moment t_{N-1} is the $\mathbf{t}$ which is an integral solution of eqn (6.20) subject to eqn (6.21) [3].

[2] However to make sure that $\mathbf{T}_b$ can access all supernodes, that is, $\mathbf{T}_b$ is bijective, we must prove that $\mathbf{T}_b$ is unimodular. In fact, note that generally, for any two matrices $\mathbf{M}_1$ and $\mathbf{M}_2$, deleting the i-th row of $\mathbf{M}_1$ then post-multiplied by $\mathbf{M}_2$ is equivalent to deleting the i-th row of $\mathbf{M}_1\mathbf{M}_2$. Letting $\mathbf{M}_i'$ indicate the matrix of $\mathbf{Q}'$ with the i-th row deleted and $\mathbf{M}_i$ indicate $\mathbf{Q}$ with the i-th row deleted, then $\mathbf{M}_i = \mathbf{M}_i'\mathbf{U}_k^u$ and $det(\mathbf{M}_i) = det(\mathbf{M}_i')$. Therefore $det(\mathbf{T}_b) = \sum_i^{N-1}(-1)^i det(\mathbf{M}_i)t_{N-1,i} = \sum_i^{N-1}(-1)^i det(\mathbf{M}_i')t_{0,i} = 1$ where $t_{N-1,i} \in \mathbf{t}_{N-1}$.

[3] It can be understood that for any given moment t_{N-1}, the solution is unique, In fact, assume that two different supernodes $\mathbf{t}' = [t'_0, \cdots, t_{N-1}]^T$ and $\mathbf{t}'' = [t''_0, \cdots, t_{N-1}]^T$ are executed at the same time t_{N-1} at the same processor. Without losing generality, suppose $t'_0 - t''_0 = \delta \neq 0$. The $\mathbf{t}'$ is allocated at $\mathbf{w}'$ in the mapped virtual array, while $\mathbf{t}''$, at $\mathbf{w}''$. Because $w_0' - w_0'' = k\delta$, $\mathbf{w}'$ and $\mathbf{w}''$ cannot be in the same processor, contradicting our assumption.

We give a quick algorithm for the integral solution

Algorithm 6.5.1 *Derive* t

$\mathbf{w}' = [w'_0, \cdots, w'_{N-2}]^T = \mathbf{w}^l + k\mathbf{j}' - \mathbf{r}t_{N-1}$
FOR i=0 TO N-2
  $w'_i := w'_i - \sum_{j=0}^{i-1} h_{i,j}t_j$, $t_i := \lceil \frac{w'_i}{k} \rceil$
End of Algorithm

where $h_{i,j} \in \mathbf{H}_k$. It is easy to check the correctness of the algorithm. In fact, for any $i \in [0, N-2]$, eqn (6.21) is equivalent to

$$w'_i \leq kt_i < w'_i + k \tag{6.22}$$

where $w'_i = w^l_i + kj_i - r_i t_{N-1} - \sum_{j=0}^{i-1} h_{i,j}t_j$. Eqn (6.22) holds if $t_i := \lceil \frac{w'_i}{k} \rceil$, due to the fact that $x \leq \lceil \frac{x}{k} \rceil k < x + k$. The parallel algorithm for LSGP Case has a form of

Loops 6.5.1 *Parallel Algorithm for (N-1)-D Array of SBC Mesh*

$\mathbf{w}^l := \mathbf{w}^l - t^l\mathbf{r}$
DOALL $j_0 := 0$ TO $l_0 - 1$
  ⋮
    DOALL $j_{N-2} := 0$ TO $l_{N-2} - 1$
      $\mathbf{w}'' := \mathbf{w}^l + k\mathbf{j}'$
      FOR $j_{N-1} := t^l$ TO t^u
        FOR i=0, N-2
          $w'''_i := w''_i - \sum_{j=0}^{i-1} h_{i,j}t_j$, $t_i := \lceil \frac{w'''_i}{k} \rceil$
        $t_{N-1} = j_{N-1}$, $\mathbf{w}' := k\mathbf{t} - \mathbf{w}'''$, $\mathbf{w}'' := \mathbf{w}'' - \mathbf{r}$
        compute supernode $\mathbf{s} = \mathbf{T}_b\mathbf{t}$

where $\mathbf{w}' = \mathbf{w} - (\mathbf{w}^l + k\mathbf{j}')$, which is useful when collecting outgoing data. The boundary of a supernode accessed by t is eqn (6.13), but note that we replace $\mathbf{T}^{-1}$ with $\mathbf{T}_b$.

The procedure of deriving t from w is a forward recursion, which is possible because $\mathbf{H}_k$ is lower-triangular. This is why we preferred to modify the space-mapping matrix and time basis to produce a lower-triangular activity matrix.

6.5.3 Outgoing Data after LSGP

Due to the LSGP partitioning, a more complex situation is introduced into the dependency relationship, see Figure 6.3.(b). For a 2-D processor array, as defined above, w′ indicates the virtual processors in a single actual processor. The dependencies $\mathbf{d}^p$'s of Section 6.3

become the dependencies between the virtual processors. The dependencies $\mathbf{d}^A$'s between processors are composed from the $\mathbf{d}^p$'s. Only the outward-oriented $\mathbf{d}^p$'s of the virtual processors around the boundary of the processor have contributions to $\mathbf{d}^A$. For instance, $\mathbf{d}^A_{0,1} = \mathbf{d}^p_{0,1}.(\mathbf{w}'_{0,2} + \mathbf{w}'_{1,2} + \mathbf{w}'_{2,2}) + \mathbf{d}^p_{-1,1}.(\mathbf{w}'_{1,2} + \mathbf{w}'_{2,2})$ and $\mathbf{d}^A_{-1,1} = \mathbf{d}^p_{-1,1}.\mathbf{w}'_{0,2}$, where $\mathbf{d}^p_i.\mathbf{w}'$ means the $\mathbf{d}^p_i$ of the $\mathbf{w}'$. However, it can be understood that $\mathbf{D}^A = \mathbf{D}^p$, where $\mathbf{D}^A$ is the dependency matrix in the processor domain, composed from all the $\mathbf{d}^A$'s.

For general cases, we may characterise a virtual processor with a location vector $\mathbf{l}^w(\mathbf{w}') = [l^w_0, \cdots, l^w_i, \cdots, l^w_{N-2}]$, where

$$l^w_i = \begin{cases} -1 & \text{if } \mathbf{w}' \text{ is on the i-th lower boundary of the processor, i.e., } w'_i = 0 \\ 1 & \text{if } \mathbf{w}' \text{ is on the i-th upper boundary, } w'_i = k-1, \\ 0 & \text{otherwise} \end{cases}$$

Notice that, in fact, the location vector shows the potential directions of outgoing data from a particular position w'_i. Furthermore, we define a special "match" operator $\otimes$ such that

$$a_i \otimes b_i = \begin{cases} a_i & \text{if } a_i = b_i \\ 0 & \text{otherwise} \end{cases}$$

and $\mathbf{a} \otimes \mathbf{b} = [a_0 \otimes b_0, a_1 \otimes b_1, \cdots]^T$, where $a_i = -1, 0, 1$ and $b_i = -1, 0, 1$. Then, for each $\mathbf{w}'$ in a processor, if $\mathbf{d}^p \otimes \mathbf{l}^w(\mathbf{w}') \in \mathbf{D}^A$ for any $\mathbf{d}^p \in \mathbf{D}^p$, we will attribute the $\mathbf{d}^p$ of the $\mathbf{w}'$ to

$$\mathbf{d}^A = \mathbf{d}^p \otimes \mathbf{l}^w(\mathbf{w}') \tag{6.23}$$

where $\mathbf{D}^p$'s is that for SBC in Section 6.3. Eqn (6.23) means that at a position $\mathbf{l}^w$, for each dimension, $\mathbf{d}^p$ becomes $\mathbf{d}^A$ only if $\mathbf{d}^p$ matches the location vector $\mathbf{l}^w$ of the $\mathbf{w}'$.

For instance, for Figure 6.3.(b), we have $\mathbf{l}^w(\mathbf{w}'_{1,2}) = [0,1]^T$, $\mathbf{l}^w(\mathbf{w}'_{2,2}) = [1,1]^T$, so $\mathbf{d}^p_{0,1} \otimes \mathbf{l}^w(\mathbf{w}'_{1,2}) = \mathbf{d}^p_{-1,1} \otimes \mathbf{l}^w(\mathbf{w}'_{1,2}) = \mathbf{d}^p_{0,1} \otimes \mathbf{l}^w(\mathbf{w}'_{1,2}) = \mathbf{d}^p_{-1,1} \otimes \mathbf{l}^w(\mathbf{w}'_{2,2}) = [0,1]^T = \mathbf{d}^A_{0,1} \in \mathbf{D}^A$; $\mathbf{l}^w(\mathbf{w}'_{0,2}) = [-1,1]^T$, so $\mathbf{d}^p_{0,1} \otimes \mathbf{l}^w(\mathbf{w}'_{0,2}) = \mathbf{d}^A_{0,1}$, too, and $\mathbf{d}^p_{-1,1} \otimes \mathbf{l}^w(\mathbf{w}'_{0,2}) = [-1,1]^T = \mathbf{d}^A_{-1,1} \in \mathbf{D}^A$.

6.5.4 Example

We try to use Example 6.1.1 to show the procedure above. But since the procedure is complex, we cannot present all the details. Therefore, the example give only a clue of how our method works.

For Example 6.1.1, the EVPA has a size of 12×12, $k = 3$. By a series of computations, we have

$$\mathbf{B} = \begin{bmatrix} 7 & 0 & 0 \\ -7 & 3 & 0 \\ -21 & -3 & 8 \end{bmatrix} \quad \mathbf{T} = \begin{bmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \\ 5 & 1 & 3 \end{bmatrix}$$

$det(\mathbf{B}) = 168$ and $det(\mathbf{T}) = 9$. Here, $\mathbf{S} = \mathbf{S}_{SBC}\mathbf{P}^{(201)}$. $\mathbf{g} = [7,3,8]^T$. Note that $\mathbf{E}$ is the same as that of eqn(6.6). Without showing the vertices $\mathbf{W}$, we just note that $t^l = 0$ and $t_u = 62$; $\mathbf{w}^l = [-11,0]^T$ and $\mathbf{w}^u = [0,11]^T$. Obviously, the size of the mapped virtual array is 12×12, which can be allocated in the 12×12 EVPA. See Figure 6.3.(a).

As regards the N-D time basis, we start from

$$\mathbf{T}_b' = \begin{bmatrix} 1 & 0 & 0 \\ -5 & -3 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

When $\mathbf{P}_p^{(10)} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, we have

$$\mathbf{A}^{(10)} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -5 & -3 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -6 & -3 \\ 1 & -1 \end{bmatrix}$$

It can be verified that $\mathbf{H}_k = \mathbf{A}^{(10)}\mathbf{U}_k$ holds if

$$\mathbf{H}_k = \begin{bmatrix} 3 & 0 \\ -2 & 3 \end{bmatrix} \text{ and } \mathbf{U}_k = \begin{bmatrix} -1 & 1 \\ 1 & -2 \end{bmatrix}$$

so we can make a 3×3 LSGP Partitioning as expected. As a result of this, $\mathbf{T}_b = \begin{bmatrix} -1 & 1 & 0 \\ 2 & 1 & 1 \\ 1 & -2 & 0 \end{bmatrix}$ and $det(\mathbf{T}_b) = 1$ with $\mathbf{r} = \mathbf{P}_p^{(10)}\mathbf{St}_{N-1} = [1,0]^T$. The permutations of $\mathbf{w}^l$, $\mathbf{w}^u$ and $\mathbf{S}$ and must be carried out. Then, we have $\mathbf{w}^l = [0,-11]^T$, $\mathbf{w}^u = [11,0]^T$ and $\mathbf{S} = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$. Substituting $\mathbf{B}^{-1}$ and $\mathbf{T}_b$ into eqn (6.13), we can produce the boundary of a supernode, which is similar to that for Algorithm 6.4.1.

Next, we discuss the outgoing data problem. Note that $\mathbf{D}^{q'}$ is the same as eqn (6.19). For all potential supernode dependencies $\mathbf{d}_{xxx}^s$ and for all $\mathbf{d}_i^{q'}$, we list their DC as $DC_{xxx,i}$'s (Notice that $DC_{xxx,i}$, whose subscript is the combination of the subscripts of $\mathbf{d}_{xxx}^s$ and $\mathbf{d}_i^{q'}$, stands for the out-going data resulting from the i-th original dependency vector and

flowing along the direction $[x,x,x]^T$ in the supernode space). For instance, for $\mathbf{d}^s_{001}$ whose area is confined by $\overline{C_0}\ \overline{C_1}C_2$, we have

$$\mathrm{DC}_{001,0} = [0,6)\times[0,3)\times[8,8);\quad \mathrm{DC}_{001,1} = [0,7)\times[0,2)\times[6,8);$$
$$\mathrm{DC}_{001,2} = [0,7)\times[0,2)\times[8,8);\quad \mathrm{DC}_{001,3} = [0,7)\times[0,2)\times[7,8)$$

Obviously, $\mathrm{DC}_{001,0} = \mathrm{DC}_{001,2} = \emptyset$ and $\mathrm{DC}_{001,1} \subset \mathrm{DC}_{001,3}$. So the DC for $\mathbf{d}^s_{001}$ is $\mathrm{DC}_{001} = \cup_{i=0}^{3}\mathrm{DC}_{001,i} = [0,7)\times[0,2)\times[6,8)$.

Similarly, we have $\mathrm{DC}_{010} = [0,7)\times[2,3)\times[0,8)$, $\mathrm{DC}_{011} = [0,7)\times[2,3)\times[6,8)$, $\mathrm{DC}_{100} = [6,7)\times[0,3)\times[0,8)$ and $\mathrm{DC}_{101} = \mathrm{DC}_{110} = \mathrm{DC}_{100} = \emptyset$.

These DC's flow directions and time-delays should be determined using eqn (6.14). Marking them to DC's as superscripts, we obtain $\mathrm{DC}^{-1,0,3}_{001}$, $\mathrm{DC}^{0,1,1}_{010}$, $\mathrm{DC}^{-1,1,4}_{011}$ and $\mathrm{DC}^{1,-1,5}_{100}$.

Finally, the parallel algorithm for LSGP Case of Example 6.1.1 is as follows

DOALL $j_0 := 0$ TO 3
  DOALL $j_1 := 0$ TO 3
    $w''_0 = 3j_0$
    $w''_1 = -11 + 3j_1$
    FOR $t_2 := 0$ TO 62
      $w'_0 = w''_0$, $t_0 := \lceil \frac{w'_0}{3} \rceil$
      $w'_1 = w''_1 - 2t_0$, $t_1 := \lceil \frac{w'_1}{3} \rceil$
      $\mathbf{w}' := k\mathbf{t} - \mathbf{w}'$, $w''_0 := w''_0 - 1$, $q''_{00} := 7t_0 - 7t_1$
      FOR $q''_0 := \max(q''_{00}, 0)$ TO $\min(q''_{00} + 20, 6)$
        $q''_{01} := q''_0 - 13t_0 + 4t_1 - 3t_2$
        FOR $q''_1 := \max(q''_{01}, 0)$ TO $\min(q''_{01} + 20, 2)$
          $q''_{02} := 3q''_0 + q''_1 - 23t_0 + 40t_1 + 3t_2$
          FOR $q''_2 := \max(\lceil \frac{q''_{02}}{2} \rceil, 0)$ TO $\min(\lfloor \frac{q''_{02}+10}{2} \rfloor, 7)$
            $A(q''_0, q''_1, q''_2) := A(q''_0 - 1, q''_1, q''_2) + A(q''_0, q''_1 - 1, q''_2 - 2) +$
            $+ A(q''_0, q''_1 - 1, q''_2) + A(q''_0, q''_1, q''_2 - 1)$
      Data transfer operations

We can check partly the correctness of our method for Example 6.1.1 in some way. As expected, there are, in total, 4851 nodes contained in 327 valid supernodes. It is difficult to describe the Data transfer operations here, since it involves a series of complex output, relay and input operations based on DC's. Chapter 7 will develop a mechnism to deal with the problem.

6.5.5 Algorithm Generation for (N-1)-D SUC Case

With regards to the (N-1)-D array with SUC mesh, we have $det(\mathbf{T}) = 1$ for the case of SUC. There is no need for the further LSGP partitioning step, as it can be taken as a special case of LSGP Case where $\mathbf{T}_b = \begin{bmatrix} \overline{\mathbf{o}} & 1 \\ \mathbf{I} & \mathbf{o} \end{bmatrix}$. Without derivation we give the parallel algorithm for Example 6.1.1

DOALL $j_0 := 0$ TO 3
  DOALL $j_1 := 0$ TO 3
    FOR $j_2 := 0$ TO 62
      $q''_{00} := -6j_0$
      FOR $q''_0 := \max(q''_{00}, 0)$ TO $\min(q''_{00} + 20, 5)$
        $q''_{01} := q''_0 + 6j_0 - 11j_1$
        FOR $q''_1 := \max(q''_{01}, 0)$ TO $\min(q''_{01+20}, 10)$
          $q''_{02} := 3q''_0 + q''_1 + 20j_0 + 13j_1 - 2j_2$
          FOR $q''_2 := \max(\lceil \frac{q''_{02}}{2} \rceil, 0)$ TO $\min(\lfloor \frac{q''_{02}+10}{2} \rfloor, 1)$
            $A(q''_0, q''_1, q''_2) := A(q''_0 - 1, q''_1, q''_2) + A(q''_0, q''_1 - 1, q''_2 - 2) +$
                $+ A(q''_0, q''_1 - 1, q''_2) + A(q''_0, q''_1, q''_2 - 1)$
      Data transfer operations

A simple check shows that the parallel algorithm covers all the 4851 nodes. The Data transfer operations are also dealt with in Chapter 7 as a special and simplified case of (N-1)-D SBC Case, finding the DC's is omitted.

6.6 From Inequalities To Boundaries

In the sections above, a derivation from a set of given inequalities to boundaries of nested loops is essential. We hope that the derivation gives the minimum valid boundaries and consists of only necessary expressions in the boundaries. It is easy to form a set of inequalities from nested loops, but, unfortunately, the reverse derivation is not so easy.

For simplicity, collect all the expressions of the set of inequalities into a matrix

$$E = \{\mathbf{A}_{m,N}\mathbf{j} + \mathbf{B}_{m,n}\mathbf{m} + \mathbf{c}\} = \{e_0, \cdots, e_{m-1}\} \tag{6.24}$$

where $e_i = a_{i,0}j_0 +, \cdots, +a_{i,N-1}j_{N-1} + b_{i,0}m_0 +, \cdots, +b_{i,n-1}m_{n-1} + c_i$. $E \leq \mathbf{0}$ is the set of inequalities, while E is a collection of expressions.

Definition 6.6.1 *E^j is a sub-set of E such that $a_{i,l} = 0\ \forall l \in [j+1, N-1]$ for all expressions it contains. L^j is a sub-set of E^j such that $a_{i,l} < 0$ for all expressions in it; U^j is a sub-set of E^j such that $a_{i,l} > 0$ for all expressions, and $\mathbf{m} = [m_0, \cdots, m_{n-1}]^T$ is variables.*

The procedure of determining the boundaries from a set of inequalities consists of two basic operations, finding all possible lower and upper bounds and deleting redundant bounds. These operations are discussed below.

6.6.1 Finding All Possible Lower and Upper Bounds

For each j_i, create all possible expressions for lower and upper boundaries, L^i and U^i, which involves all possible inequality relationships with $j_0, \cdots, j_i$. The process is implemented by a recursive procedure:

Step 1 Let $E^{N-1} := E$; $i := N-1$;

Step 2 Build L^i and U^i from E^i. Create E^{i-1} by collecting any expressions of E^i such that $a_i = 0$.

Step 3 For any pair of expressions, one from L^i and the other from U^i, make a modified sum, noted as $\oplus$, so as to cancel the a_i, produing a new expression. For instance, let $e^1 = 2j_0+2j_1+3$ and $e^2 = j_0-3j_1-1$. $e^{\oplus}e^2 = 3(2j_0+2j_1+3)+2(j_0-3j_1-1) = 8j_0+7$. The created expression is then appended to E^{i-1}.

Step 4 If E^{i-1} contains a number of expressions whose coefficients are linearly-related with positive factors, keep the one whose constant item is smallest if $a_l > 0$ (or the one whose constant item is largest if $a_l < 0$), remove the others, because these expressions are associated with the upper bound of j_l, where $l : (a_l \neq 0) \cap (a_k = 0,\ \forall k \in [l+1, i])$.

Step 5 $i := i-1$. goto Step 2.

The procedure is essentially the Fourier-Motzkin elimination method [90]

For our example, E^2 is the collection of expressions of eqn (6.17). $E^1 = \{-j_0, 3j_0 - 10\}$

$$L^2 = \left\{ \begin{array}{l} l_0^2 : 7j_0 - j_2; \\ l_1^2 : -17j_0 + 2j_1 - j_2 - 25; \\ l_2^2 : 2j_0 + j_1 - 2j_2 - 5; \\ l_3^2 : j_0 + 2j_1 - j_2 - 70; \end{array} \right\} \quad U^2 = \left\{ \begin{array}{l} u_0^2 : -7j_0 + j_2 - 25 \\ u_1^2 : 17j_0 - 2j_1 + j_2 - 1 \\ u_2^2 : -2j_0 - j_1 + 2j_2 - 30 \\ u_3^2 : -j_0 + j_2 - 40 \end{array} \right\}$$

We find that $l_0^2 \oplus u_0^2$, $l_1^2 \oplus u_1^2$ and $l_2^2 \oplus u_2^2$ leave only constants, and can be ignored. However, the following expressions will be appended to E^1.

$$\begin{array}{ll} l_0^2 \oplus u_1^2 : 24j_0 - 2j_1 - 1; & l_2^2 \oplus u_0^2 : -12j_0 + j_1 - 55 \\ l_0^2 \oplus u_2^2 : 12j_0 - j_1 - 30; & l_2^2 \oplus u_1^2 : 36j_0 - 3j_1 + 7 \\ l_0^2 \oplus u_3^2 : 6j_0 - 40; & l_2^2 \oplus u_3^2 : j_1 - 85 \\ l_1^2 \oplus u_0^2 : -24j_0 + 2j_1 - 50; & l_3^2 \oplus u_0^2 : -6j_0 + 2j_1 - 95 \\ l_1^2 \oplus u_2^2 : -36j_0 + 3j_1 - 80; & l_3^2 \oplus u_1^2 : 18j_0 - 71 \\ l_1^2 \oplus u_3^2 : -18j_0 + 2j_1 - 65; & l_3^2 \oplus u_2^2 : 3j_1 - 170 \\ l_3^2 \oplus u_3^2 : 2j_1 - 110 & \end{array}$$

It can be seen that $24j_0 - 2j_1 - 1$, $12j_0 - j_1 - 30$ and $36j_0 - 3j_1 + 7$ are linearly-related. Since $a_1 < 0$, keep $24j_0 - 2j_1 - 1$ which has the largest constant, and remove the others. Similarly, keep $-24j_0 + 2j_1 - 50$, while removing $-36j_0 + 3j_1 - 80$ and $-12j_0 + j_1 - 55$; keep $2j_1 - 110$, remove $3j_1 - 170$ and $j_1 - 85$; retain $18j_0 - 71$, remove $6j_0 - 40$; retain $3j_0 - 10$, remove $18j_0 - 71$. Then, we have

$$E^1 = \left\{ \begin{array}{lll} -j_0; & 3j_0 - 10; & -6j_0 + 2j_1 - 95 \\ -18j_0 + 2j_1 - 65; & 24j_0 - 2j_1 - 1; & -24j_0 + 2j_1 - 50 \\ 2j_1 - 110 & & \end{array} \right\}$$

and

$$L^1 = \{24j_0 - 2j_1 - 1\}; \quad U^1 = \left\{ \begin{array}{ll} 2j_1 - 110; & -24j_0 + 2j_1 - 50 \\ -6j_0 + 2j_1 - 95; & -18j_0 + 2j_1 - 65 \end{array} \right\}$$

Similarly, $L^0 = \{j_0\}$, $U^0 = \{3j_0 - 10\}$.

6.6.2 Deleting Redundant Bounds

Although explicitly redundant bounds have been removed, there may be still some implicit redundancy.

For instance, there are two expressions $e_1 : j_0 + j_1 + 3$ and $e_2 : -j_0 + j_1 + 5$. Make the modified difference, noted as $\ominus$, such that a_1 is cancelled, that is, $d = e_1 \ominus e_2 = 2j_0 - 2$. Let d^{max} and d^{min} indicate the possible maximum value and the possible minimum value of d, respectively. Because the two expressions are associated with the upper bound of j_1,

we can retain e_1 and remove e_2 if $d^{min} \geq 0$, or retain e_2 and remove e_1 if $d^{max} \leq 0$. Since $a_0 > 0$ in d, replacing j_0 with its lower bound and upper bound produces d^{min} and d^{max}, respectively. Consider the following four cases: $l_0 = 0$; $l_0 = 2$; $u_0 = 0$ and $u_0 = 2$. For the first case, we have $d^{min} \geq -2$, so e_2 cannot be removed; for the second case, because $d^{min} \geq 0$, we can remove e_2; for the third case, because $d^{max} \leq -2$, e_1 can be removed; for the last case, neither can be removed, because $d^{max} \leq 2$.

If the two expressions are $e_1 : j_0 - j_1 + 3$ and $e_2 : -j_0 - j_1 + 5$, d remains unchanged, as well as d^{max} and d^{min}. However, since the two expressions are associated with the lower bound of j_1, we can retain e_2 and remove e_1 if $d^{min} \geq 0$, or retain e_1 and remove e_2 if $d^{max} \leq 0$.

It is not an easy job to find the possible minimum value and the possible maximum value for an expression in general cases. As stated above, suppose the lower and upper bounds of j_k are noted collections of expressions $L^k = \{l_0^k, \cdots, l_{n_l^k}^k\}$ and $U^k = \{u_0^k, \cdots, u_{n_u^k}^k\}$, respectively. The l^k's are the functions of $j_0, \cdots, j_{k-1}$, as are the u^k's. The possible minimum values of a given expression $d^i = a_0 j_0 + \cdots, a_i j_i$ form a set of numbers, noted as $D^{min} = \{d_0^{min}, \cdots, d_{n^{min}}^{min}\}$, instead of a unique number. If L^k and U^k are known $\forall k \in [0, i]$, the D^{min} of d can be derived recursively:

Step 1 $D^{min} = \{d^i\}$

Step 2 Take an expression $d^{min} \in D^{min}$. For the d^{min}, if $a_i > 0$, substitute j_i with its lower bounds in L^i, that is, make $d^{min} \oplus l^i$, $\forall l^i \in L^i$ (this is possible because $a_i < 0$ for all l^i's in L^i). Replace the d^{min} with the newly created n_l^i expressions not involving j_i. If $a_i < 0$, do the same but with U^i.

Do this for all d^{min}'s of D^{min}.

Step 3 $i := i - 1$, goto Step 2.

We can find the possible minimum value set, D^{max}, of d^i in a similar way except that we work the d^{max} with U^i if $a_i > 0$, or with L^i if $a_i < 0$. We say $d^i \geq 0$ if $d^{min} \geq 0$ for all d^{min}'s in D^{min}; similarly, we say $d^i \leq 0$ if $d^{max} \leq 0$ for all d^{max}'s in D^{max}.

Removing the implicitly redundant bounds is also a recursive procedure, applied from

L^1 and U^1 to L^{N-1} and U^{N-1}. Note that L^0 and U^0 is omitted since all redundant bounds are explicit in this case. The algorithm is as follows:

Step 1 $i := 1$

Step 2 Take two expressions $l^i_{k_1}$ and $l^i_{k_2} \in L^i$, $k_1 \neq k_2$. Make $d^{i-1} = l^i_{k_1} \ominus l^i_{k_2}$ such that j_i's term is cancelled. Create D^{min} and D^{max} from d^{i-1}. Delete $l^i_{k_2}$ if $d^{i-1} \geq 0$, or delete $l^i_{k_1}$ if $d^{i-1} \leq 0$.

Repeat this for all pairs of expressions of L^i.

Step 3 Do the same for U^i as Step 2, but delete $l^i_{k_1}$ if $d^{i-1} \geq 0$, or delete $l^i_{k_2}$ if $d^{i-1} \leq 0$.

Step 4 $i := i + 1$, goto Step 2.

The operation of deleting implicitly redundant bounds is necessary for general cases. For Example 6.1.1, no implicitly redundant bounds are found for the L^1, U^1, L^2 and U^2 and so are the lower and upper boundaries of eqn (6.18).

It is interesting to see the operation of the algorithm for a simple example.

Example 6.6.1 *A Simple Example*

Given the nested loops

FOR $i_0 = 1$ to m_0
  FOR $i_1 = 2i_0$ to m_1
    FOR $i_2 = 2i_0 + i_1 - 1$ to $\min(2i_1, m_2)$

Let us transform the nested loops to a set of inequalities, then transform the set of inequalities back to nested loops by means of the algorithm. The resulting set of nested loops is:

FOR $i_0 = 1$ to $\lfloor min(\frac{m_2+1}{4}, m_0) \rfloor$
  FOR $i_1 = 2i_0$ to $\min(m_2 - 2i_0 + 1, m_1)$
    FOR $i_2 = 2i_0 + i_1 - 1$ to $\min(2i_1, m_2)$

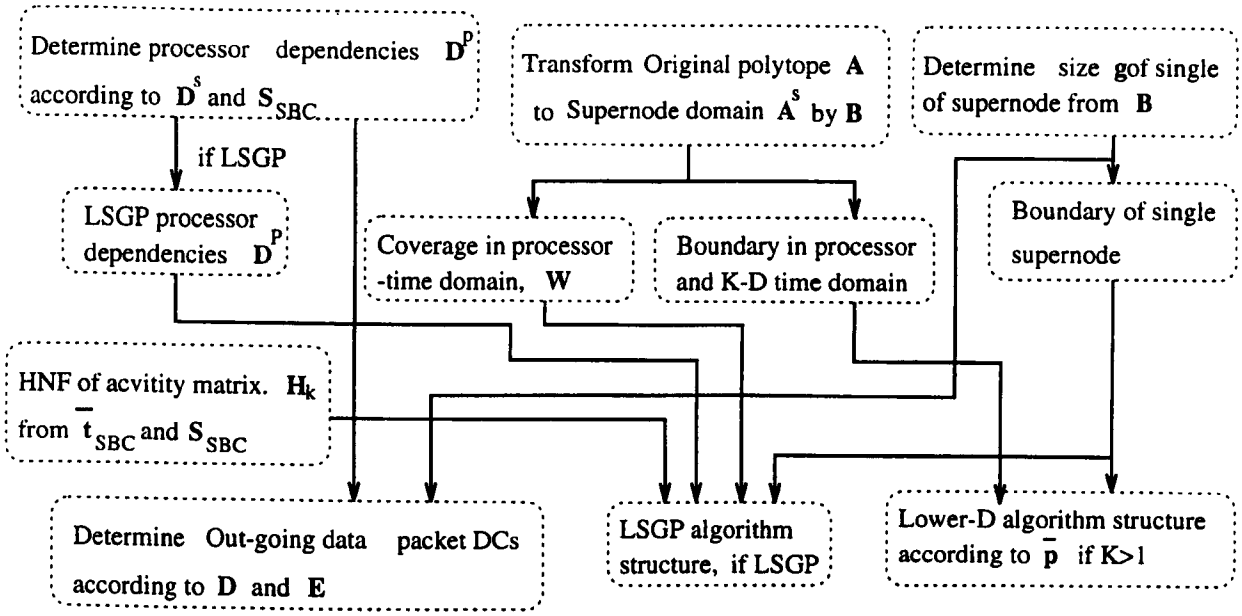


Figure 6.4: Chart of Program Transformation

More bounds are imposed on the original nested loop. It is easy to judge their correctness. Since $i_2 \geq 2i_0 + i_1 - 1$, then $i_1 > m_2 - 2i_0 + 1 \Longrightarrow i_2 > m_2$. This means that there is no room for the i_2 loop since i_2's lower bound is beyond its upper bound. Furthermore, if $i_0 > \frac{m_2+1}{4}$, substituting it into the lower and upper bounds of i_1 yields $i_1 > \frac{m_2+1}{2}$ meanwhile $i_1 < \frac{m_2+1}{2}$, so there is no room for the i_1 loop. Therefore, the algorithm is useful in that it reduces the invalid part of the polyhedron generated by the original nested loops.

6.7 Summary

We propose a method to generate parallel algorithms from sequential algorithms. Using our partitioning and mapping approach, algorithms are developed to determine the loop boundaries of the parallel algorithm. The problem of out-going data is discussed in detail, and the operational structure in a single processor is constructed. See figure 6.4.

To execute on a lower-dimensional array, special attention is paid to the synchronisation of every processor so that the parallel algorithm can run in a processor array with systolic features. Fortunately, we find a method for the synchronisation which requires a very simple mechanism and avoids complex control. For the methods involving LSGP partitioning a simple algorithm is derived to determine which supernode should be executed.

The correctness of the methods are checked.

The method presented here gives realistic approach to automatically implementing a wide range of URE problems for both hardware systolic array design and software implementation of regular arrays. In order to achieve actual wide applications, our methods are able to cope with the processing arrays with various physical limitations on dimension, shape and communications. We also pay attention to achieving high efficiency by means of supernode structure which results in packaged data communication.

Chapter 7

Parallel Code Generation

This chapter discusses methods of transforming sequential URE algorithms to parallel codes. As a result of the theoretical work, proposed in the previous chapters, actual parallel codes in Meiko C for a transputer array are generated automatically. To achieve this, some special practical problems for the generation of parallel codes are resolved, such as memory organization and data communications.

7.1 Introduction

In the last chapter, we have solved some important problems with respect to code generations, such as the boundary of the transformed algorithm, and the structure of parallel algorithms. However, the work has not yet finished. We must actually generate parallel code automatically. For this purpose, further practical problems have to be taken into consideration, such as, creating a working model for parallel programs which can result in relatively easy generation of actual parallel code. Finally, since we restrict ourselves to using a distributed-memory machine, we have to solve the problem of data storage and data communications at the machine-level.

In this chapter, we first build a processor array in Section 7.2. Section 7.3 and Section 7.4 contribute to two practical problems of the parallel codes: data storage and data communications, respectively. Section 7.5 describes the outline of the parallel codes. Finally, some minor algorithms for building data communications can be found in Appendix E.

7.2 Processor Array

Although in previous chapters we have had much theoretical preparation, to generate an actual parallel program for a specific computing facility using a specific language is a very complex and challenging problem.

7.2.1 Multi-Processor Machines

With regard to memory, there are two kinds of parallel computing machines, shared-memory and distributed-memory machines, respectively. In the distributed-memory machine, each processor has its own local memory, forming a node of an array, while for shared-memory, all processors share a global memory. Distributed-memory machines impose stricter conditions than shared-memory machines with regard to the overhead of operations and the difficulty of programming. Our methods were originally aimed to cope with the distributed case, but can be applied to shared-memory models also.

We consider regular-structured distributed-memory processor arrays in which each processor performs identical operations at each step. These kinds of arrays behave like a software-controlled systolic array. The attractive point of this arrangement is the identity of operations on the processors and the regularity of the whole array. The "regular-structured" array means a hypercube-shaped array, with unidirectional or bi-directional mesh-links (i.e., SUC and SBC).

7.2.2 Building an Array and Creating Communication Meshes

To build a processor array, we need to describe briefly the features of our available parallel computing resource.

The distributed-memory machine used in this study is a transputer array consisting of 16 T800 transputers, each with 4Mb memory. A software environment, CSTools, is also provided to create a computing array [64]. In CSTools, CSBuild facilities allow a user-written sequential C program which, running on the host computer, permits the programmer to describe the array and the physical link mesh explicitly and also allows control of the loading of the parallel codes.

The available parallel language for the transputer array is Meiko C [63]. In Meiko C, inter-process communication channels have to be created in the application. To specify a processor array, the number of dimensions, M, and the size, $l_0, \cdots, l_{M-1}$, of the array in each dimension, are given as the design parameters. In the CSBuild program, the processor array is defined as a regular array (i.e., A^M in Definition 1.2.1). Then, we specify object codes of the program fragments to be loaded into each processor of the array (in fact, all the object codes are identical). Thirdly, the coordinate, $\mathbf{a}$, of the processor in the array, must be passed to each process as a badge for building the inter-process communication mesh, as well as a parameter for confining the computation domain assigned to the processor.

In the CSBuild program, all the necessary physical links among the processors are built. The physical links are created according to the given interconnection primitives $\mathbf{P} = [\cdots, \mathbf{p}_i, \cdots]$. For each processor, we have to determine all allowable links. That is

Algorithm 7.2.1 *Build Physical Links*

For all $\mathbf{a} \in A^M$
  For all $\mathbf{p}_l \in \mathbf{P}$
    $\mathbf{a}' = [\cdots, a'_i, \cdots]^T := \mathbf{a} + \mathbf{p}_l$
    IF $\mathbf{a}' \in A^M$ (i.e., $\forall\ i,\ 0 \leq a'_i < l_i$)
      create a link from $\mathbf{a}$ to $\mathbf{a}'$

7.3 Supernode Storage

Once we have specified a connection mesh, we have to reserve local memory for each supernode allocated to a processor. For every processor, we must allocate a memory area which is as small as possible and establish a supernode storage structure which can keep the uniform data dependencies available. Problems with storage structures arise because we must keep the uniform distance of data dependencies available even where the dependencies go beyond the boundaries of supernodes. There are two ways to store a supernode.

One method is to store supernodes independently, see Figure 7.1. That is, we allocate

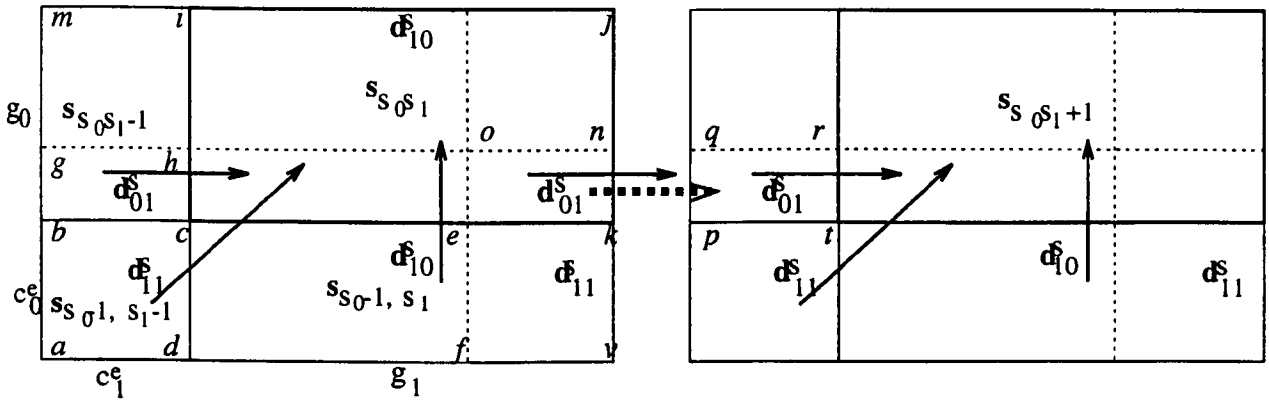


Figure 7.1: Memory layout of separate supernodes. Note that $g_0 = \overline{mb}$, $g_1 = \overline{dv}$, $c_0^e = \overline{ba}$ and $c_1^e = \overline{ad}$.

memory pieces $\prod_{i=0}^{N-1}(g_i + c_i^e)$ for each of the supernodes in a processor[1] where c_i^e (discussed later) is the extra size required to store the incoming data used by the supernode. If there are $t^u - t^l + 1$ time steps, we will have $t^u - t^l + 1$ such memory pieces.

The second method is to store all the supernodes of one processor together in such a way that they keep their neighbouring relations unchanged, so the internal data flows can be avoided, see Figure 7.2. This can be thought as storing the local supernode domain together "as a whole".

The first method suffers from a serious setback: there must be internal data flows, e.g., the area $\overline{eonke}$ of $\mathbf{s}_{s_0 s_1}$ to the area $\overline{pqrtp}$ of $\mathbf{s}_{s_0 s_1+1}$ in Figure 7.1. In fact. If $\mathbf{d}^s \neq \mathbf{0}$ but $\mathbf{d}^p = \mathbf{S}\mathbf{d}^s = \mathbf{0}$, there will be a dependency between two supernodes in the same processor. For the Lower-D Case, the number of internal data flows is greater than inter-processor data flow. The internal data flows are also costly and have to be avoided. The second approach complicates the problem of storage. We must know how large the memory space should be for the local supernode domain. Secondly, each of the supernodes must be located to a suitable position of the memory space, which may not be obvious.

We cannot say the first method is completely useless, because the second may require much larger memory space than the first. Here, we focus on the the second since it is more time-efficient. In some circumstances, the first approach may prove to be better, especially when memory is at a premium.

[1]Note that the different types of variables, e.g., double precision, float and integer, take different sizes, s, of memory, so the actual memory space should be enlarged by s. For brevity we assume s = 1.

7.3.1 From Supernode Domain to Local Supernode Domains

Obviously, it is unnecessary for every processor to allocate a memory space large enough to contain the whole supernode domain. In order to allocate sufficient and necessary memory space, we have to go back to the supernode space to determine a partitioning, called the local supernode domain which is assigned to a processor (this procedure is quite similar to that of Subsection 5.3.2). Then the local supernode domain which may be irregular has to be expanded to form a hypercube because a hypercube-shaped memory space can produce a uniform distance of data dependencies.

Eqn (6.11) defines the hyperplanes of the supernode polyhedron and $\mathbf{V}^e$ are its vertices. They provide all the information describing the supernode polyhedron. Recalling $\mathbf{j}' = [j_0, \cdots, j_{M-1}]^T = \mathbf{Ss}$ assigns supernodes $\mathbf{s}$ into the virtual processor $\mathbf{j}'$ [2] we do the reverse: for a particular $\mathbf{j}$, determine all the $\mathbf{s}$'s mapped onto it. Applying the algorithm of Subsection 5.3.2 on the polyhedron of eqn (6.11), we obtain all the vertices of the local supernode polyhedron assigned to the virtual processor $\mathbf{j}'$.

With the vertices of the local supernode polyhedron, it is easy to find its length, n_i, for each dimension. These lengths confine the sizes of a smallest hypercube which can contain the local supernode polyhedron. It is easy to find the first supernode (e.g., the left-bottom-corner point in 2-D), noted as $\mathbf{s}_0$, of the supernode hypercube, which is used later as a reference to locate any supernodes in the hypercube.

We have to repeat this procedure for every processor. It is wasteful to have a set of n_i's, $i = 0, \cdots, N-1$, for each of the processors, because we must have a set of uniform n_i's to generate uniform parallel codes for every processor. Therefore, for the i-th dimension, we choose the largest n_i of all processors to be the length n_i. Clearly for the processor we must allocate a memory space of size $\prod (n_i g_i)$ which is necessary and sufficient for the required data storage.

We can gain an idea of the shape of the local supernode polyhedron as follows. For the case of (N-1)-D array with SUC mesh, the local supernode polyhedron is a straight line segment lying along one dimension, so the local supernode hypercube containing the

[2] Note that without LSGP partitioning, a virtual processor is also the real processor, i.e., $\mathbf{a} = \mathbf{j}'$. As known, with the LSGP partitioning, c virtual processors will be compressed onto one real processor $\mathbf{a}$.

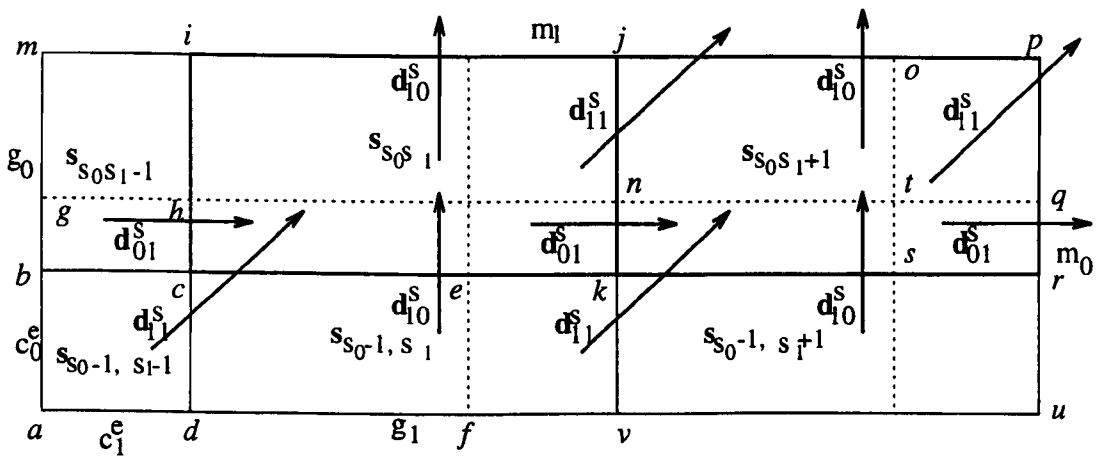


Figure 7.2: Local-supernodes memory space layout. Note that $g_0 = \overline{mb}$, $g_1 = \overline{dv}$, $c_0^e = \overline{ba}$, $c_1^e = \overline{ad}$, $m_0 = \overline{pu}$ and $m_1 = \overline{mp}$.

line segment is quite compact. For the case of SBC mesh, we also use a line segment, but, stretched in N dimensions. In this case, we have to construct a large loose hypercube to contain the line segment. The useful points in the hypercube must be very sparse. When the hypercube is too large to be allocated in available memory, we have to use the first method of storing supernodes at the cost of time efficiency.

7.3.2 Supernode Storage and in/out Data Storage

Here, we focus on the second method of supernode storage. The following problems must be addressed. Firstly, to maintain the uniform distance of data dependencies, some margin area must be attached the lower sides of the local supernode hypercube to store the data transferred from other processors. So the total allocation of memory space will be larger than $\prod (n_i g_i)$. We must know exactly how large it should be. Secondly we must determine the locations of each supernode and each of the input/output DC's in the modified local supernode hypercube (see Chapter 6). Furthermore, since the modified local supernode hypercube is actually stored as a line in the memory space, we should derive a formula to figure out the addresses of the supernodes and input DC's.

2-D example

Let us consider a 2-D example in the Q' space defined in Subsection 6.2.3. See Figure 7.2. In the layout, only $\mathbf{s}_{s_0 s_1}$ and $\mathbf{s}_{s_0 s_1+1}$ are the supernodes assigned into the same proces-

sor, while $\mathbf{s}_{s_0-1,s_1-1}$, $\mathbf{s}_{s_0-1,s_1}$, $\mathbf{s}_{s_0-1,s_1+1}$ and $\mathbf{s}_{s_0,s_1-1}$ are assigned to other processors but contribute data dependencies to the internal supernodes. Henceforth, we define genearlly that a symbol "$\overline{vxyzv}$" stands for a cubic region confined by points v, x, y and z. For instance, $\overline{abcda}$ indicates the square at the left-bottom corner in Figure 7.2.

At first, consider $\mathbf{s}_{s_0s_1}$, $\overline{cijkc}$. three DC's are received as input data: DC_{01} of size $(g_0 - c_0^e) \times c_1^e$ from $\mathbf{s}_{s_0,s_1-1}$, DC_{10} of size $c_0^e \times (g_1 - c_1^e)$ from $\mathbf{s}_{s_0-1,s_1}$ and DC_{11} of size $c_0^e \times c_1^e$ from $\mathbf{s}_{s_0-1,s_1-1}$. Consider DC_{01}. We must allocate a memory space $\overline{bmicb}$ of $g_0 \times c_1^e$ attached to the left of $\mathbf{s}_{s_0s_1}$ as a shadow $\mathbf{s}_{s_0,s_1-1}$. The same arguments hold for DC_{10} and DC_{11}, thus the memory space $\overline{dckvd}$ is attached underneath $\mathbf{s}_{s_0s_1}$ and the memory space $\overline{abcda}$ is attached to the bottom-left corner. The memory space for the 2 supernodes should be extended from $g_0 \times 2g_1$ to $m_0m_1 = (g_0 + c_0^e) \times (2g_1 + c_1^e)$, where m_i are the size of the local-supernode memory space in the m_i-th dimension.

The first point of a square is a key reference point by which the whole square can be located with its known sizes. It is easy to see that the first point of $\mathbf{s}_{s_0s_1}$ is *point* $c = [c_0^e, c_1^e]$ and the first point of $\mathbf{s}_{s_0,s_1+1}$ is *point* $k = [c_0^e, c_1^e + g_1]$.

After the processor receives all input data to an input buffer, the in-coming DC's will be transferred from an input buffer to cubic areas in supernode memory space. It is more convenient to express the first points of these cubic areas using the first point of the relevant supernode as a base address and adding the correct offset. For example, DC_{01} is moved to $\overline{bghcb}$, its first point is *point* b = *point* c - $[0, c_1^e]$; DC_{10}, to $\overline{dcefd}$, *point* d = *point* c - $[c_0^e,0]$; and DC_{11} to $\overline{abcda}$, *point* a = *point* c - $[c_0^e, c_1^e]$, etc.

For the output data, consider supernode $\mathbf{s}_{s_0,s_1+1}$. The first point of the area $\overline{njotn}$ which contributes $\mathbf{d}_{10}^s$ is *point* n = *point* $k + [g_0 - c_0^e, 0]$. The out-going data within $\overline{njotn}$ will be transferred to an output buffer. Similarly, *point* t = *point* $k + [g_0 - c_0^e, g_1 - c_1^e]$ is the first point of $\overline{topqt}$ which contributes $\mathbf{d}_{11}^s$; and *point* s = *point* $k + [0, g_1 - c_1^e]$ is the first point of $\overline{stqrs}$ which contributes $\mathbf{d}_{01}^s$.

The supernode memory square $\overline{ampua}$ will actually be stored as a line in memory. Any smaller cubes, such as a supernode and DC's, in the large cube cannot be stored continuously, so we call them segment-distributed cubes. In the parallel codes to be generated later in Section 7.5, there are two specially designed functions, **CopyFrom-**

BufferToSupernode and **CopyFromSupernodeToBuffer**, which can access all nodes of a segment-distributed cube described by the address of its first point and the size of the large hypercube $\overline{ampua}$ stored as a linear pattern.

General formula and operations

The above arguments can be extended to generalised cases. Suppose that each of DC's is noted as $DC_{\mathbf{d}_i^s}$, $i \in [0, N_{DC}-1]$, and has a size of $[h^l_{0,i}, h^u_{0,i}] \times \cdots \times [h^l_{N-1,i}, h^u_{N-1,i}]$, with supernode dependency $\mathbf{d}_i^s = [x_{0,i}, \cdots, x_{N-1,i}]^T$ and $x_{j,i} = 0, 1$. When compiling, we must do the following

Step 1 $\forall\, j \in [0, N-1]$, let $c_j^e = max_{i=0}^{N_{DC}-1} x_{j,i}(g_i - h^l_{j,i})$ and $m_j = n_j g_j + c_j^e$.

Step 2 Let $f_{N-1} = 1$. Let $f_j = m_{j-1} f_{j-1}$, j=N-2,$\cdots$,0. Let $\mathbf{f} = [f_0, \cdots, f_{N-1}]^T$.

Step 3 $\forall\, i \in [0, N_{DC}-1]$, let

$$r_i^i = -\sum_{j=0}^{N-1} f_j(1 - x_{j,i})(g_j - h^l_{j,i}) \tag{7.1}$$

r_i^i is the relative distance difference from a supernode to the segment-distributed cubic area of storing the input DC .

Step 4 $\forall\, i \in [0. N_{DC}-1]$, let

$$r_i^o = \sum_{j=0}^{N-1} f_j x_{j,i} h^l_{j,i} \tag{7.2}$$

r_i^o is the relative distance difference from a supernode to the segment-distributed cubic area collecting the output DC.

Then, in the parallel codes to be generated, the supernodes are located as follows:

Step 1 Allocate local-supernodes memory space

$$m = \prod_{j=0}^{N-1} m_j \tag{7.3}$$

Step 2 At time t, locate supernode s to address

$$a^t = \mathbf{f}^T(\mathbf{g} \otimes (\mathbf{s} - \mathbf{s}_0) + \mathbf{c}) \tag{7.4}$$

where the $\otimes$ operator is defined as $[a_0, \cdots, a_n]^T \otimes [b_0, \cdots, b_n]^T = [a_0 b_0, \cdots, a_n b_n]^T$, and $\mathbf{c}^e = [c_0^e, \cdots, c_{N-1}^e]^T$.

7.4 Data Flow and Relay

In this section, we consider the problem of data communication. In order to implement data communications, we must build a data flow and relay mechanism. We establish two data in/out buffers, IB and OB, because in this way more data can be collected to form a large data block so as to be transferred efficiently (however, as to be seen in Subsection 7.4.2, in some situations, to improve efficiency we will try to build direct data flows avoiding the buffers). Relay Buffers (RB) are also established to buffer the relay data. This mechanism will carry out the following tasks:

1. Collect data to IB and then determine their orientations and destination OB's.
2. If some data cannot arrive at their destination in one step, relay them via RB's to their destinations along appropriate directions with appropriate delays.

The mechanism provides data flow packets (IP and OP). Note that because Meiko C provides the function of transferring data by block, instead of individual items, the data flow unit is a Data Vector (DV) which contains the information about a data block and its location. A DV is described by three attributes: the buffer, the position in the buffer, and the contents. For example $DV = [OB, n(Q^0), Q^1]$ stands for Q^1 parking in OB at position $n(Q^0)$, where the Q's are data blocks and $n(Q)$ is a function determining the number of Q. A OP (or IP) consists of a number of DV. For instance, $OP = \{[OB, n(Q^0), Q^1], [RB, n(Q^2), Q^3]\}$. Obviously, the buffers involved in OP and IP are the source and destination of data blocks, respectively.

If LSGP partitioning is used, we face a more complicated situation, because a number of different IP's and OP's have to be produced for different locations of supernodes in the LSGP partitioning block. The non-LSGP case can be taken as a simplified version of the LSGP case. Therefore we deal with the LSGP case first even though it is more complex, and then describe briefly the non-LSGP and discuss the associated direct data flow problem.

Some notation should be introduced. As indicated previously, more than one supern-

ode dependency may contribute to an inter-processor dependency, so one inter-processor dependency may involve more than one DC. For instance, if $\mathbf{S} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $\mathbf{d}^p_{01}$ involves DC^{01}_{001} and DC^{01}_{101}; $\mathbf{d}^p_{10}$ involves DC^{10}_{010} and DC^{10}_{110}; and $\mathbf{d}^p_{11}$ involves DC^{11}_{011} and DC^{11}_{111}. Here, DC^{yy}_{xxx}'s are the same of those in subsection 6.3.2 but omitting the d^t from the superscript. For brevity, we merge all the DC^{yy}_{xxx}'s which share the same processor dependency into one data flow, noted as Q^{yy}. That is, $Q^{01} = DC^{01}_{001} + DC^{01}_{101}$, and so on.

7.4.1 LSGP Case

As discussed in Subsection 6.5.3, unnecessary data flows should be avoided. In the LSGP case, there are c different positions, marked with $\mathbf{w}'$, in a block of LSGP partitioning, where $c = k^{N-1}$ is the number of the supernodes in a block of LSGP partitioning and also the total LSGP compression factor mentioned before (See Figure 6.3.(b)). For each of these positions we have to carry out different input/output operations.

We have to design different $IP^{w_i}_{\mathbf{p}}$'s and $OP^{w_i}_{\mathbf{p}}$'s, where superscript w_i, $i = 0, \cdots, c-1$, indicates the ordinal number of computing each $\mathbf{w}'$, and subscript $\mathbf{p}$ shows the direction of the packet [3]. The $IP^{w_i}_{\mathbf{p}}$'s and $OP^{w_i}_{\mathbf{p}}$'s describe all the input and output operations for the $\mathbf{w}'$ marked as w_i. However, before establishing the IP's and OP's, we must mark w_i for $\mathbf{w}'$ of a processor and know the relations of the w_i to those of the adjacent processors.

Determining w_i's and finding its increment

First of all, we must determine the order of computing $\mathbf{w}'$ of a processor, which indicates the order of the IP's and OP's, too. That is, we should mark $\mathbf{w}'$with w_i, see Figure 7.3.(a). for the case of Example 6.1.1. Figure 7.3.(a) shows a LSGP block where for each $\mathbf{w}'$, "t" indicates the time to execute the $\mathbf{w}'$ and obviously also shows the order of execution. That is to mark the $\mathbf{w}'$ with w_t. For instance, $\mathbf{w}'_{0,2}$ is executed at $\mathrm{t} = 0$ and is marked as w_0. In fact, it is difficult to derive an explicit formula between $\mathbf{w}'$and w_i. Fortunately, in Algorithm 6.5.1, we have considered the problem. A short algorithm (Algorithm E.1) is presented in Appendix E to determine the computing order of every $\mathbf{w}'$. For the processor $\mathbf{a}$, letting $\Delta\mathbf{a} = \mathbf{0}$ and running Algorithm E.1, we can know which $\mathbf{w}'$ is accessed at time

[3] obviously $\mathbf{p}$ must be one of the interconnection primitives, i.e., $\mathbf{p} \in \mathbf{P}$

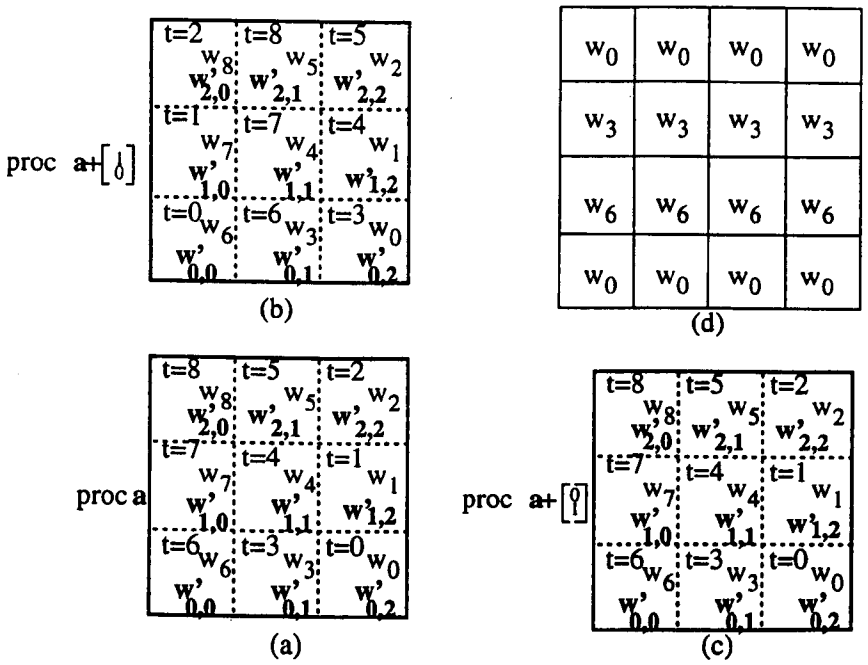


Figure 7.3: Marking w'with ordinal number w. (a), (b) and (c) show LSGP partitioning blocks; (d) shows an initial state of a processor array marked with ordinal number w

t, where Δa indicates a shift from the referenced processor in processor domain. Then, mark w' with w_t.

Comparing Figure 7.3.(b) and Figure 7.3.(a), we find an important fact: the processors in an array may not work with the same w' at a moment, so that an OP^{w_i} operation of one processor may be correspond to an $OP^{w_{i'}}$ operation of one adjacent processor, where $i \neq i'$. For instance, $OP_{\mathbf{p}}^{w_0}$ of the processor a should correspond to $IP_{\mathbf{p}}^{w_6}$ of the processor $\mathbf{a} + [1,0]^T$ when t = 0. Therefore in order to determine the corresponding relations of the sequences of the IP's and the OP's of two adjacent processors we must also know the order of computing w' of the adjacent processors.

For the adjacent processor $\mathbf{a} + [0,1]^T$, let $\Delta\mathbf{a} = [0,1]^T$, run Algorithm E.1 again. We find that it gives the same result as processor a, see Figure 7.3.(c). However, for the processor $\mathbf{a} + [1,0]^T$ in Figure 7.3.(b), letting $\Delta\mathbf{a} = [1,0]^T$ and running Algorithm E.1 once more, we see that when $t = 0$, $\mathbf{w}'_{0,0}$, which has been marked as w_6, is accessed. Thus, in this direction, we find an increment 6 with respect to w_i. Therefore, an increment vector $\overline{\Delta \mathbf{w}} = [6,0]$ can be defined. At time t, if the referencing processor a accesses w_i,

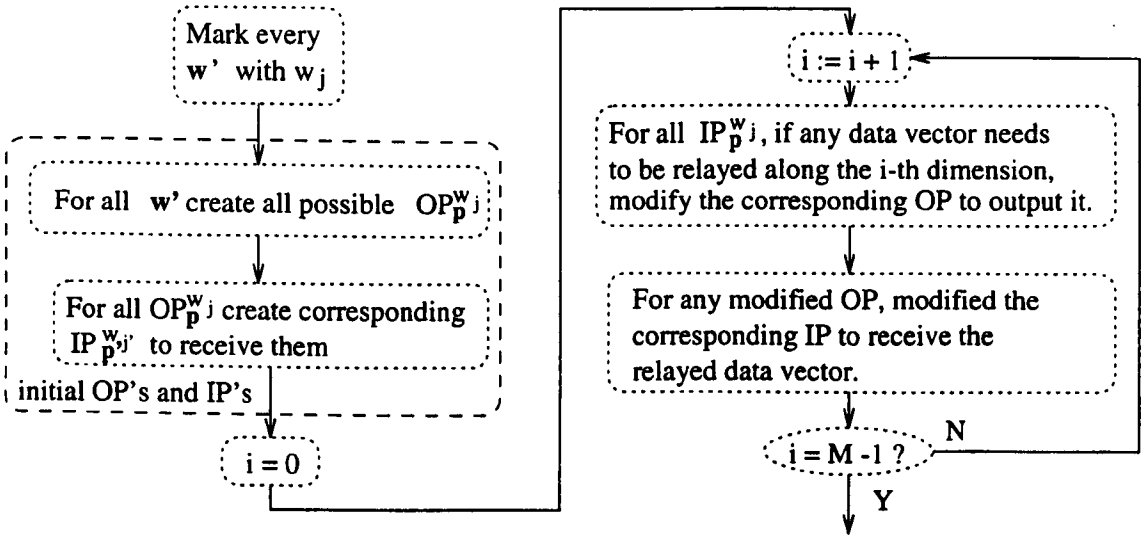


Figure 7.4: The Conceptual Chart of Creating IP's and OP's

processor $\mathbf{a}'$ will access $w_{i'}$, where $i' = (i + \overline{\Delta \mathbf{w}}(\mathbf{a}' - \mathbf{a}))$ mod c, see Figure 7.3.(d) which shows w_i's for a processor array at time $t = 0$.

Establish OP's and IP's

The procedure for establishing OP's and IP's is shown in Figure 7.4. Marking $\mathbf{w}'$ with w_j has been described above. Now for every position $\mathbf{w}'$ in the LSGP block, an OP can be figured out and is marked with the corresponding w_i. For each of the OP's we must create a properly marked IP to respond. So we have two initial sets of OP's and IP's, and have to yet consider data relay if there is any data which can not arrive at their destination in one step. Next, we use recursive modifications to make up all the IP's and OP's. For each direction in turn, we should check whether some data has not reached their terminal destination in this direction, and then modify the relevant $OP_{\mathbf{p}}^{w_i}$'s to send the data out once more and modify the corresponding IP's to receive them. At last, all data can arrive at their correct destinations.

More accurately, establishing OP's and IP's consists of the following steps.

Step 1 For each $\mathbf{w}'$ and for all $\mathbf{d}^p \in \mathbf{D}^p$, compute $\mathbf{d}^A$ by eqn (6.23). Set all entries after the first non-zero entry of $\mathbf{d}^A$ to zero, forming a vector $\mathbf{p}$, that is, for instance $\mathbf{d}^A = [0, -1, 1, 0]^T \longrightarrow \mathbf{p} = [0, -1, 0, 0]^T$. The $\mathbf{p}$ indicates the link we work with.

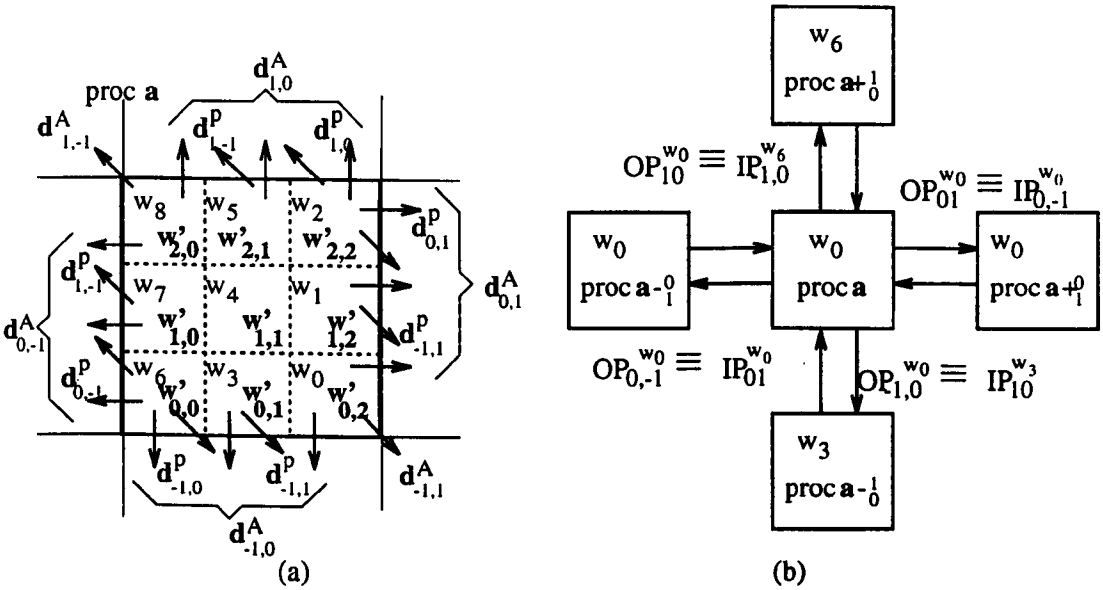


Figure 7.5: LSGP dependencies and data flows.

Create $OP_{\mathbf{p}}^{w_j}$, which consists of a number of data vectors which are attributed with $\mathbf{d}^A$.

Step 2 For each existing $OP_{\mathbf{p}}^{w_j}$, create its corresponding $IP_{\mathbf{p}'}^{w_{j'}}$'s, where $j' = j + \overline{\Delta \mathbf{w} \mathbf{p}} \bmod c$ and $\mathbf{p}' = -\mathbf{p}$. Now we have initial $OP_{\mathbf{p}}^{w_j}$'s and $IP_{\mathbf{p}}^{w_j}$'s.

Step 3 From i = 1 to i = M-1, for any data vector of any existing $IP_{\mathbf{p}}^{w_j}$'s, Null $\mathbf{d}^A$ of the data vector except the i-th element, form $\mathbf{p}$. If $\mathbf{p} \neq \mathbf{o}$, which means that the data vector has not yet arrived at its destination, attach the data vector to the $OP_{\mathbf{p}}^{w_{j'}}$, where $w_{j'} = w_j + 1 \ mod \ c$. Attach the vector to the corresponding IP of $OP_{\mathbf{p}}^{w_{j'}}$.

The actual algorithms for generating the in/out packets for general cases are provided in Appendix E, (Algorithm E.3).

It is helpful to show the procedure with a 2-D example. See Figure 7.5.(a). First, for each w_i, all the Q's which will be output should be collected into OB's which are also attributed with w_i. That is,

$$OB^{w_0} = Q^{-1,0} + Q^{-1,1} + Q^{0,1} \qquad OB^{w_1} = Q^{-1,1} + Q^{0,1}$$
$$OB^{w_2} = Q^{-1,1} + Q^{0,1} + Q^{1,-1} + Q^{1,0} \qquad OB^{w_3} = Q^{-1,1} + Q^{-1,0}$$
$$OB^{w_5} = Q^{1,-1} + Q^{1,0} \qquad OB^{w_6} = Q^{-1,1} + Q^{-1,0} + Q^{1,-1} + Q^{0,-1}$$
$$OB^{w_7} = Q^{1,-1} + Q^{0,-1} \qquad OB^{w_8} = Q^{0,-1} + Q^{1,-1} + Q^{1,0}$$

For example, the right-bottom box marked w_0 in Figure 7.5.(a) contributes to $\mathbf{d}^T_{-1,0}$, $\mathbf{d}^T_{-1,1}$ and $\mathbf{d}^T_{0,1}$, so the associated data blocks are collected to OB^{w_0} and placed one after another.

Next, we establish the initial OP's. The Q's in OP are distributed to OP's in suitable directions :

$OP^{w_0}_{-1,0} = \{[OB, 0, Q^{-1,0} + Q^{-1,1}]\}$, $OP^{w_0}_{0,1} = \{[OB, n(Q^{-1,0} + Q^{-1,1}), Q^{0,1}]\}$.
$OP^{w_1}_{0,1} = \{[OB, 0, Q^{-1,1} + Q^{0,1}]\}$,
$OP^{w_2}_{0,1} = \{[OB, 0, Q^{-1,1} + Q^{0,1}]\}$, $OP^{w_2}_{1,0} = \{[OB, n(Q^{-1,1} + Q^{0,1}), Q^{1,-1} + Q^{1,0}]\}$.
$OP^{w_3}_{-1,0} = \{[OB, 0, Q^{-1,1} + Q^{-1,0}]\}$.
$OP^{w_5}_{1,0} = \{[OB, 0, Q^{1,-1} + Q^{1,0}]\}$.
$OP^{w_6}_{-1,0} = \{[OB, 0, Q^{-1,1} + Q^{-1,0}]\}$, $OP^{w_6}_{0,-1} = \{[OB, n(Q^{-1,1} + Q^{-1,0}), Q^{1,-1} + Q^{0,-1}]\}$.
$OP^{w_7}_{1,0} = \{[OB, 0, Q^{1,-1} + Q^{0,-1}]\}$.
$OP^{w_8}_{0,-1} = \{[OB, 0, Q^{0,-1}]\}$, $OP^{w_8}_{1,0} = \{[OB, n(Q^{0,-1}), Q^{1,-1} + Q^{1,0}]\}$.

Then, we have to figure out all the corresponding IP's which receive the OP's. It can be understood that a $OP^{w_i}_{\mathbf{p}}$ is linked to $IP^{w_{i'}}_{-\mathbf{p}}$, where $i' = i + \overline{\Delta \mathbf{wp}} \bmod c$. For instance, $OP^{w_0}_{-1,0}$ corresponds to $IP^{w_3}_{1,0}$, and $OP^{w_0}_{0,1}$, to $IP^{w_0}_{0,-1}$, see Figure 7.5.(b). $IP^{w_i}_{\mathbf{p}}$ receives a number of Q's. If the Q arrives at its terminal, put the Q into IB, otherwise, put it into the relay buffer $RB_{\mathbf{p}}$. We can list all the IP's in a order of corresponding to OP's.

$IP^{w_3}_{1,0} = \{[IB^{w_3}, 0, Q^{-1,0}], [RB_{1,0}, 0, Q^{-1,1}]\}$, $IP^{w_0}_{0,-1} = \{[IB^{w_0}, 0, Q^{0,1}]\}$,
$IP^{w_1}_{0,-1} = \{[IB^{w_1}, 0, Q^{-1,1} + Q^{0,1}]\}$,
$IP^{w_2}_{0,-1} = \{[IB^{w_2}, 0, Q^{-1,1} + Q^{0,1}]\}$, $IP^{w_8}_{-1,0} = \{[IB^{w_8}, 0, Q^{1,-1} + Q^{1,0}]\}$.
$IP^{w_6}_{1,0} = \{[IB^{w_6}, 0, Q^{-1,1} + Q^{-1,0}]\}$.
$IP^{w_2}_{-1,0} = \{[IB^{w_2}, n(Q^{-1,1} + Q^{0,1}), Q^{1,-1} + Q^{1,0}]\}$.
$IP^{w_0}_{1,0} = \{[IB^{w_0}, n(Q^{0,1}), Q^{-1,1} + Q^{-1,0}]\}$, $IP^{w_6}_{0,1} = \{[IB^{w_6}, n(Q^{-1,1} + Q^{-1,0}),$
$Q^{1,-1} + Q^{0,-1}]\}$.
$IP^{w_4}_{-1,0} = \{[IB^{w_4}, 0, Q^{1,-1} + Q^{0,-1}]\}$.
$IP^{w_8}_{0,-0} = \{[IB^{w_8}, n(Q^{1,-1} + Q^{1,0}), Q^{0,-1}]\}$, $IP^{w_5}_{-1,0} = \{[RB_{-1,0}, 0, Q^{1,-1}], [IB^{w_5}, 0, Q^{1,0}]\}$.

Now the OP's and IP's must be modified to implement data relay. Obviously, all data in the RB's must be output once more, so we have to modify (or create) some OP's to relay this data. Since $IP_{1,0}^{w_3}$ puts $Q^{-1,1}$ into $RB_{1,0}$, $Q^{-1,1}$ must be output along the direction [0, 1] at the next time step, where the superscript vector of Q indicates $\mathbf{d}^A$. So we create

$OP_{0,1}^{w_4} = \{[RB_{1,0}, 0, Q^{-1,1}]\}$ and $IP_{0,-1}^{w_4} = \{[IB^{w_4}, n(Q^{1,-1} + Q^{0,-1}), Q^{-1,1}]\}$

to transmit $Q^{-1,1}$ and then to receive it, respectively. Similarly, create

$OP_{0,-1}^{w_6} = \{[RB_{-1,0}, 0, Q^{1,-1}]\}$ and $IP_{0,1}^{w_6} = \{[IB^{w_6}, n(Q^{-1,1}+Q^{-1,0}+Q^{1,-1}+Q^{0,-1}), Q^{1,-1}]\}$

to transmit $Q^{1,-1}$ and then to receive it.

Finally, we derive and list all the IB's.

$$IB^{w_0} = Q^{0,1} + Q^{-1,1} + Q^{-1,0} \qquad IB^{w_1} = Q^{-1,1} + Q^{0,1}$$
$$IB^{w_2} = Q^{-1,1} + Q^{0,1} + Q^{1,-1} + Q^{1,0} \qquad IB^{w_3} = Q^{-1,0}$$
$$IB^{w_4} = Q^{1,-1} + Q^{0,-1} + Q^{-1,1} \qquad IB^{w_5} = Q^{1,0}$$
$$IB^{w_6} = Q^{-1,1} + Q^{-1,0} + Q^{1,-1} + Q^{0,-1} + Q^{1,-1} \qquad IB^{w_8} = Q^{1,-1} + Q^{1,0} + Q^{0,-1}$$

Obviously, the IP's should be rearranged according to their ordinal number. It takes too much space and is of no real value to list all the OP's and IP's created, so they are omitted here.

All the work above is carried out during compilation and all the OP's and IP's are written into a h.file. They are are used repeatedly during the parallel program execution. In addition, we must use $w_{\overline{\Delta \mathbf{wa}} \bmod c}$ to determine the start point of the cycle, for each processor **a**. All the input and output operations must be controlled by the IP^{w_i} and OP^{w_i}, respectively, and all of copying data between supernode memory and OB and IB must be controlled by OB^{w_i} and IB^{w_i}.

7.4.2 Non LSGP Case and Direct Data Flows

Non LSGP Case

The non-LSGP case can be taken as the case $c = k = 1$ of LSGP. The method of creating IP and OP for LSGP case can be applied here and the superscript w_j for IP and OP is no longer necessary. For a 2-D SUC array, the OP's and IP's are given as:

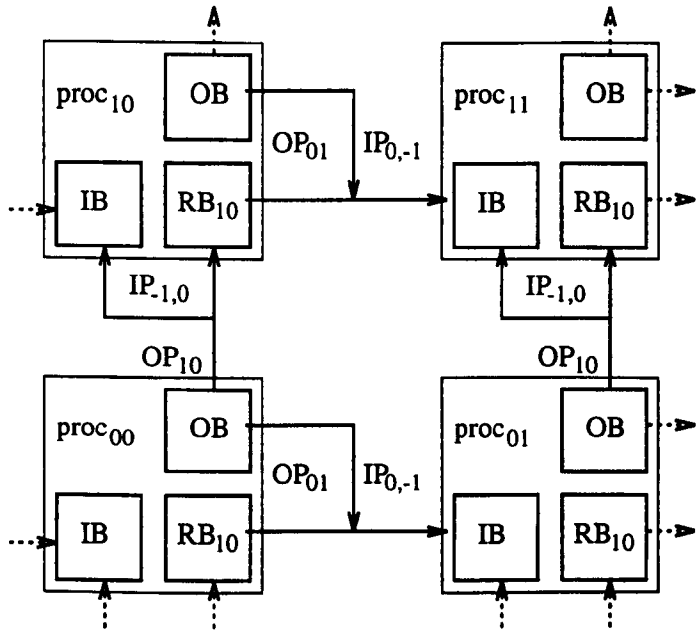


Figure 7.6: Data flows and relays.

$OP_{10} = \{[OB, 0, Q^{10} + Q^{11}]\}$.

$IP_{-1,0} = \{[IB, 0, Q^{10}], [RB_{10}, 0, Q^{11}]\}$.

$OP_{01} = \{[OB, n(Q^{10} + Q^{11}), Q^{01}], [RB_{10}, 0, Q^{11}]\}$.

$IP_{0,-1} = \{[IB, n(Q^{10}), Q^{01} + Q^{11}]\}$.

and

$OB = Q^{10} + Q^{11} + Q^{01}$.

$IB = Q^{10} + Q^{01} + Q^{11}$

This is easily checked from Figure 7.6. For instance, we first build OP_{10} for $proc_{00}$ along the dimension 0, which contains all the Q's which have the processor dependencies like "1x", i.e., $Q^{10} + Q^{11}$, so that $OP_{10} = \{[OB, 0, Q^{10} + Q^{11}]\}$. Then, we build $IP_{-1,0}$ which receives OP_{10} as a whole but separates it to two data vectors, i.e., Q^{10} to IB and Q^{11} to RB_{10}, thus $IP_{-1,0} = \{[IB, 0, Q^{10}], [RB_{10}, 0, Q^{11}]\}$.

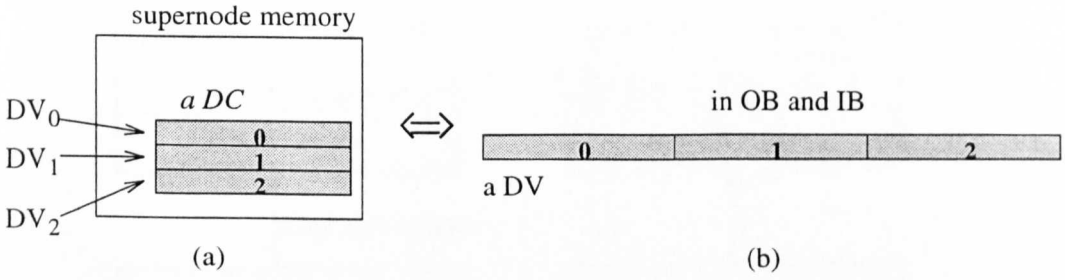


Figure 7.7: Indirect and direct transference of data.

Direct Data Flows

In the above discussions, we use OB and IB to buffer all the data flows in and out of the supernode memory space. The advantage of buffering is to collect a segment-distributed data block together so that it can be transferred as one Meiko data flow vector, i.e., the DV in Figure 7.7.(b).

However, it also has an obvious disadvantage: copying data between the supernode memory and the OB and IB takes time, so in some circumstances, we will try to make a direct data flow, instead.

We find that if the length of all of the segments is long enough, say more than 20 data items, the time for Meiko to open a new data vector is less than the time consumed to copy data to and from the buffers. Therefore, if the length of every segment is more than 20, we establish data vectors addressing on each of the segments. Thus, three DV's are created for the three data segments of Figure 7.7.(a). In the rest of this subsection, we deal with individual DC, instead of Q.

It is complicated to implement the direct data flow. Generally, for directly transferring a DC, DC^d, of size $\prod_{i=0}^{N-1} l_i$, we have to generate $n^d = \prod_{i=0}^{N-2} l_i$ DV's, noted as $DV_0^d, \cdots, DV_{n^d-1}^d$. All of them share the first point of DC^d as their formal buffer. The relative distance between the first point of their corresponding segments and the first point of DC^d are defined as their "position" in the formal buffer. Obviously, we have to make two sets of such DV^d's, one for output and the other for input.

In practice, we first generate the OP's and IP's by means of the methods outlined in the subsection above, then create the $DV_0^d, \cdots, DV_{n^d-1}^d$ to substitute for a DV of the OP's and IP's which references OB or IB and has a length l_{N-1}. This method is summarized

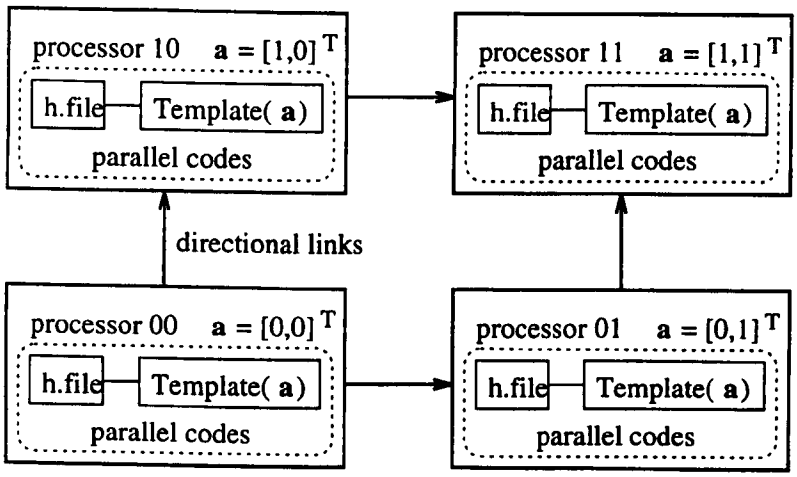


Figure 7.8: Processor Array and Parallel Codes. The parallel codes stand also for a process. All the parallel codes are identical with a as the parameter

as follows.

Step 1 Scan all the DC's, find the ones, say DC^d, whose $l_{N-1} \geq 20$;

Step 2 Remove DC^d from OB;

Step 3 Create n^d DV's for DC^d;

Step 4 Scan all DV's of OP's and modify the DV which transfers DC^d from OB with the n^d DV's. For instance, for $DV = [OB, 0, X + DC^d + Y]$, we have to change to $2 + n^d$ sequential DV's, i.e., $[OB, 0, X], DV_0^d, \cdots, DV_{n^d-1}^d, [OB, n(X), Y]$, where X and Y are arbitrary groups of DC's;

Step 5 Scan all DV's of OP and modify the DV's which do not transfer DC^d but involve DC^d in their "position". For instance, $DV = [OB, n(X + DC^d + Y), Z]$ should be changed to $DV = [OB, n(X) + n(Y), Z]$.

Step 6 Do the same for IB and IP's.

In the parallel program, these DV^d's will be given physical addresses that reference supernodes.

7.5 Outline of the Parallel Code

As mentioned before, we shall automatically produce one parallel code which will be loaded to all the processors and run synchronously. The parallel code behaves like a process. Every processor has a process. For a processor, the process running in it is also identified by the **a** of the processor.

Automatically producing the parallel code does not mean that we have to produce a completely new-program every time. Our idea is to have some program templates in advance which construct a structure and contain some unchangeable segments of the program. In the structure, operations which change for particular applications are formally expressed by macro-defined functions. It is these macro-defined functions that have to be produced automatically and be installed into a head file for each application. There are many parameters which describe the particular problem and are used by both the constant program segments and the macro-defined functions. These parameters are also collected into the head file. Figure 7.8 shows the assignment of parallel codes onto a processor array. The program template constructs the main structure of the parallel program. With **a** as the parameter which determines the supernodes involved in a particular processor, it opens communication channels connecting to neighbouring processors, allocates memory space for each of the supernodes involved and initialises them. It also computes the supernode and input/output data from and to other processor via the channels. See Figure 7.9 for its conceptual operation.

There are two kinds of program templates. One is ProcNonLSGP, which copes with the situations of Lower-Dimensional array (Lower-D) and SUC using IP's and OP's. The second, ProcLSGP, is only for SBC with LSGP partitioning and uses IP^{w_j}'s and OP^{w_j}'s. Examples of the real parallel codes are presented in Appendix G, but are not very readable for the uninitiated. We explain them as follows.

7.5.1 Parallel Codes for Non-LSGP

Codes 7.5.1 *The ProcNonLSGP*

```
Open N_links TRANSPORTs for communications          } op 1 :
Create N_links channels to neighbouring TRANSPORTs  } Open Communications
Allocate and clear memory space for all data.       }
Allocate IB's and OB's and RB's                     } op 2 :
Preparations for accessing supernode space          } Preparation
Locate DV's in IP's and OP's                        }

SupernodeSpaceLoops                                 }
{                                                   }
   IF Condition true                                } op 3 :
      Locate the supernode in the memory space      } Locate and Initialise
      InitializeData(a supernode)                   }
   Break                                            }
}                                                   }

SupernodeSpaceLoops                                 op 4 :
{
   FOR i :=0 TO N_DC - 1                            }
      IF be a direct DC                             } op 4.1
         CreateDirectInputDVs(DC i)                 } build direct DV's
         CreateDirectOutputDVs(DC i)                }
   FOR i :=0 TO N_links - 1                         }
      Receiving input data flow blocks according to IP's      } op 4.2
      Transferring output data flow blocks according to IP's  } Input/output
   FOR i :=0 TO N_links - 1                         }
      IF the TRANSPORT is input-valid               } op 4.3
         Wait until all input data received         } Waiting until input finish
   FOR i :=0 TO N_DC - 1                            }
      IF be a indirect DC                           } op 4.4
         CopyFromBufferToSupernode(DC i)            } Data from
                                                    } buffer to memory
   IF Condition true                                } op 4.5
      ComputeSupernode(a supernode)                 } Compute supernode
   FOR i :=0 TO N_links - 1                         }
      IF the TRANSPORT is output-valid              } op 4.6
         Wait until all output data transferred     } Waiting until output finish
   FOR i :=0 TO N_DC - 1                            }
      IF be a indirect DC                           } op 4.7
         CopyFromSupernodeToBuffer(DC i)            } Data from memory
                                                    } to buffer
      Break
}
```

End of Codes

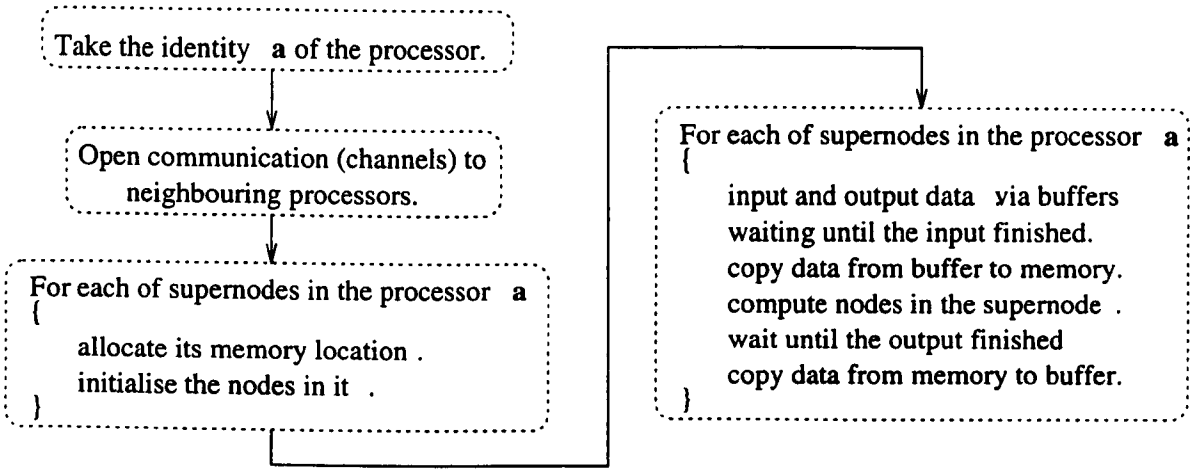


Figure 7.9: The flow Chart of the template of a Parallel Codes.

where **InitializeData**, **ComputeSupernode**, **CreateDirectInputDVs**, **CreateDirectOutputDVs**, **CopyFromBufferToSupernode** and **CopyFromSupernodeToBuffer** are macro-defined functions; **SupernodeSpaceLoops**{ } are macro-defined FOR LOOP's similar to the N-M-1 FOR-LOOP's of FOR-Loops 6.4.2, which confines the supernodes mapped into this processor; **Condition** and **Break** are macro-defined expressions. **Condition** stands for referencing a valid supernode. N_{links} stands for the number of the links around a processor. In fact, N_{links} is also the number of non-zero entries of **P**.

Brief explanations and comments are given as following.

1. In *op 1*, TRANSPORT is the port of a process for input and output data. See Figure 7.10. N_{links} TRANSPORTs should be established for a process, $TRANSPORT_{i,j}$,

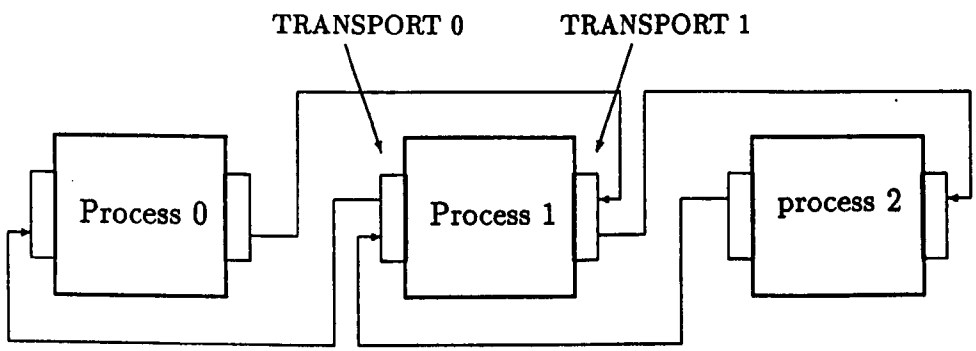


Figure 7.10: TRANSPORT and Communication channels in the case of 1-D and SBC

where $i = 0, \cdots, M-1$ and j = 1 in SUC or j = ± 1 in SBC. Inter-process communication channels must be created between the TRANSPORT's inside and outside the process.

We restrict ourselves strictly to the cases where we have to do all necessary data relay by ourselves. Therefore, the inter-process communication channels coincide exactly with the physical links, since there is one and only one process in a processor in our cases. Thus, the procedure of creating the inter-process channels is similar to that of inter-processor links, i.e., Algorithm 7.2.1, but only for the single **a**, the badge of the process.

In addition, we should make a record of the available channels for each process, because for the processors on edges of the array, to try to send data via a non-existent channel will make the process wait endlessly for response. This is achieved by marking TRANSPORT input and/or output valid.

Algorithm 7.5.1 *Build Inter-processor Communication Channels*

For all $\mathbf{p}_j \in \mathbf{P}$
  For the $p_{i,j} \in \mathbf{p}_j$ and $\neq 0$
    $\mathbf{a}' := \mathbf{a} + \mathbf{p}_j$
    IF $\mathbf{a}' \in A^M$
      Create a channel from $TRANSPORT_{i,p_{i,j}}$ of **a** to $TRANSPORT_{i,p_{i,j}}$ of $\mathbf{a}'$
      Mark $TRANSPORT_{i,p_{i,j}}$ output valid
    $\mathbf{a}' := \mathbf{a} - \mathbf{p}_j$
    IF $\mathbf{a}' \in A^M$
      Mark $TRANSPORT_{i,p_{i,j}}$ input valid

2. Before doing any data transference, in *op 2* we should give the DV's in IP's and OP's physical addresses referencing to the allocated input and output buffers. Since data transferences start and end in fixed buffers independent of supernodes, it can be done independently.

3. In *op 2*, a memory space defined by eqn (7.3) is allocated for data of all supernodes which may be involved in the process. For *op 3*, where **SupernodeSpaceLoops**{ } with **Condition** together confine and access all the supernode assigned in this process and locate each supernode to the address a^t defined by eqn (7.4) in this memory space. **InitializeData** generates an initial data matrix by the process itself. If initial data are obtained from outside it is not required.

4. The second **SupernodeSpaceLoops**{ }, i.e., *op 4*, is the real body of communications and computations.

Communication through the N_{links} TRANSPORTs: in *op 4.2* for each supernode, if the TRANSPORT is input-valid, data is transferred inwards to IB and RB by using the IP's; if the TRANSPORT is output-valid, data is transferred outwards from OB and RB by using the OP's.

Internal Data Transfer (buffer $\Rightarrow$ supernode): if the tests in *op 4.3* shows that the reception is finished, then in *op 4.4* the data which have arrived at their destination is transferred from an IB to the memory location of the relevant supernode. More accurately, at time t, $\forall\, i \in [0, N_{DC}-1]$, **CopyFromBufferToSupernode** fetch $DC_{\mathbf{d}_i^t}$ from the IB and store it to an area indicated by $a^t + r_i^i$, where r_i^i is defined by eqn (7.1).

Computation: in *op 4.5* if the test **Condition** shows that the supernode is within the valid supernode polyhedron, the supernode is computed.

Internal Data Transfer (supernode $\Rightarrow$ buffer): if the test in *op 4.6* shows that the previous output transference has finished, the output data will be collected in *op 4.7* from the memory location of the relevant supernode to an OB. More accurately, at time t, $\forall\, i \in [0, N_{DC}-1]$, **CopyFromSupernodeToBuffer** copies the DC_i of the area indicated by $a^{t-d_i^t} + r_i^o$ to the OB, where r_i^o is defined by eqn (7.2) and d_i^t is the time-delay attribute of $DC_{\mathbf{d}_i^t}$.

5. The data movements between the memory locations of supernodes and the IB and OB waste time. Obviously, we hope for data flow directly from or to supernodes. In some cases, this is beneficial, but not always. For those DCs which may be directly transferred, *op 4.1* have to give the DV^d's in IP's and OP's physical addresses referencing supernodes. Unlike other DV's which address only at the fixed OB or IB, these DV^d's have to be located dynamically by **CreateDirectInputDVs/CreateDirectOutputDVs** inside *op 4* because they are associated with individual supernodes.

7.5.2 Parallel Codes with LSGP

The algorithm of ProcLSGP is similar to ProcNonLSGP but requires some modifications. For completeness, we give it as following:

Codes 7.5.2 *The ProcLSGP*

```
Open N_links TRANSPORTs for communications                    } op 1 :
Create N_links channels to neighbouring TRANSPORTs            } Open Communications
Allocate and clear memory space for all data.                 }
Allocate OB's and IB's and RB's                               } op 2 :
Preparation for accessing supernode space                     } preparations
Locate in/out DV's in OP^{w_j} and IP^{w_j}                   }
Determine initial w_j for the processor                       } op 3

FOR t_{N-1} := t^l TO t^u                                     op 4
{
    Derivet                                                   } op 4.1
    Locate the supernode t in the memory space                } op 4.2 :
    InitializeData(supernode t)                               } Locate and Initialise
}

FOR t_{N-1} := t^l TO t^u                                     op 5
{
    Derivet                                                   } op 5.1
    Select IP^{w_j} and OP^{w_j}                              } op 5.2
    ComputeSupernode(supernode t)                             } op 5.3
    FOR i :=0 TO N_DC - 1                                     }
      IF the DC is indeed outputted according to OP^{w_j}     } op 5.4 :
        CopyFromSupernodeToBuffer(DC i)                       } Data from memory
                                                              } to buffer
    FOR i :=0 TO N_links - 1                                  }
      Receiving input data flow blocks according to IP^{w_j}  } op 5.5 :
    FOR i :=0 TO N_links - 1                                  } Input/Output
      Transferring output data flow blocks according to OP^{w_j}
    FOR i :=0 TO N_links - 1                                  }
      IF the TRANSPORT is input-valid                         }
        Wait until all input data received                    }        op 5.6 :
    FOR i :=0 TO N_links - 1                                  } Waiting input/output finish
      IF the TRANSPORT is output-valid                        }
        Wait until all output data transferred                }
    FOR i :=0 TO N_DC - 1                                     }
      IF the DC is indeed input according to IP^{w_j}         } op 5.7 :
        CopyFromBufferToSupernode(DC i)                       } Data from
                                                              } buffer to memory
    j := j + 1 mod c
}
```

End of Code

A brief explanation is given as follows:

1. Recalling FOR-Loops 6.5.1, instead of **SupernodeSpaceLoops**, in *op 4* and *op 5* we need only one loop with respect to t_{N-1} to access the supernodes assigned into the processor.

2. Derive **t**, *op 4.1* and *op 5.1*, is a function for deriving the time basis **t** by means of Algorithm 6.5.1.

3. Now let us consider *op 3*, *op 5.2* and *op 5.5* which are related to input and output packets. The c pairs of IP^{w_j}'s and OP^{w_j}'s derived in 7.4.1 are used repeatedly in *op 5* of Codes 7.5.2. When running the parallel program, for each process, *op 3* determines the initial w_j. After entering the loop of *op 5*, *op 5.2* selects the pair of IP^{w_j} and OP^{w_j}. In *op 5.5* every Receiving and Transferring must be carried out according to the pair of IP^{w_j} and OP^{w_j}. The data copying between Supernode memory space and in/out buffers, *op 5.4* and *op 5.7*, must be carried out according to IB^{w_j} and OB^{w_j}. For the next supernode, turn to $IP^{w_{j+1}}$ and $OP^{w_{j+1}}$, and so on.

7.6 Summary

As the result of the theoretical work, actual parallel codes in Meiko C are automatically generated. The parallel codes consist of two pieces. One is a designed to include or omit LSGP partitioning, and contains the main structure of the codes. The second part is a head file which collects all parameters and some specific functions. In this way, we may reduce the compilation work to minimum. To achieve this, some special practical problems for the generation of parallel codes are resolved. See figure 7.11.

The methods of code generation for the various situations differ slightly. We cannot say which is better or worse, since only the particular situation in an application decides which one will be used. Obviously, one factor is the type of processor-array we are given, and the interconnection primitives. In addition, sometimes, even having a SBC array, we may actually adopt SUC mesh. This is because SUC invokes about half the communication interfaces as SBC.In some cases, invoking communication interfaces is very costly. Furthermore, we should choose the method where the computational polyhedron is mapped to processor array with best fitness. In many cases, the last factor may play a key role.

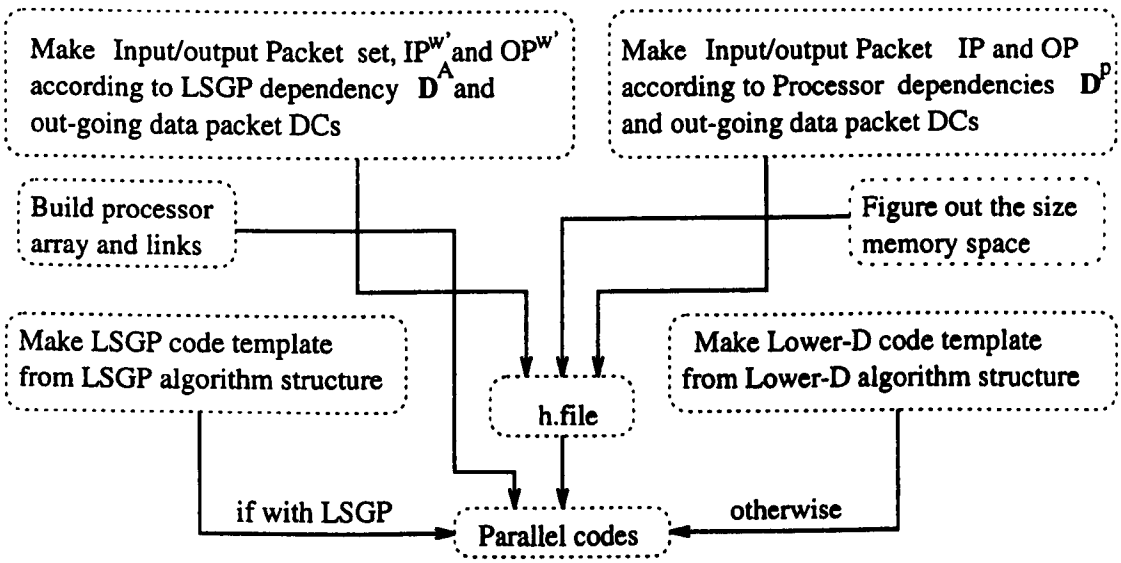


Figure 7.11: Chart of Parallel Code Generation

Chapter 8

Experimental Results and Discussions

The theory and the generated parallel codes presented in previous chapters have been tested by a great number of experiments. The experiments are designed to check the correctness of the parallel program first, then the performances in various situations. The result shows they work correctly and efficiently. Some experimental results are presented in Section 8.1. Discussions are presented for the main factors which affect the performance. Among them, we find the fitness of mapping an irregular problem onto a regular time-space domain is the dominant one.

8.1 Experimental Results

In this section, we provide some experimental results to show the correctness and performance of our methodologies.

8.1.1 Test of the Correctness

Example 6.1.1 is used to check the correctness of our work, that is, whether the automatically generated parallel programs can give the exact same results as their sequential counterpart. A sequential program is written to carry out the Loops 6.1.1. Four kinds of parallel programs are automatically generated, one for each of the four cases given below, there are a 1-D array with 4 processors and SUC mesh, 1-D array with 4 processors and SBC mesh, 2-D array with 4×4 processors and SUC mesh, and a 2-D array with 4×4

processors and SBC mesh. Since Example 6.1.1 is a recursive process, array $A(i_0, i_1, i_2)$ should be given initial values. We suppose that $A(i_0, i_1, i_2) = i_0 + i_1 + i_2$ initially.

Because of the dependencies in Example 6.1.1 and since element $A(20, 20, 10)$ is the last element to be modified, the correct value of $A(20, 20, 10)$ requires the correctness of all other modified elements of $A(i_0, i_1, i_2)$. Therefore, it is sufficient to check only element $A(20, 20, 10)$. We run the five programs, and all of them give the result $A(20, 20, 10) = 708639144$.

Meanwhile, we find that any minimal faults in these parallel programs will result in a totally different $A(20, 20, 10)$. In addition, we also check the correctness in other ways for all the parallel programs. For example, the total numbers of computed nodes are counted, which is 4851 for Example 6.1.1. The correctness of data communication is also checked. It has been checked for all nodes by an additional subroutine that when computing a node, checks all the required data are already in their correct locations to be fetched. These two checks guarantee further the correctness of computation. These facts show that all the automatically generated parallel programs work correctly.

Furthermore, a great number of experiments carried out in the following subsection are also used to establish the correctness of our method, although they are designed for testing the performances. For each of them, we have a sequential program and a parallel program both doing the same computation which produce a pair of results of the last executing node. Comparing each of the pairs of results produced by the sequential and parallel programs, we find that none of the pairs is different!

In theory it is occasionally possible that more than one error exactly compensate each other so that the last answer is correct. However, in the large amount of our experiments, we find no error occurs at all, therefore their correctness cannot be concluded as coincidence. In fact in our case, because the computation is a linear accumulating procedure, any functional fault in the program will make errors become bigger and bigger so that errors hardly compensate each other, let alone exactly.

8.1.2 Measuring Performance

The performance of the mapping method is shown with experimental results. Before giving the experimental results, we should describe the experimental set-up and the major performance factors we are concerned with, Speed-up (S) and Efficiency (E).

The experimental results are given for three kinds of processor arrays, 1-D array with SUC mesh and variable number of processors, which is designed to show the performance of linear arrays which are the most widely-used arrays and the effects of changing the size of an array; 2-D array with 4×4 processors and SUC mesh and 2-D array with 4×4 processors and SBC mesh, which is for showing the performance of multi-dimensional array with or without LSGP partitioning since the use of LSGP partitioning is an important aspect of our method. We do not intentionally change the shape of the 2-D array since the 1-D array is an extreme example of 2-D arrays and we like to make a full use of the number of processors in the parallel computing resource. Also we do not use high dimensional structures because the lower dimensional mappings are designed to alleviate problems of constructing higher dimensional array structures.

Experimental situations

Similar to Example 6.1.1, we still use 3-D problems

$$
\begin{array}{l}
\text{FOR } i_0^c := 0 \text{ TO } a \\
\quad \text{FOR } i_1^c := 0 \text{ TO } b \\
\quad\quad \text{FOR } i_2^c := 0 \text{ TO } c \\
\quad\quad\quad A_{\mathbf{i}} := A(\mathbf{i} - \mathbf{d}_0) + A(\mathbf{i} - \mathbf{d}_1) + A(\mathbf{i} - \mathbf{d}_2) + A(\mathbf{i} - \mathbf{d}_3)
\end{array}
$$

as a basic experimental model, where $A(\mathbf{i})$ stands for $A(i_0, i_1, i_2)$. The dependency matrix $\mathbf{D} = [\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3]$ can be modified in our experiments. The basic computational polyhedra are cubes of size $a \times b \times c$. Let $\mathbf{c} = [a, b, c]^T$ be an N-D constant vector describing the size of the polyhedra.

The actual computational problems are constructed by skewing the polyhedron of the basic one with a linear transformation $\mathbf{K}$, where $\mathbf{K}$ is a unimodular lower-triangular matrix. That is, at first, construct cube-shaped computational polyhedron, indexed by

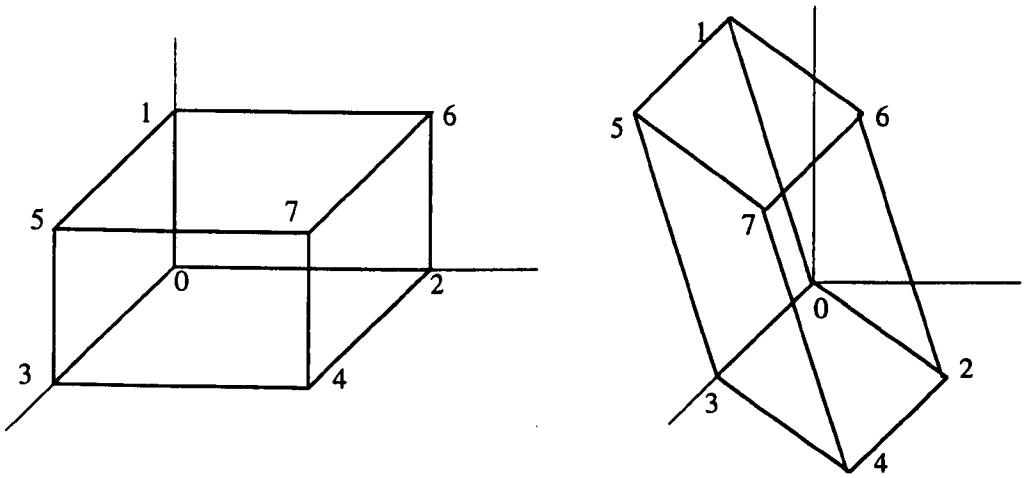


Figure 8.1: The basic cubic polyhedron and the transformed actual polyhedron

$\mathbf{i}^c$, then do transformation $\mathbf{i} = \mathbf{K}\mathbf{i}^c$ as follows

$$\begin{bmatrix} -\mathbf{I} \\ \mathbf{I} \end{bmatrix} \mathbf{i}^c \leq \begin{bmatrix} \mathbf{0} \\ \mathbf{c}-\mathbf{1} \end{bmatrix} \Longrightarrow \begin{bmatrix} -\mathbf{K}^{-1} \\ \mathbf{K}^{-1} \end{bmatrix} \mathbf{i} \leq \begin{bmatrix} \mathbf{0} \\ \mathbf{c}-\mathbf{1} \end{bmatrix}$$

where the left and the right systems of inequalities are the cubic polyhedron and the actual polyhedron, respectively. Because the transformation is bijective, the transformed polyhedron contains also $N_n = a \times b \times c$ nodes.

For example, let $\mathbf{c} = 30 \times 40 \times 50$, and

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 1 & 1 \end{bmatrix} \quad \mathbf{K}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -1 & 1 \end{bmatrix}$$

This computational graph stands for a For-Loop

FOR $i_0 := 0$ TO 29
  FOR $i_1 := -2i_0$ TO $-2i_0 + 39$
    FOR $i_2 := -i_0 + i_1$ TO $-i_0 + i_1 + 49$
      $A(i_0, i_1, i_2) := A(i_0 - 1, i_1 + 1, i_2 + 3) + A(i_0, i_1 - 1, i_2 - 1)$
        $+ A(i_0, i_1 - 1, i_2 + 1) + A(i_0, i_1 - 1, i_2)$

The cubic polyhedron and the transformed actual polyhedron are illustrated in Figure 8.1. The statement of computation in the loops has also the form of In summary, the experimental situations can be specified by **c**, **K** and **D**.

In the following experiments, we design a few situations to test. The experiments are divided into three sets, one is for lower-dimensional array (linear array here), the second

is for (N-1)-D array with SUC mesh and the third for (N-1)-D array with SBC mesh (2-D here). In each case, we test the effects of changes in the skew of the polyhedron, the dependencies, and the sizes of the computational polyhedron, (in individual dimensions or all dimensions together). For lower-D cases we also test changes in the number of processors. These tests are not exhaustive, but we claim that they give us a general idea of how our methodology behaves and they inspire later discussions to exploit factors which may affect performance.

Speedup and Efficiency

The speed-up is usually defined as $S = \frac{T_s}{T_p}$, where T_s and T_p stand for the times consumed by the sequential program and parallel program, respectively. However, in many cases, we cannot run an equivalent sequential program in one processor to measure T_s, since the processor does not provide enough memory space for large computational tasks. Alternatively, we define $T_s = t_n \times N_n$, where t_n is the time to compute a single node without any overhead operations. The t_n is obtained by measuring the time for computing a single large full supernode (the time for running **ComputeSupernode**() once) and then dividing the time with the number of nodes in the supernode. **ComputeSupernode**() is a typical set of nested loops with the recursive mechanism of computing bounds whose overheads is negligible when the supernode contains a large amount of nodes.

The efficiency is defined by $E = \frac{S}{N_p}$, where N_p is the number of processors in the array.

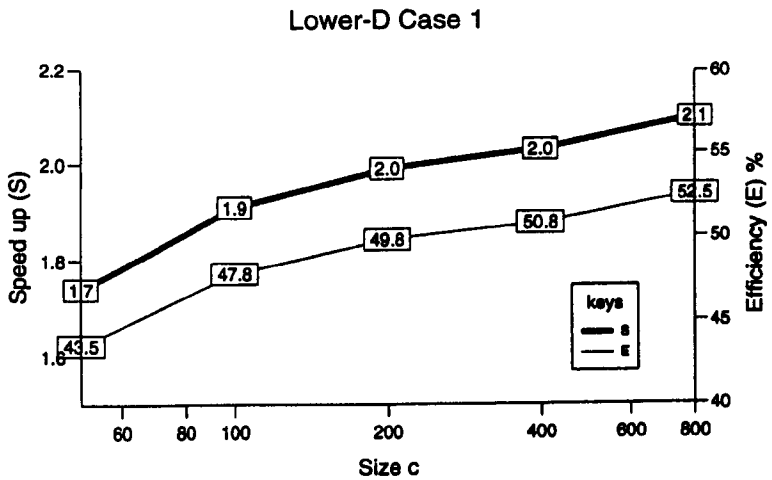
Before evaluating the speedup, t_n should be obtained. Note that t_n varies with the statements involved in iterations, independent of dependencies. In our experiments, t_n remains basically unchanged. We measure the time for computing a supernode containing 2000 nodes. Which takes 35776 μs. Thus $t_n = 17.8\mu s$. Some other situations have also been tested, results are almost the same and 17.8 is about the smallest value.

Only a few number of experimental results are presented here, while all the experimental results are collected into Appendix F.

A linear array consisting of N_p processors with SUC mesh is used in the following experiments

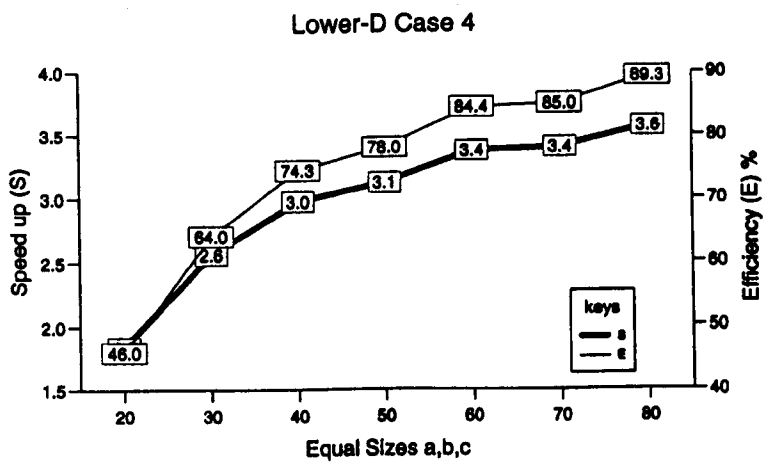
Case lower-D-1: $\mathbf{c} = \begin{bmatrix} 20 \\ 20 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 4$

c	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	204.3	372.6	715.4	1401.8	2714.5



Case lower-D-4: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 \end{bmatrix}$ $N_p = 4$

$a = b = c$	20	30	40	50	60	70	80
T_s(ms)	142.3	480.3	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	77.4	196.0	382.9	713.4	1138.7	1793.2	2553.6

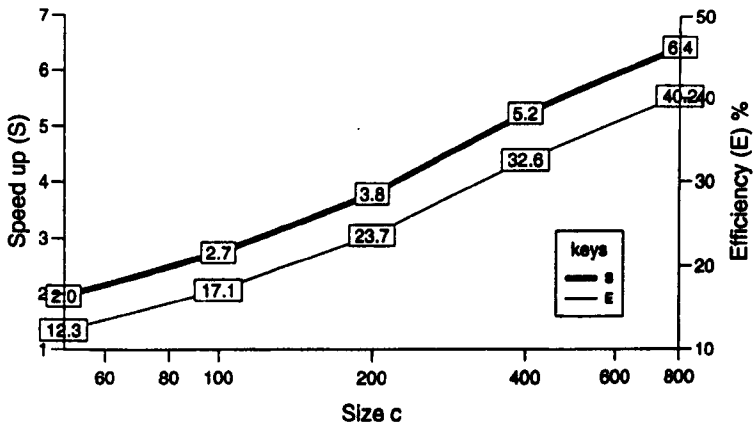


An 4×4 2-D array consisting of 16 processors with SUC mesh is used in the following experiment

$$\text{Case SUC-1: } \mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix} \quad N_p = 16$$

c	50	100	200	400	600	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	17078.4	22771.2
T_p(ms)	724.5	1038.3	1502.8	2182.2	2860.2	3540.6

2-D SUC Array Case 1

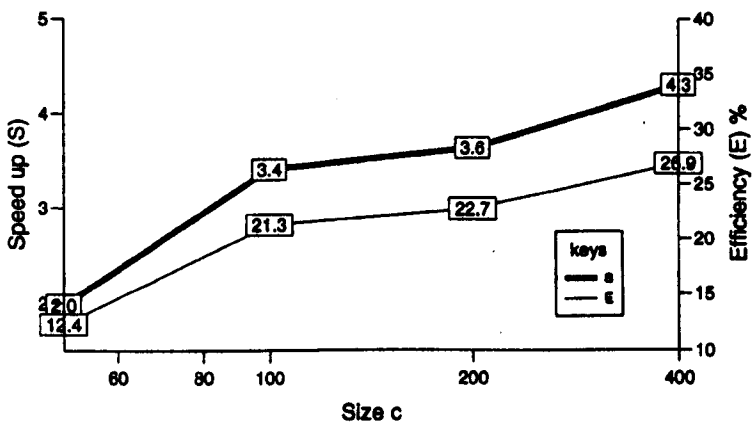


An 4×4 2-D array consisting of 16 processors with SBC mesh is used in the following experiment

Case SBC-1: **c**, **K** and **D** are the same as Case SUC-1.

c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	719.4	843.9	1574.8	2649.0

2-D SBC Array Case 1



8.2 Discussions on Factors Affecting Efficiency

We have to note a fact that the parallel algorithms are not necessarily faster than their sequential counterparts, due either to the nature of the computational problem or a poor arrangement of the parallel computation, parallel programs may run slower. In our experiments, the speedup and efficiencies vary widely, therefore it is quite worthwhile to investigate the main factors which affect the performance of our method.

8.2.1 Space and Time Fitness of the Mapping

The fitnesses of the space and time mappings are the major factors that affect the efficiency in many situations. The problem is discussed in detail as follows.

The poorly-filled supernodes

When the original computational polyhedron is clustered to a supernode, many supernodes around the boundary of the polyhedron may not be full. In some extreme cases, a supernode may consist of only a few valid nodes. But even if there is one valid node in a supernode, the supernode must be taken into account, and all operations must be performed. Even though the supernode off the boundary are all full, the number of poorly filled supernodes is significant, and the time wasted on them cannot be ignored.

Usually, the poorly-filled supernodes are difficult to avoid, for instance, the point e in Figure 6.1. Even if the facets of the original polyhedron are parallel to those of the partitioning parallelepiped, the sizes of the former are not integral multiplies of the latter, thus poorly filled supernodes appear. If the facets are not parallel to each other, the situation becomes worse.

Another factor is the size of the supernode. The larger the size, the poorer the fullness (if the supernode is small enough to contain one node, every supernode is full). This increases the difficulty of choosing a proper granularity. However, note that in our parallel programs the invalid nodes of a supernode are removed from the computation by imposing the boundary of the original polyhedron upon the boundary of supernodes (Subsection 6.2.3), but are still involved in communication for uniform communication mechanism.

As a result of our experiments, we find that the large granularity of supernodes does not result in serious shortcomings because the communication accounts for less in the large granularity situation. So we still prefer large granularity.

The fitness of the mapped supernode with the array

We have taken considerable care to scale the supernode domain such that it can be mapped within the given array by **S**. Unfortunately, because the array is regular, a good match of the mapped supernode with the regular array is not guaranteed in theory. Of course, this problem exists for all the existing partitioning and mapping methods as well.

In the 1-D linear array, every processor is assigned work to do if the partitioning is carried out properly. So usually, no processors are completely idle, but the loads on each processor may be unbalanced. For the 2-D cases, some processors around the corners of the array may be idle during the whole computing process. This is reflected clearly by the experimental results of Case SUC's (Case SUC-1 to Case SUC-16 of pages 234 to page 241) and Case SBC's (Case SBC-1 to Case SBC-14 of pages 242 to page 248) which are generally poorer than those of the 1-D arrays. As the dimension of the array increases, the situation deteriorates dramatically.

The main factors influencing the problem are the shape of the supernode domain, the transformation **B** and the space mapping matrix **S**. The load balance is the result of the combination of these factors. In our method, we consider the **B** and **S** separately, according to different requirements. So our method is guaranteed to succeed for any URE cases, but loses the freedom for space fitness.

However, it should be pointed that such freedom is usually quite limited because so many constraints and requirements are imposed upon the choices of **B** and **S**, that they do not always promise better fitness. Furthermore, despite any other considerations, we may still prefer a simple **S** as the $\mathbf{S}_{SUC}$ and $\mathbf{S}_{SBC}$, because a complex **S** will result in a more irregular mapping for reasonable polyhedra, which makes it more difficult to match a regular array.

In our experiments, it can be seen that $\mathbf{S}_{SUC}$ is preferable to $\mathbf{S}_{SBC}$. We think this mainly comes from the unfitness of $\mathbf{S}_{SBC}$ when mapping supernode domain onto pro-

cessor array. In fact, because the space mapping matrix $\mathbf{S}$, i.e., $\mathbf{S}_{SBC}$, has a form of $\begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$, a 45° rotation happens when mapping the supernode domain onto the processor array. If the supernode domain is somehow cube-shaped, the 45° rotation mapping results in a mismatch similar to a diamond within a square. However, we cannot conclude that such a rotation is always a bad thing. If the supernode domain itself is diamond-shaped, this rotation may result in a better match.

Time domain skew

There is a skew in the time domain. The nature of time skew comes from the data dependencies. In an array, a processor cannot work until the data from other processors is ready, so that, usually, all processors in an array cannot simultaneously start and finish. Because this delay effect can be accumulated, the final delay can become very significant, especially for large arrays, (e.g., long 1-D arrays). Sometimes this problem becomes the dominant factor affecting the efficiency because the whole executing time is counted from the start of the first processor to the finish of the last one.

For example, for the case of the Case-Lower-2 with 10 processors (see page 223), their running times are listed as:

$proc_0$	$proc_1$	$proc_2$	$proc_3$	$proc_4$
$0 \sim 42$	$5 \sim 47$	$10 \sim 52$	$15 \sim 57$	$20 \sim 62$
$proc_5$	$proc_6$	$proc_7$	$proc_8$	$proc_9$
$25 \sim 67$	$30 \sim 72$	$35 \sim 77$	$40 \sim 82$	$45 \sim 87$

The single delay is 5 for a single processor but is accumulated to 45 for the last processor, so the upper-bound of efficiency is only 48.8% = 43/88. The actual efficiency is 42.9%.

For the case of the Case-Lower-5 with 10 processors, the running times are:

$proc_0$	$proc_1$	$proc_2$	$proc_3$	$proc_4$
$0 \sim 79$	$1 \sim 80$	$2 \sim 81$	$3 \sim 82$	$4 \sim 83$
$proc_5$	$proc_6$	$proc_7$	$proc_8$	$proc_9$
$5 \sim 84$	$6 \sim 85$	$7 \sim 86$	$8 \sim 87$	$9 \sim 88$

The single delay is 1, so the upper-bound of efficiency is improved to 91%, while the actual efficiency is 76.7%.

The regularity of supernode domain

The shape of the supernode polyhedron is determined by applying the transformation **B** or **E** upon the original polyhedron. The **E** derived from **D** is necessary to make the whole of our methodology successful, but also has the side-effect of skewing the polyhedron. The skewing may make the supernode polyhedron more regular in some situations, such as Case lower-D-6, Case lower-D-7, $\cdots$, and Case SUC-5, but worse in other cases.

When determining the directions of the edges of the partitioning parallelepiped, we do not take the requirement of reducing the data communication as the only consideration . For instance, if there are N dependency vectors, they can be taken as the edges of the partitioning parallelepiped so that the data dependencies in supernode space lay only along coordinate axes without any relay. This is the best we can expect for data communications, but we may skew the computational polyhedron too much. Therefore, generally without detailed knowledge of the supernode polyhedron, we prefer a **E** which is as close to **I** as possible.

Comparing Case lower-D-1 and Case lower-D-3, they share the same rectangular computational polyhedron but with different dependencies. In Case lower-D-1, there are some negative entries in its dependencies, thus **E** is not an identity matrix. Therefore the supernode polyhedron is no longer rectangular, which gives a poor fitness of space and time mapping. In contrast, in Case lower-D-3, there are no negative entries in its dependencies, so **E** is an identity matrix and then its supernode polyhedron keeps rectangular and results in a good fitness of space and time mapping. As a result, they show the significant difference in their efficiencies.

If **D** consists of only positive entries, we do have $\mathbf{E} = \mathbf{I}$. Therefore, for some benchmark problems such as matrix product where **D** is non-negative and the original polyhedron is regular, a good match, and so high efficiency, can be expected. This case is also shown by Case lower-D-3, Case lower-D-4, $\cdots$, and Case SUC-2, Case SUC-7, where the shape of original polyhedron is regular. It should be noted that for the case of (N-1)-D array with SBC mesh, a similar situation does not bring about a good result, see Case SBC-2 and Case SBC-6, because the 45° rotation results in an poor match in the space mapping.

8.2.2 Data Communication

If the data has to be distributed too many places, the communications will be a major obstacle for actual applications. If the data dependency is uniform and relatively local, the cost for data communication is acceptable. For the case of UREs, some facts should be mentioned:

Communication separated from computation

Some parallel computing facilities, such as transputers, provide the ability of doing computation and communication simultaneously so that the communication of large scale data can be "hidden" behind computations. Unfortunately, this good feature can not be taken advantage of in our case. Consider $\overline{\mathbf{d}}^t = [\cdots, d_i^t, \cdots] = \overline{\mathbf{t}}\mathbf{D}^s$ which indicates the times to be offered to implement the corresponding supernode dependency. If $d_i^t = 1$, this means that the data produced at one time step is required just by the next time step. Thus the communication for the data must take place between the two computations, and so the occurs "explicitly". If $d_i^t = 2$, the data produced at one time step is required by the third time step, so we can arrange the data communication to take place simultaneously with the second computation, effectively "hiding it" behind the second computation. In our case, it is easy to see that majority of d_i^t's are "1". There are d_i^t's larger than 1, but the amount of corresponding outgoing data are relatively small. For example, Let $\overline{\mathbf{t}} = [1, 1]$ and $\mathbf{D}^s = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$. Thus, $\overline{\mathbf{d}}^t = [d_0^t, d_1^t, d_2^t,] = [1, 1, 2]$. The outgoing data associated with them are A_2, A_1, and A_3 in Figure 6.2, respectively. A_3 can flow simultaneously with a computation, but it may be too small to be worth doing, since it is an intersection of two areas.

Time for communication

The time for communication for a supernode consists of two parts. One is for the overhead operations, such as setting and invoking port interfaces, which depends on how many ports are involved and how many data vectors are to be transferred, (which is independent of the size of the supernode). The second part is for the actual data transference, (which is directly related to the size of the supernode). However, since the size of the

supernode increases, the amount of computations of a supernode increases faster than that of communication thus, increasing the size of the supernode means a reduction in the ratio of time of communication over that of computations.

It can be seen from the results of experiments that when the size of the computation problem is large, the affect of the communication becomes insignificant. For example, for the Case-Lower-4, the efficiency is up to 89.3%, and its upper-bound limited by time skew is 94%. Of course, if the size of the computation problem is small, the cost of communication becomes dominant. Also for the Case-Lower-4, the lowest efficiency is 46%, while its upper-bound limited by time skew is 87%.

8.2.3 Single Supernode Loops

From eqn (6.13) and Loops 6.4.3, we know that the single supernode loops have a form
FOR $q''_0 = \max(l_0, 0)$ TO $\min(u_0, g_0 - 1)$

...

FOR $q''_{N-1} = \max(l_{N-1}, 0)$ TO $\min(u_{N-1}, g_{N-1} - 1)$

where l_i and u_i are the linear functions of $q''_0, \cdots, q''_{i-1}$, and g_i's define the sizes of the supernode in each dimension. In some situations, the time for calculating the bounds l_i's and u_i's becomes too significant to be ignored. In fact, l_i will be calculated $\prod_{j=0}^{i-1} g_j$ times and so will u_i. For instance, for the 2-D case, if $g_0 = 100$ and $g_1 = 1$, we have to calculate l_1 and u_1 100 times, that is, to calculate them once for each node. This results in a very poor efficiency if the time for calculating the bounds is comparable to that of computing a node. In contrast, if $g_0 = 1$ and $g_1 = 100$, we need to calculate l_1 and u_1 once for the whole supernode, so the time for it will be insignificant.

The problem also exists for any general sequential nested loops and becomes serious for small size computational problems. The supernode can be regarded as a sequential-loop nest of small size. So from this point-view, we prefer large granularity.

If there is some freedom for partitioning, such as in the cases of Lower-D and SUC, the sizes of the inner loops of the supernode, especially g_{N-1}, should be as large as possible. However, the shapes of the original polyhedron also influence the choice of g_i and freedom is sometimes limited. For example, compare Case SUC-5 and Case SUC-6. They have the same K and D, they should have the same fitness. But they show significant difference in

efficiency, because the c of Case SUC-5 is larger than that of Case SUC-6, so the former has a larger g_2. Reviewing the results of Case lower-D's, we find this strategy works well since there is more freedom for partitioning. See also Case lower-D-19 and Case lower-D-20, Case lower-D-21 and Case lower-D-22. With regards the SBC cases, because there is no freedom for partitioning efficiency is affected significantly.

8.2.4 The Effect of Granularity

From the discussions above, we propose that the best strategy is to increase the size of the supernode. This conflicts with the previous criterion of optimising the size of the supernode where it is claimed that a small supernode can reduce the overall executing time. Obviously the previous criterion does not take these practical factors into consideration. How to take these factors into account remains a difficult problem. The difficulty lies in the fact that the time for an overhead operation is unpredictable and changeable for different computational problem and different computing facilities. The time ratio of communications over computations is also an important issue and is also unpredictable, because it depends on the type of data and the kind of the computation.

From our experiments, we find that the practical benefits from increasing the size of the supernode can overwhelm the theoretical loss. In fact for the Lower-D cases, when we test with the smallest supernode size, the parallel program is much slower than a sequential one, because all overhead must be carried out for each small supernode. In practice, we choose large granularity if there are freedoms left to define supernode sizes.

8.3 Summary

A large number of experiments have been carried out to test the performances of the automatically generated parallel codes. The experimental results give us a general idea of how the methodology performs. Speedups can be seen for most of the results. It should be pointed out that the real computation in these examples are small because there is only a single recurrence and the computation is only a number of simple additions. For systems of recurrence array from high level synthetic procedures and for the cases where complex computations are involved in each iteration, the computation of a node and supernode

hence may rise significantly. As a result, this will improve the ratio of the computing time over the overhead time, giving a higher efficiency. However, we restrict ourself to quite hash situations.

Generally speaking, from the three sets of experiments, the Lower-D array set is the best in efficiency, and the (N-1)-D array with SUC mesh set is reasonable, while the set of (N-1)-D array with SBC mesh produced poor results.

These experimental results and the analysis may give us some clues about how to select a processor array when given an application. If the size of computation is not too large, a linear array is recommended, otherwise we have to consider 2-D arrays. Among 2-D arrays we may try those with SUC meshes first. The 2-D arrays with SBC meshes should be taken into consideration if the computation polyhedron is diamond-shaped.

Chapter 9

Conclusion

In this final chapter, the whole design procedure we have developed is summarised. Furthermore, the evaluations and comparisons are presented from both theoretical and practical views-points. This shows that our methodology makes a remarkable progress in this area and gives a prospect of automatically parallelizations.

9.1 Summary Of The Whole Design Procedure

Except for some miscellaneous topics in Chapter 3 which improve on existing partitioning and mapping techniques, we propose a complete methodology of automatic generation of parallel programs for regular array designs, based on algorithms expressed as Uniform Recurrence Equations. We start with partitioning and end by producing parallel codes. Below we attempt to make a summary of, and evaluate, the work of each chapter.

In Chapter 4, A Positive Expression Basis $\mathbf{E}$ is derived from the original dependencies $\mathbf{D}$. This basis plays a key role in our method, by defining the direction of the partitioning parallelepiped and guarantees that all data communication will flow along the edges of the partitioning parallelepiped. If we scale the sizes of the parallelepiped properly so that no data communication may penetrate the parallelepiped, the locality of communication is ensured. More importantly, we obtain a unique data dependency relation among the partitions, i.e., the canonical dependencies, independent of particular problems. Therefore, it becomes possible to develop a uniform methodology for the whole mapping procedure.

Based on the two common patterns of interconnection primitives, SUC and SBC, we proposed the models of space-time mapping matrix $\mathbf{T}$, consisting of $\mathbf{S}$ and $\bar{\mathbf{t}}$, such that

they guarantee the implementation of data communication of the canonical dependencies by utilising the two patterns of interconnection primitives. The models of **S** must be selected before deciding the sizes of partitioning parallelepiped to achieve a fixed-size partitioning.

Then, we re-scale the **E** to **B**, which is the basis for quasi-supernode space and also define the partitioning parallelepiped, such that the quasi-supernode domain can be mapped within a given array. The procedure of re-scaling for SUC is straightforward, but is much more difficult for SBC, since $\mathbf{S}_{SBC}$ has more than one non-zero entry in each row. For SBC, a LSGP partitioning is applied to improve the efficiency. This does not affect the procedure in nature at this stage.

Chapter 5 extends the idea of Chapter 4 to the case of lower-dimensional arrays. However, the single timing function $\bar{t}$ is replaced by a map of the original domain onto a K-D virtual time domain. After this transformation we derive a valid minimum vector which re-projects the K-D virtual time domain along a 1-D domain on the condition that no more than one node maps onto one time point and the executing sequence remains as it should be and the 1-D domain is as compact as possible. Fortunately, we find that the search time is independent of the size of the problem, so it promises a fundamental advantage over other methods with respect to computational complexity.

In Chapter 6 and 7 we address the challenge of generating parallel algorithms and codes. Since the supernode domain will be the elements of our parallel programs, the features of the supernode domain are explored first, such as the boundary of supernode domain and the boundary of a single supernode. The supernode domain is then mapped to the processor-time domain which corresponds to the processor array and the executing sequence in a processor. For the lower-dimensional case, an algorithm structure is invented such that the K-D time loops can be executed as 1-D time sequence. For the LSGP case, we also propose an algorithm structure which carries out the LSGP partitioning.

Based on the two algorithm structures, we present two parallel code templates for actual code generation. Other changeable information, such as the bounds of loops, are collected in a header file which also contains some minor loops structures for actual computations and communications. The data communication problem is given special attention.

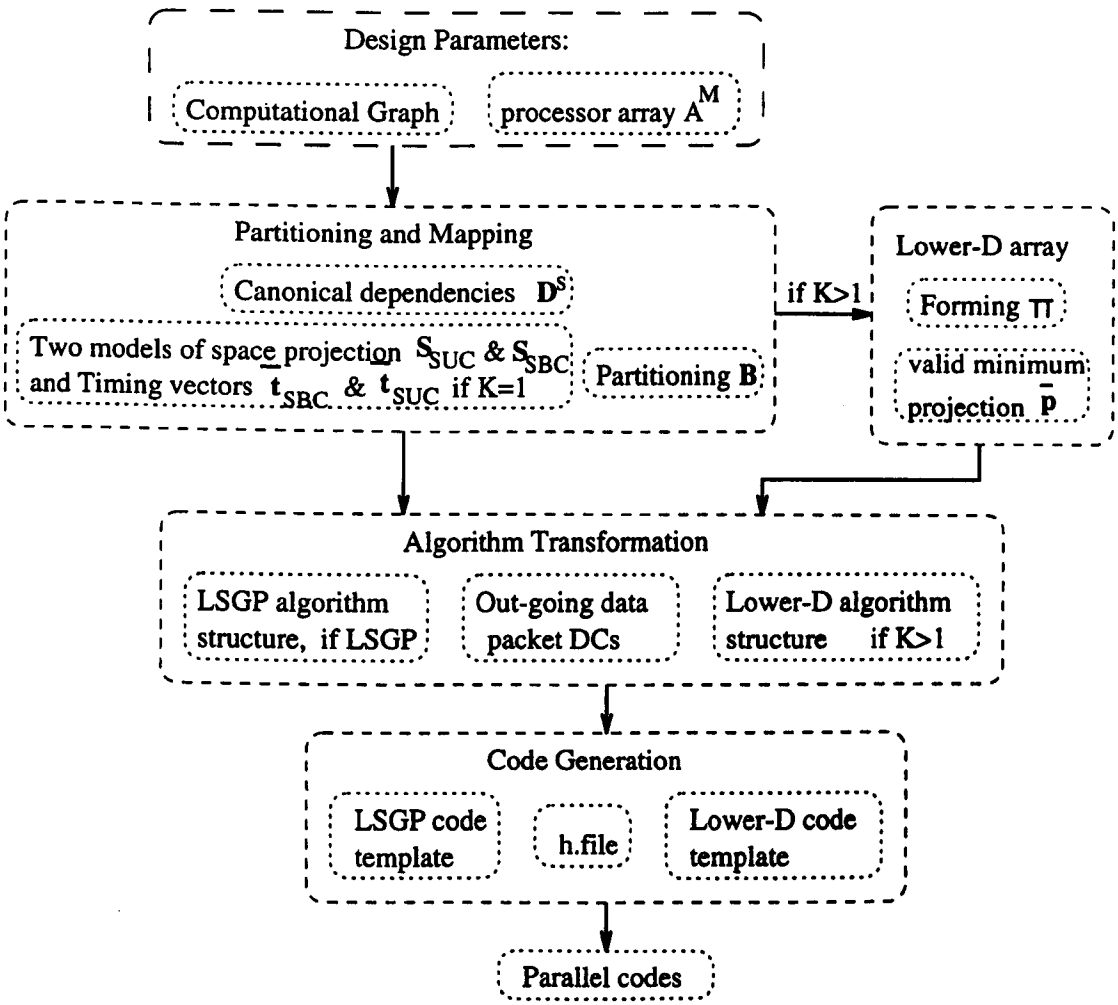


Figure 9.1: Chart of the Whole Procedure.

In Chapter 6, we figure out data flow blocks which confine the outgoing data and describe their flow directions. In Chapter 7, this data flow is embodied in in/out packets which describe the sources or destinations of the data flow and their sizes. Combining the template and the header file, we produce parallel code which can run correctly on a parallel computing platform built by existing environmental tools. Figure 9.1 is expected to give a general idea of the whole methodology.

9.2 Evaluations and Comparisons

Our work consists of both theory and practice. In the aspect of theory, we stand on the shoulders of other previous researchers, so we may see farther and better. In the aspect

	hardware	computational efficiency	mapping onto given regular array	implementing data communication
LPGS	more needed	may be low	guaranteed	not guaranteed
LSGP	no more	high	difficult to search	not guaranteed
our method	no more	high	guaranteed	guaranteed

Table 9.1: Comparison of three methods in theory. Here, "low" is due to the invalid holes in the processor-time domain; "not guaranteed" means that the method does not solve the problem for all cases.

of practice, our work is relatively original, since most other researchers have not arrived at this stage yet.

9.2.1 Theory

The main theoretical contributions are listed:

Mapping onto fixed-size Array

The canonical dependencies are the main highlight, and make it possible to generate a general modelling of the various URE problems so that the major design process is independent of individual applications. The basic models of space mapping matrix and timing vector (or matrix) gives the foundation of the universal design process. Because these models are established on the assumption of the simplest processor arrays, they guarantee the basic feasibility of the methodology generally (in any reasonable situations). Therefore, in this sense, it can be claimed that this methodology is universally applicable and feasible.

The method itself belongs to the supernode partitioning classification. Previous supernode partitioning suffers from the drawback that it is difficult to map onto a fixed-size array. Our method copes with the problem properly. In Chapter 2, we have discussed the major advantages and disadvantages of the most well-known partitioning mathods, such as LPGS and LSGP. Table 9.1 shows the comparisons of the original LPGS and LSGP methods with ours.

Lower-dimensional mapping

The extension of this methodology to lower-dimensional arrays is another great challenge, which is far from trivial. A complex procedure is presented to derive the valid minimum projection vector. This method has significant advantages over the previous works. Compared to [104], the efficiency is improved greatly. Compared to [94] and [34], instead of searching our method is based on synthesis computations which are independent of problem size, so it has a fundamental merit in design; secondly, our method can obtain optimisation when the polyhedron is irregular; finally, by means of a non-singular mapping matrix, we have found a simple way to map from M-D space domain and 1-D time domain to the original N-D computational domain which is the key to a usable practical method.

9.2.2 Practice

The theoretical work is only half of our work. A more important and potentially more difficult part is the implementation of the methodology to generate actual parallel codes.

The main contributions in the aspect can be listed briefly:

1 Derivation of boundaries of the computation in the processor-time domain. The difficulty lies in determining the confining supernode polyhedron.

2 Construct the structure of parallel algorithms for the two cases. One is the lower-dimensional array and the other is for LSGP partitioning.

3 Establish the mechanism of data communication from the data dependencies. For strictly implementing our methodology, the data relay is also implemented by ourselves

4 Suggest a feasible program structure for the parallel codes, which is easy to generate automatically.

These results pave the road to real applications. So far we cannot find similar work at the same level of detail in the literature.

From a practical view-point, we find that the supernode partitioning is necessary to carry out computation and communication, even if it also brings about a major complexity to the design procedure. In this respect, the LSGP method undergoes further setbacks. Note that the parallel algorithm of pure LSGP method, Algorithm D, does not make a use of supernode partitioning. Therefore, according to the analysis of Section 6.5, in order to decide which node to compute we have to carry out Algorithm 6.5.1 once for each point in the processor-time domain to find the corresponding node. This is too time-expensive! In addition, it is also very time-consuming to invoke the communication mechanism once for transferring data for a single node. From the view-point of data transference, the LPGS method is better somehow, that is, the data produced in one time layer can be transferred together as a packets. But its data flow packets cannot be large, since usually without supernode partitioning the time layers are very "thin". In contrast our method, which uses the supernode partitioning, is of significant advantage.

9.2.3 Results

As the result of the theoretical and practical work, a software tool has been developed which accepts the description parameters for the computation problem and the given regular processor array, and then produces parallel codes.

Many experimental results are provided to check and show the performance of the methodology. We find they show significant speedup in many cases. The linear array shows better efficiency, some as high as up to 90%. The 2-D array with SUC mesh works well, too. The 2-D array with SBC mesh behaves disappointedly in efficiency, though it also shows some speed-up. After all, it is still an alternative choice for some applications.

Unfortunately there are no experimental results presented in the literature to allow us to make direct comparisons with other synthesis method.

9.3 Closing Remarks

The achievements presented in this thesis can be regarded as a significant progress in works of automatic generation of parallel codes and regular (systolic) array design. This methodology is integrated and self-contained, and may be the only practical working

package in this area.

Obviously, the automatic parallel code generation shows incomparable advantages over a manual one. The time for generating a parallel program can be reduced dramatically, from weeks (even months) to a few minutes, and the correctness is guaranteed. To the fundamental kind of computational problem Uniform Reccurence Equation, it gives a real application prospect of parallel computing.

However it is far from true to claim that our work closes the research in this area. Obviously, one problem is to exploit the possibility of improving efficiency further. In addition, although our method is developed for the URE problem and for "software systolic array", it is possible to extend the methodology for more situations and applications where non-URE problems can be transformed to URE or similar forms. As regards general cases of non-URE, there is much work waiting for future researchers.

Bibliography

[1] M.Ander and F.Berman "Assessing Partitioning, Scheduling, Storage Trade-offs for Regular Interative Algorithms", J VLSI Integration, V.15, N.1, p25-50, 1993

[2] W. L. Athas and C. L. Seitz, " Multicomputers: Message-passing concurrent computer", *IEEE Comput. Mag.*, pp. 9-24, Aug. 1988

[3] V. Balasubramanian and P. Banerjee, "Compiler-assisted synthesis of algorithm-based checking in multiprocessors" *IEEE Trans. Computer* Vol.39, No.4 Apr.1990 pp. 436-446.

[4] U. Banerjee, "Data dependence in ordinary programs" Tech. Rep. 76-837, Univ. of Illinois Urbana-Champaign, Nov 1976

[5] U. Banerjee, "Dependence analysis for supercomputing" Boston, MA: Kluwer Academic, 1988

[6] U. Banerjee, "A theory of loop permutations" in Proc. 2nd Workshop Languages Compilers Parallel Computing, Aug. 1989

[7] U. Banerjee, "Unimodular transformations of double loops", in Proc. 3rd Workshop Languages Compilers Parallel Computing, Aug, 1989

[8] F. Berman, "On mapping parallel algorithms into parallel architectures", J. Parallel and Distributed computing, 4, 1987, pp439-458

[9] G.H. Bradley : Algorithm and bound for the greatest common divisor of n integers. Communications of ACM, Vol.13, No.7, 1970, pp433-436

[10] M.O'Boyle and G.A.Hedayat, "Load Balance of Parallel affine Loops By Unimodular Transformation", Department of Computer science, University of Manchester.

[11] J.C.Bu, E.F.Deprettere and P.Dewilde, "A Design Methodology for Fixed-Size Systolic Arrays", Pro., International Conference on Application Specific Array, IEEE Society, 1990, pp591-602

[12] C. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors", in Proc. 1988 Workshop on Programming Languages and Compilers for Parallel Computing, Aug. 1988

[13] Sharat Chandran and Larry S. Davis "Parallel vison algorithms: an approach" Parallel Processing for Scientific Computing, SIAM, pp235-254

[14] V. Chaudhary and J. K. Aggarwal "A Generalized Scheme for Mapping Parallel Algorithms", *IEEE Trans. Parallel and Distributed System* Vol.4, No.3, 1991 pp.328-346

[15] Zen Chen and Chih-Chi Chang, " Iteration-level execution of DO loops with a reduced set of dependence relations", J. Parallel and Distributed Computing, 4, 1987, pp488-504

[16] M. Chen, "A design methodology for synthesizing parallel algorithms and architecture", J. Parallel Distributed Comput., Dec. 1986. pp. 461-491

[17] D.S.Chand and S.S.Kapur, "An Algorithm for Convex Polytopes", JACM, Vol.17(1) pp78-86, 1970

[18] Xian Chen and G.M.Megson, "A General Methodology of Partitioning and Mapping for Given Regular Array", IEEE, Trans. Parallel and Distributed System Vol.6, No.10, 1991 pp.1100-1107

[19] Xian Chen and G.M.Megson, "Optimal Mapping onto Lower-dimensional Arrays", under the second review of IEEE, 1994

[20] Xian Chen and G.M.Megson, "Automatic Parallel Code Generation for Given Appay (Part 1: Theory)", Technical Report No.502, University of Newcastle, Feb, 1995

[21] Xian Chen and G.M.Megson, "Automatic Parallel Code Generation for Given Appay (Part 2: Practice and Results)", Technical Report No.506, University of Newcastle, Feb, 1995

[22] R. Cytron, "Compiler-time scheduling and optimization for asynchronous machines" Ph.D dissertation, Uni. of Illinois at Urbana-Champaign. 1984

[23] A. Darte and J. M. Delosme, "Partitioning for array processors" Technical Report, 90-23. LIP, ENS-Lyon Lyon, France, Oct. 1990"

[24] A. Darte, "Regular Partitioning for Synthesizing Fixed-Size Systolic Array", J. of VLSI Integration, 12, 1991, pp293-304

[25] J.C.Bu, E.F.Deprettere and P.Dewilde, "A Design Methodology for Fixed-Size Systolic Arrays", Pro., International Conference on Application Specific Array, IEEE Society, 1990, pp591-602

[26] R.F.DeMara and D.I.Molovan "The SNAP-1 Parallel AI Prototype", *IEEE Trans. Parallel and Distributed System* Vol.4, No.8, 1991 pp. 846-854

[27] M. L.Dowling, "Optimal code parallelization using unimodular transformations", Parallel Computing, 16 (1990) pp157-171

[28] T.Y.Feng "A summary of interconnection networks" IEEE Computer 14 (12) pp12-17, 1981

[29] M.J.Flynn "Some computer organization and their effectiveness" IEEE Trans. Comput. C-21, pp948-960, 1972

[30] J.A.B.Fortes, K.S.Fu and B.W.Wah, "Systematic Design Approaches for Algorithmically Specified Arrays", In Computer Architecture Concepts and Systems, eds Milutinovic, 1988, pp454-494

[31] J.A.B.Fortes and D. I. Moldovan, "Parallelism detection and transformation techniques useful for VLSI algorithms", J. Parallel Distributed Comput., vol. 2, pp. 277-301, 1985

[32] J. A. B. Forte "Algorithm transformations for parallel processing and VLSI architecture design", Ph.D dissertation Univ. Southern California¡ CA, Dec. 1983

[33] P.Gachet, B.Joinnault and P.Quinton, "Synthesizing Systolic Arrays Using DIASTOL", In Moor, McCabe, Uruguart, Int Workshop on Systolic Arrays, A dam-Hilger, 1986, pp25-36

[34] K.N.Ganapathy and B.W.Wah "Optimal Synthesis of Algorithm-Specific Lower-Dimensional Processor Array", Technical Report, University of Lllinois at Urbana-Champaign, CRHC-93-23

[35] D. C. Grunwald and D. A. Reed , " Networks for parallel processors: Measurement and prognostication;;, in Proc. Third Conf. Hypercube Concurrence Comput. Appl., Jan. 1988 pp. 238-253

[36] D.A.P.Haie, "Multiprocessors: Discussions of Some Theoretical and Practical Problems", Ph.D. Dissertation, Univ. of Illinois at Urbana-Champain, Urbana, IL, 1979. Rep. UIUCDCS-R-79-99.

[37] G.R.Heijer and E.F.Deprettere "From Algorithm to Parallel Implementation" in 1992 IEEE Workshop on VLSI Signal processing, Napa Valley, p342-354

[38] B.J.Hellant and R.J.Krueger, "A parallel algorithm for rapid computation of transient fields", Parallel Processing for Scientific Computing, SIAM, pp270-275.

[39] R.W.Hockney and C.R.Jesshope, "Parallel computer 2"

[40] S.Horiik, S.Nishida and T.Sakaguchi "A design method of systolic array under the constraint of the number of the processors", Int. Conf ASSP, 1987.

[41] H.M.Hsu, J.K.Peir and D.B.Haidvogel, "Performance of an ocean circulation model on LCAP" Parallel Processing for Scientific Computing, SIAM, p285

[42] T.C.Hu, Combinatorial Algorithms, Ch3, Addison-Wesley, 1982

[43] K.Hwang and Y.H.Chung, "Partitioning algorithms and VLSI structures for large-scale matrix computations," in Proc. 5th Symp. Comput. Arithmetic, May 1981, pp222-232

[44] O. H. Ibarra and S. M. Sohn, " On Mapping systolic algorithms onto the Hypercube" 1988 *IEEE Trans. Parallel and Distributed System* Vol.1, No.1 Jan.1990 pp. 48-63

[45] INMOS, Transputer Reference Manual, Prence Hall, 1988.

[46] F.Irigoin and R.Triolet "Supernode Partitioning", Pro. of 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1988, pp319-329

[47] F.Irigoin and R.Triolet "Dependence approximation and global parallel code generation for nested loops", in Parallel Distributed Algorithms, 1989

[48] K.Jainandunsing. "Parall algorithms for solving systems of linear equations and their mapping on systolic arrays." PhD thesis, Delft University of Technology, Delft, Netherlands 1989

[49] Ravindran Kannan and Achim Bachem, "Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix", SIAM J. Comput., Vol.8, No.4, 1979, pp499-507

[50] R.M.Karp, R.E.Miler and S. Winograd, "The organization of computations for unification recurrence equations", J. ACM, Vol.14, No.3, 1967, pp563-590

[51] M.T.O'Keefe and J.A.B.Fortes, "Bit Level Processor Array: Current Architectures and a Design and Programming Tool", In Proc. 1988 Int. Symp. Circuit Syst., 1988, pp2751-2755

[52] Chung-Ta King, Wen-Hwa Chou and Lionel M.Ni "Pipelined Data-Parallel Algorithms. Part II - Design", IEEE, Trans. Parallel and Distributed System, VOL.1, NO.4, 1990, pp486-499

[53] Kung, H.T. and C.E.Leiserson, " Systolic Arrays for VLSI", Proceedings of the Sparse Matrix Symposium (SIAM), 1978.

[54] S.Y. Kung. VLSI Array Processors. Prentice-Hall International Editions, 1988.

[55] S.Y. Kung. "Wavefront array processor: Language, architecture, and applications", IEEE Trans. Comput., vol. C-31, pp1054-1066, 1982

[56] L.Lamport, "The Parallel Execution of DO Loops", Commun. ACM. 1974, pp83-93

[57] C. Lengauer, M. Barnett, and D. G. Hudson, "Towards systolizing compilation", Distributed Computing, 5 (1991) pp. 7-24

[58] Pizong Lee and Z.M.Kedem "Synthesizing Linear Array Algorithms from nested For Loop Algorithms", *IEEE, Trans. Computer*, VOL.c-37, NO.12, 1988, pp1578-1598

[59] Pizong Lee and Z.M.Kedem "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays", *IEEE, Trans. Parallel and Distributed System*, VOL.1, NO.1, 1990, pp64-76

[60] Wei-ming Lin and V.K.Prasanna Kumar, "A note on the linear Transformation method for systolic array design", *IEEE Trans. Computer*, Vol.39, No.3, 1990, pp393-399

[61] G.Li and B. W. Wah, "The design of optimal systolic arrays", *IEEE Trans. Computer*, pp. 66-77, Jan 1985

[62] S. Manoharan and P. Thanisch, "Assigning dependency graphs onto processor networks", Parallel Computing, 17 (1991) pp. 63-73 63-73

[63] MEiKO Limited, "C Reference Manual for the Computing Surface"

[64] MEiKO Limited, "CS Tools Reference Manual"

[65] G.M.Megson and E.O.Eyoh, "Implementation and Evaluation of Parallel N-D Convex Hull Algorithms", To appear on PARCO'93, Grenoble, France

[66] G.M.Megson and Xian Chen, "A Survey and Analysis of Partitioning and Mapping for Regular Arrays", submitted for publication, also Technical Report No.415, University of Newcastle, 1993

[67] G.M.Megson and Xian Chen, "Partitioning and Mapping for Lower Dimensional Given Regular Array", in 1993 IEEE Euromicro Workshop on Parallel and Distributed Processing, pp149-155.

[68] G.M.Megson and Xian Chen, "Systematic Synthesis of Knapsack Problems onto Fixed Sized Arrays with Lower Dimensions", submitted for publication, also Technical Report No.486, University of Newcastle, 1994

[69] G.M.Megson and Xian Chen, "A synthesis method of LSGP partitioning for given-shape regular arrays", Proc. 9th Int Parallel Processing Symp., IEEE Computer Society Press, 1995, pp234-238

[70] Megson G.M., "Automating Systolic Algorithm Design I : (basic synthesis techniques)", Technical Report Series, No.364, University of Newcastle uopn Tyne, 1991.

[71] G.M.Megson "The Derivation of Uniform Recurrence Equations for the Knapsack Problem", J. of Parallel Algorithms and Applications, Vol 1, 1993, pp127-140.

[72] G.M.Megson "Mapping a Class of Run-Time Dependencies onto Regular Arrays", Proc. 7th Int Parallel Processing Symp., IEEE Computer Society Press, 1993, pp97-104.

[73] D.Moldovan and A.B.Fortes,"Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays", *IEEE, Trans. Computer* VOL.c-35, NO.1, 1986, pp1-12

[74] D. Moldovan : ADVIS: a software package for the design of systolic arrays. *IEEE Trans. Computer* Vol.36, No.1 Jan.1987 pp. 33-40

[75] L. Mordell, Diaphantine equations. New York, Academic. 1969

[76] H.Nelis and E.Deprettere"Automatic Design and Partitioning of Systolic/wavefront Arrays for VLSI". Circuits, System, Signal Processing 7, 1988

[77] L. M. Ni and C. T. King, "On partitioning and mapping for hypercube computing", Int. J. Parallel Programming, Vol, 17, no. 6, pp. 475-495, Dec. 1988

[78] D.A.Padua, "Multiprocessors: Discussion of Theoretical and Practical Problems", Ph.D., Dissertation., Rep.UIUCDCS-R-79-990, Univ. of Illinois at Urbana-Champaign. Urbana, 1979.

[79] A.Pashapour, G.A.Pope, K.Sepehrnoori and G.Shiles, "Application of vectorization and microtasking for reservoir simulation" Parallel Supercomputing: Methods,Algorithms and Applications, pp267-281, 1989

[80] D. A. Padua and M. J. Wolfe, " Advanced compiler optimizations for supercomputers", Commun. ACM, pp. 1184-1201, Dec. 1986

[81] J.K.Peir and R.Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors", IEEE, Trans. Computer, VOL.38, NO.8, 1989, pp1203-1211

[82] C. D. Polychronopoulos, D. J. Kuck and D. A. Padua, "Utilizing multidimensional loop parallelism on large-scale parallel processor system", *IEEE Trans. Computer* pp. 1285-1296, Sept. 1989

[83] Quinton P. and Van Dongen V., "The Mapping of Linear Recurrence Equations on Regular Arrays". J VLSI Signal Processing, 1, pp95-113, 1989

[84] Quinton P., " Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equation". Pro. 11th Symp on Computer Architecture, IEEE Computer Society Press. pp208-214, 1984

[85] S. V. Rajopadhye, "Regular iterative algorithms and their implementations on processor array", Ph.D dissertation. Stanford Univ., Stanford, CA, Oct. 1985

[86] S. V. Rajopadhye, "Sizing systolic array with control signals from recurrence equations", Distributed Computing, 3 (1989) pp. 88-105

[87] S.V.Rajopadhhye and R.M.Fujimoto, "Synthesizing Systolic Arrays from Recurrence Equations", Parallel Computing, Vol.14, 1990, pp163-189

[88] S. V. Rajopadhye, "Automating the design of systolic arrays", Integrating, The VLSI Journal 9, 1990, pp225-242

[89] L. Rapanotti, "On the Synthesis of Integral and Dynamic Recurrences" Newcastle Unversity, Nov 1995 (submitted)

[90] A. Schrijver, Theory of Linear and Integer Programming, New York: Wiley, 1986

[91] Jang-Ping Shu and Chih-Yung Chang, "Synthesizing nested loop algorithms using nonlinear transformation method." *IEEE Trans. Parallel and Distributed System* Vol.2, No.3 Jul.1991 pp. 304-317

[92] Weijia Shang and Jose A. B. Fortes : Time optimal linear schedules for algorithms with uniform dependencies. IEEE Trans. Computer, Vol.40, No.6 Jun.1991 pp. 723-742

[93] W.Shang and J.A.B.Fortes, "Independent Partitioning of Algorithms with Uniform Dependencies", IEEE, Trans. Computer, VOL.41, NO.2, 1992, pp190-206

[94] Weijia Shang and Jose A. B. Fortes, "On Time Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays", *IEEE, Trans. Parallel and Distributed System*, VOL.3, NO.3, 1992, pp350-363

[95] J.E.Shore "Second thoughts on parallel processing" Comput.Elect.Eng. 1, pp95-109, 1973

[96] J.P.Sheu and T.H.Tai, "Partitioning and Mapping Nested Loops on Multiprocessor Systems", IEEE, Trans. Parallel and Distributed System, VOL.2, NO.4, 1991, pp430-439

[97] M.A.Stoker, "The Exploitation of Parallelism on Shared Memory Multiprocessors", Ph.D Thesis, University of Newcastle upon Tyne, 1990.

[98] G. Swart "Finding the Convex Hull Facet by Facet", J.Algorithms, Vol.6, pp17-48, 1985.

[99] J. Teich and L. Thiele "Partitioning of Processor Arrays: a Piecewise Regular Approach" Integration, VLSI Journal 14, 1993, pp297-332

[100] Ping-Sheng Tseng, "A systolic array parallelizing compiler", J. Parallel and Distributed computing, 9, 1990, pp116-127

[101] T.H.Tzen and L.M.Ni "Dependence Uniformization: A Loop Parallelization Technique", *IEEE Trans. Parallel and Distributed System* Vol.4, No.5, 1991 pp.547-558

[102] S.S.Vincentelli, "Parallel Processing for simulation of VLSI circuits", Parallel Processing and Application pp3-19 1988

[103] M. Wolfe and U. Banerjee," Ddta dependence and its application to parallel processing", Int. J. Parallel Programming, Vol. 16, no. 2, pp 137-178, 1987

[104] Y.Wong and J.M.Delosme, "Optimal Systolic Implementation of N-dimensional Recurrences", *IEEE Proc. ICCD*, 1985 pp.618-621

[105] M.E.Wolf, "Improving parallelism and data locality in nested loops", Ph.D dissertation, Stanford Univ. 1991

[106] M.E.Wolf and M.S.Lam,"A data locality optimization algorithm", in Proc. ACM SIGPLAN '91 Conf. Programming Language Design Implementation, Jun. 1991, pp. 30-44

[107] M.E.Wolf and M.S.Lam,"A Loop Transformation Theory and an Algorithm to Maximize Parallelism", IEEE, Trans. Parallel and Distributed System, VOL.2, NO.4, 1991, pp452-471

[108] M.Wolfe "More Iteration Space Tiling", Proc., Supercomputer' 89

[109] M.Wolfe "Optimizing Supercompilers for Supercomputers", Cambridge, MA: MIT Press. 1989.

[110] Zhenhui Yang, Weijia Shang and J.A.B.Fortes, "Conflict-free Scheduling of Nested Loop Algorithms on Lower Dimensional Array Processor Arrays", *Proc. of Int. Parallel Processing Symp.*, 1992, pp156-164

[111] Xiaoxiong Zhong and S.Rajopadhye, "Quasi-Linear Allocation Function for Efficient Array Design", J. of VLSI Signal Processing, 4, 1992, pp97-110

Appendix A

Scaling SBC

If w_i^u and w_i^l are known for all i, this would be a simple problem to find the solutions for the system of equations (4.43). But because the mapping depends on the scaled space mapping S_F, i.e., w_i^u and w_i^l depend on f_i and f_{i+1}, we fall into a mutually recursive cycle. Eqn (4.34) is no longer true, because there are two variables in an equation.

The scaling of SBC can be divided into two stages:

(1). To break the cycle, begin with an initial point of f_k. At the initial point, it is possible to determine w_i^u and w_i^l, $\forall i$, $0 \leq i < M$.

This is a complicated procedure. w_i^u and w_i^l depend not only on f_k but also f_{i+j}, where j = 0, 1. And there is another mutually recursive cycle between w_i^u and w_i^l and f_{i+j}. Break this cycle by beginning with an initial point of f_{i+j} to establish a relation between w_i^u and w_i^l and f_{i+j}.

By this way, a set of initial w_i^u and w_i^l are found corresponding to the initial f_k.

(2). Derive functions for F^a, with these initial w_i^u and w_i^l. Then, increase f_k from the initial point. At some points, w_i^u and w_i^l are replaced by other w_i's. Derive the next piecewise functions for F^a.

A.1 Initial w_i^u's and w_i^l's

As mentioned, when f_k is taken as an argument, let $f_k = 0$ be the initial point. Immediately the (k-1)-th and the k-th equations of eqn (4.42) and eqn (4.43) become single-variable equations, that is

$$u_{k-1,j} = -f_{k-1} w_{k-1,j}, \ \forall j \ , 0 \leq j < n_v$$

$$u_{k,j} = f_{k+1} w_{k+1,j}, \quad \forall j \ , 0 \le j < n_v \tag{A.1}$$

Therefore, the method for SUC can be adopted here to determine f_{k-1} and f_{k+1}, when $f_k = 0$, i.e.,

$$\begin{aligned} f^a_{k-1} &= \frac{l_{k-1}}{\max_{0\le j<n_v}(-w_{k-1,j}) - \min_{0\le j<n_v}(-w_{k-1,j})} \\ f^a_{k+1} &= \frac{l_{k+1}}{\max_{0\le j<n_v}(w_{k+1,j}) - \min_{0\le j<n_v}(w_{k+1,j})} \end{aligned} \tag{A.2}$$

The following problem is to determine $f_{k+2}, f_{k+3}, \ldots$ and $f_{k-2}, f_{k-3}, \ldots$. Unfortunately, that is not an easy job. f_{k+2} should be determined by $u^u_{k+1} = -f_{k+1} w^u_{k+1,0} + f_{k+2} w^u_{k+1,1}$ and $u^l_{k+1} = -f_{k+1} w^l_{k+1,0} + f_{k+2} w^l_{k+1,1}$, but $\mathbf{w}^u_{k+1}$ and $\mathbf{w}^l_{k+1}$ depend on f_{k+2}. So we need to initialise f_{k+2} (from $f_{k+2} = 0$), and see what is happening as f_{k+2} increases.

Suppose $f_{k+2} = f^i_{k+2}$. f^i_{k+2} is an initial f_{k+2}. Then it is easy to determine $\mathbf{w}^u_{k+1}$ and $\mathbf{w}^l_{k+1}$ [1]

$$\begin{aligned} \mathbf{w}^u_{k+1} &= \{\mathbf{w}_j : \bar{\mathbf{s}}^F_{k+1}\mathbf{w}_j = \max_{0\le l<n_v}(\bar{\mathbf{s}}^F_{k+1}\mathbf{w}_l), 0 \le j < n_v\} \\ \mathbf{w}^l_{k+1} &= \{\mathbf{w}_j : \bar{\mathbf{s}}^F_{k+1}\mathbf{w}_j = \min_{0\le l<n_v}(\bar{\mathbf{s}}^F_{k+1}\mathbf{w}_l), 0 \le j < n_v\} \end{aligned} \tag{A.3}$$

However, as f_{k+2} increases, the $u_{k+1,j}$ projected by some vertices may increase faster than that of the $\mathbf{w}^u_{k+1}$, or decrease faster than that of the $\mathbf{w}^l_{k+1}$, so they should become the new u^u_{k+1} or u^l_{k+1}. Therefore, it is necessary to know the turning point f^t_{k+2} where such a "overstep" takes place. $\forall j$, $0 \le j < n_v$ [2], we have

$$-f_{k+1} w^u_{k+1,0} + f^u_{k+2,j} w^u_{k+1,1} = -f_{k+1} w_{k+1,j} + f^u_{k+2,j} w_{k+2,j}$$

or

$$\begin{aligned} f^u_{k+2,j} &= f_{k+1}\left(\frac{w^u_{k+1,0} - w_{k+1,j}}{w^u_{k+1,1} - w_{k+2,j}}\right), \text{ and similarly} \\ f^l_{k+2,j} &= f_{k+1}\left(\frac{w^l_{k+1,0} - w_{k+1,j}}{w^l_{k+1,1} - w_{k+2,j}}\right) \end{aligned} \tag{A.4}$$

[1] There is the possibility that more than one vertex satisfies $\bar{\mathbf{s}}^F_{k+1}\mathbf{w}_j = \max_{0\le l<n_v}(\bar{\mathbf{s}}^F_{k+1}\mathbf{w}_l)$. Among them, the vertex which diverges fastest as f_{k+2} increases will be the $\mathbf{w}^u_{k+1}$, i.e., whose (k+2)-th element should be the largest. The same argument holds for selecting $\mathbf{w}^l_{k+1}$, but that vertex should have the smallest (k+2)-th element.

[2] In principle, any vertices should be assumed to have the possibility to overstep the $\mathbf{w}^u_{k+1}$ or $\mathbf{w}^l_{k+1}$. However, in practice, we can judge some of them having least possibility or having no possibility at all. For example, the vertices which have been overstepped can be removed from the list for the following operations.

where $f^u_{k+2,j}$ (or $f^l_{k+2,j}$) is the point where $\mathbf{w}_j$ oversteps the present $\mathbf{w}^u_{k+1}$ (or $\mathbf{w}^l_{k+1}$). Then the turning point is

$$f^t_{k+2} = \min_{0 \le j < n_v} \{ f^u_{k+2,j} \cap (f^i_{k+2} < f^u_{k+2,j}), f^l_{k+2,j} \cap (f^i_{k+2} < f^l_{k+2,j}) \} \tag{A.5}$$

that is, f^t_{k+2} is a f_{k+2} where the first overstep takes place.

Now within the period of $[f^i_{k+2}, f^t_{k+2}]$, $\mathbf{w}^u_{k+1}$ and $\mathbf{w}^l_{k+1}$ are known and the f_{k+2} which produces the accurate scaling is evaluated by means of eqn (4.43)

$$f^a_{k+2} = \frac{l_{k+1} - (w^l_{k+1,0} - w^u_{k+1,0}) f_{k+1}}{w^u_{k+1,1} - w^l_{k+1,1}} \tag{A.6}$$

$$\text{If} \quad f^i_{k+2} \le f^a_{k+2} < f^t_{k+2} \tag{A.7}$$

The procedure is terminated by finding f^a_{k+2}. If not, that means by this pair of $\mathbf{w}^u_{k+1}$ and $\mathbf{w}^l_{k+1}$, it fails to determine the accurate scaling, then, letting $f^i_{k+2} := f^t_{k+2}$, repeat the procedure of eqn (A.3)[3], (A.4), (A.5), (A.6) and (A.7) once more, until find f^a_{k+2} or have no vertices in the list for eqn (A.4). Note that we obtain not only f^a_{k+2}, but also its corresponding pair of $\mathbf{w}^u_{k+1}$ and $\mathbf{w}^l_{k+1}$.

Repeating the algorithm for determining all the left f^a_i [4], we find an initial $\mathbf{S}_F$ and an initial set of vertices $\mathbf{w}^u_i$ and $\mathbf{w}^l_i$ which are projected by the initial $\mathbf{S}_F$ to the outmost vertices of U, called initial vertex set. And it is the initial vertex set that we are actually interested in. Letting $k = 1, \ldots, M-1$, we can have M-1 such initial $\mathbf{S}_F$'s and their accompanying initial vertex set.

A.2 $\mathbf{S}_F$ as a Function of f_k

Having the initial vertex set associated with an initial f^i_k, it is possible to determine the other entries of $\mathbf{S}_F$ for the accurate scaling.

[3] If $f^u_{k+2} > f^i_{k+2}$, need not search a new $\mathbf{w}^u_{k+1}$, and vice versa

[4] Only a little modification on the procedure is needed so as to be applicable for determining $f_{k-2}, f_{k-3}, \ldots$.

A.2.1 Determining f^a with a Known Set of $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$

However, it can be observed that in $\mathbf{S}_F$, f_i does not have a direct relation with f_k, but an indirect one, when $i > k+1$. So there should be a recursive operations to express f_{i+1}^a as a function of f_k, At first, suppose $f_i^a = a_i f_k + b_i$. If f_i^a is known, f_{i+1}^a is formulated by eqn (A.6). Then

$$f_{i+1}^a = af_i + b = aa_i f_k + b + ab_i = a_{i+1} f_k + b_{i+1} \tag{A.8}$$

where

$$a = \frac{w_{i,0}^u - w_{i,0}^l}{w_{i,1}^u - w_{i,1}^l}, \qquad b = \frac{l_i}{w_{i,1}^u - w_{i,1}^l} \tag{A.9}$$

A.2.2 Determining the Turning Points

As mentioned before, when f_k increases from zero, $\mathbf{S}_F$ changes, too, as a result of which some vertices of U move outwards faster, so they may become the outmost vertices in a certain dimension. Once this occurs, the $\mathbf{w}_j^u$ or $\mathbf{w}_j^l$ corresponding to the present outmost vertices u_j^u or u_j^l which ever is overstepped should be replaced by those corresponding to the new outmost vertices. Therefore $f_0^a, \ldots, f_M^a$ are piecewise functions with respect to f_k, and f_k^i is also the lower boundary of a piece. We need to find the turning point f_k^t where an overstep takes place, which also is the upper boundary of the piece, as f_k increases. This is similar to the last section in concept.

In the i-th dimension, $\mathbf{w}_j$ oversteps the present u_i^u or u_i^l at

$$\begin{aligned} f_{k,i,j}^u &= -\frac{b_i w_{i,0}^u - b_{i+1} w_{i,1}^u - b_i w_{i,j} + b_{i+1} w_{i+1,j}}{a_i w_{i,0}^u - a_{i+1} w_{i,1}^u - a_i w_{i,j} + a_{i+1} w_{i+1,j}} \\ f_{k,i,j}^l &= -\frac{b_i w_{i,0}^l - b_{i+1} w_{i,1}^l - b_i w_{i,j} + b_{i+1} w_{i+1,j}}{a_i w_{i,0}^l - a_{i+1} w_{i,1}^l - a_i w_{i,j} + a_{i+1} w_{i+1,j}} \end{aligned} \tag{A.10}$$

$\forall$ i and j[5], $0 \le i < M$ and $0 \le j < n_v$. So there are $2(M)(n_v - 1)$ such possible turning points. Some of them are invalid if they are smaller than f_k^i. Thus, the turning point f_k^t should be

$$f_k^t = \min_{0 \le i < n, 0 \le j < n_v} \{ f_{k,i,j}^u \cap (f_k^i < f_{k,i,j}^u),\ f_{k,i,j}^l \cap (f_k^i < f_{k,i,j}^l) \} \tag{A.11}$$

[5]Exclude $j = \{l : \mathbf{w}_l = \mathbf{w}_i^u \cup \mathbf{w}_i^l, 0 \le l < n_v\}$

Let $f_k^i := f_k^t$. If $f_k^i = f_{k,i,j}^u$ (or $f_k^i = f_{k,i,j}^l$), it means that the present $\mathbf{w}_i^u$ (or $\mathbf{w}_i^l$) may be replaced by $\mathbf{w}_j$, and the corresponding f_{i+1}^a should be re-evaluated by eqn (A.8) and (A.9). Note that if the above situation takes place in the i-th dimension, the a_l and b_l should be modified by eqn (A.8), $\forall l, l > i$. In this way, a new piece of the functions f_i^a with respect to f_k is created. Repeat this procedure until $\exists f_i^a < 0, \ \forall i, \ 0 \leq i < n$. The interval of from 0 to the last turning point f_k^t is the valid interval of f_k as the argument.

A.2.3 Selecting $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ from Multi-Candidates

A complicated case should be mentioned with regard to select $\mathbf{w}_i^u$ (or $\mathbf{w}_i^l$). In the i-th dimension, at $f_k = f_k^i$, when there is more than one $\mathbf{w}_j$ (include the present $\mathbf{w}_i^u$ (or $\mathbf{w}_i^l$)) such that $\bar{\mathbf{s}}_i\mathbf{w}_j = u_i^u$ (or $\bar{\mathbf{s}}_i\mathbf{w}_j = u_i^l$), they will move with different speeds as f_k increases further, and only one which moves outwards fastest should be taken as $\mathbf{w}_i^u$ (or $\mathbf{w}_i^l$).

Assume that we know the expression of $f_i^a = a_i f_k + b_i$. Suppose that there are two vertices $\mathbf{w}_{j_1}$ and $\mathbf{w}_j$ such that $\mathbf{u}_i^u = \bar{\mathbf{s}}_i\mathbf{w}_{j_1} = \bar{\mathbf{s}}_i\mathbf{w}_j$ and one vertex $\mathbf{w}_i^l$. We should select $\mathbf{w}_i^u$ from the $\mathbf{w}_{j_1}$ and $\mathbf{w}_j$ for evaluating f_i^a. If we let $\mathbf{w}_i^u = \mathbf{w}_{j_1}$, the difference between u_i^u and the images of $\mathbf{w}_j$ is

$$\begin{aligned} u_i^u - u_{i,j} &= u_{i,j_1} - u_{i,j} = -(w_{i,j_1} - w_{i,j})f_i^a + (w_{i+1,j_1} - w_{i+1,j})f_{i+1} \\ &= -(w_{i,j_1} - w_{i,j})f_i^a + (w_{i+1,j_1} - w_{i+1,j}) \\ &\times \ (\frac{w_{i,j_1} - w_{i,0}^l}{w_{i+1,j_1} - w_{i,1}^l} f_i^a + \frac{l_i}{w_{i+1,j_1} - w_{i,1}^l}) \\ &= a_d f_i^a + b_d = a_i a_d f_k + b_i a_d + b_d \end{aligned} \tag{A.12}$$

where

$$\begin{aligned} a_d &= \frac{(w_{i,j_1} - w_{i,0}^l)(w_{i+1,j_1} - w_{i+1,j}) - (w_{i,j_1} - w_{i,j})(w_{i+1,j_1} - w_{i,1}^l)}{w_{i+1,j_1} - w_{i,1}^l} \\ b_d &= \frac{(w_{i+1,j_1} - w_{i+1,j})l_i}{w_{i+1,j_1} - w_{i,1}^l} \end{aligned} \tag{A.13}$$

It is assumed that when $f_k = f_k^i$, $u_{i,j_1} - u_{i,j} = 0$. As f_k increases, i.e., $f_k = f_k^i + \Delta f_k$, $\Delta f_k > 0$, it is obtained that $u_i^u - u_{i,j} = u_{i,j_1} - u_{i,j} = a_d a_i \Delta f_k$.

Therefore, $a_i a_d$ is the criterion for judging whether choosing $\mathbf{w}_{j_1}$ as $\mathbf{w}_i^u$ is correct: if $a_i a_d \geq 0$, then $u_i^u \geq u_{i,j}$, the selection is correct; otherwise, it is incorrect. Let us rewrite $a_i a_d$ as $a_i a_d(\mathbf{w}_i^u, \mathbf{w}_i^l, \mathbf{w}_j)$, because it is a function of them. If there are a number of such

vertices, they form a set W^u. If the case takes place with the $\mathbf{w}_i^l$, i.e., there is more than one $\mathbf{w}_j$, which form a set W^l, such that $u_i^l = \bar{\mathbf{s}}_i \mathbf{w}_j$, the selection procedure is the same but the criterion should be $a_i a_d(\mathbf{w}_i^u, \mathbf{w}_i^l, \mathbf{w}_j) \leq 0$, because we need $u_i^l \leq u_{i,j}$. We have a procedure to select a pair of $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ from W^u and W^l.

(1) take a pair of vertices, one from W^u as $\mathbf{w}_i^u$ and the other from W^l as $\mathbf{w}_i^l$.

(2) check $a_i a_d(\mathbf{w}_i^u, \mathbf{w}_i^l, \mathbf{w}_j) \geq 0$, $\forall \mathbf{w}_j$, $\mathbf{w}_j \in W^u$, if any are false, choose another vertex from W^u as u_i^u, and redo (2).

(3) check $a_i a_d(\mathbf{w}_i^u, \mathbf{w}_i^l, \mathbf{w}_j) \leq 0$, $\forall \mathbf{w}_j$, $\mathbf{w}_j \in W^l$, if any are false, choose another vertex from W^l as u_i^l, and redo (2).

(4) replace the old $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ with the last pair from (2) and (3).

A procedure for deriving f_i^a, $i < k$, as a piecewise function of f_k is similar to the above with a little modification.

A.3 Delimit f_k According to Dependencies

We know $\frac{1}{f_0^a}, \ldots, \frac{1}{f_M^a}$ are the lengths of the clustering parallelepiped. As commanded above, the parallelepiped should be large enough to enclose all of the dependencies. To do so, we should have, according to eqn (4.6)

$$f_{k,i}^c = \frac{1 - b_i a_i^{max}}{a_i a_i^{max}} \qquad \forall i,\ 0 \leq i < n$$

Note that because f_i^a is a piecewise function of f_k, say n_p pieces, $f_{k,i}^c$ should be evaluated in this way: for each pair of a_i and b_i of all pieces, calculate $f_{k,i}^c$; the $f_{k,i}^c$ is valid if it is within the f_k's interval in which this pair of a_i and b_i is defined, and is invalid if outside. So, more generally, $f_{k,i}^c$ is rewritten as

$$f_{k,i,j}^c = \frac{1 - b_{i,j} a_i^{max}}{a_{i,j} a_i^{max}} \tag{A.14}$$

where $0 \leq j < n_p$, $a_{i,j}$ and $b_{i,j}$ is the a_i and b_i of the j-th piece, respectively. f_k is delimited by all the valid $f_{k,i,j}^c$. Obviously, $f_{k,i,j}^c$ is related to the upper bound of f_k if $a_{i,j} > 0$, and

to the lower bound if $a_{i,j} < 0$. Therefore [6]

$$\begin{aligned} f_k^{ub} &= \min_{0 \le i < n,\ 0 \le j < n_p} \{f_{k,i,j}^c \cap (a_{i,j} > 0)\} \\ f_k^{lb} &= \max_{0 \le i < n,\ 0 \le j < n_p} \{f_{k,i,j}^c \cap (a_{i,j} < 0)\} \end{aligned} \tag{A.15}$$

If $f_k^{ub} < f_k^{lb}$, no f_k is valid. This means that these dependency vectors are too long to find a parallelepiped enclosing all of them and keeping the property of mapping the polyhedron just onto the given array. In this case, if $f_k^{ub} < f_k^{lb}$ is made in the i-th dimension, we have to use a_i^{max} as the length of the i-th edge of the clustering parallelepiped, with a cost of wasting the processor resource.

[6] Let invalid $f_{k,i,j}^c$ be null

Appendix B

Collection of Algorithms for (N-1)-D Partitioning and Mapping

It will be helpful to make a summary by representing the method with a series of algorithms.

B.1 Pre-compilation Work

Algorithm B.1.1 *Computing the timing vector sets for Strategy 4.4.1*

FOR N = 2, ...
  create N! permutation matrices $\mathbf{P}_{Mn,i}$, forming a set of P_{Mn}.
  FOR i = 0, n!-1
    FOR j = i+1, n!-1
      delete $\mathbf{P}_{Mn,j}$ if $\mathbf{P}_{Mn,j} = \mathbf{J}\mathbf{P}_{Mn,i}$.
  create N-D $\mathbf{S}_{SBC}$ and $\bar{\mathbf{t}}_{SBC}$.
  FOR i = 0, $\frac{n!}{2}$-1
    produce a permuted version $\mathbf{S}_i$ by $\mathbf{S}_{SBC}\mathbf{P}_{Mn,i}$, $\mathbf{P}_{Mn,i} \in P_{Mn}$.
    WHILE(all possible $\bar{\mathbf{t}}$ not done){
      change $\bar{\mathbf{t}}$ such that $\sum_{j=0}^{M} t_j = g^M$ and $t_j \geq \bar{\mathbf{t}}_{SBC}\mathbf{P}_{Mi}\vec{\mathbf{i}}_j^T$. $g = 2, 3$.
      calculate the activity matrix $\mathbf{A}$ with $\mathbf{S}_i$ and $\bar{\mathbf{t}}$.
      evaluate M! HNFs of $\mathbf{A}$.
      put the $\bar{\mathbf{t}}$ into a set $\overline{T}_{n,i}$ if any of the HNFs has equal diagonal entries. }
  FOR i = 0, $\frac{n!}{2}$-1
    $\mathbf{P}_{Mn,\frac{n!}{2}+i} = \mathbf{J}\mathbf{P}_{Mn,i}$.
    $\overline{T}_{n,\frac{n!}{2}+i} = \overline{T}_{n,i}$.
End of Algorithm

B.2 Compiling Work

Given dependency $\mathbf{D}$, interconnection primitives $\mathbf{P}$, polyhedron vertices $\mathbf{V}$ and processor array size $l_0 \times \ldots \times l_{M-1}$.

Algorithm B.2.1 *Main Algorithm: Produce* $\mathbf{B}$, $\mathbf{T}$ *and* $\mathbf{k}$

derive $\mathbf{E}$ from $\mathbf{D}$ by means of Theorem 4.2.1.
yield $\mathbf{E}$ by normalising each columns of $\mathbf{E}$.
perform $\mathbf{A}_p := \mathbf{E}^{-1}\mathbf{D}$.
FOR i = 0, M
  evaluate $a_i^{max} := \max_{0 \le j < m} a_{i,j}$, $a_{i,j} \in \mathbf{A}_p$.
$\mathbf{W} := \mathbf{E}^{-1}\mathbf{V}$.
IF $\mathbf{P}$ belong to $\mathbf{P}_{SUC}$, CALL **Algorithm** B.2.2.
IF $\mathbf{P}$ belong to $\mathbf{P}_{SBC}$, CALL **Algorithm** B.2.3.
$\mathbf{B}_s := \mathbf{E}\mathbf{F}^{-1}$
CALL **Algorithm** B.2.5
End of Algorithm

Algorithm B.2.2 *Real Parallelepiped and S-T Transformation for SUC*

$\mathbf{L} := diag(+\infty, l_0, \ldots, l_{M-1})$
FOR i = 0, M
  $w_i^{u-l} := \max_{0 \le j < n_v} w_{i,j} - \min_{0 \le j < n_v} w_{i,j}$, $w_{i,j} \in \mathbf{W}$.
$\mathbf{w} := [\frac{1}{w_0^{u-l}}, \ldots, \frac{1}{w_M^{u-l}}]^T$.
FOR i = 0, n!-1
  evaluate $\mathbf{F}^a$ with eqn (4.40)
  FOR j = 0, M
    $f_j := min(f_j^a, \frac{1}{a_i^{max}})$, $f_j^a \in \mathbf{F}_i^a$.
  $\mathbf{F}_i^a := diag[f_0^a, \ldots, f_M^a]$.
  $\mathbf{V}^q := \mathbf{F}_i^a \mathbf{W}$.
  $t_i^{com} := \max_{0 \le j < n_v} \bar{\mathbf{t}}_{SUC}\mathbf{v}_j^q - \min_{0 \le j < n_v} \bar{\mathbf{t}}_{SUC}\mathbf{v}_j^q$, $\mathbf{v}_j^q \in \mathbf{V}^q$.
$t^{min} := \min_{0 \le i < n!-1} t_i^{com}$.
$k := \{i : t_i^{com} = t^{min}\}$.
$\mathbf{F}^a := \mathbf{F}_k^a$.
$\mathbf{S} := \mathbf{S}_{SUC}\mathbf{P}_{Mk}$.
$\bar{\mathbf{t}} := \bar{\mathbf{t}}_{SUC}$.
End of Algorithm

Algorithm B.2.3 *Real Parallelepiped and S-T Transformation for SBC*

$\mathbf{W}^o := \mathbf{W}$
FOR i = 0, M
  $l_k := gl_k$
FOR i = 0, n!-1 (or $\frac{n!}{2}$ -1 if $l_0 = \ldots = l_{M-1}$)
  $\mathbf{W} := \mathbf{P}_{Mi}\mathbf{W}^o$

FOR k = 1, M-1

CALL **Algorithm** B.2.4 with f_k as argument.

$f_k^{max} = \{f_k : maximise \prod_{q=0}^{M} f_q^a\}$

$p_k^{max} = \prod_{q=0}^{M} f_q^a \mid_{f_k = f_k^{max}}$

$k^{max} = \{k : p_k^{max} = \max_{0 \le q < M} p_q^{max}\}$

$p_i^{max} = p_{k^{max}}^{max}$

$\mathbf{F}_i^a = diag(f_0, \ldots, f_M) \mid_{f_k = f_{k^{max}}^{max}}$

$\mathbf{F}_i^a := \mathbf{P}_{Mi}^{-1} \mathbf{F}_i \mathbf{P}_{Mi}$.

$\mathbf{V}^q := \mathbf{F}_i^a \mathbf{W}^o$.

evaluate t_i^{op} by eqn (4.46)

determine $\overline{\mathbf{t}}_i^{op}$ by eqn (4.47)

$t_i^{min} := \frac{t_i^{op}}{p_i^{max}}$

$i^{op} = \{i : t_i^{min} = \min_{0 \le p < n!-1} t_p^{min}\}$

$\mathbf{F}^a := \mathbf{F}_{i^{op}}^a$.

$\overline{\mathbf{t}} = \overline{\mathbf{t}}_{i^{op}}^{op}$

$\mathbf{S} := \mathbf{S}_{SBC} \mathbf{P}_{Mi^{op}}$.

End of Algorithm

Algorithm B.2.4 *Deriving* $\mathbf{F}^a$ *as a Function of a Single Argument* f_k.

$f_k := 0$

FOR i = k, M-1

$f_{i+1}^i := 0$

WHILE(f_{i+1}^a not found){

determine $\mathbf{w}_i^u$ and $\mathbf{w}_i^l$ by eqn(A.3)

evaluate turning point f_{i+1}^t with eqn (A.4) and (A.5).

evaluate f_{i+1}^a by eqn (A.6)

break WHILE loop if $f_{i+1}^i \le f_{i+1}^a < f_{i+1}^t$.

$f_{i+1}^i := f_{i+1}^t otherwise.\}$

FOR i = k, 1, -1

similar way to derive f_{i-1}^a, as well as $\mathbf{w}_{i-1}^u$ and $\mathbf{w}_{i-1}^l$.

$n_p := 0$. (n_p is the number of the segments of the piecewise function)

$f_k^i := 0$

WHILE(all $f_0^a, \ldots, f_M^a$ positive){

$a_{k,n_p} = 1$

$b_{k,n_p} = 0$

FOR i = k, M-1

evaluate a_{i+1,n_p} and b_{i+1,n_p} with eqn (A.8) and eqn ((A.9)

FOR i = k, 1, -1

similar way to evaluate a_{i-1,n_p} and b_{i-1,n_p}.

FOR i = 0, M-1

FOR j = 0, n_v

evaluate $f_{k,i,j}^u$ and $f_{k,i,j}^l$ with eqn (A.10).

evaluate f_k^t with eqn (A.11)

$f_{n_p}^{ub} := f_k^i$. ($f_{n_p}^{ub}$ is the upper boundary of the segment)

$f^{lb}_{n_p} := f^t_k$. ($f^{lb}_{n_p}$ is the lower boundary of the segment)
break WHILE loop if any $a_{i,n_p} f^t_k + b_{i,n_p} < 0$.
FOR i = 0, M-1
$\mathbf{w}^u_i := \mathbf{w}_j$ if $f^u_{k,i,j} = f^t_k$.
$\mathbf{w}^l_i := \mathbf{w}_j$ if $f^l_{k,i,j} = f^t_k$.
$f^i_k := f^t_k$.
$n_p := n_p + 1$.}
FOR i = 0, M
FOR j = 0, $n_p - 1$
$f^c_{k,i,j} = \frac{1 - b_{i,j} a^{max}_i}{a_{i,j} a^{max}_i}$.
$f^c_{k,i,j} := null$ if $(f^{lb}_j > f^c_{k,i,j}) \cup (f^c_{k,i,j} > f^{ub}_j)$
determine f^{ub} and f^{lb} with eqn (A.15)
End of Algorithm

Algorithm B.2.5 *Integralization* $\mathbf{B}_s$

FOR i = 0, M
$\mathbf{B}_{-i} := \mathbf{B}_s$ deleted the i-th column.
solve $\mathbf{B}^T_{-i}\mathbf{h}_{-i} = \mathbf{o}$ for $\mathbf{h}_{-i}$.
FOR i = 0, M
$\mathbf{H}_i = [\mathbf{h}_{-0}, \ldots, \mathbf{h}_{-(i-1)}, \mathbf{h}_{-(i+1)}, \ldots, \mathbf{h}_{-(M)}, \mathbf{b}_i]$, $\mathbf{b}_i \in \mathbf{B}_s$.
FOR i = 0, M
FOR j = 0, M
$\mathbf{H}_{i,-j} := \mathbf{H}_i$ deleted the j-th column.
solve $\mathbf{H}^T_{i,-j}\mathbf{c}_{i,j} = \mathbf{o}$ for $\mathbf{c}_{i,j}$.
$\mathbf{C}_i := [\mathbf{c}_{i,0}, \ldots, \mathbf{c}_{i,M}]$
modify the direction of $\mathbf{c}_{i,j}$ such that $\mathbf{a} := \mathbf{C}^{-1}_i\mathbf{j}$ is non-positive, $\mathbf{j}$ is in the cone.
find integral vectors $\mathbf{j}_j$ around the top of $\mathbf{b}_i$, j = 0,..., 2^M.
FOR j = 0, 2^M
$\mathbf{a} := \mathbf{C}^{-1}_i(\mathbf{j}_j - \mathbf{b}_i)$
put $\mathbf{j}_j$ into a set J_i if $\mathbf{a}$ is non-negative.
re-arrange the elements of J_i according to their distances to $\mathbf{b}_i$).
WHILE(valid $\mathbf{B}$ not found){
$\mathbf{B} := [\mathbf{j}_0, \ldots, \mathbf{j}_i, \ldots, \mathbf{j}_M]$, $\mathbf{j}_i \in J_i$ and selected differently every times.
$\mathbf{U} := \mathbf{S}\mathbf{B}^{-1}\mathbf{V}$
break WHILE loop if $\mathbf{U}$ is acceptable.}
End of Algorithm

Appendix C

Collection of Algorithms for Lower-Dimensional Mapping

Algorithm C.1 *Feature of Polyhedron*

FOR i=0, $n^f - 1$ (n^f is the number of the facets of the $\mathcal{P}^{K-r}$)
Find all vertices associated with the i-th face, forming a set V_i
$f_{r,i}^{up} := \max_{j=0}^{n_{V_i}-1} v_{r,j}$ (n_{V_i} is the cardinality of V_i, $v_{r,j} \in \mathrm{v}_j \in V_i$)
$b_r^l = max(b_r^l, f_{r,i}^{up})$
$f_{r,i}^{low} := \min_{j=0}^{n_{V_i}-1} v_{r,j}$
$b_r^f = min(b_r^f, f_{r,i}^{low})$
Collect all pair of vertices from all V_i's, creating a set of possible edges, E
Remove the terms which appears only once in E, and merge the terms which are the same. (e.g., $\{(0,0),(1,1),(1,1)\} \Longrightarrow \{(1,1)\}$)
FOR i=0, $n^e - 1$ (n^e is the cardinality of E)
$max_{i,r}^e := max(v_{r,i}^1, v_{r,i}^2)$ ($v_{r,i}^1 \in \mathrm{v}_i^1$, $v_{r,i}^2 \in \mathrm{v}_i^2$, v_i^1 and $v_{r,i}^2$ is the i-th edge in E)
$min_{i,r}^e := min(v_{r,i}^1, v_{r,i}^2)$
End of Algorithm

Algorithm C.2 *Intersect Polyhedron (from $\mathcal{P}^{K-r}$ to $\mathcal{P}^{K-r-1}$ with h_r)*

FOR i=0, $n^f - 1$
if $f_{r,i}^{low} \le h_r \le f_{r,i}^{up}$, put the i-th face to the set of the faces of $\mathcal{P}^{K-r-1}$, but modifying the constant term $a_K := a_K + a_r h_r$.
FOR i=0, $n^e - 1$ (n^e is the number of the edges of the $\mathcal{P}^{K-r}$)
if $min_{i,r}^e \le h_r \le max_{i,r}^e$, evaluate a vertex by eqn (5.5) to the vertex set of $\mathcal{P}^{K-r-1}$.
CALL Algorithm C.1, finding edges and some feature of $\mathcal{P}^{K-r-1}$
End of Algorithm

Algorithm C.3 *Determine* $\mathbf{f}^{(K-r)}$

FOR i = r, K-1
$f_i^{(K-r)} := \lceil b_i^f \rceil$

DO {

CALL Algorithm C.2 to intersect the $\mathcal{P}^{K-r}$ with $i_i = f_i^{(K-r)}$, producing $\mathcal{P}^{K-i-1}$

}WHILE(If, for $\mathcal{P}^{K-i-1}$, $\lceil b_{i+1}^f \rceil \leq b_{i+1}^l$, break; otherwise $f_i^{(K-r)} := f_i^{(K-r)} + 1$)

End of Algorithm

Algorithm C.4 *Determine* $\mathbf{l}^{(K-r)}$

Similar to Algorithm C.3

Algorithm C.5 *Build* $\mathcal{SP}_{i+1}$ *from* $\mathcal{SP}_i$

FOR j = 0, $n_{SP_i} - 1$ (n_{SP_i} is the cardinality of $\mathcal{SP}_i$

$\mathcal{P}^{K-i} := \mathcal{P}_j^{K-i} \in \mathcal{SP}_i$

FOR r = 0, $n_v - 1$ (n_v is the number of the vertices of the $\mathcal{P}^{K-i}$)

$h := v_i$ (v_i is the first element of $\mathbf{v}_r$ of the vertices of $\mathcal{P}^{K-i}$)

if the h does not appear before, CALL Algorithm C.2 to intersect $\mathcal{P}^{K-i}$ with h, producing a $\mathcal{P}_s^{(K-i-1)}$, put into $\mathcal{SP}_{i+1}$

End of Algorithm

Algorithm C.6 *Determine* p_i

FOR j = 0, $n_{SP_i} - 1$ (n_{SP_i} is the cardinality of $\mathcal{SP}_i$

$\mathcal{P}^{K-i} := \mathcal{P}_j^{K-i} \in \mathcal{SP}_i$

FOR r = 0, $n_v - 1$ (n_v is the number of the vertices of the $\mathcal{P}^{K-i}$)

$h := v_i$ (v_i is the first element of $\mathbf{v}_r$ of the vertices of $\mathcal{P}^{K-i}$)

FOR s=-1, 1

CALL Algorithm C.2 to intersect $\mathcal{P}^{K-i}$ with $h + s$, producing $\mathcal{P}_s^{(K-i-1)}$.

CALL Algorithm C.3 to determine $\mathbf{f}_l^{(K-i-1)}$

CALL Algorithm C.4 to determine $\mathbf{l}_l^{(K-i-1)}$

$d^+ := \overline{\mathbf{p}}(\mathbf{l}_0^{(K-i-1)} - \mathbf{f}_1^{(K-i-1)}) + 1$

$d^- := \overline{\mathbf{p}}(\mathbf{l}_{-1}^{(K-i-1)} - \mathbf{f}_0^{(K-i-1)}) + 1$

$p_i := \max(p_i, d^+, d^-)$;

End of Algorithm

Algorithm C.7 *Main Algorithm: Find a Minimum Valid Projecting Vector*

$\mathbf{P}_t := \mathbf{\Pi W}$

Compute the Convex Hull, $\mathcal{CH}^K$, of $\mathbf{V}_t$ (by means of an algorithm [98] and a program [65];

$\overline{\mathbf{p}} := [0, \ldots, 0, 1]$

$\mathcal{SP}_0 := \mathcal{P}^K := \mathcal{CH}^K$

FOR i = 0, K-2

CALL Algorithm C.5 to build $\mathcal{SP}_{i+1}$ from $\mathcal{SP}_i$.

FOR i = K-2, 0, i := i-1

CALL Algorithm C.6 to derive p_i from $\mathcal{SP}_i$.

End of Algorithm

Appendix D

Parallel Algorithm for Pure LSGP Method

From the viewpoint of algorithm generation, the pure LSGP can be taken as the special case of $\mathbf{B} = \mathbf{I}$ in the LSGP case of Chapter 6. Omitting derivation, the parallel algorithm obtained is

DOALL $j_0 := -10$ TO 20 STEP 8
  DOALL $j_1 := -10$ TO 20 STEP 8
    $\mathbf{w} := \mathbf{j}$
    FOR $t_0 := 0$ TO 1230
      $t_0 := \lceil \frac{w_0}{8} \rceil$
      $t_1 := \lceil \frac{w_1 - t_0}{8} \rceil$
      $\mathbf{w} := \mathbf{w} + \begin{bmatrix} 5 \\ 4 \end{bmatrix}$
      $\mathbf{w}' := 8\mathbf{t} - \mathbf{w}$
      $i_0 := -2t_0 + 3t_1$
      $i_1 := 5t_0 + -5t_1 - t_2$
      $i_2 := -3t_0 + -5t_1 - 4t_2$
      IF $(0 \leq i_0 \leq 20) \bigcap (0 \leq i_0 \leq 20) \bigcap (0 \leq i_0 \leq 10)$
        $A(i_0, i_1, i_2) := A(i_0 - 1, i_1 + 1, i_2 + 3) + A(i_0, i_1 - 1, i_2 - 1) +$
        $+ A(i_0, i_1 - 1, i_2 + 1) + A(i_0, i_1 - 1, i_2)$
        Data transfer operation

Again, it is checked that the parallel algorithm covers all the 4851 nodes. The Data transfer operation can also be taken as a special case of that of (N-1)-D SBC case discussed in Chapter 7, remembering that a supernode is just one node.

Appendix E

Generating Data Flows and Relays for LSGP

A OP (or IP) may contain a few of data vectors, "$V()$", which has a number of information fields: .n is the number of nodes of the $V()$; .fb and .fbp indicate which buffer the $V()$ comes from and the start position in that buffer respectively; .tb and .tbp indicate the which buffer the $V()$ goes to and the start position in that buffer, respectively; .dep indicates the processor dependency.

Algorithm E.1 *Relation* $\mathbf{w}'$*-*$t(\Delta\mathbf{a})$

$\mathbf{w}'' := k\Delta\mathbf{a}$.

FOR $t := 0$ TO $c-1$

FOR i :=0, N-2

$w_i := w_i'' - \sum_{j=0}^{i-1} h_{i,j}t_j$

$t_i := \lceil \frac{w_i}{k} \rceil$

$\mathbf{w}'' := \mathbf{w}'' - \mathbf{r}$

$\mathbf{w}' := k\mathbf{t} - \mathbf{w}$

mark $\mathbf{w}'$ with t

(comment: $h_{i,j}$ and $\mathbf{r}$ are defined in Section 6.5)

End of Algorithm

Algorithm E.2 $CreateIP_{\mathbf{p}'}^{w_{j'}}(OP_{\mathbf{p}}^{w_j})$

$j' := j + \overline{\Delta \mathbf{w} \mathbf{p}} \bmod c$

$\mathbf{p}' := -\mathbf{p}$

$n_V := 0$

FOR $i := 0$ TO $OP_{\mathbf{p}}^{w_j}.n_V - 1$

$IP_{\mathbf{p}'}^{w_{j'}}.V(n_V).dep := OP_{\mathbf{p}}^{w_j}.V(i).dep$

$IP_{\mathbf{p}'}^{w_{j'}}.V(n_V).n := OP_{\mathbf{p}}^{w_j}.V(i).n$

IF only one non-zero entry in $OP_{\mathbf{p}}^{w_j}.V(i).dep$

$IP_{\mathbf{p}'}^{w_{j'}}.V(n_V).tb := IB^{w_{j'}}$

$IP_{\mathbf{p}'}^{w_{j'}}.V(n_V).tbp := IB^{w_{j'}}.n$

$IB^{w_{j'}}.n := IB^{w_{j'}}.n + OP_{\mathbf{p}}^{w_j}.V(i).n$

ELSE

$IP_{\mathbf{p}'}^{w_{j'}}.V(n_V).tb := RB_{\mathbf{p}}$

$IP_{\mathbf{p}'}^{w_{j'}}.V(n_V).tbp := 0$

(comment: if there is only one non-zero entry in $OP_{\mathbf{p}}^{w_j}.V(i).dep$, the data vector has arrived at its terminal, so it is put into OB; otherwise, into a relay buffer)

End of Algorithm

Algorithm E.3 *The data flows and relays for LSGP*

Relationw'-t(**0**)

Mark every $\mathbf{w}'$ with w_t,

FOR $i := 0$ TO M-1

Relationw'-t($\mathbf{1}_i$) comment[1]

Taking the t from any $\mathbf{w}'$marked as w_j, let $\Delta w_i := t - j \bmod c$

FOR $j := 0$ TO $c - 1$

Take the corresponding $\mathbf{w}'$ of the w_j

For all $\mathbf{d}^p \in \mathbf{D}^p$

$\mathbf{d}^A := \mathbf{d}^p \otimes \mathbf{l}(\mathbf{w}')$ comment[2]

IF $\mathbf{d}^A \in \mathbf{D}^A$

$\mathbf{p} := \mathbf{d}^A$

Set "0" to all entries of $\mathbf{p}$ except the first non-zero entry

$n_V := OP_{\mathbf{p}}^{w_j}.n_V$ comment[3]

$OP_{\mathbf{p}}^{w_j}.V(n_V)dep := \mathbf{d}^A$

$OP_{\mathbf{p}}^{w_j}.V(n_V).fb := OB$

$OP_{\mathbf{p}}^{w_j}.V(n_V).fbp := OP_{\mathbf{p}}^{w_j}.V(n_V - 1).fbp - OP_{\mathbf{p}}^{w_j}.V(n_V - 1).n$

$OP_{\mathbf{p}}^{w_j}.V(n_V).n := Q^{\mathrm{d}^p}$

$OP_{\mathbf{p}}^{w_j}.n_V := OP_{\mathbf{p}}^{w_j}.n_V + 1$

For all existing $OP_{\mathbf{p}}^{w_j}$'s

$CreateIP_{\mathbf{p}'}^{w_{j'}}(OP_{\mathbf{p}}^{w})$

FOR $i := 1$ TO $M - 2$

For all existing $IP_{\mathbf{p}'}^{w_{j'}}.V()$'s

IF $IP_{\mathbf{p}'}^{w_{j'}}.V().tb$ is any $RB_{\mathbf{p}''}$

$\mathbf{p} := IP_{\mathbf{p}'}^{w_{j'}}.V().dep$

Null all entries of $\mathbf{p}$ except the i-th one

IF $\mathbf{p} \neq \mathbf{0}$

$w := w' + 1 \bmod c$

$n_V := OP_{\mathbf{p}}^{w_j}.n_V$

$OP_{\mathbf{p}}^{w_j}.V(n_V).dep := IP_{\mathbf{p}'}^{w_{j'}}.V().dep$

$OP_{\mathbf{p}}^{w_j}.V(n_V).fb := RB_{\mathbf{p}''}$

$OP_{\mathbf{p}}^{w_j}.V(n_V).fbp := 0$

$OP_{\mathbf{p}}^{w_j}.V(n_V).n := IP_{\mathbf{p}'}^{w_{j'}}.V().n$

$OP_{\mathbf{p}}^{w_j}.n_V := n_V + 1$

$CreateIP_{\mathbf{p}'}^{w_{j'}}(OP_{\mathbf{p}}^{w})$

Re-order OP's and IP's according to the ordinal number w

(comment[1]: $\mathbf{1}_i$ is a vector such that only the i-th element is "1", and others are "0".

$\Delta w_i \in \overline{\Delta \mathbf{w}}$)

(comment[2]: see Subsection 6.5.3 for the operator $\otimes$ and function $\mathrm{l}(\cdots)$.)

(comment[3]: field $.n_V$ indicates the number of data vector of a packet. This segment makes out the initial OP's.)

End of Algorithm

Appendix F

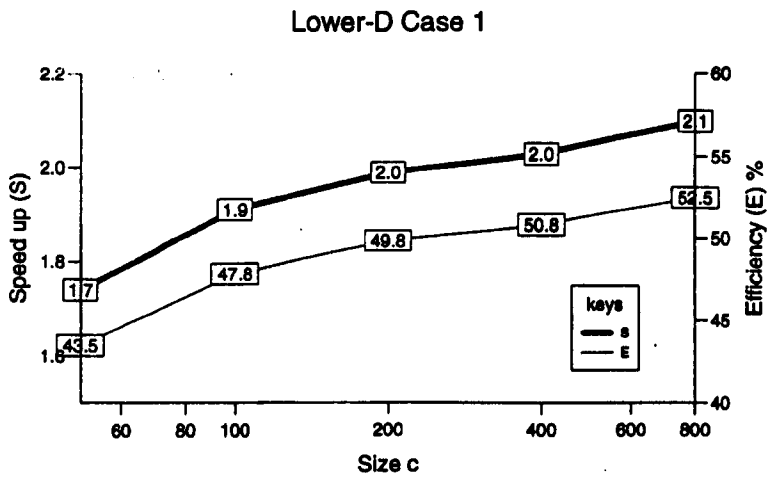
The Collection of Experimental Results

1-D array with SUC mesh

In the following experiments, a linear array consisting of N_p processors is used.

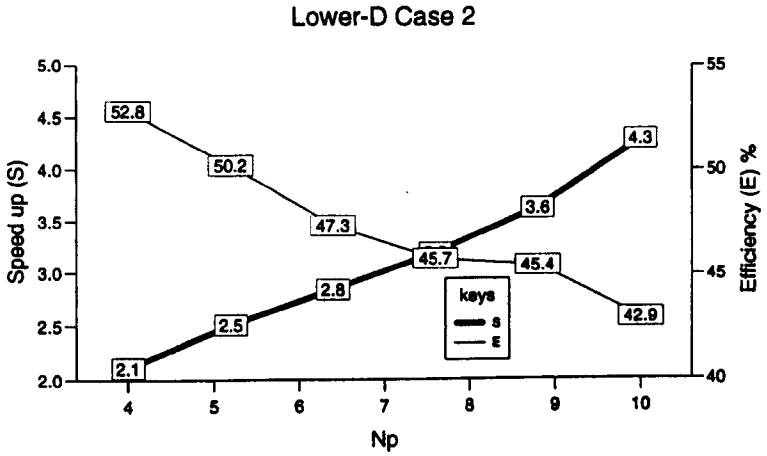
$$\text{Case lower-D-1: } \mathbf{c} = \begin{bmatrix} 20 \\ 20 \\ c \end{bmatrix} \quad \mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix} \quad N_p = 4$$

c	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	204.3	372.6	715.4	1401.8	2714.5



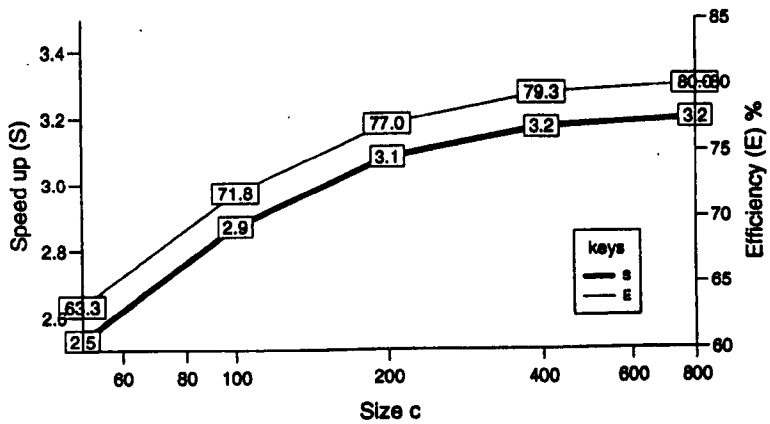
Case lower-D-2: $\mathbf{c} = [40, 40, 200]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-1

N_p	4	5	6	7	8	10
T_s(ms)	5692.8	5692.8	5692.8	5692.8	5692.8	5692.8
T_p(ms)	2696.7	2266.6	2006.3	1781.4	1570.2	1326.1



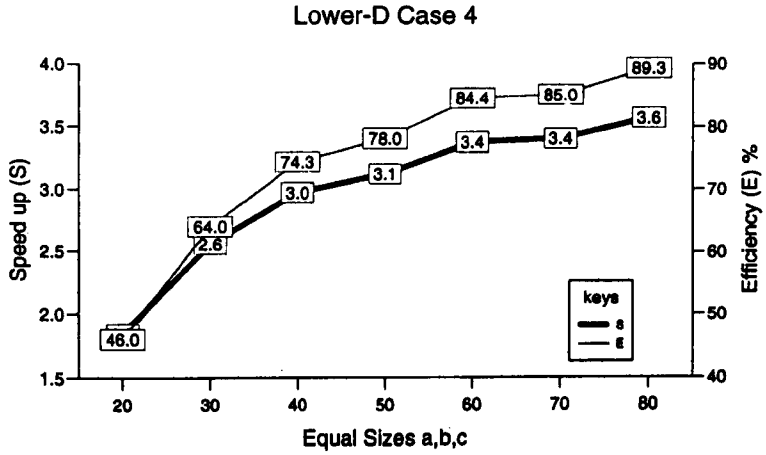
Case lower-D-3: $\mathbf{c} = \begin{bmatrix} 20 \\ 20 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 \end{bmatrix}$ $N_p = 4$

c	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	140.8	248.3	461.8	897.6	1778.7



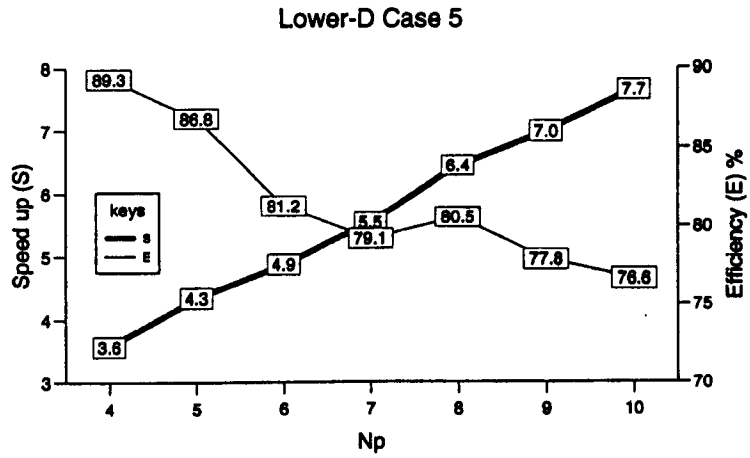
Case lower-D-4: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-3 $N_p = 4$

$a = b = c$	20	30	40	50	60	70	80
T_s(ms)	142.3	480.3	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	77.4	196.0	382.9	713.4	1138.7	1793.2	2553.6



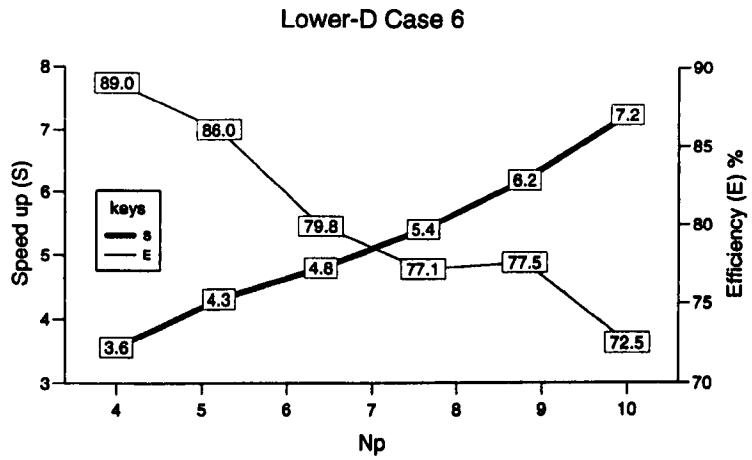
Case lower-D-5: $\mathbf{c} = [80, 80, 80]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-3

N_p	4	5	6	7	8	9	10
T_s(ms)	9108.5	9108.5	9108.5	9108.5	9108.5	9108.5	9108.5
T_p(ms)	2553.6	2097.2	1872.2	1644.0	1414.5	1300.9	1188.6



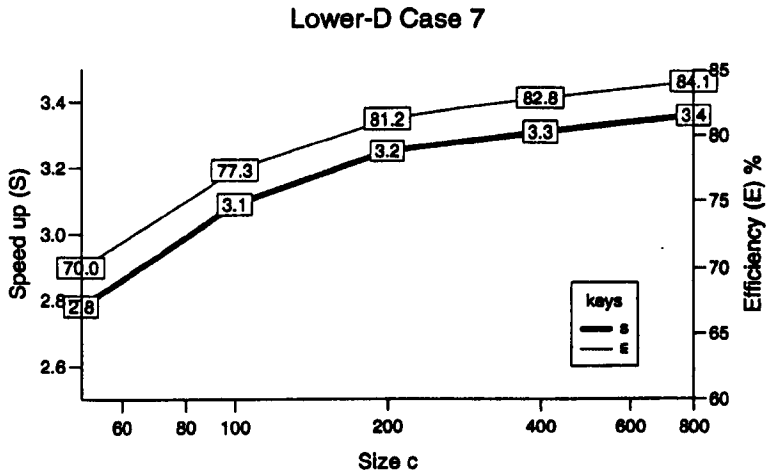
Case lower-D-6: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ 200 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$

N_p	4	5	6	7	8	10
T_s(ms)	5692.8	5692.8	5692.8	5692.8	5692.8	5692.8
T_p(ms)	1596.9	1323.6	1188.6	1053.2	917.3	785.5



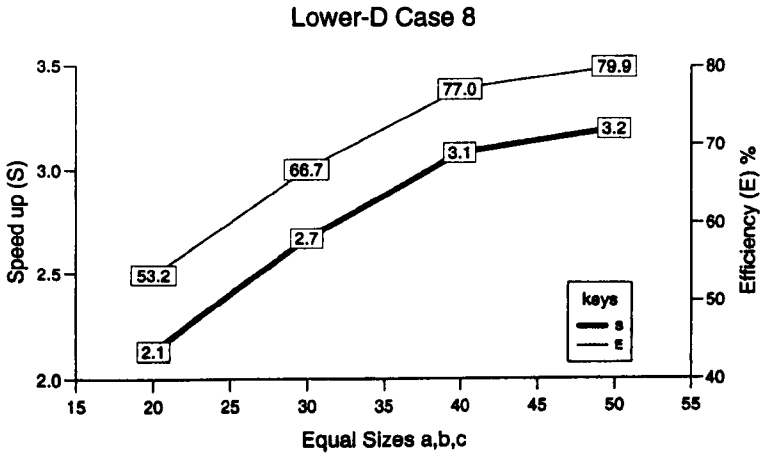
Case lower-D-7: $\mathbf{c} = [40, 40, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-6, $N_p = 4$

c	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	127.9	230.1	438.1	859.7	1692.5



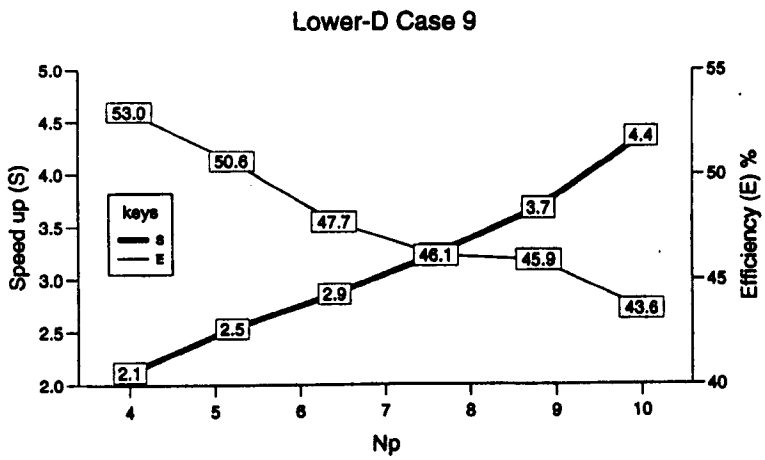
Case lower-D-8: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-6, $N_p = 4$

$a = b = c$	20	30	40	50
T_s(ms)	142.3	480.3	1138.6	2223.8
T_p(ms)	66.9	180.0	369.5	696.0



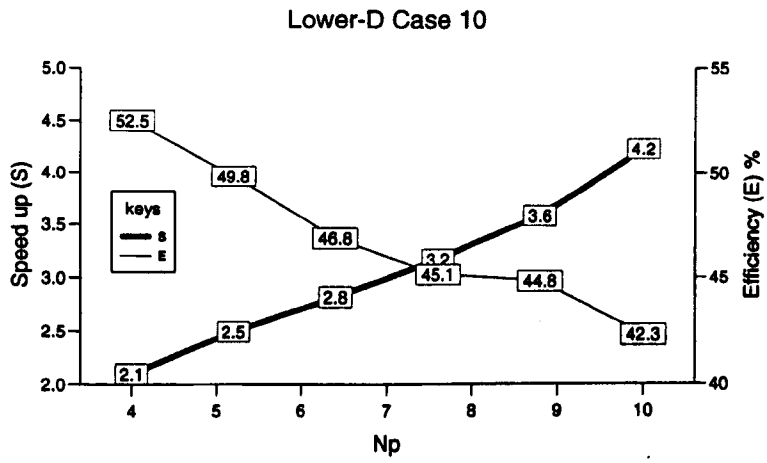
Case lower-D-9: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ 200 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$

N_p	4	5	6	7	8	10
T_s(ms)	5692.8	5692.8	5692.8	5692.8	5692.8	5692.8
T_p(ms)	2683.5	2250.6	1989.4	1763.4	1550.6	1305.3



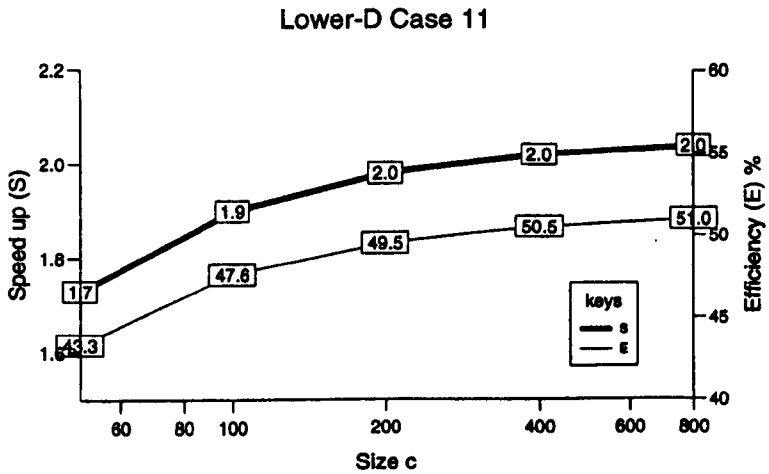
Case lower-D-10: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ 200 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$

N_p	4	5	6	7	8	10
T_s(ms)	569.3	56.9	5692.8	5692.8	5692.8	5692.8
T_p(ms)	2715.1	2284.2	2025.5	1800.9	1588.6	1345.7



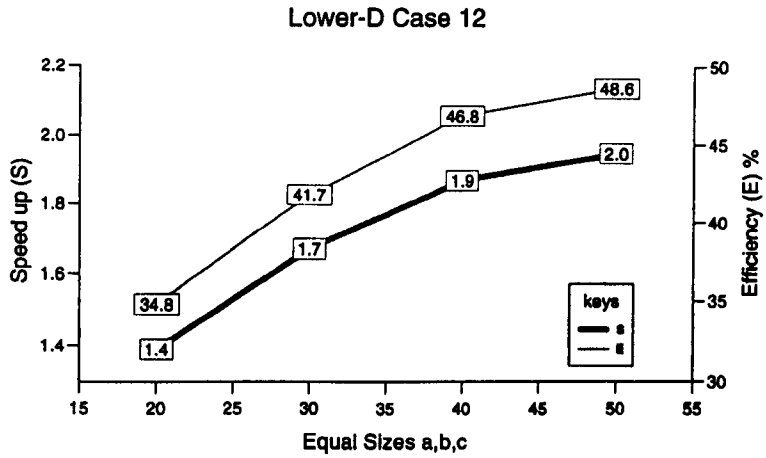
Case lower-D-11: $\mathbf{c} = [20, 20, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-10, $N_p = 10$

c	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	205.4	374.0	718.8	1409.2	2782.7



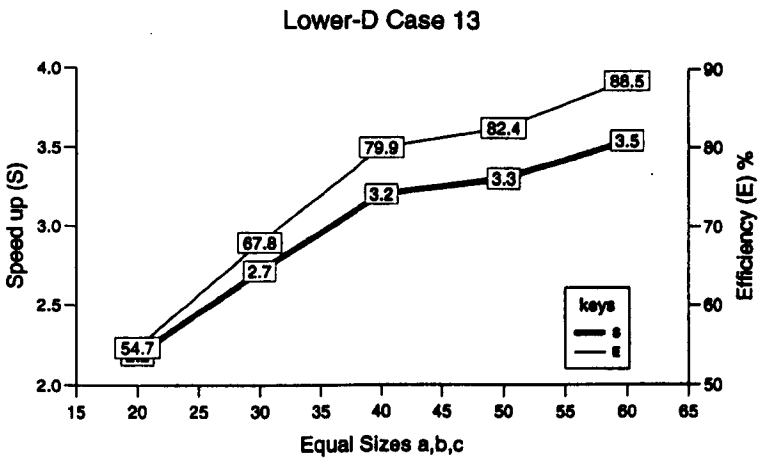
Case lower-D-12: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 4$

$a = b = c$	20	30	40	50
T_s(ms)	142.3	480.3	1138.6	2223.8
T_p(ms)	102.2	287.6	608.6	1142.8



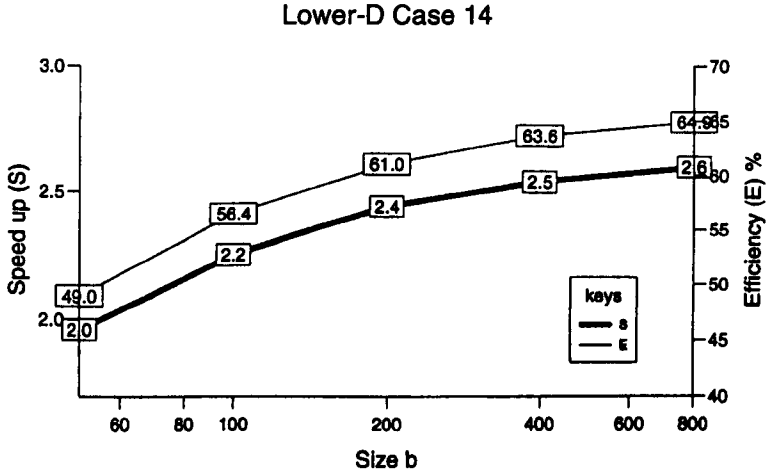
Case lower-D-13: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 4$

$a = b = c$	20	30	40	50	60
T_s(ms)	142.3	480.3	1138.6	2223.8	3842.6
T_p(ms)	65.1	177.0	356.2	674.9	1085.9



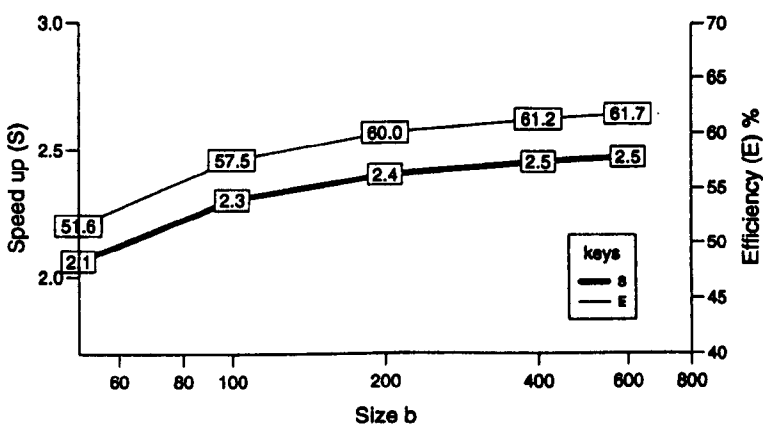
Case lower-D-14: $\mathbf{c} = \begin{bmatrix} 20 \\ b \\ 20 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ -1 & 0 & 1 & 1 \end{bmatrix}$ $N_p = 4$

b	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	28748.6	57497.3
T_p(ms)	181.5	315.6	583.5	1119.6	2191.7



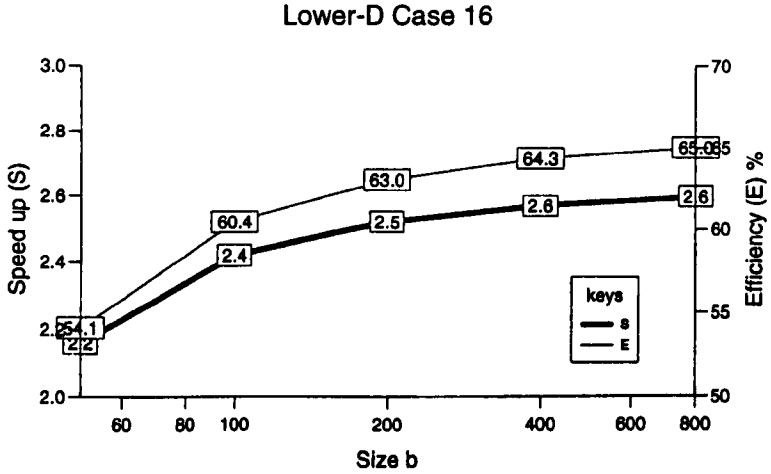
Case lower-D-15: $\mathbf{c} = \begin{bmatrix} 20 \\ b \\ 20 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ -3 & 1 & -2 & 0 \end{bmatrix}$ $N_p = 4$

b	50	100	200	400	600
T_s(ms)	355.8	711.6	1423.2	2846.4	4269.6
T_p(ms)	172.4	309.5	592.8	1162.2	1729.1



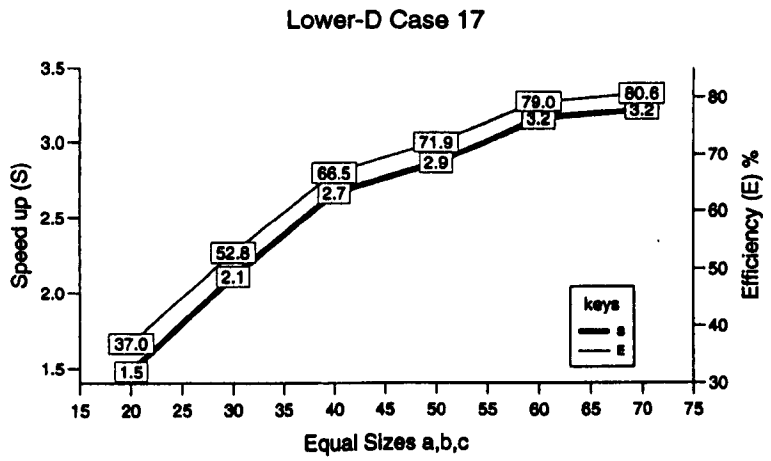
Case lower-D-16: $\mathbf{c} = \begin{bmatrix} 20 \\ b \\ 20 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ -3 & 1 & -2 & 0 \end{bmatrix}$ $N_p = 4$

b	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	164.5	294.4	564.9	1107.4	2191.0



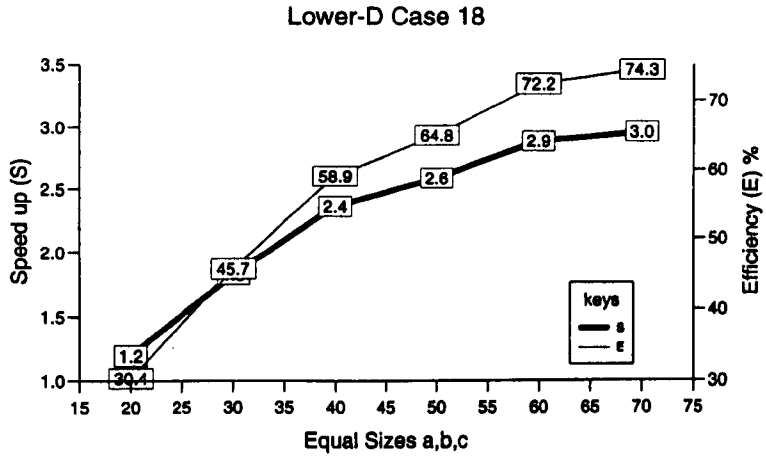
Case lower-D-17: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-16 $N_p = 4$

$a = b = c$	20	30	40	50	60	70
T_s(ms)	142.3	480.3	1138.6	2223.8	3842.6	6102.0
T_p(ms)	96.3	227.6	427.8	773.2	1215.7	1893.6



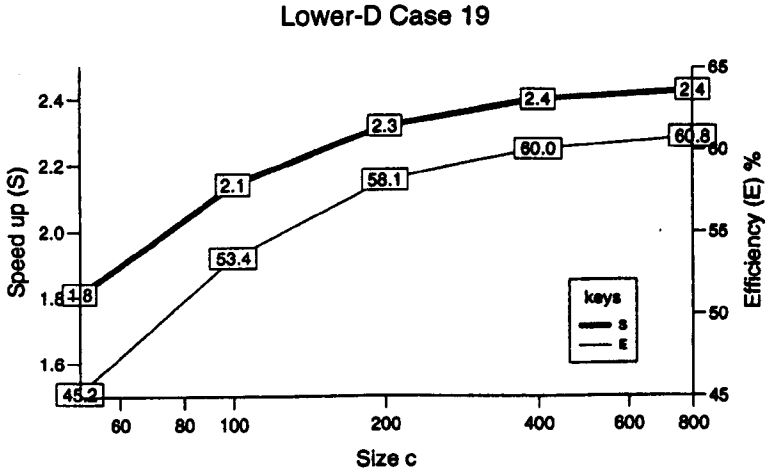
Case lower-D-18: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 2 & 2 & -3 & 1 \\ -3 & 1 & 2 & 0 \end{bmatrix}$ $N_p = 4$

$a = b = c$	20	30	40	50	60	70
T_s(ms)	142.3	480.3	1138.6	2223.8	3842.6	6102.0
T_p(ms)	117.2	262.6	483.5	858.2	1331.2	2051.8



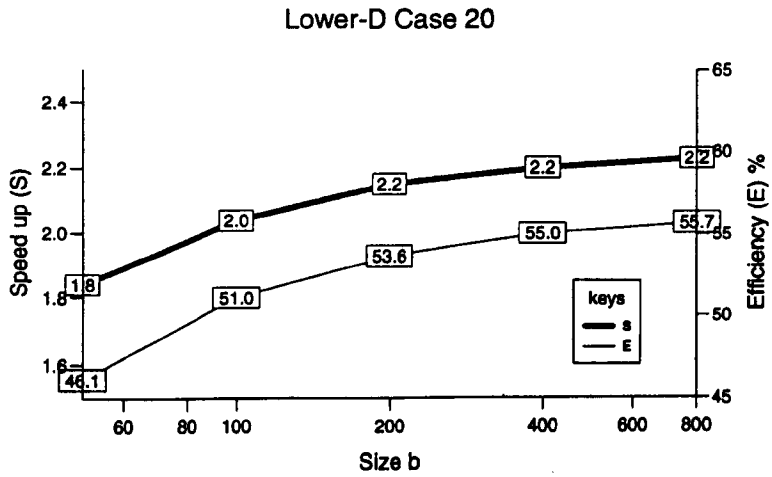
Case lower-D-19: $\mathbf{c} = [20, 20, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-18, $N_p = 4$

c	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	196.9	333.1	612.4	1186.7	2341.0



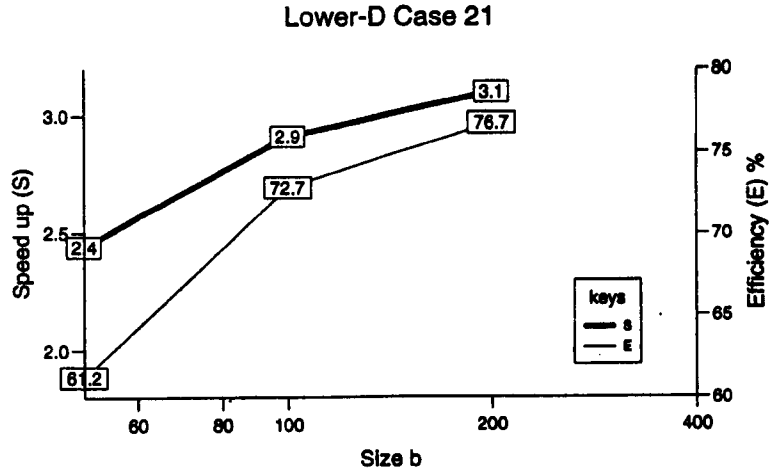
Case lower-D-20: $\mathbf{c} = [20, b, 20]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-18, $N_p = 4$

b	50	100	200	400	800
T_s(ms)	355.8	711.6	1423.2	2846.4	5692.8
T_p(ms)	193.0	349.0	663.5	1293.2	2552.7



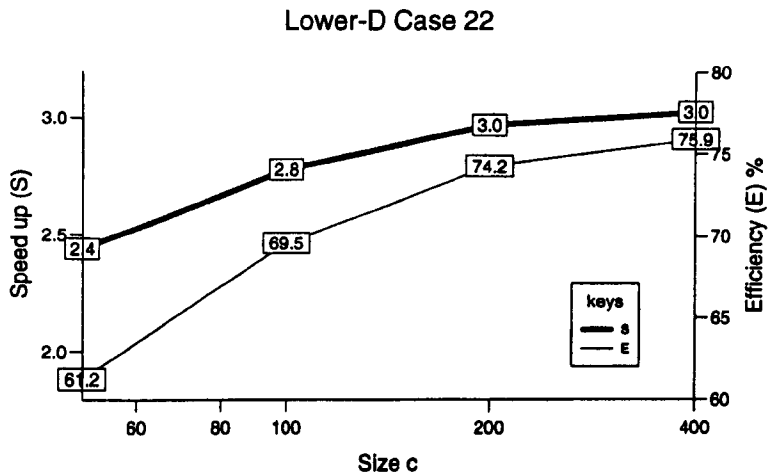
Case lower-D-21: $\mathbf{c} = [40, b, 40]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-18, $N_p = 4$

b	50	100	200
T_s(ms)	1423.2	2846.4	5692.8
T_p(ms)	581.4	979.3	1855.9



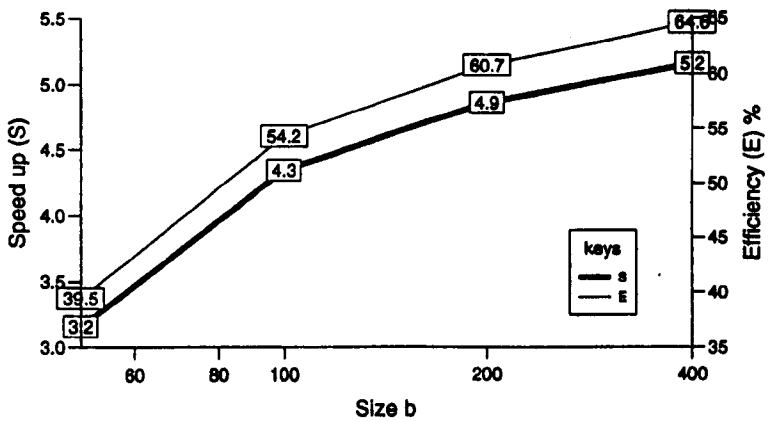
Case lower-D-22: $\mathbf{c} = [40, 40, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-18, $N_p = 4$

c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	582.1	1023.9	1919.3	3751.8



Case lower-D-23: $\mathbf{c} = [40, b, 40]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case lower-D-18, $N_p = 8$

b	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	450.5	655.9	1172.5	2204.5



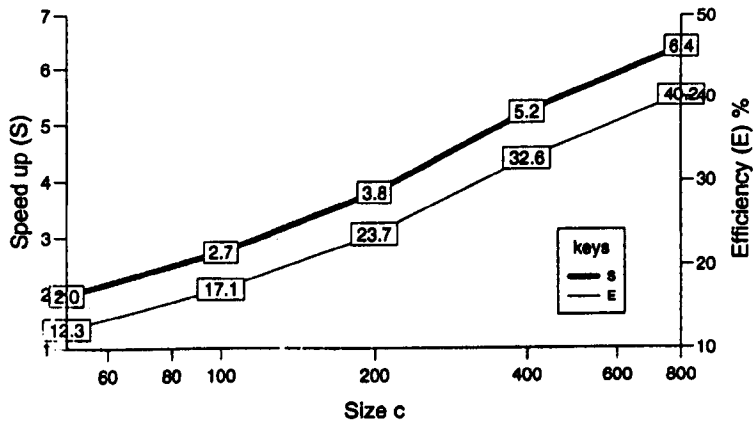
2-D array with SUC mesh

A 4 × 4 2-D array with SUC mesh are used for the experiments. Thus $N_p = 16$.

Case SUC-1: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

c	50	100	200	400	600	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	17078.4	22771.2
T_p(ms)	724.5	1038.3	1502.8	2182.2	2860.2	3540.6

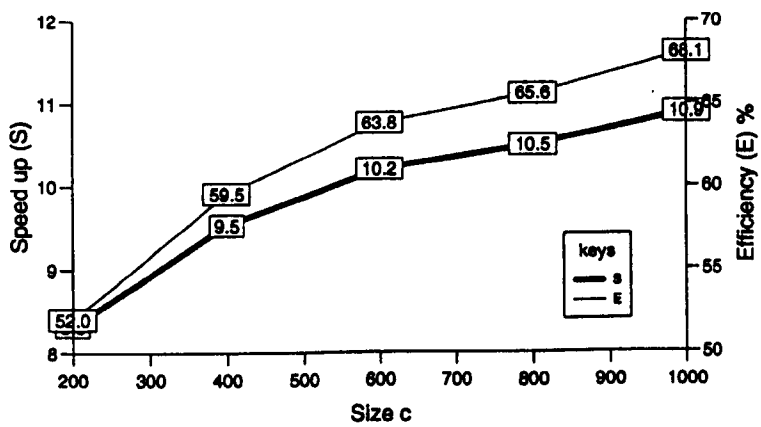
2-D SUC Array Case 1



Case SUC-2: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 \end{bmatrix}$ $N_p = 16$

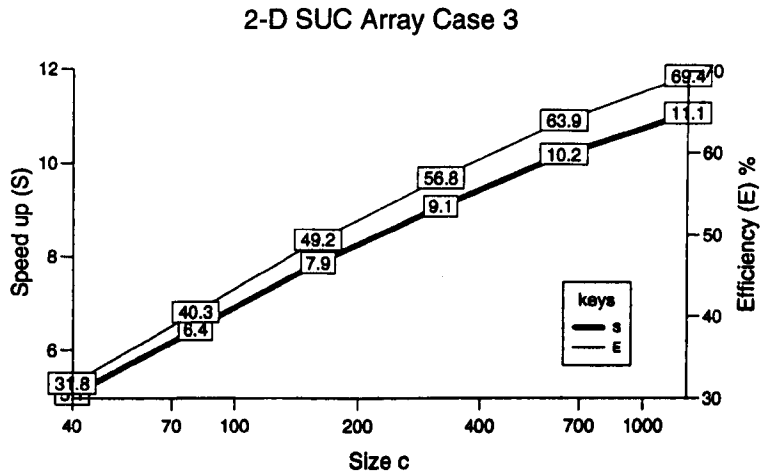
c	200	400	600	800	1000
T_s(ms)	5692.8	11385.6	17078.4	22771.2	28464.0
T_p(ms)	684.5	1196.3	1682.4	2176.1	2619.3

2-D SUC Array Case 2



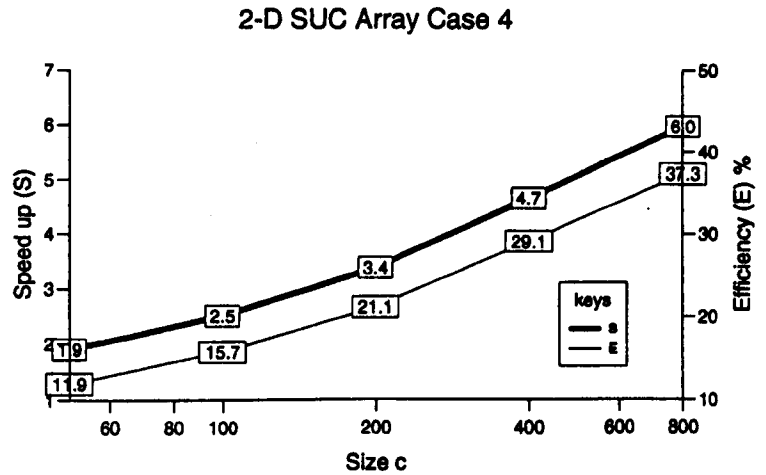
Case SUC-3: $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SUC-2 $N_p = 16$

c	40	80	160	320	640	1280
T_s(ms)	1138.6	2277.1	4554.2	9108.5	18217.0	36433.9
T_p(ms)	223.6	353.7	578.6	1001.9	1782.1	3274.4



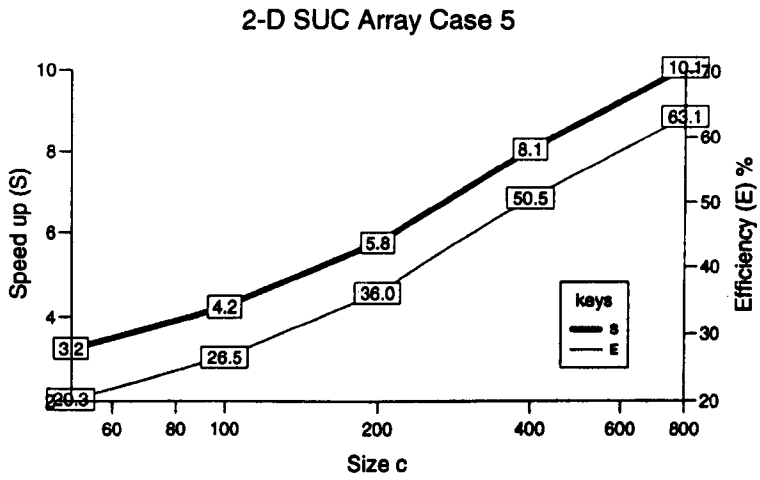
Case SUC-4: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

c	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	748.9	1133.2	1683.6	2446.8	3815.4



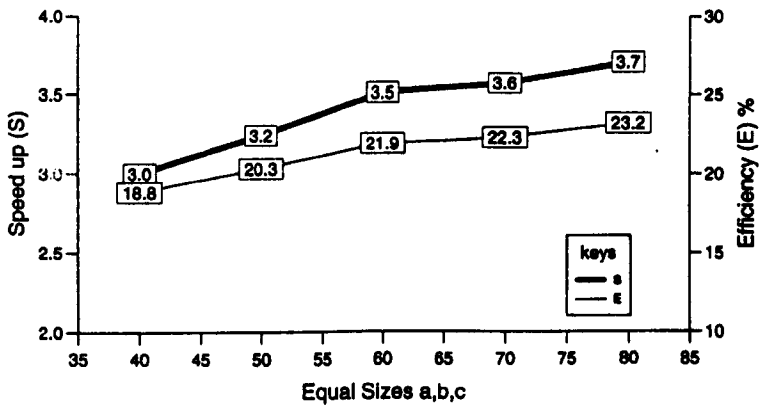
Case SUC-5: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

c	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	437.8	672.1	987.9	1409.4	2244.8



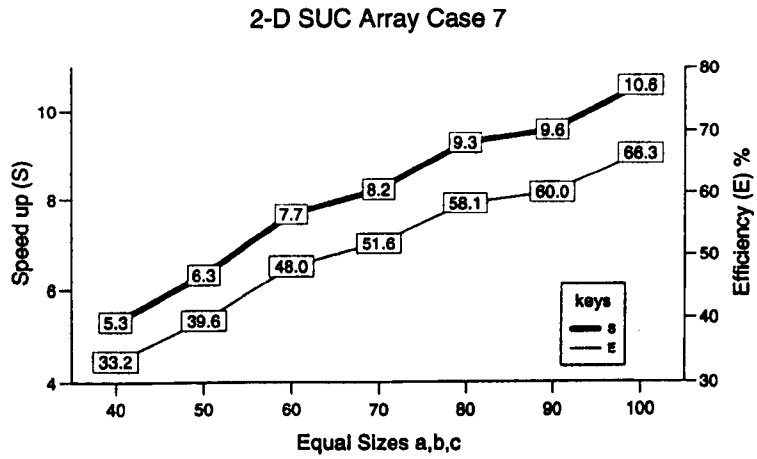
Case SUC-6: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SUC-5 $N_p = 16$

$a = b = c$	40	50	60	70	80
T_s(ms)	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	378.0	685.9	1095.2	1707.4	2457.4



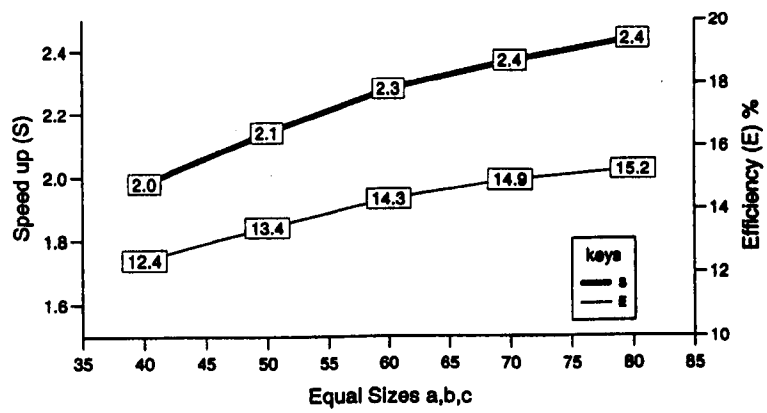
Case SUC-7: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70	80	90	100
T_s(ms)	1138.6	2223.8	3842.6	6102.0	9108.5	12968.9	17790.0
T_p(ms)	214.3	351.4	500.5	740.0	979.9	1351.2	1677.2



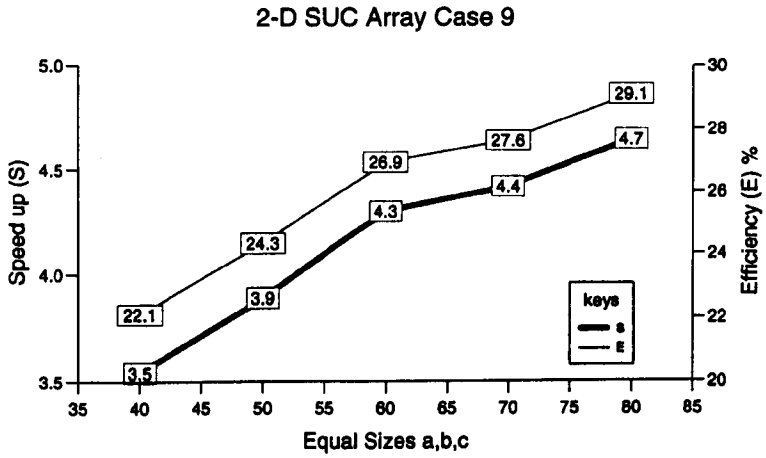
Case SUC-8: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70	80
T_s(ms)	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	574.3	1041.1	1681.7	2579.5	3738.7



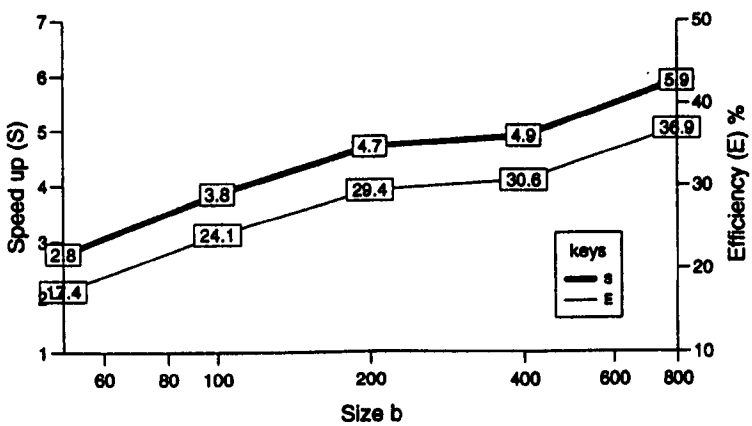
Case SUC-9: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70	80
T_s(ms)	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	321.3	572.0	895.6	1378.5	1957.1



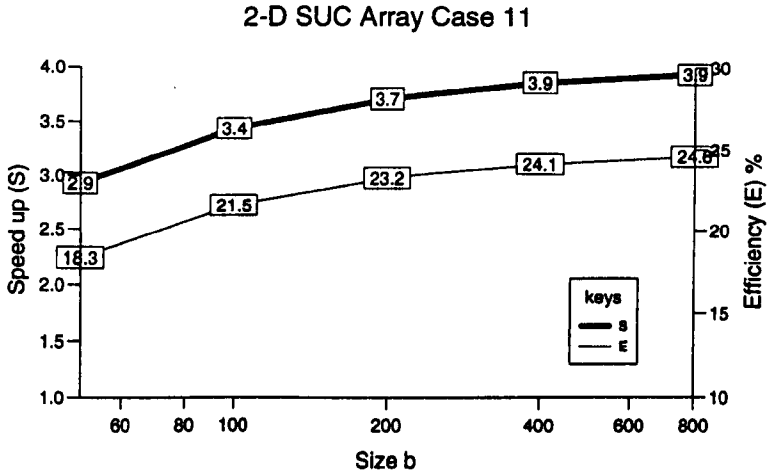
Case SUC-10: $\mathbf{c} = \begin{bmatrix} 40 \\ b \\ 40 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ -1 & 0 & 1 & 1 \end{bmatrix}$ $N_p = 16$

b	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	512.3	739.6	1212.3	2330.2	3858.7



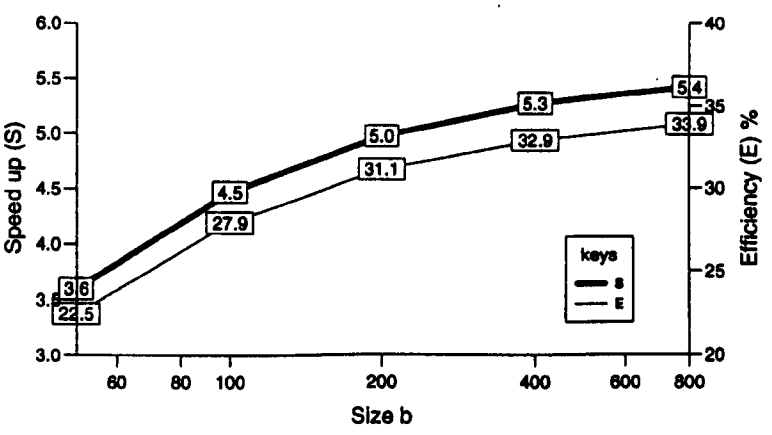
Case SUC-11: $\mathbf{c} = \begin{bmatrix} 40 \\ b \\ 40 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ -3 & 1 & -2 & 0 \end{bmatrix}$ $N_p = 16$

b	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	485.3	827.7	1534.7	2948.7	5783.0



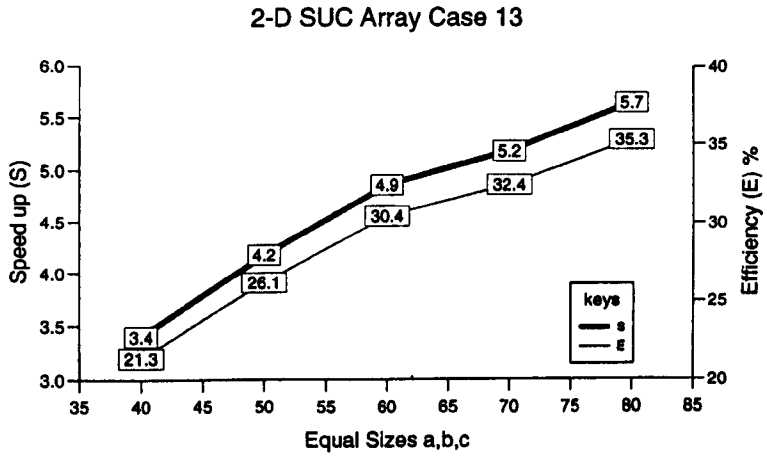
Case SUC-12: $\mathbf{c} = \begin{bmatrix} 40 \\ b \\ 40 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ -3 & 1 & -2 & 0 \end{bmatrix}$ $N_p = 16$

b	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	394.4	638.6	1146.6	2161.5	4200.8



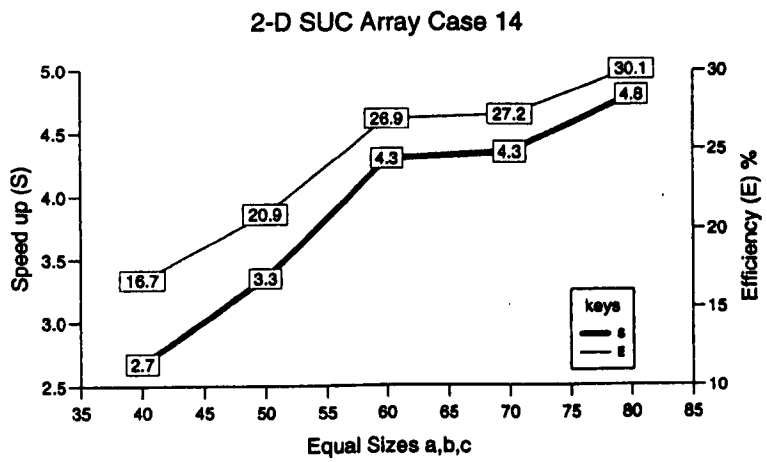
Case SUC-13: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SUC-12 $N_p = 16$

$a = b = c$	40	50	60	70	80
T_s(ms)	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	334.8	531.9	791.2	1177.2	1611.8



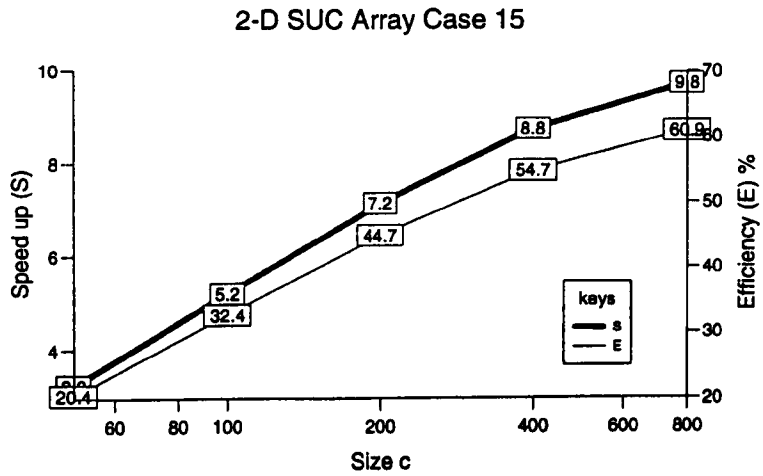
Case SUC-14: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 2 & 2 & -3 & 1 \\ -3 & 1 & 2 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70	80
T_s(ms)	1138.6	2223.8	3842.6	6102.0	9108.5
T_p(ms)	426.1	663.1	953.8	1402.6	1893.4



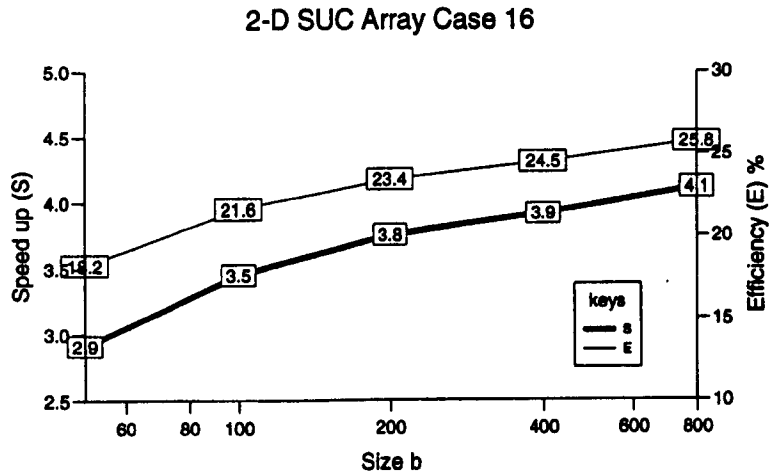
Case SUC-15: $\mathbf{c} = [40, 40, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SUC-14 $N_p = 16$

c	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	437.1	548.7	795.9	1301.4	2336.6



Case SUC-16: $\mathbf{c} = [40, b, 40]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SUC-15 $N_p = 16$

b	50	100	200	400	800
T_s(ms)	1423.2	2846.4	5692.8	11385.6	22771.2
T_p(ms)	488.5	825.0	1518.5	2906.8	5525.3



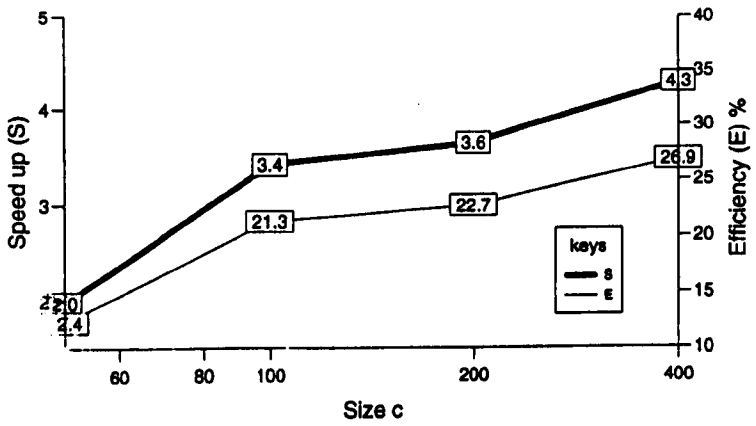
2-D array with SBC mesh

A 4×4 2-D array with SBC mesh are used for the experiments. Thus $N_p = 16$.

Case SBC-1: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	719.4	843.9	1574.8	2649.0

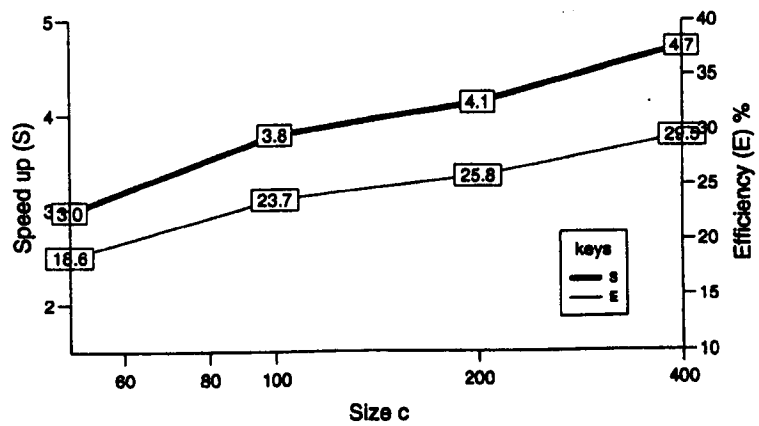
2-D SBC Array Case 1



Case SBC-2: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 \end{bmatrix}$ $N_p = 16$

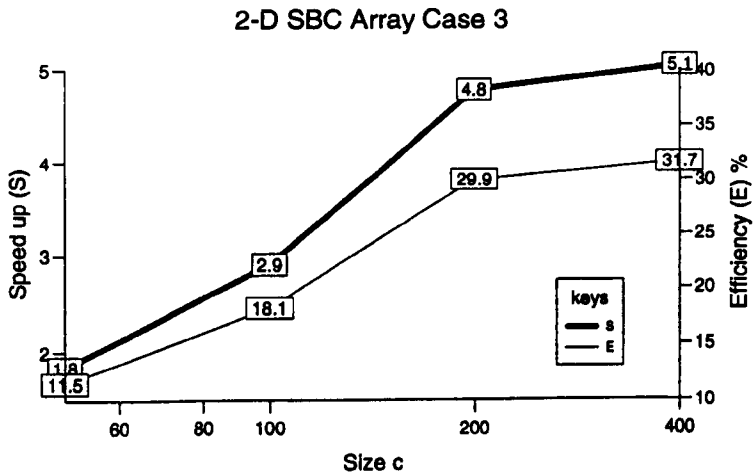
c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	479.2	752.1	1379.8	2415.3

2-D SBC Array Case 2



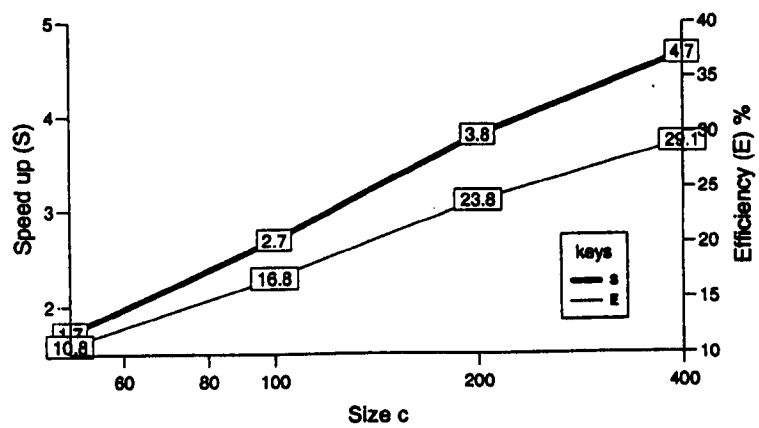
Case SBC-3: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & -1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	774.5	981.2	1190.1	2246.7



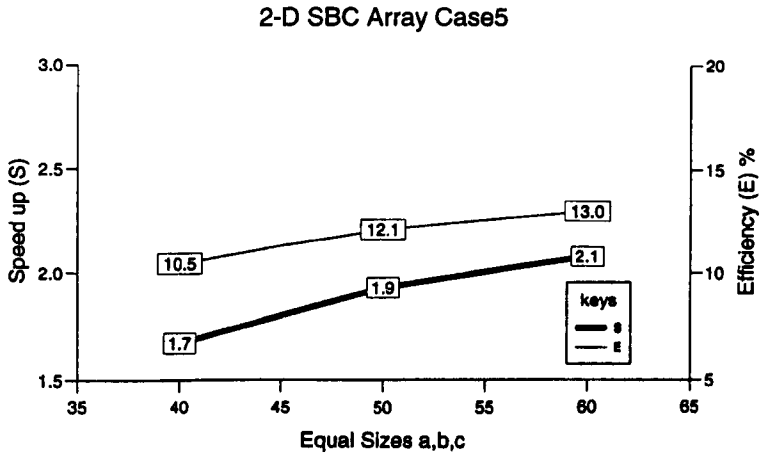
Case SBC-4: $\mathbf{c} = \begin{bmatrix} 40 \\ 40 \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	827.1	1057.4	1497.4	2441.6



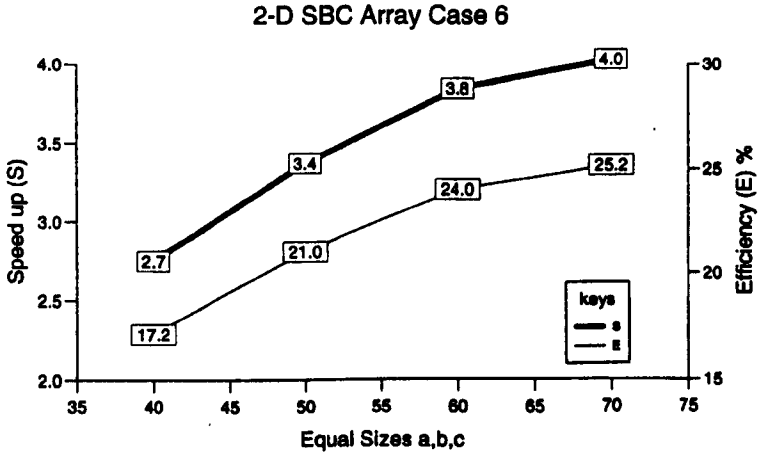
Case SBC-5: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -3 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60
T_s(ms)	1138.6	2223.8	3842.6
T_p(ms)	680.3	1150.4	1848.1



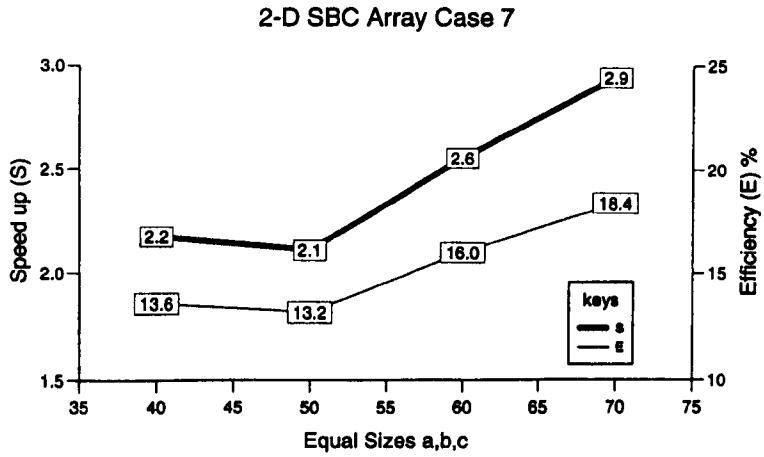
Case SBC-6: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70
T_s(ms)	1138.6	2223.8	3842.6	6102.0
T_p(ms)	414.6	660.5	1000.5	1513.9



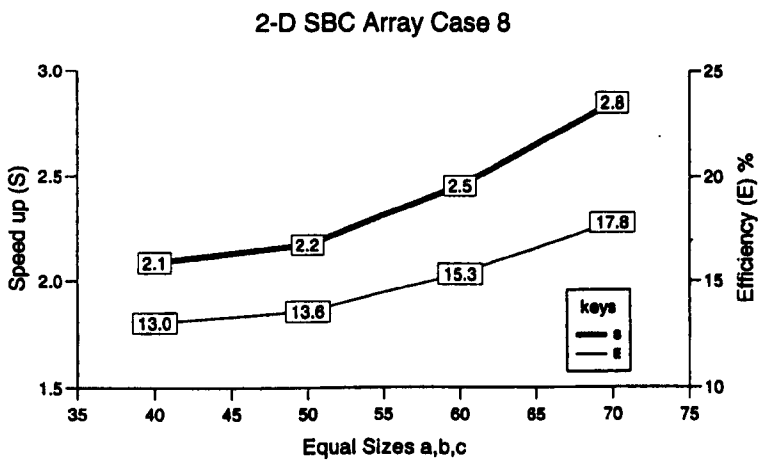
Case SBC-7: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70
T_s(ms)	1138.6	2223.8	3842.6	6102.0
T_p(ms)	523.7	1055.4	1505.2	2073.2



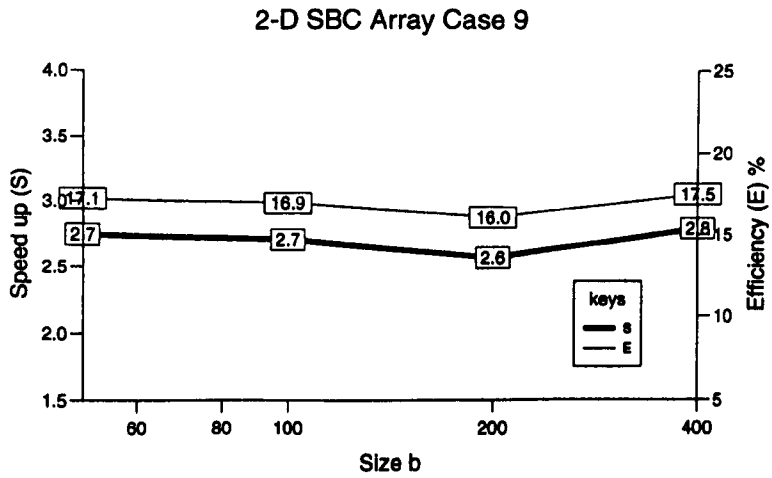
Case SBC-8: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70
T_s(ms)	1138.6	2223.8	3842.6	6102.0
T_p(ms)	545.7	1025.0	1567.9	2143.9



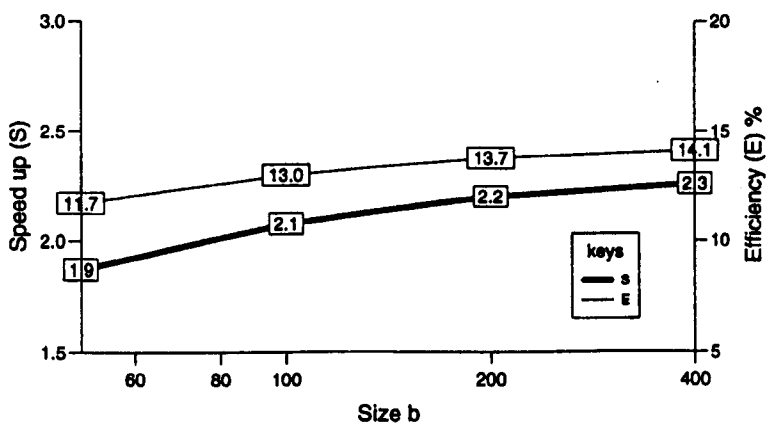
Case SBC-9: $\mathbf{c} = \begin{bmatrix} 40 \\ b \\ 40 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ -1 & 0 & 1 & 1 \end{bmatrix}$ $N_p = 16$

b	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	518.8	1055.2	2224.3	4071.6



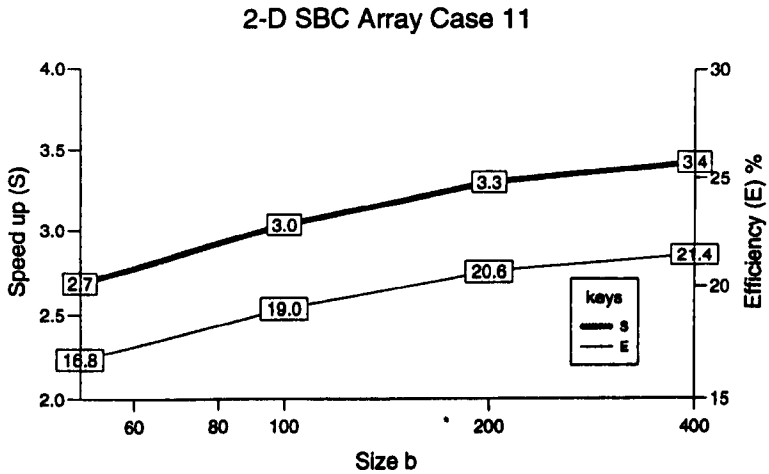
Case SBC-10: $\mathbf{c} = \begin{bmatrix} 40 \\ b \\ 40 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ -3 & 1 & -2 & 0 \end{bmatrix}$ $N_p = 16$

b	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	759.9	1370.6	2591.6	5038.9



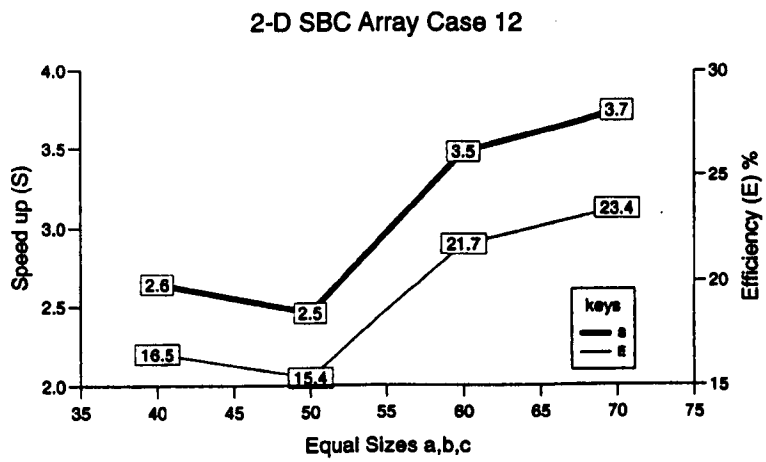
Case SBC-11: $\mathbf{c} = \begin{bmatrix} 40 \\ b \\ 40 \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 \\ -3 & 1 & -2 & 0 \end{bmatrix}$ $N_p = 16$

b	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	530.8	937.5	1723.7	3318.8



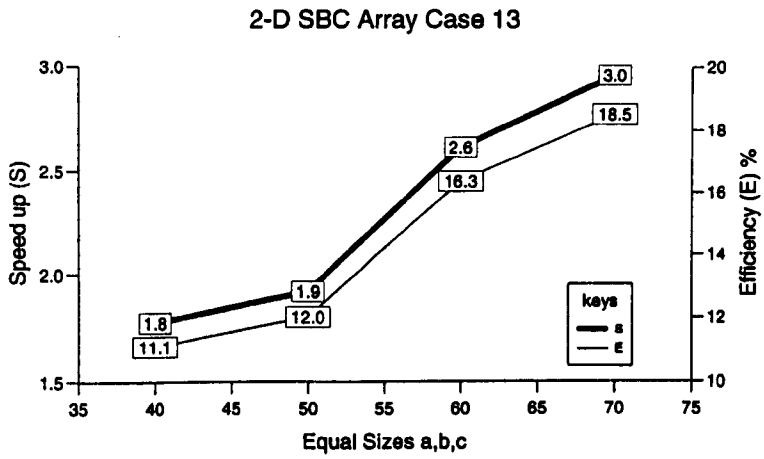
Case SBC-12: $\mathbf{c} = [a, b, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SBC-11 $N_p = 16$

$a = b = c$	40	50	60	70
T_s(ms)	1138.6	2223.8	3842.6	6102.0
T_p(ms)	431.2	904.7	1105.1	1629.0



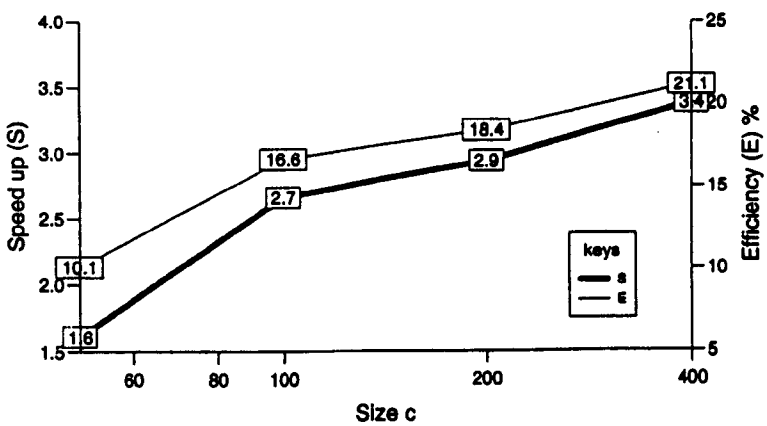
Case SBC-13: $\mathbf{c} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ $\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$ $\mathbf{D} = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 2 & 2 & -3 & 1 \\ -3 & 1 & 2 & 0 \end{bmatrix}$ $N_p = 16$

$a = b = c$	40	50	60	70
T_s(ms)	1138.6	2223.8	3842.6	6102.0
T_p(ms)	641.5	1157.4	1469.6	2061.0



Case SBC-14: $\mathbf{c} = [40, 40, c]^T$, $\mathbf{K}$ and $\mathbf{D}$ are the same as Case SBC-13 $N_p = 16$

c	50	100	200	400
T_s(ms)	1423.2	2846.4	5692.8	11385.6
T_p(ms)	883.5	1071.5	1936.1	3367.2



Appendix G

Examples of Automatically Generated Parallel Codes

G.1 Parallel Codes for Non-LSGP

G.1.1 h.file

```
static int cont_up_bds[3] = {4, 0, 499};
static int spnd_size[6] = {5, 1, 500, 24000, 500, 1};
static int jump_extra_vol[3] = {25000, 1000, 1};
static int jump_j_2_extra_vol[3] = {124000, 1000, 500};
static int d_q[4] = {-25000, -1002, -1000, -1001};
static int n_cubes_dep_spnd[3] = {1, 1, 1};
static int extra_vol[3] = {10, 2, 1000};
static int delay_data[1] = {0};
static int n_notes[1] = {500};
static int size_cubes[6] = {1, 1, 500, 24000, 500, 1};
static int bgs_to_extra_vol[1] = {101500};
static int l_inc1[3] = {0, -1, 1};
static int l_inc2[3] = {0, 0, -500};
static int l_v_c[12] = {-5, 0, 0, 0, 6, -1, 0, 0, 14, 1, -500, 0};
static int u_inc1[3] = {0, -1, 1};
static int u_inc2[3] = {0, 0, -500};
static int u_v_c[12] = {-5, 0, 0, 19, 6, -1, 0, 19, 14, 1, -500, 399};
static int bgs_out[1] = {226500};
static int para_data_flows[6] = {3, 0, 500, -1, 0, 500};
static int size_data_flows[3] = {500, 0, 500};
static int n_vtes_flows[2] = {1, 1};
static int direct_data_flows[2] = {1, 0};
static int p[2] = {1, 1};
static int time_comsumed[4] = {23, 29, 35, 41};
static int time_low_bds[4] = {0, 6, 12, 18};
static int lowerbds_sizes[12] = {0, 0, 0, 1, 6, 0, 2, 12, 0, 3, 18, 0};
```

```
static int order_vtr[1] = {0};
static int E[9] = {1, 0, 0, -1, 1, 0, -3, -1, 1};
static int B_j_2_i[9] = {5, 0, 0, -6, 1, 0, -14, -1, 500};
static int array[1] = {4};
static int i0=0, i1=0, i2=0;
static int j0=0, j1=0, j2=0;
static int pj0=0, pj1=0, pj2=0;
static int vol_elg_spnd=20000, spnd_subpoly_size=250000, time_used=42, min_t=0,
max_t=41, N_DC=1, N_links=1, n_cubes=1, lgst =500, n_para_data_flows=2,
n_dep_array=0, bg_spnd=126500, n_note_data_flows=1500, edge_extra=126500,
n_total_nodes=160000, n_prcsrs=4;
static data_type *elg_spnds[42], *data_flow[3];
static int la100=1,la200=3,la201=1;
static int ua100=1,ua200=3,ua201=1;
static int cl000,cl100,cl101,cl200,cl201,cl202;
static int cu000,cu100,cu101,cu200,cu201,cu202;
static int lsa100=6,lsa200=14,lsa201=1;
static int usa100=6,usa200=14,usa201=1;
static int cls000=0,cls100=0,cls101,cls200=416,cls201,cls202;
static int cus000=19,cus100=23,cus101,cus200=399,cus201,cus202;
static int ll_1[3],ll_2[3];
static int uu_1[3],uu_2[3];
static int *q_ad0,*q_ad1,*q_ad2,*q_ad3;

#define CopyFromBufferToSupernode(a,b,zise_a) {\
tmp_ptr2 = b;\
for(i0=0;i0<*(zise_a+0);i0++)\
{\
    for(i1=0;i1<*(zise_a+1);i1++)\
    {\
        for(i2=0;i2<*(zise_a+2);i2++)\
        {\
            *tmp_ptr2++ = *a++;\
        }\
        tmp_ptr2 += *(zise_a+4);\
    }\
    tmp_ptr2 += *(zise_a+3);\
}\
}\

#define CreateDirectInputDBV(b,zise_a) {\
tmp_ptr2 = b;\
for(i0=0;i0<*(zise_a+0);i0++)\
{\
    for(i1=0;i1<*(zise_a+1);i1++)\
```

```
    {\
        (data_flow_vec_tmp++)->iov_base = (caddr_t) tmp_ptr2;\
        tmp_ptr2 += *(zise_a+2)**(zise_a+4);\
    }\
    tmp_ptr2 += *(zise_a+3);\
}\
}\

#define CopyFromSupernodeToBuffer(a,b,zise_b) {\
tmp_ptr1 = a;\
for(i0=0;i0<*(zise_b+0);i0++)\
{\
    for(i1=0;i1<*(zise_b+1);i1++)\
    {\
        for(i2=0;i2<*(zise_b+2);i2++)\
        {\
            *b++ = *tmp_ptr1++;\
        }\
        tmp_ptr1 += *(zise_b+4);\
    }\
    tmp_ptr1 += *(zise_b+3);\
}\
}\

#define CopyFromSupernodeToBuffer0(a,b,zise_b) {\
tmp_ptr1 = a;\
for(i0=0;i0<*(zise_b+0);i0++)\
{\
    for(i1=0;i1<*(zise_b+1);i1++)\
    {\
        for(i2=0;i2<*(zise_b+2);i2++)\
        {\
            *b++ = *tmp_ptr1;\
        }\
    }\
}\
}\

#define CreateDirectOutputDBV(a,zise_b) {\
tmp_ptr1 = a;\
for(i0=0;i0<*(zise_b+0);i0++)\
{\
    for(i1=0;i1<*(zise_b+1);i1++)\
    {\
        (data_flow_vec_tmp++)->iov_base = (caddr_t) tmp_ptr1;\
```

```
        tmp_ptr1 += *(zise_b+2)+*(zise_b+4);\
    }\
    tmp_ptr1 += *(zise_b+3);\
}\
}\

#define CreateDirectOutputDBV0(a,zise_b) {\
tmp_ptr1 = a;\
for(i0=0;i0<*(zise_b+0);i0++)\
{\
    for(i1=0;i1<*(zise_b+1);i1++)\
    {\
        (data_flow_vec_tmp++)->iov_base = (caddr_t) tmp_ptr1;\
    }\
}\
}\

#define InitializeData(bounds) {q_ad0 = elg_spnds[t] + bg_spnd;\
for(q[0]=max_fn(cl000,0),cl101=cl100+q[0]*la100,cl201=cl200+q[0]*la200,
cu101=cu100+q[0]*ua100,cu201=cu200+q[0]*ua200,q_ad1=q_ad0+q[0]*jump_extra_vol[0];
q[0]<=min_fn(cu000,*(bounds+0));q[0]++,cl101+=la100,cl201+=la200,cu101+=ua100,
cu201+=ua200,q_ad1+=jump_extra_vol[0])\
    for(q[1]=max_fn(cl101,0),cl202=cl201+q[1]*la201,cu202=cu201+q[1]*ua201,
    q_ad2=q_ad1+q[1]*jump_extra_vol[1];q[1]<=min_fn(cu101,*(bounds+1));q[1]++,
    cl202+=la201,cu202+=ua201,q_ad2+=jump_extra_vol[1])\
        for(q[2]=max_fn(cl202,0),q_ad3=q_ad2+q[2];q[2]<=min_fn(cu202,*(bounds+2));
        q[2]++,q_ad3++)\
        {\
            ct++;\
            matrix_int_multiplication(E,q,i_index,dim,dim,1,1,0);\
            add_2_int_vtrs(i_index,i_from_j,i_index,N,1);\
            matrix_int_multiplication(jump_step,i_index,q_ad3,1,dim,1,1,0);\
        }\
}\

\*
add_2_int_vtrs(a,b,c,...) is a function to add vectors a and b to c.
matrix_int_multiplication(A,B,C,....) is a function to multiply matrices A and B to
*\


#define ComputeSupernode(bounds) {q_ad0 = elg_spnds[t] + bg_spnd;\
for(i0=max_fn(cl000,0),cl101=cl100+i0*la100,cl201=cl200+i0*la200,cu101=cu100+i0*ua1
cu201=cu200+i0*ua200,q_ad1=q_ad0+i0*jump_extra_vol[0];i0<=min_fn(cu000,*(bounds+0))
i0++,cl101+=la100,cl201+=la200,cu101+=ua100,cu201+=ua200,q_ad1+=jump_extra_vol[0])\
```

```
    for(i1=max_fn(cl101,0),cl202=cl201+i1*la201,cu202=cu201+i1*ua201,
    q_ad2=q_ad1+i1*jump_extra_vol[1];i1<=min_fn(cu101,*(bounds+1));i1++,cl202+=la20
    cu202+=ua201,q_ad2+=jump_extra_vol[1])\
        for(i2=max_fn(cl202,0),q_ad3=q_ad2+i2;i2<=min_fn(cu202,*(bounds+2));
        i2++,q_ad3++)\
        {\
            *q_ad3 = *q_ad3+*(q_ad3+d_q[0])+*(q_ad3+d_q[1])+*(q_ad3+d_q[2])
            +*(q_ad3+d_q[3]);\
        }\
}\

#define cl_cu_0_set_bef0 j0=*(a_i_to+0),cls101=cls100+j0*lsa100,cls201=cls200
+j0*lsa200,cus101=cus100+j0*usa100,cus201=cus200+j0*usa200,ll_1[0]=l_0[0],
ll_1[1]=l_0[1],ll_1[2]=l_0[2],uu_1[0]=u_0[0],uu_1[1]=u_0[1],uu_1[2]=u_0[2]

#define cl_cu_0_set_aft0

#define cl_cu_0_set_bef1 cls202=cls201+j1*lsa201,cus202=cus201+j1*usa201

#define cl_cu_0_set_aft1 ,ll_2[0]=ll_1[0]+j1*l_inc1[0],
ll_2[1]=ll_1[1]+j1*l_inc1[1],ll_2[2]=ll_1[2]+j1*l_inc1[2],
uu_2[0]=uu_1[0]+j1*u_inc1[0],uu_2[1]=uu_1[1]+j1*u_inc1[1],
uu_2[2]=uu_1[2]+j1*u_inc1[2],pj1=p[0]*j1

#define cl_cu_0_set_bef2

#define cl_cu_0_set_aft2 ,cl000=ll_2[0]+j2*l_inc2[0],cl100=ll_2[1]+j2*l_inc2[1],
cl200=ll_2[2]+j2*l_inc2[2],cu000=uu_2[0]+j2*u_inc2[0],cu100=uu_2[1]+j2*u_inc2[1],
cu200=uu_2[2]+j2*u_inc2[2]

#define cl_cu_0_increase1 ,ll_2[0]+=l_inc1[0],ll_2[1]+=l_inc1[1],ll_2[2]+=l_inc1[2]
uu_2[0]+=u_inc1[0],uu_2[1]+=u_inc1[1],uu_2[2]+=u_inc1[2],pj1+=p[0]

#define cl_cu_0_increase2 ,cl000+=l_inc2[0],cl100+=l_inc2[1],cl200+=l_inc2[2],
cu000+=u_inc2[0],cu100+=u_inc2[1],cu200+=u_inc2[2]

#define condition j2>=l_innerest_bd

#define J_2_J_vtr j_vtr[0]=j0;j_vtr[1]=j1;j_vtr[2]=j2;

#define BREAK if(++t>time_end){if(t<=max_t){j2=l_innerest_bd-1000000;}
else{j2=u_innerest_bd+1;}}

#define SupernodeSpaceLoops \
for(cl_cu_0_set_bef0,j1=cls101 cl_cu_0_set_aft1;j1<=cus101;j1++ cl_cu_0_increase1)\
```

```
    for(cl_cu_0_set_bef1,l_innerest_bd=cls202/500,u_innerest_bd=cus202/500,
    j2=t-pj1 cl_cu_0_set_aft2;j2<=u_innerest_bd;j2++ cl_cu_0_increase2)\
```

G.1.2 Parallel Code

```
#include <stdio.h>
#include <math.h>
#include <csn/csn.h>
#include <csn/csnuio.h>
#include <csn/names.h>
#include <cs.h>
#include "../parameters.h"
#include "../pn.h"
#include "macro_def.c"
#include "prcsr.h"

main (argc, argv)
int argc;
char **argv;
{
    Transport transport[D_array*(bi_link_orig+1)];
    netid_t   net_ID[D_array*(bi_link_orig+1)];
    int n_netid_t=0,in,*dep_array,m,a_i[D_array],a_i_to[10*D_array];
    int sequenceCount,*sequence,id_pos=2,array[D_array],rx_status=0,tx_status=0;
    int i,ii,j,j_vtr[N],jj,r,l,data_flow_bank,n_data_flow_vec=4*n_para_data_flows,
    N2=2*N,tmp_i,uni_link=1-bi_link_orig,bi_link=bi_link_orig,dim=N,k=N-D_array;
    int t,ptr_elg_spnd=0,ptr_data_flow=0,link_out_valid[D_array*(bi_link_orig+1)],
    link_in_valid[D_array*(bi_link_orig+1)];
    int *direct_data_flows_out=direct_data_flows;
    int *direct_data_flows_in=direct_data_flows+n_cubes;
    int x,y,l_innerest_bd,u_innerest_bd,time_end,l_0[N+1],u_0[N+1];
    int q[N],n_vec[8*D_array],i_index[N],i_from_j[N],*i_from_j_inc,jump_step[N],
    tmp_vtr[N],j_vtr_init[N],ct=0,ctt=0,*ptr1,*ptr2,op_spnd;
    data_type *tmp_ptr1,*tmp_ptr2,*tmp_ptr3,*elg_spnds_base,*idle_spnd;
    char name[20],here_name[20],there_name[20];
    struct iovec *data_flow_vec,*data_flow_vec_tmp;

    if (!cs_import ("int vectorSize", &sequenceCount))
        debugf("cannot import 'vectorSize'\n", 1);
    sequence = (int *) malloc(sequenceCount*sizeof(int));
    if (!cs_import ("int *vector", &sequence))
        debugf("cannot import 'vector'\n", 1);
    cp_vtrs(sequence,array,D_array);
    cp_vtrs(sequence+D_array,a_i,D_array);
```

```
    dep_array = sequence+2*D_array;
    m = (sequenceCount-2*D_array)/D_array;

    csn_init();
    strcpy(name,ints_2_string(a_i,D_array));
    strcpy(here_name,"transpot");
    strcat(here_name,name);
    for(i=0;i<N_links;i++)
    {
        strcpy(there_name,here_name);
        strcat(there_name,int_2_alph(i));
        if( csn_open( CSN_NULL_ID, &transport[i] ) != CSN_OK )
            debugf( "master: cannot open transport\n", 1 );
        if( csn_registername( transport[i],there_name ) != CSN_OK )
            debugf( "cannot register Transport \n",1 );
    }

    r = 0;
    for(i=0;i<D_array;i++)
        for(j=1;j>=-bi_link_orig;j -= 2,r++)
        {
            copy_int_array(a_i,a_i_to,1,D_array,1,D_array,0,0);
            a_i_to[i] += j;
            if(0>a_i_to[i]||a_i_to[i]>=array[i])
                link_out_valid[r] = 0;
            else
            {
                strcpy(there_name,"transpot");
                strcat(there_name,ints_2_string(a_i_to,D_array));
                strcat(there_name,int_2_alph(r));
                if(csn_lookupname(&net_ID[r],there_name,1)!=CSN_OK)
                    debugf("cannot lookup %s to %s\n",a_i,there_name);
                link_out_valid[r] = 1;
            }
            a_i_to[i] -= 2*j;
            if(0>a_i_to[i]||a_i_to[i]>=array[i])
                link_in_valid[r] = 0;
            else
                link_in_valid[r] = 1;
        }
\* cp_vtrs(a,b,...) and copy_int_array(a,b,...) are functions of copy vector a to b

    if((elg_spnds_base = (data_type *) malloc(spnd_subpoly_size*sizeof(data_type)))
    ==NULL)
```

```
{
    debugf("Err: no enough memory for elg_spnds_base %d\n",
    spnd_subpoly_size*sizeof(data_type));
    exit(1);
}

idle_spnd = (data_type *) malloc(lgst*sizeof(data_type));

for(i=0;i<spnd_subpoly_size;i++)
    *(elg_spnds_base+i) = 0;

for(i=0;i<lgst;i++)
    *(idle_spnd+i) = 0;

for(i=0;i<N_links+2;i++)
{
    if(!i)
        data_flow[i] = (data_type *) malloc(n_note_data_flows*sizeof(data_type)
    else
    {
        data_flow[i] = data_flow[i-1];
        data_flow[i]  += 2*size_data_flows[i-1];
    }
}
for(i=0;i<n_note_data_flows;i++)
    *(data_flow[0]+i)= 0;

data_flow_vec = (struct iovec *) malloc(2*n_para_data_flows*sizeof(struct iovec

l = 0;
for(t=0,r=0;t<2;t++)
    for(i=0,j=0;i<N_links;i++)
        for(ii=0;ii<2;ii++)
        {
            n_vec[l] = 0;
            for(jj=0;jj<n_vtes_flows[i*N_links+ii];jj++,j+=3)
            {
                data_flow_bank = 0;
                if(para_data_flows[j]<(N_links+1)&&t==ii)
                    data_flow_bank = size_data_flows[para_data_flows[j]];
                if(0<=para_data_flows[j]&&para_data_flows[j]<N_links+2)
                    (data_flow_vec+r)->iov_base = (caddr_t)
                    (data_flow[para_data_flows[j]] + para_data_flows[j+1]
                    +data_flow_bank);
                (data_flow_vec+r)->iov_len = para_data_flows[j+2]
```

```
                    *sizeof(data_type);
                    if((data_flow_vec+r)->iov_len)
                    {
                        r++;
                        n_vec[l]++;
                    }
                }
                l++;
            }

    l_0[N] = 1;
    u_0[N] = 1;
    re_order_int_rows(a_i,a_i_to,order_vtr,D_array,1);
    time_end = time_comsumed[array_2_ad(a_i,array,D_array)];
    ptr2=(lowerbds_sizes+dim*array_2_ad(a_i,array,D_array));

    transpose_int(B_j_2_i,B_j_2_i,N,N);
    re_order_int_rows(B_j_2_i,B_j_2_i,order_vtr,D_array,N);
    transpose_int(B_j_2_i,B_j_2_i,N,N);
    cp_vtrs(ptr2,a_i_to,D_array);
    cp_vtrs(a_i_to,l_0,D_array);
    cp_vtrs(a_i_to,u_0,D_array);
    re_order_int_rows(l_0,l_0,order_vtr,D_array,1);
    re_order_int_rows(u_0,u_0,order_vtr,D_array,1);

\*
transpose_int(A,B,...) is a function to transpose matrix A to B.
re_order_int_rows(A,B,c,....) is a function to permute the rows of matrix A to B
according to vector c.
*\

    cp_vtrs(ptr2,j_vtr_init,dim);

    for(i=D_array;i<N;i++)
    {
        l_0[i] = 0;
        u_0[i] = 0;
    }

    matrix_int_multiplication(l_v_c,l_0,l_0,N,N+1,1,1,0);
    matrix_int_multiplication(u_v_c,u_0,u_0,N,N+1,1,1,0);

    for(i=1,jump_step[N-1]=1;i<N;i++)
    jump_step[N-1-i] = 1*jump_step[N-i];
```

```
t=0;
SupernodeSpaceLoops
{
    if(condition)
    {
        J_2_J_vtr;
        add_2_vtrs(j_vtr_init,j_vtr,tmp_vtr,N,-1);
        inner_prct(tmp_i,tmp_vtr,jump_j_2_extra_vol,N);
        elg_spnds[t] = (data_type*) elg_spnds_base + tmp_i;
        matrix_int_multiplication(B_j_2_i,j_vtr,i_from_j,N,N,1,1,0);
        InitializeData(cont_up_bds);
    }
    else
        elg_spnds[t] = idle_spnd;

    if(*ptr2==100000)
        l_innerest_bd = u_innerest_bd;
    BREAK;
}

data_flow_bank = 0;
ii = 0;
t = 0;

SupernodeSpaceLoops
{
    for(i=0,r=0;i<N_DC;i++,r+=N2)
        if(direct_data_flows_out[i]!=-1)
        {
            data_flow_vec_tmp = data_flow_vec + ii + direct_data_flows[i+n_cube
            CreateDirectInputDBV(elg_spnds[t]+bgs_to_extra_vol[i],size_cubes+r)
        }

    for(i=0,r=0;i<N_DC;i++,r+=N2)
        if(direct_data_flows_out[i]!=-1)
        {
            data_flow_vec_tmp = data_flow_vec + ii + direct_data_flows[i];
            op_spnd = t - delay_data[i] - 1;
            if(op_spnd<0||elg_spnds[op_spnd]==idle_spnd)
            {
                tmp_ptr3 = idle_spnd;
                CreateDirectOutputDBV0(tmp_ptr3,size_cubes+r);
            }
            else
            {
```

```
            tmp_ptr3 = elg_spnds[op_spnd]+bgs_out[i];
            CreateDirectOutputDBV(tmp_ptr3,size_cubes+r);
        }
    }

for(l=0,i=0;i<N_links;i++)
{
    if(link_in_valid[i]&&n_vec[l])
        csn_rxnbv(transport[i],data_flow_vec+ii,n_vec[l]);
    ii += n_vec[l++];
    if(link_out_valid[i]&&n_vec[l])
        csn_txnbv(transport[i],0,net_ID[i],data_flow_vec+ii,n_vec[l]);
    ii += n_vec[l++];
}

for(i=0;i<N_links;i++)
    if(link_in_valid[i])
        csn_test(transport[i],CSN_RXREADY,-1,NULL,NULL,&rx_status);

tmp_ptr1 = data_flow[N_links+1];

if(condition)
    for(i=0,r=0;i<N_DC;i++,r+=N2)
        if(direct_data_flows_in[i]==-1)
            CopyFromBufferToSupernode(tmp_ptr1,elg_spnds[t]
            +bgs_to_extra_vol[i],size_cubes+r);

if(condition)
    ComputeSupernode(cont_up_bds);

tmp_ptr2 = (data_flow_bank)? data_flow[0]:data_flow[0]+size_data_flows[0];

for(i=0;i<N_links;i++)
    if(link_out_valid[i])
        csn_test(transport[i],CSN_TXREADY,-1,NULL,NULL,&tx_status);

for(i=0,r=0;i<N_DC;i++,r+=N2)
    if(direct_data_flows_out[i]==-1)
    {
        op_spnd = t - delay_data[i];
        if(op_spnd<0||elg_spnds[op_spnd]==idle_spnd)
        {
            tmp_ptr3 = idle_spnd;
            CopyFromSupernodeToBuffer0(tmp_ptr3,tmp_ptr2,size_cubes+r);
        }
```

```
                else
                {
                    tmp_ptr3 = elg_spnds[op_spnd]+bgs_out[i];
                    CopyFromSupernodeToBuffer(tmp_ptr3,tmp_ptr2,size_cubes+r);
                }
            }

        data_flow_bank++;
        if(data_flow_bank==2)
        {
            data_flow_bank = 0;
            ii = 0;
        }

        if(*ptr2==100000)
            l_innerest_bd = u_innerest_bd;
        BREAK;

    }
}
```

G.2 Parallel Codes for LSGP

G.2.1 h.file

```
static int cont_up_bds[3] = {3, 3, 11};
static int spnd_size[6] = {4, 4, 12, 7488, 144, 1};
static int jump_extra_vol[3] = {8112, 156, 1};
static int jump_j_2_extra_vol[3] = {31800, -31212, 12};
static int d_q[4] = {-8112, -158, -156, -157};
static int n_cubes_dep_spnd[4] = {1, 1, 1, 1};
static int extra_vol[3] = {8, 8, 24};
static int delay_data[4] = {2, 0, 3, 2};
static int n_notes[4] = {48, 24, 48, 8};
static int size_cubes[24] = {4, 1, 12, 7956, 144, 1, 4, 3, 2, 7644, 154, 1, 1, 4,
12, 7488, 144, 1, 4, 1, 2, 7956, 154, 1};
static int bgs_to_extra_vol[4] = {32928, 33082, 24972, 32926};
static int l_inc2[3] = {0, 0, -12};
static int l_v_c[12] = {-4, 4, 0, 0, 8, -12, 0, 0, 32, 8, -12, 0};
static int u_inc2[3] = {0, 0, -12};
static int u_v_c[12] = {-4, 4, 0, 39, 8, -12, 0, 39, 32, 8, -12, 49};
static int bgs_out[4] = {33552, 33094, 57420, 33562};
static int lower_array_bds[2] = {-11, 0};
static int H[4] = {3, 0, -2, 3};
static int h0[3] = {-1, 0, 0};
```

```
static int ptn_array[278] = {9, 1, 2, 1, 0, 0, 48, 0, 0, 0, 1, 2, 0, 0, 0, 1, 0,
0, 48, 3, 1, 2, 3, 0, 2, 0, 0, 24, 0, 72, 8, 0, 1, 0, 24, 48, 1, 2, 1, 0, 0, 48,
0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 48, 2, 1, 3, 0, 2, 0, 0, 24, 0, 24, 8, 0, 0, 3, 0,
2, 3, 1, 0, 48, 48, 0, 3, 0, 0, 48, 0, 96, 8, 1, 0, 8, 0, 2, 0, 3, 0, 0, 2, 0, 0,
48, 0, 48, 8, 0, 3, 0, 1, 3, 0, 2, 0, 48, 24, 0, 72, 8, 1, 0, 0, 48, 0, 9, 1, 2,
-1, 0, 1, 5, 0, 48, 0, 0, 0, 1, 2, -1, 3, 0, 0, 0, 1, 5, 0, 48, 3, 2, 1, 3, -1,
0, -1, 3, 1, 1, 0, 2, 5, 48, 24, 5, 72, 8, 0, 1, 5, 0, 48, 0, 1, 2, 0, 48, 0, 0,
0, 1, 2, 0, 3, 0, 0, 0, 1, 5, 0, 48, 1, 1, 0, 1, 0, 2, 5, 0, 24, 1, 0, 8, 0, 0,
4, 2, 0, 3, 3, -1, 0, -1, 0, 0, 2, 2, 2, 1, 5, 0, 48, 0, 3, 5, 48, 48, 5, 96, 8,
5, 104, 8, 0, 2, 0, 3, 0, -1, 2, 2, 0, 0, 2, 5, 0, 48, 5, 48, 8, 0, 3, 1, 3, 0,
0, -1, 0, 1, 1, 2, 0, 2, 5, 0, 24, 5, 24, 8, 1, 5, 32, 48, 0, 3, 0, 104, 8, 48,
0, 0, 0, 0, 8, 48, 0, 0, 112};
static int cubes_out_group[8] = {0, 1, 1, 2, 2, 3, 3, 4};
static int D[12] = {1, 0, 0, 0, -1, 1, 1, 1, -3, 1, -1, 0};
static int size[3] = {39, 39, 49};
static int lowerbds_sizes[48] = {300000, 200000, 900000, 26, 19, 87, 20, 16, 69,
16, 14, 57, 5, 4, 21, 3, 3, 15, 6, 6, 24, 9, 9, 33, 0, 0, 3, 3, 3, 12, 6, 6, 21,
9, 9, 30, 0, 0, 0, 3, 3, 10, 6, 6, 20, 300000, 200000, 900000};
static int order_vtr[2] = {0, 1};
static int E[9] = {1, 0, 0, -1, 1, 0, -3, -1, 1};
static int B_j_2_i[9] = {4, -4, 0, -8, 12, 0, -32, -8, 12};
static int array[2] = {12, 12};
static int i0=0, i1=0, i2=0;
static int j0=0, j1=0, j2=0;
static int pj0=0, pj1=0, pj2=0;
static int vol_elg_spnd=1536, spnd_subpoly_size=356928, time_used=123, min_t=0,
max_t=122, N_DC=4, N_links=4, n_cubes=4, lgst =12, n_para_data_flows=16,
n_dep_array=0, bg_spnd=33084, n_note_data_flows=600003, edge_extra=33084,
n_total_nodes=80000, n_prcsrs=16;
static data_type *elg_spnds[123], *data_flow[6];
static int la100=1,la200=3,la201=1;
static int ua100=1,ua200=3,ua201=1;
static int cl000,cl100,cl101,cl200,cl201,cl202;
static int cu000,cu100,cu101,cu200,cu201,cu202;
static int ll_2[3];
static int uu_2[3];
static int *q_ad0,*q_ad1,*q_ad2,*q_ad3;


#define CopyFromBufferToSupernode(a,b,zise_a) {\
tmp_ptr2 = b;\
for(i0=0;i0<*(zise_a+0);i0++)\
{\
    for(i1=0;i1<*(zise_a+1);i1++)\
    {\
```

```
        for(i2=0;i2<*(zise_a+2);i2++)\
        {\
            *tmp_ptr2++ = *a++;\
        }\
        tmp_ptr2 += *(zise_a+4);\
    }\
    tmp_ptr2 += *(zise_a+3);\
}\
}\

#define CopyFromSupernodeToBuffer(a,b,zise_b) {\
tmp_ptr1 = a;\
for(i0=0;i0<*(zise_b+0);i0++)\
{\
    for(i1=0;i1<*(zise_b+1);i1++)\
    {\
        for(i2=0;i2<*(zise_b+2);i2++)\
        {\
            *b++ = *tmp_ptr1++;\
        }\
        tmp_ptr1 += *(zise_b+4);\
    }\
    tmp_ptr1 += *(zise_b+3);\
}\
}\

#define CopyFromSupernodeToBuffer0(a,b,zise_b) {\
tmp_ptr1 = a;\
for(i0=0;i0<*(zise_b+0);i0++)\
{\
    for(i1=0;i1<*(zise_b+1);i1++)\
    {\
        for(i2=0;i2<*(zise_b+2);i2++)\
        {\
            *b++ = *tmp_ptr1;\
        }\
    }\
}\
}\

#define InitializeData(bounds) {q_ad0 = elg_spnds[t] + bg_spnd;\
for(q[0]=max_fn(cl000,0),cl101=cl100+q[0]*la100,cl201=cl200+q[0]*la200,
cu101=cu100+q[0]*ua100,cu201=cu200+q[0]*ua200,q_ad1=q_ad0+q[0]*jump_extra_vol[0];
q[0]<=min_fn(cu000,*(bounds+0));q[0]++,cl101+=la100,cl201+=la200,cu101+=ua100,
cu201+=ua200,q_ad1+=jump_extra_vol[0])\
```

```
    for(q[1]=max_fn(cl101,0),cl202=cl201+q[1]*la201,cu202=cu201+q[1]*ua201,
    q_ad2=q_ad1+q[1]*jump_extra_vol[1];q[1]<=min_fn(cu101,*(bounds+1));q[1]++,
    cl202+=la201,cu202+=ua201,q_ad2+=jump_extra_vol[1])\
        for(q[2]=max_fn(cl202,0),q_ad3=q_ad2+q[2];q[2]<=min_fn(cu202,*(bounds+2));
        q[2]++,q_ad3++)\
        {\
            ct++;\
            matrix_int_multiplication(E,q,i_index,dim,dim,1,1,0);\
            add_2_int_vtrs(i_index,i_from_j,i_index,N,1);\
            matrix_int_multiplication(jump_step,i_index,q_ad3,1,dim,1,1,0);\
        }\
}\

#define ComputeSupernode(bounds) {q_ad0 = elg_spnds[t] + bg_spnd;\
for(i0=max_fn(cl000,0),cl101=cl100+i0*la100,cl201=cl200+i0*la200,cu101=cu100+i0*ua1
cu201=cu200+i0*ua200,q_ad1=q_ad0+i0*jump_extra_vol[0];i0<=min_fn(cu000,*(bounds+0))
i0++,cl101+=la100,cl201+=la200,cu101+=ua100,cu201+=ua200,q_ad1+=jump_extra_vol[0])\
    for(i1=max_fn(cl101,0),cl202=cl201+i1*la201,cu202=cu201+i1*ua201,q_ad2=q_ad1
    +i1*jump_extra_vol[1];i1<=min_fn(cu101,*(bounds+1));i1++,cl202+=la201,cu202+=ua
    q_ad2+=jump_extra_vol[1])\
        for(i2=max_fn(cl202,0),q_ad3=q_ad2+i2;i2<=min_fn(cu202,*(bounds+2));i2++,
        q_ad3++)\
        {\
            *q_ad3 = *q_ad3+*(q_ad3+d_q[0])+*(q_ad3+d_q[1])+*(q_ad3+d_q[2])+*(q_ad3
            +d_q[3]);\
        }\
}\

#define LL_2_CL cl000=ll_2[0],cu000=uu_2[0],cl100=ll_2[1],cu100=uu_2[1],cl200=ll_2[
cu200=uu_2[2]
```

G.2.2 Parallel Codes

```
#include <stdio.h>
#include <math.h>
#include <csn/csn.h>
#include <csn/csnuio.h>
#include <csn/names.h>
#include <cs.h>
#include "../parameters.h"
#include "../pn.h"
#include "macro_def.c"
```

```
#include "prcsr.h"

main (argc, argv)
int argc;
char **argv;
{
    Transport transport[D_array*(bi_link_orig+1)];
    netid_t   net_ID[D_array*(bi_link_orig+1)];
    int n_netid_t=0,in,*dep_array,m,a_i[D_array],a_i0[D_array],a_i_to[10*D_array];
    int sequenceCount,*sequence,id_pos=2,array[D_array],rx_status=0,tx_status=0;
    int i,ii,j,j_vtr[N+1],j_vtr_1[N],j_vtr0[N],jj,r,l,data_flow_bank,
    n_data_flow_vec=4*N_links*n_data_vtes,N2=2*N,tmp_i,uni_link=1-bi_link_orig,
    bi_link=bi_link_orig,dim=N,k=N-D_array;
    int t,s[D_array],s0[D_array],ptr_elg_spnd=0,ptr_data_flow=0;
    int link_out_valid[D_array*(bi_link_orig+1)];
    int link_in_valid[D_array*(bi_link_orig+1)];
    int l_0[N+1],u_0[N+1],ll_1[N],uu_1[N],x,y,l_innerest_bd,u_innerest_bd,time_end;
    int q[N],i_index[N],i_from_j[N],*i_from_j_inc,jump_step[N],tmp_vtr[N],
    j_vtr_init[N],ct=0,ct0,ctt=0,*ptr1,*ptr2,prct;
    int l_v_t[N*D_array],u_v_t[N*D_array];
    data_type *tmp_ptr1,*tmp_ptr2,*tmp_ptr3,*elg_spnds_base,*idle_spnd,
    *data_flow_out[2*D_array+2],*data_flow_in[2*D_array+2];
    char name[20],here_name[20],there_name[20];
    struct iovec *data_flow_vec_out,*data_flow_vec_in,*tmp_vec;
    int n_spnd_in_prc=*ptn_array,*ptn=ptn_array,ptn_out_inc,ptn_ptr,*size_buffer_ou
    *size_buffer_in,*buffer,*delay_minus,*port,active_d_p;
    MCC_SBC_PRCSR_DATA_FLOW *ptns_out,*ptns_in,*tmp_S_out,*tmp_S_in;

    if (!cs_import ("int vectorSize", &sequenceCount))
        debugf("cannot import 'vectorSize'\n", 1);
    sequence = (int *) malloc(sequenceCount*sizeof(int));
    if (!cs_import ("int *vector", &sequence))
    debugf("cannot import 'vector'\n", 1);
    cp_vtrs(sequence,array,D_array);
    cp_vtrs(sequence+D_array,a_i,D_array);

    dep_array = sequence+2*D_array;
    m = (sequenceCount-2*D_array)/D_array;

    cp_vtrs(a_i,a_i0,D_array);
    re_order_int_rows(a_i,a_i,order_vtr,D_array,1);
    csn_init();
    strcpy(name,ints_2_string(a_i,D_array));
    strcpy(here_name,"transpot");
    strcat(here_name,name);
```

```
    for(i=0;i<N_links;i++)
    {
        strcpy(there_name,here_name);
        strcat(there_name,int_2_alph(i));
        if( csn_open( CSN_NULL_ID, &transport[i] ) != CSN_OK )
            debugf( "master: cannot open transport\n", 1 );
        if( csn_registername( transport[i],there_name ) != CSN_OK )
            debugf( "cannot register Transport \n",1 );
    }

    r = 0;
    for(i=0;i<D_array;i++)
        for(j=1;j>=-bi_link_orig;j -= 2,r++)
        {
            copy_int_array(a_i,a_i_to,1,D_array,1,D_array,0,0);
            a_i_to[i] += j;
            if(0>a_i_to[i]||a_i_to[i]>=array[i])
                link_out_valid[r] = 0;
            else
            {
                strcpy(there_name,"transpot");
                strcat(there_name,ints_2_string(a_i_to,D_array));
                strcat(there_name,int_2_alph(r));
                if(csn_lookupname(&net_ID[r],there_name,1)!=CSN_OK)
                    debugf("cannot lookup %s to %s\n",a_i,there_name);
                link_out_valid[r] = 1;
            }
            a_i_to[i] -= 2*j;
            if(0>a_i_to[i]||a_i_to[i]>=array[i])
                link_in_valid[r] = 0;
            else
                link_in_valid[r] = 1;
        }

    cp_vtrs(a_i0,a_i,D_array);
    re_order_int_rows(a_i,a_i_to,order_vtr,D_array,1);

    ptn = ptn_array;
    ptns_out= (MCC_SBC_PRCSR_DATA_FLOW *) array_2_ptns(ptn,ptn_array,
    D_array,&ptn_out_inc,0);
    ptn = ptn_array + ptn_out_inc;
    ptns_in = (MCC_SBC_PRCSR_DATA_FLOW *) array_2_ptns(ptn,ptn_array,D_array,
    &ptn_out_inc,1);

/* ptns_out and ptns_in are functions of producing OP's and IP's from the data
```

```
of ptn_array. */

    inner_prct(ptn_ptr,a_i_to,ptn_array + ptn_out_inc,D_array);
    ptn_ptr = round_off(ptn_ptr,n_spnd_in_prc);
    ptn += D_array;
    size_buffer_out = ptn_array + ptn_out_inc + D_array;
    size_buffer_in = size_buffer_out + 2*D_array +2;

    data_flow_vec_out = (struct iovec *) malloc(2*N_links*N2*sizeof(struct iovec));
    data_flow_vec_in = (struct iovec *) malloc(2*N_links*N2*sizeof(struct iovec));
    for(i=0;i<N_links+2;i++)
    {
        data_flow_out[i] = (data_type*)malloc(*(size_buffer_out+i)*sizeof(data_type
        data_flow_in[i] = (data_type *) malloc(*(size_buffer_in+i)*sizeof(data_type
    }

    idle_spnd = (data_type *) malloc(vol_elg_spnd*sizeof(data_type));
    elg_spnds_base = (data_type *) malloc(spnd_subpoly_size*sizeof(data_type));

    l_0[N] = 1;
    u_0[N] = 1;
    j_vtr[N] = 1;

    for(i=0;i<spnd_subpoly_size;i++)
        *(elg_spnds_base+i) = 0;
    for(i=0;i<vol_elg_spnd;i++)
        *(idle_spnd+i) = 0;

    ptr2=(lowerbds_sizes+dim*array_2_ad(a_i,array,D_array));

    cp_vtrs(ptr2,j_vtr_init,dim);

    for(i=0;i<N;i++)
    {
        l_0[i] = 0;
        u_0[i] = 0;
    }

    matrix_int_multiplication(l_v_c,l_0,l_0,N,N+1,1,1,0);
    matrix_int_multiplication(u_v_c,u_0,u_0,N,N+1,1,1,0);
    add_2_vtrs(l_inc2,l_0,l_0,N,min_t);
    add_2_vtrs(u_inc2,u_0,u_0,N,min_t);
    cp_vtrs(l_0,ll_1,N);
    cp_vtrs(u_0,uu_1,N);
    transpose_int(l_v_c,l_v_c,N,N+1);
```

```
transpose_int(u_v_c,u_v_c,N,N+1);

for(i=1,jump_step[N-1]=1;i<N;i++)
    jump_step[N-1-i] = 1*jump_step[N-i];

cp_vtrs(lower_array_bds,s,D_array);
scalar_vtr_prct(*H,a_i_to,s,s,D_array);
add_2_vtrs(h0,s,s,D_array,-min_t);
cp_vtrs(s,s0,D_array);
j_vtr[D_array] = min_t;
for(i=0;i<D_array;i++)
{
    inner_prct(prct,H+i*(D_array),j_vtr,i);
    j_vtr[i] = ceil(1.*(s[i]-prct)/(*(H+i*(D_array)+i)));
}
cp_vtrs(j_vtr,j_vtr0,N);
matrix_int_multiplication(j_vtr,l_v_c,ll_2,1,N+1,N,1,0);
matrix_int_multiplication(j_vtr,u_v_c,uu_2,1,N+1,N,1,0);
cp_vtrs(j_vtr,j_vtr_1,N);
cp_vtrs(ll_2,l_0,N);
cp_vtrs(uu_2,u_0,N);

for(j_vtr[D_array]=min_t,t=0;j_vtr[D_array]<=max_t;j_vtr[D_array]++,t++)
{
    add_2_vtrs(j_vtr_init,j_vtr,tmp_vtr,N,-1);
    inner_prct(tmp_i,tmp_vtr,jump_j_2_extra_vol,N);
    elg_spnds[t] = (data_type*) elg_spnds_base + tmp_i;

    matrix_int_multiplication(B_j_2_i,j_vtr,i_from_j,N,N,1,1,0);
    add_2_vtrs(j_vtr,j_vtr_1,j_vtr_1,N,-1);
    for(i=0;i<N;i++)
        if(tmp_i=*(j_vtr_1+i))
        {
            add_2_vtrs(l_v_c+i*N,ll_2,ll_2,N,-tmp_i);
            add_2_vtrs(u_v_c+i*N,uu_2,uu_2,N,-tmp_i);
        }
    cp_vtrs(j_vtr,j_vtr_1,N);

    LL_2_CL;
    ct0 = ct;

    InitializeData(cont_up_bds);

    if(ct0==ct)
            elg_spnds[t] = idle_spnd;
```

```
    add_2_vtrs(h0,s,s,D_array,-1);
    for(i=0;i<D_array;i++)
    {
        inner_prct(prct,H+i*(D_array),j_vtr,i);
        j_vtr[i] = ceil(1.*(s[i]-prct)/(*(H+i*(D_array)+i)));
    }
}

cp_vtrs(s0,s,D_array);
cp_vtrs(j_vtr0,j_vtr,N);
cp_vtrs(j_vtr,j_vtr_1,N);
cp_vtrs(l_0,ll_2,N);
cp_vtrs(u_0,uu_2,N);

for(j_vtr[D_array]=min_t,t=0;j_vtr[D_array]<=max_t;j_vtr[D_array]++,t++)
{
    add_2_vtrs(j_vtr,j_vtr_1,j_vtr_1,N,-1);

    for(i=0;i<N;i++)
        if(tmp_i=*(j_vtr_1+i))
        {
            add_2_vtrs(l_v_c+i*N,ll_2,ll_2,N,-tmp_i);
            add_2_vtrs(u_v_c+i*N,uu_2,uu_2,N,-tmp_i);
        }
    cp_vtrs(j_vtr,j_vtr_1,N);

    LL_2_CL;

    ComputeSupernode(cont_up_bds);

    add_2_vtrs(h0,s,s,D_array,-1);
    for(i=0;i<D_array;i++)
    {
        inner_prct(prct,H+i*(D_array),j_vtr,i);
        j_vtr[i] = ceil(1.*(s[i]-prct)/(*(H+i*(D_array)+i)));
    }

    tmp_S_out = ptns_out + N_links*ptn_ptr;
    tmp_S_in = ptns_in + N_links*ptn_ptr;
    delay_minus = tmp_S_in->nth_d_p + tmp_S_in->n_nth_d_p;
    port = tmp_S_in->nth_d_p + 2*tmp_S_in->n_nth_d_p;

    tmp_ptr2 = data_flow_out[0];
    for(ii=0;ii<tmp_S_out->n_nth_d_p;ii++)
```

```
{
    active_d_p = *(tmp_S_out->nth_d_p+ii);
    for(i=cubes_out_group[2*active_d_p],r=i*N2;i<cubes_out_group[2*active_d
    +1];i++,r+=N2)
        if(elg_spnds[t]==idle_spnd)
        {
            copy_2_vtrs(elg_spnds[t],tmp_ptr2,n_notes[i]);
            tmp_ptr2 += n_notes[i];
        }
        else
        {
            tmp_ptr3 = elg_spnds[t]+bgs_out[i];
            CopyFromSupernodeToBuffer(tmp_ptr3,tmp_ptr2,size_cubes+r);
        }
}

for(i=0;i<N_links;i++)
{
    if(link_in_valid[i]&&(tmp_S_in+i)->n_d_p)
    {
        for(j=0,buffer=(tmp_S_in+i)->buffer;j<(tmp_S_in+i)->n_d_p;j++,
        buffer+=3)
        {
            (data_flow_vec_in+i*N2+j)->iov_base = (caddr_t)
            (data_flow_in[*buffer] + *(buffer+1));
            (data_flow_vec_in+i*N2+j)->iov_len =*(buffer+2)*sizeof(data_typ
        }
        csn_rxnbv(transport[i],data_flow_vec_in+i*N2,(tmp_S_in+i)->n_d_p);
    }

    if(link_out_valid[i]&&(tmp_S_out+i)->n_d_p)
    {
        for(j=0,buffer=(tmp_S_out+i)->buffer;j<(tmp_S_out+i)->n_d_p;
        j++,buffer+=3)
        {
            if(*buffer)
                (data_flow_vec_out+i*N2+j)->iov_base = (caddr_t)
                (data_flow_in[*buffer] + *(buffer+1));
            else
                (data_flow_vec_out+i*N2+j)->iov_base = (caddr_t)
                (data_flow_out[*buffer] + *(buffer+1));
            (data_flow_vec_out+i*N2+j)->iov_len = *(buffer+2)
            *sizeof(data_type);
        }
        csn_txnbv(transport[i],0,net_ID[i],data_flow_vec_out+i*N2,
```

```
            (tmp_S_out+i)->n_d_p);
        }
    }

    for(i=0;i<N_links;i++)
        if(link_in_valid[i]&&(tmp_S_in+i)->n_d_p)
            csn_test(transport[i],CSN_RXREADY,-1,NULL,NULL,&rx_status);

    for(i=0;i<N_links;i++)
        if(link_out_valid[i]&&(tmp_S_out+i)->n_d_p)
            csn_test(transport[i],CSN_TXREADY,-1,NULL,NULL,&tx_status);

    tmp_ptr1 = data_flow_in[N_links+1];
    for(ii=0;ii<tmp_S_in->n_nth_d_p;ii++)
    {
        active_d_p = *(tmp_S_in->nth_d_p+ii);
        for(i=cubes_out_group[2*active_d_p],r=i*N2;i<cubes_out_group[2*active_d
        +1];i++,r+=N2)
        {
            l=t+delay_data[i]-*(delay_minus+ii)+1;
            if(link_in_valid[*(port+ii)]&&l<=max_t&&elg_spnds[l]!=idle_spnd)
            {
                CopyFromBufferToSupernode(tmp_ptr1,elg_spnds[l]
                +bgs_to_extra_vol[i],size_cubes+r);
            }
            else
                tmp_ptr1 += n_notes[i];
        }
    }

    ptn_ptr++;
    if(ptn_ptr==n_spnd_in_prc)
        ptn_ptr = 0;
}
}
```